信息与计算专业教材

并行计算导论

张林波　中国科学院数学与系统科学研究院
迟学斌　中国科学院计算机网络信息中心
莫则尧　北京应用物理与计算数学研究所
李　若　北京大学数学学院

2006 年 7 月 17 日

图书在版编目 (CIP) 数据

并行计算导论/张林波等编著. — 北京：清华大学出版社, 2006.6
(高等院校信息与计算科学专业系列教材)
ISBN 7–302–12760–3

I. 并... II. 张... III. 并行算法 – 高等学校 – 教材 IV. TP301.6
(8132?)

中国版本图书馆 CIP 数据核字 (2006) 第 026693 号

内 容 简 介

本书是并行计算，特别是分布式并行计算环境和消息传递并行编程的入门教材，目的是促进并行计算技术在我国的应用。书中介绍了并行计算的基础知识、Linux/UNIX 的基本使用、基于 Linux 机群的并行计算平台的建立、并行算法的设计和 MPI 消息传递并行编程的基本概念与方法。书中还提供了一批典型科学计算问题的并行算法与程序设计实例，介绍了一些当前国际上流行的科学计算软件工具及平台。本书力求从简单入手，循序渐进，读者不需要有太多的预备知识，在逐步学习使用的过程中学会利用并行计算解决自己学习和工作中的问题。

本书侧重介绍高性能计算的实用技术，可作为信息技术与计算专业的专业基础课教材，也可作为其他理工科非计算机专业的并行计算课程教材。此外，本书还可作为有关 Linux 机群建造、MPI 消息传递并行算法设计与编程及常用高性能科学计算软件的参考书使用。

前　言

随着高性能并行计算机，尤其是 Linux 微机机群在我国应用部门、大学和科研机构的普及，并行计算已经成为许多科研和工程技术人员亟待掌握的一项研究开发手段。但是，在我国，对并行计算的基本原理、并行算法设计、并行程序的设计与实现、并行性能优化以及一些成熟的科学计算软件工具箱、库等的普及教育和推广应用还明显不够，大大制约了并行计算技术在科学研究与工程设计中应有作用的发挥。本书是并行计算，特别是分布式并行计算环境及消息传递并行编程的入门教材。它侧重于介绍利用现有的微机条件建立并行计算的软硬件环境，以及并行算法设计、MPI 消息传递并行编程的基本概念与方法。作者希望通过本书，一方面加强和规范普通高校的并行计算课程，另一方面在科学研究与工程应用领域普及并行计算技术，推进高性能计算技术的应用。

全书分为三部分，共包括九章和两个附录。

第 1 部分包括第 1 章至第 4 章，主要介绍并行计算的基础知识。第 1 章讲述并行计算机的发展历史、并行计算机体系结构以及并行计算、并行算法设计的基本名词和概念。第 2 章介绍 Linux 操作系统的安装、Linux 操作系统的基本命令和工具，以及 Linux 操作系统中的程序开发、调试。第 3 章讲述标准消息传递并行编程接口 MPI，并介绍如何利用目前流行的免费 MPI 软件 MPICH 和 Linux 系统在局域网上建立实用的并行计算平台。考虑到本书的篇幅限制，并鉴于国内已有一些专门关于 MPI 编程的书，本书中关于 MPI 编程的部分仅限于讲述 MPI 编程的基本概念和介绍一些 MPI 的重要函数，同时在附录 B 中提供 MPI 变量和函数的完整参考方便读者查询。第 4 章讲述程序性能评价与优化方面的有关知识。

第 2 部分包括第 5 章至第 9 章。在这部分中，通过一些典型并

行算法设计及并行程序实现的实例，介绍并行算法设计的基本思想与 MPI 并行程序实现的基本技巧。第 5 章介绍一个自适应数值积分算法的并行算法设计与 MPI 并行程序实现。第 6 章介绍矩阵计算的并行算法与程序。第 7 章介绍快速傅里叶变换 (FFT) 的并行算法及在消息传递并行环境中的实现方法。第 8 章以二维 Poisson 方程 5 点差分格式的点 Jacobi 迭代算法为例，介绍基于区域分解方法的并行算法设计与并行程序编制。第 9 章通过二维热传导方程的 ADI 格式介绍基于流水线方法的并行算法设计的基本思想及实现方法。

第 3 部分由两个附录构成。附录 A 介绍一些重要的高性能计算、并行计算的工具和平台，包括 BLAS，LAPACK，ScaLAPACK，FFTW 和 PETSc。附录 B 是一个为方便读者速查而整理的较完整的 MPI 参考手册。

书的最后是 MPI 函数、变量索引和名词、概念索引。大部分章后面附有习题，一部分习题是为了巩固、加深对该章内容的理解，而另一部分习题则是对正文内容的扩充。

本书可作为一本课堂使用或者自学使用的教材，在编写过程中考虑了适应尽可能广的读者群体，以便读者不需要很多的预备知识，就能够在逐步的使用过程中掌握利用并行计算技术解决自己学习和工作领域中的计算问题的知识和技术。本书的特点是：注重实效性，使得读者能够学到具体有用的知识和技术，而不必首先建立对整个知识系统结构的认识；强调实践性，读者应该一边学习一边操作，认真分析和重复书中提供的具体例子，将介绍的知识应用于各自的专业领域中；强调普适性，侧重于科学计算领域面临的一些共性问题的并行计算方法与技术，并注意介绍一些国际流行的库、软件和工具箱。希望读者能够结合自己的实际情况，练习简单微机机群环境建立的实际操作及 Linux 系统的管理和使用，与自己的工作领域相结合编写具体模型问题的例子程序，举一反三，学以致用，通过本书的学习，迅速掌握并行计算机的使用与编程，并且能够利用已有条

件采用微机机群技术自行建立实验性或实用的并行计算环境，用于解决实际问题。

本书的目的是介绍运用高性能并行计算机深入解决科学计算问题所必需掌握的并行计算原理、并行算法设计、并行程序设计和性能优化等方面的基本知识和技术手段。鉴于此，本书写作过程中尽量避免枯燥的系统介绍，而是侧重于培养读者解决实际问题的能力，并注意介绍编著者在这方面的经验。限于作者的学识以及时间，书中错误与片面之处在所难免，恳请读者不吝指正。

在本书的作者中，张林波 (中国科学院数学与系统科学研究院) 编写了第 3 章、第 5 章、第 9 章和附录 B，以及第 2 章、第 4 章和附录 A 中的部分内容；迟学斌 (中国科学院计算机网络信息中心) 编写了第 6 章、第 7 章，以及附录 A中的部分内容；莫则尧 (北京应用物理与计算数学研究所) 编写了第 1 章、第 4 章和第 8 章；李若 (北京大学数学科学学院) 编写了第 2 章。

本书中的程序实例可以从下面的网址下载：

`ftp://ftp.cc.ac.cn/pub/home/zlb/bxjsbook/code/`

鸣谢：下述人员提供了附录 A 中的部分内容：程强 (A.5)、陈江 (A.3)、赵韬 (A.2)、谷同祥 (A.4)，在此一并致谢。

作者

2006 年 7 月 17 日于北京

目　　录

表格目录

插图目录

第 1 部分

基 础 知 识

第 1 章　预备知识

本章主要介绍学习并行计算所需要了解的一些基础知识。首先，给出并行计算的一个简单定义，并讨论并行计算的主要研究目标和内容，以及推动并行计算发展的主要动力。其次，简要介绍并行计算赖以存在的硬件平台－并行计算机的发展历史。然后，简要介绍当前并行机的体系结构，以及运行在这些并行机上的操作系统和并行编程环境，使读者对并行计算机系统有一个大致的了解，为深入学习并行编程奠定基础。最后，简要讨论并行算法。

1.1　并行计算的主要研究目标和内容

并行计算是伴随并行机的出现，在近 30 年来迅速发展的一门交叉学科，涵盖的内容非常广泛。参考文献 [1] 较为全面地综述了并行计算在各个方面的最新进展，内容包括并行机体系结构、编译系统、并行算法、并行编程、并行软件技术、并行性能优化与评价、并行应用等。因此，很难全面并精确地给出并行计算的定义。但是，从交叉学科的角度，并行计算可以定位为连接并行机系统和实际应用问题之间的桥梁。它辅助科学、工程及商业应用的领域专家，为在并行机上求解领域问题提供具有共性的关键支撑。本节基于这点认识，首先给出并行计算一个粗浅的定义，然后讨论并行计算的主要研究目标和内容，以及推动并行计算发展的主要动力。

1.1.1　什么是并行计算

并行计算（parallel computing）是指，在并行机上，将一个应用分解成多个子任务，分配给不同的处理器，各个处理器之间相互协同，并行地执行子任务，从而达到加速求解速度，或者求解应用问题

规模的目的。

由此，为了成功开展并行计算，必须具备三个基本条件：

(1) 并行机。并行机至少包含两台或两台以上处理机，这些处理机通过互连网络相互连接，相互通信。

(2) 应用问题必须具有并行度。也就是说，应用可以分解为多个子任务，这些子任务可以并行地执行。将一个应用分解为多个子任务的过程，称为并行算法的设计。

(3) 并行编程。在并行机提供的并行编程环境上，具体实现并行算法，编制并行程序，并运行该程序，从而达到并行求解应用问题的目的。

1.1.2 并行计算的主要研究目标和内容

对于具体的应用问题，采用并行计算技术的主要目的在于两个方面：

(1) 加速求解问题的速度。例如，给定某应用，在单处理器上，串行执行需要 2 个星期（14 天），这个速度对一般的应用而言，是无法忍受的。于是，可以借助并行计算，使用 100 台处理器，加速 50 倍，将执行时间缩短为 6.72 个小时。

(2) 提高求解问题的规模。例如，在单处理器上，受内存资源 2GB 的限制，只能计算 10 万个网格，但是，当前数值模拟要求计算千万个网格。于是，也可以借助并行计算，使用 100 个处理器，将问题求解规模线性地扩大 100 倍。

并行计算之所以必需，主要在于，当前的单处理器性能不可能满足大规模科学与工程计算及商业应用的需求，并行计算是目前唯一能满足实际大规模计算需求的支撑技术。例如，即使是当前较为

先进的微处理器（Itanium–II 1.5GHz、POWER4 1.5GHz 等），其峰值性能也仅为 60 亿次/秒。近 2 年内，微处理器的峰值性能也不会超过 100 亿次/秒。并行计算之所以可行，主要在于，并发性是物质世界的一种普遍属性，具有实际应用背景的计算问题在许多情况下都可以分解为能并行计算的多个子任务。

综上所述，并行计算的主要目标在于，在并行机上，解决一批具有重大挑战性计算任务的科学、工程及商业计算问题，满足不断增长的应用问题对速度和内存资源的需求。

由并行计算的三个必备条件可知，并行计算的主要研究内容大致可分为四个方面：

(1) 并行机的高性能特征抽取。主要任务在于，充分理解和抽取当前并行机体系结构的高性能特征，提出实用的并行计算模型和并行性能评价方法，指导并行算法的设计和并行程序的实现。

(2) 并行算法设计与分析。针对应用领域专家求解各类应用问题的离散计算方法，设计高效率的并行算法，将应用问题分解为可并行计算的多个子任务，并具体分析这些算法的可行性和效果。

(3) 并行实现技术，主要包含并行程序设计和并行性能优化。基于并行机提供的并行编程环境，例如消息传递平台 MPI[20] 或者共享存储平台 OpenMP[8]，具体实现并行算法，研制求解应用问题的并行程序。同时，结合并行机的高性能特征和实际应用的特点，不断优化并行程序的性能。

(4) 并行应用。这是并行计算研究的最终目的。通过验证和确认并行程序的正确性和效率，进一步将程序发展为并行应用软件，应用于求解实际问题。同时，结合实际应用出现的各种问题，不断地改进并行算法和并行程序。一个没有经过实际应用验证和确认的算法和程序，不能说是一个好的算法和程序。

以上四个部分相互耦合，缺一不可。第 1 部分是开展并行计算研究的基础，第 2、3 部分是并行计算研究的核心，也是在并行机上高效求解应用问题的基本保证，第 4 部分是并行计算研究的目的，也是验证和确认并行计算研究成果的最有效的途径。第 4 部分将为第 2、3 部分的研究提供取之不尽的源泉。

需要说明的是，并行计算不同于分布式计算（distributed computing）。后者主要是指，通过网络相互连接的两个以上的处理机相互协调，各自执行相互依赖的不同应用，从而达到协调资源访问，提高资源使用效率的目的。但是，它无法达到并行计算所倡导的提高求解同一个应用的速度，或者提高求解同一个应用的问题规模的目的。对于一些复杂应用系统，分布式计算和并行计算通常相互配合，既要通过分布式计算协调不同应用之间的关系，又要通过并行计算提高求解单个应用的能力。

下面给出并行计算和分布式计算的几个例子，以示区别。

例 1.1:（并行计算）N 个数被分布存储在 P 台处理器，P 台处理器并行执行 N 个数的累加和。首先，各个处理器累加它们各自拥有的局部数据，得到部分和；然后，P 台处理器执行全局通信操作，累加所有部分和，得到全局累加和。

例 1.2:（并行计算）给定二维规则区域上的 Dirichlet 问题 $-\Delta u = f$，采用标准 5 点有限差分格式离散。平均分配 $N = N_x \times N_y$ 个网格单元给 $P = P_x \times P_y$ 台处理机。所有处理机并行计算，执行 Jacobi 迭代，求解 Dirichlet 问题，从而达到缩短求解问题的时间，或者扩大网格规模 N 的目的。

例 1.3:（分布式计算）观众点播，远程驾驭式可视化，电视会议等。

1.1.3 推动并行计算发展的主要动力

大规模科学与工程计算应用对并行计算的需求是推动并行计算

快速发展的主要动力。长期以来，它们对并行计算的需求是无止境的。例如，全球气象预报中期天气预报模式要求在 24 小时内完成 48 小时天气预测数值模拟，此时，至少需要计算 635 万个网格点，内存需求大于 1TB，计算性能要求高达 25 万亿次/秒。又如，美国在 1996 年开始实施 ASCI计划，要求分四个阶段，逐步实现万亿次、十万亿次、30 万亿次和 100 万亿次的大规模并行数值模拟，实现全三维、全物理过程、高分辨率的核武器数值模拟。除此之外，在天体物理、流体力学、密码破译、海洋大气环境、石油勘探、地震数据处理、生物信息处理、新药研制、湍流直接数值模拟、燃料燃烧、工业制造、图像处理等领域，以及大量的基础理论研究领域，存在计算挑战性问题，均需要并行计算的技术支持。

近 10 年来，美国在大规模科学与工程计算应用领域，启动了三次重大计划，极大地推动了并行计算的发展。通过这三次计划，美国无论在并行计算机的研制方面，还是在并行应用程序的开发方面，均取得了国际领先的绝对优势，极大地推动了并行计算在科学与工程计算各个领域的应用，全面加速了科学技术的发展。

第一次是在 1983 年，配合星球大战战略防御计划而开展的战略计算机计划（SCP），研制了每秒十亿次的 CRAY 并行机。

第二次是在 1993 年，由美国科学、工程、技术联邦协调委员会向国会倡议的“重大挑战性项目：高性能计算与通信（HPCC）计划”，目的是研制能够提供 3T 性能目标（1Tflops 计算能力、1TB 内存容量和 1TB/s 的 I/O 带宽，1T=10^{12}）的高性能并行机，解决科学与工程计算中的重大挑战性课题，保持其在高性能计算和计算机通信领域中的世界领先地位，利用高性能计算机与网络技术刺激生产，以提高国民经济、国家安全、教育和整体环境的竞争力。该计划由高性能计算机系统（HPCS）、先进算法与软件（ASTA）、国家科研与教育网（NREN）、基本研究与人类资源（BRHR）四部份组成，其目标是：1）对一大批重要应用问题，计算性能要达到每秒万亿次运

算；2）发展相关的系统软件，对一大批问题改进算法；3）国家研究网能力要达到每秒十亿位；4）充分保证计算机科学与工程领域的科研人员的需求。

HPCC 计划提出的背景是一大批巨大挑战性问题需要解决，其中包括：天气与气候预报，分子、原子与核结构，大气污染，燃料与燃烧，生物学中的大分子结构，新型材料特性，国家安全等有关问题。而近期内要解决的问题包含：磁记录技术，新药研制，高速城市交通，催化剂设计，燃料燃烧原理，海洋模型模拟，臭氧层空洞，数字解剖，空气污染，蛋白质结构设计，金星图象分析和密码破译技术。世界上第一台峰值速度超过 1Tflops 的高性能计算机由 Intel 公司于 1996 年 12 月成功研制。

日本相继提出了真实世界计算计划，欧洲提出了万亿次机计划，我国 863 高科技计划也将并行计算机的研制列为关键攻关技术。

第三次在 1996 年，美国能源部联合美国三大核武器实验室（Los Alamos 国家实验室、Lawrence Livermore 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（ASCI）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。为此，三大实验室分别向美国三大公司（Intel、IBM 和 SGI）预定了峰值速度超过 1Tflops 的并行机，计划分四个阶段，分别实现万亿次、10 万亿次、30 万亿次和 100 万亿次的高性能并行机。目前，以 TOP 500[9] 中排名第 1 的峰值性能为 185 万亿次的 IBM Blue Gene/L 为标志，四个阶段的并行机研制已经初步实现。目前，整个 ASCI计划发展到第 2 个阶段，改名为“先进模拟计算计划 (ASC)”。

对应于美国的 ASCI计划，日本和欧洲也提出了相应对策。尤其是日本，以 2002 年研制的 Earth Simulator[9] 及其在大气海洋环流等大规模科学与工程计算中的高效率应用为标志，将并行计算推向了一个新的高度。

除了大规模科学与工程计算应用外，微电子技术与大规模集成电路 VLSI 的发展是推进并行计算发展的另一个主要动力。当前，Moore 定律仍在延续，计算机微处理器的速度每 3 年翻两番，内存容量每两年翻 3–4 倍；互连网络技术飞速发展，专用并行机网络的延迟可低于 1 个微秒，带宽可达 6.4GB/s，商用并行机网络延迟可低于 6 个微秒，带宽可达 1.25GB/s。这些基本构件性能的改进，无疑会改变并行机的体系结构、操作系统和编译系统、并行编程环境，从而对并行算法和并行编程模式带来新的挑战。当前，如何快速地开发并行应用程序，高效率地发挥当前并行机的峰值性能，已经成为当前并行计算研究面临的一个挑战性问题。

1.2 并行计算机发展历史

并行计算机从 70 年代的开始，到 80 年代蓬勃发展和百家争鸣，90 年代体系结构框架趋于统一，近 5 年来机群技术的快速发展，并行机技术日趋成熟。本节以时间为线索，简介并行计算机发展的推动力和各个阶段，以及各类并行机的典型代表和它们的主要特征。

1.2.1 应用需求的推动作用

市场需求一直是推动并行计算机发展的主要动力，大量实际应用部门，例如数值天气预报、核武器、石油勘探、地震数据处理、飞行器数值模拟和大型事务处理、生物信息处理等，都需要每秒执行万亿次、数十万亿次、乃至数百万亿次浮点运算的计算机。正如引言中所指出的，基于这些应用问题本身内部存在的并行性和单机性能的限制，并行计算是满足他们需求的唯一和可行途径。

高性能并行计算机的研制与应用水平，一直是以美国和日本为首的各个发达国家共同追逐的目标，是衡量一个国家科技、经济和国防综合实力的重要标志。在以明确应用需求为背景的 HPCC 计划

和 ASCI 计划的推动下，美国在这方面取得了绝对的领先地位。

1.2.2 70 年代

1972 年，世界上诞生了第一台并行计算机 ILLIAC IV，它含 32 个处理单元，环型拓扑连接，每台处理机拥有局部内存，为 SIMD 类型机器。对大量流体力学程序，ILLIAC IV 获得了 2–6 倍于当时性能最高的 CDC 7600 机器的速度。

70 年代诞生的并行机还有阵列机 ICLDAP、Goodyear MPP，以及向量机 CRAY–1、STAR-100 等，它们都属于 SIMD 类型，其中向量机 CRAY–1 获得了很好的向量计算效果。70 年代的并行计算机引起了人们的极大兴趣，吸引了大量的专家学者从事于并行计算机研制和并行程序的设计，为 80 年代并行计算机的蓬勃发展奠定了坚实的基础。

1.2.3 80 年代早期

80 年代早期，以 MIMD 并行机的研制为主。首先诞生的是 Denelcor HEP，含 16 台处理机，共享存储，能同时支持细粒度和大粒度并行，并且被应用到实际计算中，使许多人学会了并行计算。其次，诞生了共享存储向量多处理机 CRAY X–MP/22（2 个向量机结点）、IBM 3090（6 个向量机结点），取得了很好的实际并行计算性能。同时，以超立方体结构连接的分布式存储 MIMD 结构原型机开始出现。

1.2.4 80 年代中期

80 年代中期，共享存储多处理机系统得到了稳定发展。两个成功的机器为 Sequent （20 个结点）、Encore（16-32 个结点），它们提供稳定的 UNIX 操作系统，实现用户间的分时共享，对当时 VAX 系列串行机构成了严重的威胁。同时，还诞生了 8 个结点的向量多处理机 Alliant，它提供了非常好的自动向量并行编译技术；诞生了 4

个结点的向量处理机 CRAY–2。这些向量多处理机系统在实际应用中均取得了巨大的成功。与此同时，人们对共享存储多处理机系统的内存访问瓶颈问题有了较清楚的认识，纷纷寻求解决办法，以保证它们的可扩展性。

此期间还诞生了可扩展的分布存储 MIMD MPP nCUBE，这台机器含 1024 个结点，CPU 和存储单元均分布包含在结点内，所有结点通过超立方体网络相互连接，支持消息传递并行编程环境，并真正投入实际使用。由于该机对流体力学中的几个实际应用问题获得了超过 1000 的加速比，引起了计算机界的轰动，改变了人们对 Amdahl 定律的认识，排除了当时笼罩并行计算技术的阴影。

当时，在分布式存储体系结构中，处理机间的消息传递与消息长度、处理机间的距离有较大的关系。因此互连网络最优拓扑连接和数据包路由选择算法的研究引起了人们的大量注意，目的在于减少处理机远端访问的花费。

1.2.5 80 年代后期

80 年代后期，真正具有强大计算能力的并行机开始出现。例如，Meiko 系统，由 400 个 T800 Transputer 通过二维 Mesh 相互连接构成，适合于中等粒度的并行；三台 SIMD 并行机：CM–2，MasPar 和 DAP，其中 CM–2 对 Linpack 测试获得了 5.2Gflops 的性能；超立方体连接的分布存储 MIMD 并行机 nCUBE–2 与 Intel iPSC/860，分别可扩展到 8K 个结点和 128 个结点，峰值性能达 27Gflops 和 7Gflops；由硬件支持共享存储机制的 BBN TC2000，用 Butterfly 多级互连网连接处理机和存储模块，可扩展到 500 台处理机，本地 cache、内存和远端内存访问的延迟时间为 $1:3:7$；共享存储向量多处理机系统 CRAY Y–MP，能获得很好的实际运算性能。

1.2.6 90 年代早期

进入 90 年代，得益于微电子技术的发展，基于 RISC 指令系统的微处理芯片的性能几乎以每 18 个月增长 1 倍、内存容量每年几乎增长 1 倍的速度发展。而网络通信技术也得到了快速增长。它们都对并行计算机的发展产生了重要影响。

为了满足 HPCC 计划中提出的高性能计算要求，考虑到共享存储并行机不可避免的内存访问瓶颈问题，人们纷纷把眼光瞄准了分布式存储 MPP 系统，使得 MPP 的硬件和软件系统得到了长足的发展。由于微处理芯片性能和网络技术的发展，MPP 并行机大量采用商用微处理芯片作为单结点，通过高性能互连网连接而成。由于普遍采用虫孔（wormhole）路由选择算法，使得消息传递时间不再与它所经过的结点个数相关，即处理机间消息传递花费不再与距离相关，或者相关程度可以忽略不计。互连网络拓扑结构趋于统一。分布式存储并行程序设计以消息传递为主，少量的也支持数据并行高性能 Fortran（HPF）。

这一时期，MIMD 类型占据绝对主导位置。用于科学与工程计算的 SIMD 类型并行机和单纯的向量机已逐渐退出历史舞台，但以单个向量机为结点构成的 MIMD 并行机仍然在实际应用中发挥重要作用。

这段时间出现的分布式存储 MPP 并行机主要有：

- Intel Touchstone Delta，含 512 个 i860 微处理芯片，二维 Mesh 连接，峰值性能为 32Gflops，8GB 内存。
- CM–5E，含 16–1K 个标量 RISC SPARC 处理器（含四个向量部件，峰值性能 128Mflops，32MB 内存），胖树数据网、二叉控制网及四叉诊断网连接各个处理器。
- Intel Paragon XP/S，含 24–3K 个 i860/XP 微处理芯片（主频

50MHz），二维 Mesh 连接，内存 122GB。

- CRAY T3D，16–1K 个结点，每结点含 2 个处理器（64 位 RISC DECchip 21064，峰值 150Mflops），局部内存 64MB，最大存储规模 128GB，结点间双向三维 Torus 连接。
- SP2，含 16 个机柜，每个机柜含 16 个处理器（POWER–2 芯片，主频 66.5MHz，128-256MB 内存），机柜内部采用高性能开关 HPS 连接，机柜之间采用信息传递光纤网络连接。
- Fujitsu VP500，128 个结点（向量机），单结点内存 256MB，同时支持粗、细粒度并行。
- 其他还有 CM–5，Convex SPP1000，Hitachi，DEC AS–8400 等。

在共享存储方面，由向量机构成的并行机 CRAY Y-MP C90 (16 台向量处理机，峰值性能 16Gflops) 和国产 YH-2 (4 台向量机，峰值性能 1Gflops) 也诞生了，在应用问题中发挥了重要作用。

此外，为了让共享存储并行机具有可扩展性以适用高性能计算需求，并且继承共享存储并行机并行程序设计的容易性，分布共享存储的思想已经被人们接受。这方面的代表机型为 1991 生产的 Kendall Square KSR–1，它提供给用户透明的共享存储结构，每个环含 32 个结点，多个环以层次结构相互连接，可扩展到 1024 个结点，峰值性能为 15Gflops。

1.2.7 90 年代中后期

90 年代中期，微处理器的性能已经非常强大，能提供每秒几亿到十几亿次的浮点运算速度。例如：

- IBM P2SC，主频 135MHz，峰值性能 500Mflops。
- SGI MPIS R10000，主频 195MHz，峰值性能 400Mflops。

- SUN Ultra SPARC，主频 250MHz，峰值性能 1Gflops。
- DEC Alpha 21164，主频 600MHz，峰值性能 1.2Gflops。

同时，互连网络点对点通信能达到每秒超过 500MB 的带宽。高性能微处理器和网络通信技术为并行计算硬件环境带来了新的面貌，使得它们呈现以下几个发展趋势。

第一，以高性能微处理芯片和互连网络通信技术为基础，共享存储对称多处理机（SMP）系统得到了迅速发展。它们大多以高性能服务器的面目出现，能提供每秒几百亿次的浮点运算能力、几十个 GB 的内存和超过 10GB/Sec 的访存带宽，具有丰富的系统软件和应用软件，很强的容错能力、I/O 能力、吞吐量、分时共享能力和稳定性，友好的共享存储并行程序设计方式和使用方便的并行调试、性能分析工具，为大量中小规模科学与工程计算、事务处理、数据库管理部门所欢迎。因此，它们出现以后，迅速抢占了原属于共享存储向量并行机的市场，成为几百亿次以下并行计算机的主导机型。但是，它们仍然具有可扩展性差的弱点，不可能满足超大规模并行计算的要求。

以 SUN Ultra E10000 为例。它采用共享存储对称多处理机结构，可扩展到 64 台 Ultra SPARC 处理器（主频 250MB，2MB cache），峰值性能为 25Gflops，内存容量为 64GB，互连网络（访存）带宽为 12.8GB/Sec，I/O 带宽为 64GB/Sec。它采用标准 UNIX 操作系统，支持共享存储、消息传递并行程序设计。类似的 SMP 系统还有 SGI Power Challenge R10000，HP C–240，DEC Alphaserver 400C 等，这里不一一介绍。

第二，以微处理芯片为核心的工作站能提供近 1Gflops 的计算速度，几十 MB 的内存，能单独承担一定的计算任务。并且，将多台这样的同构或异构型工作站通过高速局域网相互连接起来，再配备一定的并行支撑软件，形成一个松散耦合的并行计算环境，协同

地并行求解同一个问题，称之为工作站机群（NOWs：Network Of Workstations）。它们可以利用本局域网范围内空闲的工作站资源，动态地构造并行虚拟机，能提供几十亿或几百亿次的计算性能。例如，多台通过快速以太网（100Mbps）相互连接的相同或不同类型的工作站，并配备 PVM、MPI 消息传递并行程序设计软件支撑环境，就可以称之为一个工作站机群。

由于 NOWs 具有投资风险小、结构灵活、可扩展性强、软件财富可继承、通用性好、异构能力强等较多优点而被大量中、小型计算用户和科研院校所接受，成为高性能并行计算领域的一个新的发展热点，占据了原属于传统并行计算机的部分市场。但是，它们仍然具有结构不稳定、并行支撑软件少、并行开销大、通信带宽低、负载平衡和并行程序设计难等许多亟待解决的问题，吸引了大量国内外专家学者的注意力。

第三，由于分布式存储的并行计算机具有并行程序设计难、不容易被用户接受的缺点，单纯的分布式存储并行机已经朝分布共享方向发展。它们都采用最先进的微处理芯片作为处理单元，单元内配备有较大的局部 cache 和局部内存，所有局部内存都能实现全局共享，所有结点通过高性能网络相互连接，从而用户可以采用共享存储或数据并行的并行程序设计方式，并且自由地申请结点个数和内存大小。

基于 DSM 的并行机主要有两种结构

1）基于 cache 一致性（coherent）的 NUMA 结构（CC–NUMA）。CC–NUMA 结构具有比 SMP 更好的可扩展性，并且能保持 SMP 的共享存储特性，使得共享存储并行程序设计能获得较好的并行计算性能。但是，它并不能完全避免 SMP 结构中出现的内存访问瓶颈问题，因此不具备分布式存储并行机的可扩展性，为分布式存储与 SMP 的折衷机型。

CC-NUMA 结构的典型代表为 SGI Origin 系列超级服务器，它是基于目录（directory）的 CC-NUMA 结构，如图 1.1 所示。以 Origin –2000 为例，它可扩展到 8 个机柜，每个机柜含 8 个结点，结点是构成 Origin–2000 的基本单位，它包含：

- 1–2 个主频为 195MHz 的 R10000 CPU，每个 CPU 含 4MB 的二级 cache；
- 内存 512MB–4GB，分主存（main memory）和目录内存（directory memory，用于保持结点间的 cache 一致性）；
- 集线器（HUB）含 4 个端口（interface）：CPU 端口、内存端口、XIO 端口、CrayLink 互连网络端口，采用交叉开关实现两个 CPU、内存、输入输出和互连网络路由器（router）之间的全互联（crossbar），分别提供 780MB/s、780MB/s、1.5GB/s、1.5GB/s 的传输速度。

Origin–2000 的所有结点通过 CrayLink 网络相互连接。路由器是构成 CrayLink 的基本单位，含 6 个端口，采用交叉开关实现端口间全互连，可视需要进行端口间连接的任意快速切换，具有 9.3GB/s 的峰值带宽。每个路由器的两个端口用于连接结点，其余 4 个端口实现路由器间的互连，形成互连网络拓扑结构。CrayLink 的半分带宽与结点个数成线性递增关系，并且对任意两个结点，至少能提供两条路径，保证了结点间的高带宽、低延迟连接和互连网络的稳定性和容错能力。

同时，Origin–2000 具有可扩展的、功能强大的 I/O 子系统，这里不再介绍。总体而言，Origin–2000 能提供超过 50Gflops 的峰值性能，512GB 的内存，已逼近千亿次并行计算的要求。

Origin–2000 采用标准 UNIX 操作系统，支持共享存储、数据并行和消息传递三种并行程序设计方式。

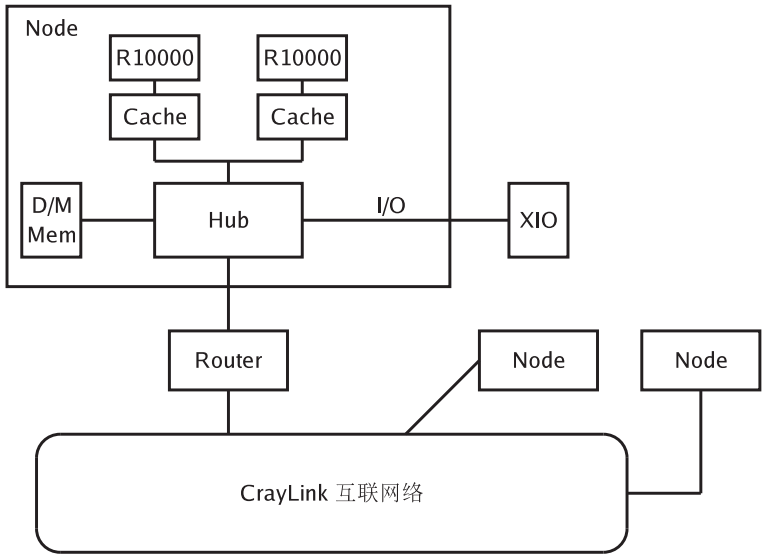


图 1.1 Origin–2000 结构图

2）NUMA 结构。即在原来分布式存储并行机的基础上，增加局部内存的全局共享功能，提供共享存储并行编程环境。它们能继续保持分布式存储并行机的可扩展性，但当处理机台数较多时，也只有消息传递并行程序设计方式才能发挥它们的潜在并行计算性能。这方面的典型代表为 Cray T3E 和 Cray T3E-1200，最多可扩展到 2048 个 CPU，采用了当时最先进的微处理芯片（主频 600MHz），峰值性能达到 2.5Tflops。

1.2.8 2000 年到当前

2000 年以来，受重大挑战计算需求的牵引和微处理器及商用高速互连网络持续发展的影响，高性能并行机得到前所未有的大踏步发展。至 2005 年底，国内陆续安装到位的万亿次并行机将近 20 台

套。从并行机应用的领域分类，这些并行机大致可分为两类。一类是通用型的并行机系统，以微机机群为典型代表，它们具有优良的性能价格比，占据了高性能计算机的大部分市场；另一类为面向某类重大应用问题而定制的 MPP 系统，通常为国家的战略应用而特殊定制。从体系结构的角度，当前并行机的体系结构可分为如下三类 [10]：

(1) 机群 (cluster)。它们有三个明显的特征：

- 系统由商用结点构成，每个结点包含 2-4 个商用微处理器，结点内部共享存储。
- 采用商用机群交换机连接结点，结点间分布存储。
- 在各个结点上，采用机群 Linux 操作系统、GNU 编译系统和作业管理系统。

目前，机群采用的典型商用 64 位微处理器代表为 IBM PPC 970 2.2GHz，Intel Itanium–II 1.5GHz，AMD Opteron 2.2GHz，峰值性能分别可达每秒 88 亿次、60 亿次和 44 亿次。单结点配置内存空间可达数十个 GB。商用机群互连网络以商用交换机 Myrinet 2000 (点对点延迟 9 微秒、带宽 256MB/s)、Quadrics (点对点延迟 5 微秒、带宽 400MB/s)、InfiniBand (点对点延迟 5 微秒、带宽 1.25GB/s) 等为典型代表。通常，单台交换机可扩展至 128 个端口，多台交换机堆叠可连接数百上千个结点。目前，2005 年 6 月 TOP 500 排名第 5 位的 IBM JS20 机群（巴塞罗拉超级计算机中心、4800 颗 IBM PPC 970 2.2GHz、Myrinet 互连、峰值性能每秒 42 万亿次）、排名第 7 位的 CDC Thunder (LLNL、4096 颗 Intel Itanium–II Tiger4 1.4GHz、Quadrics 互连、峰值性能每秒 22.9 万亿次）、排名第 31 位国产曙光 4000A (上海超级计算中心、2560 颗 AMD Opteron 2.2GHz、Myrinet 互连、峰值性能

每秒 11.2 万亿次）均属于这一类。2005 年 6 月的 TOP 500 中，机群系统占据了 304 台套，占绝对优势。

(2) 星群 (constellation)。它们也有三个明显的特征：

- 系统由结点构成，每个结点是一台共享存储或者分布共享存储的并行机子系统，包含数十、数百、乃至上千个微处理器，计算功能强大。
- 采用商用机群交换机连接结点，结点间分布存储。
- 在各个结点上，运行专用的结点操作系统、编译系统和作业管理系统。

星群的典型代表为 TOP 500 排名第 3 位的 SGI Columbia 系统（美国 NASA/Ames 研究中心/NAS、20 个结点通过 Voltaire InfiniBand 网络连接、每个结点为 SGI Altix 3700 系统（含由 SGI NUMAlink 连接的 512 颗 Itanium–II 1.5GHz 处理器）、SGI Linux 操作系统、峰值性能为每秒 61 万亿次）。2005 年 6 月的 TOP 500 中，星群占据 79 台套。

(3) 大规模并行机系统（MPP：Massively Parallel Processing）。它们的主要特征为：

- 系统由结点构成，每个结点含 10 个左右处理器，共享存储。处理器采用专用或者商用 CPU。
- 采用专用高性能网络互连，结点间分布存储。
- 系统运行专用操作系统、编译系统和作业管理系统。

此类系统是传统意义的大规模并行处理系统。它区别于其他两种体系结构的一个特征就是，它的处理器或者结点间的互连网络是针对应用需求而特殊定制的，在某种程度上，带有专用并

行机的特点。当前，该类并行机系统大多为政府直接支持，处理器个数可扩展到数十万个。MPP 的典型代表为排名第 1 位的 IBM Blue Gene/L (美国 DOE/NNSA/LLNL、65536 颗 IBM PowerPC 440 700MHz 处理器、IBM Proprietary 专用互连网、峰值性能每秒 184 万亿次)，另一个典型代表是排名第 4 位的日本 NEC Earth-Simulator(日本地球模拟中心、640 个结点通过专用 Multi-satge 交叉开关互连、每个结点含 8 颗向量处理器、峰值性能为每秒 41 万亿次)，还有一个值得关注的典型代表是排名第 10 位的 CRAY Red Storm（美国 Sandia 国家实验室、1250 个结点通过专用 CRAY XT3 互连网络连接、每个结点含 4 个 AMD Opteron 2GHz 的微处理器、峰值性能为每秒 20 万亿次）。2005 年 6 月的 TOP 500 中，MPP 占据 117 台套。

以上仅简单地讨论了并行机的发展。如果读者希望详细了解当前并行机的状况，请参考 TOP 500 排名 [9]。如果希望了解国内并行机的发展状况，也请参考国内 TOP 100 排名 [11]。

1.3 并行机体系结构

了解并行机体系结构是开展并行计算研究的基础。为了设计一个高效率的并行算法，实现一个高效率的并行程序，需要对并行机体系结构有一定的了解。本节从入门角度介绍组成并行机的各个部分，力争使得读者对并行机有一个初步的认识，为深入学习并行算法的设计和并行程序的编制奠定基础。

当前，如图 1.2 和图 1.3 所示，组成并行机的三个要素为：

- 结点（node）。每个结点由多个处理器构成，可以直接输入输出（I/O）。

- 互联网络（interconnect network）。所有结点通过互联网络相互连接相互通信。
- 内存（memory）。内存由多个存储模块组成，这些模块可以如图 1.2 所示，与结点对称地分布在互联网络的两侧，或者，如图 1.3 所示，位于各个结点的内部。

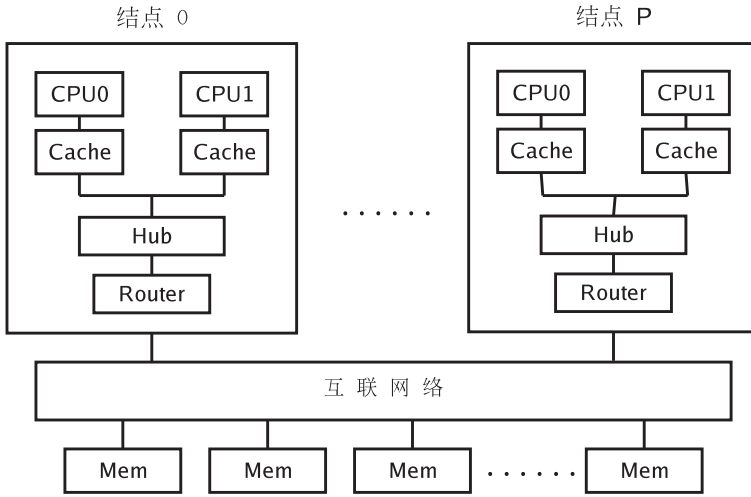


图 1.2 并行机体系结构示意图：内存模块与结点分离

下面分别从结点、互连网络和内存三个方面来简要讨论并行机的体系结构。

1.3.1 结点

结点是构成并行机的最基本单位。以图 1.3 为例，一个结点包含 2 个或 2 个以上微处理器（CPU），并行程序执行时，程序分派的各个进程将并行地运行在结点的各个微处理器上。每个微处理器拥有

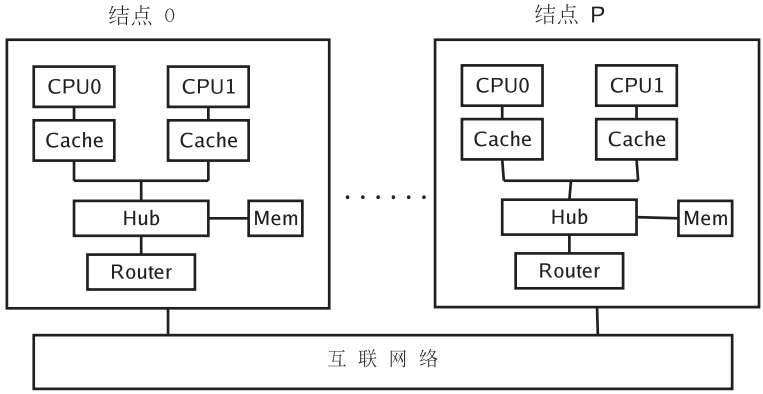


图 1.3 并行机体系结构示意图：内存模块位于结点内部

局部的二级高速缓存（L2 cache）。L2 cache 是现代高性能微处理器用于弥补日益增长的 CPU 执行速度和内存访问速度差距（访存墙）而采取的关键部件。它按 cache 映射策略缓存内存访问的数据，同时为 CPU 内部的一级 cache 提供计算数据。CPU 内部的一级 cache 为寄存器提供数据，寄存器为逻辑运算部件提供数据。有关 cache 的工作原理将在 1.3.3 结合多级存储结构做简单介绍。但是，有关 CPU 内部的体系结构，需要大量的篇幅，一般的读者没必要深入了解这些内容。有兴趣者请参考专著 [12]。

在结点内部，多个微处理器通过集线器（HUB）相互连接，并共享连接在集线器上的内存模块和 I/O 模块，以及路由器（router）。当前，集线器可以提供给微处理器每秒数十 GB 的访存带宽和百个纳秒之内的访存延迟，以及最快每秒 6.4GB 的互联网络访问带宽。

当处理器个数较少时，例如 8–16 个以内，集线器一般采用高速全交互交叉开关，或者高带宽总线完成；如果处理器个数较多，例如数十个，则集线器就等同于并行机的互联网络。有关这些互联的拓扑结构将在下一个小节中介绍。

随着处理器速度的日益增长，结点内配置的内存容量也在增长。传统地，1 个单位的浮点运算速度配 1 个字节的内存单元，是比较合理的。但是，考虑到日益增长的内存墙的影响，这个比例可以适当缩小。例如，1 个单位的浮点运算速度配 0.4 个字节的内存单元。如果以单个微处理器速度为每秒 60 亿次计算，包含 4 个处理器的单结点的峰值运算速度可达 240 亿次，内存空间需要 8GB 以上。于是，在当前并行机的结点内，一般需要采用 64 位的微处理器，才能操作如此大的内存空间。

1.3.2 并行机互联网络拓扑结构

互联网络是连接所有结点成并行机的高速网络。对于一般的并行机使用者，无须知道互联网络底层复杂的通信原理，而只需从拓扑结构的角度了解互联网络。本节简要介绍当前互联网络拓扑结构，以及度量处理器间网络通信性能的一些基本概念。由此，读者可以正确地认识并行机的网络通信能力。

互联网络的拓扑结构可用无向图表示。其中，图中的结点唯一地代表并行机的各个结点，图中的边表示在两个端点代表的并行机结点之间，存在直接连接的物理通信通道。

首先，介绍互联网络拓扑结构的几个重要定义。

- 并行机规模：并行机包含的结点总数，或者包含的 CPU 总数。
- 结点的度：拓扑结构图中，以某个结点为端点的边的条数，称为该结点的度。结点的度表示，存在多少个结点，与该结点有直接的物理连接通道。
- 结点距离：两个结点之间跨越的图的边的最少条数。
- 网络直径：网络中任意两个结点之间的最长距离。

- 点对点带宽：图中边对应的两个结点之间相互连接的物理通道的通信带宽。
- 点对点延迟：图中任意两个结点之间的一次零长度消息传递必须花费的时间。延迟与结点间距离相关，其中所有结点之间的最小延迟称为网络的最小延迟，所有结点之间的最大延迟称为网络的最大延迟；
- 折半宽度 (对分带宽)：对分图成相同规模的两个子图（它们的结点数相等，或者至多相差 1）时，必须去掉的边的最少条数，或者，这些边对应的网络点对点带宽的最小总和；
- 总通信带宽：所有边对应的网络通信带宽之和；

显然，评价一个互联网络的基本准则应该为：固定并行机包含的结点个数，如果点对点带宽越高、折半宽度越大、或者网络直径越小、点对点延迟越小，则互联网络质量可以说越高。

注 1.1: 通常情形下，图中任意两个结点之间的点对点延迟 L 与它们之间的距离 m 有一个简单的关系式

$$L \approx L_0 + \delta \times m \tag{1.1}$$

其中，L_0 为相邻两个结点之间的延迟，δ 表示消息跨越一个结点所需的开销。一般的，$\delta \ll L_0$。因此，当并行机包含的结点个数较少（例如，少于 256），而网络直径又较小时（小于 20），则并行机的任意两个结点之间的点对点网络延迟是基本相等的。也就是说，此时，可近似认为点对点延迟与结点间的距离无关。当前，即使结点规模为 1000 个以内的并行机，点对点的最大延迟和最小延迟之间，也可控制在 2 个微秒差之内。

注 1.2: 点对点带宽是指连接两个结点之间的物理通道在单位时间内可传输的字节总数。通常地，这种传输是双工的，即沿两个不同的

方向，可同时传输数据。点对点带宽一般是指，单位时间内，沿两个方向传输的字节总数。显然，点对点带宽应该与结点之间的距离无关。

注 1.3: 假设图中任意两个结点之间的点对点延迟为 L，点对点带宽为 B，则它们之间一次长度为 N 的消息传递的时间可近似写为：

$$T(N) \approx L + N/B \tag{1.2}$$

按结点间连接的性质，拓扑结构可分为静态拓扑结构、动态拓扑结构和宽带互联网络三类。下面一一介绍。

1. 静态拓扑结构

如果结点之间存在固定的物理连接，且在程序的执行过程中，结点间的连接方式不变，则称该并行机的互联网络拓扑结构是静态的。拓扑结构中，如果两个结点之间存在直接物理连接，则称之互为相邻结点。论著 [2] 详细地讨论了当前并行机采用的各类静态拓扑结构，包括阵列 (array)、环 (ring)、网格 (mesh)、网格环 (torus，也叫环面)、树 (tree)、超立方体 (hypercube)、蝶网 (butterfly)、Benes 网等，以及这些结构的性质。这里简单介绍几个典型的代表，它们是当前 MPP 并行机高速互联网络采用的基本拓扑结构。

一维阵列和环 如图 1.4(a) 所示，规模为 P 的一维阵列由 P 个结点组成，结点 p_i 和结点 $p_j, i, j = 0, \ldots, P-1$，相互连接的充要条件是：

$$|i - j| = 1 \quad i, j = 0, ..., P-1 \tag{1.3}$$

如图 1.4(b) 所示，规模为 P 的环由 P 个结点组成，结点 p_i 和结点 p_j 相互连接的充要条件是：

$$i = j \pm 1 \mod P \quad i, j = 0, \ldots, P-1 \tag{1.4}$$

显然，环通过在一维阵列中增加一条连接边，就可将折半宽度提高到 2，网络直径从 $P-1$ 减少到 $\lceil \frac{P}{2} \rceil - 1$，是对一维阵列的有效改进。对一维阵列和环，结点的度为 2。

(a) 一维阵列　　(b) 环

图 1.4 含 4 个结点的一维阵列和环的拓扑结构

网格和网格环 如图 1.5(a) 所示，d $(d \geqslant 2)$ 维网格包含 $P = P_0 \times P_1 \times \cdots \times P_{d-1}$ 个结点，所有结点排列成 d 维阵列，结点 $(i_0, i_1, \cdots, i_{d-1})$ 和结点 $(j_0, j_1, \cdots, j_{d-1})$ 相互连接的充要条件是

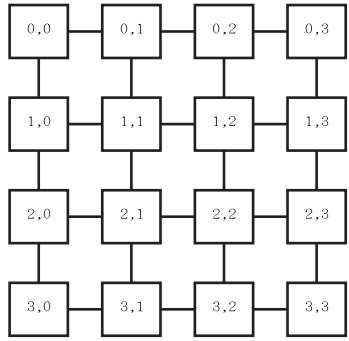
$$\exists k \quad (0 \leqslant k \leqslant d-1) : \begin{cases} i_l = j_l & l \neq k \\ |i_k - j_k| = 1 & l = k \end{cases} \tag{1.5}$$

如图 1.5(b) 所示，d $(d \geqslant 2)$ 维网格环包含 $P = P_0 \times P_1 \times \cdots \times P_{d-1}$ 个结点，所有结点排列成 d 维阵列，结点 $(i_0, i_1, \cdots, i_{d-1})$ 和结点 $(j_0, j_1, \cdots, j_{d-1})$ 相互连接的充要条件是

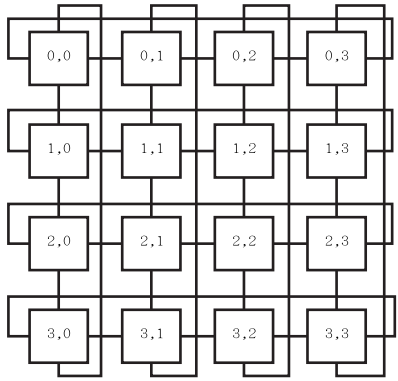
$$\exists k \quad (0 \leqslant k \leqslant d-1) : \begin{cases} i_l = j_l & l \neq k \\ i_k = j_k \pm 1 \mod P_k & l = k \end{cases} \tag{1.6}$$

容易看出，网格和网格环的结点的度均为 $2d$，网格的网络直径等于 $\sum_{k=0}^{d-1}(P_k - 1)$，而网格环的网络直径等于 $\sum_{k=0}^{d-1}(\lceil \frac{P_k}{2} \rceil - 1)$。有关网格和网格环的折半宽度，请读者自己计算 (作业 3)。

树 如图 1.6(a) 所示，标准二叉树拓扑结构包含 $P = 2^N$ 个叶结点和 $2^N - 1$ 个内结点。叶结点分别对应并行机的结点，内结点负责



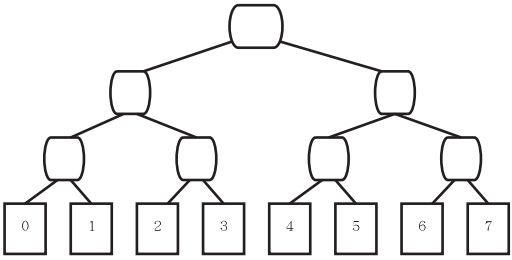
(a) 二维网格



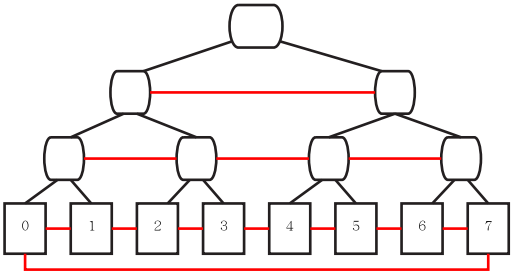
(b) 二维网格环

图 1.5 含 4×4 个结点的二维网格和网格环的拓扑结构

这些叶结点之间的通信。显然，二叉树的网络直径仅为 $2\log P$，非常有利于叶结点之间的全局通信。例如，求累加和、收集数据、广播数据等，但是，它的折半宽度只为 1，不利于结点之间的大数据量通信。



(a) 二叉树



(b) X–树

图 1.6 含 8 个结点的二叉树和X–树的拓扑结构

为了改进二叉树的缺陷，可采用如图 1.6(b) 所示的 X- 树，也就是将每层结点形成环；或者采用胖树，也就是多叉树，每个内结点的

孩子结点个数增加为 4 个以上，减少树的深度。还有一种做法就是，采用混合的拓扑结构。例如，在高维网格环的基础上，沿每个坐标方向，分别建立胖树，然后，将这些胖树的根结点再形成一个胖树。这样，利用网格环的高折半带宽进行大数据量通信，又利用胖树近似最优的网络直径，加速全局通信，获得较优的整体通信性能。

超立方体 如图 1.7(a) 和图 1.7(b) 所示，d 维超立方体 $(d \geqslant 2)$ 包含 $P = 2^d$ 个结点，结点 i 的二进制编号为

$$i = i_0 i_1 i_2 \cdots i_{d-1} \quad i_l = 0, 1 \tag{1.7}$$

结点 $i_0 i_1 \cdots i_{d-1}$ 和结点 $j_0 j_1 \cdots j_{d-1}$ 相互连接的充要条件是

$$\exists k \quad (0 \leqslant k \leqslant d-1) : \begin{cases} i_l = j_l & l \neq k \\ i_k = 1 - j_k & l = k \end{cases} \tag{1.8}$$

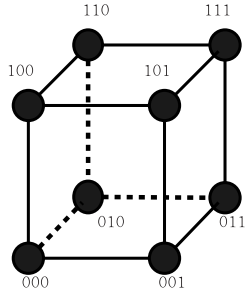
超立方体是一个具有很好性质的拓扑结构，其网络直径仅为 $\log P$，折半带宽为 2^{d-1}。但是，结点的度为 d，随并行机规模的增加而增加，这给网络实现带来了一定的困难。因此，通常地，超立方体一般不超过 5 维。例如，SGI Origin 系列并行计算机的 SuperLinker 网络，采用 5 维的嵌套超立方体拓扑结构，实现高效的通信。

2. 动态拓扑结构

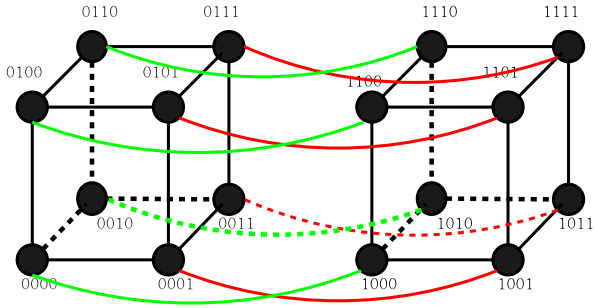
动态拓扑结构是指，结点之间无固定的物理连接，而是在连接路径的交叉点处用电子开关、路由器或仲裁器等提供动态连接，主要包含单一总线、多层总线、交叉开关、多级互联网络等类型。

单一总线 连接处理器、存储模块和 I/O 设备等的一组导线和插座，在主设备（处理器）和从设备（存储器）之间传递数据，其特征主要为：

- 公用总线以分时工作为基础，各处理器模块分时共享总线带宽，即在同一个时钟周期，至多只有一个设备能占有总线；



(a) 三维超立方体



(b) 四维超立方体

图 1.7 超立方体拓扑结构

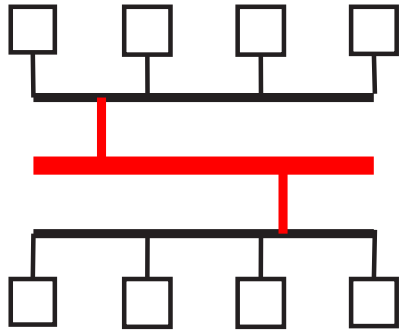
- 总线带宽 = 总线主频 × 总线宽度，例如假设主板的总线频率为 150MHz，总线宽度为 64 位，则该总线的带宽为 1.2GB/s；
- 采用公平的监听协议与仲裁算法，以确定在某个时刻选择哪个设备占有总线；

例如，SGI Power Challenge XL 系列共享存储服务器，多个 CPU 共享总线，在处理器和存储模块之间传递数据。又如，IBM 并行机 ASCI White 的每个结点包含 16 个 IBM PowerPC 处理器，结点内部通过总线共享局部存储器，共享网路通信端口。

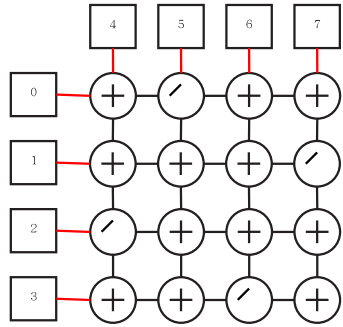
显然，由于多个处理器共享总线，比单一总线拓扑结构的可扩展能力较差。一般地，处理器个数不会大于 32 个。当前，这种拓扑结构大多数用于充当并行机单个结点内部多个 CPU 之间的连接通道。

多层总线 在并行机的结点内部，多个处理器共享本地总线，而结点之间再以另一系统总线相互连接。实际上，这种拓扑结构是对单一总线结构的推广，以提高总线拓扑结构的可扩展能力。图 1.8(a) 给出了一个两层总线的示意图，其中，每条二级总线连接 4 个处理器，而一级总线连接了两条二级总线。

交叉开关 (crossbar switch) 所有结点通过交叉开关阵列相互连接，每个交叉开关均为其中两个结点之间提供一条专用连接通路，同时，任意两个结点之间也能找到一个交叉开关，在它们之间建立专用连接通路。交叉开关的状态可根据程序的要求动态地设置为“开”和“闭”两种状态。图 1.8(b) 给出了一个 4×4 的交叉开关拓扑结构，连接 8 个结点。其中，结点用方框表示，交叉开关用圆圈表示，圆圈内的符号表示了开关的当前状态。其中，符号“+”为“闭”状态，表示该开关同时提供左右通道和上下通道的连接，其他符号表示为“开”状态，并标识当前通信仅能沿符号连接的方向传递。交叉开关拓扑结构具有如下特征：



(a) 两级总线



(b) 4 × 4 交叉开关

图 1.8 动态拓扑结构

- 结点之间的连接：交叉开关一般构成 $N \times N$ 阵列，但在每一行和每一列同时只能有一个交叉点开关处于“开”状态，从而它同时只能接通 N 对结点。

- 一般地，结点和存储器模块作为连接的对象，分别分布在拓扑结构的两侧。

- 结构为 $N \times N$ 的交叉开关只能提供 $2 \times N$ 个端口，这限制了它在大规模并行机中的应用。交叉开关一般仅适合数个处理器的情形，或者，在结点内部为处理器和存储器之间提供快速高效的通道。

多级互联网络 (MIN: Multistage Interconnection Network) 由多个单级交叉开关级连接起来形成大型交叉开关网络，相邻交叉开关级之间存在固定的物理连接拓扑。为了在输入与输出之间建立连接，可以动态地设置开关状态。多级互联网络的典型代表为蝶网、CCC 网和 Benes 网，它们均是超立方体的推广。这里不再介绍，有兴趣者请参考 [45] [2]。

宽带互联网络 随着网络技术的成熟，商用宽带互联网络逐步成为连接微机而构成简易并行机的互联网络，并且，相继推出了专用于微机机群的宽带交换机。它们的出现，极大地丰富了并行机的市场，简化了并行机的研制难度，大幅度降低了并行机的成本，使得微机机群成为科研经费较少的研究所和大学的可用并行机系统。当前，除了专用 MPP 系统采用静态的拓扑结构外，微机机群均采用宽带互联网络连接各个计算结点。下面介绍几个典型的代表。

(1) 以太网 (Ethernet)。以太网络是早期网络连接最常用的，IEEE 802.3 国际标准相继推出了三代网络，性能分别为 10Mbps（82 年）、100Mbps（94 年）和 1Gbps（97 年）。以太网的特征类似于单一总线，也是分时共享的网络协议。例如，如果网络总带宽

为 100Mbps，连接 8 台处理机，则每台处理机的平均带宽仅为 12.5Mbps。

(2) 商用交换机（switch）。商用交换机内部采用全互联方式，可同时为 $N/2$ 对端口提供 100Mbps 的直接连接通路，其中，N 为端口总数。多个 switch 可以堆叠，可形成多级 switch。

(3) 专用微机机群互联网络，它们的功能类似于商用交换机，但是，可提供更强大的通信性能。它们的典型代表为：

- Myrinet。当前，它的点对点带宽可达 250MB/s，延迟小于 $10\mu s$。请参考 [13]。
- Quadrics。当前，它的点对点带宽可达 400MB/s，延迟小于 $6\mu s$。请参考 [14]。
- InfiniBand。当前，它的点对点带宽可达 1.25GB/s，延迟小于 $6\mu s$。请参考 [15]。

1.3.3 多级存储体系结构

得益于主频和超标量指令级流水线技术的发展，现代微处理器的发展仍然遵循 Moore 定律，峰值运算速度每 18 个月翻一番。同时，内存模块的容量以每年几乎翻一番的速度发展。但是，内存模块的访问速度却没有得到平衡的发展。相比较而言，内存的访问速度要比处理器执行速度慢很多，难以发挥微处理器的峰值速度，这就是所谓的内存墙 (memory wall) 性能瓶颈问题。

为了尽量克服内存墙对性能的影响，一个简捷的方法是，在内存和处理器之间增加一个高速缓冲区，称为 cache，其主要目的是，缓存内存模块的部分数据，尽量将内存的访问转换为对 cache 的访问，这样，可以大幅度缩短内存数据的访问时间。一般地，在结点内部的 cache 称为二级 cache (L2 cache)。在处理器内部，还存在一

个更小容量的 cache，称之为一级 cache (L1 cache)。L1 cache 连接 CPU 寄存器和 L2 cache，负责缓存 L2 cache 中的数据到寄存器中。因此，如图 1.9 左端所示，并行机的存储空间可以分解为多个层次，位于最顶层的是 CPU，它从寄存器中读取数据；寄存器从 L1 cache 中读取数据。CPU、寄存器和 L1 cache 构成微处理器芯片 (chip)。L1 cache 从 L2 cache 中读取数据，而后者从本地局部内存中获取数据。微处理器芯片和 L2 cache 通常可视为微处理器。多个微处理器和它们共享的局部内存模块构成一个完整的结点。在本地局部内存之外，还存在其他结点的局部内存模块，称之为远程内存空间。显然，远程内存空间可构成结点的海量“外存”空间。

如图 1.9 右端所示，从下往上，每个字节的成本越来越高，但是，访存速度越来越快，即 访存延迟越来越小，带宽越来越高。以

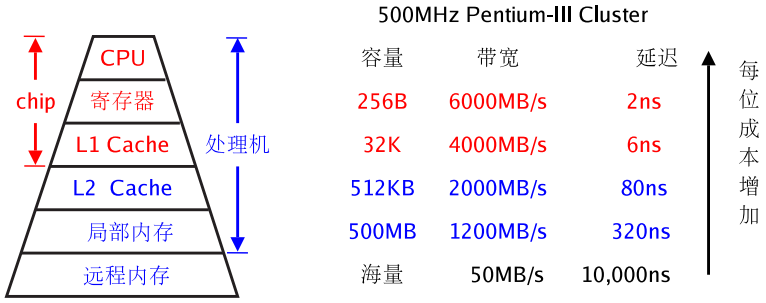


图 1.9 多级存储结构示意图

500MHz Intel Pentium–III 为处理器的微机机群为例，假设处理器之间通过 400Mbps 的网络交换机相互连接，则可以看出，内存访问时，数据在 cache、局部内存，还是在远程内存，所需时间差距很大，高达几十甚至数百上千倍。因此，为了编制发挥处理器峰值性能的高性能并行程序，必须对并行机的多级存储结构有一定的了解。主要

包括两个方面，一个是 cache 的映射策略，即 cache 是如何从内存中取数并存储的；另一个是结点内部或者结点之间内存的访问模式。下面主要从这两个方面来介绍并行机的多级存储体系结构。这些内容在当代并行机体系结构中，具有普遍的意义。

本小节以 L2 cache 为例，简要介绍 cache 的原理。从 L1 cache 到 L2 cache 的访问，也是类似的原理。

Cache 以 cache 线 (line) 为其基本组成单位，每条 cache 线包含 L 个字，每个字为 8 个字节。例如，$L = 4$，则表示 cache 线包含 $4 \times 8 = 32$ 个字节；$L = 8$，则表示 cache 线包含 $8 \times 8 = 64$ 个字节。内存空间分割成块（block），每块大小与 cache 线长度一致（L 个字）。数据在内存和 cache 之间的移动，以 cache 线为基本单位，即数据从内存调入 cache 时，不是以该单个数据字为单位，而是以该数据所在的内存块为单位，将该块的 L 个字一次调入 cache，存储在对应的 cache 线中；同时，如果 cache 中的数据单元要求写入内存空间，则也必须以 cache 线为单位，即该数据单元所在 cache 线中的所有内容一次写入内存中对应的块中。

显然，cache 能减少访存次数的一个内在原因就是，程序的数据访问具有局部性，即程序中连续使用的数据一般存储在内存的连续位置。因此，通过 cache 线的一次调入，随后的数据访问可能就落在 cache 线中，从而达到减少内存访问的次数。例如，考虑如下简单的 Fortran 循环。

```
DO I = 1, M
   A(I) = A(I) + 5.0 * B(I)
ENDDO
```

假设没有 cache，则内存读访问次数为 $2M$ 次。如果有 cache，则内存访问次数下降为 $2M/L$ 次。从而，由前面的多级存储结构性能分析数据可知，cache 的使用必定加快程序的执行速度。

一次内存访问操作，如果操作数存在于 cache 中，则称该次访

问是 *cache* 命中的，否则，称该次操作是 *cache* 失效的。为了衡量 cache 命中的频率，定义 *cache* 命中率为，程序执行过程中，cache 命中的总次数和内存访问总次数之比值。

Cache 设计必须考虑的几个关键问题为 cache 容量、cache 线大小、cache 个数、cache 的映射策略、cache 线的置换策略、cache 数据一致性策略等。

1. Cache 的容量

Cache 容量越大，则程序执行的性能将越高。但是，在实际应用中，大量数据表明，超大容量 cache 并不能显著提高性能，反而显著提供处理器的价格。因此，为了追求价格和性能的平衡，L2 cache 的容量一般为 1MB–16MB，且只复制程序的数据；L1 cache 的容量比 L2 cache 的容量小几个数量级，一般为数 KB 到数十 KB。

2. Cache 线的大小

Cache 线越大，则一次载入的内存数据也越多，提高性能的潜力就越大。但是，给定 cache 的容量，则 cache 线越大，cache 线的条数就越少，产生 cache 访问冲突的可能性就越大。因此，cache 线的大小也应该有一个合适的长度。当前，cache 线一般为 4–8 个字。

3. Cache 的个数

在当前的大多数微处理器中，cache 一般可分为两级，其中，一级 cache 又称为 on-chip cache，二级 cache 称为 off-chip cache。一级 cache 一般还分为两个，一个存储指令，另一个存储数据。二级 cache 既要存储指令，又要存储数据。在某些微处理器中，cache 可分为三级。

4. Cache 映射策略

如何在内存块和 cache 线之间建立相互映射关系，是具体实现 cache 读取内存操作的必要环节。按映射策略，内存块的数据能够且只能复制到被映射的 cache 线中，而 cache 线中的数据也能够且只能被映射到对应的内存块中。当前，cache 的映射策略可分为直接映射策略、K–路组关联映射策略和全关联映射策略三种。下面一一介绍。

- 直接映射策略

  所谓直接映射策略，指每个内存块只能被唯一地映射到一条 cache 线中。

  假设 cache 总共包含 $M = 2^r$ 条 cache 线 (每条线的长度为 $L = 2^w$ 个字)，所有内存块按序编号，则第 j 块内存将被映射到第 i 条 cache 线中，其中

  $$i = j \mod M \tag{1.9}$$

  假设内存地址空间（以字为单位）总共可用 $D = s + w = (s-r)+r+w$ 位表示，则显然，内存块和 cache 线之间的映射关系可以唯一地用内存地址的编号来自然地实现硬件的映射，且每条 cache 线对应 2^{s-r} 个内存块。例如，假设 $D = 20, r = 8, w = 2$ 表示，内存总共包含 2^{20} 个字，cache 包含 $256 = 2^8$ 条 cache 线，每条 cache 线的长度等于 $4 = 2^2$ 个字。则内存中地址为 $a = a_0a_1 \cdots a_{19}a_{20}$ 的字将被唯一地映射到第 $b = a_{11} \cdots a_{18}$ 条 cache 线的第 $c = a_{19}a_{20}$ 个字节处。每条 cache 线对应 1024 个内存块。

  直接映射策略很容易产生 *cache* 冲突（cache thrashing），即连续访问的数据单元位于不同的内存块中，而这些块被映射到

同一条 cache 线中，连续的数据访问将引起频繁的 cache 线置换。例如：假设 cache 包含 M 条 cache 线，$M = 2^r$，cache 线包含 $L = 2^w$ 个字，令 $N = M \times L = 2^{r+w}$，假设程序有四个数组：

```
COMMON /XXX/ A(N), B(N), C(N), D(N)
```

则下面的循环将发生严重的 cache 冲突

```
DO I = 1, N
   A(I) = A(I) + B(I) + C(I) + D(I)
ENDDO
```

这是因为，四个数组中，下标为 I 的同一个元素被映射到同一条 cache 线，于是，每次循环均将引起 4 次 cache 线置换，cache 对程序的性能没有任何改进。

为了改进上面的 cache 冲突，从程序设计的角度，通常采用“补边法 (padding)”，在 COMMON 块的设计中，每个数组的长度尽量避免出现 2 的幂次。如果出现了 2 的幂次，需要在各个数组之间增加一些辅助数组，使得各个数组至少错开一条 cache 线的距离，避免下标相同的元素被映射到同一条 cache 线。例如，

```
COMMON /XXX/ A(N), X1(K), B(N), X2(K), C(N), X3(K), D(N)
```

其实，在某些编译器中，有自动补边的优化功能。

- K–路组关联映射策略 (K–way set association mapping strategy)

  Cache 被分解为 V 个组，每个组由 K 条 cache 线组成，内存块按直接映射策略映射到某个组，但是在该组中，内存块可

以映射到任意一条 cache 线，也就是说，每个内存块可以映射到 cache 中的 K 条 cache 线的任何一条。具体算法为：

$$i = j \mod V \tag{1.10}$$

其中，i 为 cache 组的序号，j 为内存块的序号，V 为 cache 中 cache 组的个数。

类似地，按内存地址的编号，可以实现内存地址和 cache 地址之间的自然映射。假设 cache 线长度为 2^w 个字，cache 包含 $V = 2^d$ 个组，每个组包含 $K = 2^r$ 条 cache 线，内存空间包含 2^{s+w} 个字，则内存空间每隔 2^{s-d-r} 个字的内存块被映射到同一个 cache 组中。

显然，K–路组关联映射策略可以减少 cache 冲突的可能性，例如，对前面的例子，如果 $K = 4$，则 cache 冲突就可被消除。当前，大多数并行机采用的 cache 映射策略是2–路或4–路组关联映射策略。

- 全关联映射策略 (full association mapping strategy)

  内存块可以被映射到 cache 中的任意一条 cache 线，这种映射策略的实现比较复杂，通常只有理论上存在的价值，在实际中并不多见。

5. Cache 线的置换策略

对K–路组关联映射策略，当某个内存块请求被置入时，如何选择组中的某条 cache 线，将其数据写回内存（如果该条 cache 线的数据被修改），有多种算法。比较常用的算法包括：

- LRU (Least Recently Used) 算法：置换没引用时间最长的 cache 线；

- FIFO (First Input First Output) 算法：置换最先置入的 cache 线；
- LFU (Least Frequently Used) 算法：置换使用频率最低的 cache 线；
- 随机算法：随机选择一条 cache 线置换。

6. Cache 数据的一致性策略

显然，为了保持计算结果的正确性，必须设计某种策略，保持 cache 数据和内存数据的一致性。通常有两种策略：

- Write–through 策略：cache 线中的数据一旦被修改，则立即写入内存块。它的缺点是，增加了许多不必要的内存访问。
- Write–back 策略：当且仅当要求进行 cache 线置换时，或者有外部请求访问内存块时，将 cache 线的数据写入内存。

综上所述，以读操作为例，cache 的工作流程可用图 1.10 示意。

1.3.4 访存模型

根据内存访问的性质，并行机的访存模型可以分为均匀访存模型 (UMA：Uniform Memory Access model)、非均匀访存模型 (NUMA：Non-Uniform Memory Access model)、分布式访存模型 (DMA：Distributed Memory Access model) 及混合访存模型 (HMA：Hybrid Memory Access model) 四类。本小节只是简单地介绍这些模型的主要特征，它们对应的并行机系统将在并行机分类一节中介绍。

(1) **均匀访存模型**。如图 1.2 所示，内存模块与结点分离，分别位于互联网络的两侧，互联网络一般采用系统总线、交叉开关或多级网络，称之为紧耦合系统（tightly coupled system）。该模型具有如下典型特征：

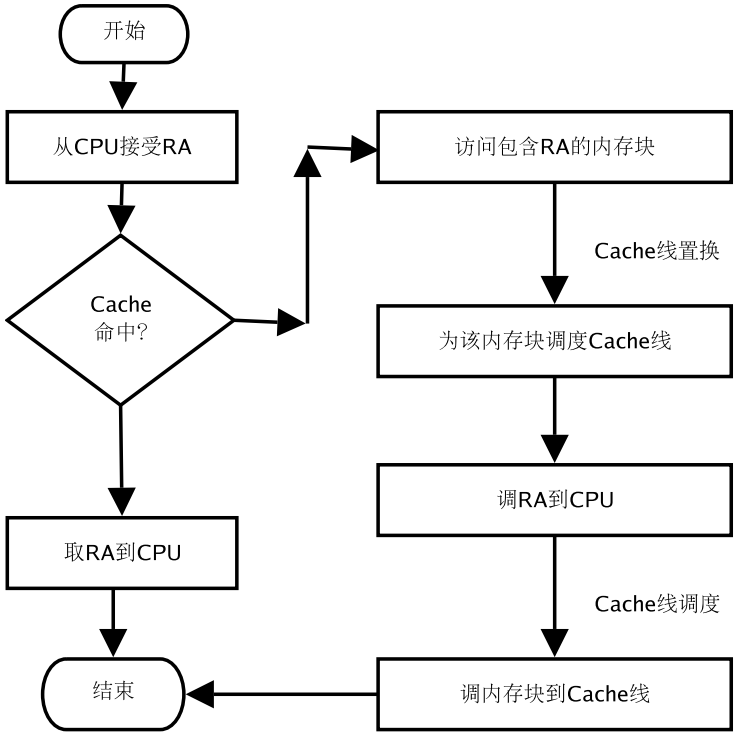


图 1.10　Cache 读操作工作流程示意图

- 物理存储器被所有结点共享；
- 所有结点访问任意存储单元的时间相同；
- 发生访存竞争时，仲裁策略平等对待每个结点，即每个结点机会均等；
- 各结点的 CPU 可带有局部私有高速缓存（cache）；
- 外围 I/O 设备也可以共享，且每个结点有平等的访问权利。

当前对称多处理共享存储并行机（SMP: Symmetric Multi-Processing）均采用这种模型。在并行机的分类中将再讨论此类并行机的主要特征。

(2) **非均匀访存模型**。如图 1.3 所示，内存模块局部在各个结点内部，所有局部内存模块构成并行机的全局内存模块。并行机的内存模块在物理上是分布的，但是，在逻辑上是全局共享的。非均匀访存模型也可称为分布共享访存模型。该模型具有如下典型特征：

- 物理存储器被所有结点共享，任意结点可以直接访问任意内存模块；
- 结点访问内存模块的速度不同：访问本地存储模块的速度一般是访问其他结点内存模块的 3 倍以上；
- 发生访存竞争时，仲裁策略对结点可能是不等价的；
- 各结点的 CPU 可带有局部私有高速缓存（cache）；
- 外围 I/O 设备也可以共享，但对各结点是不等价的。

非均匀访存模型的典型例子为 SGI Origin 系列并行机，它们采用基于 cache 目录一致性的非均匀访存模型（CC–NUMA：Cache–Coherent Nonuniform Memory Access），设计了专门的硬件，保证在任意时刻，各结点 cache 中数据与全局内存数据的一致性。限于篇幅，这里不介绍并行机各个结点之间的 cache 一致性原理，有兴趣的读者，请参考并行机体系结构的专著。

(3) **分布访存模型** 该模型的内存模块的物理分布也如图 1.3 所示，但与非均匀访存模型不同的是，各个结点的存储模块只能被局部 CPU 访问，对其他结点的内存访问只能通过消息传递程序设计来实现。一般地，每个结点均是一台由处理器、存储器、I/O

设备组成的计算机。当前的 MPP 并行机各个结点之间，或者微机机群各个结点之间，均是这种访存模型。

(4) **混合访存模型** 混合访存模型是前面 3 类访存模型的优化组合。典型的星群系统中，每个结点内部是均匀访存模型或者非均匀访存模型，结点之间是分布访存模型。当前 MPP 系统中，大多采用混合访存模型。

1.3.5 并行机分类

并行机的分类是随并行机体系的发展而发展的。从不同的角度，并行机有不同的分类。如果按经典的指令与数据流进行分类，则并行机可以分为三类：

- 单指令多数据流（SIMD）：按同一条指令，并行机的各个不同的功能部件同时对不同的数据进行不同的处理，例如：传统的向量机、80 年代初期的阵列机 CM–2。目前，这类并行机已经退出历史舞台。
- 多指令多数据流（MIMD）：不同的处理器可同时对不同的数据执行不同的指令，目前所有并行机均属于这一类。
- 多指令单数据流（MISD）：至今没出现。

按内存访问模型、微处理器和互联网络的不同，当前流行的并行机可分为对称多处理共享存储并行机（SMP：Symmetric Multi-Processing）、分布共享存储并行机（DSM：Distributed Shared Memory）、机群（cluster）、星群（constellation）和大规模并行机（MPP：Massively Parallel Processing）等五类。

(1) **对称多处理共享存储并行机（SMP）**。如图 1.11 所示，内存模块和处理器对称地分布在互联网络的两侧，内存访问属典型的均匀访问模型。SMP 并行机有如下主要特征：

- 对称共享存储：系统中任何处理器均可直接访问任何存储模块中的存储单元和 I/O 模块，且访问的延迟、带宽和访问成功的概率是一致的。所有内存单元统一编址。各个处理器之间的地位等价，不存在任何特权处理器。操作系统可在任意处理器上运行。
- 单一的操作系统映像：全系统只有一个操作系统驻留在共享存储器中，它根据各个处理器的负载情况，动态地分配各个进程到各个处理器，并保持各处理器间的负载平衡。
- 局部高速缓存 cache 及其数据一致性：每个处理器均配备局部 cache，它们可以拥有独立的局部数据，但是这些数据必须与存储器中的数据保持一致。
- 低通信延迟：各个进程通过读/写操作系统提供的共享数据缓存区来完成处理器间的通信，其延迟通常小于网络通信的延迟。
- 共享总线带宽：所有处理器共享总线的带宽，完成对内存模块和 I/O 模块的访问。
- 支持消息传递、共享存储并行程序设计。

SMP 并行机具有如下缺点：

- 欠可靠：总线、存储器或操作系统失效可导致系统崩溃。
- 可扩展性（scalability）较差：由于所有处理器共享总线带宽，而总线带宽每 3 年才增加 2 倍，跟不上处理器速度和内存容量的增加步伐，因此，SMP 并行机的处理器个数一般少于 32 个，且只能提供每秒数百亿次的浮点运算性能。

SMP 的典型代表为：

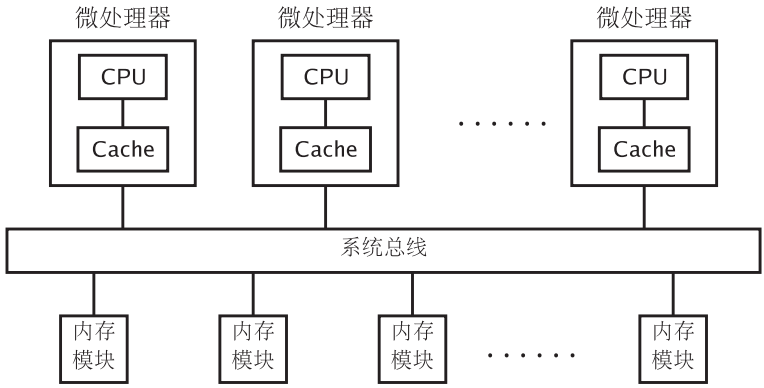


图 1.11 SMP 体系结构典型示意图

- SGI POWER Challenge XL 系列并行机（可扩展至 36 个 MIPS R10000 微处理器）。
- COMPAQ Alphaserver 84005/440 （含 12 个 Alpha 21264 微处理器）。
- HP9000/T600 （含 12 个 HP PA9000 微处理器）。
- IBM RS6000/R40（含 8 个 RS6000 微处理器）。

(2) **分布共享存储并行机（DSM）**。如图 1.3 所示，内存模块局部在各个结点内部，并被所有结点共享。这样，可以较好地改善对称多处理共享存储并行机的可扩展能力。DSM 并行机具有如下主要特征：

- 并行机以结点为单位，每个结点包含一个或多个 CPU，每个 CPU 拥有自己的局部 cache，并共享局部存储器和 I/O 设备，所有结点通过高性能互联网络相互连接；
- 物理上分布存储：内存模块分布在各结点中，并通过高性

能互联网络相互连接，避免了 SMP 访存总线的带宽瓶颈，增强了并行机的可扩展能力。

- 单一的内存地址空间：尽管内存模块分布在各个结点，但是，所有这些内存模块都由硬件进行统一编址，并通过互联网络连接形成了并行机的共享存储器。各个结点既可以直接访问本地局部内存单元，又可以直接访问其他结点的局部内存单元。
- 非一致内存访问（NUMA）模式：由于远端访问必须通过高性能互联网络，而本地访问只需直接访问局部内存模块，因此，远端访问的延迟一般是本地访问延迟的 3 倍以上。
- 单一的操作系统映像：类似于 SMP，在 DSM 并行机中，用户只看到一个操作系统，它可以根据各结点的负载情况，动态地分配进程。
- 基于 cache 的数据一致性：通常采用基于目录的 cache 一致性协议来保证各结点的局部 cache 数据与存储器中数据的一致性。同时，也称这种 DSM 并行机结构为 CC–NUMA 结构。
- 低通信延迟与高通信带宽：专用的高性能互联网络使得结点间的延迟很小，通信带宽可以扩展。例如，SGI Origin 3000 的双向点对点通信带宽可达 3.2GB/s，而延迟小于 1 个 μs。
- DSM 并行机可扩展到数百个结点，能提供每秒数千亿次的浮点运算性能。例如，SGI Origin–2000 可以扩展到 64 个结点（128 个 CPU），SGI Origin 3800 系统可扩展到 256 个结点（512 个 CPU），SGI Altix 系统可以扩展到 512 个结点（1024 个 CPU）。但是，由于受 cache 一致性要求和互联

网络性能的限制，当结点数目进一步增加时，DSM 并行机的性能也将下降。

- 支持消息传递、共享存储并行程序设计。

分布共享存储并行机的典型代表为 SGI Origin–2000（图 1.1）、SGI Origin 3800 和 SGI Altix 系统，这些系统具有很好的综合性能，使用非常方便、高效。以 SGI Altix 系统为例，它提供给用户的是一个单一操作系统的用户交互式界面，可以象工作站一样被用户使用。同时，由于它实现了全局硬件编址的分布共享存储，进程间消息传递的延迟低于 1 个微秒，带宽高达 6.4GB/秒，全局并行文件系统的性能与计算性能相互匹配。对于科学计算用户，此类并行机是一个很好的选择，但是，该并行机的价格也比较昂贵。

(3) 机群系统。请参考第 1.2.8 节 (第 18 页)。

(4) 星群系统。请参考第 1.2.8 节 (第 19 页)。

(5) 大规模并行机系统（MPP）。大规模并行机系统是典型的分布存储系统，其体系结构如图 1.12 所示。其典型特征为：

- 由数百个乃至数千个计算结点和 I/O 结点组成，每个结点相对独立，并拥有一个或多个微处理器。这些结点配备有局部 cache，并通过局部总线或互联网络与局部内存模块和 I/O 设备相连接。通常地，这些微处理器针对应用特征，进行了某些定制，与商用的通用微处理器略有不同。
- 这些结点由局部高性能网卡 (NIC) 通过高性能互联网络相互连接。互联网络与机群互联网络不同，它一般采用由多种静态拓扑结构耦合而成的混合拓扑结构，其通信延迟和通信带宽均明显优于机群互联网络。

- MPP 的各个结点均拥有不同的操作系统映像。一般情况下，用户可以将作业提交给作业管理系统，由它负责调度当前最空闲、最有效的计算结点来执行该作业。但是，MPP 也允许用户登录到某个特定的结点，或在某些特定的结点上运行作业。
- 各个结点间的内存模块相互独立，且不存在全局内存单元的统一硬件编址。一般情形下，各个结点只能直接访问自身的局部内存模块，如果要求直接访问其他结点的局部内存模块，则必须有操作系统的特殊软件支持。
- 仅支持消息传递或者高性能 Fortran 并行程序设计，不支持全局共享的 OpenMP 并行程序设计模式。

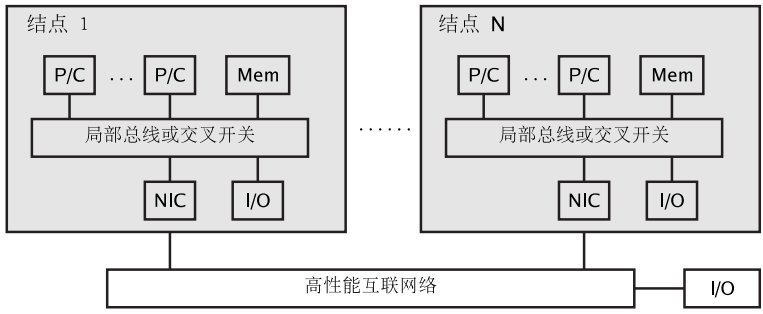


图 1.12 MPP 体系结构典型示意图

当前，TOP 500 中排名第一的 Blue Gene/L 就是典型的 MPP 系统。其他 MPP 系统，请参考第 1.2.8 节 (第 19 页)。

1.4 操作系统与并行编程环境

为了在并行机上开展并行计算研究工作，或者设计并行程序，除了对并行机体系结构有一定的了解外，还需要对并行机系统软件（操作系统、编译系统等）有一定的了解。本节简单地介绍这些方面的一些基本概念，重点在于学习消息传递或者共享存储并行编程所必须了解的基本概念：进程和线程。

UNIX 操作系统几乎是当前所有高性能并行机（SMP、DSM、cluster、constellation、MPP）采用的标准操作系统，其中包括 SGI 公司的 IRIX，COMPAQ 公司的 True64，HP 公司的 HPUX，IBM 公司的 AIX、SUN 公司的 Solaris 和自由软件 Linux 等。虽然各并行机厂商研制的 UNIX 操作系统的实现原理不尽相同，但是，它们给用户提供的基本 UNIX 操作系统界面是大体一致的。因此，用户只要对 UNIX 操作系统有一定的了解，就可以方便地使用以上介绍的各类并行机。

在程序设计语言方面，SMP、DSM 和 MPP 并行机一般均提供符合国际标准的 Fortran 77/90、C/C++、HPF 等语言，而机群系统一般免费提供 GNU Fortran 77/95、GNU C/C++ 等语言。特别地，1.4.2 将介绍各类并行机支持的并行程序设计平台。

1.4.1 进程、进程间通信与线程

现代 UNIX 操作系统中，与消息传递并行程序设计密切相关的一个重要概念便是进程 (process)。正是由于多个进程之间的相互通信，才决定了各类消息传递并行程序设计平台的出现。本节主要介绍进程和进程间通信的基本概念。

1. 进程

进程（process）可表示成四元组 (P, C, D, S)，其中 P 是程序代码、C 是进程的控制状态、D 是进程的数据、S 是进程的执行状态。

任何进程总和程序联系在一起，程序一旦在具体操作系统环境中投入运行，就变成了进程。各个进程拥有独立的执行环境，其中包括内存数据和指令地址空间、程序计数器、寄存器、栈空间、文件系统、I/O 设备等，并在操作系统的控制、管理、保护和调度下，在不同的时刻，动态地申请和占有计算资源。特别地，称进程的内存地址空间为该进程的局部内存空间。

进程具有两个明显的特征：一个是资源特征，包括那些程序执行所必需的计算资源，例如程序代码、内存地址空间、文件系统、I/O 设备、程序计数器、寄存器、栈空间等；另一个是执行特征，包括那些在进程执行过程中动态改变的特征，例如指令路径（即进程执行的指令序列）、进程的控制与执行状态等。进程的资源特征反映了进程是资源拥有的最小单元，而执行特征反映了进程是操作系统调度的基本单元。即使是同一个程序，不同的程序执行也将产生不同的进程，因为这些进程拥有完全不同的执行特征。

任何进程，在执行过程中，均涉及如下几种状态：

- **非存在状态**：进程依赖的程序还没有投入运行；
- **就绪状态**：进程由其父进程（例如，操作系统的内核进程或 Shell 进程，或其他应用程序进程）调入并准备运行；
- **运行状态**：进程占有 CPU 和其他必需的计算资源，并执行指令；
- **挂起状态**：由于 CPU 或其他必需的计算资源被其他进程占有，或必需等待某类事件的发生，进程转入挂起状态，以后一旦条件满足，由操作系统唤醒并转入就绪状态；
- **退出状态**：进程正常结束或因异常退出而被废弃。

只对消息传递并行程序设计感兴趣的读者，了解以上的进程概念就够了，有关进程的详细定义，将涉及到进程的映像、进程的执行

模式、进程的现场活动、进程描述符和进程控制等诸多方面的知识，有兴趣的读者请参考专门的操作系统专著。

2. 进程间通信

多个进程可同时存在于同一台处理机中，拥有独立的执行环境，在操作系统的调度下，分时共享计算机资源。但是，它们拥有的内存指令和数据地址空间必须互不相交，且每个进程只能访问自己的局部内存空间。如果一个进程执行的某条指令要求访问该进程局部内存空间之外的内存地址单元，则隐含该进程的程序存在错误，而进程的执行可能会中断。当然，位于不同处理机中的多个进程也拥有独立的执行环境，在各自操作系统的调度下，占有各自的计算机资源。

无论位于同一台处理机，还是位于不同处理机，进程始终是操作系统资源调度的基本单位，且各个进程不能直接访问其他进程的局部内存空间。但是，现代操作系统提供基本的系统调用函数，允许位于同一台处理机或不同处理机的多个进程之间相互交流信息，具体表现为三种形式：通信、同步和聚集。

- 通信：进程间的数据传递称为进程间通信。在同一台处理机中，通信可以通过读/写操作系统提供的共享数据缓存区来实现；在不同处理机中，通信可以通过网络传输数据来实现。特别地，称两个进程之间传递的数据为消息，称这种操作为消息传递。显然，消息传递可以在同一台处理机的多个进程之间发生，也可以在不同处理机的多个进程之间发生。

- 同步：同步是使位于相同或不同处理机中的多个进程之间相互等待的操作，它要求进程的所有操作均必须等待到达某一控制状态之后才进行。其实，同步也是进程之间相互通信的一种方式。

- 聚集 (或归约)：聚集将位于相同或不同处理机中的多个进程的局部结果综合起来，通过某种操作，例如求最大值、最小值、累加和，产生一个新的结果，存储在某个指定的或者所有的进程的变量中。其实，聚集也是进程间相互通信的一种形式。

在以后的讨论中，为了方便，将进程间相互操作的三种形式：通信、同步和聚集，统称为进程间通信，而操作的具体数据对象为消息，具体操作为消息传递。

进程间通信的具体实现大体可以分为两类：(1) 在共享存储环境中，通过读/写操作系统提供的共享数据缓存区来实现；(2) 在分布式存储网络环境中，通过网络通信来实现。但是，无论哪种形式，实现的具体细节对并行编程的用户都是屏蔽的，用户看到的均是统一的应用程序接口（API）。

在以后各章消息传递 MPI 并行编程的详细介绍中，读者将会逐步深入地理解进程间通信各种形式的具体含义。

3. 线程

为了更好地理解消息传递并行编程环境，这里介绍另一个重要概念：线程（threads），它是在进程的基础上，基于对称多处理的现代操作系统的一个重要发明。

由于进程具有独立的局部内存空间，使得操作系统对它们的管理非常费时。例如，当 UNIX 进程执行 `fork()` 系统调用生成一个子进程时，操作系统必须为该子进程分配内存地址空间和寄存器、复制父进程描述符、设置运行栈空间、保留进程上下文、切换进程，所有这些操作是非常费时的。为此，该类 UNIX 进程称为重量级进程。

由于管理重量级进程的高开销较大地影响了并行机性能的发挥，因此，该类进程不适合细粒度的共享存储并行程序设计。为了在共享存储环境下有效地开发应用程序的细粒度并行度，将一个进程分解两个部分，其中一部分由其资源特征构成，仍称之为进程；另一部

分由其执行特征构成，称之为线程，或者轻量级进程。具体地，如图 1.13 所示，进程的指令路径可以分解为并行的互不相关的多条子路径（用曲线表示），而每条子路径可由一个线程来执行。因此，进程可由单个线程来执行，即通常所说的串行执行；同时，进程也可由多个线程来并行执行，此时，多个线程将共享该进程的所有资源特征，并可以使用不同的 CPU，对不同的数据进行处理，从而达到提高进程执行速度的目的。

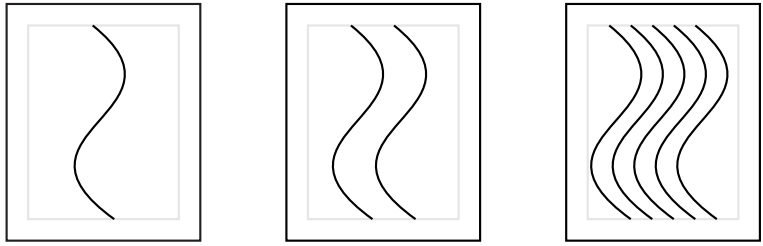


图 1.13 单进程多线程执行示意图，其中曲线表示执行的指令路径

线程由操作系统内核施行管理，由线程库具体实现。进程产生时，其执行特征构成一个线程，称之为主线程。主线程调用线程库函数，可以动态地创建新的线程，称之为从线程。主线程和从线程共享进程的资源特征。当一个从线程产生时，操作系统不必为该线程分配局部内存地址空间，而只需为它创建指令执行所必需的线程上下文和局部数据栈空间、分配寄存器和程序计数器等资源。同时，线程间的切换开销也远远小于进程间的切换开销。因此，线程的管理开销远远小于进程的管理开销，比较适合细粒度的共享存储的并行程序设计。

有关线程的详细定义，请读者参考操作系统专著，这里不再深入讨论。

1.4.2 并行编程环境

在当前并行机上，比较流行的并行编程环境可以分为三类：消息传递、共享存储和数据并行，它们的典型代表、可移植性、并行粒度、并行操作方式、数据存储模式、数据分配方式、学习难度、可扩展性等方面的比较在表 1.1 中给出。由该表可以看出：

(1) 共享存储并行编程基于线程级细粒度并行，仅被 SMP 和 DSM 并行机所支持，可移植性不如消息传递并行编程。但是，由于它们支持数据的共享存储，所以并行编程的难度较小，但一般情形下，当处理机个数较多时，其并行性能明显不如消息传递编程。

(2) 消息传递并行编程基于大粒度的进程级并行，具有最好的可移植性，几乎被当前流行的各类并行机所支持，且具有很好的可扩展性。但是，消息传递并行编程只能支持进程间的分布存储模式，即各个进程只能直接访问其局部内存空间，而对其他进程的局部内存空间的访问只能通过消息传递来实现。因此，学习和使用消息传递并行编程的难度均大于共享存储和数据并行两种编程模式。

本书的主要目的是全面介绍消息传递并行编程环境 MPI，因此，在以后的篇幅中，将不再讨论共享存储和数据并行编程环境，有兴趣者请参考相关文献。

1. 消息传递并行机模型

由于当前流行的各类 SMP、DSM、MPP 和微机机群等并行机均支持消息传递并行程序设计，因此，有必要对这些具体并行机的体系结构进行抽象，设计一个理想的消息传递并行机模型。基于该模型，用户可以在不考虑具体并行机体系结构的条件下，组织消息传递并行程序设计，从而简化并行程序设计，增强程序的可移植性。

表 1.1 三种并行编程环境主要特征一览表

特征	消息传递	共享存储	数据并行
典型代表	MPI、PVM	OpenMP	HPF
可移植性	所有流行并行机	SMP、DSM	SMP、DSM、MPP
并行粒度	进程级大粒度	线程级细粒度	进程级细粒度
并行操作方式	异步	异步	松散同步
数据存储模式	分布式存储	共享存储	共享存储
数据分配方式	显式	隐式	半隐式
学习入门难度	较难	容易	偏易
可扩展性	好	较差	一般

图 1.14 给出了一个理想的消息传递进程拓扑结构。其中，“P”表示 MPI 进程，“M”表示每个进程的局部内存空间，多个“P/M”进程/内存模块通过互联网络相互连接，构成一个分布式存储的进程拓扑结构。在该结构中，各个进程之间可以直接通信，但是各个进程只能直接访问自身的局部内存空间，对其他进程的局部内存空间的访问只能调用消息传递函数，通过进程间通信才能实现。因此，该进程拓扑结构的核心是连接进程的互联网络，也就是消息传递标准函数库，而构成该函数库的所有函数就构成了用户面对的消息传递并行编程环境。

如果将图 1.14 的每个 P/M 模块替换成处理器，且规定每个处理器只能分配用户程序的一个进程，则所得的理想并行机模型就是消息传递并行机模型。不难看出，消息传递并行程序设计所依赖的并行机模型实际上属于典型的分布式存储并行机，且每台处理器只能分配用户程序的一个进程。基于该并行机模型，用户可以自由地调用消息传递函数库中的函数来组织具体的并行程序设计，且程序研制成功后，便可以在任何支持该并行机模型隐含的进程拓扑结构的所有具体并行机上运行。

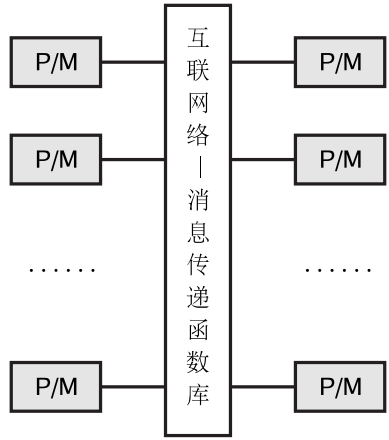


图 1.14 消息传递进程拓扑结构和并行机模型

这里，有必要说明的是，消息传递分布式存储并行机模型和具体并行机体系结构没有必然的联系。无论将该模型映射到何种类型的并行机（SMP、DSM、cluster、constellation、MPP），用户面对的都是该模型隐含的进程拓扑结构，只是各类具体并行机实现的消息传递函数库的方式不同，但用户无须知道这些细节。例如，在共享存储 SMP、DSM 并行机中，消息传递是通过共享数据缓存区来实现的；在 MPP 并行机中，消息传递是通过网络通信来实现的；在机群和星群并行机中，消息传递在 SMP、DSM 并行机内部是通过共享数据缓存区实现的，而在 SMP、DSM 并行机之间是通过网络通信来实现的。因此，无论哪种类型的具体并行机，呈现在消息传递并行程序设计用户面前的必然是图 1.14 所示的分布式存储并行机模型。

2. 标准消息传递界面 MPI

1994 年 6 月，全球工业、政府和科研应用部门联合推出消息传递并行编程环境的标准用户界面（MPI），它将消息传递并行编程环

境分解为两个部分，第一是构成该环境的所有消息传递函数的标准接口说明，它们是根据并行应用程序对消息传递功能的不同要求而制定的，不考虑该函数能否具体实现；第二是各并行机厂商提供的对这些函数的具体实现。这样，用户只需学习 MPI 库函数的标准接口，设计 MPI 并行程序，便可在支持 MPI 并行编程环境的具体并行机上执行该程序。通常意义下所说的 MPI 系统就是指所有这些具有标准接口说明的消息传递函数所构成的函数库。

在标准串行程序设计语言（C、Fortran、C++）的基础上，再加入实现进程间通信的 MPI 消息传递库函数，就构成了 MPI 并行程序设计所依赖的并行编程环境。MPI 吸收了众多消息传递系统的优点，例如 P4、PVM、Express、PARMACS 等，是目前国内外最流行的并行编程环境之一。当前，大量工业、科学与工程计算部门（例如气象、石油、地震、空气动力学、核物理等）的科研与工程软件已经移植到 MPI 平台。

相对其他并行编程环境，MPI 具有许多优点：

- 具有很好的可移植性，被当前所有并行环境支持；
- 具有很好的可扩展性，是目前高效率的大规模并行计算（数百个处理器）最可信赖的平台；
- 比其他消息传递系统好用；
- 有完备的异步通信功能；
- 有精确的定义，从而为并行软件的发展提供了必要的条件。

MPI 1.0 版于 1994 年推出，并同时获得了各并行机产商的具体实现；MPI 2.0 版于 1998 年 10 月推出，它在 1.0 版的基础上，增加了如下的消息传递功能：(1) 并行 I/O：允许多个进程同时读/写同一个文件；(2) 线程安全：允许 MPI 进程的多个线程执行，即支持

与 OpenMP 的混合并行编程；(3) 动态进程管理：允许并行应用程序在执行过程中，动态地增加和删除进程个数；(4) 单边通信：允许某个进程对其他进程的局部内存单元直接执行读/写访问，而不需要对方进程的显式干预；(5) 并行应用程序之间的动态互操作：允许各个 MPI 并行应用程序之间动态地建立和删除消息传递通信通道。

1.5 并行算法

并行算法是适合在并行机上实现的算法，一个好的并行算法应该具备充分发挥并行机潜在性能的能力。

例如，给定一个由偏微分方程描述的实际应用问题，采用合适的计算方法数值离散后，剩下的问题就是如何设计并行算法，编制并行程序，在并行机上求解离散格式，获得问题的数值近似解。

并行计算机的出现来源于实际应用程序中存在内在并行度这一基本事实，因此，应用问题中是否存在可挖掘的并行度是并行计算机应用的关键，而并行算法作为应用程序开发的基础，自然在并行计算机应用中具有举足轻重的地位。

1.5.1 并行算法的分类

目前，并行算法根据运算基本对象的不同可分为：

- **数值并行算法** 主要为数值计算方法而设计的并行算法；
- **非数值并行算法** 主要为符号运算而设计的并行算法，如图论算法、遗传算法等。

根据并行进程间相互执行顺序关系的不同可分为：

- **同步并行算法** 进程间由于运算执行顺序而必须相互等待的并行算法，如通常的向量算法、SIMD 算法、MIMD 并行机上进程间需要相互等待通信结果的算法等；

- **异步并行算法** 进程间执行相对独立，不需要相互等待的一种算法，通常针对消息传递 MIMD 并行机设计，其主要特征是在计算的整个过程中均不需要等待，而是根据最新消息决定进程的继续或终止；
- **独立并行算法** 进程间执行是完全独立的，计算的整个过程不需要任何通信。

根据各进程承担的计算任务粒度的不同，可分为：

- **细粒度并行算法** 通常指基于向量和循环级并行的算法；
- **中粒度并行算法** 通常指基于较大的循环级并行；
- **大粒度并行算法** 通常指基于子任务级并行的算法，例如通常的基于区域分解的并行算法，它们是当前并行算法设计的主流。

其实，并行算法的粒度是一个相对的概念。如果处理器的计算功能强大，则原来的粗粒度算法也可以被认为是细粒度算法。

此外，还存在根据应用问题的不同、通信方式的不同等进行分类的方式，这里不再一一给出。

1.5.2 并行算法的发展阶段

算法和应用程序的内在并行度导致了并行计算机的出现和发展，反过来，并行机的出现也很大程度上影响着并行算法的发展。实际上，并行机和并行算法的发展是相互相依存的，缺一不可。

并行算法的几个重要发展阶段可分为：

基于向量运算的并行算法设计阶段

该类算法是随着向量机 (如 CRAY–1，YH–1 等) 的出现而出现的，70 年代末和 80 年代初是其研究的顶峰时期，这方面最著名的成果为递归问题的向量化。我国在这方面有很好的研究和应用成果，并系统地总结在李晓梅等人的专著中 [3]。

基于多向量处理机的并行算法设计阶段

该类算法是随着多向量处理并行机（如 CRAY Y-MP，YH-2）的出现而出现的，其特点是既要考虑到多处理机间的任务级大粒度并行，又要考虑到单处理机上的向量级细粒度并行。该类算法在 80 年代初期和中期比较流行，目前日本一直坚持在这方面的研究和应用软件开发工作。

SIMD 类并行机上的并行算法设计阶段

由于该类并行机由大量功能简单的微处理器构成，因此通常要求并行算法能很好地在各处理器上分配局部数据，并组织数据在处理器间的合理流动，从而求解整个问题。典型代表为 CM-2、Systolic 阵列、Transputer 等类并行机上的并行算法。该类并行算法研究在 80 年代中期比较热门，但由于缺少一般性，过分依赖于机器，程序设计复杂，因此，随着 80 年代后期高性能并行机的出现，很快被淘汰。尽管如此，它们对帮助用户理解并行算法设计的技巧、并行机设计和通信原理是很有益处的。

MIMD 类并行机上的并行算法设计阶段

该类并行机的显著特点是处理器功能比较强大，能独立处理各类复杂运算，处理器间通过显式或隐式的通信来相互协同，共同求解同一问题。并且，处理器间单位数据的通信开销通常远远高于处理器的单位运算开销。因此，该类并行机通常需要任务级的大粒度并行。也就是说，需要设计大粒度并行算法来使用该类并行机。兴起于 80 年代初期，并一直为并行计算研究热门的求解偏微分方程的区域分解算法是这方面的典型代表。

现代并行算法设计

目前，并行算法的设计仍以 MIMD 类为主流，并要求具有可扩展性和可移植性。但随着微处理器和互联网络速度的发展，可扩

展高性能的获取必须要求并行算法设计兼顾两个发展方向：第一，可扩展、可移植的大粒度任务级并行；第二，在每个进程，组织便于发挥单机性能的合理数据结构、程序设计和通信方式。只有兼顾了这两个方面，才能真正发挥当前由高性能微处理器和互连网络构成的并行计算机的潜在高性能。

习 题

1. 给定 N 个数和 P 台处理机，假设处理机之间数据通信开销为 0，请设计一个算法，在最短的时间内完成这 N 个数相加。

2. 例 1.2 中，Gauss–Seidel 迭代具有并行度吗？请阐述理由。

3. 假设网络包含 $P = 2^N = M^3$ 个结点，请给出一维阵列（环）、二维网格（torus）、三维网格（torus）、超立方体、二叉树（叶结点个数为 P）的结点度、点对点延迟（以跨越的边的条数为单位）、折半宽度（以边的条数为单位）、网络直径。

4. 假设有 8 个结点，分别连接在 1Gbps 的快速以太网和 100Mbps 的 24 端口的 switch 上，请问任意两个结点间的平均带宽为多少，如果结点数增加一倍，则平均带宽又为多少。

5. 简单阐述当前并行计算机的多级存储结构，以及在程序设计中如何适应这种结构。

6. 阐述当前流行的并行计算机的分类，以及各类并行机体系结构的主要特征。

第 2 章　Linux 操作系统与程序开发环境

由于当前流行的并行计算机主要由结点上安装 Linux 操作系统的 PC Cluster 构成，所以了解 Linux 操作系统的一些基本知识有助于有效地使用它们。本章介绍 Linux 的安装和基本使用，以及在 Linux 操作系统上进行程序开发的简单知识。如果读者对于 Linux 操作系统已经比较了解，但是还不太熟悉该系统中程序设计方面的知识，可以直接进入本章 2.3 节。如果对于该系统下的程序设计也已经了解，则可以跳过本章，直接进入下一章的阅读。

本章的目的是帮助读者迅速掌握使用 Linux 操作系统进行工作的基本知识，内容基本来源于所讨论的软件及工具的文档。

可以认为，Linux 是一个小型的、适合于个人计算机的 UNIX 操作系统。这里假设读者对于计算机的基本使用已经相当熟悉，比如能够非常熟练地使用 Windows[1] 操作系统完成日常工作，并且具有程序设计的基本能力，能够读懂、编写 C 语言或者 Fortran 语言程序。本章将在此基础上讲述 Linux 操作系统的安装、使用以及在 Linux 操作系统下编程的基本知识。

2.1　Linux 安装与使用入门

2.1.1　Linux 系统的安装

为了在一台个人计算机上安装 Linux 系统，需要进行一些硬件方面的准备工作。首先需要一台个人计算机。对这台机器没有什么特殊的要求，但一般说来，最好不要过于陈旧或者过于先进，这样可以避免出现一些硬件支持方面的问题。Linux 系统对太过时的硬件

[1]Windows 是 Microsoft 公司的注册商标。

现在可能已经不再支持，而太新的硬件的支持有时也不太好。如果这台计算机上面已经安装了 Windows 或其他操作系统，安装 Linux 操作系统时可以保留已有的操作系统，但必须在硬盘上留出安装 Linux 所需要的空间。例如，可以删除 Windows 中的某个非系统分区，或是借助一些工具，如 PartitionMagic，调整 Windows 分区的大小来达到这一目的。建议至少保留 5GB 的空间给 Linux 系统用，其中 4 – 4.5GB 用于 Linux 的文件系统，0.5 – 1GB 作为交换分区。

另外一个准备是一套 Linux 系统的安装光盘。光盘安装是目前最简单的安装方式。Linux 系统还可以通过许多其他方式安装，如软盘、U 盘、网卡，甚至直接拷贝硬盘分区或文件系统，可以等熟悉了 Linux 以后，再去体验这些安装方式。Linux 系统的安装光盘可以通过商店或网上购买，也可以自己刻一套。目前市场上销售或是网上提供下载的 Linux 有很多版本，它们被称为不同的发行版 (distribution)，这样局面的出现是因为 Linux 是一个开放源码的操作系统，任何人都可以将 Linux 内核和一些应用软件集合到一起，做成一个发行版。目前常见的发行版有：RedHat (Fedora)，Mandrake，Debian，Suse 等，它们各有特点，安装都非常方便。如果所使用的计算机没有什么特别的硬件，应该和安装 Windows 操作系统没有什么显著差别，过程非常简单，界面也很友好，只需要跟随安装程序的提示进行操作就可以了。鉴于本章后续介绍主要基于由 RedHat (红帽子) 公司支持发行的 Fedora，建议最好安装 Fedora Core (`http://fedora.redhat.com/`)，以方便本章的阅读和练习。

安装时将第一张光盘放入光驱，接通计算机的电源，设置计算机从光盘上启动，就开始了安装过程。然后跟随提示进行，对于不太清楚的问题，一般回答默认选项即可。如果遇到麻烦，可以请教身边有经验的人或者在网络上寻求帮助。另外，市场上大量关于 Linux 系统安装的入门资料亦可参考。

在安装过程中，安装程序会要求建立一个普通用户的帐号并设置口令，该帐号将用于完成日常的工作。Linux 中有一个特殊用户：超级用户，用户名为 `root`，它是计算机的管理员帐户，在安装过程中也要为超级用户设置一个口令。一般来说，在使用 Linux 系统工作的时候，应该在一个普通用户的户头下进行操作，避免因为偶然的错误操作损伤系统。只有当进行系统软件安装或者修改系统配置文件的时候，才需要使用管理员的身份。

如果计算机配有网卡，安装程序会要求输入有关机器网络设置的信息，主要包括主机名、IP 地址、子网掩码、域名服务器等内容。如果对这方面内容不清楚的话，可以向网络管理员寻求帮助。

每个发行版都带有丰富的软件。如果为 Linux 系统留出的硬盘空间足够，建议将尽量多的软件都安装上，将来对 Linux 操作系统比较熟悉后，可以再卸载根本不用的软件包。现代 Linux 的各种发布版本都提供方便的工具，用于安装新的软件包或卸载一个已安装的软件包。

安装程序结束后，重新启动计算机，安装过程就完成了。Linux 安装程序通常会替换掉硬盘引导区中原有的引导程序，代之以一个支持 Linux 启动的引导程序，如 LILO 或 Grub。新引导程序允许用户启动时选择安装在计算机上的不同操作系统，如 Windows、Linux 等。选择启动 Linux，便可以进入到刚刚安装的 Linux 系统中。

Linux 系统默认的文件系统格式是 `ext2`，或它的后继 `ext3`。和其他操作系统一样，Linux 的文件系统采用的也是树形结构，树的根就是根目录 (root directory)，用一个正斜杠 “/” 表示，树的叶子是文件，中间结点则是目录。要注意的是，Linux 的路径名中用正斜杠 “/” 分隔目录名，而 Windows 中用的则是反斜杠 “\”。Linux 系统的根目录下通常有下面一些子目录：

- `/usr`：其中安装着主要的系统文件和软件；

- /home：普通用户的家目录所在的位置，可以选择将它放在一个不同的硬盘分区上，这样在重新安装系统时可以保留用户的文件；
- /etc：系统配置文件；
- /dev：设备文件，主要用于驱动各类硬件；
- /proc：该目录中的内容不是硬盘上的普通文件，而是 Linux 内核的一些接口，通过它可以在运行时获取、改变系统内核的许多参数；
- /mnt：外挂设备的挂接点；
- /root：超级用户的家目录；
- /sbin：包含一些主要供超级用户用的可执行程序；
- /bin：一些最常用的可执行程序；
- /boot 和 /initrd：系统启动用的文件；
- /lib：库文件；
- /tmp：用于创建临时文件或目录；
- 其他目录：/opt，/misc，/var：这些目录普通用户基本上不用；需要指出的是系统日志文件在目录 /var/log 下面。

/usr 是所有目录中最复杂的一个，因为除了最基本的系统软件外，其他软件都安装在 /usr 目录下 (有些发行版也会把部分软件安装在 /opt 或 /usr/local 下面)。其中包含的主要目录有：

- bin 和 sbin：包含可执行文件，其中 sbin 中的文件主要是系统管理员用的；

- `lib`：主要的库文件的位置，它常常是最占空间的；
- `include`：头文件的位置，编写程序时需要包含的头文件大都在这个目录中；
- `src`：内核和软件的源代码的位置；
- `local`：这是安装一些外来软件的地方，有些软件也可能选择安装在 `/opt`；
- `X11R6`：和 X 有关的文件，包括可执行命令、头文件、库文件等等。X 是 Linux/UNIX 的图形界面软件，图形用户界面的软件通常都在该目录中；
- `share/man` 和 `share/doc`：帮助文件所在的位置；

如同超级用户的家目录在 `/root` 一样，每个普通用户也有一个家目录。通常，普通用户的家目录放在 `/home` 目录下，名字和用户名相同。普通用户能够修改的主要内容局限在自己的家目录中。例如，作者的用户名是 `rli`，家目录的位置就是 `/home/rli`。

2.1.2 基本使用与管理

启动进入 Linux 系统后，首先看到的是一个登录界面，通常是图形登录界面。登录界面的作用就是确认使用者的身份。在用户名的位置输入安装 Linux 系统时为自己创建的用户名，紧接着根据提示输入该用户的口令，便可以登录到系统中。

虽然 Linux 系统提供了与 Windows 操作系统类似的图形界面，但对于编程者来说，学会直接使用 Linux 的命令行操作依然十分重要，因为许多操作，特别重复性的操作，通过命令行完成往往比拖拉、点击鼠标更加快速、便捷。如果用户是通过文字界面登录的，则直接面对的就是一个可以输入命令的 shell。如果是通过图形界面登

录的，则需要首先启动一个 *shell* 窗口，例如点击系统菜单中的“系统工具 — 终端”，然后才能在窗口中输入命令。*Shell* 字面上的意思是“壳”，它提供用户与 Linux 系统间的交互，执行用户输入的命令，并将命令执行结果返回给用户。Shell 本身也是一个应用程序，有很多不同类型的 shell 可供选择，比较常用的有 Bourne shell、C shell、Korn shell、TC shell 和 Zsh shell 等。在 Linux 中，最常用的是 Bash，其全称是 Bourne–Again SHell，它是 Bourne shell 的一个扩展版本。Shell 窗口也叫模拟终端，其中运行着一个交互式 shell，类似于 Windows 操作系统中的“命令窗口”。Shell 程序运行时会给出一个提示符，等待用户输入命令。Shell 提示符的形式是可以配置的，如下面的样子：

```
[rli@arena tmp]$ _
```

其中 `rli` 是用户名，`arena` 是主机名，`tmp` 是当前所在的目录。字符 `$` 也是提示符的一部分，下划线代表光标。用户每次在 shell 提示符前输入一条命令，shell 便会解释执行该命令。Shell 接受的命令一般形式为：

```
命令名 [可选项1 可选项2 ...] 参数1 参数2 ...
```

最前面是命令名，然后是一系列的可选项和参数。可选项和参数的写法是由所执行的命令决定的，有些命令对于可选项和参数的写法要求很严格，有些则比较自由。本章在介绍命令格式时，按照 Linux 下的通常习惯，用方括号来表示可以省略的部分。

Linux 最基本的系统管理工作包括系统配置、用户帐户管理、安装和卸载软件包。为了进行系统管理工作，首先需要获得管理员的身份。普通用户可以用命令 `su` 转换成为超级用户，为此，只需在 shell 提示符前输入命令：

```
[rli@arena tmp]$ su -
Password:
```

```
[root@arena tmp]#
```

系统会提示要求输入口令。输入超级用户的口令后，系统提示符会由 “$” 变成 “#”，后者是默认的超级用户提示符。在 su 命令后面加上可选项 “-” 的主要作用是为了得到超级用户的路径，以便直接使用 /sbin 和 /usr/sbin 中的命令。

1. 帐号管理

Linux 系统最重要的配置文件之一是 /etc/passwd，它记载着系统中所有用户的信息。在传统的 UNIX 系统中，该文件存储用户的用户名、用户号、口令等信息。现代系统中出于安全考虑，把用户口令单独存储在另外一个文件 /etc/shadow 中，所以 /etc/passwd 中实际上并没有存储口令。进行用户帐户管理，事实上就是修改这两个文件。对初学者而言，可以在进行帐户管理操作时，比较这两个文件的内容在命令执行前后的变化来了解命令的确切作用。Linux 中添加新用户的命令为 useradd，它的简单语法为

```
useradd [-d 家目录] [-g 组名] [-s shell] 用户名
```

由于后续介绍中将采用上面的格式来介绍许多 Linux 命令的语法，这里稍稍做些解释。在上面的格式中，命令名为 useradd，写在方括号中的是可选的参数项，最后有一个必须的参数用户名。在这些可选参数中，家目录指定新用户的家目录 (默认值为 /home/用户名)，组名指定新用户所属的用户组 (稍后解释)，shell 指定用户的默认 shell，即用户登录时自动获得的 shell (默认值为 /bin/bash)。例如，下面的命令

```
[root@arena tmp]# useradd -g users -s /bin/bash aaa
```

添加一个名为 aaa 的新用户，该用户属于用户组 users，shell 为 /bin/bash。上述命令的执行会使得 /etc/passwd 文件中增加下面一行内容

```
aaa:x:502:100::/home/aaa:/bin/bash
```

文件 /etc/passwd 中每行定义一个用户，由用冒号分割开的几部分构成，其含义依次为用户名 (uname)、用户口令 (password，这里是一个 x，真实的口令存储在文件 /etc/shadow 中)、用户号 (uid)、组号 (gid)、用户信息 (如真实姓名、电话、E-mail 等，本例中用户信息是空的)、家目录和 shell。

删除一个用户的命令为

```
userdel [-r] 用户名
```

其中可选项 "-r" 的含义是要求在删除用户的同时，将该用户的家目录、邮箱等一并删除。

修改用户口令的命令是 passwd，语法如下：

```
[root@arena tmp]# passwd [用户名]
```

超级用户可以修改任何用户的口令，普通用户只能修改自己的口令，并且会被要求先输入当前口令。输入新的口令时，会被要求输入两遍以确保无误。

命令 usermod 可用来修改一个帐号的属性。它的命令行参数和 useradd 基本一样。如：

```
[root@arena tmp]# usermod -s /bin/csh aaa
```

会将用户 aaa 的默认 shell 改成 C shell (/bin/csh)。

为了便于管理，Linux 系统中的用户被分成一个个的用户组，每个用户可以同时属于多个用户组，/etc/passwd 中给出的用户组是该用户的默认用户组。例如，用户 aaa 的默认用户组是 users。用户组的信息存储在文件 /etc/group 中。对于用户组的管理也有相应的命令，如 groupadd、groupdel 等。为了将一个用户加入某个组或者从某个组去掉，可以使用命令 gpasswd。请看下面的例子：

```
[root@arena tmp]# groupadd visitor
[root@arena tmp]# gpasswd -a aaa visitor
[root@arena tmp]# gpasswd -d aaa visitor
[root@arena tmp]# groupdel visitor
```

上述命令中，首先创建了一个叫做 visitor 的用户组，然后将用户 aaa 加入到该组。这样，用户 aaa 便同时属于 users 组和 visitor 组。随后两行命令分别将 aaa 从用户组 visitor 中去除，和将用户组 visitor 删除。当然，上述操作也可以通过直接编辑 /etc/group 文件来实现，该文件中每行内容具有如下格式：

```
users:x:100:rli,aaa
```

其中每列的含义依次为：组名 (gname)，组口令 (group password)，组号 (gid)，最后是属于该用户组的用户名列表。

读者在使用 Linux 系统的过程中或许已经注意到，系统有时候会给出一个提示，说用户的口令已经过期了。口令的期限可以通过命令 chage 来查询或设置，它的语法为：

```
chage [-m 最小天数] [-M 最大天数] [-d 上次修改日期] [-I 非活跃帐号锁定的天数]
     [-E 帐号失效期] [-W 口令失效前警告的天数] 用户名
```

例如命令

```
[rli@arena rli]$ chage -l rli
```

列出关于用户 rli 的口令期限的信息：

```
Minimum:        -1
Maximum:        99999
Warning:        -1
Inactive:       -1
Last Change:             4月 15, 2003
Password Expires:       Never
Password Inactive:      Never
Account Expires:        Never
```

2. 软件包管理

系统管理的另外一项重要工作是安装和卸载软件。Fedora 提供了一个叫做 RPM 的软件，专门用来管理系统中的软件包。RPM 软件的功能非常强大，参数非常繁杂，最基本的使用包括安装软件包、查询软件包、卸载软件包等。Fedora 中，所有系统软件都可以用 RPM 来管理。RPM 是 “RedHat Package Manager” 的缩写，它是由红帽子公司开发的一个软件包格式，以 “.rpm” 作为文件的扩展名。管理 RPM 软件包的程序名为 rpm。

下面通过例子说明 rpm 程序的使用。假设从网上下载了一个名为 gaim-0.64-1.i386.rpm 的软件包，它是在 Linux 下进行即时通信的软件，支持很多即时短信协议，其中 gaim 是软件包的名称，后面的 0.64-1 是软件的版本，i386 表示这是适合于 IntelIntel 是 Intel 公司的注册商标。 i386 型结构的机器的软件包。安装软件包的命令如下：

```
[rli@arena rli]$ su
Password: ************
[root@arena rli]$ rpm -ivh gaim-0.64-1.i386.rpm
warning: gaim-0.64-1.i386.rpm: V3 DSA signature: NOKEY, key ID 883c1c14
Preparing...                ########################################### [100%]
   1:gaim                   ########################################### [100%]
[root@arena rli]$
```

上述命令中，首先用 su 命令转换成超级用户身份，然后再进行操作。在 rpm 命令中使用了三个选项，“-i”、“-v” 和 “-h”。许多 Linux 命令，包括 rpm 命令，允许将多个选项写在同一个 “-” 后面，因此这里将它们写成了 “-ivh” 的形式。选项 “-i” 表示要求安装 (install) 软件包，“-v” 表示要求显示安装过程中的信息，而 “-h” 表示要求打印出一串 “#” 来显示安装的进度。安装软件包时，除了可以用选项 “-i” 之外，也可以用选项 “-U” (意为 upgrade，即升级)，它与前

者的区别是只有当系统中尚未安装该软件包，或者待安装的软件包版本比系统中已安装的新时才实际执行安装操作。还有一个非常有用的选项，“-F” (意为 freshen)，可用于软件包的升级，它仅当系统中已经安装有该软件包，并且待安装的软件包版本比系统中已安装的新时才执行安装操作。例如，假如从网上下载了一批软件包的升级版本放在一个目录中，则可以在该目录中用命令

```
rpm -Fvh *.rpm
```

来更新系统中的软件，它只升级系统中已安装的软件，而不会装入新的软件。

rpm 命令中查询软件包信息的选项是 “-q” (通常结合其他选项一起使用)，删除软件包的选项是 “-e”。下面归纳的是一些常见的用法：

- rpm -e 软件包名 [软件包名...]
  删除指定的软件包。可以指定多个软件包名，中间用空格隔开。注意这里 “软件包名” 指的是软件包的名称 (如 gaim)，可以包含版本号 (如 gaim-0.64-1)，而不是软件包的文件名，即不能包括 .rpm 扩展。该命令可以一次删除多个软件包。通常需要将相互间有依赖关系的软件包同时删除 (除非使用了 “--nodeps” 选项)。

- rpm -q 软件包名
  显示指定软件包的版本信息。

- rpm -qi 软件包名
  显示指定软件包的详细描述。

- rpm -qa
  列出系统中已安装的所有软件包。

- `rpm -ql` 软件包名
  列出指定软件包所包含的所有文件。

- `rpm -qf` 文件名
  显示指定文件 (必须用绝对路径) 所属的软件包名。

上述 "`-q*`" 形式的选项中，可以加入 "`-p`" 选项来查询一个尚未安装的软件包。例如，命令

```
rpm -qlp gaim-0.64-1.i386.rpm
```

可以在安装 `gaim` 软件包之前列出其中所包含的文件。

上面介绍的命令都是对编译好的软件包 (二进制包) 进行操作。RPM 格式的软件包分为源码包和二进制包两种，前者包含该软件的源代码，后者是由前者通过编译产生的。安装软件时必须使用二制包。所有软件包文件名均以 "`.xxx.rpm`" 结尾，其中 "`xxx`" 代表软件包的类型，常见的有以下几种：

- `.src.rpm`: 源码包
- `.noarch.rpm`: 适合所有平台的二进制包
- `.i386.rpm`: 适合 Intel x86 的二进制包，要求 386 以上的 CPU
- `.i486.rpm`: 适合 Intel x86 的二进制包，要求 486 以上的 CPU
- `.i586.rpm`: 适合 Intel x86 的二进制包，要求 586 以上的 CPU
- `.i686.rpm`: 适合 Intel x86 的二进制包，要求 686 (Pentium II，Xeon 等) 以上的 CPU
- `.ia64.rpm`: 适合 Intel Itanium 的二进制包
- `.x86_64.rpm`: 适合 Intel EM64T (64 位至强) 和 AMD Opteron 的二进制包

- .ppc.rpm：适合 POWER PC 的二进制包

选择二进制包时，应该选择与自己的平台兼容的、对应于尽可能高级别 CPU 的包，以达到最佳性能。如果下载的软件包是源码包，即扩展为 .src.rpm，则需要先对它进行编译，得到二进制包，然后再进行安装。编译源码包用 rpmbuild 命令，该命令在 rpm-build 包中。以 gaim 为例：

```
rpmbuild --rebuild gaim-0.64-1.src.rpm
```

如果编译正常完成，假设用户的平台是 x86，则会在目录

/usr/src/redhat/RPMS/i386/

中生成相应的二进制包。编译软件包时，编译所需要的其他软件包，特别是一些 -devel 包，必须事先已经安装，否则 rpmbuild 会报错，提示缺少编译时所需要的某些软件包。

有关 RPM 更详细的使用方法以及如何自行制作 RPM 包可参看网上及出版的相关资料，推荐参考 http://www.rpm.org/max-rpm/。

用 rpm 安装或升级软件包需要自己去寻找要安装或升级的软件包文件。Fedora 发行版中提供了一个名为 yum 的程序，利用它可以方便地自动从网上安装或升级指定的软件包，但通常仅限于由 Fedora 发行的系统软件。例如：

```
yum -y install ypserv yptools
yum -y update gaim
```

上述第一条命令安装 ypserv 和 yptools 包，而第二条命令则升级 gaim 包 (如果网上的版本比本机安装的版本新的话)。当 “update” 中不指定软件包时表示升级本机上已安装的所有软件包。

有时，可能拿到的软件没有 RPM 的格式，而直接是源代码。通常，这些源代码是一个压缩的归档文件形式。在 Linux 下，人们比较喜欢使用 gzip 或者 bzip2 进行文件的压缩，这两种压缩文件的

扩展名分别是 “.gz” 和 “.bz2”。同时，在压缩以前，一般会使用 tar 工具对源程序的整个目录进行归档，这类归档文件的扩展名为 “.tar”。这样，一个软件包源码的整个名字通常是 “xxxx.tar.gz” 或者 “xxxx.tar.bz2” 的形式，有时候也可能会将 “.tar.gz” 合并为 “.tgz”。对于这样的软件包，其编译、安装步骤大都是类似的。以 Kile 为例，它是 Linux 下编写 TEX 文档的一个前端软件 (事实上，Kile 已经是 Fedora Core 中的标准包，可以在安装 Linux 系统时直接选择或用 yum 安装)。假设源码文件名为 kile-1.5.tar.gz，放在临时目录 /tmp 下。

- 解开源码包：

```
[rli@arena tmp]$ tar xzvf kile-1.5.tar.gz
```

这里使用了 tar 命令的几个选项，“x” 表示展开档案，“v” 表示打印出操作时的信息，“f” 表示对紧随其后的文件进行操作，“z” 表示是 gzip 格式的压缩文件 (如果是 bzip2 的压缩文件则需用 “j”)。下面的命令也可以做到同样的事情，它通过稍后要介绍的管道进行操作：

```
[rli@arena tmp]$ zcat kile-1.5.tar.gz | tar xvf -
```

软件包解开后，会在当前目录下产生一个子目录 kile-1.5。进入该子目录，可以看到很多文件，包括：configure，configure.in，configure.ac，Makefile.in，Makefile.am 等等，它们用于 Kile 软件的配置与编译。目录下还有帮助用户了解和安装该软件的一些文档，比如 README，INSTALL，NEWS 等，它们都是普通的文本文件，其含义一目了然，编译、安装前最好先仔细阅读这些文档。

- 运行 configure 脚本进行系统检测，产生 Makefile 文件：

```
[rli@arena kile-1.5]$ ./configure --prefix=/path/to/install
```

其中 --prefix 选项指定软件的安装路径，默认的安装路径一般是 /usr/local。configure 运行完毕后，会产生一个新的 Makefile 文件。

- 编译软件：

```
[rli@arena kile-1.5]$ make
```

make 是 Linux 系统中的命令，它根据当前目录下的 Makefile 文件完成复杂的编译和链接软件包的工作。本章后面将会对 make 及相关工具进行专门介绍。

- 安装软件：

```
[rli@arena kile-1.5]$ make install
```

要注意的是，运行安装命令时要求对安装目录有写的权限，必要时需先用 su 命令转换成超级用户再执行上面的命令。

- 卸载软件包：某些软件，如这里的 Kile，在 Makefile 中提供了卸载功能，可用下面的命令卸载安装的软件

```
[rli@arena kile-1.5]$ make uninstall
```

2.2 Linux 基本命令和概念

2.2.1 一些基本命令

下面介绍 Linux 系统的一些基本命令。在介绍这些命令的同时，将顺便介绍一些 Linux 操作系统的基本概念。Linux 系统中提供了非常丰富的命令用于完成各种任务，同时也提供许多途径获取信息，来了解这些命令的功能与使用方法。

1. pwd

该命令显示当前工作目录。任何时候，用户都有一个当前目录的概念。刚登录进系统的时候，当前目录是用户的家目录。许多命令的执行结果会和当前目录有关系，所以知道当前目录非常重要。

2. cd

改变当前目录。使用该命令可以在整个文件系统中穿梭，在不同的位置做不同事情。请看下面的例子：

```
[rli@arena rli]$ cd /usr/share/texmf/tex/latex
[rli@arena latex]$ pwd
/usr/share/texmf/tex/latex
[rli@arena latex]$ cd /usr/local
[rli@arena local]$ pwd
/usr/local
[rli@arena local]$ cd -
[rli@arena latex]$ pwd
/usr/share/texmf/tex/latex
[rli@arena latex]$ cd ~/src/appl
[rli@arena appl]$ pwd
/home/rli/src/appl
[rli@arena appl]$ cd
[rli@arena rli]$ pwd
/home/rli
[rli@arena rli]$ cd ..
[rli@arena home]$ pwd
/home
```

上例中，“cd -” 回到上一次所在的目录。“~” 表示用户的家目录。单独一个 “cd” 也是回到家目录。“..” 表示当前目录的上一级目录。“.” 指的就是当前目录。

3. `ls`

`ls` 来自于英文单词 list。默认情况下 `ls` 会列出当前目录下的文件和子目录名，请看下面的例子：

```
[rli@arena rli]$ ls                 # 列出当前目录下的文件和目录
bak  bin  data  doc  include  lib  src  tmp
```

进一步，可以用 `-l` 选项来让 `ls` 显示更多的信息：

```
[rli@arena rli]$ ls -l              # 列出当前目录下的文件和目录的细节信息
total 268
drwxr-xr-x    5 rli      users        8192 11月 27 20:22 bak
drwxrwxr-x    2 rli      users        4096 11月  2 14:17 bin
drwxrwxr-x    4 rli      users        4096 10月  5 13:52 data
drwxr-xr-x   13 rli      users        4096 10月 17 10:19 doc
drwxrwxr-x    2 rli      users        4096  4月 15  2003 include
drwxrwxr-x    2 rli      users        4096  4月 20  2003 lib
drwxr-xr-x   11 rli      users        4096 11月 21 13:27 src
lrwxrwxrwx    1 rli      users           4  4月 22  2003 tmp -> /tmp
```

上面的输出中包含了许多重要的信息与概念，下面详细解释一下。在 Linux 系统中，所有的东西都被处理成一个文件，包括硬盘上真正的文件、目录、硬件设备，等等。上面的输出中，第一列的第一个字符给出文件的类型，这里，除了 `tmp` 是 “`l`” 以外，其余都是 “`d`”。`d` 表示目录，而 `l` 表示符号链接。除了 `d` 和 `l` 外，文件类型还可以是 `-`、`s`、`b`、`c` 和 `p`，分别表示普通文件、具有 SUID 属性的文件、块设备、字符设备和管道。

Linux 是一个多用户的操作系统，系统中的每个文件都有属主 (owner)，即拥有该文件的用户。大部分系统文件的属主是超级用户，普通用户无权修改它们。每个文件除了有属主以外，还有一个用户组的属性。默认情况下，文件的组就是属主的默认组。这样，对于一个文件来说，系统上的所有用户被分成三类：第一类是该文件的属主，第二类是该文件的用户组中的用户，第三类是所有其他用户。在上

面的输出中，第三列是属主 (rli)，第四列是文件的用户组 (users)。对一个文件的访问权限描述，也相应分成三个部分，在上面的输出中，第一列除第一个字符外的其余 9 个字符用于描述不同用户对文件的访问权限，它们每三个字符构成一组，共三组，分别表示属主、用户组成员和其他用户对该文件的访问权限。每组三个字符依次是 r (read, 表示读的权限)、w (write, 表示写的权限) 和 x (execute, 表示执行权限)，如果具有该权限，则显示相应的字符，如果不具有该权限，则显示一个短横线 -。比如上面输出行，

```
drwxrwxr-x    4 rli      users        4096 10月  5 13:52 data
```

表示对于目录 data，用户 rli 有读、写和执行权限，users 组中的用户有读、写和执行权限，而其他用户只有读和执行权限，没有写权限。对于一个目录而言，具有执行权限意味着可以进入该目录。

上面的输出结果中第一行表示目录中所有文件所占的空间。其余行的各列依次是权限位、硬链接数、属主名、组名、文件大小、最后修改时间和文件名。ls 还有很多其他用法，下面是一些例子：

```
[rli@arena rli]$ ls /usr              # 列出目录 /usr 下的文件和目录
bin   doc  games            include   lib      local  sbin   src  X11
dict  etc  i486-linux-libc5  kerberos  libexec  man    share  tmp  X11R6
```

```
[rli@arena rli]$ ls -l data/*.dat     # 列出目录 data 下 *.dat 文件的细节信息
-rw-r--r--    1 rli      users       15415  5月  8  2003 data/u_h_comform.dat
-rw-r--r--    1 rli      users      238736  8月 16 22:27 data/u_h.dat
```

可以看到，可以在 ls 命令后面指定要列出的文件或目录名。与 Windows 或 MSDOS 中类似，可以在文件或目录名中使用通配符 “*” 和 “?”，前者匹配任意字符串，后者匹配任意单个字符。在 Linux 系统中，通配符的使用比在 Windows 或 MSDOS 中更加灵活，例如，可以用 “ls -l */*/*/*.dat” 列出所有第三级子目录下以 “.dat” 为

扩展的文件。

4. echo

显示命令行参数。该命令将命令行参数原封不动地显示出来。如：

```
[rli@arena rli]$ echo ABC
ABC
[rli@arena rli]$ _
```

这条命令看似没用，实际上非常有用。与输出重定向结合，可用来将特定内容写入文件；在 shell 文件中，可用来输出信息；利用管道，可以为其他命令提供输入。后面的例子中经常会用到它。

5. file

确定一个文件大致的类型与性质。如

```
[rli@arena rli]$ file src                  # 看看 src 是什么
src: directory                             # 哦，是一个目录
[rli@arena rli]$ file /dev/pts/0           # 那么这个是什么？
/dev/pts/0: character special (136/0)      # 这是一个特殊字符设备
[rli@arena rli]$ file /etc/rc.d/rc.sysinit # 这个文件是 Bash 脚本文件
/etc/rc.d/rc.sysinit: Bourne-Again Shell script text executable
```

6. man

获取在线帮助。该命令取自英文单词 “manual” (手册) 的开头，是获取命令帮助最为简单有效的手段。当对任何一条命令、函数、甚至关键词有疑问的时候，可以试试用 man 命令加上相应命令、函数的名称或关键词，看是否有相关的帮助信息。例如，“man man” 给出 man 命令本身的帮助信息：

```
[rli@arena rli]$ man man
man(1)                                                    man(1)
```

```
NAME
       man - format and display the on-line manual pages
       manpath - determine user's search path for man pages

SYNOPSIS
       man  [-acdfFhkKtwW]  [--path]  [-m system] [-p string] [-C config_file]
       [-M pathlist] [-P pager] [-S section_list] [section] name ...

DESCRIPTION
       man formats and displays the on-line manual pages.  If you specify sec-
       tion,  man  only looks in that section of the manual.  name is normally
       the name of the manual page, which is typically the name of a  command,
       function,  or  file.  However,  if  name contains a slash (/) then man
       interprets it as a file specification, so that you can do  man  ./foo.5
       or even man /cd/foo/bar.1.gz.

       See  below  for  a  description  of where man looks for the manual page
       files.

OPTIONS
       -C  config_file
... ...
```

```
[rli@arena rli]$ man ls                    # 获得 ls 的帮助
```

阅读帮助信息时，可以用空格键向下翻页，用 Ctrl-B 向上翻页，用字母 “q” 退出帮助信息的阅读。当终端类型配置正确时，也可以用 <PgUp>、<PgDn> 以及箭头键来上下翻页或滚屏。

Linux 的各种在线帮助被分成了不同类别，参看表 2.1。例如，可以用 “man 3 scanf” 来查找 C 的库函数 scanf 的帮助。如果不知道一个关键字的确切类别，但是又有多个类别包含该关键字，可以考虑使用 “-a” 选项，它使得 man 列出所有类别中关于该关键字的帮

表 2.1 Linux 常用在线帮助的分类

级别	所包含的帮助类型
1	可执行文件和 shell 命令
2	系统调用
3	库函数
4	/dev 中的设备
5	配置文件
6	游戏
7	宏包
8	管理员命令
9	非标准的内核进程
n	其他

助信息。例如，类别 “n”、“1” 和 “3” 中均有关于 “read” 的帮助信息，可以用 “man -a read” 将所有关于 read 的帮助信息找出来。

在 man 打印出来的帮助信息中，开始是命令或关键字的名称和一个简单描述 (NAME)，紧跟着是它的语法 (SYNOPSIS)，然后是其功能的详细介绍 (DESCRIPTION)，最后是命令行参数和选项的描述 (OPTIONS)。常常还有一个叫做 SEE ALSO 的部分，列出和该命令相关的其他命令或关键字。

在阅读本章内容时，应该用 man 命令阅读遇到的每条命令的在线帮助。在线帮助中的描述清晰准确，并且非常完整，通常是关于该命令最权威的资料。man 不但提供各种命令的帮助信息，还有所有的库函数的开发文档，系统的结构说明等等内容。

Linux 系统中除了传统的 man 格式的在线帮助之外，还有 info 格式的帮助文档。info 格式的帮助文档通常比 man 更详尽。下面是用 info ls 命令浏览 ls 手册时的屏幕抓图：

```
File: coreutils.info,  Node: ls invocation,  Next: dir invocation,  Up: Directo\
```

```
ry listing

`ls': List directory contents
============================

   The `ls' program lists information about files (of any type,
including directories).  Options and file arguments can be intermixed
arbitrarily, as usual.

   For non-option command-line arguments that are directories, by
default `ls' lists the contents of directories, not recursively, and
omitting files with names beginning with `.'.  For other non-option
arguments, by default `ls' lists just the file name.  If no non-option
argument is specified, `ls' operates on the current directory, acting
--zz-Info: (coreutils.info.gz)ls invocation, 48 lines --Top--------------------
*** Footnotes appearing in the node ``ls invocation'' ***

   (1) If you use a non-POSIX locale (e.g., by setting `LC_ALL' to
`en_US'), then `ls' may produce output that is sorted differently than
you're accustomed to.  In that case, set the `LC_ALL' environment
variable to `C'.

-----Info: *Footnotes*, 7 lines --All-----------------------------------------
Welcome to Info version 4.2. Type C-h for help, m for menu item.
```

在使用 `info` 的时候，可以用 `p`、`u`、`f` 等键以类似于浏览网页的方式进行阅读，用 `?` 键可以得到关于这些快捷键的帮助。很多情况下，也可以用上下翻页、箭头等键浏览。

7. `mkdir`

创建一个新目录

```
[rli@arena rli]$ ls
bak  bin  data  doc  include  lib  src  tmp
[rli@arena rli]$ mkdir newdir                    # 建立一个名为 newdir 的目录
```

```
[rli@arena rli]$ ls
bak  bin  data  doc  include  lib  newdir  src  tmp
```

8. rm

删除文件或目录

```
[rli@arena rli]$ touch newfile                    # 建立一个新文件 newfile
[rli@arena rli]$ ls
bak  bin  data  doc  include  lib  newdir  newfile  src  tmp
[rli@arena rli]$ rm newfile                       # 删除文件 newfile
[rli@arena rli]$ ls
bak  bin  data  doc  include  lib  newdir  src  tmp
[rli@arena rli]$ rm -rf newdir                    # 删除文件 newdir
[rli@arena rli]$ ls
bak  bin  data  doc  include  lib  src  tmp
```

上面在 rm 命令的后面加了 -rf 参数，这是一种危险的使用方式。参数 “-r” 表示按照目录树递归操作，“-f” 表示不做任何提示强制删除目录下的任何文件及子目录。进行这类操作时一定要非常小心，并且避免使用通配符，以免误删有用的文件或目录。

9. cp

拷贝目录或文件。它可以为一个文件建立一份新的拷贝，或者将一个或者多个文件拷贝到一个目标目录中。如果希望整个拷贝一个目录的话，可以使用 -R、-r 或 -a 选项。对许多命令而言，-R 选项表示按照目录树进行递归操作。

10. mv

移动目录或者文件。它将文件或者目录移动成为一个新的文件或者目录，或移动到另外一个目录中。请看一些例子：

```
[rli@arena tmp]$ cp /home/rli/src/test.c .
```

```
[rli@arena tmp]$ ls -l
total 4
-rw-r--r--    1 rli      users         449 10月  7 15:22 test.c
[rli@arena tmp]$ mkdir test_dir
[rli@arena tmp]$ mv test.c test_dir
[rli@arena tmp]$ ls -l test_dir
total 4
-rw-r--r--    1 rli      users         449 10月  7 15:22 test.c
[rli@arena tmp]$ cp -R test_dir ~
[rli@arena tmp]$ ls -l ~/test*
total 4
-rw-r--r--    1 rli      users         449 10月  7 15:24 test.c
```

Linux 中没有类似于 Windows 或 MSDOS 中的 `rename` 命令，后者的功能可以通过 `mv` 命令来实现。

11. `ln`

为文件或目录建立链接 (link)。Linux 系统中，一个文件或目录的链接对应于 Windows 中的“快捷方式”，本质上相当于一个文件或目录具有多个名字。链接分硬链接和软链接 (也叫符号链接) 两种。合理使用链接往往可以方便许多系统及文件的管理工作。默认情况下，`ln` 命令创建硬链接。如果希望创建符号链接，则应该加上“`-s`”选项。一个文件的硬链接必须与原来的文件位于同一文件系统内 (硬盘的同一个分区)，而符号链接则无此限制。下面是创建一个目录的符号链接的例子：

```
[rli@arena rli]$ ln -s ../../usr/include inc
[rli@arena rli]$ ls -l inc
lrwxrwxrwx  1 root root 7 11月 23 15:45 inc -> ../../usr/include
```

注意，如果一个符号链接的对象包含的是相对路径 (即路径名不是以“/”开头)，则该对象的位置是相对于符号链接所在的目录而言的。例如在上例中，假设当前目录是 `/home/rli`，则 `inc` 实际所指的

对象为 /home/rli/../../usr/include，即 /usr/include。

12. touch

改变文件的最后修改时间。该命令将文件的最后修改时间设置成为现在，或者任意 (通过选项) 指定的时间。该命令也常被用来创建一个新的空文件。

13. cat，more，less，lv，head，tail

查看文件内容。这些命令用不同方式显示文本文件的内容。cat 将输入文件的内容连接起来输出到标准输出；more 将输入文件分屏交互地显示出来；less 是对 more 功能的增强；head 显示输入文件的头十行或指定数目的行；tail 显示输入文件的最后十行或指定数目的行。这些命令支持一个或多个输入文件名做为参数。当不给出输入文件名时，它们会从标准输入，默认情况下是用户的键盘，读入内容，这种情况下需要用 Ctrl-D 来结束输入：

```
[rli@arena rli]$ cat > tmp.txt
This is a tmp file.
I use it to check how cat works.
(Ctrl-D)
[rli@arena rli]$ cat tmp.txt
This is a tmp file.
I use it to check how cat works.
```

上面第一个命令中使用了一个 “>” 符号，这叫做重定向，它将本来应该输出到屏幕的内容转到了文件 “tmp.txt” 中。后面会详细介绍有关重定向的内容。

用 more 显示文件时，文件的内容会一页一页的显示，按回车键屏幕下滚一行，按空格键下翻一页，按 Ctrl-B 可以回翻一页。less 比 more 更加方便，可以使用上下箭头、<PgDn>、<PgUp> 等键，而且能够搜索文档 (按 “/” 键加搜索内容)。要注意的是，less 不能正

确显示汉字，如果想显示包含汉字的内容的话，可以用程序 lv 替代 less。head 显示文件开头的内容，默认情况下显示 10 行。tail 则显示文件尾部的内容，默认也是 10 行。命令 tail 还有一个非常有用的选项 -f，它会自动检测文件的改动，实时显示文件新增加的内容。

14. chmod，chgrp，chown

修改文件属性。chmod 修改文件的权限位，chgrp 修改文件所属的组，chown 则修改文件的属主。在用 ls -l 列出文件的细节信息时可以看到每个文件的属性，包括属主、组以及权限。这几个命令就是对这些属性进行修改的工具。下面是 chmod 命令的例子：

```
[rli@arena rli]$ ls -l newfile
-rw-rw-r--    1 rli      users           0 11月 30 14:34 newfile
[rli@arena rli]$ chmod a+x newfile          # 赋予所有用户执行权限
[rli@arena rli]$ ls -l newfile
-rwxrwxr-x    1 rli      users           0 11月 30 14:34 newfile
[rli@arena rli]$ chmod 654 newfile          # 使用八进制方式修改权限
[rli@arena rli]$ ls -l newfile
-rw-r-xr--    1 rli      users           0 11月 30 14:34 newfile
```

chmod 命令修改文件访问权限时可以用文本或八进制两种方式来描述。使用文本描述方式时，其语法为 [ugoa][+-][rwx]，其中第一部分表示修改哪部分用户的权限，字母 “u”、“g”、“o” 和 “a” 分别表示属主、用户组、其他用户和所有用户，第二部分表示增加 (+) 还是去除 (-) 相应的权限，而第三部分则表示是读 (r)、写 (w) 还是执行权限 (x)。比如 g+w 表示加上组中用户的写权限，o-x 表示去除其他用户的执行权限。如果只想增加或去除目录的执行权限 (表示是否允许进入)，但不希望改变普通文件的执行权限，可以用大写的 “X”。例如，“chmod a+X *” 会将当前目录中的所有子目录加上执行权限，但不改变普通文件的执行权限。

使用文本方式修改文件属性，命令比较容易读懂，而使用八进制方式修改文件属性则更加快捷。chmod 命令用三个 8 进制数来表示三类用户的权限，分别对应属主、用户组和其他用户，可读加 4，可写加 2，可执行加 1，如果是去除权限就减去相应位上的数。例如，上例中，“chmod 654 newfile” 等价于命令 “chmod uo-x,g-w newfile”。

使用 chown 和 chgrp 修改属主和组的例子如下：

```
[root@arena rli]# chgrp ftp newfile          # 注意 ftp 是系统上的一个组
[root@arena rli]# ls -l newfile
-rwxr-xr--    1 rli      ftp             0 10月  7 11:42 newfile
[root@arena rli]# chown root newfile
[root@arena rli]# ls -l newfile
-rwxr-xr--    1 root     ftp             0 10月  7 11:42 newfile
```

注意这里需要以超级用户，即 root 的身份进行操作。

如果希望同时修改整个目录中所有文件和子目录的属性，可以使用选项 -R。

Linux 中的文件或者目录除了拥有上面这些属性外，还可以定义一些叫做 attribute 的特性，它们在表 2.2 中列出。命令 chattr 和 lsattr 用于修改和查看这些特性，帮助用户更好地控制机器上的文件和目录。比如一些系统目录是不许修改其中的文件的，可以给这些目录加上 i 特性。合理使用这些特性有助于增强系统的安全性。

表 2.2 Linux 文件的特性

表示特性的字母	特性的含义
a	写操作的时候只能添加，不能删除现有内容。
c	内核会自动对文件进行压缩。
i	不能删除、移动、链接、写入。
s	文件删除时，磁盘空间也会清理乾净。
S	修改时直接写入磁盘，而不是写在缓冲区中。

15. ps，kill，nice，renice，top

查看和管理进程。直接使用 ps 命令可以看到现在正在运行的进程的信息；使用 kill 可以向某个进程发一个信号，通常用于改变进程的运行状态或杀死进程；使用 nice 和 renice 可以调整进程的优先级别，从而使得机器能够集中精力于更重要的进程；top 则是一个交互式的查看系统状况的工具。请看下面的例子：

```
[rli@arena rli]$ ps -u rli
    1170 ?        00:00:00 fetchmail
    5734 ?        00:00:00 .Xclients-defau
    5782 ?        00:00:19 scim
    5785 ?        00:00:00 ssh-agent
    5786 ?        00:00:08 fvwm
    5804 ?        00:00:01 FvwmPager
    5805 ?        00:00:17 FvwmTaskBar
    5806 ?        00:00:00 FvwmAnimate
    5810 ?        00:00:01 xscreensaver
    7218 ?        00:00:09 emacs
    7283 ?        00:00:00 xterm
    7285 pts/2    00:00:00 bash
    7317 pts/2    00:00:00 ps
    7318 pts/2    00:00:00 sh
```

上例显示了用户 rli 所有的进程。其中可以看到窗口管理器的一些进程，该用户在用 fetchmail 检查邮件，用 Emacs 编辑器进行编辑(事实上编辑的正是本章的排版源文件)，并且还启动了一个模拟终端 xterm 来运行命令 ps -u rli。下面用 kill 来杀掉邮件检查程序 fetchmail：

```
[rli@arena rli]$ kill -9 1170
[rli@arena rli]$ ps -u rli
    5734 ?        00:00:00 .Xclients-defau
    5782 ?        00:00:19 scim
    5785 ?        00:00:00 ssh-agent
```

```
5786 ?        00:00:08 fvwm
5804 ?        00:00:01 FvwmPager
5805 ?        00:00:17 FvwmTaskBar
5806 ?        00:00:00 FvwmAnimate
5810 ?        00:00:01 xscreensaver
7218 ?        00:00:09 emacs
7283 ?        00:00:00 xterm
7285 pts/2    00:00:00 bash
7317 pts/2    00:00:00 ps
7318 pts/2    00:00:00 sh
```

命令 kill 向指定进程发送一个信号 (signal)，上例中的信号是 “9” (KILL)，表示强制进程终止。在指定信号时，既可用信号值，如 “-9”，“-15” 等，也可用信号的名称，如 “-KILL”、“-TERM” 等。当不指定信号时，默认信号为 TERM，该信号通常终止进程的运行。表 2.3 列出了不同信号的意义。信号可以是系统产生的，例如当发生某种异常 (常见的有浮点溢出、非法内存访问等) 或事件时，也可以由一个进程发送给另一个进程。大部分信号能够被进程捕获，进程可以自行决定如何处理这些信号。但是 9 (KILL) 是不能被捕获的，进程会立即终止。因此，除非有必要，通常应该用 “kill -TERM” 来杀死进程，以便进程在退出前完成一些必要的清理和善后的工作。

nice 命令用于在启动进程时指定它的优先级别。普通用户只能降低进程的优先级别 (这是命令名 nice 的含义)，而系统管理员可以调高进程的优先级。其语法为 “nice -n 调整值 [命令 [参数]]”，所启动的进程会按照调整后的优先级运行。如果希望改变一个正在运行的进程的优先级，则需要用 renice 命令。

top 用于交互地监视系统的状态。运行命令时，屏幕就会成为一个字符界面的窗口，在中间分列显示系统各方面的情况，同时还可以进行一些交互操作。图 2.1 是一个 top 显示的例子。它分列显示了各个进程的进程号、属主、优先级、nice 值、占用内存 (代码

表 2.3 Linux 常用信号

信号名称	信号值	含义
HUP	1	挂起 (Hangup)，挂断终端时产生。
INT	2	中断 (Interrupt)，按 Ctrl-C 键时产生。
QUIT	3	退出 (Quit)，按 Ctrl-/ 键时产生。
BUS	7	总线错误。
KILL	9	强制性立即终止 (Kill)。
SEGV	11	段错误 (非法内存操作)。
TERM	15	一般性终止。
CONT	18	继续 (Continue)。
STOP	19	暂停 (Stop)，按 Ctrl-Z 键时产生。

段＋数据段＋堆栈段) 大小、使用的实际物理内存大小、共享内存大小、进程状态、占用 CPU 的百分比、占用内存的百分比、已使用的总 CPU 时间和命令名。按 r 键，可以交互地 renice 一个进程；按 k 键，可以杀死一个进程；还可按 m 键查看内存的使用情况。

16. &，nohup

在后台运行程序。当运行一个比较花时间、而又不是非常急迫的程序时，可以把它放到后台去运行，只要在命令后面加上一个 “&” 就可以了。如果希望在后台程序运行的同时退出系统 (logout)，可以用 nohup 命令来运行程序，避免从系统退出时程序被 HUP 信号杀掉：

```
[rli@arena rli]$ big_job &
[rli@arena rli]$ ps | grep big_job
1173 ?        00:00:08 big_job
[rli@arena rli]$ nohup even_bigger_job &
[1] 2341
[rli@arena rli]$ nohup: appending output to `nohup.out'
```

```
  9:52pm  up  7:57,  4 users,  load average: 0.31, 0.17, 0.12
83 processes: 82 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  0.3% user,  0.5% system,  0.0% nice, 99.0% idle
Mem:   247040K av,  238784K used,    8256K free,       0K shrd,   10440K buff
Swap:  262136K av,  112308K used,  149828K free                   97152K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 1101 root      17   0  178M  24M  2628 S     0.5 10.0   5:00 X
 7583 rli       14   0  1052 1052   828 R     0.1  0.4   0:00 top
    1 root       8   0   468  428   408 S     0.0  0.1   0:03 init
    2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
    3 root       9   0     0    0     0 SW    0.0  0.0   0:00 kapmd
    4 root      19  19     0    0     0 SWN   0.0  0.0   0:00 ksoftirqd_CPU0
    5 root       9   0     0    0     0 SW    0.0  0.0   0:00 kswapd
    6 root       9   0     0    0     0 SW    0.0  0.0   0:00 kscand/DMA
    7 root       9   0     0    0     0 SW    0.0  0.0   0:01 kscand/Normal
    8 root       9   0     0    0     0 SW    0.0  0.0   0:00 kscand/HighMem
    9 root       9   0     0    0     0 SW    0.0  0.0   0:00 bdflush
   10 root       9   0     0    0     0 SW    0.0  0.0   0:00 kupdated
   11 root      -1 -20     0    0     0 SW<   0.0  0.0   0:00 mdrecoveryd
   18 root       9   0     0    0     0 SW    0.0  0.0   0:00 knodemgrd
   23 root       9   0     0    0     0 SW    0.0  0.0   0:00 kjournald
   75 root       9   0     0    0     0 SW    0.0  0.0   0:00 khubd
  217 root       9   0     0    0     0 SW    0.0  0.0   0:00 kjournald
```

图 2.1 top 命令抓图

用 nohup 运行的命令会自动将它的输出定向到文件 nohup.out 中。

17. fg，bg，jobs

控制前台或者后台程序的运行。fg 命令将一个后台运行的进程转到前台运行，bg 则是使得一个进程在后台运行。在 nohup 的例子中，命令提交后系统会返回信息 “[1] 2341”，表示后台运行的命令的进程号为 2341，而它在当前终端里的作业号是 1。如果希望将作业转到前台来运行，只需使用命令 “fg %1”。在 fg 命令中，还可以用字符串来指定作业，比如 “%s*r” 表示命令行能够匹配上字符串

“s*r” 的进程。

按 Ctrl-Z 键，可以将正在前台运行的进程放到睡眠状态而将控制权交还给当前的 shell。随后后在命令行使用 bg 命令将使得该进程在后台继续运行。

命令 jobs 显示出当前 shell 中的所有作业及状态 (“Running”、“Stopped” 等)。所显示的作业中带有 “+” 号者为当前作业。fg、bg 等命令中如果不指定作业，则其操作对象为当前作业。

18. find，locate

搜索文件。find 命令在指定目录中查找文件：

```
[rli@arena rli]$ find /usr/share -name latex.ltx
find: /usr/share/ssl/CA: 权限不够
/usr/share/texmf/tex/latex/base/latex.ltx
[rli@arena rli]$
```

上例查找目录 /usr/share 下名为 latex.ltx 的文件。在查找过程中，如果命令遇到用户没有权限进入的目录，系统会给出相应的信息。命令 locate 也用于查找文件。当指定的目录中子目录、文件很多时，find 的运行会非常慢，这时，使用 locate 来查找往往会快得多。例如：

```
[rli@arena rli]$ locate latex.ltx
/usr/share/texmf/ptex/platex/base/platex.ltx
/usr/share/texmf/tex/latex/base/latex.ltx
/usr/share/texmf/tex/latex/carlisle/mylatex.ltx
[rli@arena rli]$
```

locate 的工作原理是当计算机空闲时预先搜索文件系统，将文件信息保存在一个数据库中。当用户查找文件时，它直接查找数据库，而不是到硬盘中去搜索，因此速度很快。通常，locate 会在机器最可能空闲时，如后半夜，更新自己的数据库。如果计算机经常处于关闭

状态的话，则 locate 可能找不到机会更新自己的数据库，从而导致查询的结果不对。

19. grep

寻找文本中的特定信息。例如，如果想在名为 ftp_log 的日志文件中找到有哪些是来自于北京大学 (IP 地址为 162.105.x.x 形式) 的服务请求，可以使用下面的命令：

```
grep '162\.105\.' ftp_log
```

下面是另外一个例子

```
[rli@arena rli] grep "北京大学" doc/*
```

它列出子目录 doc 下面所有包含 “北京大学” 字样的文件及相应行的内容。除 grep 外，还有 egrep、fgrep 和 zgrep 等命令，它们共同构成了 grep 命令家族。这些命令用正则表达式描述查找的字符串。关于正则表达式将在后面介绍。

20. cut

分列处理文本。通过指定分隔符号来将句子分列，然后将指定的列提取出来。这条命令用来处理文本简单而方便。后面还会介绍更加复杂的用于文本处理的工具，如 sed、awk 等。下面是几个 cut 的例子：

```
[rli@arena rli]$ cat > tmp.txt
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
(Ctrl-D)
[rli@arena rli]$ cut -d ":" tmp.txt -f2,5
x:FTP User
[rli@arena rli]$ cut -d ":" tmp.txt -f3-6
14:50:FTP User:/var/ftp
[rli@arena rli]$ cut -d "s" tmp.txt -f2
er:/var/ftp:/
```

21. tr

对文本中的字符进行替换。该命令将标准输入流中的字符进行一些简单的替换、缩并或者是删除操作。比如在 Windows 下编辑的文件转到 Linux 下时，每一行末尾都有一个 "^M" 字符，十分难看，还会导致有些程序不能正确处理。用下面的命令可以将文件中的 "^M" 去掉：

```
cat source_file | tr -d '\r' > dest_file
```

该例中，选项 "-d" 表示删除输入文本中的指定字符，"\r" 代表字符 "^M" (ASCII 字符 13)。

22. who，whoami，id，w

命令 who 和 w 显示出当前登录的所有用户的信息，whoami 和 id 则显示出当前用户的信息。请自行观察它们的输出。

23. mount，umount，df

这些命令分别用来挂载、卸载文件系统和显示当前挂载的文件系统。Linux 系统中可以同时使用多个文件系统，它们物理上可以位于不同的硬盘、硬盘分区或存储媒介上 (如光盘、移动硬盘等)。在 Linux 系统中，最主要的文件系统称为根文件系统 (root filesystem)，即包含根目录的文件系统。其他文件系统都被挂接在根文件系统的某级子目录中。这样，不论实际有多少个文件系统同时在使用，从逻辑结构上看整个文件系统依然是一个单一的树型结构。这一点和 Windows 系统有很大的不同。在 Windows 系统中，不同文件系统被映射成不同的逻辑分区，如 C:、D: 等，每个逻辑分区构成一个独立的树型结构。

在 Linux 系统中，挂载一个文件系统时，必须将它挂载到一个已有的子目录上，该子目录称为该文件系统的挂载点。当一个文件

系统挂载成功后，挂载点中原有的内容变得不可见，取代它们的是新挂载的文件系统中的内容。

挂载一个文件系统的命令是 mount，而卸载一个文件系统的命令是 umount。这两条命令只有超级用户才有权力运行。当一个文件系统卸载后，挂接点下面原有的内容又会重新暴露出来。命令 df 则显示出当前挂载的所有文件系统。当卸载一个文件系统时，如果其中的文件或目录被正在运行的进程使用，则系统将拒绝卸载该文件系统。此时，需先终止使用该文件系统的进程，然后再执行卸载操作。命令 lsof 列出当前打开的所有文件及打开文件的进程，通过它结合 grep 命令可以找出到底哪些进程在使用欲卸载的文件系统。

系统配置文件 /etc/fstab 中定义了系统启动时自动挂载的文件系统。当然，系统启动过程中必须首先挂载根文件系统，然后才能在其上挂载其他文件系统。做为例子，下面列出的是是作者机器上的 /etc/fstab 文件中的部分内容：

```
/dev/hda4            /                ext3    defaults          1 1
/dev/hda5            /home            ext3    defaults          1 2
/dev/fd0             /mnt/floppy      auto    noauto,owner,kudzu 0 0
/dev/sda1            /mnt/usb         auto    noauto,users      0 0
/dev/cdrom           /mnt/cdrom       auto    noauto,owner      0 0
```

其中，第一列是文件系统所使用的设备 (硬盘分区、软驱、光驱、USB 盘等) 或文件，第二列是挂载点，后面几列分别是文件系统类型，挂载选项，最后两个数字分别说明是否进行 dump 备份标识和启动时对文件系统进行检查，这里就不详细解释了。上面列出的第一行就是根文件系统，它对应着硬盘 hda 上的第 4 个分区 (/dev/hda4)。第二行是用户家目录所在的文件系统 (在硬盘的第 5 个分区上)。后面三行分别是软盘驱动器、U 盘和光驱的挂载信息，由于使用了选项 “noauto”，系统启动时不会自动挂载它们。Linux 系统中，通常用 hda、hdb 等表示 IDE 硬盘，用 sda、sdb 等表示 SCSI 硬盘。USB 盘

被内核模拟为 SCSI 盘，因此它的设备名为 sda、sdb 等形式。

除了可以挂载真实的文件系统外，也可以挂载存储在文件中的一个文件系统的映像。例如可以用下述命令挂载一个光盘映像文件：

```
mount -o loop iso_file.iso /mnt/cdrom
```

要想卸载一个文件系统，在 umount 后面指定设备名或者挂载点均可。如

```
umount /dev/sda1
```

或

```
umount /mnt/usb
```

24. bc

任意精度计算器。Linux 系统中有各种各样的计算器，其中一些是图形界面的，如 xcalc、kcalc (KDE 的计算器)、gcalctool 等，它们的使用与 Windows 下的计算器类似。另一些是命令行形式的，它们在一个终端中运行，通过命令行或标准输入接受表达式，这里介绍的 bc 就属于这一类。命令行形式的计算器实际上使用效率更高，并且可以用在 shell 脚本中，弥补 shell 本身只能进行整数运算的缺陷。下面是使用 bc 的一些例子：

```
[rli@arena rli]$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
3/5                                          # 计算表达式 3/5
0                                            # 默认结果为整数
scale=50                                     # 设定结果精度为50位
3/5                                          # 重新计算表达式 3/5
.60000000000000000000000000000000000000000000000000
```

```
quit                                          # 退出 bc
[rli@arena rli]$ bc -l
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
3/5                                           # 计算表达式 3/5
.60000000000000000000                         # 默认结果为浮点数
scale=50                                      # 设定精度为50位
s(3)                                          # 计算表达式sin(3)
.14112000805986722210074480280811027984693326425226
quit                                          # 退出 bc
[rli@arena rli]$ echo "1/sqrt(3)" | bc -l     # 直接在命令行计算1/√3
.57735026918962576451
[rli@arena rli]$
```

当不加 `-l` 选项时，`bc` 默认进行整数运算，并且不能使用某些数学函数。而加 `-l` 选项后，`bc` 默认进行浮点数运算，并且允许使用所有它所支持的数学函数，如上例中 `s(3)` 表示 sin(3)。`bc` 的一个重要特征是它可以进行任意精度的浮点运算，通过设定 `scale` 变量的值可以指定结果的精度。上面最后一个例子是通过管道直接在命令行中完成计算，关于管道会在后面介绍。有关 `bc` 更进一步的说明请参看它的 `man` 或 `info` 在线资料。

2.2.2 Shell

本节介绍更多关于 shell 的知识。每个人都有自己习惯使用的 shell，并不拘泥于一定要使用哪个 shell。这里主要介绍 Bash，因为 Linux 的许多发行版将 Bash 作为默认的 shell。

首先介绍几个 shell 使用的基本技巧。

1. 自动补全

当在 shell 中输入命令时，只要输入命令名、目录或文件的开头几个字符，然后按 Tab 键，Bash 便会自动将名称补全。比方说，要查看一下 /etc/sysconfig 目录中的内容，只要输入 “ls /etc/sys” 然后接连按两下 Tab 键，会将 /etc/ 目录下所有以 sys 开头的文件和目录显示出来，包括 sysconfig、sysctl.conf 和 syslog.conf。如果输入 “ls /etc/sysc” 再重复这个动作，那么显示结果就只剩下 sysconfig 和 sysctl.conf 了，因为以 sysc 开头的只有这两个文件。如果输入 “ls /etc/sysco” 再按 Tab，Bash 会将 /etc/sysconfig 文件名补全。如果所输入的部分对应的文件或目录是唯一的，只要按一下 Tab 就能补全，否则会听到一下响铃声，这时再按一下 Tab，会将所有匹配的文件列出来。当符合条件的文件太多时，系统会先显示符合条件的文件数目，用户回答 y(es) 后才显示。用这种方式，输入文件名时往往只要输入开头的几个字符即可。对于命令名字本身，Bash 也可以自动补全。例如，只要输入 xscr 然后按 Tab 就能够自动将 xscreensaver 命令补全。

2. 历史记录

Bash 在每次执行完用户输入的命令后，会将命令行保存在存放在命令历史记录 (command history) 中。命令历史记录中保存的每条命令有一个编号。用 history 命令可以查看当前的命令历史记录。用上下箭头键可以找出出命令历史记录中的某条命令，这里按回车键便会再次执行该命令。当然，在执行历史记录中的命令前，还可以用左右箭头、Backspace、Delete 等键对它进行编辑、修改。其他一些重复执行命令历史记录中的命令的方法包括：输入 !n (其中的 n 是 history 命令显示的命令记录号码) 执行指定编号的命令，输入 !! 执行最后一条命令，输入 !ls 执行最近一条以 ls 开头的命令。当退出 Bash 时，命令历史记录会被保存到文件 ~/.bash_history

中，下次进入 Bash 时，Bash 会从该文件中调入所保存的命令历史记录。不过，Bash 只保持一定数量的命令记录，可以通过设定环境变量 HISTFILESIZE 来指定历史记录的保存数目。

3. 命令别名

对于某些长命令或参数，可以定义命令别名 (alias) 来简化它们的输入。Bash 中，命令别名是用单个命令名来代替一长串命令及参数。例如命令 “alias shdn='shutdown -h now'” 定义了一个命令别名 shdn，每次输入 “shdn” 命令就相当于输入 “shutdown -h now”。命令别名的另外一个功能是为某些命令设定一些预定的参数，例如，“alias rm='rm -i'” 表示每次输入 “rm” 命令时，系统会执行 “rm -i”，即自动加上 “-i” 选项，该选项在删除一个文件前会要求用户进行确认。如果想取消一个命令别名，可以用 unalias 命令，如 “unalias shdn”。

Bash 开始运行时，会自动执行一些初始化文件，这些文件包括 /etc/bashrc、~/.bashrc 等。通常，可以将命令别名的定义放在这些初始化文件中，这样，每次登录时便会自动定义这些命令别名。

4. 重定向和管道

在 Linux 系统中有三个特殊文件，称为基本输入输出文件，它们分别是标准输入 (stdin)、标准输出 (stdout) 和标准错误 (stderr)。一个进程开始运行时会自动打开这三个文件，其文件号分别为 1、2 和 3。通常，命令运行时从标准输入读入输入，然后再将处理的结果输出到标准输出，如果处理过程中遇到错误，错误信息会显示在标准错误中。

默认情况下，基本输入输出文件通常对应于用户的终端：标准输入就是键盘，而标准输出和标准错误则是屏幕或窗口。在 shell 命令行上可以用字符 “>”、“<” 和 “&” 对基本输入输出进行重新定向，

将它们转到指定的文件或设备中。下面是一些输入输出重定向的例子：

- 重定向标准输入

```
[rli@arena rli]$ tr ":" "|" </etc/passwd
```

将文件 /etc/passwd 作为命令 tr 的标准输入。

- 重定向标准输出

```
[rli@arena rli]$ cat /etc/passwd >tmp
```

将文件 tmp 作为命令 cat 的标准输出。

- 重定向标准错误

```
[rli@arena rli]$ find /usr/share/texmf -name latex.ltx 2>/dev/null
```

将设备 /dev/null 作为命令 find 的标准错误。“>” 前面的数字代表文件号，这里是 “2”，表示标准错误。/dev/null 是一个特殊设备，所有输入给它的内容都会被丢弃。

- 同时重定向标准输入、标准输出和标准错误

```
[rli@arena rli]$ tr ":" "|" </etc/passwd >tmp 2>/dev/null
```

- 将标准错误定向到标准输出

```
[rli@arena rli]$ find /usr/share/texmf -name latex.ltx >tmp 2>&1
```

该例中，首先将标准输出定向到文件 tmp，再将标准错误定向到标准输出。注意，“2>&1” 的含义是将标准错误 (2) 定向到标准输出 (1) 当前所关联的文件或设备，因此，“>tmp 2>&1” 与 “2>&1 >tmp” 的结果是不同的，前者中标准错误被定向到文件 tmp，而后者中标准错误被定向到终端，因为当遇到 “2>&1” 时标准输出依然是终端!

当用 “>” 将标准输出或标准错误重定向到一个文件时，如果指定的文件不存在，则会建立一个新文件；如果文件已经存在，那么，该文件的内容会首先被清空。如果想保留文件的原有内容，而将输出添加在文件的最后，可以用连续两个大于号 “>>” 代替 “>”。

对于标准输入，也可以用连续两个 “<”，即 “<<”，来进行重定向。看下面的例子：

```
[rli@arena rli]$ write userb <<END
> Hello!
> I'm in room S2560
> Where are you now?
> END
[rli@arena rli]$
```

注意 “<<” 后面跟随的不是一个文件名或设备名，而是一个用于标志标准输入结束的字符串。如果在一个 shell 脚本中使用 “<<”，则脚本中的后续内容会被作为标准输入的内容，直到遇到标志结束的字符串为止。稍后会介绍有关 shell 脚本的知识。

管道的英文单词是 pipe，在 shell 中，它指用符号 “|” 将一个命令和另一个命令连接起来，将前一个命令的标准输出作为后一个命令的标准输入，这样，后一个命令就可以直接处理前一个命令输出的结果。下面的例子通过一串管道操作，取出本机器的 IP 地址。用命令 `/sbin/ifconfig` 可以得到关于网卡的信息，例如

```
[rli@arena rli]$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 03:F8:74:4A:6E:E2
          inet addr:162.105.42.214  Bcast:162.105.68.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:849285 errors:0 dropped:0 overruns:0 frame:0
          TX packets:158131 errors:0 dropped:0 overruns:0 carrier:0
          collisions:1475 txqueuelen:100
          RX bytes:65686865 (62.6 Mb)  TX bytes:75337627 (71.8 Mb)
          Interrupt:9 Base address:0xdcc0 Memory:ff6e0000-0
```

利用管道，对上面的输出进行进一步处理，可以将想要的信息 (IP 地址) 提取出来：

```
[rli@arena rli]$ /sbin/ifconfig eth0 | grep "Bcast:"
          inet addr:162.105.42.214  Bcast:162.105.42.255  Mask:255.255.255.0
[rli@arena rli]$ /sbin/ifconfig eth0 | grep "Bcast" | tr -s ' ' ' '
 inet addr:162.105.42.214 Bcast:162.105.42.255 Mask:255.255.255.0
[rli@arena rli]$ /sbin/ifconfig eth0 | grep "Bcast:" | tr -s ' ' ' ' \
| cut -d ' ' -f3
addr:162.105.42.214
[rli@arena rli]$ /sbin/ifconfig eth0 | grep "Bcast:" | tr -s ' ' ' ' \
| cut -d ' ' -f3 | cut -d ':' -f2
162.105.42.214
```

在上例中，可以看到机器的 IP 地址在 `ifconfig` 命令的输出的第二行，该行中包含有一个特有的词 “`Bcast:`”。因此，首先用 `grep` 查找关键词 “`Bcast:`” 将第二行单独提取出来。然后，用 `tr` 命令将连续多个空格合并成为一个空格，再用空格作为分隔符将该行内容分成四列，用 `cut` 命令取出第三列得到字符串 “`addr:162.105.42.214`”。最后，用 “:” 作为分隔符取出第二列，便是机器的 IP 地址。该例中通过管道，用不同命令反复对输出进行处理，提取出最终的信息。这是 shell 中处理文本最常用的办法。Linux 系统提供了许多功能强大的文本处理程序，利用它们结合管道功能，能够实现对文本的几乎任何处理。使用管道不仅使得整个命令连贯易读，而且与通过文件传递中间结果相比效率也高得多，特别是当处理的内容非常多时优点更为明显。如果系统中有多个处理器的话，利用管道还可以实现并行处理。

5. 环境变量

Linux 系统启动的时候，会启动一个特殊的进程，名为 `init`，它负责控制整个系统的运行及启动所有其他进程。当用户登录进入系统的时候，首先得到一个 shell，它也是一个进程，其中中执行的命令

都是该 shell 的子进程。事实上，只有当获得一个 shell 后，用户才真正开始和计算机沟通，例如输入命令、执行程序等等。在一个 shell 中，可以再进入另外一个 shell，即启动一个子 shell，然后还可以再进入更深一层的 shell。每次输入 `exit` 命令则退出一层 shell，回到上一个 shell 中。任何一个进程都可以启动其他一些进程，这些进程称为该进程的子进程，而该进程则被称为这些进程的父进程。在实际应用中，往往需要从父进程向子进程传递一些初始参数或设置。通常，有两个方法从父进程向子进程传递初始参数，第一个方法是在启动子进程时通过命令行参数传递，第二个方法是通过环境变量传递。

在 Linux 的进程中，变量是一个具有自己的名字，其值能够在一定范围内保持和被改变的量。当启动一个子进程时，父进程中定义的部分变量会被子进程所继承。由于这些变量起着设定进程的运行环境的作用，因而被称为环境变量 (environment variables)。运行同一个命令，如果环境变量的设置不同，则命令的运行方式和运行结果也有可能不同。需要注意的是，环境变量是一个进程所固有的，只能在父进程和子进程之间传递。同时运行的不同进程可以拥有不同的环境变量，每个进程只能查看、修改自己的环境变量。

环境变量名称通常用大写字母表示。Linux 只负责在进程间传递环境变量的值，至于如何使用这些环境变量是由进程自行决定的。但是，有一组环境变量的作用是所有进程共同遵循的，这里称它们为标准环境变量。表 2.4 中列出了一些常用的标准环境变量，以及 Bash 中常用的一些环境变量。

在 shell 中，如果想引用某个环境变量的值，只要在变量名称前面加上一个 “$”。例如

```
[rli@arena rli]$ echo $PWD
/home/rli
[rli@arena rli]$ echo $$
```

表 2.4 常用环境变量

环境变量	所代表的含义	环境变量	所代表的含义
$HOME	用户的家目录	$PWD	当前工作目录
$PATH	命令名搜索路径	$MANPATH	man 搜索路径。
$USER	使用者的用户名	$UID	使用者的用户 ID
$LANG，$LC_*	有关语言的设定	$SHELL	shell 程序
$TERM	终端类型	$OSTYPE	操作系统类型
$MAIL	邮箱	$MAILCHECK	自动邮件检查 (秒)
$IFS	预设分列符	$PPID	父进程号
$$	当前 shell 进程号	$?	上一个命令退出码
$HISTCMD	命令的历史记录号	$HISTFILE	历史记录的文件名
$HISTSIZE	历史记录文件大小	$LINENO	当前命令的行号
$OLDPWD	上次所在的目录	$TMOUT	自动退出的闲置时间
$SECONDS	当前 shell 墙上时间		

```
1347
[rli@arena rli]$ echo $?
0
```

其中，第一个命令将当前目录的路径显示出来，和执行 pwd 命令的结果是一样的；第二个命令将当前 shell 的进程号 (1347) 显示出来，可以和命令 “ps -u rli” 的结果进行对比；第三个命令将前一个命令的返回值 (退出码) 显示出来。通常，在 Linux 中，返回值为 0 表示程序的运行顺利完成，否则则表示程序在运行过程中发生了某类错误。

如果想定义一个新的环境变量，或者改变现有环境变量的值，用等号 “=” 就可以了。例如：

```
[rli@arena rli]$ MY_CAT_NAME="Mi Mi"
[rli@arena rli]$ echo $MY_CAT_NAME
```

```
Mi Mi
```

假如想要删除一个环境变量，可以用 `unset` 命令：

```
[rli@arena rli]$ unset MY_CAT_NAME
[rli@arena rli]$ echo $MY_CAT_NAME

[rli@arena rli]$
```

在 shell 中，新定义的环境变量默认是不传递给 shell 的子进程的。如果想将新定义的环境变量传递给子进程，必须用 `export` 命令输出该变量。作为一个子进程，它只继承自己启动时父进程所传递的环境变量。子进程开始运行后是无法得到父进程新定义的环境变量、或是修改的环境变量值的。反过来，子进程对环境变量的改变对父进程没有影响。请看下面的例子：

```
[rli@arena rli]$ MY_CAT_NAME="Mi Mi"
[rli@arena rli]$ echo $MY_CAT_NAME
Mi Mi
[rli@arena rli]$ export MY_CAT_NAME
[rli@arena rli]$ bash
[rli@arena rli]$ echo $MY_CAT_NAME
Mi Mi
[rli@arena rli]$ export MY_CAT_NAME="Xiao Mi Mi"
[rli@arena rli]$ echo $MY_CAT_NAME
Xiao Mi Mi
[rli@arena rli]$ exit
[rli@arena rli]$ echo $MY_CAT_NAME
Mi Mi
```

另外，在定义环境变量时还需要注意一些细节：

- 定义环境变量时，“=” 号两边不能有空格；
- 环境变量的名称，只能是字母、数字和下划线，且不能以数字开头；

- 系统预定义的环境变量均为大写；
- 如果环境变量的值中带有特殊字符或空格，必须用 “\” 转义，或用引号将环境变量的值引起来；

下面的例子说明 shell 中引号的作用：

```
[rli@arena rli]$ MY_CAT_NAME="Mi Mi"
[rli@arena rli]$ echo 'My cat name is "$MY_CAT_NAME"'
My cat name is "$MY_CAT_NAME"
[rli@arena rli]$ echo "My cat name is \"$MY_CAT_NAME\""
My cat name is "Mi Mi"
```

括在单引号 “'” 中的内容，所有字符都作为普通字符处理，任何特殊字符，除单引号外，都失去其特殊含义。但单引号中不能再使用单引号。而在双引号中，某些特殊字符，如 “$”、“\” 等，依然保留其特殊的功能。

除了可以用外部命令，如 grep、cut 等，处理环境变量中的字符串，Bash 本身也提供了一些对环境变量的值进行过滤、处理的功能。Bash 中常用的过滤语法和规则在表 2.5 中给出。在这些过滤规则中，字符 “*” 用于匹配任意字符串。

表 2.5 Bash 的环境变量字符串过滤 (假设 V=/home/rli/rli.cpp.bak)

过滤格式	含义	过滤结果
${V}	显示该环境变量	/home/rli/rli.cpp.bak
${V##/*/}	去掉以 /*/ 开头的最长部份	rli.cpp.bak
${V#/*/}	去掉以 /*/ 开头的最短部份	rli/rli.cpp.bak
${V%.*}	去掉以 .* 结尾的最短部份	/home/rli/rli.cpp
${V%%.*}	去掉以 .* 结尾的最长部份	/home/rli/rli
${V/rli/who}	将第一个 rli 替换成 who	/home/who/rli.cpp.bak

用户登录进系统工作时，通常需要设定一组特定的环境变量，如可执行程序的搜索路径 PATH，在线手册的搜索路径 MANPATH，终端类型 TERM，语言设定 LANG 等。Bash 在用户登录时会自动执行一些初始化脚本，通常可以在这些脚本中设置所需要的环境变量。这些初始化脚本包括 /etc/profile、/etc/bashrc、~/.bash_profile、~/.bashrc，以及 /etc/profile.d/ 目录中所有以 ".sh" 为扩展名的脚本。这些脚本中，一些是通过另外一些导入的。通常，/etc 下的初始化脚本中的设置对所有用户起作用，而每个用户家目录下的初始化文件中的设置则仅对该用户起作用。一个 Bash 脚本中，可以用 "." 命令导入另外一个脚本，其效果相当于将被导入的脚本直接插入到导入的位置。Bash 的一些初始化脚本，如 /etc/profile.d/ 中的脚本，就是被另一些脚本，如 /etc/profile，用这种方式导入的。

Bash 变量还具有一些属性，可以用内部命令 declare 来赋予。这些属性会对 Bash 变量的展开操作产生影响。例如，当用命令

```
name=value
```

将字符串 value 赋给变量 name 的时候，Bash 通常会在赋值前对 value 进行波浪号展开、文件名展开、参数和变量展开、命令替换、算术展开和引号去除等操作。但如果变量 name 具有 "整数" 属性的话，则 Bash 将只对字符串 value 进行算术展开。关于这些展开操作将在稍后介绍。在 Bash 中，变量赋值表达式可以直接作为 "declare"、"typeset"、"export"、"readonly"、"local" 等内部命令的参数。例如，可以将

```
name=value
export name
```

写成

```
export name=value
```

的形式。

6. 命令行展开

在前面的例子中看到，可以用命令

```
[rli@arena rli]$ ls -l data/*.dat
```

将目录 data 下面扩展名为 .dat 的文件列出来。这里，“*” 代表任何不包含 “/” 的字符串。Bash 在执行该命令时，先将 “data/*.dat” 展开成与之匹配的文件名，称为文件名展开，然后再执行展开后的命令。事实上，执行命令前，Bash 除了对命令行的内容进行文件名展开外，还进行许多其他展开和处理，包括花括号展开、波浪号展开、变量展开、命令替换、算术展开和词分割。下面一一介绍这些展开。

- 花括号展开：这是最先被实施的展开，展开结果是依次用花括号中逗号隔开的每个字符串替代整个花括号得到的所有字符串。例如 “a{d,c,b}e” 的展开结果为 “ade ace abe”。但是，如果花括号前面是 “$”，例如 “${PATH}”，则 Bash 将花括号内的部分当作一个变量名处理而不进行花括号展开，这是为了避免和后面的参数展开发生冲突。花括号展开可以嵌套，如 “a{d{c,f?},{*g,h_}}e” 的展开结果为

  ```
  adc*ge adch_e adf?*ge adf?h_e
  ```

- 波浪号 (~) 展开：一个单独的 “~” 展开为当前用户的家目录，即环境变量 $HOME 的值，而 “~rli” 则展开为用户 rli 的家目录；

- 变量展开：一个变量名前面加上一个 “$” ，就会进行变量展开，展开的结果就是这个变量的值。变量名可以放在一对花括号中，前面已经说过，这对花括号不会被当作花括号展开处理。当变

量名是一个多于一位数字的位置变量，或者变量名后面紧跟着允许出现在变量名中的字符的话，必须加上花括号才能避免混淆。如果在花括号中的变量名前加一个感叹号 “!”，那么展开结果是以变量的值为名称的变量的值，但如果同时花括号中的字符串后面以 “*” 结尾，则展开结果是所有以该字符串开头的变量名。如 “${!USER*}” 会展开为 “USER USERNAME” 等。其他一些特殊用法列举在下面，其中 “PARAMETER” 代表变量名，“WORD” 代表字符串，“OFFSET” 和 “LENGTH” 代表整数或算术表达式：

- ${PARAMETER:-WORD}：如果 PARAMETER 没有值或者是空串则使用 WORD 的展开，否则使用 PARAMETER 的值；
- ${PARAMETER:=WORD}：如果 PARAMETER 没有值或者是空串，则先将 WORD 的展开结果赋值给它，然后，使用 PARAMETER 的值；
- ${PARAMETER:?WORD}：如果 PARAMETER 没有值或者是空串则将 WORD 的展开输出到标准输出 (如果 shell 不是交互方式则会退出)；否则使用 PARAMETER 的值。更为确切地说，该表达式的含义是当 PARAMETER 没有定义时显示给定的信息，否则使用 PARAMETER 值；
- ${PARAMETER:OFFSET} 和 ${PARAMETER:OFFSET:LENGTH}：展开为 PARAMETER 的展开中从 OFFSET 开始、最多包含 LENGTH 个字符 (第二种形式) 的子串。OFFSET 和 LENGTH 可以是算术表达式，LENGTH 的结果必须是非负整数，而 OFFSET 可以取负值，表示从 PARAMETER 的值的最后位置往前数。PARAMETER 的第一个字符 (倒数时最后一个字符) 的位置为 0。如果 PARAMETER 是 “@” 的话，则表示从第 OFFSET 个位置参数开始的最多 LENGTH 个位置参数 (关于位置参数参看 126 页“命令行参数”)；

- `${!PREFIX*}`：展开为所有以 “PREFIX” 作为开头的所有变量名，变量名间用环境变量 IFS 的值中的第一个字符隔开；
- `${#PARAMETER}`：展开成为 PARAMETER 的值的长度。如果 PARAMETER 是 “*” 或者 “@”，则展开为位置参数的个数；如果 PARAMETER 是一个用 “*” 或者 “@” 做脚标的数组，则展开为该数组中成员的个数 (关于 Bash 数组不在此介绍)；
- `${PARAMETER#WORD}` 和 `${PARAMETER##WORD}`：首先对 WORD 进行展开，然后将展开的结果与 PARAMETER 的值的开头部分进行匹配。第一个表达式 (“#” 的情形) 的展开结果是 PARAMETER 的值去掉最短匹配后剩下的部分。第二个表达式 (“##” 的情形) 的展开结果是去掉最长匹配后剩下的部分。如果 PARAMETER 是 “*” 或者 “@” 的话，相应操作会应用到所有的位置参数上，展开结果是字符串列表；如果 PARAMETER 是一个用 “*” 或者 “@” 做脚标的数组的话，相应操作会应用到数组中的每个成员，展开结果是字符串列表；
- `${PARAMETER%WORD}` 和 `${PARAMETER%%WORD}`：与 “#” 和 “##” 完全类似，但是与 PARAMETER 的值的尾部进行匹配；
- `${PARAMETER/PATTERN/STRING}` 和 `${PARAMETER//PATTERN/STRING}`：首先展开 PATTERN，然后将 PARAMETER 展开的结果中与 PATTERN 匹配的最长部分替换为 STRING。第一个用法中 (单个 “/”)，只替换第一个匹配的部分。第二个用法中 (连续两个 “/”)，替换所有匹配的部分。如果 PATTERN 以 # 开头，则只对 PARAMETER 的头部进行匹配。如果 PATTERN 以 % 开头，则只对 PARAMETER 的尾部进行匹配。如果 STRING 是个空串，那么匹配的部分会被删除掉。如果 PARAMETER 是 “*” 或者 “@” 的话，相应的

操作会应用到所有位置参数上，展开的结果是字符串列表。如果 PARAMETER 是一个用 “*” 或者 “@” 做脚标的数组，则相应的操作会应用到数组的每个成员，展开结果是字符串列表。

- 命令替换：命令替换指用一个命令执行的结果来替换这个命令本身。命令替换可以用两个方式实现：“$(命令)” 或者 “`命令`” (注意这里 “`” 是反向单引号，它通常位于键盘左上角的 “Esc” 键下面)。命令执行结果的最后一个换行会被自动去掉 (中间的换行不会被去掉，但是有可能在词法分析时被去掉)。命令替换 “$(cat 文件名)” 和 “$(<文件名)” 的效果是一样的，但是后者的执行速度会更快 (因为不用启动外部命令)。命令替换可以嵌套。在使用反单引号方式的时候，嵌套的反单引号需用反斜杠进行转义。如果命令替换出现在一对双引号中，得到的结果将不再进行词法分析和文件名展开。

- 算术展开：算术展开的形式为 “$((算术表达式))”。表达式中的所有部分会首先进行参数展开、命令展开和引号去除，然后再进行算术运算，运算结果便是展开结果。算术展开也可以进行嵌套。如果表达式是无效的或非法的，Bash 会在标准错误上显示一条错误信息，而不进行展开。

- 参数分割：将上面的各种展开后的结果分解为分离的一个一个词。环境变量 IFS 中列出的字符被作为断词用的分隔符。如果没有设置环境变量 IFS 或者它的值是空串，则默认分隔符为 “<Space><Tab><Newline>”。括在双引号或单引号间的内容不会被分开。

- 文件名展开：在参数分割以后，如果命令中没有 “-f” 选项，Bash 会扫描每个词，在里面寻找 “*”、“?” 和 “[” 这三个字符。如果

找到了，那么这个词就被作为一个模板，替换成一串与其相匹配的文件名。这些文件名按照字母顺序排列。如果没有任何与之相匹配的文件，而 Bash 选项 “nullglob” 又被禁止的话，那么这个词将会原封不动地保留下来；如果没有禁止 Bash 选项 “nullglob”，而又没有匹配到任何文件，那么这个词就会被去掉。如果设置了 Bash 选项 “nocaseglob”，那么在进行匹配的时候将忽略字母大小写的不同。

当对一个模板进行文件名展开时，如果没有设置 Bash 选项 “dotglob”，则当字符 “.” 出现在一个文件名的开头或者在一个斜杠之前时必须精确地和 “.” 匹配，即不包含在 “?” 或 “*” 中。斜杠 “/” 总是被精确匹配。其他情况下，字符 “.” 与其他字符一样，可以被 “?” 或 “*” 匹配。

进行模板匹配的规则如下：如果不是下面列举的特殊字符，那么字符将和它自己相匹配。在模板中，不能出现 NUL (ASCII 0) 字符。如果想匹配特殊字符，可以用引号将它们引起来。模板中的特殊字符及含义如下：

- “*”：和任何字符串相匹配，包括空字符串；
- “?”：和任何单个字符相匹配；
- “[...]”：和方括号中指定的任何一个字符相匹配。方括号中指定字符有很多方法，最基本的方法是将所有字符全部写出来。例如，[acd;] 表示和 “a”、“c”、“d”、“;” 相匹配。此外，可以用 “-” 来表示一对字符间的所有字符，如 [a-z0-9] 表示所有小写字母和数字。如果紧跟着左方括号的是字符 “!” 或者 “^” 的话，则表示与方括号中指定的字符之外的字符相匹配。如果想要匹配连字符 “-”，可把它放在方括号中开头或者结尾的位置。如果想要匹配 “]”，必须将它作

为方括号中的第一个字符。最后，还可以用 POSIX 1003.2 标准中定义的字符类别来指定。例如，用 “[:alpha:]” 表示所有字母。POSIX 1003.2 标准定义的字符类别有下面一些：

```
alnum alpha ascii blank cntrl digit graph lower print
punct space upper word xdigit
```

它们的含义从字面上就可以猜到。

- 引号去除：在完成前面所有这些展开以后，所有没有被引起来的反斜杠、单引号和双引号，如果不是前面这些展开的结果、而是最初的输入话，会全部被扔掉。

7. Shell 脚本

前面介绍的 shell 使用方法都是所谓交互式的 (Interactive) shell，用户输入一条命令，shell 马上执行命令，将结果返回来。Shell 也能以非交互的方式运行，类似于 MSDOS 中的批命令文件。即将一大段 shell 命令放在一个文件中，然后让 shell 以批处理的方式执行所有命令。这样的文件叫做 *shell* 脚本 (shell script)。Shell 脚本是解释执行的，它们和用 C 语言或 Fortran 语言编写的程序不同，后者通过编译、链接，变成一个可执行程序后才能被执行。一个脚本写好后，马上就可以执行。Linux 系统中有许多不同的脚本语言，而 shell 脚本是其中最重要、也是最基本的一种。它们大量地存在于系统中，从控制系统运行、进行系统配置，到许多实用程序，shell 脚本可以说无处不在。Fedora Linux 中，系统服务程序大都是通过目录 `/etc/init.d/` 下的 shell 脚本启动的；用户登录之后的环境设定也是通过 shell 脚本完成，如前面介绍过的 `.bash_profile`、`.bashrc` 等文件。掌握编写 shell 脚本的知识对于高效使用 Linux 系统，快速完成一些复杂的处理是非常有帮助的。

Bash 的工作流程是从输入读入一行命令，对命令中的字符串进行处理、解释、执行，然后等待读入新的命令。确切地，Bash 读入、执行一条命令的过程可分解为下面七个步骤：

(1) 从文件或者用户的终端读入字符串；

(2) 将输入分解成为“词”和“操作符”。在这一步中，会对引号进行处理，进行 alias 替换；

(3) 将这些“词”和“操作符”分解成简单命令或者组合命令；

(4) 进行各种各样的 shell 展开，将展开结果分解成命令名、文件名、参数等；

(5) 进行输入输出重定向并将重定向符号和重定向参数从参数表中去掉；

(6) 执行上面一系列处理后得到的最终命令；

(7) 等待命令执行完成并得到命令执行的结果 (返回码)；如果将命令放到后台执行，则 Bash 将不等待命令的完成而直接转入处理下一条命令。

下面比较详细地介绍一下 Bash 作为一种程序语言的语法。首先介绍 Bash 的词法和保留词。Bash 将一行命令分解成为一个个单独的“词”，分隔这些词的字符包括下面一些：

```
| & ; ( ) < > Space Tab
```

其中空格和制表符的作用是一样的，它们都被称为“空白”。这些词中有两类比较特殊。一类是“标识符”，由字母、数字和下划线组成，而且开头不能是数字。另一类是“控制符”，控制符只有下面几个：

```
|| & && ; ;; ( ) | 换行
```

它们控制 Bash 程序的执行流程。

有一组词对于 Bash 来说有特殊的含义，它们叫做保留词。当这些词没有被引号引起来的时候会起到特殊的作用。这些保留词有：

```
! case do done elif else esac fi for function if in select then until
while { } time [[ ]]
```

Bash 的语法基本上可以用下面几条来描述：

- 单个命令 (simple command)：单个命令包括一个命令名、用空格分开的参数、重定向操作符和一个控制符结尾。简单命令的返回值是所执行的命令的退出码。如果命令执行时被一个信号终止，那么返回值是 128 加上信号值。

- 流水线 (pipeline)：流水线指一系列用管道连接起来的命令。由于管道具有比重定向更高的优先级别，因此如果同时使用了管道和重定向，输出内容将被定向到管道中。流水线的语法是

  ```
  [time [-p]] [ ! ] 命令1 [ | 命令2 ... ]
  ```

  整个命令的返回值是最后一条命令的返回值。如果加了感叹号“!”，则返回值是最后一条命令的返回值的“逻辑否”。如果前面加了关键字“time”，则命令结束时 Bash 会报告执行命令所花费的时间，包括墙上时间和 CPU 时间。如果在 time 后面使用了选项“-p”，则输出运行时间时将采用 POSIX 格式。

- 命令序列 (list)：一个命令序列就是用下面的符号连接起来的一串单个命令或流水线：

  ```
  ;  &  &&  ||
  ```

  这些命令的最后必须用“;”、“&”或者换行结束。用“&”结束表示将命令放到后台去执行。后台执行的命令实际上是在一个子

shell 中执行，此时命令序列的返回值是 0。对于其他情形，命令序列的返回值为最后一条命令的返回值。

用 “;” 连接起来的命令按照顺序依次执行，Bash 会一直等待它们全部执行完。

“&&” 和 “||” 的作用类似于 C 语言中的同名逻辑运算符，前者表示“与”，后者表示“或”，操作对象是命令的返回值，返回值为 0 表示“真”，返回值非 0 则表示“假”，采用短路规则。它们的语法分别是

```
命令1 && 命令2
```

和

```
命令1 || 命令2
```

对第一种情况，当命令1 的返回值为 0 (“真”) 时命令2 才执行；而对于第二种情况，当命令1 的返回值是非 0 (“假”) 时命令2 才会执行。两种情况中命令序列的返回值都是最后所执行的命令的返回值。

- 组合命令 (compound command)：组合命令有下面几种。
  - “(命令序列)”：命令序列括在一对小括号中。小括号中的命令会在一个子 shell 中执行，因此，它们对环境的改动不会影响当前 shell；
  - “{ 命令序列; }”：命令序列括在一对花括号中。注意这里的分号，如果没有分号的话，则必须用一个换行来结束花括号中的命令序列。命令在当前 shell 中执行，花括号只是起到将命令分组的作用。组合命令的返回值就是命令序列的返回值。要特别注意的是，花括号和命令序列之间必须用空格隔开，否则会与 Bash 花括号展开的语法产生冲突！

– “((算术表达式))”：括在嵌套的两对小括号中的算术表达式。Bash 会按照算术求值规则计算出一个数来，当该数为非 0 时，整个表达式的返回值为 0，当该数为 0 时，整个表达式的返回值为 1。注意连续两个 “(” 之间和连续两个 “)” 之间不能有空格!

– “[[逻辑表达式]]”：括在嵌套的两对方括号中的逻辑表达式。对于其中的逻辑表达式，Bash 不作词法分析和路径展开，而只进行波浪号展开、参数和变量展开、算术展开、命令替换、进程替换和引号去除。如果逻辑表达式中包含比较符 “==” 或 “!=”，则比较符右边的部分会被作为模版与左边的部分进行匹配。如果匹配成功，对于 “==” 返回 0，对于 “!=” 返回 1。如果匹配失败，对于 “==” 返回 1，对于 “!=” 返回 0。如果逻辑表达式中使用了比较符 “~=”，则比较符右边的部分会被作为正则表达式与与左边的部分进行匹配，但其中用引号引起来的部分会被当作普通字符串处理。可以对逻辑表达式使用下述操作符进行运算：

  * “(逻辑表达式)”：直接返回表达式的值；这里小括号和算术运算中的小括号的作用是相同的，用于改变表达式的结合顺序；
  * “! 逻辑表达式”：表示对表达式取否；
  * “逻辑表达式1 && 逻辑表达式2”：当两个表达式都为真时才返回 0，否则返回 1；
  * “逻辑表达式1 || 逻辑表达式2”：当两个表达式都为假时才返回 1，否则返回 0；

  上面的这些操作符的结合优先度是按顺序减小的。可以看到，这些操作符事实上和 C 语言中同名操作符的含义完全一样，并且 “&&” 和 “||” 的处理也采用短路算法。

下面描述的这些结构语句也属于组合命令的范畴。因为它们的格式与前面的组合命令不太一样，所以重新开始一个列表进行介绍：

- 条件判断：

```
if 命令序列; then
  命令序列;
[ elif 命令序列; then
  命令序列; ]
 ...
[ else
  命令序列; ]
fi
```

其中方括号表示可以省略的部分。下面是一个条件语句的例子，它相当于用 C 语言的 “%03d” 格式输出一个非负整数的值：

```
if test $i -lt 10; then
    echo 00$i
elif test $i -lt 100; then
    echo 0$i
else
    echo $i
fi
```

- 循环语句：循环语句有 4 种形式。第一种形式为：

```
for 变量名 [ in 词 ]; do 命令序列; done
```

Bash 首先对 “in” 后面跟随的内容进行展开处理，然后对于展开结果中的每一个词，将变量 “变量” 的值赋为该词，然后执行一次 命令序列。当然，一般说来 命令序列 中应该用到 “变量名” 的值。请看下面的简单例子：

```
[rli@arena rli]$ days="Monday Tuesday Wednesday Thursday Friday Saturday"
[rli@arena rli]$ for day in $days; do echo "I'm working on" $day; done
```

```
I'm working on Monday
I'm working on Tuesday
I'm working on Wednesday
I'm working on Thursday
I'm working on Friday
I'm working on Saturday
[rli@arena rli]$
```

第二种循环形式为：

```
for (( 表达式1; 表达式2; 表达式3 )); do 命令序列; done
```

它类似于 C 语言中的 `for` 循环。返回值是 *命令序列* 中最后一条被执行的命令的返回值。

其他两种循环形式分别为：

```
while 命令序列; do 命令序列; done
```

```
until 命令序列; do 命令序列; done
```

这些循环的用法可用下面四个例子说明，它们完成同样的工作：

```
for i in 0 1 2 3 4; do echo $i; done
for (( i = 0; i < 5; ++i )); do echo $i; done
i=0; while test $i -lt 5; do echo $i; i=$((i+1)); done
i=0; until test $i -eq 5; do echo $i; i=$((i+1)); done
```

最后，还有一种选择形式的复合命令，句法如下：

```
case 词 in
    模版1) 命令序列1;;
    模版2) 命令序列2;;
    ...
esac
```

其含义是执行与“词”相匹配的第一个模版后面的命令。在模版中可以用“*”匹配任意子串，用“|”表示“或”运算，如“abc*|def*)”表示匹配以“abc”或“def”开头的字符串。如果模版由单独一个“*”构成，则表示它与任意字符串相匹配，通常放在最后用来定义默认处理。选择命令的例子如下：

```
for a in "$@"
do
    case "$a" in
        --*) echo "long option name:  $a" ;;
        -*)  echo "short option name: $a" ;;
        *)   echo "argument:          $a" ;;
    esac
done
```

脚本文件的第一行通常是下面的形式：

```
#!/bin/sh
```

或：

```
#!/bin/bash
```

这里，#! 后面的文件的名叫做命令解释器 (command interpreter)。如果是 /bin/bash 的话，文件中的内容用 Bash 来解释；如果是 /usr/bin/perl 的话，就用 Perl 来解释。不同的解释器所使用的语法不一样，非常严格。就算同是 shell 脚本，不同 shell 之间的格式也不尽相同。在 Linux 系统中，/bin/sh 与 /bin/bash 是等效的，因为 /bin/sh 实际上是 bin/bash 的一个符号链接。但在许多其他 UNIX 系统中 /bin/sh 和 /bin/bash 是不一样的。此外，不同系统中的 shell 程序所在的路径也不一定相同。所以，直接指定 shell 的路径比较安全一些。本书总是假设使用的 shell 是 /bin/bash。

下面是 shell 脚本的基本结构：

- 简单说来，一个 shell 脚本就是一连串命令行，再加上一些条件判断、循环、跳转等语句；
- 每当 shell 解释器在脚本中读到一个回车符的时候，就尝试执行该行命令；
- Shell 解释器会忽略空白行，以及一行开头的空白和 <Tab>；
- 回车符同样可以用 “\” 符号进行转义，转义后的回车符相当于一个空白，通常用来表示连续行，即将下一行的内容和这一行合并起来当作一行处理；
- “#” 是注释符号，从它开始至当前行尾的内容都是注释。利用注释符号，可以在脚本中插入一些注解。

Shell 脚本文件的命名没有一定规则，但经常用 .sh 做为它的扩展名。有两个方法执行一个 shell 脚本，一个方法是将文件名作为 bash 命令的参数，此时 Bash 会忽略脚本文件第一行 “#!” 后面指定的解释器。另外一个方法是直接将脚本文件名作为命令名执行。要想直接执行一个 shell 脚本文件，用户必须对它有执行权限。用文件编辑器新建立的文件一般都是没有执行权限的，需要用 chmod 命令加上。如果脚本文件不在环境变量 PATH 指定的路径中，执行时还需要加上路径名。一个比较常见的情况是，PATH 中不包含当前目录 (“.”)，这种情况下执行当前目录下的一个脚本可以在脚本文件名前面加上 “./”。

Bash 有一条内部命令 “test”，能够对一些条件进行检测。例如：“test -f ~/src/rli.cpp.bak” 检测文件 ~/src/rli.cpp.bak 是否存在并且是否是一个普通文件，若文件存在则返回 0，否则返回 1。除了 “-f” 外，test 还支持很多测试选项，见表 2.6，它们在 shell 脚本中非常有用。test 命令通常用在 if 语句中，根据检测的结果控制脚本的执行流程。

表 2.6 Bash 的文件检测操作

选项	检测的内容	选项	检测的内容
-L	存在并且是符号链接	-S	存在并且是 socket
-b	存在并且是块设备	-c	存在并且是字符设备
-d	存在并且是目录	-e	存在
-f	存在并且是普通文件	-p	存在并且是命名管道
-r	存在并且是可读的	-s	存在并且非空
-u	存在并且具有 SUID 属性	-w	存在并且可写
-x	存在并且可执行		

除了对单个文件进行检测外，test 命令也可以用来比较两个文件。例如："test 文件1 -nt 文件2" 检测文件1 是否比文件2 新。这些检测使用的比较操作符在表 2.7 中给出。

除了检测文件外，test 命令也可以进行算术比较和字符串检测。用于算术比较和字符串检测的操作符列由表 2.8 给出。事实上，可以检测的项目还有很多，用 man test 和 man bash 可以得到详尽的说明。

在脚本语言中熟练而巧妙地使用变量是体现 shell 脚本编写能力的一个重要方面。另外一个方面是能够熟练地进行文本处理，从字符串中间抽取想要的信息，对字符串进行变换等等。

Bash 可以通过 "$((...))" 展开进行算术表达式运算。这种方式只能进行整数计算，而且，对溢出也不做检测。在这些算术运算

表 2.7 Bash 比较两个文件的操作

操作符	代表意思
-nt	Newer Than：第一个文件比第二个文件新。
-ot	Older Than：第一个文件比第二个文件旧。
-ef	Equal File：两个文件实际上是同一文件 (如符号链接)。

表 2.8 Bash 的算术比较及字符串检测操作

操作符	检测的内容	操作符	检测的内容	操作符	检测的内容
-gt	大于	-ge	大于或等于	-lt	小于
-le	小于或等于	-eq	等于	-ne	不等于
<	小于	>	大于	=	等于
!=	不等于	-a	“与”	-o	“或”
-z	空字符串	-n	非空字符串		

中使用的操作符和 C 语言中完全一样，它们列在表 2.9 中。这些操作符的优先级在表中按降序排列。Shell 的变量可以作为算术表达式中的操作数。在求值以前，会优先进行参数展开。还有一点要指出的是，在 “$((...))” 的表达式中引用 shell 变量时可以省略变量前面的 “$”，例如：

```
[rli@arena rli]$ a=123
[rli@arena rli]$ echo $((a*2))
246
[rli@arena rli]$
```

算术表达式中的常数当以 0 开头时被解释为八进制数，但如果是以 0X 或者 0x 开头，则被解释为 16 进制数。除非进行特别指定，其他数都被解释成 10 进制数。进行逻辑运算时与 C 语言一样用非 0 表示“真”，0 表示“假”。要注意的是算术表达式中的真假值和命令返回码中的真假值正好是相反的。

如果想在 shell 中进行浮点数运算，可以借助于外部程序 bc 来进行，例如：

```
[rli@arena tmp]$ a=2
[rli@arena tmp]$ b=$(echo $a/3 | bc -l)
[rli@arena tmp]$ echo $b
.66666666666666666666
[rli@arena tmp]$ b=$(echo "scale=50; sqrt($a)" | bc -l)
```

表 2.9 Bash 的算术表达式

操作符	操作内容
变量名++，变量名--	变量后作用自加 (减) 1
++变量名，--变量名	变量先作用自加 (减) 1
-，+	一元正负操作符
!，~	逻辑否和位操作取反
**	指数 (Fortran 格式)
*，/，%	乘，除和模余量
+，-	二元加减运算
<<，>>	位操作左右移位
<=，>=，<，>	比较操作符
==，!=	等于和不等于
&	位操作与
^	位操作异或
|	位操作或
&&	逻辑与
||	逻辑或
a?b:c	条件求值
=，*=，/=，%=，+=，-=，<<=，>>=，&=，^=，|=	赋值
a,b	逗号

```
[rli@arena tmp]$ echo $b
1.41421356237309504880168872420969807856967187537694
```

注意上面倒数第三行用了双引号，知道为什么吗？如果用单引号结果会怎样？

8. 命令行参数

当运行一个 Bash 脚本或函数 (关于 shell 函数的的说明参看下节) 的时候，命令行上脚本或函数名后面给出的参数会被传递给脚

本程序或函数。Bash 根据分隔字符 (由环境变量 `IFS` 定义，通常是空格) 将参数分割成一个个的词，每个词作为一个参数，从 1 开始编号。如此分割后的参数称为位置参数。如果一个参数本身包含分隔字符，但又不希望 Bash 将其当作两个参数处理，可以在命令行上用双引号或单引号将它括起来，或是用 "\" 对分隔字符进行转义。Bash 脚本中用 "`$1`"、"`$2`" 等引用位置参数，其中 "`$1`" 表示第一个参数，"`$2`" 表示第二个参数，依此类推。引用一个不存在的参数得到的是空字符串。此外，"`$0`" 表示 Bash 脚本本身的文件名，"`$#`" 表示位置参数的个数，"`$*`" 或 "`$@`" 表示所有位置参数 (list)。作为一个特例，"`"$@"`" (注意这里的双引号) 也表示全部位置参数，它被展开后，每个参数会括在一对双引号中，可用于处理参数中包含分隔字符的情形。命令 "`shift [`n`]`" 对位置参数进行"平移"操作，表示删除前 n 个位置参数，省略 n 时删除第一个位置参数。

位置参数除了可以从命令行获得外，也可以用 Bash 的内部命令 "`set`" 来指定或改变，这里不做介绍。

9. Shell 函数

Bash 允许用户定义函数。一个函数一旦定义，可以像其他内部或外部命令一样使用。Bash 的函数定义采用下面形式：

```
函数名 () {
  ... ...
  函数体
  ... ...
}
```

Bash 函数中也可以使用位置参数，它们就是调用函数时给出的参数。函数中用 "`return [返回码]`" 返回，其中 "返回码" 的含义和其他命令的返回码是一样的，0 表示成功。在 Bash 函数中，还可以用 `local` 命令声明局部变量。

下面是一个比较复杂的 Bash 函数实例，它接受一个路径名作为参数，对路径名进行规范化处理，将一个不包含符号链接、“.” 和 “..” 的绝对路径名输出到标准输出上。这个例子中用到了文本处理工具 sed 对路径名进行转换，有关说明可参看 133 页“sed 和 awk”。

文件名: code/linux/canonicalize.sh

```
canonicalize () {
    # 标准化路径名 "$1" (消除其中的 ., .. 和符号链接并转换成绝对路径)
    local path dir d0 d target          # 局部变量

    if [ $# -ne 1 ]; then
        echo 1>&2 "Usage: canonicalize pathname"
        return 1                        # 返回非 0 表示错误
    fi

    path="$1"                           # 待处理的路径名
    if test "${path#/}" = "$path"; then path="`pwd`/$path"; fi
                                        # 转成绝对路径

    # 第一遍扫描: 消去符号链接
    dir=""
    # 对文件名中的每级目录依次循环 (为处理文件名中的空格，将它们用 '#' 代替)
    for d in `echo "$path" | sed -e 's/ /#/g' -e 's/\// /g'`; do
        d="`echo $d|sed -e 's/#/ /g'`"  # 将 '#' 变回空格
        d0="$dir"                       # 当前目录
        dir="$dir/$d"                   # 下一层目录
        while test -h "$dir"; do
            # 替换符号链接
            target=`/bin/ls -l "$dir" | awk -F' -> ' '{print $2}'`
            if test "${target#/}" != "$target"; then
                dir="$target"           # 目标是绝对路径名
            else
                dir="$d0/$target"       # 目标是相对路径名
            fi
            d0="${dir%/${dir##*/}}"     # 新的当前目录
```

```
30          done
31      done
32      path="$dir"
33
34      # 第二遍扫描: 消去 "." 和 ".."
35      dir=""
36      for d in `echo "$path" | sed -e 's/ /#/g' -e 's/\// /g'`; do
37          d="`echo $d | sed -e 's/#/ /g'`"
38          case "$d" in
39              ..) dir="${dir%/[^/]*}" ;;
40               .) continue ;;
41               *) dir="$dir/$d" ;;
42          esac
43      done
44      path="$dir"                         # path 中包含最终处理结果
45
46      echo $path                          # 新路径名 -> stdout
47  }
```

若想进一步了解函数的使用，可以参考 `/etc/init.d/functions` 脚本，其中定义了一组函数，它们是目录 `/etc/init.d/` 中的脚本所共用的 (每个脚本会在开头用 "`. /etc/init.d/functions`" 来导入这些函数)。用户也可以将函数定义放在 Bash 初始化文件中 (如 `/etc/bashrc`、`~/.bashrc` 等)，这样定义的函数可以直接在命令行中使用，与命令别名的功能类似，但可以完成比后者复杂得多的处理。

Shell 的强大编程功能是 Linux/UNIX 系统的一大特色，结合系统中大量的文本编辑、处理程序，如后面将要介绍的 `sed`、`awk` 等，以及重定向和管道，可以快速、方便地完成各种各样的文件处理和系统管理任务。当处理文本文件时，shell 脚本往往比用高级语言编写程序更加方便。Linux 系统中包含了大量用 shell 脚本编写的程序。例如，`/etc/init.d`、`/etc/rc*.d` 等目录下的系统管理程序，它们构成了系统启动、运行的基本支撑。许多用户程序也是 shell 脚本。例

如，用命令：

```
file /usr/bin/* | grep 'shell script'
```

可以找出大量用 shell 脚本编写的用户命令。学习 shell 编程最好的方法是参考、阅读 Linux 系统中的这些脚本文件。

2.2.3 文本文件处理

Linux 中有很多方便的工具用来处理文本，其中一些的功能非常强大，自身甚至构成一个完备的编程语言。它们可以帮助用户从文本中快速抽取信息，对文本进行自动编辑。运用这些工具，可以快速方便地完成许多复杂的任务。在 shell 脚本中，经常需要对文本进行处理，所以文本处理是 shell 编程的一个基本技能。这里只是简单介绍几个工具，如何灵活地应用它们去完成自己的工作需要通过实际使用经验去摸索。

1. 正则表达式

Linux 的很多工具都用到正则表达式。正则表达式是一种特殊格式的模版，能够和其他字符串进行灵活的匹配，从而迅速完成一些复杂的文本处理工作。关于正则表达式的具体的使用方法将在后面结合支持正则表达式的工具介绍。这里先给出正则表达式的语法及匹配规则。在 Linux 中，可以用 “`man 7 regex`” 得到 POSIX 1003.2 正则表达式语法的在线文档。

需要指出的是，各个应用软件中使用的正则表达式可能会有一些微小差别，主要在于一些元字符的使用上。一些软件，包括一些版本的 `sed`、`vi`、`awk` 等，依然使用老的正则表达式格式，它们将 “+”、“(”、“)”、“|” 等当作普通字符处理，必须在它们前面加上 “\” 才具有下面介绍的元字符的功能 (例如，需要将正则表达式 “`(a|b)+`” 写成 “`\(a\|b\)\+`” 的形式)。请参看相应的应用软件的文档。应付这类情况的一个简单的方法是，如果某个软件中某些元字符起不到

应有的作用，则很可能该软件用的是老的正则表达式格式，因此可以试试在这些元字符前面加上 “\”。

一个正则表达式就是一个字符串。字符串中的字符分成两类，一类是普通字符，另外一类是特殊字符，叫做元字符。元字符有下面一些：

```
* ? . ^ [ ] { } \ + | ( )
```

这些元字符的基本涵义如下：

- “.”：匹配除换行符外的任意单个字符。例如，正则表达式 “a.b” 会与第一个字符为 “a”、第三个字符为 “b”、由三个字符构成的字符串匹配；
- “*”：作为后缀使用，表示将其前面匹配的字符串连续地重复 0 次到任意多次。比如 “o*” 会匹配一连串任意多个 “o” (包括空字符串)。要注意的是，它只是对前面的最小匹配进行重复，比如 “fo*” 会匹配 “f” 后面紧跟着零个或者任意多个 “o” 的情况，而不是零个或者任意多个 “fo”。用 “*” 进行匹配时，会首先尽量匹配多的内容，直到不能匹配为止，然后，会根据跟随其后的表达式的匹配需要，回吐一部分已经匹配的内容；
- “+”：和 “*” 的含义相近，但要求前面的表达式至少匹配上一次。例如 “ca+r” 会与 “car”，“caaaar” 匹配，但是却不会与 “cr” 匹配；
- “?”：将前面的表达式匹配一次或零次。例如，“ca?r” 只与 “car” 或 “cr” 匹配；
- “{n}’：这里 n 代表一个整数，表示将前面匹配的内容连续匹配 n 次。例如，“x{4}” 与 “xxxx” 匹配；

- “{n,m}”：表示将前面的内容连续至少匹配 n 次，至多匹配 m 次。省略 m 时表示 m 为无穷。容易看出，“{0,1}” 和 “?” 是等价的，“{0,}” 和 “*” 是等价的，而 “{1,}” 和 “+” 是等价的；

- “[...]”：表示一个字符集，由一个左方括号开始，一个右方括号结束，中间列出字符集中的字符。在最简单的情况下，括号中的字符集就是匹配的字符。例如，“[ad]” 与单个字符 “a” 或 “d” 匹配，“[ad]*” 与全部由 “a” 或 “d” 构成的字符串匹配，包括空字符串，而 “c[ad]*r” 则与 “cr”、“car”、“cdr”、“caddaar” 等字符串匹配。在方括号中的字符集合中可以用区间表示的方法，如 “[a-z]” 与任何小写字母匹配。区间表示方法可以和单个字符混用，例如，“[a-z$%.]” 与所有小写字母、字符 “$”、“%” 以及句点号相匹配。注意，除了 “]”、“-” 和 “^” 这三个字符以外，所有其他元字符在方括号中都被当成普通字符处理。字符 “]”、“-” 和 “^” 也可以包含在方括号中，但它们出现的位置有所限制：“]” 必须作为第一个字符，“-” 必须作为第一个或者最后一个字符，而 “^” 则不能作为第一个字符；

- “[^ ...]”：匹配的字符集为方括号中的字符集的补集，即与任何不属于方括号中的字符集的字符匹配。“^” 只有作为方括号中第一个字符出现的时候才表示取补集的意思，在其他位置被当成一个普通字符处理；

- “*?”，“+?”，“??”：通常情况下，“*”，“+” 和 “?” 会匹配最长的字符串，但如果在它们后面加上一个问号，它们则会匹配最短的字符串。例如，用 “a[bc]*c” 和 “a[bc]*?c” 与字符串 “abcbcbc” 进行匹配时，前者会与整个字符串匹配，而后者只与开头的 “abc” 匹配；

- “^”：如果不在方括号中，则表示一个行的行首，也就是说，紧

跟在其后的表达式必须匹配一行开头的字符串；

- “$”：和 “^” 相对应，表示一行的结尾；
- “\”：用于将元字符进行转义。比如想匹配一个 “[” 的话必须使用 “\[”，想匹配一个 “$” 的话必须使用 “\$”；
- “|”：表示“或”的意思，用它将一组正则表达式连接起来构成一个新的正则表达式，一个字符串只要与这些正则表达式中的任何一个匹配，便与整个正则表达式匹配。例如，表达式 “ab|cd” 同时匹配 “ab” 和 “cd”；
- “(...)”：其中括号中是一个正则表达式。圆括号在正则表达式中的作用与普通表达式中类似，用来将一组正则表达式组合在一起，当作一个整体看待。它们的用途主要有三个。一是指定后缀 “*”、“+”、“?” 和 “{n}” 的作用范围，如 “(abc)*” 表示与字符串 “abc” 重复 0 次或多次相匹配，如：空字符串、“abc”、“abcabc” 等；二是帮助界定 “|” 的作用对象；三是在进行字符串替换时用来界定 “\1”、“\2” 等所代表的子串，参看下节关于 sed 的说明。

2. sed **和** awk

sed 是所谓的流编辑器 (stream editor)。它所完成的工作就是对一个输入的文本流进行一定的转换，就象是对文件进行了编辑一样。sed 和其他编辑器的一个重要不同之处在于它能够在管道上进行操作，是非交互式的，处理的效率很高。

sed 在工作时接受一个输入流，并同时需要获得对流进行操作的命令脚本。输入流可以通过管道、命令行上的文件名、重定向或者是一个字符串等方式获得，而命令脚本是通过命令行的 -e 或者 -f 选项获得的。如果是 -e 选项，那么后面紧跟的字符串就是脚本，如

果是 -f 选项，那么后面是一个文件名，该文件中的文本就是命令脚本。记录着 sed 命令的脚本文件叫做 sed 程序，sed 程序中的内容是一条一条的 sed 命令。sed 从输入流中读取一段数据，存储在缓冲区中，然后依次使用命令脚本中的每条命令对缓冲区中的内容进行处理，最后将得到的结果输出到标准输出。通过重定向，可以将 sed 运行的结果保存到文件中。

下面看一个简单的例子：

```
sed -e '20,$s/c\(a*\)r/b\1t/g' file.txt
```

这里 -e 选项后面紧跟的就是命令脚本。这条命令的意思是这样的：对于文本中从第 20 行到最后一行进行替换操作，将和正则表达式“ca*r” 匹配的字符串的第一个字符 “c” 替换为 “b”，最后一个字符 “r” 替换为 “t”，中间部分保持不变。使用 “\(” 和 “\)” 括起来的部分被赋予了一个名字叫做 “\1”，可以在替换的字符串中对其进行引用。如果正则表达式中有多个这样的部分，则分别用 “\1”，“\2” 等表示。“\0” 表示整个能匹配上的表达式。这条命令由下面几部分构成：首先指定操作的范围，这里是 “20,$”，表示从第二十行到最后一行；然后是命令 “s”，表示进行替换，它是 sed 中最常用的命令，其他命令还有删除 (d)、打印 (p) 等，由于不常用到，这里不做介绍；接着给出被替换的字符串 (正则表达式) 和替换的结果；最后的 “g” 是一个操作标识，意为 global，表示替换所有匹配的字符串 (没有 “g” 时只替换每行中第一个匹配的字符串)，其他的标识请参考 sed 的在线文档。“s” 命令的一般形式为：

```
[范围描述]s/正则表达式/替换结果/[操作标识]
```

其中方括号表示可以省略的部分 (即“范围描述”和“操作标识”，省略前者时表示操作范围为所有行)。

在替换命令中，字符 “/” 用来分隔正则表达式和替换结果。如果正则表达式或替换结果中包含字符 “/”，则需要在它的前面加 “\”

进行转义。除了 “/” 之外，sed 也允许用其他字符做为分隔符，例如，下例中的两条命令都将字符串 “/usr/share/texmf/tex” 中的 “/” 替换成空格，第二条命令中用字符 “!” 做分隔符：

```
[rli@arena rli]$ echo /usr/share/texmf/tex | sed -e 's/\// /g'
 usr share texmf tex
[rli@arena rli]$ echo /usr/share/texmf/tex | sed -e 's!/! !g'
 usr share texmf tex
```

下面再举一个例子。103 页上给出了一个从命令 ifconfig 的输出中提取 IP 地址的例子，当时使用了 tr 和 cut 命令。用下面的 sed 命令可以完成同样的工作：

```
$ /sbin/ifconfig eth0 | grep "Bcast:" | sed -e 's/^.*addr:\| *Bcast:.*$//g'
162.105.42.214
```

这里是将 IP 地址前后的字符串删除，只留下 IP 地址。还可以用下面的方法：

```
$ /sbin/ifconfig eth0 | grep "Bcast:" | sed -e 's/^.*addr:\([0-9.]*\).*$/\1/'
162.105.42.214
```

请仔细研究这些例子，并将它们做一些变化进行练习。

程序 awk 的名称来源于设计该程序的三个人的名字。在 Linux 系统中，一般使用的是 gawk (GNU awk) 或 mawk，/usr/bin/awk 实际上是它们的一个符号链接。awk 是一种解释性的编程语言，其语法是比较复杂的，当然功能也非常强大。它的基本工作方式和 sed 相近，对一个流进行扫描，遇到特定的字符串时执行特定的操作，但是它的功能比 sed 要丰富得多。和 sed 一样，awk 在运行的时候，需要获得两个输入信息，一个是要处理的流，另外一个是匹配规则与处理命令。要处理的流可以是标准输入，也可以是用命令行参数指定的文件。处理命令可以直接在命令行中给出，也可以写在一个脚本文件中。awk 的处理命令的格式为：

```
/正则表达式/ {处理方法}
```

其中处理方法部分的语法和 C 语言的语法类似，但是 awk 有自己的一套函数，可以参考 awk 的在线文档了解有关这些函数的细节。除了函数之外，awk 还有一些预定义的基本变量，在进行字符串处理时经常用到。awk 处理时假设文本是分列的，它用变量 “$$n$$”，n 是一个整数，表示第 n 列，$0 表示整个行。下面用几个简单例子来展示 awk 的用法。

第一个例子直接在命令行中输入处理命令：

```
awk '/^2.*/ {if ($2 != $3) print NR, $0;}' file.txt
```

它的处理过程是：寻找以字符 “2” 开头的行，对其第二列和第三列进行比较，如果第二列和第三列不一样，就将这一行的行号和内容打印出来。其中变量 NR 代表记录号，也就是行号。

第二个例子稍稍复杂些，处理命令写在一个脚本文件中，处理内容包括：

(1) 在文本的开头和结尾加入一部分额外的格式信息；

(2) 对文本中的内容进行一些统计；

(3) 对文本中的内容进行一些选择和替换；

下面就是这个例子的处理命令：

```
BEGIN { n_this = 0
        total_score = 0
        print "We can add some text at the beginning." }
/^that/ {}
/this/ { ++ n_this; print $0; }
/unknown?/ { if ($3~/[0-9]+/) total_score += $3;
             print $0;
           }
```

```
END { printf "total \%d \"this\"s found in the file.", n_this
      printf "total score is \%d.", total_score
      print "We can add some text at the end."
    }
```

使用 BEGIN 和 END 的子句会分别作用在开头和结尾，上述命令统计整个文本中单词 this 出现的次数，删除第一个单词为 that 的行，如果行中包含有 unknow 或者 unknown 的话，检查这行的第三列是否是数字，如果是数字，那么将它加到变量 total_score 上。当然，这段代码没有任何实际意义，只是显示一下 awk 命令的书写格式和处理。事实上，通过写一段 awk 的指令，可以实现很多其他应用程序的功能，包括 cut、split、egrep、tee、uniq、wc 和 id 等，在 gawk 的在线文档中可以看到这些例子。

下面再看看如何用 awk 来实现前面提取 IP 地址的例子：

```
[rli@arena rli]$ /sbin/ifconfig eth0 \
| awk -F: '/Bcast:/ {print $2}' | awk '{print $1}'
162.105.42.214
```

其中第一个 awk 命令以 “:” 为分隔符 (通过 “-F” 选项指定)，将包含字符串 “Bcast:” 的行的第二列显示出来，得到的结果是：

```
162.105.42.214  Bcast
```

而第二个 awk 命令则用 (默认的) 空格作分隔符，将第一列显示出来。

除了 sed 和 awk，以及前面提到的 grep 外，后面介绍的编辑器 vi 和 Emacs 也支持正则表达式。

3. diff 和 patch

diff 用来将两个文件的不同之处找出来，它的基本工作方式是按行进行操作的，所以一般用于文本文件。其命令的基本形式为

```
diff [选项] 文件1 文件2
```

它的两个参数可以是文件和目录，当一个参数是目录的时候，会比较该目录下与另外一个参数同名的文件，如果两个参数都是目录，会对两个目录下的所有同名文件进行比较。如果使用了选项 -r，则会对两个目录下的子目录进行递归比较。事实上，可用的选项是很多的，这些选项的主要目的有两个，一是指定比较方式，二是指定输出格式。比较常用的输出格式选项是 -u，下面是一个例子：

```
--- gnome-python-1.4.1/configure.in.orig        Wed Mar 27 22:58:46 2002
+++ gnome-python-1.4.1/configure.in     Wed Mar 27 22:59:58 2002
@@ -28,7 +28,9 @@
  build_gtkhtml=no)
 if test "x$build_gtkhtml" != xno; then
   GTKHTML_LIBS=`$GNOME_CONFIG --libs gtkhtml`
+  GTKHTML_CFLAGS=`$GNOME_CONFIG --cflags gtkhtml`
   AC_SUBST(GTKHTML_LIBS)
+  AC_SUBST(GTKHTML_CFLAGS)
 fi
 AM_CONDITIONAL(BUILD_GTKHTML, test "x$build_gtkhtml" != xno)
```

它是一个软件中某个文件的补丁文件，其中首先指出这是哪两个文件的不同，然后指出不同发生在第一个文件的第 28 行后面的 7 行和第二个文件的第 28 行后面的 9 行之间，不同之处是第二个文件多出来两行，即以 “+” 开头的两行。diff 的输出格式还有其他许多，各自有着不同的用途，例如，给行编辑命令 ed 使用。diff 的输出可以作为 patch 程序的输入。如同字面上一样，patch 就是打补丁的意思。例如，用户从网上下载了一个比较大的软件，后来该软件有一个比较小的修改，此时当然希望仅仅下载修改的部分，而不用重新下载整个软件。软件作者通常会提供用 diff 命令得到的补丁文件，只要下载补丁文件，然后用命令 patch 就可以自动更新原先下载的软件。patch 的命令行形式有两种，第一种是：

```
patch [选项] 原始文件 补丁文件
```

其中“补丁文件”便是 diff 命令产生的输出。对于从网上下载的软件补丁，经常使用的是第二种形式：

```
patch -p数字 <补丁文件
```

它可以自动更新整个目录中修改过的文件。其中选项“-p数字”表示寻找要修改的文件时去掉补丁文件中指定的路径名开头的几层目录，“数字”是一个整数，用来指定要去除的目录层数。运行 patch 后，目录中的相关文件就会变成更新后的版本。

利用 diff 和 patch 可以对文本文件进行非常灵活的操作，对于文本文件的修改、维护及协同操作非常有用。许多版本维护软件，如后面要介绍的 CVS (Concurrent Versions System)、Subversion 等，都是基于 diff 和 patch 的原理工作的，它们对于大型软件的开发维护是必不可少的工具。例如，本书排版采用的是 LaTeX 系统，其编写、排版、修订过程就是由几位作者在网上借助 CVS 协同完成的。

2.3 程序开发环境

本节介绍 Linux 下面的程序开发。与 Windows 系统不同，在 UNIX 或 Linux 系统中，程序开发环境可以说是与生俱来的。目前使用最广泛的 C 语言，就是在 UNIX 系统的设计过程中发明的。经过数十年的发展，UNIX 系统中形成了一整套成熟的程序开发与调试环境，产生了大量的程序开发工具。Linux 系统中，这些工具得到了继承及进一步的发展。因此，在 Linux 系统下开发程序是非常方便的。

2.3.1 第一个程序 (C 程序)

首先通过一个简单例子介绍 Linux 系统下进行 C 程序开发的过程和工具。这里以大家非常熟悉的 “Hello，the world!” 程序为例，几乎每本 C 语言教程都将它作为第一个例子。

1. 程序的编辑

要写一个程序，第一个问题当然是怎么编辑它。Linux 系统下面有许多类似于 Windows 下的 Visual Studio 或 Fortran Powerstation 的集成开发环境，如 Kdevelop、Anjuta IDE 等等，甚至像 vim、Emacs 等编辑器，经过适当配置后，也可形成功能强大的集成开发环境。但这里主要介绍 UNIX 中传统的程序开发方式，即基于命令行的程序开发工具。掌握它们，有助于深入了解程序的编译、链接过程，对于使用其他集成开发环境也会大有帮助。

首先介绍一个非常古老而又富有生命力的、适用于字符界面的编辑器：vi。它是 UNIX 下最通用的文本编辑器。运行命令

```
[rli@arena ~]$ vi
```

便进入 vi 编辑器。整个终端变成了 vi 的工作窗口。vi 在工作时有四种模式，刚刚启动时一般处于命令模式。其他三种模式分别是插入模式、替换模式和附加模式。在命令模式下，按 k、j、h 或 l 键可以上下左右移动光标。为了输入程序，按一下 i 键，就进入了插入模式。在插入模式下，键入的字符被插入到所编辑的文件的当前位置。现在输入下面的内容：

```
/**
 * @file   hello_world.c
 * @author Ruo Li
 * @date   Thu Feb 19 13:14:21 2004
 *
 * @brief  our first C program under Linux
```

```
 */

#include <stdio.h>

int main(void)
{
  printf("Hello, the world!\n");
  return 0;
}
```

输入完毕后，按 Esc 键，就回到了命令模式。在命令模式下键入“:w hello_world.c”后再回车，便将上面所输入的内容存到了文件 hello_world.c 中。键入 :q 后再回车便可从 vi 中退出。请注意上面命令中的冒号，在命令模式下，键入冒号就表示要输入一条命令，光标会自动跑到屏幕下方的命令区等待用户输入命令，按回车便执行命令，然后光标又回到原来的位置。

在 vi 中，Esc 用于离开其他模式返回到命令模式，也可用来放弃正在输入的命令。如果用户不清楚自己所处的模式，反复按 Esc 键，最终将会回到命令模式。

下面简单地介绍一下 vi 中的基本命令，这些命令只有通过日常的练习、使用才能熟练地掌握。

- 移动光标：在命令状态下，移动光标的命令主要有下面一些。
  - h、j、k、l 用于上下左右移动光标，如果终端类型配置正确的话也可以用四个箭头键。
  - w 和 b 分别将光标向前和向后移动一个单词；
  - ^ 和 $ 分别将光标移到当前行的开头和结尾；
  - Ctrl-D 和 Ctrl-U 分别将光标向前和向后移动半个屏幕；
  - Ctrl-F 和 Ctrl-B 分别将光标向前和向后移动一个屏幕(翻页)，如果终端类型配置正确的话也可以用 <PgUp> 和

<PgDn> 键；

-) 和 (分别将光标向前和向后移动一个句子；
- } 和 { 分别将光标向前和向后移动一个段落；
- H，M 和 L 分别将光标移动到屏幕的最上面、中间和最下面一行上；
- nG 将光标移动到第 n 行上，其中 n 是一个整数；

- 输入文本：在命令状态下按 i 键便进入到插入状态，此时输入的内容被插入到当前编辑的内容中，插入状态下可以按 Esc 键返回到命令状态；

- 删除文本：在命令状态下，x 删除一个字符，dw 删除一个词，d$ 删除当前位置到行尾的所有内容，dd 删除一行；

- 替换文本：在命令状态下按 R 键便进入替换状态，此时新键入的字符会替换光标下原有的字符，按 Esc 键可以回到命令状态；如果在命令状态下按 r 键则会在替换当前光标下的一个字符后自动返回到命令状态；

- 查找和替换文本：在命令状态下按 / 键可以输入一个正则表达式来查找与它匹配的字符串；按 : 键，然后用 s 命令可以进行字符串的替换，命令格式与 sed 的 “s” 命令完全一样。需要注意的是，如同前面指出过的，不同版本的正则表达式在元字符处理上可能会略有不同；

- 拷贝和粘贴文本：当使用删除命令删除了一部分内容以后，被删除的文本被存储在缓冲区中，可以用 p 键将缓冲区中的文本粘贴到当前光标位置的后面；也用 y 命令将指定的文本放到缓冲区中，供后面的粘贴操作使用，y 来源于英文单词 yank，比

如 yw 将会拷贝一个词到缓冲区，yy 会拷贝整个一行到缓冲区，等等；

- 撤销上一次的操作：在命令状态下按 u 键可以撤销上一次的操作。

Linux 系统中提供了 vi 程序的一个改进版本 Vi IMproved，其命令名为 vim，它除了支持传统 vi 的所有命令外，还提供了许多非常有用的增强功能，包括关键字的自动高亮显示、灵活的用户定制能力和强大的脚本功能。此外，vim 还提供了一个图形界面的版本 gvim 供选择使用。

Linux 下面还有一个被称为是“第一个人工生命”开发环境的 Emacs。它是一个功能强大、配置灵活的编辑器，在一些其他附加软件的帮助下，可以构成一个优美的集成开发环境，支持 C、Fortran、Java、PHP、TEX、shell 脚本等等，甚至 Matlab 的开发。本章的 TEX 排版源码就是用 Emacs 写的。图 2.2 是用 Emacs 编辑本章时的屏幕截图。 Emacs 分别提供字符和图形两种版本，可以直接在命令行敲 emacs 启动。

从 Emacs 中退出的命令是 Ctrl-X Ctrl-C。许多第一次试用 Emacs 的用户往往会因为怎么也关不掉它而火冒三丈，从此失去了了解它的机会！关于 Emacs 的使用这里不做介绍，网络上有大量讨论如何设置和使用 Emacs 的文章，也可以阅读 Emacs 自带的使用说明。关于 Emacs 有一句名言：“学习使用 Emacs 的最好的方法，就是一天到晚使用 Emacs；学会了使用 Emacs 的结果也是一天到晚使用 Emacs。”

2. 程序的编译和运行

在 Linux 下，最常用的 C 编译器是 GNU C 编译器，其命令名为 gcc。只要运行

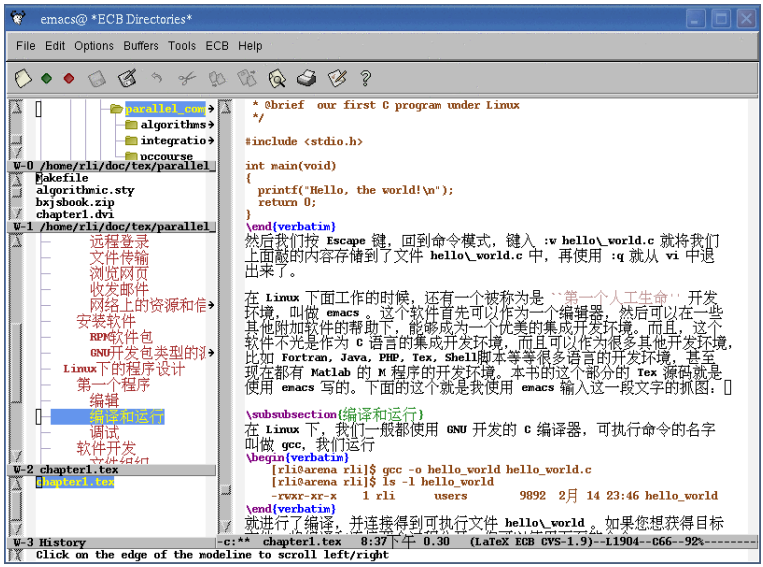


图 2.2 Emacs 屏幕截图

```
[rli@arena rli]$ gcc -o hello_world hello_world.c
[rli@arena rli]$ ls -l hello_world
-rwxr-xr-x    1 rli      users        9892  2月 14 23:46 hello_world
```

就完成了编译，并链接得到可执行文件 `hello_world`。如果想生成目标文件，将编译和链接两个过程分开，可以使用下面的命令：

```
[rli@arena rli]$ gcc -c hello_world.c
[rli@arena rli]$ ls -l hello_world.o
-rw-r--r--    1 rli      users         816  2月 14 23:47 hello_world.o
[rli@arena rli]$ gcc -o hello_world hello_world.o
```

其中文件 `hello_world.o` 就是目标文件。上面第一步使用 `gcc -c` 对源程序进行编译，第二步使用 `gcc` 命令将目标文件链接成为可执行文件。如果想运行编译产生的程序，只需键入：

```
[rli@arena rli]$ ./hello_world
Hello, the world!
```

注意这里在文件名前面加上了“./”，即写成“./hello_world”的形式。如果省略路径名，直接输入“hello_world”，则 shell 会依次搜索由环境变量 PATH 指定的目录去寻找一个名为“hello_world”的可执行文件，并执行它所找到的第一个文件，如果当前目录不包含在 PATH 中，则 shell 会抱怨“找不到文件`hello_world'”而拒绝执行命令。更糟糕的是，如果 PATH 变量中的另一个目录下恰好也有一个名为“hello_world”的可执行文件，则所执行的不是刚刚编译生成的程序，而是另外一个目录中的程序！调试程序时，往往会被这类问题搞得晕头转向。因此，在当前目录的可执行文件名前面加上“./”是一个良好的习惯，它可以有效地避免这类问题，还可以避免程序名与命令别名、shell 函数、shell 内部命令等发生冲突。

GNU 的编译器和常用的库文件，一般在安装系统的时候都会装上。GNU 所附带的编译器包括 C 和 C++、Fortran 77、JAVA、Object C、ADA 几种。现在 GNU 的 Fortran 95 编译器项目也已经接近最后完成，已经随 GCC 4.0 发布。在进行数值计算时，一般会选择使用 C、Fortran 或者 C++ 作为程序设计语言。

如果是 Intel Itanium CPU 的话，由 Intel 开发的编译器具有比较好的编译效果。它可以从 Intel 的网站上下载，并可免费获得科研教育的使用授权。

上面的编译过程是在命令行上进行的。用这样的方式，可以根据实际需要方便地选择最合适的编译选项和参数。下面简单介绍一下比较常用的编译选项，这些编译选项的详细说明请参考在线文档。前面已经看到的编译选项有“-c”，它阻止编译器进行链接，还有“-o”，用于指定输出文件的名称。其他比较重要的选项还有：

- -I目录名 用来指定头文件的搜索路径；

- -L目录名 用来指定库文件的搜索路径；
- -l库名 用来指定库文件，加载的真实库文件名为 “lib库名.a” 或 “lib库名.so”，其中以 “.a” 为扩展名的是静态链接库，以 “.so” 为扩展名的是动态链接库或共享库，后者在程序运行时才实际加载。例如，数学库函数库的文件名为 libm.a，如果需要与数学库进行链接，应该使用选项 “-lm”；
- -g 该选项使得供程序调试用的关于源程序的信息被写入到目标文件和可执行文件中；
- -On 该选项指定编译的优化级别。-O0 表示不优化，要进行优化的话，可以用 -O1，-O2，-O3 等。通常，默认的优化级别是 -O2，它既能达到比较高的执行速度，又比较安全。-O3 以上的优化会进行一些比较激进的处理，有时会导致编译的结果不对；
- -W 这是一族用于控制编译警告信息的选项，后面跟随不同的参数选择不同类型的警告，比较常用的是 “-Wall”，它开启大部分警告信息，这些信息往往有助于在编译时发现代码中的一些潜在错误；
- -f参数 这是一族用于控制代码生成的选项，请参考在线文档了解这些选项的及用途；

编译器选项中，有一些是和机器的硬件架构相关的，它们使得编译器可以针对不同的硬件对代码进行优化，达到最好的执行效果。

下面是比较常用的编译命令行：

```
gcc -Wall -O3 -g -I/opt/include -o bin_file source_file.c -L/opt/lib -lgsl
```

上例中，编译器会在目录 /opt/include 下寻找头文件，链接器会在 /opt/lib 下面寻找库文件，要链接的库名为 gsl，如果是静态

库，指的是文件 libgsl.a，如果是动态库，指的是文件 libgsl.so。当然，除了指定的 /opt/include 和 /opt/lib 目录外，编译器还会搜索默认的系统头文件和库的目录，如 /usr/include 和 /usr/lib。在 Linux 系统中，/usr/local/include 和 /usr/local/lib 通常也在编译器的默认搜索路径中。

3. 程序的调试

在 Linux 下有很多方便的程序调试工具，其中最基本的调试器是 gdb。gdb 是一个功能非常强大的调试器，下面介绍一下它的简单用法。当需要调试程序时，应该在编译时加上调试选项 “-g”，它使得可执行文件中包含关于源文件的信息。例如，

```
[rli@arena rli]$ gcc -g hello_world.c -o hello_world
[rli@arena rli]$ ls -l hello_world
-rwxr-xr-x    1 rli      users       14196  2月 15 20:39 hello_world
```

可以看到，得到的可执行文件的大小前面已经不一样了，因为其中增加了有关源文件的信息。运行命令

```
[rli@arena rli]$ gdb hello_world
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type ``show copying'' to see the conditions.
There is absolutely no warranty for GDB.  Type ``show warranty'' for details.
This GDB was configured as ``i386-redhat-linux''...
(gdb)
```

就可以开始进行调试了。可以看到 gdb 输出了一些关于它自己的信息，然后给出了一个提示符 (gdb)，等待用户输入调试命令。程序调试的主要工作一般包括：了解程序现场的状态、设置断点、观察变量、控制运行等。在 gdb 下输入 help 命令可以得到帮助信息。几个

常用的命令见表 2.10。每个命令都有一个简写的形式，方便用户输入。 有了这些命令，便可以完成比较基本的程序调试工作。gdb 还有很多其他调试功能，可以在以后的使用中慢慢去掌握。进行程序调试的步骤一般是：先进入 gdb；然后用命令 set args 设置命令行参数，用命令 set environment 设置环境变量；然后设置断点，例如，如果想在函数 main 的开始设置一个断点，用命令 b main 即可；设好断点后，便可用命令 r 开始运行程序。程序到断点时便会停下来等待用户的下一步命令。此时可以检查一些变量的值，看运行是否正常，设置其他断点，修改变量的值，或者让程序一步一步执行。

程序的调试是一项非常技巧性的工作，没有通用的规律，在实际使用中积累经验是最重要的。但一般说来，只要找到了程序中出问题的位置及问题类型，要确定问题的原因往往并不困难。而 gdb 可以帮助迅速找到程序出问题的位置。在 gdb 下运行程序，如果程序出现异常会自动停止在出错的位置。并且，如果编译时使用了调试选项 (-g) 的话，会显示出源文件的文件名和行号。输入 bt 或 where 命令，可以显示出程序至出错位置的调用堆栈信息。通常，这些信息足以帮助程序员找出程序中的错误。如果程序的运行是在 gdb 之外进行的，出现异常时操作系统会将出错现场写入一个名为 core 或 core.进程号 的文件 (是否产生 core 文件取决于 Shell 的一些设置，Bash 用户可以用 “ulimit -a” 命令查看对 core file size 的限制，C shell 用户可以用 “limit” 命令查看对 coredumpsize 的限制，必要时修改它们以便在程序出错时得到 core 文件)。在命令行上运行命令 “gdb 程序名 core文件名”，便可以进入到程序出错的现场，与在 gdb 下运行程序得到的现场是一样的。

Linux 中还有一组针对目标文件的工具，包括 nm、objdump 等。请自行阅读 “info binutils” 的内容来了解它们。

顺便指出，新推出的 MPICH2 中提供了对 MPI 并行程序进行调试的工具 mpigdb，它是基于 gdb 的，只是增加了一个新的命令以

表 2.10 gdb 的基本命令

命令名	简写	功能
list	l	列出源程序
file		指定要调试的程序
run	r	运行程序
step	s	单步运行一行
next	n	本子程序中单步运行一行
cont	c	继续运行
finish	fin	运行到本子程序结束
backtrace	bt	显示当前程序调用堆栈
frame	f	选择当前程序调用栈
print	p	打印表达式的值
display	disp	持续显示表达式的值
break	b	设置断点
watch	wa	对表达式设置检测点

选择对哪个进程进行调试操作。

在进行程序调试的时候，经常遇到因为非法内存访问导致程序崩溃的问题。常见的非法内存使用包括：内存没有分配就使用、访问越界内存地址等，还有可能出现的问题是内存分配了没有释放导致内存泄漏、分配内存的命令和释放内存的命令不配套，等等。内存问题占程序问题的很大一部分，有的比较容易定位，有的非常隐蔽，是程序员最头疼的问题之一。Linux 系统中有不少专门针对内存使用的检查、调试工具，例如 GNU Checker、ElectricFence、Purify、TotalView、Valgrind 等等。这里简单介绍一下 Valgrind 的用法。Valgrind 的主页是 `http://valgrind.org`，它是一个免费软件，Linux 发行版中通常包含该程序。用 Valgrind 检查程序中的内存问题，不需要对源程序进行重新编译，只要在 Valgrind 下运行编

译好的程序便可以了。Valgrind 提供了不同类型的异常检查，包括 memcheck (这是最常用的)、addrcheck、cachegrind、helgrind 等。假设要检查的可执行程序名为 a.out，检查内存使用是否合法只需要用下述命令运行程序即可：

```
valgrind --tool=memcheck a.out
```

当然，最好在编译生成可执行文件时使用调试选项 (-g)，这样在 Valgrind 的输出中会包含源文件的行号信息。运行过程中，Valgrind 会输出一些信息到 stderr 上，告诉用户程序的那些地方可能存在内存问题。下面是 Valgrind 的文档中给出的出错信息实例：

```
==25832== Invalid read of size 4
==25832==    at 0x8048724: BandMatrix::ReSize(int, int, int) (bogon.cpp:45)
==25832==    by 0x80487AF: main (bogon.cpp:66)
==25832==    by 0x40371E5E: __libc_start_main (libc-start.c:129)
==25832==    by 0x80485D1: (within /home/sewardj/newmat10/bogon)
==25832==    Address 0xBFFFF74C is not stack'd, malloc'd or free'd
```

其中最前面的数字是进程号，后面的部分是提示信息。它表明源程序 bogon.cpp 的第 45 行访问了一个非法内存地址 (0xBFFFF74C)。根据此类提示，程序员通常很容易找出程序中引起问题的原因。

2.3.2 Fortran 程序的开发

Fortran 语言在数值算法程序中有着重要的地位。在 Linux 环境下编写、编译 Fortran 程序和编写、编译 C 程序没有什么不同。Linux 中默认的 Fortran 编译器是 g77 (如果用 GCC 4 的话也可能是 f95)，它的用法与 C 编译器完全一样，只是输入源文件的扩展名变成了 “.f”、“.F” 或 “.f90” 等。例如：

```
g77 -o bin_file -g source.f -llapack
```

如果使用 Intel 的 Fortran 编译器，编译命令名是 ifc 或者 ifort。

用 Fortran 编写 MPI 并行程序时，一个比较有用的 g77 的选项是 “-Wno-globals”，它关闭对同一个子程序的不同调用中参数类型不同的警告，因为调用 MPI 函数时同一位置的参数经常会使用不同类型的变量。

Fortran 程序的调试也用 gdb。事实上，gdb 支持多种语言的调试，包括 C、C++、Objective C、Modula 2、Ada、Fortran 等。由于编译 Fortran 程序时，编译器会对程序中的变量名进行某种变换，如转换为小写字母、在后面加一个或两个下划线等，调试 Fortran 程序的时候可能看到的变量或函数的名称和源程序中不完全一样。这里给出一个 Fortran 程序的小例子，下面是源代码：

```
c****f** test/test
c FUNCTION
c   test  -- do nothing, just try to show what will appear in the
c           binary file.
c AUTHOR
c   Ruo Li, Jan 09, 2005.
c SEE ALSO
c   g77, nm, robodoc
c***

c****f** test/cos2
c FUNCTION
c   cos2  -- calculate cos(2*x) if sin(x) is known using formula
c           cos(2*x) = 1 - 2*sin^2(x)
c USED BY
c   main
c***
      subroutine cos2(sinx)
        double precision sinx

        sinx = 1.0D0 - 2.0D0*sinx*sinx
        return
      end
```

```
c****f** test/main
c FUNCTION
c   main  -- the code will calculate the sum of cos(2*i) for i from 1
c            to 20 and print the result.
c***
      program main
        implicit none

        integer i
        double precision a, b

        b = 0.0D+0
        do 10 i=1, 20
          a = sin(dble(i))
          call cos2(a)
          b = a + b
10        continue
        write (*,*) "b = ", b
      end

c
c end of file
c
```

这一小段程序计算了 $\sum_{i=1}^{20}\cos(2i)$ 的值。这段代码写成上面的样子主要是为了多使用一些 Fortran 语言的元素，包括不同的数据类型、流控制、子程序、库函数调用等。注意这段代码中注释行的写法，后面将会讨论到它，这里先留下一个悬念。用 g77 对上面这段小程序进行编译得到可执行文件，假设可执行文件名名为 a.out。使用工具 nm 查看可执行文件中的内容：

```
nm a.out
```

会得到下面的信息

```
0804881c T atexit
08049a04 A __bss_start
08048654 t call_gmon_start
08049a04 b completed.1
080486e0 T cos2_
080499b0 d __CTOR_END__
080499ac d __CTOR_LIST__
         U __cxa_atexit@@GLIBC_2.1.3
080498a8 D __data_start
080498a8 W data_start
08048850 t __do_global_ctors_aux
08048678 t __do_global_dtors_aux
         U do_lio
080498ac d __dso_handle
080499b8 d __DTOR_END__
080499b4 d __DTOR_LIST__
080498cc A _DYNAMIC
08049a04 A _edata
080498c8 d __EH_FRAME_BEGIN__
08049a08 A _end
         U e_wsle
         U exit@@GLIBC_2.0
         U f_exit
08048874 T _fini
         U f_init
08048890 R _fp_hw
080486b4 t frame_dummy
080498c8 d __FRAME_END__
         U f_setarg
         U f_setsig
080498b4 d __g77_cilist_0.0
080499c0 A _GLOBAL_OFFSET_TABLE_
         w __gmon_start__
0804853c T _init
08048894 R _IO_stdin_used
080499bc d __JCR_END__
```

```
080499bc d __JCR_LIST__
         w _Jv_RegisterClasses
08048810 T __libc_csu_fini
08048804 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
080487c0 T main
080486fa T MAIN__
080498b0 d p.0
         U sin@@GLIBC_2.0
         U s_stop
08048630 T _start
         U s_wsle
```

请注意下面这些行：

```
080486e0 T cos2_
080487c0 T main
080486fa T MAIN__
         U sin@@GLIBC_2.0
```

其中，第一行是代码中的子程序 cos2，编译器在函数名称后面加了一个下划线。第三行是代码中的主程序 main。前面的大写字母 T 表示函数是在这个文件中定义的。第四行则是被调用的库函数 sin，为了避免和不同的库相混淆，编译器将它的名字转换成了 sin@GLIBC_2.0。那么怎么确定代码中的主程序 main 对应的是第三行中的 MAIN__，而不是第二行中的 main 呢？从下面的调试过程就能知道。可以看到，为了在调试的时候能够在正确的位置设置断点，必须了解这些可执行文件中的信息。幸好 g77 仅仅对函数名进行了变换，对变量名还是保持原样。现在就来对该程序进行调试。首先启动 gdb：

```
[rli@arena tmp]$ gdb a.out
GNU gdb Red Hat Linux (6.3.0.0-0.29rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
```

```
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb)
```

然后在 MAIN__ 中设置断点：

```
(gdb) b MAIN__
Breakpoint 1 at 0x8048700: file test.f, line 36.
```

信息显示断点设置成功，位于源文件 test.f 中的第 36 行。接着开始运行程序：

```
(gdb) r
Starting program: /tmp/a.out

Breakpoint 1, MAIN__ () at test.f:36
36              b = 0.0D+0
Current language:  auto; currently fortran
```

可以看到，程序停止在了刚才设置的断点位置。下面再在函数 cos2 中设置一个断点，然后继续运行

```
(gdb) b cos2_
Breakpoint 2 at 0x80486e3: file test.f, line 21.
(gdb) c
Continuing.

Breakpoint 2, cos2_ (sinx=0xbfffed88) at test.f:21
21              sinx = 1.0D0 - 2.0D0*sinx*sinx
```

现在，程序停在了子程序 cos2 的开头。为了判断上面提到的 main 和 MAIN__ 到底那个是 Fortran 主程序 main，可以看一看当前的调用栈：

```
(gdb) bt
```

```
#0  cos2_ (sinx=0xbfffed88) at test.f:21
#1  0x0804874d in MAIN__ () at test.f:39
#2  0x080487f6 in main ()
```

非常清楚，源代码中的程序 `main` 对应的是可执行文件中的 `MAIN__`。调试 Fortran 程序时，可以和调试 C 程序一样检查和修改变量的值，这里就不一一演示了。请亲自动手实践一下上面的调试过程，以便真正掌握程序调试的基本操作。

2.3.3 软件开发

本节介绍有关软件开发中软件组织、文档管理方面的一些基本工具、技巧和注意事项。目前的主要软件大体上可以分为三类。第一类是商业软件，第二类是学术性软件，第三类是公开软件 (包括自由软件、免费软件等)。当然，它们之间的界限并不是非常分明的，许多学术性软件往往也是公开软件。商业软件一般是很多人协作的结果，当然，也是大量金钱堆积出来的结果，它们的开发过程往往用到许多软件工程的方法和工具。这里侧重介绍学术性或公开软件开发中使用的工具和应该遵循的原则。这类软件的开发形式有其自身的特点，经过多年的发展，已经形成了专门的一套方法和工具。

作为一个软件，通常有下面一些特点：

- 规模比单独一个程序要大得多，一般包含很多个源代码文件，按照一定的目录结构组织在一起；
- 需要附带一些图形文件、数据文件等；
- 需要同时有比较完整清晰的文档；
- 能够比较容易地在不同的机器硬件和软件环境间移植；
- 能够提供一定的技术支持；

要做到所有这些要求，并不是很不容易的事情。

1. 文件组织

既然现在面对的不仅仅是一个单独的程序源码文件，那么这些文件到底怎么摆放就是一个值得研究的问题。当软件中文件的数量较少时，比如十来个文件，可能还不是问题。但是随着开发工作的继续，软件中的文件个数会逐渐增多，使用 `ls -l` 已经不能在一屏之内显示所有文件的时候，就应该开始考虑对文件进行分类了。许多人开始时常常觉得某些文件不知道该分到哪个类比较好，因为它们可能和几个不同的类别都有联系，于是就将它们随意放在某个地方。当经过较长时间，文件变得越来越多以后，可能已经忘记它们在什么地方了，或是感觉它们的位置并不合适，于是又重新调整。与其这么被动地来进行这项工作，不如一开始就计划好，有条理地组织好文件，使得整个软件的结构更加合理。

那么，到底应该怎样进行文件的组织呢？这在很大程度上取决于软件的特点、参与开发的人员构成、最终可能达到的目标和规模以及软件的发布方式和用户对象等。在设计一个软件时，应该首先对这些问题进行仔细考虑，选择最合理的软件组织方式。总体上来说，通常可以遵循下面一些原则：

(1) 头文件和代码文件分开放在不同的目录下；

(2) 如果没有必须放在一起的理由，尽量将程序写在不同的文件中；

(3) 文档放在单独的目录下；

(4) 数据文件和图形文件分别放在单独的目录下；

(5) 如果源代码能够分成不同的功能模块，那么将它们分开放置；

基本原则就是一句话：能够分开的东西，就尽量不要放在一起。

当文件分得比较细，又分散在很多不同的目录中时，可能刚开始会觉得有点不是很方便，但是只要合理地运行软件开发工具来管

理，随着软件内容变得越来越复杂，这样的组织方式就会显现出它的优势。

2. 实用工具 make

对于一个大型软件，其编译、维护是一个复杂而耗时的过程。它涉及到大量的文件、目录，这些文件可能是在不同的时间、由不同的人、在不同的地方分别写的，其中一些是程序，有些是数据，有些是文档，有些是衍生文件。甚至参与开发的人员也不一定清楚所有文件的细节，包括如何处理它们。此外，构成软件的文件数目可能达到成百上千，甚至成千上万个，开发过程中当修改了少量几个文件后，往往只需要重新编译、生成少数几个文件。有效地描述这些文件之间的依赖关系以及处理命令，当个别文件改动后仅执行必要的处理，而不必重复整个编译过程，可以大大提高软件开发的效率。本节介绍的 make 便是针对这些问题而设计的软件开发实用工具。

这里用一个简单软件的例子来介绍如何使用 make 来完成软件的编译过程。假设该软件包括两个头文件，文件名分别为 parameter.h 和 solver.h，从字面上容易知道一个文件包含一些参数的声明，而另外一个文件包含求解器函数的声明。相应地，软件的源程序文件有下面三个：main.c、solver.c 和 parameter.c，它们分别包括整个程序的入口函数——这部分特别分开来写可以使得程序的结构更加清楚、求解器函数和对参数分配存储和设定值的函数。软件的其余部分包括几个程序要读入的数据文件，假定已经准备好了，文件名分别叫做 data1.dat 和 data2.dat。再就是为这个软件准备的文档，因为和数学有关系，假设是一个 TEX 格式的文档，文件名叫做 solver.tex (随后还要介绍更多的文档开发的知识，但是暂且直接手工来维护文档)。一个更复杂的软件大体上也包括这些部分，只是每一部分都要庞大很多，并且每部分可能又被分成很多小的部分，甚至是很多这样的结构再组合构成整个软件。假设这些文件相

互之间有一些依赖关系，比如 main.c 和 solver.c 中包含了头文件 parameter.h 和 solver.h，parameter.c 中则包含了 parameter.h。假定 parameter.c 中有些参数的设定是和数据文件有关系的，如果数据文件被修改的话，某些 parameter.c 中的参数也要做一些调整，而且这些调整不易自动进行，需要进行人工修改。TEX 格式的文件则需要编译成 PDF 格式的文档以方便软件使用者的阅读。那么，使用 make 要达到的目标就是：当一个文件被修改后，make 会自动处理依赖于这个文件的其他文件，使得所有的衍生文件都处于最新状态。下面便是按这样的要求写的 Makefile 文件，它是 make 的输入文件。文件中每部分内容都附有详细的解释，看完这个例子就应该知道使用 make 的基本方法了。这里介绍的是 GNU Make 的功能。

```
# 以一个 # 字符开头的行是注释，这里通过这些注释来解释 Makefile 中每行的含
# 义和作用。

# Makefile 是一个包含一系列规则的文本文件。每个规则定义一个目标所依赖的对象
# 和处理命令。每个规则由两部分组成。第一部分是依赖关系，第二部分是处理命令，
# 其中第二部分可以省略。当更新一个指定的目标时，make 会先检查并且在必要时更
# 新它所依赖的每个对象，然后将目标文件的修改时间依次与每个依赖对象文件的修改
# 时间进行比较，如果任何一个依赖对象比目标新，则执行处理命令来对目标进行更新。
# 由于检查依赖关系时，所依赖的对象会先被检查，而这些对象可能又依赖于其他对象，
# 从而检查过程是递归进行的，整个依赖关系形成一个复杂的树形结构。

# 这里，首先定义一个叫做 all 的默认目标 (target)。它依赖于两个文件，一个是源
# 程序编译产生的可执行文件，叫做 main，另一个是 TeX 文件编译产生的 PDF 文件，
# 叫做 main.pdf。这里只有依赖关系，没有处理命令。当输入命令 "make all" 时，会
# 使得 make 去检查、更新 main 和 main.pdf。
all : main main.pdf

# 编译过程分两步完成，第一步是从 C 源文件编译得到目标文件，第二步对这些目
# 标文件进行链接，得到可执行文件 main。这样，在修改了某个源文件的时候，只有
# 受到影响的目标文件需要重新编译，从而节省编译时间。这在软件规模比较大时尤
# 其有意义。
```

```
# 下面是文件 main 的依赖关系及处理(链接)命令。它依赖于三个目标文件，通过运
# 行 gcc 命令对这三个目标文件进行链接生成。注意，所有包含处理命令的行必须
# 以 <Tab> 开始。
main : main.o solver.o parameter.o
        gcc -o main main.o solver.o parameter.o

# 现在定义关于目标文件 main.o 的规则。它依赖于 C 源文件 main.c 和头文件
# parameter.h，因为在 main.c 中包含语句 `#include "parameter.h"'。一旦
# main.c 和 parameter.h 中任何一个被修改，则需要重新编译生成 main.o。
main.o : main.c parameter.h
        gcc -g -c main.c

# 接下来是关于 solver.o 和 parameter.o 的规则，它们和上面关于 main.o 的规则是
# 完全类似的。
solver.o : solver.c solver.h parameter.h
        gcc -g -c solver.c

parameter.o : parameter.c parameter.h
        gcc -g -c parameter.c

# 当数据文件被修改时，parameter.c 中的相应部分也需要进行修改。这里显示一条信
# 息提醒用户手工修改 parameter.c。
parameter.c : data1.dat data2.dat
        @echo "Data files modified, please revise \"parameter.c\"."

# main.pdf 是用 dvipdf 命令处理 main.dvi 得到的:
main.pdf : main.dvi
        dvipdf -o main.pdf main.dvi

# 而 main.dvi 文件则是用命令 latex 处理 main.tex 产生的:
main.dvi : main.tex
        latex main.tex

# 目标 clean 用于清除所有衍生文件。
```

```
clean :
	rm -f main *.o main.pdf main.dvi main.aux main.log
```

将该文件保存在程序所在的目录下，文件名就叫“Makefile”。然后，只要在命令行上执行“make all”便可编译生成可执行程序“main”和文档 main.pdf。运行 make 时，可以直接指定要检查更新的目标，例如，可以运行“make main”来仅仅生成可执行程序 main，或者运行“make solver.o”来仅仅编译 solver.c。通过比较文件的修改时间，make 只运行必要的命令。例如，如果 solver.o 依赖的所有文件在上次编译后都没有变化，那运行“make solver.o”，make 将不会重新进行编译。如果想强制重新运行编译命令，可先用命令“touch solver.c”来改变文件 solver.c 的最后修改时间，然后运行“make solver.o”。

上面的 Makefile 虽然达到了本节开始提出的要求，但是其中包含一些重复的内容。比如将 .c 文件编译成为相应的 .o 文件的命令都是一样的格式。那么是否可以用统一的方法来描述这些重复的部分呢？另外，如果想换一个编译器，是否可以不用一个一个地修改 Makefile 中的 gcc？还有，如果系统中的头文件和库文件的路径和预期的不太一样，那么又怎么办？下面介绍的 make 的功能可以解决这些问题。

Makefile 中可以使用变量，这些变量也叫做宏。例如，可以将编译器的名字定义为一个变量，如果要使用不同的编译器，只需要修改一下这个变量的值就行了。其次，make 中可以定义隐含规则，例如，可以统一定义如何从相应的以 .c 结尾的文件来生成以 .o 结尾的文件，而不必逐个文件去定义。下面应用这些功能，将上面的 Makefile 改写成如下形式：

```
# 定义变量 CC 和 CFLAGS，它们分别包含编译命令和编译选项
CC = gcc
CFLAGS = -g
```

```
# 这是所有目标文件的列表
OBJECTS = main.o solver.o parameter.o

# 定义从 .c 文件生成 .o 文件的隐式规则。编译命令用前面定义的变量给出。
# Makefile 中引用变量时，将变量的名字放在小括号中，并在前面加一个 $ 即可。
# 注意处理命令中用到了 "$@" 和 "$<"，它们是 make 的内部变量，称为自动变量，
# 前者表示规则中的目标，后者表示第一个依赖对象。
.c.o:
        $(CC) $(CFLAGS) -c -o $@ $<

# 这一行是原来的 Makefile 中的
all : main main.pdf

# 下面的规则中，自动变量 "$^" 表示所有依赖对象
main : $(OBJECTS)
        gcc -o $@ $^

# 下面列出依赖关系
main.o : main.c parameter.h
solver.o : solver.c solver.h parameter.h
parameter.o : parameter.c parameter.h

# 这一行不变
parameter.c : data1.dat data2.dat
        @echo "Data file modified, perhaps parameter.c should be revised"

# 下面的几行也基本上不变，但使用了 make 的自动变量。
main.pdf : main.dvi
        dvipdf -o $@ $<

main.dvi : main.tex
        latex $<

clean :
        rm -f main $(OBJECTS) main.pdf main.dvi main.aux main.log
```

```
# 下面这个古怪的句子用于声明检查 clean 和 all 这两个目标时不要去检查相应的文
# 件，而是直接执行处理命令。(否则当目录中凑巧有一个名为 "clean" 或 "all" 的
# 文件时会干扰 make 的判断)
.PHONY : clean all
```

这个例子中使用了自动变量，如 “$@”、“$<” 等。make 的自动变量在定义隐式规则时非常有用。表 2.11 中列出了 GNU Make 支持的一些常用的自动变量。

当运行 make 时，可以在命令行上定义新的变量，或是改变 Makefile 中的变量的值。例如，如果运行命令：

```
make clean
make CC=ifc CFLAGS="-O3 -g" all
```

则 make 编译时将使用编译命令 “ifc” 和编译选项 “-O3 -g”。注意，在 make 命令行上定义的变量其优先级要高于 Makefile 中的定义！另外，当一个变量没有定义，而同名的环境变量存在时，make 用环境变量的值作为该变量的值。因此，可以将环境变量直接作为 make 的变量使用。

make 还有许多其他功能。此外，与 make 相关的还有许多其他工具，如 Autoconf、Automake、Libtool、aclocal、Autoscan 等等。限

表 2.11 GNU Make 常用自动变量及含义

变量名	含义
$@	目标文件名
$<	第一个依赖对象
$?	全体比目标新的依赖对象
$*	不含扩展名的依赖对象，用在隐式规则中
$^	全体依赖对象，去掉重复的对象
$+	全体依赖对象

于篇幅，这里不做介绍。感兴趣的者可以参阅在线文档或其他书籍。

3. 文档开发和维护

前面已经反复提到在程序开发的同时，进行文档开发的重要性。文档开发之所以常常得不到实现，最主要的原因有两个：一个是没有对文档的编写给予足够的重视；另外一个原因是因为文档的开发没有很方便的工具。软件开发中，经常的工作方式是，一边开发程序，一边在程序中加入注释，同时还要另外维护一份文件，即文档，用来描述程序都在做什么，怎么用。写在程序中的注释一般是给开发者看的，而文档的对象则可能是用户、开发者或其他人。这样就面临一个问题：需要同时维护两份文件，其中一份是程序和注释，另外一份是文档。那么，保持这些文件的同步就显得非常重要了。如果文档提供的是不正确的信息，或许比没有文档要好，但是常常会给用户带来很多麻烦。下面介绍两个专门为了解决这类问题而产生的开发文档的工具，它们是适用于 C 语言的 Doxygen 和适用于 Fortran 语言的 ROBODoc。这些工具的工作方式是：在程序中按照规定的格式写下注释，利用它们对源文件进行处理，将其中的注释提取出来，生成各种格式的文档。这样，仅仅需要维护一份文件，便于在修改程序的同时对文档进行修改。

先简单介绍一下 Doxygen 的用法。Doxygen 的主页在

`http://www.stack.nl/~dimitri/doxygen`

它包含在许多 Linux 发行版中。使用该软件生成文档，需要完成的事情包括：在程序中按规定的格式写注释、准备一个配置文件、运行 Doxygen。Doxygen 软件附带有非常详尽的资料，这里只是简单演示一下它是如何工作的。首先准备一段程序，其中包含 Doxygen 格式的注释。

文件名: `code/doc/test.cpp`

```
1 /**
```

```
 * @file   test.cpp
 * @author Robert Lie
 * @date   Wed Apr 13 20:25:12 2005
 *
 * @brief  这个例子主要是为了列举一下 Doxygen 要求的注释的格式，
 *         这里读者可以写下关于该文件的简要说明
 *
 */

int n; /**< 这是一个全局变量 */

/**
 * 这是一个函数的注释方式，可以在注释中加入用 TeX 格式书写
 * 的数学公式，比如嵌在行内的数学公式 \f$ \alpha^2 + \beta^2 \f$，
 * 公式前后分别用字符串 "\f$" 括起来。也可以写居中对齐的数学公式，例如
 * \f[
 *   \int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx
 * \f]
 * 这里，"\f[" 表示一个居中对齐的公式。
 *
 * @param x 对函数参数的注释
 * @param y 另外一个参数的注释
 *
 * @return 函数的返回值的注释
 */
double f(double x, double y);

/**
 * @brief 这是对一个类的注释方式。可以在这里先写一个摘要
 *
 * 然后，在这里详细描述这个类的具体功用，实现技术等内容。在注释中
 * 还可以使用 HTML 语言，产生灵活的 HTML 格式的文档。例如，
 * 可以加入一个到<a href=\"http://www.pku.edu.cn\">北京大学</a>的
 * 链接，在产生的文档中，就会出现一个相应的链接。至于具体哪些 HTML
 * 的命令可在 Doxygen 中使用请参考其文档。也可以非常方便地产生列表:
```

```
37  *   - 列表的第一条;
38  *   - 列表的第二条;
39  * 上面两行将被格式化为列表。Doxygen 会自动分析所有的类之间的相互
40  * 继承关系，从而自动生成类的继承关系图表。
41  *
42  */
43 class C : public base
44 {
45  private:
46   /**
47    * @defgroup Coord 坐标
48    * 对于内容联系在一起的一些变量或者函数可以对它们统一写注释，
49    * 下面就是一个这样的例子，其中先定义了一个组叫做 Coord
50    */
51   /*\@{*/
52   /**
53    * @addtogroup Coord
54    * x, y 坐标变量
55    */
56   double x_;
57   double y_;
58   /*\@}*/
59  public:
60   /*\@{*/
61   /**
62    * @addtogroup Coord
63    * x, y 坐标变量的只读获取函数
64    */
65   const double& x() const;
66   const double& y() const;
67   /*\@}*/
68 };
```

程序就这么长，其中基本上都是注释，文件名为 `test.cpp`。下面来准备 Doxygen 需要的配置文件。在命令行运行

```
[rli@arena tmp] doxygen -g
```

Doxygen 会自动产生一个配置文件的模板，文件名为 `Doxyfile`，直接编辑它便可获得所需要的配置文件。在配置文件中，通过对一些变量赋值来控制 Doxygen 的处理方式。下面列举的是配置文件中的几个主要变量。自动生成的配置文件模板中对每一个变量都进行了详细的注释，可以通过这些注释了解每个变量的意义及允许的值。

```
# 项目名称
PROJECT_NAME = test
# 输出文件的目录
OUTPUT_DIRECTORY = /tmp/doc
# 文档使用的语言，这里使用中文
OUTPUT_LANGUAGE = Chinese
# 是否将所有注释都产生出来
EXTRACT_ALL = YES
# 对哪些文件进行处理，这里选用以 cpp 做扩展名的文件
FILE_PATTERNS = *.cpp
```

这样 Doxygen 的配置文件就算写好了，将它放在源程序 `test.cpp` 所在的目录中，文件名仍然叫做 `Doxyfile`，然后在目录中输入命令：

```
[rli@arena tmp] doxygen
```

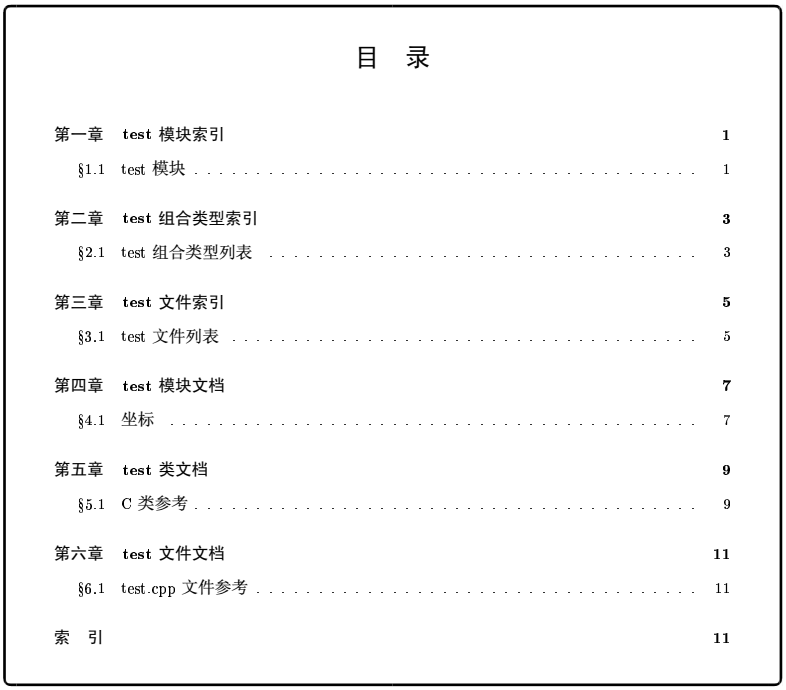
Doxygen 便开始处理。处理过程中会显示很多信息。处理结束后，可以看到目录下多出了一个名为 `doc` 的目录，其中有两个子目录，分别是 `latex` 和 `html`，存储着 LaTeX 格式的文档和 HTML 格式的文档，图 2.3 和图 2.4 分别是它们的屏幕截图。需要注意的是，对于中文文档的 LaTeX 文件，需要手工或是通过一个后处理脚本加入支持中文的 LaTeX 宏包，如 CCT 或 CJK，图 2.3 是将 `refman.tex` 的第一行

```
\documentclass[a4paper]{book}
```

改成

```
\documentclass[a4paper,CJK]{cctbook}
```

后再用 LaTeX 编译的结果。

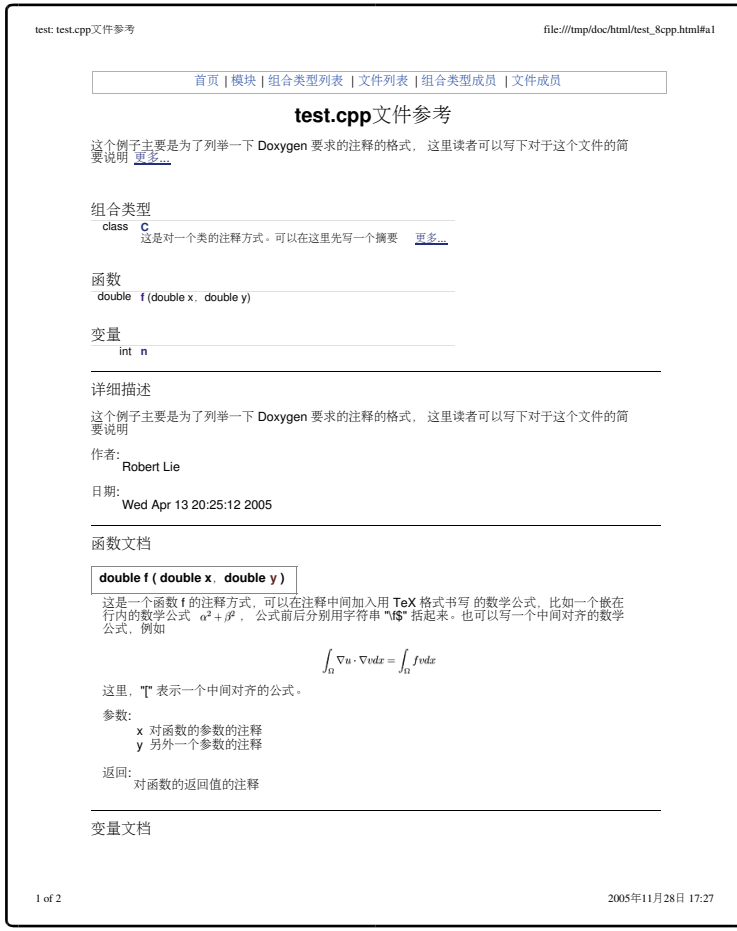


目 录

图 2.3 Doxygen 生成的 LaTeX 文档

当然，除 Doxygen 外，还有其他一些文档生成工具，如 KDOC，其使用方法基本是类似的。

相对于 C 语言而言，对 Fortran 语言进行文档加工的工具则非常少。下面介绍的 ROBODoc 不算很好，但在能处理 Fortran 语言的文档生成工具中算是较好的一个。



test: test.cpp文件参考 file:///tmp/doc/html/test_8cpp.html#a1

首页 | 模块 | 组合类型列表 | 文件列表 | 组合类型成员 | 文件成员

test.cpp文件参考

这个例子主要是为了列举一下 Doxygen 要求的注释的格式， 这里读者可以写下对于这个文件的简要说明 更多...

组合类型

class C
这是对一个类的注释方式。可以在这里先写一个摘要 更多...

函数

double f (double x，double y)

变量

int n

详细描述

这个例子主要是为了列举一下 Doxygen 要求的注释的格式， 这里读者可以写下对于这个文件的简要说明

作者:
Robert Lie

日期:
Wed Apr 13 20:25:12 2005

函数文档

double f (double x，double y)

这是一个函数 f 的注释方式，可以在注释中间加入用 TeX 格式书写 的数学公式，比如一个嵌在行内的数学公式 $\alpha^2+\beta^2$， 公式前后分别用字符串 "\f$" 括起来。也可以写一个中间对齐的数学公式，例如

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx$$

这里，"[" 表示一个中间对齐的公式。

参数:
x 对函数的参数的注释
y 另外一个参数的注释

返回:
对函数的返回值的注释

变量文档

1 of 2 2005年11月28日 17:27

图 2.4 Doxygen 生成的 HTML 文档

ROBODoc 的主页在

`http://www.xs4all.nl/~rfsber/Robo/index.html`

这里也通过一段程序例子来说明它的用法，这段例子就是 2.3.2 节中给出的程序实例。可以看到，使用 ROBODoc 进行文档处理时，注释都是写在一个形如

```
c****X** 父模块名/子模块名
c 注释在这里
c***
```

的结构中间。注释开始的行由四个星号，后跟一个字符，这里用 "X" 表示，然后再接两个星号，最后是一对以斜线分开的名称。字符 "X" 表示对哪类对象进行注释，例如，"f" 表示对函数进行注释，"d" 表示对常量进行注释，等等。斜线分开的名称使得 ROBODoc 能够将文档按照树型的结构组织起来。在注释中，ROBODoc 可以识别一些关键词，比如 NAME，COPYRIGHT，FUNCTION，AUTHOR，等等，可以对这些部分分别做相应的注释。每块注释用连续的三个星号结束。其他请参阅 ROBODoc 自带的使用指南，该文档只有几页，非常简单。

4. 版本管理和协同工作

本节最后介绍版本管理工具，它们对于复杂软件的协同开发非常有用。在程序开发中，可能常常遇到这样的情况：一个程序已经可以运行得到正确的结果了，但是还想将程序修改得完美一些，然而结果是修改后的程序运行得不到正确的结果了，反复检查，就是找不到问题出在什么地方，想将程序恢复到开始的状态也不可能了。或许有的开发者比较有经验，每次在进行程序的修改前进行备份，这样的结果是几个月以后，已经有了几十个不同的备份，怎么也想不起来某个备份是因为什么原因留下来的了。还有一种情况是，在几个人合作进行开发的时候，总是不能同步地进行文件的相互传递和信息的沟通，使得合作的结果还不如一个人完成所有的工作来得快。

遇到这类问题的时候，版本管理软件可以大显身手。目前常用的版本管理软件有 CVS 和 Subversion。虽然后者改进了前者中的一些缺陷，并增加了新的功能，但前者在使用的广泛程度上依然大大超过后者。因此，这里简单介绍一下 CVS 的基本功能和使用方法。

所谓版本管理，其实就是一个可以记录下开发过程中所有文件的所有变化的工具。文件变化的过程被作为不同的版本记录下来，事后可以随意获得开发过程中任何一个时刻的文件，不用担心因为程序的改动而造成不可挽回的问题。如果多人合作进行开发，版本管理软件帮助开发者们同步他们的改动。开发者分别独立工作，当需要其他合作者的最新版本的文件的时候，可以随时拿到。甚至当不同开发者同时修改了同一个文件时，版本管理软件会将这些修改整合到一起。如果不同开发者对同一个文件做出了互相矛盾的修改，版本管理软件会发现相互矛盾的地方并提醒开发者，使得问题能够及时被纠正。本书的编写过程就是通过 CVS 协同完成的。

要使用 CVS，首先需要有提供 CVS 服务的服务器。Linux 系统就能够提供这样的服务，只需要置 CVS 服务器。配置 CVS 服务器的过程并不复杂，但是需要系统管理员权限及一些关于系统管理方面的知识，这方面的内容请参考 CVS 软件本身附带的文档。这里主要介绍 CVS 的使用。

CVS 的命令为 `cvs`。它的命令行格式如下：

```
cvs [全局选项] 命令 [命令选项] [参数]
```

紧跟 `cvs` 命令之后的是一些全局选项，然后是一条 CVS 命令，然后是该命令的选项和参数。

使用 CVS 命令前首先要进行登录。CVS 的登录命令是 `login`，其格式为：

```
cvs -d :pserver:用户名@主机名:[端口:]/目录 login
```

其中 “端口” 为 CVS 服务器使用的端口，默认值为 `2401`。键入上述

命令后，系统会提示输入口令。除了可以通过 “-d” 选项指定 CVS 服务器外，也可以通过设置环境变量 CVSROOT 来指定：

```
export CVSROOT=:pserver:用户名@主机名:[端口:]/根目录
cvs login
```

在成功登录后，口令信息会保存在用户家目录中的文件 .cvspass 中，以后不用重新登录 (除非执行 “cvs logout”)。下面是一个通过匿名 CVS 获取 Gnuplot 源码的过程示例，Gnuplot 是一个著名的免费二维绘图软件。

```
[rli@arena tmp]$ CVSROOT=:pserver:anonymous@cvs.sourceforge.net:/cvsroot/gnuplot
[rli@arena tmp]$ export CVSROOT
[rli@arena tmp]$ cvs login
Logging in to :pserver:anonymous@cvs.sourceforge.net:2401/cvsroot/gnuplot
CVS password: (因为是匿名帐号，因此直接键入回车即可)
[rli@arena tmp]$ cvs -z3 co gnuplot
cvs checkout: Updating gnuplot
U gnuplot/.cvsignore
U gnuplot/BUGS
U gnuplot/ChangeLog
U gnuplot/ChangeLog.0
U gnuplot/CodeStyle
U gnuplot/Copyright
U gnuplot/FAQ
... ...
cvs checkout: Updating gnuplot/win
[rli@arena tmp]$
```

上例中，“co” 是 CVS 命令 “checkout” (检出) 的简写形式，用于下载软件的最新版本，其中的选项 “-z3” 告诉 CVS 下载时对数据进行压缩以提高网络传输的速度，数据压缩对于用户是透明的。

进行一个项目开发，第一步是为该项目建立基本的目录结构，接着就应该将它用 CVS 管理起来。在 CVS 服务器中设立一个新项目，首先需要用 CVS 的 import 命令导入项目的初始文件和目录。假设

项目名称为 myproject，项目的初始文件和子目录保存在同名目录下。只需进入目录 myproject，并运行命令 (这里假设已经设置了环境变量 CVSROOT 并成功地进行了登录)

```
cvs -m "initial import" import myproject rli start
```

便可将项目导入 CVS 服务器。命令运行成功后，项目的所有文件和目录便安全地保存在 CVS 服务器中。

现在试试从服务器上下载刚刚导入的项目。首先删除本地的目录 myproject 和其中的所有文件，然后运行命令

```
cvs co myproject
```

便又在本地得到了一份完整的目录和文件。该目录和它下面的每个子目录中多出了一个名为 CVS 的子目录，它们中保存着当前文件的版本信息，因此请不要改动或删除它们。从服务器上下载到本地的拷贝叫做工作拷贝，而本地的 myproject 目录称为工作目录。

当几个开发者合作进行程序开发时，每个人都在自己的机器上保存一份工作拷贝，并且在其中进行开发和修改。通常，每天的工作方式是：早上开始工作的时候，在工作目录中运行命令 “cvs up -d” 来更新自己的工作拷贝；在每天工作结束之前，在工作目录中运行命令 “cvs ci” 将当天修改过的内容整合到服务器中去。这样在每天开始工作的时候，拿到的文件都是开发小组中所有的人员昨天提交的最新版本。

运行 “ci” (commit) 命令时，CVS 会进入一个 vi 编辑窗口，要求输入关于所做的修改的说明，这些说明会和文件的相应版本一起保存在 CVS 服务器中，将来可以随时查看每位开发者每次修改都做了些什么事情。

在项目中加入新的文件或子目录的命令是

```
cvs add 文件或目录名
```

从项目中删除文件或子目录的命令是

```
cvs remove 文件或目录名
```

在运行 CVS 删除文件或子目录的命令之前，必须先删除工作拷贝中的这些文件或子目录。CVS 的添加和删除命令只是将要添加入或删除的文件或子目录记录下来，当运行 `ci` 命令时才实际进行操作。

其他还有一些常用的命令，包括：“`cvs log`” 用来查看日志信息，“`cvs status`” 用来查看文件状态，“`cvs diff`” 用来比较不同版本文件的不同，“`cvs tag`” 为某个版本的文件加上标签。还有一个非常有用的脚本 `rcs2log`，它将每次提交修改时的说明按时间、文件整理成规范的修改日志 (changelog)。这里就不一一介绍了。

第 3 章　消息传递编程接口 MPI

MPI (Message Passing Interface) 是由全世界工业、科研和政府部门联合建立的一个消息传递编程标准，其目的是为基于消息传递的并行程序设计提供一个高效、可扩展、统一的编程环境。它是目前最为通用的并行编程方式，也是分布式并行系统的主要编程环境。MPI 标准中定义了一组函数接口用于进程间的消息传递。这些函数的具体实现由各计算机厂商或科研部门来完成。除各厂商提供的 MPI 系统外，一些高校、科研部门也在开发免费的通用 MPI 系统，其中比较著名的有：

- MPICH (`http://www.mcs.anl.gov/mpi/mpich`)
- LAM MPI (`http://www.lam-mpi.org/`)

它们均提供源代码，并支持目前绝大部分并行计算机系统 (包括微机和工作站机群)。事实上许多厂商提供的 MPI 系统是在 MPICH 的基础上经过针对特定硬件的优化形成的。

MPI 标准的第一个版本 MPI 1.0 于 1994 年公布。最新标准为 2.0 版，于 1998 年公布，但许多 MPI 系统目前尚未实现 2.0 版规定的全部内容。

一个 MPI 系统通常由一组库、头文件和相应的运行、调试环境构成。MPI 并行程序通过调用 MPI 库中的函数来完成消息传递，编译时与 MPI 库链接。而 MPI 系统提供的运行环境则负责一个 MPI 并行程序的启动与退出，并提供适当的并行程序调试、跟踪方面的支持。

3.1 MPICH 安装与程序编译、运行、调试

MPICH是目前使用最广泛的免费 MPI 系统，它支持几乎所有 Linux/UNIX 以及 Windows 9x、NT、2000 和 XP 系统。利用 MPICH 既可以在单台微机或工作站上建立 MPI 程序的调试环境，使用多个进程模拟运行 MPI 并行程序，也可以在 SMP 系统或机群环境上建立实用的并行计算环境。事实上，它是运行在目前大部分机群系统上的主要并行环境。

本节介绍如何在 Linux 系统中安装 MPICH，以及 MPI 程序在 MPICH 环境中的运行与调试。为了使用 MPICH，必须安装某些其他软件包，在下面的介绍中会列出需要安装的这些软件包，几乎所有 Linux 发行版都包含它们。关于 RedHat/Fedora Linux 中安装软件包的方法可参看 72 页“软件包的管理”。

MPICH 的网址是 `http://www.mcs.anl.gov/mpi/mpich`，从该处可以下载源程序 `mpich.tar.gz`，以及一些说明、补丁等。

3.1.1 单机环境下 MPICH 的安装

1. 配置 `rsh`

为了在单机中安装使用 MPICH，必须在 Linux 中安装并启用 `rsh` 服务程序和客户程序。如果使用 RedHat 或 Fedora Linux 的话，需要安装 `xinetd`、`rsh-server` 和 `rsh` 包，并运行命令

```
/sbin/chkconfig xinetd on
/sbin/chkconfig rsh on
/sbin/chkconfig kshell on
```

来开启 `rsh` 服务 (其中第三行开启 `kshd` 服务，在较新的 Linux 系统中经常使用基于 Kerberos 认证的 `rsh` 服务来增强系统的安全性)。

在 MPICH 程序中，`rsh` 被用于启动 MPI 进程。MPICH 也可以使用 `ssh` 来启动 MPI 进程，用 `ssh` 代替 `rsh` 启动 MPI 进程具有

更好的安全性，缺点是启动速度较慢。关于如何配置 MPICH 用 ssh 代替 rsh 启动 MPI 进程不在这里介绍，感兴趣的者可参考 MPICH 附带的文档 (如源程序中的 doc/mpichman-chp4.pdf 文件)。

为了能够正确运行 MPICH 并行程序，除了安装 rsh 客户和服务程序以及开启 rsh 服务之外，还需要允许用户直接通过 rsh 在本机上启动程序，这一过程称为建立 rsh 信任关系。

有两种方式建立 rsh 的信任关系。用户可以通过配置家目录下的 .rhosts 文件来指定允许哪些机器直接通过 rsh 在本机以自己的身份运行程序。系统管理员则可以配置文件 /etc/hosts.equiv，它对所有用户都起作用。.rhosts 文件或 /etc/hosts.equiv 文件中只需列出允许通过 rsh 访问本机的主机名或 IP 地址，一行一个 (它表示允许所列出的主机通过 rsh 命令在本机远程执行命令而不需要给出密码)。当配置单机上的 MPICH 时，只要在 .rhosts 或 /etc/hosts.equiv 文件中加入本机的主机名 (即 hostname 命令显示的主机名) 即可。特别要注意文件 /etc/hosts.equiv 和 .rhosts 的权限，出于安全考虑如果这些文件的权限不正确的话 rsh 将拒绝访问。/etc/hosts.equiv 文件应该为 root 所拥有，而 .rhosts 文件则应该为相应的用户拥有，建议将这两个文件的访问权限设成 0600。

注：现代 rsh 的认证通过 PAM (Pluggable Authentication Modules) 设定，Linux 系统中认证方式由文件 /etc/pam.d/rsh 定义。上面介绍的配置方法是默认情况下的行为，也是传统 rsh 的认证方式。此外，文件 /etc/hosts.allow，/etc/hosts.deny 也用于控制对机器的访问。如果不了解的话不要轻易改动这些文件，以免影响 rsh 的正常工作，导致 MPICH 程序无法正常启动。

rsh 配置完毕后，可以用下面的命令测试它是否工作正常：

```
rsh 主机名 true
```

上述命令应该立即运行完毕并且不产生任何输出。如果有错误信

息或其他输出信息，请参照上面介绍的过程检查 rsh 的配置 (文件 /var/log/messages 中通常有拒绝 rsh 访问的具体原因)，特别注意检查 shell 初始化文件 (/etc/profile.d/*、/etc/csh.cshrc、/etc/profile、/etc/bashrc，用户家目录下的 .profile、.bashrc、.cshrc 等等)，确保它们不要输出任何信息到 stdout 或 stderr。另外一个经常引起问题的原因是 /etc/hosts 中本机的 IP 地址定义与实际不符。

2. 编译、安装 MPICH

首先，从前面给出的 MPICH 的网址下载 MPICH 的源程序打包文件 mpich.tar.gz，并展开：

```
tar xzpvf mpich.tar.gz
```

源程序展开后将在当前目录下形成一个子目录 mpich-1.2.x，其中 1.2.x 为 MPICH 的版本号 (如 1.2.6，实际操作时请将 1.2.x 换成正确的版本号)。

最简单的情况下，编译、安装过程只需要下面几步：

```
cd mpich-1.2.x
./configure --prefix=/usr/local
make
su
make install
```

其中 configure 命令用于对 MPICH 进行配置，它自动检测、收集编译 MPICH 所需要的信息，包括 C/C++/Fortran/Java 编译器及库函数、MPICH 的安装目录、底层通信库的选取，等等。运行 configure 时，可以通过选项或环境变量改变 MPICH 的参数，具体请参考 MPICH 的文档或命令 "./configure --help" 的输出。

上述命令完成后会将 MPICH 安装到目录 /usr/local/mpi 中。如果希望将 MPICH 安装到其他目录，可修改命令中 "--prefix" 选

项的参数。MPICH 的安装目录中通常包含下述几个子目录：

`/usr/local/mpi/bin`
其中包含用于编译、运行 MPI 程序的脚本 (如 `mpicc`，`mpicxx`，`mpirun`，等等)。

`/usr/local/mpi/doc`
其中包含 MPICH 的文档 (PS 和 PDF 文件)。

`/usr/local/mpi/etc`
其中保存着 MPICH 的配置。

`/usr/local/mpi/include`
其中包含 MPICH 的头文件 (如 `mpi.h`，`mpif.h`，等等)。

`/usr/local/mpi/lib`
其中包含 MPICH 库。

`/usr/local/mpi/man`
其中包含 MPI 函数的在线手册 (man pages)。

`/usr/local/mpi/sbin`
其中包含 MPICH 的一些辅助程序 (`chp4_servs` 等)。

`/usr/local/mpi/www`
其中包含 MPI 函数的 HTML 文档。

`/usr/local/mpi/examples`
其中包含 MPICH 提供的一些 MPI 程序实例 (`cpi.c`，`pi3.f`，等等)。

建议将目录 `/usr/local/mpi/bin` 加入到环境变量 `PATH` 中去，以方便使用该目录中的脚本。在 RedHat 或 Fedora Core Linux 中，只

须在目录 /etc/profile.d 中创建两个文件 mpich.sh 和 mpich.csh，它们分别适用于用户的登录 Shell 是 Bash 和 C–Shell 的情况。文件 mpich.sh 包含如下内容：

```
export MANPATH=${MANPATH}:/usr/local/mpi/man
export PATH=${PATH}:/usr/local/mpi/bin
```

文件 mpich.csh 包含如下内容：

```
if ( $?MANPATH == 0 ) then
    setenv MANPATH :/usr/local/mpi/man
else
    setenv MANPATH ${MANPATH}:/usr/local/mpi/man
endif
setenv PATH ${PATH}:/usr/local/mpi/bin
```

注意上述两个文件中的设置在重新登录后才会起作用。完成这些设置后，运行 /usr/local/mpi/bin 中的脚本时只须输入文件名而不必包含路径，并且可以用命令“man MPI函数名”阅读 MPI 函数的在线手册。

3. MPI 程序的编译

为了方便用户编译、链接 MPI 程序，MPICH 提供了一组 Shell 脚本，包括 mpicc，mpicxx (MPICH 1.2.5 之前的版本中该脚本名为 mpiCC)，mpif77，mpif90 等，分别用于 C，C++，Fortran 77/90 等 MPI 程序的编译和链接。这些脚本的用法与 Linux/UNIX 中普通编译器的用法完全相同，区别是它们在编译时会自动加上 MPICH 头文件的路径 (-I/usr/local/mpi/include)，链接时会自动加上 MPICH 库的路径及所需要的库。此外，这些脚本中会包含一些优化或调试的编译选项，具体选项与编译 MPICH 时运行 configure 的参数以及一些环境变量，如 CFLAGS、FFLAGS 等有关。

下面以 MPICH 附带的程序实例为例，分别说明 C，C++ 和

Fortran 程序的编译方法。这些程序实例可在目录

```
/usr/local/mpi/examples
```

中找到。

(1) C 程序：`mpicc -o cpi cpi.c`

(2) C++ 程序：`mpicxx -o hello++ hello++.cxx`

(3) Fortran 77 程序：`mpif77 -o pi3 pi3.f`

(4) Fortran 90 程序：`mpif90 -o pi3f90 pi3f90.f90`

上述例子中生成的可执行文件名分别为 `cpi`，`hello++`，`pi3` 和 `pi3f90`。

另外，文件 `/usr/local/mpi/share/Makefile.sample` 提供了一个编译上述实例的 `Makefile` 模版，用户编写自己的 `Makefile` 时可以参考该文件。

4. MPICH 程序的运行

采用前面介绍的配置方法编译产生的 MPICH 系统使用 `ch_p4` 作为底层通信支持。下面介绍的启动 MPICH 并行程序的方法仅对 `ch_p4` 的 MPICH 程序有效。使用其他通信机制的 MPICH 程序的启动方式各不相同，请自行参看相关的资料。

基于 `ch_p4` 的 MPICH 程序可以像普通 UNIX 可执行文件一样直接执行，但默认情况下它只启动一个进程。当需要启动多个进程时，有两个方法来控制启动的进程数目，第一个方法是使用选项 `-p4pg`，第二个方法是利用 MPICH 提供的一些脚本文件，如 `mpirun`。对于单机的情形，最方便的是用命令 `mpirun` 来运行 MPICH 程序，因此这里仅对它作一些介绍。3.1.2 中将介绍如何采用第一个方法启动 MPI 程序。

MPICH 提供的脚本 mpirun 用于运行 MPICH 程序。它通过一组选项来控制程序的启动方式及启动的进程数。命令 “mpirun --help” 给出 mpirun 选项的简要说明。最简单、也是最常用的形式如下：

```
mpirun [-machinefile 文件名] [-np 进程数] 程序名 [命令行参数]
```

方括号中为可选参数。文件名给出包含主机列表的文件，它列出运行 MPICH 程序的主机名或 IP 地址。当省略 “-machinefile 文件名” 选项时 mpirun 使用文件

/usr/local/mpi/share/machine.XXXX

中列出的主机，其中 XXXX 对应于操作系统的类型，对于 Linux 系统而言 XXXX = LINUX。

在单机 Linux 环境中运行 MPICH 程序，最简单的方法是修改文件：

/usr/local/mpi/share/machine.LINUX

在其中列出本机主机名 (hostname 命令显示的主机名)，然后运行命令

mpirun -np 进程数 程序名

即可，mpirun 会自动根据 -np 后面的参数启动指定数目的 MPI 进程。

3.1.2 机群环境下 MPICH 的安装

机群环境下 MPICH 对 Linux 系统的安装要求与前一节单机环境一样。此外，在开始下面的步骤之前还应该先配置好 TCP/IP 网络连接。为避免额外麻烦，在安装、测试 MPICH 时请不要开启任何防火墙设置。

下面将以一个通过以太网连接、包含四台微机的机群为例来说明机群中 MPICH 的安装及配置。假设四台结点机的主机名/IP 地址分别为 node1/10.0.0.1，node2/10.0.0.2，node3/10.0.0.3 和

node4/10.0.0.4。在这些微机上已经安装了 Linux 系统并且配置好了网络。建议在这四台机器上使用相同的 /etc/hosts 文件，其中包含下面的内容：

```
127.0.0.1    localhost.localdomain   localhost
10.10.10.1   node1.mydomain  node1
10.10.10.2   node2.mydomain  node2
10.10.10.3   node3.mydomain  node3
10.10.10.4   node4.mydomain  node4
```

这样，无论是用主机名还是 IP 地址四台机器都可以互相访问。

1. 配置 NFS

为了方便 MPICH 的安装及并行程序的运行，最好将 MPICH 的安装目录及用户家目录通过 NFS 网络文件系统共享。对于仅包含几个结点的较小的集群系统，可以任意指定其中一个结点作为 NFS 服务器。对较大的集群系统，最好设定一个或数个结点专门用于文件服务，这些结点称为 I/O 结点，它们专门负责存储设备的管理，不参加计算。这里选择 node1 作为 NFS 服务器，将它的 /home 和 /usr/local 目录输出给其他三个结点，相应的配置步骤如下。

第一步，以 root 身分登录到 node1 上。确保 node1 上安装了 NFS 程序 (nfs-utils 包)。首先运行一遍下述命令来开启 NFS 服务：

```
/sbin/chkconfig nfs on
/sbin/chkconfig nfslock on
/etc/init.d/nfslock restart
/etc/init.d/nfs restart
```

然后编辑文件 /etc/exports，在其中加入下面二行 (如果该文件不存在则创建一个新文件)：

```
/home       10.0.0.0/255.255.255.248(rw,async,no_root_squash)
```

```
/usr/local  10.0.0.0/255.255.255.248(rw,async,no_root_squash)
```

做好上述修改后执行下面的命令：

```
/sbin/exportfs -a
```

便完成了 /home 和 /usr/local 目录的输出。

第二步，以 root 身份登录到其余三个结点，在文件 /etc/fstab 中加入下面两行：

```
node1:/home       /home       nfs  defaults 0 0
node1:/usr/local  /usr/local  nfs  defaults 0 0
```

并且运行一次下述命令：

```
/sbin/chkconfig netfs on
mount -t nfs -a
```

完成上面的步骤后，node2，node3 和 node4 应该共享 node1 的 /home 和 /usr/local 目录。可以在任何一个结点上用 df 命令来验证，例如：

```
# df
... ...
node1:/home            248632644 224028484  11974280  95% /home
node1:/usr/local       246966888 200888560  33533076  86% /usr/local
... ...
```

2. 配置 NIS

接下来配置 NIS (Network Information Service)，以便在各结点机间共享用户信息。这里仍然用 node1 作为 NIS 服务器。以 root 身份登录到 node1 上，确认安装了 ypserv 和 yp-tools 包。在文件 /etc/sysconfig/network 中加入下面一行：

```
NISDOMAIN=mycluster
```

(这里选取 NIS 域名为 mycluster，可以根据自己的情况任意选择其他名称作为 NIS 域名)，然后执行下述命令开启 NIS 服务并初始化 NIS 数据库：

```
/sbin/chkconfig ypserv on
/etc/init.d/ypserv restart
/usr/lib/yp/ypinit -m
```

ypinit 程序运行时会要求用户输入从服务器主机名，这里只需直接按 Ctrl-D，然后回答 “y(es)” 再回车即可。

完成 NIS 服务器的设置后再配置 NIS 客户。依次登录到结点 node1，node2，...，node4 上，确认安装了 ypbind 和 yp-tools 包，在文件 /etc/sysconfig/network 中加入：

```
NISDOMAIN=mycluster
```

然后执行下述命令启动 NIS 客户程序：

```
/sbin/chkconfig ypbind on
/etc/init.d/ypbind restart
```

默认情况下，NIS 客户程序 ypbind 通过广播的方式搜索 NIS 服务器。也可以在文件 /etc/yp.conf 中加入一行 “ypserver node1” 来直接指定 NIS 服务器 (这一步应该在启动 NIS 客户程序之前做)。

配置完 NIS 服务和客户后，应该在 node2，node3 和 node4 上分别用命令 ypwhich 和 ypcat 来检查是否能够从 NIS 服务器得到所需要的信息：

```
# ypwhich
node1
# ypcat passwd
(应该显示出 node1 上的 passwd 信息)
... ...
```

为了能够使用 NIS 数据库中的用户、用户组等信息，需要修改文件 /etc/nsswitch.conf 中的配置，将有关的行改成如下形式：

```
passwd: files nis
shadow: files nis
group:  files nis
hosts:  files nis dns
```

完成这些步骤后，所有定义在 node1 上的用户账号 (系统帐号除外) 均可直接在其他结点中使用，并且具有相同的密码。每次添加新的用户账号时，只需在 node1 上进行，添加完帐号后必须在 node1 运行下述命令来刷新 NIS 数据库：

```
cd /var/yp
make
```

用户可以在任何一台结点机上修改密码，但不能用 passwd 命令，而必须用 yppasswd 命令。密码修改后对所有结点起作用。一定要注意只在 node1 上创建用户账号，不要在其他结点上建账号，以免产生冲突，影响并行程序的执行。

3. 配置 rsh

rsh 的配置与单机环境完全类似，但必须分别在每个结点上进行。首先，确认已经安装了 rsh 和 rsh-server 包并且开启了 rsh 服务 (/sbin/chkconfig rsh on)。然后创建文件 /etc/hosts.equiv，在其中加入下述内容：

```
node1
node2
node3
node4
```

完成后可以用任何一个普通用户的身份来检查是否可以从任何一个结点在其他结点上远程运行命令，例如可以在 node1 上运行命令

“rsh node2 true” 等等。

需要指出的是，/etc/hosts.equiv 文件中的设置只对普通用户起作用。如果希望允许 root 用户通过 rsh 命令在其他结点上运行命令，则 root 的 ~/.rhosts 中必须包含所有结点机名，并且应该在所有结点机的 /etc/securetty 文件中加入一行 “rsh”。允许 root 通过 rsh 访问各结点，可以大大方便一些系统配置工作，但同时也会增加系统不安全的因素。

4. 编译、安装 MPICH

由于 /usr/local 目录是所有结点机共享的，因此 MPICH 只要安装一次即可，不必重复地在每个结点上安装。编译、安装过程与单机情形完全一样，可以在任何一个结点上进行。

MPICH 编译安装完毕后，应该在每个结点的 /etc/profile.d 目录中创建文件 mpich.sh 和 mpich.csh，内容与单机时一样，以方便用户编译、运行 MPICH 程序。

5. 运行 MPICH 程序

多机环境中运行 MPICH 程序与单机环境类似，可以用 mpirun 来进行。运行程序前先创建一个 machinefile 文件，其中列出要使用的结点机名，然后用命令 “mpirun -machinefile 文件名 ...” 来在指定的结点上运行程序。例如，假设用户登录在结点 node2 上，文件 mfile 中包含下述内容：

```
node3
node4
```

则命令：

```
mpirun -machinefile mfile -np 3 cpi
```

将用 node2，node3 和 node4 来运行程序 cpi，每个结点一个进程，这是因为默认情况下 mpirun 总是将当前结点添加到程序的结点机

列表中。如果不希望使用当前结点 (node2)，可以加上 -nolocal 选项：

```
mpirun -nolocal -machinefile mfile -np 3 cpi
```

选项 -np 给出的进程数与 -machinefile 给出的文件中的结点机数不一定要相等。如果进程数少于结点机数，则程序只使用其中的一部分结点。如果进程数多于结点机数，则一些结点上会运行多于一个进程。

下面介绍另外一个启动 MPICH ch_p4 程序的方法。基于 ch_p4 的 MPICH 程序启动时由第一个进程根据一个称作 “p4 group file” 的文件 (简称 p4pg 文件) 来启动其他进程。用 mpirun 命令运行程序时，脚本 mpirun 会自动为用户创建一个临时 p4pg 文件，并且在程序运行结束后删除该文件，临时文件名通常为 PIxxxx (可以用 “-keep_pg” 选项让 mpirun 在程序结束后保留 p4pg 文件)。用户可以自行创建 p4pg 文件来运行 MPICH 程序，好处是可以精确控制进程在结点上的分布，并且可以启动非 SPMD 模式的程序 (即一个程序由多个不同的可执行文件组成，如 master/slave 模式)。

p4pg 文件的格式如下：

```
主机名1  0  程序名1
主机名2  1  程序名2
... ...
主机名n  1  程序名n
```

每行中间的数字代表该结点上的进程数 (第一行的进程数要减 1)，使用 ch_p4 时它们只能取为 1。各行上的主机名和程序名可以相同，也可以不同，程序名必须用绝对路径名。下面通过一个具体例子来说明如何通过 p4pg 文件控制一个 MPICH 程序的进程数及各个进程在结点机上的分布。假设文件 p4file 中包含如下内容：

```
node1 0 /home/user/test/cpi
```

```
node2 1 /home/user/test/cpi
node3 1 /home/user/test/cpi
node2 1 /home/user/test/cpi
```

假设当前结点机为 node1，当前目录为 /home/user/test，目录下有 MPICH 程序 cpi，则命令：

```
./cpi -p4pg p4file
```

将用 4 个进程运行程序 cpi，进程 0 在 node1 上，进程 1 在 node2 上，进程 2 在 node3 上，进程 3 在 node2 上。在这个例子中 node2 上有两个进程。注意，启动程序的结点必须与 p4pg 文件中第一行上的结点一样，否则程序将会报错。

不难根据上面的例子构造 p4pg 文件来启动更复杂的 MPICH 程序。

3.2 MPI 编程

本节介绍 MPI 编程的基本概念以及 MPI 的基本通信函数。这里只对一些常用函数的功能及用途进行简要描述。附录 B 中提供了所有 MPI 函数接口及变量的说明，读者可以通过 447 页 B.1 中的函数、变量名索引方便地查到所关心的函数或变量。关于 MPI 编程更详尽的介绍请参阅 [18, 19, 20, 21]。

3.2.1 MPI 编程的基本概念

一个 MPI 并行程序由一组运行在相同或不同计算机/计算结点上的进程或线程构成。这些进程或线程可以运行在不同处理机上，也可以运行在相同的处理机上。为统一起见，MPI 程序中一个独立参与通信的个体称为一个进程 (process)。一个 MPI 进程通常对应于一个普通进程或线程，但是在共享存储/消息传递混合模式程序中，一个 MPI 进程可能代表一组 UNIX 线程。

一个 MPI 程序中由部分或全部进程构成的一个有序集合称为一个进程组 (process group)。进程组中每个进程被赋予一个该组中的序号 (rank)，用于在该组中标识该进程，称为进程号。进程号的取值范围从 0 开始。

MPI 程序中进程间的通信、同步等通过通信器 (communicator) 进行 (一些资料中将通信器翻译成通信子，本书中将统一使用术语通信器)。MPI 的通信器有域内通信器 (intra-communicator) 和域间通信器 (inter-communicator) 两种，前者用于属于同一进程组的进程间的通信，后者用于分属两个不同进程组的进程间的通信。这里只对域内通信器进行介绍，后文中除非特别提及，“通信器”一词一律指域内通信器。

一个通信器由它所包含的进程组及与之相关联的一组属性 (例如进程间的拓扑连接关系) 构成。通信器提供进程间通信的基本环境，MPI 程序中所有通信都必须在特定的通信器中完成。MPI 程序启动时会自动创建两个通信器，一个称为 `MPI_COMM_WORLD`，它包含程序中的所有进程，另一个称为 `MPI_COMM_SELF`，它是每个进程独自构成的、仅包含自己的通信器。

在 MPI 程序中，一个 MPI 进程由一个通信器 (或进程组) 和进程在该通信器 (或进程组) 中的进程号唯一标识。注意进程号是相对于通信器或进程组而言的：同一个进程在不同的通信器 (或进程组) 中可以有不同的进程号。进程号是在通信器 (或进程组) 被创建时赋予的。MPI 系统提供了一个特殊进程号 `MPI_PROC_NULL`，它代表空进程 (不存在的进程)，与 `MPI_PROC_NULL` 进行通信相当于一个空操作，对程序的运行没有任何影响，它的引入可以方便一些程序的编写。

MPI 程序中进程间的通信 (communications) 通过消息的收发或同步操作完成。一个消息 (message) 指在进程间进行的一次数据交换。在 MPI 中，一个消息由通信器、源地址、目的地址、消息标签

和数据构成。

3.2.2 程序基本结构

一个 MPI 程序的各个进程通过调用 MPI 函数进行通信，协同完成一项计算任务。

在 MPI 的 C 语言接口中，所有函数名均采用 `MPI_Xxxxx` 的形式，如 `MPI_Send`，`MPI_Type_commit` 等等，它们以 `MPI_` 开始，以便与其他函数名相区别，前缀 `MPI_` 之后的第一个字母大写，其余字母一律小写。MPI 的 Fortran 接口定义为一组 `SUBROUTINE`，名称与 C 接口函数相同，由于 Fortran 语言中不区分字母的大小写，因此 MPI 函数、变量的名称写成大写或小写均可，不过建议在 Fortran 程序中也采用与 C 语言一致的大小写方式。

MPI 的 C 语言接口函数通常返回一个整数值表示操作成功与否，返回值为 `MPI_SUCCESS` (0) 表示操作成功，否则表示操作的错误码。Fortran 接口比相应的 C 接口函数多出一个整型参数，用于返回错误码。唯一两个例外是 `MPI_Wtime` 和 `MPI_Wtick`，它们在 C 和 Fortran 中均采用函数的形式，返回值类型分别为 `double` (C) 和 `DOUBLE PRECISION` (Fortran)。

MPI 接口中除了函数和 `SUBROUTINE` 外，还定义了一组常量及 C 变量类型，它们的命名规则为：所有常量的名称全部大写，如 `MPI_COMM_WORLD`，`MPI_INT` 等；而 C 变量类型的命名则与 C 函数一样，如 `MPI_Datatype`，`MPI_Status` 等。

1. C 语言 MPI 程序结构

下面是 C 语言 MPI 程序的典型结构

```
#include "mpi.h"
... ...
int main(int argc, char *argv[])
{
```

```
    int myrank, nprocs;
    ... ...
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    ... ...
    MPI_Finalize();
    ... ...
}
```

用 C 语言编写的 MPI 程序中每个源文件必须包含 MPI 的 C 语言头文件 `mpi.h`，以便得到 MPI 函数的原型说明及 MPI 预定义的常量和类型。注意源文件中包含头文件 "`mpi.h`" 时不要含路径，必要时可在编译时通过 "`-I`" 选项指定 `mpi.h` 所在的路径，以方便程序在不同 MPI 系统间的移植。

`MPI_Init` 函数用于初始化 MPI 系统。在调用其他 MPI 函数前 (除 `MPI_Initialized` 外) 必须先调用该函数。在许多 MPI 系统中，第一个进程通过 `MPI_Init` 来启动其他进程。注意要将命令行参数的地址 (指针) `&argc` 和 `&argv` 传递给 `MPI_Init`，因为 MPI 程序启动时一些初始参数是通过命令行传递给进程的，这些参数被添加在命令行参数表中，`MPI_Init` 通过它们得到 MPI 程序运行的相关信息，如需要启动的进程数、使用那些结点、以及进程间的通信端口等，返回时会将这些附加参数从参数表中去掉。因此一个 MPI 程序如果需要处理命令行参数，最好在调用 `MPI_Init` 之后再进行处理，这样可以避免遇到 MPI 系统附加的额外参数。

函数 `MPI_Comm_size` 与 `MPI_Comm_rank` 分别返回指定通信器 (这里是 `MPI_COMM_WORLD`，它包含了所有进程) 中进程的数目以及本进程的进程号。

`MPI_Finalize` 函数用于退出 MPI 系统。调用 `MPI_Finalize` 之后不能再调用任何其他 MPI 函数。

2. Fortran 语言 MPI 程序结构

以下是 Fortran 语言 MPI 程序的典型结构：

```
PROGRAM MPIPRG
INCLUDE 'mpif.h'
INTEGER MYRANK, NPROCS, IERR
... ...
CALL MPI_Init(IERR)
CALL MPI_Comm_rank(MPI_COMM_WORLD, MYRANK, IERR)
CALL MPI_Comm_size(MPI_COMM_WORLD, NPROCS, IERR)
... ...
CALL MPI_Finalize(IERR)
... ...
END
```

其结构与含义与 C 语言程序是完全对应的。注意 Fortran 接口的 MPI 头文件是 `mpif.h`。本节其余部分将主要使用 C 语言接口来介绍 MPI 的重要概念及函数。

3.2.3 MPI 的原始数据类型

MPI 系统中数据的发送与接收操作都必须指定数据类型。数据类型可以是 MPI 系统预定义的，称为原始数据类型，也可以是用户在原始数据类型的基础上自己定义的数据类型。

MPI 为 C 和 Fortran 77 预定义的原始数据类型在表 3.1 中给出。除表中列出的外，某些 MPI 系统可能支持更多的原始数据类型，如 `MPI_INTEGER2`，`MPI_LONG_LONG_INT`，等等。

3.2.4 点对点通信函数与通信模式

MPI 最基本的通信模式是在一对进程之间进行的消息收发操作：一个进程发送消息，另一个进程接收消息。这种通信方式称为点对点通信 (point to point communications)。

表 3.1 MPI 原始数据类型

(a) C 数据类型

MPI 数据类型	C 类型
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_SHORT	short
MPI_LONG	long
MPI_CHAR	char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned
MPI_UNSIGNED_LONG	unsigned long
MPI_LONG_DOUBLE	long double
MPI_BYTE	unsigned char
MPI_PACKED	无

(b) Fortran 77 数据类型

MPI 数据类型	Fortran 77 类型
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER*1
MPI_BYTE	
MPI_PACKED	

MPI 提供两大类型的点对点通信函数。第一种类型称为阻塞型 (blocking)，第二种类型称为非阻塞型 (non blocking)。

阻塞型函数需要等待指定操作的实际完成，或至少所涉及的数据已被 MPI 系统安全地备份后才返回。如 `MPI_Send` 和 `MPI_Recv` 都是阻塞型的。`MPI_Send` 函数返回时表明数据已经发出或已被 MPI 系统复制，随后对发送缓冲区的修改不会影响所发送的数据。而 `MPI_Recv` 返回时，则表明数据已经接收到并且可以立即使用。阻塞型函数的操作是非局部的，它的完成可能需要与其他进程进行通信。

非阻塞型函数调用总是立即返回，而实际操作则由 MPI 系统在后台进行。非阻塞型函数名 `MPI_` 前缀之后的第一个字母为 "`I`"，最常用的非阻塞型点对点通信函数包括 `MPI_Isend` 和 `MPI_Irecv`。在调用了一个非阻塞型通信函数后，用户必须随后调用其他函数，如 `MPI_Wait` 或 `MPI_Test` 等，来等待操作完成或查询操作的完成情况。在操作完成之前对相关数据区的操作是不安全的，因为随时可能与正在后台进行的通信发生冲突。非阻塞型函数调用是局部的，因为它的返回不需要与其他进程进行通信。在有些并行系统上，通过非阻塞型函数的使用可以实现计算与通信的重叠进行。

此外，对于点对点消息发送，MPI 提供四种发送模式，这四种发送模式的相应函数具有一样的调用参数，但它们发送消息的方式或对接收方的状态要求不同。

标准模式 (standard mode) 由 MPI 系统来决定是先将消息拷贝至一个缓冲区然后立即返回 (此时消息的发送由 MPI 系统在后台进行)，还是等待将数据发送出去后再返回。大部分 MPI 系统预留了一定大小的缓冲区，当发送的消息长度小于缓冲区大小时会将消息拷贝到缓冲区然后立即返回，否则则当部分或全部消息发送完成后才返回。标准模式发送操作是非局部的，

因为它的完成需要与接收方联络。标准模式阻塞型发送函数是 MPI_Send。

缓冲模式 (buffered mode) MPI 系统将消息拷贝至一个用户提供的缓冲区然后立即返回，消息的发送由 MPI 系统在后台进行。用户必须确保所提供的缓冲区足以容下采用缓冲模式发送的消息。缓冲模式发送操作是局部的，因为函数不需要与接收方联络即可立即完成 (返回)。缓冲模式阻塞型发送函数为 MPI_Bsend。

同步模式 (synchronous mode) 在标准模式的基础上要求确认接收方已经开始接收数据后函数调用才返回。显然，同步模式的发送是非局部的。同步模式阻塞型发送函数为 MPI_Ssend。

就绪模式 (ready mode) 调用就绪模式发送时必须确保接收方已经处于就绪状态 (正在等待接收该消息)，否则将产生一个错误。该模式设立的目的是在一些以同步方式工作的并行系统上由于发送时可以假设接收方已经准备好接收而减少一些握手开销。如果一个使用就绪模式的 MPI 程序是正确的，则将其中所有就绪模式的消息发送改为标准模式后也应该是正确的。就绪模式阻塞型发送函数为 MPI_Rsend。

表 3.2 中汇总了 MPI 各种类型、各种模式的点对点通信函数。

使用阻塞型标准模式消息收发函数 MPI_Send 和 MPI_Recv 时要特别小心，使用不当很容易引起发送和接收操作不匹配从而导致程序死锁。例如下面的例子。

代码 3.1: 标准通信模式可能出现死锁的程序实例。

文件名: code/mpi/deadlock.c

```
1 #include <stdio.h>
2 #include <string.h>
```

表 3.2 MPI 点对点通信类型及模式汇总

函数类型	通信模式	阻塞型	非阻塞型
消息发送函数	标准模式	MPI_Send	MPI_Isend
	缓冲模式	MPI_Bsend	MPI_Ibsend
	同步模式	MPI_Ssend	MPI_Issend
	就绪模式	MPI_Rsend	MPI_Irsend
消息接收函数		MPI_Recv	MPI_Irecv
消息检测函数		MPI_Probe	MPI_Iprobe
等待通信完成或查询完成情况		MPI_Wait	MPI_Test
		MPI_Waitall	MPI_Testall
		MPI_Waitany	MPI_Testany
		MPI_Waitsome	MPI_Testsome
释放通信请求		MPI_Request_free	
取消通信			MPI_Cancel
			MPI_Test_cancelled

```
3  #include "mpi.h"
4
5  #define SIZE 16
6
7  int
8  main(int argc, char **argv)
9  {
10     static int buf1[SIZE], buf2[SIZE];
11     int nprocs, rank, tag, src, dst;
12     MPI_Status status;
13
14     MPI_Init(&argc, &argv);
15     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);     /* 获取总进程数 */
16     MPI_Comm_rank(MPI_COMM_WORLD, &rank);       /* 获取本进程的进程号 */
17
18     /* 初始化 buf1 */
```

```
19      memset(buf1, 1, SIZE);
20
21      tag = 123;
22      dst = (rank > nprocs - 1) ? 0 : rank + 1;
23      src = (rank == 0) ? nprocs - 1 : rank - 1;
24      MPI_Send(buf1, SIZE, MPI_INT, dst, tag, MPI_COMM_WORLD);
25      MPI_Recv(buf2, SIZE, MPI_INT, src, tag, MPI_COMM_WORLD, &status);
26
27      MPI_Finalize();
28
29      return 0;
30  }
```

代码 3.1 中的程序在大部分 MPI 系统上不会出问题，这是因为发送的数据量很少，MPI 系统会将待发送的数据复制到一个内部缓冲区，因此函数 MPI_Send 会立即返回，而实际发送在后台进行，各进程接着分别会调用 MPI_Recv 函数来接收对方发来的数据，完成通信。但是如果加大消息中的数据量，例如将

```
#define SIZE 16
```

改成

```
#define SIZE (16*1024*1024)
```

则程序很可能会死锁，因为此时数据长度超过了 MPI 的内部缓冲区长度，MPI_Send 函数无法立即返回，所有进程都在等待对方接收。具体当 SIZE 多大时会出现死锁与 MPI 系统有关。如果将程序中的 MPI_Send 和 MPI_Recv 调用的顺序颠倒，则程序在任何情况下都会死锁，因为所有进程都在等待接收数据，没有进程发送数据。解决这类通信死锁问题的一个方法是调整消息收发的顺序使得消息的接收与发送正好配对，例如在上面的例子中，可以让偶数号的进程先发后收，奇数号的进程先收后发。但对于实际应用中的复杂情况，往往不容易做到消息发送与接收的完美匹配，此时最好使用 MPI_Sendrecv

函数或者非阻塞型消息收发函数 (MPI_Isend 和 MPI_Irecv)。例如，如果将上例中的 MPI_Send 和 MPI_Recv 调用替换成

```
MPI_Sendrecv(buf1, SIZE, MPI_INT, dst, tag, buf2, SIZE, MPI_INT, src, tag,
    MPI_COMM_WORLD, &status);
```

就可以有效避免程序出现死锁。

许多算法中，程序会以相同的方式重复进行一些消息传递，这些消息使用完全一样的的通信器、收发缓冲区、数据类型与长度、源/目的地址和消息标签。MPI 提供持久通信 (persistent communications) 的方式来简化这一类操作，目的是减少处理消息的开销及简化 MPI 程序。持久通信的工作原理是首先提交一个持久通信请求，其中包括对消息的详细描述，MPI 系统返回给程序该持久通信请求的句柄，随后可以反复使用该句柄来进行多次数据传递。持久通信请求提交函数有持久消息发送请求函数 MPI_Send_init 和持久消息接收请求函数 MPI_Recv_init。MPI_Send_init 是标准模式持久消息发送请求，类似地还有缓冲模式 MPI_Bsend_init，同步模式 MPI_Ssend_init 和就绪模式 MPI_Rsend_init。一个持久通信请求提交后，调用 MPI_Start 或 MPI_Startall 便可完成消息的实际发送。持久通信请求与其他请求一样，当不再需要它们时应该调用 MPI_Request_free 函数释放它们所占用的资源。

3.2.5 聚合通信与同步

聚合通信指在一个通信器的所有进程间同时进行的通信。聚合通信总是在一个通信器中的所有进程间进行，调用一个聚合通信函数时，通信器中的所有进程必须同时调用同一函数，共同参与操作。聚合通信包括障碍同步 (MPI_Barrier)、广播 (MPI_Bcast)、数据收集 (MPI_Gather)、数据散发 (MPI_Scatter)、数据转置 (MPI_Alltoall) 和归约 (MPI_Reduce)。

1. 障碍同步

障碍同步函数 MPI_Barrier 用于一个通信器中所有进程的同步。调用该函数时进程将处于等待状态，直到通信器中所有进程都调用了该函数后才继续执行。

2. 广播

指一个进程 (称为根进程) 同时发送同样的消息给通信器中的所有其他进程。MPI 的广播函数是 MPI_Bcast。

3. 数据收集

数据收集操作指一个进程，称为根进程，从指定通信器中的所有进程，包括根进程自己，收集数据。MPI 的基本数据收集函数为 MPI_Gather，它从每个进程收集相同长度的数据。如果从各个进程收集的数据长度不同，则应该调用函数 MPI_Gatherv。

函数 MPI_Allgather 用于在通信器中的所有进程中同时进行数据收集，它的作用相当于先用 MPI_Gather 将数据收集到一个进程中，紧接着用 MPI_Bcast 将收集到的数据广播给其他进程。类似地，MPI_Allgatherv 用于收集不同长度的数据到通信器中的所有进程中。

4. 数据散发

数据散发函数 MPI_Scatter 正好是数据收集函数 MPI_Gather 的逆向操作，它将一个进程中的数据按块散发给通信器中的所有进程，散发给每个进程的数据块长度相同。函数 MPI_Scatterv 用于散发不同长度的数据块。

5. 数据转置

函数 MPI_Alltoall 用于同时进行收集和散发操作：通信器中所有进程从其他进程收集数据，同时将自己的数据散发给其他进程。

它的作用相当于将一个分布式存储的数据场在处理机间进行一次转置。函数 MPI_Alltoall 要求参与操作的所有数据块长度一样，如果数据块长度不同，则应该调用 MPI_Alltoallv 函数。

6. 归约

归约运算是指在分布在不同进程的数据间进行指定的运算，常用的运算有求和、求最大或最小值等。MPI 的归约函数中可以使用预定义的运算 (如 MPI_SUM，MPI_MAX 等，参看 B.2.4)，也可以使用用户自行定义的运算 (参看 MPI_Op_create)。MPI 用于归约操作的基本函数是 MPI_Reduce，它在指定的进程 (称为根进程) 中返回归约运算结果。如果希望所有进程都得到归约运算的结果，则可使用函数 MPI_Allreduce。

此外，MPI 还提供一个函数 MPI_Scan，称为前缀归约或扫描归约，用于计算数据的部分和。

3.2.6 自定义数据类型

MPI 系统的原始数据类型只适合于收发一组在内存中连续存放的数据。当要收发的数据在内存中不连续，或由不同数据类型构成时，则需要将数据打包或者使用自定义的数据类型。自定义数据类型用于描述要发送或接收的数据在内存中的确切分布。数据类型是 MPI 的一个重要特征，它的使用可有效地减少消息传递的次数，增大通信粒度，并且，与数据打包相比，在收/发消息时可以避免或减少数据在内存中的拷贝、复制。

一个 MPI 的数据类型采用递归的方式定义，它由两个 n 元序列构成，其中 n 为正整数。第一个序列是一组已定义的数据类型，称为类型序列 (type signatures)：

$$\mathtt{Typesig} = \{\mathtt{type}_0, \mathtt{type}_1, \ldots, \mathtt{type}_{n-1}\}$$

第二个序列是一组整数位移值，称为位移序列 (type displacements)：

$$\texttt{Typedisp} = \{\texttt{disp}_0, \texttt{disp}_1, \ldots, \texttt{disp}_{n-1}\}$$

注意，位移序列中位移是以字节为单位表示的。

构成类型序列的数据类型称为该数据类型的基本数据类型，它们可以是原始数据类型，也可以是任何已定义的数据类型。为叙述方便起见，称非原始数据类型为复合数据类型。

类型序列刻划了数据的类型特征，包括数据的类型与大小。位移序列则刻划了数据的位置特征。类型序列和位移序列的元素一一配对构成的序列：

$$\texttt{Typemap} = \{(\texttt{type}_0, \texttt{disp}_0), (\texttt{type}_1, \texttt{disp}_1), \ldots, (\texttt{type}_{n-1}, \texttt{disp}_{n-1})\}$$

称为类型图 (type map)。假设数据缓冲区的起始地址为 `buff`，则由上述类型图所定义的数据类型包含 n 块数据，第 i 块数据的地址为 $\texttt{buff} + \texttt{disp}_i$，类型为 $\texttt{type}_i$，$i = 0, 1, \ldots, n-1$。

MPI 的原始数据类型的类型图可以写成 $\{(\text{类型}, 0)\}$ 的形式。例如，原始数据类型 `MPI_INTEGER` 的类型图为 `{(MPI_INTEGER,0)}`。为简洁起见，后文在类型图中有时会省略 MPI 原始数据类型的 “`MPI_`” 前缀，如将 `MPI_INTEGER` 的类型图写成 `{(INTEGER,0)}`。

位移序列中的位移不必是单调上升的，表明数据类型中的数据块不要求按顺序排放。位移也可以是负的，即数据类型中的数据可以位于缓冲区起始地址之前。

数据类型的大小 (size) 指该数据类型中包含的数据长度 (字节数)，它等于类型序列中所有基本数据类型的大小之和。数据类型的大小就是消息传递时需要实际发送或接收的数据长度。假设数据类型 `type` 的类型图为：

$$\{(\texttt{type}_0, \texttt{disp}_0), (\texttt{type}_1, \texttt{disp}_1), \ldots, (\texttt{type}_{n-1}, \texttt{disp}_{n-1})\}$$

则该数据类型的大小为：

$$\texttt{size}(\texttt{type}) = \sum_{i=0}^{n-1} \texttt{size}(\texttt{type}_i)$$

数据类型的下界 (lower bound) 定义为数据的最小位移。数据类型的上界 (upper bound) 定义为数据的最大位移加 1，再加上一个使得数据类型满足操作系统地址对界要求 (alignment) 的修正量 ε。数据类型的域 (extent) 定义为上界与下界之差：

$$\begin{aligned}
\texttt{lb}(\texttt{type}) &= \min_i\{\texttt{disp}_i\} \\
\texttt{ub}(\texttt{type}) &= \max_i\{\texttt{disp}_i + \texttt{sizeof}(\texttt{type}_i)\} + \varepsilon \\
\texttt{extent}(\texttt{type}) &= \texttt{ub}(\texttt{type}) - \texttt{lb}(\texttt{type})
\end{aligned}$$

其中，地址对界修正量 ε 是使得数据类型的域能被该数据类型的对界量整除的最小非负整数。

一个数据类型的对界量定义如下：原始数据类型的对界量由系统决定，通常取决于计算机的体系结构，而复合数据类型的对界量则定义为构成它的所有基本数据类型的对界量的最大值。地址对界要求指一个数据在内存中的 (字节) 起始地址必须是它的对界量的整数倍，其主要目的是为了优化内存访问。例如在 C 语言的结构 (`struct`) 中，编译器通常会自动在结构的每个成员后面填入适当的空间 (称为 padding)，使得它们满足对界要求。编译、运行代码 3.2，可以观察 C 语言中的变量对界情况。

代码 3.2: C 语言中的变量对界。

文件名: `code/mpi/padding.c`

```
1 #include <stdio.h>
2
3 typedef struct {
```

```
    double d;
    char   c;
} CS;

typedef struct {
    char   c1;
    double d;
    char   c2;
} CS1;

int
main()
{
  CS  a;
  CS1 b;

  printf("sizeof(CS)=%d\n", sizeof(CS));
  printf("offset(a.d)=%d, offset(a.c)=%d\n",
        (char *)&a.d - (char *)&a, (char *)&a.c - (char *)&a);

  printf( "sizeof(CS1)=%d\n", sizeof(CS1));
  printf( "offset(b.c1)=%d, offset(b.d)=%d, offset(b.c2)=%d\n",
        (char *)&b.c1 - (char *)&b, (char *)&b.d  - (char *)&b,
        (char *)&b.c2 - (char *)&b);

  return 0;
}
```

例如，假设 MPI_DOUBLE_PRECISION 和 MPI_INTEGER 的对界量均为 4，MPI_BYTE 的对界量为 1，则类型图

{(DOUBLE_PRECISION,0),(INTEGER,8),(BYTE,12)}

的对界量为 4，下界为 0，上界为 16，域为 16，$\varepsilon = 3$。

MPI 系统提供了两个特殊数据类型 MPI_LB 和 MPI_UB，称为伪数据类型 (pseudo datatype)。它们的大小都是 0，当它们出现在一个

类型图中时不影响数据类型的实际数据，其作用是指定数据类型的上界或下界。MPI 规定：如果一个数据类型 type 的基本类型中含有 MPI_LB，则它的下界定义为：

$$\mathtt{lb(type)} = \min_i\{\mathtt{disp}_i \mid \mathtt{type}_i = \mathtt{MPI\_LB}\}$$

类似地，如果一个数据类型 type 的基本类型中含有 MPI_UB，则它的上界定义为：

$$\mathtt{ub(type)} = \max_i\{\mathtt{disp}_i \mid \mathtt{type}_i = \mathtt{MPI\_UB}\}$$

例如，类型图

```
{(LB,-4),(UB,20),(DOUBLE_PRECISION,0),
 (INTEGER,8),(BYTE,12)}
```

的下界为 −4，上界为 20，域为 24。

MPI 提供了一组函数，以 “MPI_Type_” 为前缀，用于构造新的数据类型。MPI–1 提供的数据类型构造函数包括：

```
MPI_Type_contiguous
MPI_Type_vector，MPI_Type_hvector
MPI_Type_indexed，MPI_Type_hindexed
MPI_Type_struct
```

有关它们的说明请参看附录 B。对在内存中任意分布的数据，借助 MPI 提供的数据类型创建函数，均可定义出相应的 MPI 数据类型进行描述。用户自定义的 MPI 数据类型在首次用于通信前，必须调用 MPI_Type_commit 函数进行提交。一个数据类型在被提交后就可以和 MPI 原始数据类型完全一样地用于消息传递中。如果一个数据类型仅仅用于创建其他数据类型的中间步骤而并不直接在消息传递中使用，则不必将它提交，一旦基于它的其他数据类型创建完毕即

可立即将它释放。当一个自定义的数据类型不再需要时，应该调用 MPI_Type_free 函数将其释放，以免无谓地占用系统资源。

MPI 还提供了另外一个特殊数据类型 MPI_PACKED 用于数据打包。用户可以调用函数 MPI_Pack 以类似于 PVM 的方式将不同的数据进行打包然后再发送出去，接收方在收到消息后调用 MPI_Unpack 函数对数据进行拆包。一般不推荐使用数据打包函数，因为它可能增加内存中的数据拷贝从而降低通信性能。

在 MPI–2 中，MPI 数据类型除用于通信外，还用于文件输入输出操作，参看 3.2.9。

在许多需要使用使用地址的 MPI 函数中，例如 MPI_Address、MPI_Type_hvector 等，C 与 Fortran 77 对一些参数使用了不同的变量类型：它们在 C 函数中的类型为 MPI_Aint，而在 Fortran 77 中的类型为 MPI_INTEGER。MPI_Aint 是用于位移及数据大小操作的 C 变量类型，因为一些 64 位系统中用 int 存储地址、数据大小或位移可能不够，在这些系统上 MPI_Aint 被定义成与地址空间的大小相对应的整数类型，而在 32 位系统中 MPI_Aint 则通常定义为 int。因此，统一用 MPI_Aint 来表示地址、大小或位移的程序在 32 位与 64 位操作系统间是可移植的。

MPI–1 没有为 Fortran 77 提供相对应的变量类型，而是统一使用 INTEGER。因此如果一个 Fortran 77 程序使用了这些函数的话，它在 32 与 64 位操作系统间的可移植性可能会受到影响，并且用 Fortran 77 接口函数处理超过 2GB (或 4GB) 的位移、数据大小等参数时也会产生问题。

此外，一些 C 接口的定义也有问题，如 MPI_Type_size 中 size 类型为 int *，这就隐含限制了它返回的值不能超过 2GB。

在 MPI–2 中对这些问题进行了澄清，并且引入了一组新接口函数，如 MPI_Type_create_hvector，MPI_Get_address 等，这些函数的 Fortran 接口标准采用 Fortran 90 的形式，例如：

```
INTEGER(kind=MPI_ADDRESS_KIND)
```

相关的讨论请参看 MPI–2 文档 [19]。

3.2.7 进程组与通信器

通信器 (communicator) 构成 MPI 消息传递的基本环境，所有通信都是在特定的通信器中进行。上下文是通信器的一个固有性质，它为通信器划分出特定的通信空间，消息在一个给定的上下文中进行传递，不同上下文间不允许进行消息的收发，这样可以确保不同通信器中的消息不会混淆。此外，MPI 要求点对点通信与聚合通信是独立的，它们间的消息不会互相干扰。上下文对用户是不可见的，它是 MPI 实现的一个内部概念。

进程组是一组进程的有序集合，它定义了通信器中进程的集合及进程号。MPI 中进程组与通信器类似通过句柄来进行操作。MPI 预定义了两个进程组句柄：一个是 `MPI_GROUP_EMPTY`，它代表由空进程组集合构成的进程组，另一个是 `MPI_GROUP_NULL`，表示非法进程组。

需要注意的是，通信器和进程组的句柄是进程所固有的，只对本进程有意义。因此将通信器或进程组的句柄通过通信在进程间进行传递是没有意义的。

MPI 的通信器分为域内通信器 (intra-communicator) 和域间通信器 (inter-communicator)。

域内通信器由进程组和上下文构成。一个通信器的进程组中必须包含定义该通信器的进程作为其成员。此外，为了优化通信以及支持处理机实际的拓扑连接方式，通信器中还可以定义一些附加属性，如进程间的拓扑连接方式等。对于用户而言经常使用的通信器属性是进程拓扑结构。域内通信器可以用于点对点通信，也可以用于聚合通信。MPI 系统预定义了二个域内通信器，它们是 `MPI_COMM_WORLD` 和 `MPI_COMM_SELF`，前者包含构成并行程序的所有进程，后者包含单

个进程 (各进程自己)，其他通信器可在这两个通信器的基础上构建。另外，MPI_COMM_NULL 代表非法通信器。

域间通信器用于分属于不同进程组的进程间的点对点通信。一个域间通信器由两个进程组构成。域间通信器不能定义进程的拓扑连接信息，也不能用于聚合通信。

MPI 标准中定义了一组函数用于通信器及进程组操作的函数，其名称具有 MPI_Comm_xxxx 和 MPI_Group_xxxx 的形式，关于这些函数请参看附录 B。

3.2.8 进程拓扑结构

进程拓扑结构是 (域内) 通信器的一个附加属性，它描述一个通信器各进程间的逻辑连接关系。进程拓扑结构的使用一方面可以方便、简化一些并行程序的编制，另一方面可以帮助 MPI 系统更好地将进程映射到处理机以及组织通信的流向，从而获得更好的通信性能。

MPI 的进程拓扑结构定义为一个无向图，图中结点 (node) 代表进程，而边 (edge) 则代表进程间的连接。MPI 进程拓扑结构也被称为虚拟拓扑结构，因为它不一定对应处理机的物理连接。

MPI 提供了一组函数用于创建各种进程拓扑结构。应用问题中较为常见、也是较为简单的一类进程拓扑结构具有网格形式，这类结构中进程可以用笛卡尔坐标来标识，MPI 中称这类拓扑结构为笛卡尔拓扑结构 (cartesian topology)，并且专门提供了一组函数对它们进行操作，这些函数的名称为 MPI_Cart_xxxx 的形式，如 MPI_Cart_create，MPI_Cart_coords 等。

MPI 提供的用于一般拓扑结构操作的函数名为 MPI_Graph_xxxx 的形式，如 MPI_Graph_create，MPI_Graph_map 等。

3.2.9 文件输入/输出

MPI–2 标准中定义了一组文件输入输出 (I/O) 函数，函数接口定义包含 C，C++，和 Fortran 三种。出于严谨性考虑，Fortran 接口中一些参数使用了 Fortran 90 类型 (如 `INTEGER(MPI_OFFSET_KIND)`)，而 Fortran 77 代码中这些参数在不同的平台上可能需要采用不同的写法 (如 `INTEGER*4`，`INTEGER*8` 等等)，会影响代码的可移植性。

本节介绍的函数在 MPICH 1.2.1 以后的版本中已经全部实现，但在有的 MPI 版本中可能还不能使用。因此，在使用本节介绍的函数前应先确认所使用的 MPI 系统是否支持它们。在有些 MPI 版本中使用文件输入输出函数还可能需要包含额外的头文件，例如使用 MPICH 1.2.1 的 Fortran 程序需要包含头文件 "`mpiof.h`"，而 MPICH 1.2.2 以后的版本则不用，请根据所使用的 MPI 系统选择适当的头文件。

下面是 MPI–2 有关文件操作的一些基本术语。

文件 (file) MPI 的"文件"可以看成由具有相同或不同类型的数据项构成的序列。MPI 支持对文件的顺序和随机访问。MPI 的文件是和进程组相关联的：MPI 打开文件的函数 (`MPI_File_open`) 中要求指定一个通信器，并且该通信器中所有进程必须同时对文件进行打开或关闭操作。

起始位置 (displacement) 一个文件的起始位置指相对于文件开头以字节为单位的一个绝对地址，它用来定义"文件视窗"的起始位置。

基本单元类型 (etype) 基本单元类型 (elementary type) 是定义一个文件最小访问单元的 MPI 数据类型。一个文件的基本单元类型可以是任何预定义或用户构造的并且已经递交的 MPI 数据类型，但其类型图中的位移必须非负并且位移序列是 (非严格)

单调上升的。MPI 的文件操作完全以基本单元类型为单位：文件中的位移 (offset) 以基本单元的个数而非字节数为单位，文件指针总是指向一个基本单元的起始地址。

文件单元类型 (filetype) 文件单元类型也是一个 MPI 数据类型，它定义了对一个文件的存取图案。文件单元类型可以等于基本单元类型，也可以是在基本单元类型基础上构造并已递交的任意 MPI 数据类型。文件单元类型的域必须是基本单元类型的域的倍数，并且文件单元类型中间的“洞”的大小和位置也必须是基本单元类型的域的倍数。

视窗 (view) 文件视窗指一个文件中当前可以访问的数据集。文件视窗由 3 个参数定义：起始位置，基本单元类型，文件单元类型。文件视窗指从起始位置开始将文件单元类型连续重复排列构成的图案，MPI 对文件进行存取操作时将“跳过”图案中的“空白”。

位移 (offset) MPI 的输入输出函数中位移总是相对于文件起始位置 (当前视窗) 计算，并且以基本单元类型的域为单位。

文件大小 (file size) 文件大小指从文件开头到文件结尾的字节数。

文件指针 (file pointer) MPI 的文件指针是由 MPI 管理的内部位移 (隐式位移)，用于记录文件当前的位置。MPI 在每个进程中为每个打开的文件定义了两个文件指针，一个供本进程独立使用，称为独立文件指针 (individual file pointer)，另一个供打开文件的进程组中的所有进程共同使用，称为共享文件指针 (shared file pointer)。

文件句柄 (file handle) MPI 打开一个文件后，返回给调用程序一个文件句柄，供以后访问及关闭该文件时用。MPI 的文件句柄

在文件关闭时被释放。

例如，假设 ext(MPI_REAL) = 4，etype = MPI_REAL，打开文件 fh 的进程组包括 4 个进程 p_i，$i = 0, 1, 2, 3$，四个进程中文件单元类型分别定义如下：

$$
\begin{aligned}
p_0: &\quad \text{filetype} = \{(\texttt{MPI\_REAL},\ 0),\quad (\texttt{MPI\_LB}, 0),\quad (\texttt{MPI\_UB}, 16)\}\\
p_1: &\quad \text{filetype} = \{(\texttt{MPI\_REAL},\ 4),\quad (\texttt{MPI\_LB}, 0),\quad (\texttt{MPI\_UB}, 16)\}\\
p_2: &\quad \text{filetype} = \{(\texttt{MPI\_REAL},\ 8),\quad (\texttt{MPI\_LB}, 0),\quad (\texttt{MPI\_UB}, 16)\}\\
p_3: &\quad \text{filetype} = \{(\texttt{MPI\_REAL}, 12),\quad (\texttt{MPI\_LB}, 0),\quad (\texttt{MPI\_UB}, 16)\}
\end{aligned}
$$

如果四个进程中独立文件指针均为 0，并且它们同时调用函数：

```
MPI_File_read(fh, &a, 1, MPI_REAL, &status)
```

则文件开头的四个数被依次赋给四个进程中的变量 a。

MPI–2 对一个文件的操作与普通操作系统对文件操作的步骤类似，即调用函数 MPI_File_open 打开或创建文件并得到用于文件访问的句柄，然后调用函数 MPI_File_set_view 设定文件视窗，调用函数 MPI_File_*read* 或 MPI_File_*write* 对文件进行读或写操作，所有操作完成后调用函数 MPI_File_close 关闭文件。与普通操作系统不同的是，MPI–2 打开、关闭文件时必须是一个通信器中的所有进程同时打开和关闭同一个文件。此外，普通操作系统只有一个文件指针，而 MPI–2 有两个文件指针，即独立文件指针和共享文件指针。关于 MPI 文件指针的使用将在稍后介绍。

除了基于文件指针的操作外，MPI–2 还允许在读写文件时直接指定文件中的位置，这一类操作称为基于显式位移的操作，相应的函数以 _at 为后缀，如 MPI_File_read_at 等。

此外，在用函数 MPI_File_set_view 设定文件视窗时可以通过字符串参数 datarep 指定文件中的数据格式，它可以取下面一些值：

"native" 文件中数据完全按其在内存中的格式存放。使用该数据格式的文件不能在数据格式不兼容的计算机间交换使用。

"internal" 指 MPI 内部格式，具体由 MPI 的实现定义。使用该数据格式的文件可以确保能在使用同一 MPI 系统的计算机间进行交换使用，即使这些计算机的数据格式不兼容。

"external32" 使用 IEEE 通用数据表示格式 external data representation (简称 XDR)。使用该数据格式的文件可以在所有支持 MPI 的计算机间交换使用。该格式可用于在数据格式不兼容的计算机间交换数据。

许多 MPI 系统目前尚未实现全部上述三种格式，但所有 MPI 系统都应该支持 "native" 格式。除上述数据格式外，还可以通过 MPI 函数 MPI_Register_datarep 定义其他数据格式。用户必须自行保证在设定文件窗口时指定的数据表示格式与文件中的实际数据表示格式相符。

特别需要注意的是，当 datarep 不等于 "native" 时，基本单元类型 (etype) 和文件单元类型 (filetype) 在文件中的形式有可能与它们在内存中的形式不一样。此时，如果用作基本单元类型的数据类型是“可移植的”(portable datatype)，则 MPI 的 I/O 函数会自动对数据类型的位移和域进行调整 (缩放) 以便与文件中的数据表示格式相匹配。如果用作基本单元类型的数据类型不是“可移植”的，则用户必须保证它们与文件中的数据表示格式相符，必要时使用 MPI_Type_lb 和 MPI_Type_ub 来进行调整。MPI 的原始数据类型都是可移植的，基于可移植数据类型使用下列函数

MPI_Type_contiguous，

MPI_Type_vector，

MPI_Type_indexed，

MPI_Type_dup

创建的新数据类型也是可移植的。因此，可移植数据类型的位移和上下界都是以某一预定义的数据类型为单位的。换言之，可移植的数据类型在其构造过程中不能使用下述函数：

MPI_Type_hindexed，

MPI_Type_hvector，

MPI_Type_struct

(即不能直接以字节为单位来设定数据类型的位移和上下界)。

表 3.3 汇总了 MPI 文件读写操作函数，其中 xxxx 代表 read 或 write，分别对应于读操作和写操作。

MPI 文件读写操作函数按指定文件位置的方式分为使用显式位移 (直接在函数中指定位移量，以基本单元类型的域为单位)、使用独立文件指针和使用共享文件指针三类。每种类型的操作不会对其他类型操作的位置产生影响，例如使用显式位移的操作不会改变独立文件指针或共享文件指针，使用独立文件指针的操作不会改变共享文件指针，而使用共享文件指针的操作也不会改变独立文件指针。

当一个进程使用独立文件指针或共享文件指针对文件进行操作时，文件中的位移由文件指针的当前值决定。操作完成后，该文件指针的值被自动刷新，指向文件中的下一个数据。独立文件指针是各进程私有的，它的刷新仅依赖于本进程，不受其他进程读写操作的影响。而共享文件指针则被进程组中所有进程共享，当多个进程同时使用共享文件指针进行读写时，每个进程的读写操作都会移动共享文件指针，文件指针总的移动量相当于所有读写操作的叠加。基于共享文件指针的读操作相当于从文件到各进程的数据散发，而基于共享文件指针的写操作则相当于从各进程到文件的数据收集。

文件读写操作函数按进程组中进程间的协同方式分为非聚合式 (noncollective) 和聚合式 (collective) 两种。非聚合式函数的完成只依赖于本进程，它们不要求进程组中的所有进程同时调用，而由各

表 3.3 MPI–2 文件读写函数汇总

定位方式	同步方式	进程组进程间的协同方式	
		非聚合式	聚合式
显式位移	阻塞型	MPI_Xxx_at	MPI_Xxx_at_all
	非阻塞或分裂型	MPI_Ixxx_at	MPI_Xxx_at_all_begin MPI_Xxx_at_all_end
独立指针	阻塞型	MPI_Xxx	MPI_Xxx_all
	非阻塞或分裂型	MPI_Ixxx	MPI_Xxx_all_begin MPI_Xxx_all_end
共享指针	阻塞型	MPI_Xxx_shared	MPI_Xxx_ordered
	非阻塞或分裂型	MPI_Ixxx_shared	MPI_Xxx_ordered_begin MPI_Xxx_ordered_end

注：“xxx”代表“read”或“write”，“Xxx”代表“Read”或“Write”。

进程分别独立地调用，当多个进程同时调用非聚合式函数时，不同进程间对数据读写的先后顺序是不确定的。而聚合式函数的完成依赖于同组所有进程间的协调，它们要求进程组中全部进程同时调用，各进程对数据读写的先后顺序由进程号确定。

按照函数调用是否阻塞，即函数是否需要等待操作结束后再返回，MPI 的文件读写函数又分为阻塞型 (blocking)、非阻塞型 (non-blocking) 和分裂型 (split) 三种。阻塞型函数返回后即表明读写操作已经“完成”，进程马上可以对数据区进行后续操作或关闭文件。非阻塞型文件读写函数与非阻塞型消息传递函数类似，只向系统发出一个读或写的请求，随后 (特别是在关闭文件前) 进程需要调用 MPI_Wait 或 MPI_Test 等函数来等待或检查操作的完成。分裂型函数将文件的读写操作分解成开始 (_begin) 和结束 (_end) 两步，以便允许进程在操作开始和结束之间进行其他计算或通信。

使用 MPI–2 输入输出函数时要特别注意当多个进程同时对同一个文件进行访问时的相容性。MPI 称一组访问是相容的，如果这些访问可以等效地被看成是以某种顺序依次进行的，即便它们的先后顺序是不确定的。换言之，对同一个文件的多个访问是相容的，如果正在进行的访问不会在操作过程中间由于被另一个访问打断或干扰而影响到访问的结果。用户可以调用函数 `MPI_File_set_atomicity` 设置对一个文件访问的“原子性”，它告诉 MPI 系统是否需要保证属于与该文件关联的进程组中的进程对该文件的访问的相容性。MPI 系统只能保证属于打开文件的进程组的进程间对该文件访问的原子性，如果同一个文件同时被不同的进程组打开，则当两个进程组对该文件的访问存在冲突时，用户必须通过其他手段 (如在程序中调用 `MPI_File_sync` 以及 `MPI_Barrier` 等) 来保证文件访问的相容性与访问顺序。

3.3 MPI 程序主要结构

并行程序设计是并行软件开发的基础之一，在不同的并行计算机以及不同的并行实现平台上，其实现方式是不同的。本节简单介绍 MPI 并行程序的基本结构。

MPI 并行程序和串行程序没有很大的差别，它们通过对 MPI 函数的调用来实现特定的并行算法。一个 MPI 并行程序主要由三个部分组成：

(1) 进入并行环境：调用 `MPI_Init` 来启动并行计算环境，它包括在指定的计算结点上启动构成并行程序的所有进程以及构建初始的 MPI 通信环境和通信器 `MPI_COMM_WORLD`、`MPI_COMM_SELF`。

(2) 主体并行任务：这是并行程序的实质部分，所有需要并行来完成的任务都在这里进行。在这个部分中，实现并行算法在并行

计算机上的执行过程。

(3) 退出并行环境：调用 `MPI_Finalize` 退出并行环境。一般说来，退出并行计算环境后程序的运行亦马上结束。

代码 3.3 是取自 MPICH 的一个程序实例，它展示了 C 语言 MPI 并行程序的结构。该程序用下面的公式计算定积分 $\pi = \int_0^1 4/(1+x^2)\,\mathrm{d}x$ 的近似值：

$$h \sum_{i=0}^{n-1} f(x_i) \tag{3.1}$$

其中 $n > 0$ 为积分区间数，$h = 1/n$ 为积分步长，$x_i = (i + 0.5)h$ $(i = 0, \ldots, n-1)$ 为积分区间的中点，被积函数 $f(x) = 4/(1+x^2)$。假设总进程数为 p (程序中的 `numprocs` 变量)，各进程分别负责计算式 (3.1) 中的一部分计算区间，然后再调用 `MPI_Reduce` 将各进程的结果加起来。代码中计算区间采用循环分配的方式，即将计算公式写成：

$$\sum_{k=0}^{p-1} \quad h \sum_{\substack{0 \leqslant j < n, \\ j \bmod p = k}} f(x_j)$$

每个进程独立计算上式中的一个内层求和，然后再将这些结果加起来。

代码 3.3： MPI 程序实例：数值积分 (π 值计算)。

文件名: `code/mpi/cpi.c`

```
1 /* 程序来源: MPICH examples/cpi.c */
2 #include "mpi.h"
3 #include <stdio.h>
4
5 double f( double a ) { return (4.0 / (1.0 + a*a)); }
6
```

```
int main( int argc, char *argv[])
{
    int n, myid, numprocs, i, namelen;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;
    double startwtime, endwtime;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s\n", myid, processor_name);
    if (myid == 0) {
        n=10000;
        startwtime = MPI_Wtime();
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid; 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) {
        endwtime = MPI_Wtime();
        printf("pi is approximately %.16f, error is %.16f\n", pi, pi - PI25DT);
        printf("wall clock time = %f\n", endwtime-startwtime);
    }
    MPI_Finalize();

    return 0;
}
```

MPI 并行程序从程序结构上可以分成三种编程模式，包括主从模式 (Master–slave)、单程序多数据模式 (SPMD，即 Single Program Multiple Data) 和多程序多数据模式 (MPMD，即 Multiple Programs Multiple Data)。这些编程模式既可以从源代码的组织形式来划分，也可以从实际程序所执行的代码来划分。它们之间有时并没有非常明确的界线。

如果从源代码的组织形式来划分，SPMD 模式的 MPI 程序中只有一套源代码，所有进程运行的都是该代码；master/slave 模式的 MPI 程序包含两套源代码，主进程运行其中一套代码，而从进程运行另一套代码；MPMD 模式则包含多套源代码，不同进程分别执行其中的一套代码。

如果从实际程序所执行的代码来划分，一个并行程序属于哪种编程模式，取决于程序中各进程实际执行的代码是否相同，以及是否具有 client/server 的特征。如果各进程执行的代码大体是一样的则可以看作 SPMD 模式，如果具有 client/server 特征则被认为是 master/slave 模式，否则则为 MPMD 模式。在这种编程模式划分中，master/slave 和 MPMD 模式也可以只用一套源码，不同进程执行的代码通过在程序中对进程号的条件判断来实现。实际编程时，相对于使用多套不同的源码而言，使用一套源码更便于代码的维护，并且 MPI 并行程序的启动也更方便，因为不需要分别指定哪个进程运行哪个可执行文件。

下面分别对这三种编程模式进行介绍与讨论。

1. 主从模式

构成并行程序的进程中有一个主进程 (master)，通常是进程 0，其余为从进程 (slave)。主进程与从进程运行不同的代码，但所有从进程运行的代码是相同的。在这种模式中，主进程一般负责整个并行程序的控制，分配数据和计算任务给从进程，而从进程负责完成

分配给它的数据的处理和计算任务。当然，主进程也可以参与对数据的处理和计算。

在主从模式中，从进程通常处于一个无限循环过程：它首先等待主进程给它分配计算任务，接到任务后，完成指定的计算，然后通知主进程所分配的计算已完成，接着继续等待主进程分配的新任务或结束指令。主进程将所有计算分解成一系列的计算单元，将一至数个计算单元依次分配给各从进程，然后等待从进程报告计算的完成，当一个从进程完成了所分配的计算后，从剩余的计算单元中分配新的计算任务给它，如果所有计算单元已分配完毕，则通知完成计算的从进程退出。实际实现中，数据可以存储在主进程中，分配任务时将数据一起发送给从进程，从进程完成计算后将结果发回给主进程；也可以将数据和结果分布存储在各从进程中，主进程只发布用于控制计算的少量参数。后一种通信方式往往要复杂一些，但它有助于避免在主进程中形成通信、内存容量方面的瓶颈。

主从模式便于处理某些动态负载平衡的问题，特别是在异构并行中各处理机的容量和速度不同时的负载平衡问题。但在大规模并行程序中，主进程需要管理大量从进程，容易成为性能瓶颈，影响并行可扩展性。主从模式可以看成一个二层树型模式，主进程是根，从进程是叶子，它的一个自然扩展是多层树型模式。例如在三层树形模式中，一个主进程管理数个“从”主进程，每个“从”主进程再分别管理数个从进程。多级主进程的设立有助于缓解或消除可扩展性瓶颈，但它也使得并行程序更加复杂。

2. SPMD 模式

在这种编程模式中，没有主从进程之分，各个进程的地位是相同的，它们运行的代码是一样的。当然事实上在实际并行实现中，总有一个进程，通常是进程 0，或多或少会担负一些基本控制任务。这种模式由于没有明显的性能瓶颈并且便于有效利用 MPI 的聚合通

信函数，往往能够达到理想的并行可扩展性，非常适合于大规模并行。

在 SPMD 模式中，各进程负责计算的部分通常由各进程根据它们的进程号以及总问题的规模来自动确定。在确定计算任务的划分方案时需要综合考虑各进程间的负载平衡、进程间的通信、进程间的数据相关性等多方面因素，以实现总体性能的最优。

3. MPMD 模式

并行程序的各进程分别运行多个不同的代码。不同的进程所执行的代码可能相同，也可能不同。这种模式在实际并行应用程序中比较少见。

习　题

1. 在装有 Linux 或 UNIX 操作系统的 PC 机或工作站上编译、安装 MPICH，并用它编译、运行其中自带的 MPI 程序实例，如 `cpi.c`、`pi3.f` 等 (如果没有超级用户权限的话可以将 MPICH 安装在自己的家目录中)。

2. 用 `MPI_Isend` 和 `MPI_Irecv` 代替代码 3.1 中的 `MPI_Send` 和 `MPI_Recv` 来避免出现程序死锁。

3. 设 Fortran 中 `REAL` 的长度为 4，数据类型 `TYPE` 的类型图为 `{(REAL,4),(REAL,12),(REAL,0)}`，则下面的语句：

```
REAL A(100)
... ...
CALL MPI_SEND(A, 1, TYPE, ...)
```

将发送哪些数据?

4. 设 Fortran 中 `REAL` 的长度为 4，数据类型 `TYPE` 的类型图为 `{(REAL,-4),(REAL,0),(REAL,4)}`，则下面的语句：

```
REAL A(3)
... ...
CALL MPI_SEND(A(2), 1, TYPE, ...)
```

将发送哪些数据?

5. 给定 Fortran 三维数组 `REAL*8 A(100,200,300)`，试定义三个数据类型 `type1`，`type2` 和 `type3`，它们分别描述三个方向的二维子数组 (相当于三维区域中的一个平面)：

$$\texttt{type1} \Rightarrow \{\texttt{A}(.,j,k) \mid 1 \leqslant j \leqslant 200, 1 \leqslant k \leqslant 300\}$$

$$\texttt{type2} \Rightarrow \{\texttt{A}(i,.,k) \mid 1 \leqslant i \leqslant 100, 1 \leqslant k \leqslant 300\}$$

$$\texttt{type3} \Rightarrow \{\texttt{A}(i,j,.) \mid 1 \leqslant i \leqslant 100, 1 \leqslant j \leqslant 200\}$$

6. 假设数据类型 `type1` 和 `type2` 由下面的 C 语言语句定义：

```
MPI_Type_vector(2, 2, 10, MPI_DOUBLE, &type1);
MPI_Type_vector(4, 2, 10, MPI_DOUBLE, &type2);
```

则

```
MPI_Send(buffer,2,type1,...)
```

和

```
MPI_Send(buffer,1,type2,...)
```

所发送的数据有何不同?

7. 修改代码 3.3，将任务分配改为按块分配：假设共有 P 个进程，将 n 个计算区间顺序分成 P 块，每个进程负责一块的计算。注意当 n 不是 P 的倍数时应该尽量保持负载平衡。

第 4 章　程序性能评价与优化

给定并行算法，采用并行程序设计平台，通过并行实现获得实际可运行的并行程序后，一个重要的工作就是，在并行机上运行该程序，评价该程序的实际性能，揭示性能瓶颈，指导程序的性能优化。性能评价和优化是设计高效率并行程序必不可少的重要工作。本章主要介绍当前流行的并行程序性能评价方法，并讨论有效的性能优化方法。

在此开始

4.1　并行程序执行时间

评价并行程序的性能之前，必须清楚并行程序的执行时间是由哪些部分组成的。众所周知，独享处理器资源时，串行程序的执行时间近似等于程序指令执行花费的 CPU 时间。但是，并行程序相对复杂，其执行时间（execution time）等于从并行程序开始执行，到所有进程执行完毕，墙上时钟走过的时间，也称之为墙上时间（wall time）。对各个进程，墙上时间可进一步分解为计算 CPU 时间、通信 CPU 时间、同步开销时间、同步导致的进程空闲时间。

计算 CPU 时间 进程指令执行所花费的 CPU 时间，它可以分解为两个部分，一个是程序本身指令执行占用的 CPU 时间，即通常所说的用户时间（user time），主要包含指令在 CPU 内部的执行时间和内存访问时间，另一个是为了维护程序的执行，操作系统花费的 CPU 时间，即通常所说的系统时间（system time），主要包含内存调度和管理开销、I/O 时间、以及维护程序执行所必需要的操作系统开销等。通常地，系统时间可以忽略。

通信 CPU 时间 包含进程通信花费的 CPU 时间。

同步开销时间 包含进程同步花费的时间。

进程空闲时间 当一个进程阻塞式等待其他进程的消息时，CPU 通常是空闲的，或者处于等待状态。进程空闲时间是指并行程序执行过程中，进程所有这些空闲时间的总和。

显然，进程的计算 CPU 时间小于并行程序的墙上时间，而并行程序的墙上时间才是用户真正关心的时间，是评价一个并行程序执行速度的时间。

以上讨论均假设并行程序在执行过程中，各个进程独享处理器资源。如果进程与其他并行程序的进程共享处理器资源，则该进程和其他进程只能分时共享处理器资源，因此，这样会人为地延长并行程序的墙上时间。在本章中，总是假设并行程序的各个进程是独享处理器资源的。

4.2 并行加速比与效率

在处理器资源独享的前提下，假设某个串行应用程序在某台并行机单处理器上的执行时间为 T_S，而该程序并行化后，P 个进程在 P 个处理器并行执行所需要的时间为 T_P，则该并行程序在该并行机上的加速比 S_P 可定义为：

$$S_P = \frac{T_S}{T_P} \tag{4.1}$$

效率定义为：

$$E_P = \frac{S_P}{P} \tag{4.2}$$

这里，需要说明的是，T_1 指处理器个数为 1 时，并行程序的执行时间。通常情形下，T_1 大于 T_S，因为并行程序往往引入一些冗余的控制和管理开销。

加速比和效率是衡量一个并行程序性能的最基本的评价方法。显然，执行最慢的进程将决定并行程序的性能。

在以上加速比和效率的定义中，有一个基本的假设，要求并行机的各个处理器是同构 (homogeneous) 的，即并行机各个处理器的结构完全一致（包含 CPU 类型、内存大小与性能、cache 特征等等），或者说，串行程序在各个处理器执行的墙上时间相等。

如果并行机的各个处理器功能不一致，称之为异构并行机。对此，以上加速比和效率的定义不是很合适。其中，两个突出的问题就是，串行程序的执行时间是选择最快的处理器运行，还是选择最慢的处理器运行？在效率定义中，处理器个数选择为 P 是否合适？一个比较好的方法就是，将所有处理器以最快的处理器为基准，进行归一化处理。

本章中，总是假设并行机是同构的。

4.3 并行程序性能评价方法

上节介绍的加速比和效率，只能反映并行程序的整体执行性能，但是，无法反映并行程序的性能瓶颈。性能评价的主要目的在于，揭示并行程序的性能瓶颈，指导并行程序的性能优化。因此，有必要进一步分解加速比和效率，提出更细致的性能评价方法。这里，引入文献 [4] 介绍的性能评价方法。

4.3.1 浮点峰值性能与实际浮点性能

在现代微处理器中，微处理器的浮点峰值性能等于 CPU 内部浮点乘加指令流水线的条数、每条流水线每个时钟周期完成的浮点运算次数、处理器主频三者的乘积。为了获得处理器的浮点峰值性能，所有浮点乘加指令流水线必须不间断运行。但是，由第 1 章中多级存储结构的讨论可知，任何一次 cache 访问失效均可能中断流

水线的执行，因此，通常情形下，程序是无法达到峰值性能的。一般的串行程序也只能发挥峰值性能的几个到十多个百分点。

并行程序的实际浮点性能等于并行程序的总的浮点运算次数和并行程序执行时间的比值。并行机的峰值性能等于处理器峰值性能和处理器个数的乘积。同样，并行程序的实际浮点性能是无法达到并行机峰值性能的。进一步定义，并行程序发挥并行机浮点峰值性能的比率为并行程序的实际浮点性能和并行机的峰值性能的比值。

实际浮点性能是衡量一个并行程序的绝对指标，而加速比和效率是相对于串行程序，衡量一个并行程序并行性的相对指标。显然，如果串行程序发挥的浮点峰值性能比率越高，它的执行时间就越短，获得的加速比就可能越低。

4.3.2 数值效率和并行效率

将并行程序的墙上时间分解为：

$$T_P = C_i + D_i \qquad i = 1, 2, \ldots, P \tag{4.3}$$

其中，C_i 为第 i 个进程花费的 CPU 时间，D_i 为第 i 个进程的空闲时间。进一步分解，有：

$$C_i = L_i + O_i \qquad i = 1, 2, \ldots, P \tag{4.4}$$

其中，L_i 为第 i 个进程数值计算指令执行花费的 CPU 时间，O_i 为第 i 个进程通信、同步花费的 CPU 时间。

基于以上时间分解，引入如下几个概念。

并行计算粒度 进程指令数值计算时间与墙上时间的比值，即：

$$\gamma_i = \frac{L_i}{T_P} \qquad i = 1, 2, \ldots, P \tag{4.5}$$

非数值冗余 由于并行引入的额外非数值计算开销

$$W_i = D_i + O_i \qquad i = 1, 2, \ldots, P \tag{4.6}$$

负载平衡 为了减少无谓的空闲时间，各个进程分配的 CPU 时间尽量相等，为此，定义负载平衡效率如下：

$$\eta_P = \frac{\sum_{i=0}^{P-1} C_i}{\max_{i=0}^{P-1} C_i \times P} \tag{4.7}$$

负载平衡对并行程序的性能影响很大。例如，假设并行程序总共需要计算 100 个时间单位，并取 $P = 4$。如果负载是平衡的，每个进程分配 25 个单位，则可获得加速比 4，负载平衡效率为 100%。假设负载不平衡，某个进程分配 50 个单位，2 个进程各自分配 20 个单位，而另外一个进程分配 10 个单位，则并行执行时间依赖于执行最慢的进程，因此，负载平衡效率仅为 50%，导致加速比仅为 2。

显然，为了缩短并行程序的墙上时间，应该极小化非数值冗余 W_i，极大化并行计算粒度，保证负载平衡。

基于以上时间分解公式，$C_T = \sum_i C_i$，$D_T = \sum_i D_i$，则 $C_T + D_T = P \times T_P$，效率公式可以写为：

$$E_P = \frac{S_P}{P} = \frac{T_S}{C_T} \times \frac{C_T}{C_T + D_T}$$

分别定义数值效率和并行效率如下：

$$\begin{cases} \mathrm{NE}_P = \dfrac{T_S}{C_T} & \text{数值效率} \\ \mathrm{PE}_P = \dfrac{C_T}{C_T + D_T} & \text{并行效率} \end{cases}$$

则

$$E_P = \mathrm{NE}_P \mathrm{PE}_P$$

数值效率反映了并行计算引入的额外 CPU 时间开销，这种开销来自两个方面。一方面，并行执行时，随着处理器个数的增长，各个进程的 cache 命中率将提高，有助于缩短总的计算 CPU 时间，从而提高数值效率；另一方面，并行计算可能引入额外的开销，例如并行算法增加计算量、并行通信 CPU 时间和同步开销等，它们将延长计算 CPU 时间，从而降低数值效率。如果缩短的 CPU 时间小于额外的开销，则数值效率将大于 1，否则，数值效率将小于 1。

并行效率反映了并行程序具体执行的并行性能，它依赖于并行机网络的通信性能，以及并行程序的负载平衡等方面，并行效率总是小于 1 的。

如果数值效率低，说明并行算法或者程序设计引入的额外 CPU 时间太大，有必要修改数值算法或者并行程序设计策略；如果并行效率低，说明并行计算网络通信同步导致的进程间空闲时间太长，有必要在负载平衡、通信结构等多个方面组织优化。

由于效率等于数值效率和并行效率的乘积，因此，如果数值效率大于 1，则可能效率将大于 1，也就是，并行程序的加速比将大于处理器的个数，此时，称之为超线性加速比。由以上的分析可知，需要辩证地看待超线性加速比。如果串行程序能够很好地发挥单处理器的峰值性能，则并行程序几乎不可能获得超线性加速比。反之，如果出现超线性加速比，说明串行程序需要进一步的性能优化。

4.4 可扩展分析

给定并行算法（程序）和并行机，如何调整参与并行计算的处理器个数 P 和求解问题的计算规模 W，使得随着处理器个数的增长，

并行计算的效率可以保持不变，称之为并行程序和并行机相结合的可扩展分析。

可扩展分析是并行计算一个重要研究课题，被广泛应用于描述并行算法（程序）能否有效利用可扩展的处理器个数的能力。通常地，它具有四个目的：

选择合理的算法与结构组合 确定求解某类问题的何种并行算法与何种并行机的组合，它可以有效地利用所期望的处理器规模。

性能预测 对于运行在某台并行机上的某种算法（程序），根据算法（程序）在小处理器规模上的运行性能，预测该算法（程序）移植到大处理器规模上后运行的性能。

最优性能选择 对某类算法，假设问题规模固定，确定在某类并行机上最优的处理器个数和可获得的最优的加速比。

指导性能优化 指导改进并行算法（程序），使得并行算法充分利用可扩展的处理器规模。

下面介绍两种常见的可扩展分析方法。

等效率度量（Kumar 1987[6]） 对于某类算法和并行机，如何保持问题规模 W 与处理器个数 P 之间的关系 $W(P)$，使得随着处理器个数 P 的增长，保持并行计算的效率不变。也就是求出等效率函数：

$$W = f_E(P) \qquad E \text{ 固定} \tag{4.8}$$

等效率值越小，则当处理器个数增多时为保持相同效率所需增加的问题规模就越小，因此就有更好的可扩展性。

等速度度量（Sun 1994[7]） 对于运行在并行机上的某个算法，当处理器个数增加时，需要增加多大的计算量，才能保持并行程序的平均速度不变。定义平均速度 $\bar{V} = \frac{V}{P} = \frac{W}{PT_P}$，$V$ 为并行程序的执行速度，问题规模从 (W, P) 变化到 (W', P')，则等速度可扩展度量公式可写为：

$$\Psi(P, P') = \frac{W/P}{W'/P'} = \frac{W}{W'} \times \frac{P'}{P} = \frac{T}{T'} \tag{4.9}$$

$0 < \Psi(P, P') < 1$，$\Psi(P, P')$ 越接近 1，说明可扩展性越好。

4.5 程序性能优化

一个程序的实际执行性能取决于程序的实现方式及所使用的高性能计算机的体系结构。本节介绍编写高性能计算程序时的一些注意事项以及一些常用的程序性能优化技巧。在编写程序时注意这些方面的问题，有助于得到合理的性能。

4.5.1 串行程序性能优化

串行程序性能的优化是并行程序性能优化的基础。一个好的并行程序首先应该拥有良好的单机性能。影响程序单机性能的主要因素是程序的计算流程和处理器的体系结构。在基于微处理器的高性能计算机上，提高程序单机性能的关键是改善程序的访存性能、提高 cache 命中率、以及充分挖掘 CPU 多运算部件、流水线的处理能力。

1. 调用高性能库

充分利用已有的高性能程序库是提高应用程序实际性能最有效的途径之一。许多著名的高性能数学程序库如优化的 BLAS、FFTW

(参看附录 A) 等，由于经过厂商或第三方针对特定处理机进行的专门优化，其性能一般大大优于用户自行编写的同样功能的程序段或子程序。合理地调用这些高性能库中的子程序，可以成倍、甚至成数量级地提升应用程序的性能，达到事半功倍的效果。

2. 选择适当的编译器优化选项

现代编译器在编译时能够对程序进行优化从而提高所生成的目标代码的性能。这些优化功能通常通过一组编译选项来控制。比较通用的优化选项有 "-O"、"-O0"、"-O2"、"-O3" 等，"-O0" 表示不做优化，"-O1"、"-O2"、"-O3" 等表示不同级别的优化，优化级别越高，生成的代码的性能可能会越好，但采用过高级别的优化会大大降低编译速度，并且可能导致错误的运行结果。通常，"-O2" 的优化被认为是安全的，它可以保证程序运行的正确性。对于一般程序的编译而言，使用优化选项 "-O2" 或 "-O3" 就可以了，进一步的优化可以参考所使用的编译器的手册，通过实验比较来找出一组理想的优化选项组合 (参看习题 4)。

3. 合理定义数组维数

现代计算机提高内存带宽的一个重要手段是采用多体交叉并行存储系统，即使用多个独立的内存体，对它们统一编址，将数据以字为单位采用循环交替方式均匀地分布在不同的内存体中。为了充分利用多体存储，在进行连续数据访问时应该使得地址的增量与内存体数的最大公约数尽量小，特别要避免地址增量正好是体数的倍数的情况，因为此时所有的访问将集中在一个存储体中。对于组关联的 cache 结构也有类似的问题，应该使被访问的数据均匀地分布在尽可能多的 cache 组中才能获得好的执行性能。由于内存体数和 cache 组数通常是 2 的幂，因此连续数据访问时应该避免地址增量正好是 2 的幂的情形。

很多情况下，合理地声明数组维数有助于避免或缓解内存体或 cache 组冲突的问题。以 Fortran 为例，假设内存访问的字长为 8 字节，内存体数为 S，当对二维数组 `REAL*8 A(M,N)` 第二维上的数据进行顺序访问时：

```
DO J = 1, N
   ... A(I,J) ...
ENDDO
```

数据的增量等于第一维的维数 `M`。当 `M` 是 S 的倍数时，所访问的数据集中在一个存储体中，此时的访存性能是最差的。而当 `M` 与 S 互素时，所访问的数据均匀分布在所有的存储体中，此时访存性能是最好的。对于前一种情况，即 `M` 恰好是 S 的倍数时，一个有效的优化方法是将数组声明成 `REAL*8 A(M+1,N)`，即给数组增加一条“边”，这样虽然浪费了少量存储空间，但是经常可以大幅度地提高程序的执行效率 (参看习题 5)。

4. 注意嵌套循环的顺序

提高 cache 使用效率的一个简单原则是尽量改善数据访问的局部性。数据访问的局部性分为空间局部性和时间局部性。空间局部性指访问了一个地址后，应该接着访问它的邻居，而时间局部性则指对同一地址的多次访问应该在尽可能相邻的时间内完成。在嵌套的多重循环中，循环顺序往往对循环中数据访问的局部性有很大的影响，例如下面的循环：

```
DO I = 1, N
DO J = 1, M
   A(I,J) = D
ENDDO
ENDDO
```

内层循环中数据访问是跳跃的，地址增量为 `N` 个数，因此当 `N` 较大时数据访问的空间局部性较差。如果交换上述内外层循环的顺序，将

“DO I” 做为内层循环，“DO J” 做为外层循环，则数据的访问是连续的，空间局部性好，因而程序的性能会大幅度提高。在编写嵌套的多重循环代码时，一个通用的原则是尽量使得最内层循环的数据访问连续进行，这一点不难作到，而且往往可以大幅度提高程序的性能，是编写高性能计算程序时首先要注意的问题。(参看习题 6)。

5. 数据分块

当处理大数组时，对数组、循环进行适当分块有助于同时改善访存的时间和空间局部性。下面是一个典型的例子 (引自 [1])：

```
DO I = 1, N
DO J = 1, N
   A(I) = A(I) + B(J)
ENDDO
ENDDO
```

如果对数组 B 进行分块，可以将循环改写成下面的形式：

代码 4.1: 利用分块技术改进数据访问的时间局部性。

```
DO J = 1, N, S
DO I = 1, N
   DO JJ = J, MIN(J+S-1, N)
      A(I) = A(I) + B(JJ)
   ENDDO
ENDDO
ENDDO
```

代码 4.1 中 S 为分块大小。当 $S \geqslant N$ 时，相当于原始循环；当 $S = 1$ 时相当于交换 I 和 J 的循环顺序。根据 cache 的大小选择适当的 S 值，使得 B(J:J+S-1) 能够被容纳在 cache 中，可以改善对数组 B 的访问的时间局部性。

数据分块是一项比较复杂的优化技术，好的分块方式与分块参数的确定需要对代码及 cache 结构进行非常细致的分析或通过大量

的实验才能得到，因此该项技术一般只在对一些关键代码段进行深层次优化时才使用 (参看习题 7 和习题 8)。

6. 循环展开

循环展开是另一个非常有效的程序优化技术。它除了能够改善数据访问的时间和空间局部性外，还由于增加了每步循环中的指令与运算的数目，亦有助于 CPU 多个运算部件的充分利用。

下面是一个一维循环的例子：

```
DO I = 1, N
   D = D + A(I)
ENDDO
```

将它进行 4 步循环展开的代码如下：

```
DO I = 1, MOD(N,4)
   D = D + A(I)
ENDDO
DO I = MOD(N,4)+1, N, 4
   D = D + A(I) + A(I+1) + A(I+2) + A(I+3)
ENDDO
```

上面的代码中第一个循环用于处理 N 除以 4 的余数，第二个循环是展开后的循环。

再给出一个二重循环的例子，它计算一个矩阵的转置与一个向量的乘积：

```
DO J = 1, M
   T = 0.0
   DO I = 1, N
      T = T + A(I,J) * X(I)
   ENDDO
   Y(J)=T
ENDDO
```

对 I 循环展开 3 步、J 循环展开 2 步，再对内层循环进行合并后的结果如下，这里为了简化循环展开后的代码，假设 N 是 3 的倍数、M 是 2 的倍数：

```
DO J = 1, M, 2
   T0 = 0.0
   T1 = 0.0
   DO I = 1, N, 3
      T0 = T0 + A(I,J)*X(I)+A(I+1,J)*X(I+1)+A(I+2,J)*X(I+2)
      T1 = T1 + A(I,J+1)*X(I)+A(I+1,J+1)*X(I+1)+A(I+2,J+1)*X(I+2)
   ENDDO
   Y(J) = T0
   Y(J+1) = T1
ENDDO
```

手工编写多重循环展开代码往往非常麻烦，并且只能对固定的循环展开步数进行，不便于寻找最优的循环展开步数。现代编译系统亦提供编译选项或编译指导语句，实现自动循环展开的功能，例如 GNU 编译器 (gcc，g77 等) 提供了选项 "`-funroll-loops`"，用于指定对代码中的循环进行展开，但它们一般局限于使用固定的循环展开步数，通常达不到最好的性能。对于一些复杂的情况，可以借助于专门设计的工具来对代码进行自动或半自动的循环展开处理，例如，文献 [65] 中研究了借助 m4 宏语言进行 Fortran 程序的循环展开，其中定义了一套 m4 宏命令，可将一些常见的循环写成可以任意指定循环展开步数的形式，参看习题 9。

7. 其他程序优化方法

前面主要介绍的优化技术主要是针对访存的优化。除此之外，还有许多其他的优化方法，如针对 CPU 的指令调度、分支预测等等的优化。另外，有许多通用的或由 CPU 厂商开发的、针对特定 CPU 的性能分析工具，如 Intel VTune [66] 等。

4.5.2 并行程序性能优化

并行程序的性能优化相对于串行程序而言更加复杂，其中最主要的是选择好的并行算法及通信模式。在并行算法确定之后，影响并行程序效率的主要因素是通信开销、由于数据相关性或负载不平衡引起的进程空闲等待、以及并行算法引入的冗余计算。在设计并行程序时，可以采用多种技术来减少或消除这些因素对并行效率的影响。本节对常用的一些并行程序优化技术进行简单介绍与讨论，主要给出一些原则性的考虑。

1. 减少通信量、提高通信粒度

在消息传递并行程序中，花费在通信上的时间是纯开销，因此如何减少通信时间是并行程序设计中首先要考虑的问题。减少通信时间的途径主要有三个：减少通信量、提高通信粒度和提高通信中的并发度 (即不同结点对间同时进行通信，要注意的是，这些手段都是相对于特定条件而言的，例如，在网络重负载的情况下，提高通信并行度并不能改善程序的性能)。

例如，在求解 PDE 的区域分解算法中，为了减少通信量，应该尽量将通信局限在相邻的子区域之间，避免整个数据场的全局通信。在划分子区域时，应该极小化各子区域内边界点的数目。对于规则区域而言，通常采用高维块划分比一维条划分子区域内边界点数更少。

提高通信粒度的有效方法是减少通信次数，即尽可能将可以一起传递的数据合并起来一次传递。在收发不同类型的数据时，定义适当的 MPI 数据类型来避免内存中的数据拷贝。

2. 全局通信尽量利用高效聚合通信算法

当组织多个进程之间的聚合通信时，使用高效的通信算法可以大大提高通信效率、降低通信开销。对于标准的聚合通信，如广播、

归约、数据散发与收集等，尽量调用 MPI 标准库中的函数，因为这些函数往往经过专门优化。但使用标准库函数的一个缺点是整个通信过程被封装起来，无法在通信的同时进行计算工作，此时，可以自行编制相应通信代码，将其与计算过程结合起来，以达到最佳的效果。

3. 挖掘算法的并行度，减少 CPU 空闲等待

一些具有数据相关性的计算过程会导致并行运行的部分进程空闲等待。在这种情况下，可以考虑改变算法来消除数据相关性。某些情况下数据相关性的消除是以增加计算量做为代价的，这方面的典型例子有用 Jacobi 迭代替换 Gauss–Seidel 或超松弛迭代、三对角线性方程组的并行解法等。当算法在某个空间方向具有相关性时，应该考虑充分挖掘其他空间方向以及时间上的并行度，在这类问题中流水线方法往往发挥着重要的作用，例如，参看第 9.5 节。

4. 负载平衡

负载不平衡是导致进程空闲等待的另外一个重要因素。在设计并行算法时应该充分考虑负载平衡问题，必要时使用动态负载平衡技术，即根据各进程计算完成的情况动态地分配或调整各进程的计算任务。动态调整负载时要考虑负载调整的开销及由于负载不平衡而引起的空闲等待对性能的影响，寻找最优负载调整方案。

5. 通信、计算的重叠

通过让通信与计算重叠进行，利用计算时间来屏蔽通信时间，是减少通信开销的非常有效的方法。实现通信与计算重叠的方法一般基于非阻塞通信，先发出非阻塞的消息接收或发送命令，然后处理与收发数据无关的计算任务，完成这些计算后再等待消息收发的完成。通信与计算的重叠能否实现，除了取决于算法和程序的实现方

式之外，还取决于并行机的通信网络及通信协议。第 339 页代码 8.3 提供了一个如何实现通信与计算重叠的实例。

6. 通过引入重复计算来减少通信

有时通过适当引入一些重复计算，可以减少通信量或通信次数。由于当前大部分并行机的计算速度远远快于通信速度，并且一些情况下，当一个进程计算时，其他进程往往处于空闲等待状态，因而适当引入重复计算可以提高程序的总体性能。

例如一些公共量的计算，可以由一个进程完成然后再发送给其他进程，也可以各进程分别独立计算。后一个做法在性能上通常要好于前一个做法。另外一个通过引入重复计算来提高通信粒度的例子参看第 8 章习题 7。

习 题

1. 假设你使用的是一台异构并行机，请问如何评价一个并行程序的加速比和效率比较好？它相对于传统的同构并行机上的评价准则有何优点？

2. 在一台并行机上，为了测试一个并行程序的加速比和效率，必须做哪些准备工作？为什么？

3. 影响并行程序执行时间的一些主要因素是什么？如何合理地评价一个并行程序的性能？

4. 下面列出的是一个计算矩阵乘积的 Fortran 程序。统计采用不同优化选项编译该程序生成的代码的运行时间，根据程序的计算量和运行时间计算出程序的实际浮点性能（以 Mflops 为单位）和效率 (实际性能/处理器峰值性能)，并将结果填写在下表中 (根据需要加行)。

   提示：用 “`time` 程序名” 可以得到程序的运行时间。

优化选项	运行时间 (秒)	性能 (Mflops)	效率 (%)

文件名: code/performance/AxB.f

```
      PARAMETER(N=1024)
      REAL*8 A(N,N), B(N,N), C(N,N)
*
      DO J = 1, N
         DO I = 1, N
            A(I,J) = 1.D0
            B(I,J) = 1.D0
            C(I,J) = 0.D0
         ENDDO
      ENDDO
*
      DO J = 1, N
      DO K = 1, N
      DO I = 1, N
         C(I,J) = C(I,J) + A(I,K) * B(K,J)
      ENDDO
      ENDDO
      ENDDO
*
      STOP
      END
```

5. 测试下面程序的运行时间，然后将其中的 "`REAL*8 A(N,N)`" 改为 "`REAL*8 A(N+1,N)`"，比较修改前后运行时间的差异。

文件名: code/performance/conflicts.f

```
      PARAMETER(N=2048)
```

```
2       REAL*8 A(N,N)
3       DO K = 1, 100
4          DO I = 1, N
5          DO J = 1, N
6             A(I,J) = 1.D0
7          ENDDO
8          ENDDO
9       ENDDO
10      STOP
11      END
```

6. 改变习题 4 的程序中三重循环 (11–19 行) 的顺序，统计不同循环顺序的运行时间、性能及效率，将结果填写在下表中。

循环顺序	运行时间 (秒)	性能 (Mflops)	效率 (%)
I,J,K			
I,K,J			
J,K,I			
J,I,K			
K,I,J			
K,J,I			

7. 编写代码 4.1 中关于循环分块的完整程序，选取一个适当的 N 值，测试不同分块大小下的程序运行时间，并画出曲线。

8. 网址 ftp://ftp.cc.ac.cn/pub/home/zlb/bxjsbook/code/performance/ 下的 A+Bt.f 是计算一个矩阵和另一个矩阵的转置相加的程序，block.f 是它的分块版本。试分析分块前后 cache 使用的差异，并测试不同分块大小 (NB) 对程序性能的影响。

9. 网址 `ftp://ftp.cc.ac.cn/pub/home/zlb/bxjsbook/code/performance/` 中有一个利用 m4 宏语言进行循环展开的例子。从该网址下载文件 `AxB.m4`、`defs.m4` 和 `m4post.c`，编译 `m4post.c` 得到可执行程序 `m4post`，然后便可用下述命令：

```
m4 -P defs.m4 AxB.m4 | ./m4post >tmp1.f
m4 -P -Dni=2 -Dnj=2 -Dnk=2 defs.m4 AxB.m4 | ./m4post >tmp2.f
... ...
```

得到对习题 4 中的矩阵乘法程序进行循环展开的代码，其中 `ni`、`nj` 和 `nk` 分别给出循环 `I`、`J` 和 `K` 的循环展开步数 (注意运行上述命令时必须使用 GNU m4，其他版本的 m4 程序不支持 "`-P`" 选项)。试比较习题 4 中的代码与循环展开后的代码，为了便于阅读，可以利用 Emacs 编辑器的 "`C-A-q`" 命令 (`Ctrl-Alt-q`) 或菜单项 "`Fortran->Ident subprogram`" 对代码进行缩进 (identation) 处理。测试不同循环展开步数的性能，没法找出达到最佳性能的循环展开步数，并将测试结果填写在下表中 (可以在表中加行)。

循环展开步数	运行时间 (秒)	性能 (Mflops)	效率 (%)
1 1 1			

第 2 部分

并行算法设计与实现实例

第 5 章　自适应数值积分

第 3 章代码 3.3 中 (216 页) 给出了一个用梯形公式积分计算 π 的程序实例。本章对这个例子进行扩展，采用梯形公式结合自适应局部区间加密，计算一个函数在给定区间上的定积分达到指定精度。本章所介绍的算法、程序虽然简单并且不一定有实用价值，但它反映了并行算法、并行编程模式中的一些重要概念，如负载平衡、主从模式等。

5.1　梯形积分公式

设 $f(x)$ 是定义在区间 $[a,b]$ 上的函数，计算 $f(x)$ 在区间 (a,b) 上的近似积分的梯形公式 (trapezoid rule) 为：

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\frac{f(a)+f(b)}{2} \tag{5.1}$$

梯形积分公式 (5.1) 的误差为：

$$-\frac{1}{12}(b-a)^3 f''(\xi) \tag{5.2}$$

其中 $\xi \in (a,b)$。为了提高计算精度，将积分区间 (a,b) 分成 n 个小区间：$a = x_0 < x_1 < \ldots < x_n = b$，在每个小区间上应用梯形公式计算积分的近似值，然后对这些小区间的近似值求和来得到整个区间 (a,b) 上的近似积分，该方法称为复化梯形公式 (composite trapezoid rule)。计算公式如下：

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{1}{2}\sum_{i=1}^{n}(x_i - x_{i-1})\big(f(x_{i-1}) + f(x_i)\big) \tag{5.3}$$

当每个小区间的长度相等时，复化梯形公式变为：

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{2}\left[f(a)+f(b)+2\sum_{i=1}^{n-1} f(x_i)\right] \tag{5.4}$$

其中 $h=(b-a)/n$ 为小区间的长度 (也称为积分步长)，$x_i=a+ih$。

用公式 (5.4) 计算积分的误差为：

$$-\frac{1}{12}h^2 f''(\xi) \tag{5.5}$$

其中 ξ 为区间 (a,b) 中的某个点。

5.2 局部二分自适应区间加密

从公式 (5.2) 可以看出，梯形公式的计算误差与函数的二阶导数，即函数梯度的变化速度 (曲线 $\big(x,f(x)\big)$ 的弧度) 有关。为了达到在给定精度的前提下减少计算函数次数的目的，可根据函数导数的变化情况在不同地方使用不同长度的积分步长。这里采用下面的局部二分自适应区间加密算法。

算法 5.1： 二分法计算近似积分。

(0) 取当前计算区间为 $(x_0,x_1)=(a,b)$

(1) 用梯形公式 (5.1) 计算函数在当前计算区间 (x_0,x_1) 上的积分的近似值。

(2) 判断近似值的误差，如果误差满足要求，则该值即为该区间上的计算结果，该区间上的计算过程结束。

(3) 否则将区间一分为二：令 $x_c=(x_0+x_1)/2$，分别对子区间 (x_0,x_c) 和 (x_c,x_1) 重复步骤 1–3，直到它们的计算误差分别满足要求，然后将子区间 (x_0,x_c) 和 (x_c,x_1) 上的计算结果之和做为区间 (x_0,x_1) 上的计算结果。

算法 5.1 是一个递归过程。为了判断当前区间上的计算结果是否满足精度要求，需要对误差进行估计。根据式 (5.2)，区间 (x_0, x_1) 上梯形公式的计算误差为：

$$-\frac{1}{12}(x_1 - x_0)^3 f''(\xi) \tag{5.6}$$

其中 ξ 为 (x_0, x_1) 中某点。实际计算时，可以用该区间的中点 $x_c = \frac{1}{2}(x_0 + x_1)$ 近似代替 ξ 来对误差进行估计，当区间长度较小时，它们的差别是很小的。然后再用二阶中心差商近似代替区间中点 x_c 处的二阶导数值：

$$f''(\xi) \approx f''(x_c) \approx 4\frac{f(x_0) - 2f(x_c) + f(x_1)}{(x_1 - x_0)^2} \tag{5.7}$$

将式 (5.7) 代入式 (5.6) 便得到下面的近似误差估计：

$$-\frac{1}{3}(x_1 - x_0)\big(f(x_0) - 2f(x_c) + f(x_1)\big) \tag{5.8}$$

假设希望整个区间 (a, b) 上的计算误差小于 ε，则只需让每个小区间上的误差小于 $h\varepsilon/(b-a)$ 即可，其中 h 代表小区间的长度。对区间 (x_0, x_1) 而言，$h = x_1 - x_0$，因此，可以通过下式来判断误差是否满足要求：

$$|f(x_0) - 2f(x_c) + f(x_1)| < 3\varepsilon/(b-a) \tag{5.9}$$

记 $h = x_1 - x_0$ 为当前计算区间 (x_0, x_1) 的长度，v_0 为梯形公式计算得到的积分近似值：

$$v_0 = \frac{1}{2}h\big(f(x_0) + f(x_1)\big) \tag{5.10}$$

注意到式 (5.9) 中需要计算区间中点的函数值，可以利用它来计算一个比 v_0 更为精确的近似值 v：

$$v = \frac{1}{4}h\big(f(x_0) + f(x_1) + 2f(x_c)\big) = \frac{1}{2}\big(v_0 + hf(x_c)\big) \tag{5.11}$$

v 的误差近似等于 v_0 的误差的 $\dfrac{1}{4}$，因此如果用 v 作为近似积分结果，则误差判据可以改成：

$$|f(x_0) - 2f(x_c) + f(x_1)| < 12\varepsilon/(b-a) \tag{5.12}$$

由式 (5.10)–(5.11) 易知：

$$f(x_0) - 2f(x_c) + f(x_1) = \frac{4}{h}(v_0 - v)$$

因此可以在程序中使用下面的误差判据：

$$|v_0 - v| < 3h\varepsilon/(b-a) \tag{5.13}$$

即当一个区间上计算的值与将区间分半后计算的值间的误差小于该区间的长度的 3 倍乘以 $\varepsilon/(b-a)$ 时，便认为计算达到了精度要求，并将 v 作为该区间上的计算结果。由此导出了下面的算法。

算法 5.2: 梯形公式结合自适应逐次区间分半方法，计算函数 $F(x)$ 在区间 (a,b) 上的定积分值。当计算区间上的梯形积分公式不满足精度要求时，该函数将计算区间分半，并对自己进行递归调用来分别计算两个子区间上的积分值 (递归过程在算法的第 12 行)。

1: **Function** `Integr` $(a, f_a, b, f_b, \varepsilon, \mathbb{F})$
2: **begin**
3: **if** $a = b$ **then**
4: **return** 0
5: **end if**
6: $x_c := (a+b)/2$; $h := b - a$; $v_0 := h * (f_a + f_b)/2$
7: **if** $x_c = a$ **or** $x_c = b$ **then**
8: **return** v_0
9: **end if**
10: $f_c := \mathbb{F}(x_c)$; $v := (v_0 + f_0 * h)/2$; $e := |v - v_0|$
11: **if** $e \geqslant 3 * h * \varepsilon$ **then**

```
12:       return Integr(a, f_a, x_c, f_c, ε, 𝔽) + Integr(x_c, f_c, b, f_b, ε, 𝔽)
13:     else
14:       return v
15:     end if
16: end
```

5.3 串行程序

代码 5.1 是根据算法 5.2 编制的 C 语言程序 (integration.c 为源程序，integration.h 为包含函数原型说明的头文件，供其他文件引用)。与算法 5.2 略有不同的是程序中增加了两个常量 MAX_DEPTH 和 MACHINE_PREC，它们用来控制递归深度，避免出现过深的嵌套递归 (MAX_DEPTH 的值设得非常大，所以实际上不起作用)。

代码 5.2 给出的是一个测试 integration.c 的主程序。外部函数 F 负责计算被积函数在指定点的值，在文件 function.c 中定义，这里被积函数取为

$$f(x) = \frac{4}{1+x^2}$$

积分区间取为 $(0,1)$，积分的准确值为 π。此外，function.c 中还包含一个统计时间的函数 gettime()，供测试程序统计运行时间用。由于计算量太小不便统计程序运行时间，函数 F 中增加了一个空循环来增加它的计算时间 (第 14 行)。

代码 5.1: 二分法递归计算积分的 C 函数。

文件名: code/integration/integration.h

```
1 /* $Id: integration.h,v 1.2 2005/06/20 03:28:52 zlb Exp $ */
2
3 double integration(double a, double fa, double b, double fb, double eps,
4                    double(*F)(double x));
```

文件名: code/integration/integration.c

```
/* $Id: integration.c,v 1.4 2006/05/03 06:49:25 zlb Exp $ */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "integration.h"

#define MAX_DEPTH       1024            /* 最大递归深度 */
#define MACHINE_PREC    1e-15           /* 机器精度 */

double
integration(double a, double fa, double b, double fb, double eps,
          double(*F)(double x))
/* 梯形公式递归计算积分值 */
{
    static int depth = 0;               /* 当前递归深度 */
    double fc, v0, v, h, xc;

    ++depth;
    v = 0.0;
    if (a != b) {
        xc = (a + b) * 0.5;
        h = b - a;
        v = v0 = h * (fa + fb) * 0.5;
        if (xc != a && xc != b) {
            double err;
            fc = (*F)(xc);
            v = (v0 + fc * h) * 0.5;
            err = fabs(v - v0);
            if (err >= 3.0 * h * eps && /*depth < MAX_DEPTH &&*/
                fabs(h) >= (1.0 + fabs(xc)) * MACHINE_PREC) {   /* 递归 */
                v = integration(a, fa, xc, fc, eps, F) +
                    integration(xc, fc, b, fb, eps, F);
            }
        }
```

```
35      }
36      --depth;
37      return v;
38  }
```

代码 5.2: 计算积分的串行程序。

文件名: code/integration/main.c

```
1   /* $Id: main.c,v 1.2 2005/06/20 03:28:52 zlb Exp $ */
2
3   /* 递归方式自适应数值积分串行主程序 */
4
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <math.h>
8   #include "function.h"
9   #include "integration.h"
10
11  int
12  main(int argc, char **argv)
13  {
14      double a = 0.0, b = 1.0;
15      double res, cpu0, cpu1, wall0, wall1;
16      if (argc != 2) {
17          fprintf(stderr, "Usage:   %s epsilon\n", argv[0]);
18          fprintf(stderr, "Example: %s 1e-4\n", argv[0]);
19          return 1;
20      }
21      gettime(&cpu0, &wall0);
22      res = integration(a, F(a), b, F(b), atof(argv[1]) / (b - a), F);
23      gettime(&cpu1, &wall1);
24      printf("result=%0.16lf, error=%0.4le, cputime=%0.4lf, wtime=%0.4lf\n",
25                  res, res - RESULT, cpu1 - cpu0, wall1 - wall0);
26      printf("%u function evaluations.\n", evaluation_count);
27      return 0;
```

```
28 }
```

代码 5.3: 计算被积函数值。

文件名: code/integration/function.c

```
1  /* $Id: function.c,v 1.2 2005/06/20 03:28:52 zlb Exp $ */
2
3  #include <stdio.h>
4  #include <sys/time.h>
5  #include <sys/resource.h>
6
7  int evaluation_count = 0;
8
9  /* 被积函数 */
10 double
11 F(double x)
12 {
13     size_t i;
14     for (i = 0; i < 5000000; i++);      /* 为增加计算时间引入的空循环 */
15     evaluation_count++;
16     return 4.0 / (1.0 + x * x);         /* f(x) = 4/(1+x^2) */
17 }
18
19 /* 积分精确值 (用于检验结果) */
20 double RESULT = 3.141592653589793;
21
22 void
23 gettime(double *cpu, double *wall)
24 /* 返回当前 CPU 和墙上时间 */
25 {
26     struct timeval tv;
27     struct rusage ru;
28     if (cpu != NULL) {
29         getrusage(RUSAGE_SELF, &ru);
30         *cpu = ru.ru_utime.tv_sec + (double)ru.ru_utime.tv_usec * 1e-6;
```

```
    }
    if (wall != NULL) {
        gettimeofday(&tv, NULL);
        *wall = tv.tv_sec + (double)tv.tv_usec * 1e-6;
    }
}
```

文件名: code/integration/function.h

```
/* $Id: function.h,v 1.2 2005/06/20 03:28:52 zlb Exp $ */

extern int evaluation_count;
extern double RESULT;
double F(double x);
void gettime(double *cpu, double *wall);
```

下面是程序编译、链接及运行示例。

```
$ gcc -O2 -Wall -o main main.c function.c integration.c
$ ./main
Usage:   ./main epsilon
Example: ./main 1e-4
$ ./main 4e-4
result=3.1415632348954290, error=-2.9419e-05, cputime=0.0000, wtime=0.0001
37 function evaluations.
```

图 5.1 画出了运行上例得到的最终积分区间。从图中可以看出在 $x = 0$ 附近积分区间要密集些。

5.4 基于简单区域分解的并行算法

为了用 P 个进程实现算法 5.2 的并行计算，最简单的方法是将积分区间等分成 P 个子区间，每个进程独立地调用 integration 函数计算一个子区间上的近似积分，然后对这些子区间上的近似积分

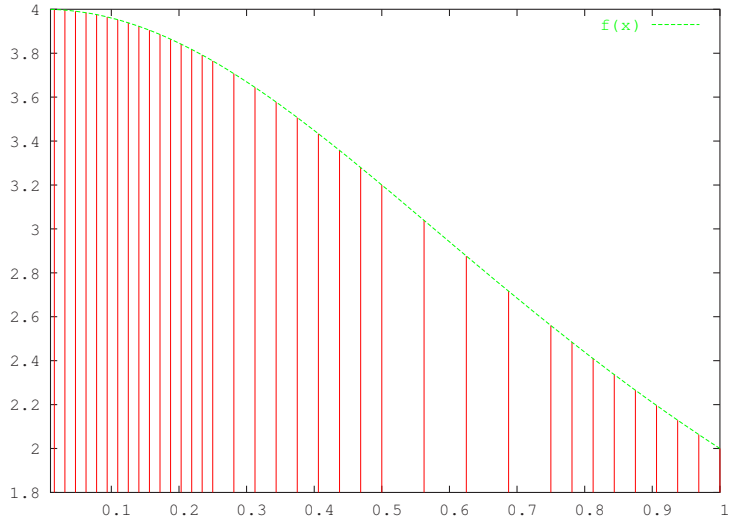


图 5.1 自适应梯形公式计算定积分 $\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$

值求和即得到计算结果。只需修改串行程序的主程序 (代码 5.2) 即可得到 MPI 并行程序。

代码 5.4: 采用均匀区间划分的 MPI 并行程序。

文件名: `code/integration/main-mpi1.c`

```
1 /* $Id: main-mpi1.c,v 1.2 2005/06/20 03:28:52 zlb Exp $ */
2
3 /* 递归方式自适应数值积分MPI主程序: 均匀区间划分 */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <math.h>
8
9 #include "mpi.h"
```

```
#include "integration.h"
#include "function.h"

int
main(int argc, char **argv)
{
    int nprocs, myrank;
    double a = 0.0, b = 1.0;
    double eps, res0, res, a0, b0, cpu0, cpu1, wall0, wall1, wall2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        if (argc != 2) {
            fprintf(stderr, "Usage:   %s epsilon\n", argv[0]);
            fprintf(stderr, "Example: %s 1e-4\n", argv[0]);
            MPI_Abort(MPI_COMM_WORLD, 1);
            return 1;
        }
        eps = atof(argv[1]);
    }

    /* 将 eps 广播给所有进程 */
    MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* 计算本进程的积分区间 */
    a0 = a + (myrank + 0) * (b - a) / nprocs;
    b0 = a + (myrank + 1) * (b - a) / nprocs;

    /* 开始时间 */
    gettime(&cpu0, &wall0);
    res0 = integration(a0, F(a0), b0, F(b0), eps / (b - a), F);
    gettime(&cpu1, &wall1);
    printf("\tRank=%d, # of evaluations=%u, cputime=%0.4lf, wtime=%0.4lf\n",
```

```
46              myrank, evaluation_count, cpu1 - cpu0, wall1 - wall0);
47      MPI_Reduce(&res0, &res, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
48      gettime(NULL, &wall2);
49      if (myrank == 0) {
50          printf("result=%0.16lf, error=%0.4le, wtime=%0.4lf\n",
51                  res, res - RESULT, wall2 - wall0);
52      }
53
54      MPI_Finalize();
55      return 0;
56  }
```

代码 5.4 在运行结束时打印出每个进程独立计算部分所花费的 CPU 时间和墙上时间，以及整个计算花费的墙上时间。下面是在由快速以太网联结的四台 PIII 550MHz 微机构成的微机机群上的编译、运行过程。

```
$ mpicc -O2 -o main-mpi1 main-mpi1.c function.c integration.c
$ mpirun -np 4 ./main-mpi1 4e-4
mpirun, v0.7, by ZLB
qsub: waiting for job 31555.s1 to start
qsub: job 31555.s1 ready
Starting program on nodes: c18 c17 c16 c15
        Rank=0, # of evaluations=17, cputime=0.3100, wtime=0.3094
result=3.1415632348954290, error=-2.9419e-05, wtime=0.3098
        Rank=1, # of evaluations=9, cputime=0.1600, wtime=0.1638
        Rank=2, # of evaluations=5, cputime=0.0900, wtime=0.0910
        Rank=3, # of evaluations=9, cputime=0.1600, wtime=0.1638
Cleanup...
qsub: job 31555.s1 completed
```

表 5.1 中汇总了一些运行结果，为便于比较，表中亦给出了串行程序的相应计算时间，以及并行加速比。从表 5.1 中可以看出每个进程计算的函数值数目及所花费的 CPU 时间不同，它反映出各个进程上的计算负载是不均衡的。

表 5.1 代码 5.4 在 4 结点快速以太网微机机群上的运行时间统计

要求误差	实际误差	并行墙上时间	串行墙上时间	加速比	进程 0		进程 1		进程 2		进程 3	
					点数	CPU	点数	CPU	点数	CPU	点数	CPU
4e-4	2.9e-5	0.31	0.68	2.19	17	0.31	9	0.16	5	0.09	9	0.16
4e-5	2.9e-6	0.71	1.84	2.59	39	0.71	29	0.52	15	0.27	21	0.38
4e-6	3.1e-7	2.35	6.06	2.58	129	2.35	97	1.76	45	0.82	65	1.18
4e-7	2.9e-8	9.33	21.2	2.27	513	9.33	271	4.93	125	2.28	257	4.68
4e-8	2.8e-9	18.6	51.7	2.78	1025	18.6	897	16.30	411	7.46	513	9.20

为了计算负载平衡效率 (227 页式 4.7)，需要各进程运行的 CPU 时间。由于实际程序中常常不容易单独统计进程的 CPU 时间，当各进程 (处理机) 的计算速度一样时，可以等效地用各进程所完成的工作量来计算负载平衡效率。以表 5.1 中最后一行数据为例，各进程的工作量可以用它们所计算的函数值个数来代表，4 个进程的工作量分别为 1025，897，411，513，因此程序的负载平衡效率为 $(1025 + 897 + 411 + 513)/(4 \times 1025) \approx 0.69$，而程序的实际并行效率也是 0.69，说明在该算例中，由于函数值计算所花费的时间远远大于通信所花费的时间，因此负载平衡是影响并行效率的关键因素。

5.5 基于主从模式的并行算法

本节介绍实现负载均衡的一个简单方法。该方法采用主从模式的程序结构，主进程负责调度，将被积函数值的计算分配给各个从进程，从进程完成指定函数值的计算并返回给主进程。积分求和由主进程负责。这种算法适用于被积函数值的计算计算量非常大的情况。

为了用主从模式实现自适应梯形公式积分，需要对计算过程进行重新组织，因为递归形式的计算流程不适合并行计算。新的算法如下：

算法 5.3: 梯形公式结合自适应逐次区间分半方法，计算函数 $f(x)$ 在区间 (a,b) 上的积分。本算法采用非递归的实现方式，将需要计算的区间存储在集合 $\mathbb{C}$，每步计算对 $\mathbb{C}$ 中的区间进行循环，将新产生的计算区间插入到 $\mathbb{C}$ 中，而将完成计算的区间从 $\mathbb{C}$ 中删除。

1: **Function** `Integr_nr` $(a, b, \varepsilon, \mathbb{F})$
2: **begin**
3: **if** $a = b$ **then**
4: **return** 0
5: **end if**
6: $\varepsilon := \varepsilon * 3/(b-a)$
7: $r := 0$
8: $\mathbb{C} := \{(a,b)\}$
9: **while** $\mathbb{C}$ 非空 **do**
10: **for all** (x_0, x_1) in $\mathbb{C}$ **do**
11: $h := x_1 - x_0$
12: $x_c := (x_0 + x_1)/2$
13: $v_0 := h * (\mathbb{F}(x_0) + \mathbb{F}(x_1))/2$
14: $v_1 := (v_0 + \mathbb{F}(x_c) * h)/2$
15: **if** $|v_1 - v_0| < h * \varepsilon$ **then**
16: $r := r + v_1$
17: 从 $\mathbb{C}$ 中删除区间 (x_0, x_1)
18: **else**
19: 从 $\mathbb{C}$ 中删除区间 (x_0, x_1)
20: 将区间 (x_0, x_c) 和 (x_c, x_1) 插入 $\mathbb{C}$
21: **end if**
22: **end for**
23: **end while**
24: **return** r
25: **end**

观察算法 5.3 不难看出，第 10–22 行的循环中所有函数值的计算可以并发进行，此外，已经计算过的函数值需要储存起来以避免重复计算。

函数 integration_nr 是算法 5.3 的 C 语言实现。该函数用一个双向链表 (数据结构 interval_t，定义在头文件 integration_nr.h 中) 来存放区间集合 $\mathbb{C}$ 和被积函数值。

代码 5.5: 二分法非递归方式计算积分的头文件和 C 函数。

文件名: code/integration/integration_nr.h

```
1  /* $Id: integration_nr.h,v 1.2 2005/06/20 03:28:52 zlb Exp $ */
2
3  typedef struct INTERVAL_T {
4      double x0, f0, x1, f1, fc;
5      struct INTERVAL_T *last, *next;
6  } interval_t;
7
8  double
9  integration_nr(double a, double b, double eps, void(*F1)(double x, double *f));
```

文件名: code/integration/integration_nr.c

```
1  /* $Id: integration_nr.c,v 1.3 2006/05/03 06:49:25 zlb Exp $ */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <math.h>
6  #include "integration_nr.h"
7
8  #define MACHINE_PREC    1e-15                   /* 机器精度 */
9
10 double
11 integration_nr(double a, double b, double eps, void(*F1)(double x, double *f))
12 {
```

```
    double res;
    interval_t *root, *I, *J;                   /* 区间集合链表指针 */
    int pass, inter_count = 1, inter_max = 0;

    if (a == b)
        return 0.0;
    eps *= 3.0 / (b - a);

    root = malloc(sizeof(*root));
    (*F1)(root->x0 = a, &root->f0);
    (*F1)(root->x1 = b, &root->f1);
    root->next = root->last = NULL;

    res = 0.0;
    while (inter_count > 0) {                   /* 主循环 */
        for (pass = 0; pass < 2; pass++) {
            I = root;
            if (pass) (*F1)(0.0, NULL);         /* 等待所有函数值的计算完成 */
            while (I != NULL) {
                double h, v0, v1, xc;
                xc = 0.5 * (I->x1 + I->x0);     /* 计算区间中点 */
                if (pass == 0) {
                    (*F1)(xc, &I->fc);          /* 请求计算函数值 */
                    I = I->next;
                    continue;
                }
                h = I->x1 - I->x0;
                v0 = 0.5 * h * (I->f0 + I->f1);
                v1 = (v0 + I->fc * h) * 0.5;
                if (fabs(v1 - v0) < h * eps ||  /* 误差满足要求或区间长度太小 */
                    fabs(h) < (1.0 + fabs(xc)) * MACHINE_PREC) {
                    res += v1;
                    if (I->last != NULL) I->last->next = I->next;
                    if (I->next != NULL) I->next->last = I->last;
                    J = I;
                    I = I->next;
```

```
49                  free(J);                    /* 删除计算已完成的区间 */
50                  if (J == root) root = NULL;
51                  inter_count--;
52                  continue;
53              }
54              J = malloc(sizeof(*J));         /* 申请一个新区间 */
55              if (++inter_count > inter_max) inter_max = inter_count;
56              J->x1 = I->x1;
57              J->f1 = I->f1;
58              I->x1 = J->x0 = xc;
59              I->f1 = J->f0 = I->fc;
60              J->next = I->next;
61              if (I->next != NULL) I->next->last = J;
62              I->next = J;
63              J->last = I;
64              I = J->next;
65          }
66      }
67  }
68
69  printf("Maxi. nunmber of intervals allocated: %d.\n", inter_max);
70  return res;
71  }
```

函数 integration_nr 的入口参数中，a，b 和 eps 给出积分区间和精度要求，F1 是计算被积函数在指定点上的值的外部函数。函数 F1 要求两个参数：x 和 f，它计算被积函数在 x 处的值，并将结果在放在由指针 f 指定的地址。为了后面并行计算的需要，F1 对被积函数值的计算可以延迟进行，即它可以在实际计算完成前就返回。如果 f == NULL，则强制 F1 完成所有尚未完成的计算，此时参数 x 被忽略。函数 integration_nr 中对每次所要处理的区间进行两遍扫描 (关于 pass 的循环)，第一遍扫描中通过函数 F1 递交所有要计算的函数值，第一遍扫描结束时调用 F1(0,NULL) 等待函数值的计算

完成，然后再进行第二遍扫描，这种控制流程使得每步中所有函数值的计算可以并发进行。函数 F1 的工作方式是实现并行化的关键，后面介绍的并行程序都是通过修改 F1 实现的。

串行主程序非常简单。由于不用并发计算被积函数在不同点上的值，函数 F1 采用了非延迟的方式，即每次立即完成函数值的计算然后返回。

代码 5.6: 二分法非递归方式计算积分的的串行程序。

文件名: code/integration/main_nr.c

```
1  /* $Id: main_nr.c,v 1.2 2005/06/20 03:28:52 zlb Exp $ */
2
3  /* 非递归方式自适应数值积分串行主程序 */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <math.h>
8  #include "integration_nr.h"
9  #include "function.h"
10
11 static void
12 F1(double x, double *f)
13 {
14     if (f == NULL) return;
15     *f = F(x);
16 }
17
18 int
19 main(int argc, char **argv)
20 {
21     double a = 0.0, b = 1.0;
22     double res, cpu0, cpu1, wall0, wall1;
23     if (argc != 2) {
24         fprintf(stderr, "Usage:   %s epsilon\n", argv[0]);
```

```
25          fprintf(stderr, "Example: %s 1e-4\n", argv[0]);
26          return 1;
27      }
28      gettime(&cpu0, &wall0);
29      res = integration_nr(a, b, atof(argv[1]), F1);
30      gettime(&cpu1, &wall1);
31      printf("result=%0.16lf, error=%0.4le, cputime=%0.4lf, wtime=%0.4lf\n",
32                  res, res - RESULT, cpu1 - cpu0, wall1 - wall0);
33      printf("%u function evaluations.\n", evaluation_count);
34      return 0;
35  }
```

下面是程序编译、运行示例。

```
$ gcc -O2 -o main_nr main_nr.c integration_nr.c function.c
$ ./main_nr 4e-4
Maxi. nunmber of intervals allocated: 16.
result=3.1415632348954277, error=-2.9419e-05, cputime=0.7000, wtime=0.7012
37 function evaluations.
```

该程序的运行结果除舍入误差外应该与递归程序的结果完全一样，它们间的差别是因为两种计算流程中积分值的求和顺序不同，因而舍入误差的积累不同。

5.5.1 基于非阻塞通信的并行程序

传统的主从编程模式中通常使用两个源程序，一个用于主进程，一个用于从进程。为了简化 MPI 程序的启动，这里对主从进程使用了同一个源程序，当进程号等于 0 时它执行主进程的代码，而当进程号不等于 0 时它执行从进程的代码。

本节介绍的程序中利用非阻塞通信来实现并行计算。函数 `F` 只在主进程中被调用。当参数 `f` 不等于 `NULL` 时它调用 `MPI_Isend` 将 `x` 值发送给从进程，再调用 `MPI_Irecv` 等待从进程的计算结果，然后立即返回以便继续处理下一个函数值的计算。当 `f` 等于 `NULL` 时，

它调用 MPI_Waitall 等待所有函数值计算的完成。

主进程利用消息标签 (message tag) 来通知从进程计算过程的结束。从进程每次从主进程接收一个数据并检查消息标签，如果消息标签不等于 0，则计算由该数据给出的点上的函数值并将结果发回给主进程，然后等待下一个数据，如果消息标签等于 0 则表明计算已经结束，从进程会退出循环。因此，主进程在计算完成后需要向所有从进程发送一个标签为 0 的消息通知它们退出。

代码 5.7: 基于非阻塞通信的二分法非递归方式计算积分的 MPI 并行程序。

文件名: code/integration/main_nr-mpi1.c

```
1  /* $Id: main_nr-mpi1.c,v 1.2 2005/06/20 03:28:52 zlb Exp $ */
2
3  /* 非递归方式自适应数值积分串行主程序: 非阻塞通信 */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <math.h>
8  #include <mpi.h>
9  #include "integration_nr.h"
10 #include "function.h"
11
12 static size_t count = 0;
13 static int nprocs, myrank;
14
15 static void
16 F1(double x, double *f)                /* 该函数只在 Master 中调用 */
17 {
18     static MPI_Request *req = NULL;
19     static MPI_Status *sta = NULL;
20     static double *xsave = NULL;
21     static int size = 0, n = 0, slave;
22
```

```
23      if (f == NULL) {
24          if (n) {
25              MPI_Waitall(n + n, req, sta);
26              n = 0;
27          }
28      }
29      else {
30          if (n >= size) {
31              req = realloc(req, 2 * (size += 128) * sizeof(*req));
32              sta = realloc(sta, 2 * size * sizeof(*sta));
33              xsave = realloc(xsave, size * sizeof(*xsave));
34          }
35          slave = count % (nprocs - 1) + 1;
36          xsave[n] = x;
37          MPI_Isend(xsave + n, 1, MPI_DOUBLE, slave, 1, MPI_COMM_WORLD, req+n+n);
38          MPI_Irecv(f, 1, MPI_DOUBLE, slave, 1, MPI_COMM_WORLD, req+n+n+1);
39          n++;
40          count++;
41      }
42  }
43
44  int
45  main(int argc, char **argv)
46  {
47      MPI_Init(&argc, &argv);
48      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
49      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
50      if (argc != 2) {
51          fprintf(stderr, "Usage:   %s epsilon\n", argv[0]);
52          fprintf(stderr, "Example: %s 1e-4\n", argv[0]);
53          MPI_Abort(MPI_COMM_WORLD, 1);
54      }
55      if (nprocs < 2) {
56          fprintf(stderr, "This program needs at least two processes.\n");
57          MPI_Abort(MPI_COMM_WORLD, 1);
58      }
```

```
    if (myrank == 0) {                  /* Master */
        double res, wall0, wall1;
        int i;
        gettime(NULL, &wall0);
        res = integration_nr(0.0, 1.0, atof(argv[1]), F1);
        gettime(NULL, &wall1);
        printf("result=%0.16lf, error=%0.4le, wtime=%0.4lf\n",
               res, res - RESULT, wall1 - wall0);
        printf("%u function evaluations.\n", count);

        /* Tell the slaves to quit */
        for (i = 1; i < nprocs; i++)
            MPI_Send(&res, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
    }
    else {                              /* Slave */
        while (1) {
            double d;
            MPI_Status sta;
            MPI_Recv(&d, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &sta);
            if (sta.MPI_TAG == 0) break;
            d = F(d);
            MPI_Send(&d, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
    return 0;
}
```

使用非阻塞通信的优点是程序实现简单，缺点是通信粒度小，当每个函数值的计算时间非常短时并行效率会非常差。此外，后台通信由 MPI 系统管理，在效率上与资源占用方面可能不如直接在用户程序中管理。必要时，特别是当需要同时处理大量积分点时，可以考虑在程序中增加对挂起的消息数目的限制，参看习题 7。

5.5.2 基于散发/收集通信的并行程序

本节介绍另外一个并行实现方式。主进程的函数 F 在接到一个函数值的计算请求时并不马上发送给从进程去计算，而是将需要计算函数值的 x 值保存起来，当 f 等于 NULL 时再将所有保存起来的 x 值平均分发给从进程去计算，然后再收集从进程的计算结果。

由于每次分配给各个从进程计算的函数值的数目事先是未知的，因此，主进程先调用 MPI_Scatter 告诉每个从进程需要它计算的函数值数目，再调用 MPI_Scatterv 函数分发需要从进程计算的 x 值，然后调用 MPI_Gatherv 收集计算结果。计算结束后，将 -1 作为函数值数目发送给从进程，从进程当接收到的函数值数目为负时则立即退出。

代码 5.8: 基于收集/散发的二分法非递归方式计算积分的 MPI 并行程序。

文件名: code/integration/main_nr-mpi2.c

```
1  /* $Id: main_nr-mpi2.c,v 1.2 2005/06/20 03:28:52 zlb Exp $ */
2
3  /* 非递归方式自适应数值积分串行主程序: Scatter/Gather方式 */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <math.h>
8  #include <mpi.h>
9  #include "integration_nr.h"
10 #include "function.h"
11
12 static size_t count = 0;
13 static int nprocs, myrank;
14 static int *cnts = NULL, *displs = NULL;
15
16 static void
```

```
17 F1(double x, double *f)                /* 该函数只在 Master 中调用 */
18 {
19     static double *xsave = NULL, **fsave = NULL;
20     static int size = 0, n = 0;
21
22     if (f == NULL) {
23         int i;
24         if (cnts == NULL) cnts = realloc(cnts, nprocs * sizeof(*cnts));
25         if (displs == NULL) displs = realloc(displs, nprocs * sizeof(*displs));
26         cnts[0] = 0;
27         displs[0] = 0;
28         for (i = 1; i < nprocs; i++) {
29             cnts[i] = n / (nprocs - 1) + (i <= n % (nprocs - 1) ? 1 : 0);
30             displs[i] = displs[i - 1] + cnts[i - 1];
31         }
32         MPI_Scatter(cnts, 1, MPI_INT, &i, 1, MPI_INT, 0, MPI_COMM_WORLD);
33         MPI_Scatterv(xsave, cnts, displs, MPI_DOUBLE, NULL, 0, MPI_DOUBLE, 0,
34                      MPI_COMM_WORLD);
35         MPI_Gatherv(NULL, 0, MPI_DOUBLE, xsave, cnts, displs, MPI_DOUBLE, 0,
36                      MPI_COMM_WORLD);
37         for (i = 0; i < n; i++) *(fsave[i]) = xsave[i];
38         count += n;
39         n = 0;
40     }
41     else {
42         if (n >= size) {
43             xsave = realloc(xsave, (size += 128) * sizeof(*xsave));
44             fsave = realloc(fsave, size * sizeof(*fsave));
45         }
46         xsave[n] = x;
47         fsave[n++] = f;
48     }
49 }
50
51 int
52 main(int argc, char **argv)
```

```
{
    int i, n;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (argc != 2) {
        fprintf(stderr, "Usage:   %s epsilon\n", argv[0]);
        fprintf(stderr, "Example: %s 1e-4\n", argv[0]);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    if (nprocs < 2) {
        fprintf(stderr, "This program needs at least two processes.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    if (myrank == 0) {                  /* Master */
        double res, wall0, wall1;
        gettime(NULL, &wall0);
        res = integration_nr(0.0, 1.0, atof(argv[1]), F1);
        gettime(NULL, &wall1);
        printf("result=%0.16lf, error=%0.4le, wtime=%0.4lf\n",
               res, res - RESULT, wall1 - wall0);
        printf("%u function evaluations.\n", count);

        /* Tell the slaves to quit */
        for (i = 0; i < nprocs; i++) cnts[i] = -1;
        MPI_Scatter(cnts, 1, MPI_INT, &i, 1, MPI_INT, 0, MPI_COMM_WORLD);
    }
    else {                              /* Slave */
        while (1) {
            static double *xsave = NULL;
            static int size = 0;
            MPI_Scatter(NULL, 0, MPI_INT, &n, 1, MPI_INT, 0, MPI_COMM_WORLD);
            if (n < 0) break;
            if (n > size) {
                if (xsave != NULL) free(xsave);
```

```
89                  xsave = malloc((size = n) * sizeof(*xsave));
90              }
91              MPI_Scatterv(NULL, 0, 0, MPI_DOUBLE, xsave, n, MPI_DOUBLE, 0,
92                      MPI_COMM_WORLD);
93              for (i = 0; i < n; i++) xsave[i] = F(xsave[i]);
94              MPI_Gatherv(xsave, n, MPI_DOUBLE, NULL, 0, 0, MPI_DOUBLE, 0,
95                          MPI_COMM_WORLD);
96          }
97      }
98      MPI_Finalize();
99      return 0;
100 }
```

采用收集/散发方式的优点是通信粒度较大、开销相对较小。如果不同 x 函数值的计算量不同，则需要对算法作进一步修改，如将计算任务分组，再将每组任务动态地分配给空闲的从进程，才能达到负载均衡的目的。

5.6　基于动态负载调度的并行算法

本节对基于自适应区间二分法的积分算法进行一些动态负载平衡技术方面的考虑。通常可以将这类问题表述为一个二叉树的叶子结点遍历问题，然后考虑考虑确定相应的并行方案来完成算法的实现等。受篇幅的限制，这里仅就可能的方案做一些讨论，而不给出并行程序的实例。

可以看到，在上面的自适应区间加密算法中，之所以任务分配不均衡，是因为计算任务划分是按照区间长度进行的。如果一开始就知道最终的积分区间分布，则可以直接按照区间的个数进行平均划分。但很多情况下，积分区间的最终分布事先是不知道的，而是通过自适应过程逐步确定的。自适应过程相当于在一棵二叉树上操作，树的叶子对应于当前计算区间，区间二分则相当于在树上添加新的

分枝。

并行程序运行的时候，每个进程会将自己搜索部分的所有非叶子结点存储在一个栈数据结构中。进程所负责的任务就是对自己维护的所有结点的后代进行遍历。

MPI 2.0 除了能够支持进程间的协同通信外，也支持单边通信方式。使用单边通信可以简化动态平衡技术的实现。否则的话，则需要使用比较迂回的方式来实现。

动态平衡和事先详细地进行任务分配的本质不同在于：不论分配的算法多么细致，程序实现多么巧妙，后者在程序开始运行以前，每个进程将要承担的工作都已经确定了，每次运行这个程序，相应的进程总是会做完全相同的工作。但是对于动态平衡来说，在程序运行之前，每个进程将会完成哪部分工作是完全不知道的，而且，每次运行程序，每个进程所做的工作通常是不一样的，具体哪个进程完成了哪些工作，依赖于程序运行时的具体环境。

动态负载平衡程序运行时，某个进程可能会暂时处于闲置状态，然后重新获得任务，从而需要考虑下面几个问题：

(1) 一个进程处于闲置状态时，它怎么申请和向谁申请新的任务？

(2) 一个进程收到申请任务的请求时，怎么将自己的任务分配出去？

(3) 整个程序怎么判断工作已经完成，从而退出？

从实现角度说，如果一个进程不能够接收其他进程随时传来的消息，那么就必须有一个机制来让所有的进程能够交换这样的信息。比如可以采用所谓公告板的模式：用一个进程来负责处理所有进程关于请求任务和分配任务的消息，所有进程轮流和这个进程交换信息，从而该进程成为所有申请任务的请求的集中地。当该进程和一个提交请求的进程通信的时候，将请求加入到的请求列表之中。而

当它和一个请求任务的进程进行通信的时候，则按一定的策略将一个请求分配给对方。

当一个进程处于闲置状态的时候，它向哪个进程申请任务是个策略问题，并没有确定的最好答案，因为到底最多的剩余任务位于那个进程是很难预知的，否则也不用采用动态平衡方法了。一般说来，有两个比较常用的策略：一个是轮流策略，另外一个是随机策略。顾名思义，所谓轮流策略就是指所有不处于闲置状态的进程被排在一个队列中，当一个请求任务的申请到达时，这个申请被发给队列的第一个进程，然后将队列的第一个进程和申请任务的进程都排到队列尾部。所有处于闲置状态的进程也排成一个队列，当一个进程完成现有的工作，从工作状态中转入闲置状态时，就将该进程从工作进程的队列中删除，放到闲置进程的队列中，并将它请求任务的申请排上日程。这样，所有进程在工作和空闲状态之间转换，轮流获取任务。所谓随机策略，是指将所有进程分成两个集合，一个集合中是所有处于闲置状态的进程，一个集合中是所有处于工作状态的进程。闲置状态集合中的进程会随机地挑选工作状态集合中的一个进程，然后向其申请任务，获得任务后便离开闲置状态集合，进入工作状态集合。当一个进程完成任务进入空闲状态，则离开工作状态集合进入闲置状态集合。对这两个策略不能简单地进行好与坏的评价，它们和所计算的具体问题、程序的实际运行环境有很大关系，包括硬件环境、软件环境以及程序运行时刻的具体状态。

如果一个进程收到其他进程发来的申请任务的请求，那么它应该把自己的哪部分任务分配出去呢？前面已经指出，这里的算法是一个树的叶子结点的遍历问题，进程在向后代搜索的时候，需要将祖先结点压栈，因此，每个进程事实上维护着一个结点的栈数据结构，进程需要处理的对象是这个栈中的结点的所有后代结点，将任务分配出去就是将栈中的一部分结点转移给申请任务的进程。通常采用的分配策略有三种：第一种是将栈底的第一个结点分配出去；第二

种是将栈的底下一半结点分配出去；第三种是将整个栈中除最上面的结点以外的其他结点全部分配出去。第二种策略显然是第一和第三种策略的折中方案。对这三种分配方式也很难预测孰优孰劣，完全依赖于具体的问题和运行环境。

关于如何判断整个计算过程何时结束，这里介绍一个通过任务权重进行判别的方法。实现步骤如下：将整个任务的总体工作量权重设为 1，在计算开始时，将所有任务分配给 0 号进程，其他进程全部设置为空闲。当一个进程向另一个进程申请任务时，分配任务的进程将自己所拥有的任务权重分一半给接收任务的进程。一个进程完成自己拥有的任务后，将自己的权重返还给 0 号进程。0 号进程负责收集所有已完成的任务的权重，当它增加到 1 时，意味着所有任务已经完成，整个程序可以结束了。需要注意的是，虽然从原理上将整个任务的权重设为 1，但为了避免浮点计算误差的影响，实际实现中通常采用长整数操作或者具有比较长字节的复合整数操作来计算权重，例如用一个 16 字节长的整数来进行权重计算，一般就能够满足需要了。

需要采用动态平衡方法来进行计算的问题，常具有非常大的任务分配不均匀性，导致计算过程中反复出现任务分配的问题。在进行任务分配的时候，有时候还伴随着大的数据传递。对于这些问题，可以具体问题具体对待，采取一些特殊的处理。例如，在结束任务的进程进入重新申请任务的状态前加入适当的延迟，以避免在任务接近总体完成时出现所有进程抢任务的现象。还可以在工作过程中根据已经完成的计算情况对分配策略进行动态调整，尽量减少任务重新分配的次数，使得动态平衡过程更加智能化。另外，可以留下一些供计算过程中进行人工干预的接口，使得在比较关键的时候通过人工干预来改变程序运行方式，提高程序的性能。

另外一类处理动态负载平衡的方法是所谓的动态重分。其基本思想是程序开始时首先对计算任务做一个初始划分，各进程分别负

责一部分工作。计算过程中，周期性地根据各进程任务进展情况以及对剩余工作量的某种估计重新对任务进行划分。与上面介绍的方法相比，该方法实现上更简单，更适合 MPI–1 的通信机制，但算法的最终效率取决于许多因素，包括任务重新划分的频度、任务划分的算法与开销、程序对剩余工作量预估的能力等等。任务重新划分的频度需要在负载平衡与重划分的开销之间进行折中，重划分算法的设计则需要考虑减少数据的迁移、极小化通信开销，而对剩余工作量的正确预估则有助于有效减少任务重分的次数及需要迁移的数据量。一个比较常见的做法是根据程序的负载平衡效率 (参考 227 页式 4.7) 来确定是否需要进行一次任务重新划分，当负载平衡效率低于某个阈值时，便进行一次任务重新划分。就本章讨论的例子而言，负载平衡效率的计算可以基于各进程所负责计算的区间上当前误差的某个函数，例如误差绝对值的某个方次之和。

习　题

1. 算法 5.2 中为什么要将函数值 f_a 和 f_b 放在 `Integr` 的参数表中?

2. 算法 5.2 中为什么需要第 7 行上的判断?

3. 将算法 5.2 中的梯形积分公式改为 Gauss 积分公式，比较它们的优缺点。

4. 说明为了防止过深的递归嵌套，代码 5.1 第 30 行上的判断是必要的。

5. 试对函数 `Integr` 进行修改，采用 Romberg 方法对近似值 v_0 和 v 进行一次外插 (等价于 Simpson 公式)，通过数值实验比较计算结果的精度。

6. 根据式 (5.6)–(5.9)，当指定计算精度要求时，积分区间长度的平方应与 $|f''(x)|$ 成反比，这里 $f(x) = 4/(1+x^2)$。试根据这个性质设计一种区间划分策略，使得各个进程的计算量尽量接近，然后修改代码 5.4 并通过数值实验检验改进效果。比较改进前后程序的并行效率。

7. 修改代码 5.7 对后台通信的消息数目加以限制，保证程序运行过程中最多只有 `M` 个尚未完成的非阻塞通信在同时进行，其中 `M` 是一个指定的整型常数。

8. 为了提高通信粒度，可以将需要计算的函数值用分组的方式发送给从进程。试修改代码 5.7，每次将 `N` 个 `x` 值发送给一个从进程，其中 `N` 是预先指定的常量 (正整数)。说明 `N` 值的大小对并行效率的影响。

9. 代码 5.8 的通信中，需要分别调用 `MPI_Scatter`、`MPI_Scatterv` 和 `MPI_Gatherv`。试设计一个函数接口，能够一次完成代码 5.8 中的 `MPI_Scatter` 和 `MPI_Scatterv` 通信，该函数实现时应该借助什么样的 MPI 通信函数？

第 6 章　矩阵并行计算

在科学与工程计算的许多问题中经常需要进行矩阵计算。矩阵相乘、求解线性方程组和矩阵特征值问题是矩阵计算最基本的内核。随着 MPP 并行计算机、机群以及消息传递并行环境（MPI 等）的不断完善，针对分布式并行计算机的并行计算方法的研究变得越来越重要。本章着重介绍矩阵乘积、求解线性方程组和矩阵特征值问题的并行算法，相关的向量化算法 [50, 51] 不在这里介绍。为叙述方便，本章假设算法针对的是一台有 p 个处理机的并行系统，每个处理机上运行一个进程，P_j 表示第 j 个处理机或进程，P_{myid} 表示当前的处理机或进程，`send(`x`,`j`)`和 `recv(`x`,`j`)` 分别表示在 P_{myid} 中把 x 传送到 P_j 和从 P_j 中接收 x。此外，用 $i \mod p$ 表示 i 对 p 取模运算。

程序设计与机器实现是密不可分的，计算效率与编程技术有很大的关系，尤其是在并行计算机环境下，研制高质量的程序对发挥计算机的性能起着至关重要的作用。本章将结合并行算法研究给出并行机上的一些重要例子来阐述程序设计思想，对并行算法采用程序方式描述，以利于并行实现。

关于通信模式、数据传输方式等有许多不同的假设，这些假设对于在理论上讨论并行算法的加速比及效率是非常重要的。对于消息传递型并行系统，通常考虑的通信模式为：$T_c = \alpha + \beta \times N$，其中 α 是启动时间，β 是传输单位数据所需的时间，N 是数据的传输量。

在矩阵并行计算中，一个非常重要的问题是矩阵在处理机中的存放方式。通常采用的是矩阵在处理机阵列中按卷帘方式存放。假

设分块矩阵是 8×8，处理机阵列是 3×2，则矩阵的存放方式如下：

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \\ A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} \tag{6.1}$$

$$\left(\begin{array}{cccc|cccc} A_{00} & A_{02} & A_{04} & A_{06} & A_{01} & A_{03} & A_{05} & A_{07} \\ A_{30} & A_{32} & A_{34} & A_{36} & A_{31} & A_{33} & A_{35} & A_{37} \\ A_{60} & A_{62} & A_{64} & A_{66} & A_{61} & A_{63} & A_{65} & A_{67} \\ \hline A_{10} & A_{12} & A_{14} & A_{16} & A_{11} & A_{13} & A_{15} & A_{17} \\ A_{40} & A_{42} & A_{44} & A_{46} & A_{41} & A_{43} & A_{45} & A_{47} \\ A_{70} & A_{72} & A_{74} & A_{76} & A_{71} & A_{73} & A_{75} & A_{77} \\ \hline A_{20} & A_{22} & A_{24} & A_{26} & A_{21} & A_{23} & A_{25} & A_{27} \\ A_{50} & A_{52} & A_{54} & A_{56} & A_{51} & A_{53} & A_{55} & A_{57} \end{array}\right) \tag{6.2}$$

对于一般 $m \times n$ 分块矩阵和一般的处理机阵列 $p \times q$，小块 A_{ij} 存放在处理机 P_{kl} $(k = i \mod p,\ l = j \mod q)$ 中。

从数值代数的角度出发，矩阵计算问题可以粗略地划分为 4 大类：

- 线性代数方程组 $Ax = b$
- 线性最小二乘问题 $\min_{x \in R^n} \|Ax - b\|_2$，$b \in R^m$
- 矩阵特征值问题 $Ax = \lambda x$，$Ax = \lambda Bx$
- 矩阵奇异值分解 $A = U\Sigma V^{\mathrm{T}}$

6.1 并行矩阵乘法

矩阵乘积在实际应用中经常用到，许多先进的计算机上都配有高效的串行程序库。为了在并行计算环境上实现矩阵乘积，研究并行算法是非常必要的。本节考虑的计算问题是

$$C = A \times B \tag{6.3}$$

其中 A 和 B 分别是 $m \times k$ 和 $k \times n$ 矩阵，C 是 $m \times n$ 矩阵。不失一般性，假设 $m = \bar{m} \times p$，$k = \bar{k} \times p$ 和 $n = \bar{n} \times p$，下面考虑基于矩阵 A 和 B 在处理机中的不同存储方式的并行计算方法。

6.1.1 串行矩阵乘法

例 6.1: 串行矩阵乘积子程序 (i-j-k 形式)

```
do i=1, m
  do j=1, L
    do k=1, n
      c(i, j) = c(i, j) + a(i, k) * b(k, j)
    enddo
  enddo
enddo
```

例 6.2: 串行矩阵乘积子程序 (j-k-i 形式)

```
do j=1, L
  do k=1, n
    do i=1, m
      c(i, j) = c(i, j) + a(i, k) * b(k, j)
    enddo
  enddo
enddo
```

6.1.2 行列划分算法

这里将矩阵 A 和 B 分别划分为如下的行块子矩阵和列块子矩阵：

$$A = \begin{bmatrix} A_0^{\mathrm{T}} & A_1^{\mathrm{T}} & \cdots & A_{p-1}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}}, \quad B = \begin{bmatrix} B_0 & B_1 & \cdots & B_{p-1} \end{bmatrix} \tag{6.4}$$

这时 $C = (C_{i,j}) = (A_i \times B_j)$，其中 $C_{i,j}$ 是 $\bar{m} \times \bar{n}$ 矩阵。A_i、B_i 和 $C_{i,j}$，$j = 0, \ldots, p-1$ 存放在 P_i 中，这种存放方式使数据在处理机中不重复。由于使用 p 个处理机，每次每个处理机计算出一个 $C_{i,j}$，计算 C 需要 p 次来完成。$C_{i,j}$ 的计算是按对角线进行的，计算方法如下：

算法 6.1:

mp1 $\equiv$ myid+1 mod p, mm1 $\equiv$ myid$-$1 mod p
for $i = 0$ to $p-1$ do
  $l \equiv i$+myid mod p
  $C_l = A \times B$
  if $i \neq p-1$, send(B, mm1), recv(B, mp1)
end{for}

在这个算法中，$C_l = C_{\mathrm{myid},l}$，$A = A_{\mathrm{myid}}$，$B$ 在处理机中每次循环向前移动一个处理机，每次交换数据为 $k \times \bar{n}$ 矩阵，交换次数为 $p-1$ 次。如果用 DTA_1 表示算法 6.1 中数据的交换量，CA_1 表示算法 6.1 中的计算量，则有 $\mathrm{DTA}_1 = 2 \times k \times (n - \bar{n})$，$\mathrm{CA}_1 = m \times k \times n/p$。

6.1.3 行行划分算法

这里将矩阵 A 和 B 均划分为行块子矩阵，矩阵 A 的划分同式 (6.4)，B 的划分如下：

$$B = \begin{bmatrix} B_0^{\mathrm{T}} & B_1^{\mathrm{T}} & \cdots & B_{p-1}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}} \tag{6.5}$$

C_i 为和 A_i 相对应的 C 的第 i 个块，进一步把 A_i 按列分块与 B 的行分块相对应，记

$$A_i = \left[\begin{array}{cccc} A_{i0} & A_{i1} & \cdots & A_{i,p-1} \end{array} \right]$$

从而有

$$C_i = A_i \times B = \sum_{j=0}^{p-1} A_{i,j} \times B_j$$

初始数据 A，B 和 C 的存放方式与 6.1.2 相同，在结点 P_{myid} 上的计算过程可归纳为算法 6.2。

算法 6.2:

mp1 ≡ myid+1 mod p, mm1 ≡ myid-1 mod p
for $i = 0$ to $p - 1$ do
  $l \equiv i$+myid mod p
  $C = C + A_l \times B$
  if $i \neq p - 1$, send(B, mm1), recv(B, mp1)
end{for}

这个算法中的数据交换量和计算量与算法 6.1 相同，所不同的只是计算 C 的方式，其中 $A_l = A_{myid,l}$。

6.1.4 列列划分算法

这里将矩阵 A 和 B 均划分为列块子矩阵，B 的划分与式 (6.4) 相同，A 划分为如下形式：

$$A = \left[\begin{array}{cccc} A_0 & A_1 & \cdots & A_{p-1} \end{array} \right] \tag{6.6}$$

这时 C 的划分与 B 的划分相对应，也是按列分块的，进一步把 B_i 按行分成与 A 的列分块相对应的行分块，记

$$B_i = \left[\begin{array}{cccc} B_{i0}^{\mathrm{T}} & B_{i1}^{\mathrm{T}} & \cdots & B_{i,p-1}^{\mathrm{T}} \end{array} \right]^{\mathrm{T}}$$

从而有下面计算 C 的方法。

$$C_j = A \times B_j = \sum_{i=0}^{p-1} A_i \times B_{i,j}$$

这时的计算过程是传送矩阵 A 而不是 B，具体的算法描述如下：

算法 6.3:

mp1 $\equiv$ myid+1 mod p, mm1 $\equiv$ myid-1 mod p
for $i = 0$ to $p - 1$ do
  $l \equiv i$+myid mod p
  $C = C + A \times B_l$
  if $i \neq p - 1$, send(A, mm1), recv(A, mp1)
end{for}

算法 6.3 的计算量与算法 6.1 和算法 6.2 是相同的，算法 6.3 的数据交换量是 $\mathrm{DTA}_3 = 2 \times m \times (k - \bar{k})$。当 $m \neq n$ 时，$\mathrm{DTA}_1 \neq \mathrm{DTA}_3$。两种算法数据交换量的大小是由 m 和 n 决定的，即当 $m \leqslant n$ 时，$\mathrm{DTA}_3 \leqslant \mathrm{DTA}_1$。由于它们的计算量是相同的，因此只要按通信量大小选择算法就可以得到好的并行效率。

6.1.5 列行划分算法

这里将矩阵 A 和 B 分别划分为列和行块子矩阵，A 的划分与式 (6.6) 相同，B 的划分与式 (6.5) 相同。由此得到

$$C = A \times B = \sum_{i=0}^{p-1} A_i \times B_i$$

C 的计算是通过 p 个规模和 C 相同的矩阵之和得到的。从对问题的划分可以看出，并行算法的关键是计算矩阵的和，设计有效地计算矩阵和的算法，对发挥分布式并行系统的效率起着重要作

用。假设结果矩阵 C 也是按列分块存放在处理机中的，记 $B_i = \begin{bmatrix} B_{i0} & B_{i1} & \cdots & B_{i,p-1} \end{bmatrix}$ 则有

$$C_j = \sum_{i=0}^{p-1} A_i \times B_{ij}$$

因此，可以给出如下的算法：

算法 6.4:

$C = A \times B_{\text{myid}}$
for $i = 1$ to $p-1$ do
  $l \equiv i+\text{myid} \bmod p$, $k \equiv p-i+\text{myid} \bmod p$
  $T = A \times B_l$
  send(T, l), recv(T, k)
  $C = C + T$
end{for}

这里给出的算法简洁易懂，其通信量 $\text{DTA}_4 = 2 \times m \times (n - \bar{n})$。如果采用按行分块方式计算 C，算法 6.4 也同样适合，且通信量也是不变的，因此选择何种方式计算 C 可根据需要而定。

6.1.6 Cannon 算法

假设矩阵 A，B 和 C 可以分成 $m \times m$ 块矩阵，也即，$A = (A_{ij})_{m\times m}$，$B = (B_{ij})_{m\times m}$，和 $C = (C_{ij})_{m\times m}$，其中 A_{ij}，B_{ij} 和 C_{ij} 是 $n \times n$ 矩阵，进一步假定有 $p = m \times m$ 个处理机。为了讨论 Cannon 算法，引入块置换矩阵 $Q = (Q_{ij})$ 使得

$$Q_{ij} = \begin{cases} I_n, & j \equiv i+1 \bmod m \\ 0_n, & \text{其他情况} \end{cases}$$

其中 I_n 和 0_n 分别是 n 阶单位矩阵和零矩阵。定义对角块矩阵 $D_A^{(l)} = \text{diag}(D_i^{(l)}) = \text{diag}(A_{i,i+l \bmod m})$，容易推导出 $A = \sum_{l=0}^{m-1} D_A^{(l)} \times Q^l$。因

此

$$C = A \times B = \sum_{l=0}^{m-1} D_A^{(l)} \times Q^l \times B = \sum_{l=0}^{m-1} D_A^{(l)} \times B^{(l)}$$

其中 $B^{(l)} = Q^l \times B = Q \times B^{(l-1)}$。利用这个递推关系式，并把处理机结点编号从一维映射到二维，即有 $P_{\text{myid}} = P_{\text{myrow,mycol}}$，数据 A_{ij}，B_{ij} 和 C_{ij} 存放在 P_{ij} 中，容易得到下面的在处理机 P_{myid} 结点上的算法。

算法 6.5:

$C = 0$
mpc1 ≡ mycol+1 mod m; mmc1 ≡ mycol-1 mod m;
mpr1 ≡ myrow+1 mod m; mmr1 ≡ myrow-1 mod m;
for $i = 0$ to $m - 1$ do
  k ≡ myrow+i mod m;
  $r \equiv k - i \bmod m$;
  if mycol=k & myrow=r then
    send(A, (myrow, mpc1)); copy(A, tmpA);
  else if myrow=r
    recv(tmpA, (myrow, mmc1));
    if k ≠ mpc1, send(tmpA, (myrow, mpc1));
  end{if}
  $C = C$+tmpA $\times B$;
  if $i \neq m - 1$ then
    send(B, (mmr1, mycol)); recv(B, (mpr1, mycol));
  end{if}
end{for}

该算法具有很好的负载平衡，其特点是在同一行中广播 A，计算出 C 的部分值之后，在同列中滚动 B。数据交换量 $\text{DTA}_5 = m \times$

$2\times n^2+(m-1)\times 2\times n^2=2(2m-1)n^2=4m^2n^2/\sqrt{p}-2m^2n^2/p$。由于计算量对每个处理机来说是相同的，因此在选择算法时只需考虑通信量。从所给出的这五个并行计算矩阵乘积的算法可以看到，对于方阵的乘积，当 $p\geqslant 4$ 时，Cannon 算法具有优越性。

6.2 线性代数方程组并行求解方法

这里考虑的问题是

$$Ax=b \tag{6.7}$$

其中 A 是系数矩阵，b 是右端项。并行求解方程组 (6.7) 的过程可以分为两部分，一是并行计算矩阵 A 的 LU 分解，其中 L、U 分别是下三角和上三角矩阵，也即存在一排列矩阵 Q，使 $QA=LU$；二是并行求解三角形方程组，即求解方程组 $Ly=b$ 和 $Ux=y$。下面给出有关算法的描述。

6.2.1 分布式系统的并行 LU 分解算法

首先考虑 $n\times n$ 矩阵 $A=(a_{ij})$ 的串行LU 分解法，根据求解线性方程组的需要，采用部分选主元的 Gauss 消去法进行列消元，使得 L 是单位下三角矩阵。在算法中 A_k 表示 A 的第 k 行。

算法 6.6:

for $j=0$ to $n-2$ do
  find l: $|a_{lj}|=\max\{|a_{ij}|,i=j,\ldots,n-1\}$
  if $l\neq j$, swap A_j and A_l
  if $a_{jj}=0$, A is singular and return
  $a_{ij}=a_{ij}/a_{jj}$, $i=j+1,\ldots,n-1$
  for $k=j+1$ to $n-1$ do
    $a_{ik}=a_{ik}-a_{ij}\times a_{jk}$, $i=j+1,\ldots,n-1$

end{for}

end{for}

在算法 6.6 中，主要计算工作量是修正矩阵 A，即做 $a_{ik} = a_{ik} - a_{ij} \times a_{jk}$。因此，并行计算的主要任务就是在多处理机上同时对矩阵 A 的不同部分做修正。在并行计算机上，为了高效率计算 LU 分解，一个重要工作是使负载尽可能的平衡。为描述简便，在各处理机上矩阵 A 采用一维卷帘 (wrap) 存储方式，即把矩阵 A 的第 i 列存放在 $P_{i \bmod p}$ 中。假设 $n = p \times m$，在下面算法中 A 的第 i 列为原来 A 的第 $i \times p + \text{myid}$ 列，矩阵在处理机中的存放方式为：

$$\left[\begin{array}{ccc|ccc|ccc|c} A_{00} & A_{0p} & \cdots & A_{01} & A_{0,p+1} & \cdots & A_{02} & A_{0,p+2} & \cdots & \cdots \\ A_{10} & A_{1p} & \cdots & A_{11} & A_{1,p+1} & \cdots & A_{12} & A_{1,p+2} & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{n-1,0} & A_{n-1,p} & \cdots & A_{n-1,1} & A_{n-1,p+1} & \cdots & A_{n-1,2} & A_{n-1,p+2} & \cdots & \cdots \end{array}\right] \tag{6.8}$$

下面给出在 P_{myid} 上的算法描述。

算法 6.7:

icol= 0

for $j = 0$ to $n - 2$ do

if myid=$j \bmod p$ then

find l: $|a_{l,\text{icol}}| = \max\{|a_{i,\text{icol}}|, i = j, \ldots, n-1\}$

if $l \neq j$, swap $a_{j,\text{icol}}$ and $a_{l,\text{icol}}$

if $a_{j,\text{icol}} = 0$, A is singular and kill all processes

$a_{i,\text{icol}} = a_{i,\text{icol}}/a_{j,\text{icol}}$, $i = j+1, \ldots, n-1$

$f_{i-j-1} = a_{i,\text{icol}}$, $i = j+1, \ldots, n-1$

send(l, myid+1) and send(f, myid+1)

icol+1 $\rightarrow$ icol

else

recv(l, myid-1) and recv(f, myid+1)
if myid+1 $\neq j$ mod pi then
send(l, myid+1)
send(f, myid+1)
end{if}
end{if}
if $l \neq j$, swap A_j and A_l
for k=icol to $m-1$ do
$a_{ik} = a_{ik} - f_i \times a_{jk}$, $i = j+1, \ldots, n-1$
end{for}
end{for}

算法 6.7 是在分布式并行计算机上做 LU 分解的一种常用方法，如果采用分块二维卷帘方式存储，增大算法的粒度，对大规模处理机系统能够使负载更加平衡，具有非常好的效果。目前 TOP500 的 Linpack 测试程序，就是采用分块二维卷帘方式存储。

6.2.2 三角方程组的并行解法

本节考虑三角方程组的并行计算方法，不失一般性，仅讨论并行求解下三角方程组 $Lx = b$。三角方程组的有效并行求解是并行求解线性方程组不可缺少的，它的并行效率对求解整个问题有直接的影响，这里主要给出分布式并行计算环境下的并行实现方法。首先给出一个串行算法。

算法 6.8:

for $i = 0$ to $n-1$ do
$x_i = b_i / l_{ii}$
for $j = i+1$ to $n-1$ do
$b_j = b_j - l_{ji} \times x_i$

end{for}
end{for}

在这个算法中每次对 b 进行修正时用到 L 的一列，如果按这种方式并行修正 b，则称之为列扫描方法。对于列扫描算法，原始数据 L 适合于按行存放，当修正 b 的值时，就可以并行计算。同时为使每个处理机的工作量尽可能均衡，要采取卷帘方式存放数据。为了实现并行计算，需要将每步计算出来的解的一个分量传送到所有其他处理机中，其通信次数是很多的，而且每次并行计算的工作量较小，这对于消息传递型并行计算机系统是不太适合的，但是对于有共享存储的系统是可以采用这种计算方案的。

下面介绍一种在分布式并行机上的下三角方程组的求解方法，参见文献 [54]。该方法采用按列卷帘方式存放数据，每次传递的是部分修正的右端项，而不是新求出的解，通过叠加的方式计算下次的新解。这个算法在分布式系统上被广泛应用，是非常有效的并行算法。图 6.1 是使用 3 个处理机求解下三角线性代数方程组的并行计算过程示意。其算法的具体描述形式如下：

算法 6.9:

$k = 0$
if myid=0, then
  $u_i = b_i$, $i = 0, \ldots, n-1$, $v_i = 0$, $i = 0, \ldots, p-2$
else
  $u_i = 0$, $i = 0, \ldots, n-1$
for $i = myid$ step p to $n-1$ do
  if $i > 0$, recv(v, $i - 1 \bmod p$)
  $x_k = (u_i + v_0)/l_{ik}$
  $v_j = v_{j+1} + u_{i+1+j} - l_{i+1+j,k} \times x_k$, $j = 0, \ldots, p-3$
  $v_{p-2} = u_{i+p-1} - l_{i+p-1,k} \times x_k$

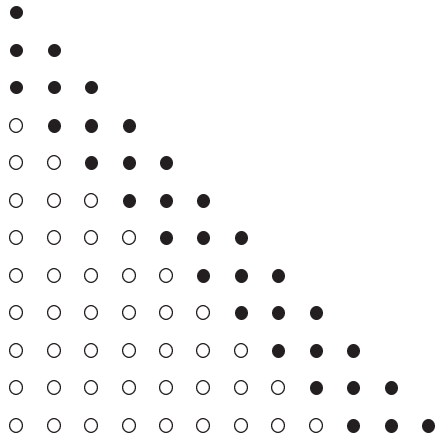


图 6.1 使用 3 个处理机求解下三角线性代数方程组

send(v, $i+1 \bmod p$)

$u_j = u_j - l_{jk} \times x_k$, $j = i+p, \ldots, n-1$

$k+1 \to k$

end{for}

这个算法在支持非阻塞通信的并行计算机系统上，只在开始时有等待，之后便可以并行计算。它很好地利用了运算的可结合性，展现了并行算法的魅力。

6.3 对称正定线性方程组的并行解法

对于对称正定矩阵 A 的分解，通常采用 Cholesky 分解，也即 $A = R^{\mathrm{T}}R$，其中 R 是上三角矩阵。关于三角方程组的并行求解已经在前一节中给出，这里仅考虑对称正定矩阵的并行 Cholesky 分解。一个方法是在传统的 Cholesky 分解列格式算法的基础上，对于发送

数据不需等待的并行系统，吸收求解三角方程组的方法思想，给出不需要等待数据交换的并行算法；另一个方法是用双曲旋转变换的方式来做 Cholesky 分解，这种分解具有很好的并行性。

6.3.1 Cholesky 分解列格式的并行计算

这里给出的并行 Cholesky 分解算法，是在传统的 Cholesky 分解列格式算法的基础上，结合分布式并行计算机系统的特点给出的 [55]。串行 Cholesky 分解算法如下：

算法 6.10:

for $j = 0$ to $n-1$ do

$a_{jj} = a_{jj} - \sum_{k=0}^{j-1} a_{jk} \times a_{jk}$, $a_{jj} = \sqrt{a_{jj}}$

for $i = j+1$ to $n-1$

$a_{ij} = (a_{ij} - \sum_{k=0}^{j-1} a_{jk} \times a_{ik})/a_{jj}$

end$\{for\}$

end$\{for\}$

在这个算法中，矩阵 R^{T} 存放在矩阵 A 的下三角位置，它对于 j 循环来说，每次计算出 R^{T} 的一列，故称之为列格式算法。由于第 j 列的计算用到前面的 $j-1$ 列的值，因此在并行计算 R 时就要把它之前的列的信息传送到该列所在的结点上。在该串行算法的基础上，引入分解因子变量 F，它记录当前处理机之前的 $p-1$ 个处理机上的分解因子，是 $(p-1) \times n$ 矩阵，原始矩阵 A 按行卷帘方式存放在处理机中，则在结点 P_{myid} 上的算法如下：

算法 6.11:

for $i = 0$ to $m-1$ do

$k = i \times p + \mathrm{myid}$, $l = k - p + 1$

if $k > 0$ then, recieve G from $P_{\mathrm{myid}-1}$

for $j=0$ to $p-2$ do

$a_{i,j+l}=(a_{i,j+l}-\sum_{t=0}^{j+l-1}a_{it}\times g_{jt})/g_{j,j+l}$

$F_j=G_{j+1}$

end$\{for\}$

$a_{ik}=a_{ik}-\sum_{t=0}^{k-1}a_{it}\times a_{it}$

$a_{ik}=\sqrt{a_{ik}}$, $F_{p-2}=A_i$

Send F to $P_{\text{myid}+1}$

for $e=i+1$ to $m-1$ do

for $j=0$ to $p-2$ do

$a_{e,j+l}=(a_{e,j+l}-\sum_{t=0}^{j+l-1}a_{et}\times g_{jt})/g_{j,j+l}$

$a_{ek}=(a_{ek}-\sum_{t=0}^{k-1}a_{et}\times a_{it})/a_{ik}$

end$\{for\}$

end$\{for\}$

该并行算法的特点是每步计算出 R^{T} 的 p 列，在同一循环中，各处理机计算出的 R^{T} 的 p 列是不相同的，从而实现计算与通信的异步进行，减少处理机的等待。这只是一种并行计算 Cholesky 分解的方法，对于现在的并行计算机结构来说，使用这样的方法不一定会取得很好的效果，主要是在使用高速缓存方面存在缺陷。因此，在求解 Cholesky 分解时，采用类似 LU 分解的方法更为有效。

6.3.2 双曲变换 Cholesky 分解

双曲变换 Cholesky 分解是指做 Cholesky 分解时使用如下的双曲变换：

$$H=\begin{bmatrix}\cosh\phi & \sinh\phi\\ \sinh\phi & \cosh\phi\end{bmatrix}$$

在研究 Cholesky 分解的算法中，使用下面的变换形式：

$$H = (1-\rho^2)^{-\frac{1}{2}} \begin{bmatrix} 1 & -\rho \\ -\rho & 1 \end{bmatrix}$$

其中 $\rho = \tanh(-\phi)$。以下讨论通过这个变换，把矩阵 A 化成 $R^{\mathrm{T}}R$。

假设 $A = D + U^{\mathrm{T}} + U$，其中 D 是对角矩阵，U 是严格上三角矩阵，记 $W = D^{-1/2}U$，和 $V = D^{1/2} + W$。通过简单的推导有

$$A = V^{\mathrm{T}}V - W^{\mathrm{T}}W$$

从而有下面的关系式成立：

$$\begin{bmatrix} R^{\mathrm{T}} & 0 \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} V^{\mathrm{T}} & W^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} V \\ W \end{bmatrix}$$

其中 I 是 $n \times n$ 单位矩阵，为使用双曲变换实现对称正定矩阵的 Cholesky 分解，要用到下面的一些定义与引理。

定义 6.1：如果一个 $2m \times 2m$ 矩阵 Θ 满足下面的关系式：

$$\Theta^{\mathrm{T}} \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \Theta = \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix}$$

其中 I 是 $m \times m$ 单位矩阵，则称之为是伪正交的 (pseudo-orthogonal) 矩阵。

从这个定义可以看到，如果存在一个伪正交矩阵 Q 使得

$$Q \begin{bmatrix} V \\ W \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

那么，从前面的关系式中就可得出 $A = R^{\mathrm{T}}R$。因此计算双曲变换 Cholesky 分解，其主要任务就是寻找伪正交矩阵 Q。为此，下面不加证明地给出一些重要引理，参见文献 [56]。

引理 6.1: 如果 R 和 S 都是 $n\times n$ 上三角矩阵，使得 $R^{\mathrm{T}}R-S^{\mathrm{T}}S$ 对称正定，则 R 是可逆的，并满足：

$$\mid s_{kk}r_{kk}^{-1}\mid<1,\qquad 1\leqslant k\leqslant n$$

引理 6.2: 如果 R 和 S 都是 $n\times n$ 上三角矩阵，使得 $R^{\mathrm{T}}R-S^{\mathrm{T}}S$ 对称正定，并令 $\rho_k\equiv s_{kk}r_{kk}^{-1}$，$1\leqslant k\leqslant n$，$\hat{Q}=\tilde{Q}^{(n)}\tilde{Q}^{(n-1)}\cdots\tilde{Q}^{(1)}$，其中 $\tilde{Q}^{(k)}$ 的元素定义如下：

$$\tilde{q}_{ij}^{(k)}=\begin{cases}1, & i=j\neq k \text{ 或 } i=j\neq n+k\\ (1-\rho_k^2)^{-\frac{1}{2}}, & i=j=k \text{ 或 } i=j=n+k\\ -(1-\rho_k^2)^{-\frac{1}{2}}\rho_k, & (i,j)=(k,n+k) \text{ 或 } (i,j)=(n+k,k)\\ 0, & \text{其他}\end{cases}$$

此外，如果 $\begin{bmatrix}\tilde{R}\\ \tilde{S}\end{bmatrix}=\hat{Q}\begin{bmatrix}R\\ S\end{bmatrix}$，则 $\hat{Q}$ 是伪正交矩阵，并且 $\tilde{R}$ 是上三角矩阵，$\tilde{S}$ 是严格上三角矩阵。

从 $\tilde{Q}^{(k)}$ 的定义可以看到，它是作用在 R 和 S 的第 k 行的一个变换，如果

$$H_k=(1-\rho_k^2)^{-\frac{1}{2}}\begin{bmatrix}1 & -\rho_k\\ -\rho_k & 1\end{bmatrix}$$

则 $H_k\begin{bmatrix}R_k\\ S_k\end{bmatrix}$ 与 $\tilde{Q}^{(k)}\begin{bmatrix}R\\ S\end{bmatrix}$ 的第 k 行和第 $k+n$ 行是相同的。

假设 Q 是 $n\times n$ 循环置换矩阵，其中 $p_{1,n}=1$ 和 $p_{i,i-1}=1$, $2\leqslant i\leqslant n$。根据引理 6.2，双曲变换 Cholesky 分解的算法可描述成如下：

算法 6.12:

$V^0=V$, $W^0=W$, $A=V^{\mathrm{T}}V-W^{\mathrm{T}}W$

for $i=0$ to $n-1$ do

$$\begin{bmatrix} V^{i+1} \\ W^{i+1} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & Q \end{bmatrix} \hat{Q}^{(i)} \begin{bmatrix} V^{i} \\ W^{i} \end{bmatrix}$$

end { for }

算法中 $\hat{Q}^{(i)}$ 与引理 6.2 中 $\hat{Q}$ 的定义相同，这时的 $\hat{Q}^{(i)}$ 是把 W_i 的对角线上的元素消为 0。如果矩阵 A 的逆是半带宽为 β 的带状矩阵，则算法 6.12 中的循环变量 i 只需到 β 就可计算出 A 的 Cholesky 分解，从而减少计算时间，详细的讨论请参见文献 [56]。由于每个的计算 H_k 是相互独立的，因此易于并行计算，这里就不再详述其并行算法。

6.3.3 修正的双曲变换 Cholesky 分解

在算法 6.12 中 $\hat{Q}^{(i)}$ 是 $\tilde{Q}^{(k)}$ 的乘积，而每个 $\tilde{Q}^{(k)}$ 只影响 V^i 和 W^i 的第 k 行，实际上就是双曲变换作用到一个 $2 \times l$ 矩阵上，假设 M 是 $2 \times l$ 矩阵，H 是双曲变换，目的是计算 $\overline{M} = HM$，使得 $\bar{m}_{21} = 0$ 的一系列计算过程中减少计算量。

为使 $\bar{m}_{21} = 0$，H 是容易计算的，即可以选择 $\rho = m_{21}m_{11}^{-1}$，但是这需要开方运算和 $6 \times l$ 次算术运算，为达到不开方和减少算术运算的目的，假设 $M = KB$，其中 $K = \mathrm{diag}(K_1, K_2)$ 是 2×2 正对角矩阵，即 $K_1 > 0$ 和 $K_2 > 0$。令 $G = \bar{K}^{-1}HK$，其中 $\bar{K}$ 是 2×2 对角矩阵。如果 $\bar{B} = GB$，则 $\bar{M} = \bar{K}\bar{B}$。通过适当选取 $\bar{K}$，可以达到减少运算次数和开方运算的目的。因此，在这里将并不直接计算 $\bar{K}$，而是用它的平方形式。假设 $L = K^2$，$\bar{L} = \bar{K}^2$，则 $\bar{L}$ 的计算由下面的引理给出。

引理 6.3: 假设 $\alpha = \dfrac{L_2}{L_1}$, $\beta = \dfrac{b_{21}}{b_{11}}$。如果选取 $\bar{L} = (1 - \alpha\beta^2)^{-1}L$, 则

$$G = \begin{bmatrix} 1 & -\alpha\beta \\ -\beta & 1 \end{bmatrix}$$

证明. 从 H 的定义可知 $\rho = m_{21}m_{11}^{-1} = \dfrac{K_2 b_{21}}{K_1 b_{11}}$。因此有

$$\begin{aligned} HK &= (1-\rho^2)^{-\frac{1}{2}} \begin{bmatrix} K_1 & -\dfrac{K_2^2 b_{21}}{K_1 b_{11}} \\ -\dfrac{K_2 b_{21}}{b_{11}} & K_2 \end{bmatrix} \\ &= (1-\alpha\beta^2)^{-\frac{1}{2}} K \begin{bmatrix} 1 & -\alpha\beta \\ -\beta & 1 \end{bmatrix} \end{aligned}$$

□

引理 6.4: 如果 R 和 S 都是 $n \times n$ 上三角矩阵，并且 E 和 F 是对角矩阵，使得 $R^{\mathrm{T}}ER - S^{\mathrm{T}}FS$ 是正定的，并假设 $\alpha_k = \dfrac{F_k}{E_k}$，$\beta_k = \dfrac{s_{kk}}{r_{kk}}$，如果

$$\begin{bmatrix} \tilde{R} \\ \tilde{S} \end{bmatrix} = \hat{Q} \begin{bmatrix} R \\ S \end{bmatrix}$$

其中 $\hat{Q} = \tilde{Q}^{(n)}\tilde{Q}^{(n-1)}\cdots\tilde{Q}^{(1)}$，$\tilde{Q}^{(k)}$，$E$ 和 F 的元素是如下定义的：

$$\tilde{q}_{ij}^{(k)} = \begin{cases} 1, & i = j \\ -\alpha_k\beta_k, & i = k, j = n+k \\ -\beta_k, & i = n+k, j = k \\ 0, & \text{其他} \end{cases}$$

和

$$\tilde{E}_k = \frac{E_k}{1-\alpha_k\beta_k^2}, \qquad \tilde{F}_k = \frac{F_k}{1-\alpha_k\beta_k^2}, \qquad 1 \leqslant k \leqslant n$$

则 $\tilde{R}^{\mathrm{T}}\tilde{E}\tilde{R} - \tilde{S}^{\mathrm{T}}\tilde{F}\tilde{S} = R^{\mathrm{T}}ER - S^{\mathrm{T}}FS$，并且 $\tilde{R}$ 是上三角矩阵，$\tilde{S}$ 是严格上三角矩阵。

这个引理的证明是容易的，故此略去。假设

$$A = V^{\mathrm{T}}EV - W^{\mathrm{T}}FW$$

则修正的双曲变换 Cholesky 分解算法可描述成如下形式：

算法 6.13:

$V^0 = V$, $W^0 = W$, $E^0 = E$, $F^0 = F$

for $i = 0$ to $n-1$ do

$$\begin{bmatrix} V^{i+1} \\ W^{i+1} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix} \hat{Q}^{(i)} \begin{bmatrix} V^i \\ W^i \end{bmatrix}$$

$$E_k^{i+1} = \frac{E_k^i}{1-\alpha_k^i {\beta_k^i}^2}, \qquad \widetilde{F}_k^{i+1} = \frac{F_k^i}{1-\alpha_k^i {\beta_k^i}^2}, \qquad 1 \leqslant k \leqslant n$$

$$F_1^{i+1} = \widetilde{F}_n^{i+1}, \quad F_{k+1}^{i+1} = \widetilde{F}_k^{i+1}, \qquad 1 \leqslant k \leqslant n-1$$

end { for }

这里 $\hat{Q}^{(i)}$ 的定义与引理 6.4 中 $\hat{Q}$ 的定义相同，与算法 6.12 一样都是易于并行实现的。

6.4 三对角方程组的并行解法

解三对角线性方程组在偏微分方程数值解中非常重要，因此已经有了很多关于它的并行算法，这方面的工作可参见文献 [57, 58]。这里着重基于区域分解的分裂方法，以加强对并行计算的了解，掌握并行算法的特点。这里求解的问题是 $Ax = d$，其中

$$A = T(c,a,b) = \begin{bmatrix} a_0 & b_0 & & & & \\ c_1 & a_1 & b_1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & c_{n-2} & a_{n-2} & b_{n-2} & \\ & & & c_{n-1} & a_{n-1} & \end{bmatrix} \tag{6.9}$$

对于一般性的三对角线性代数方程组并行求解方法的研究已经有许多工作，这里重点考虑对称正定三对角线性方程组的解法，它是基

于对矩阵分块的一种算法，故称之为分裂法 [60, 58]。其分解形式为如下的块三对角矩阵：

$$A=\begin{bmatrix} A_0 & B_0 & & & \\ B_0^{\mathrm{T}} & A_1 & B_1 & & \\ & \ddots & \ddots & \ddots & \\ & & B_{p-3}^{\mathrm{T}} & A_{p-2} & B_{p-2} \\ & & & B_{p-2}^{\mathrm{T}} & A_{p-1} \end{bmatrix}$$

其中 B_i 是 $m=n/p$ 阶、只有左下角的一个元素不为零的矩阵，A_i 是 m 阶对称正定三对角矩阵。因此可以对 A_i 做 LDL^{T} 分解，这里 D 是对角矩阵，L 是单位下三角矩阵。假设 $A_i=L_iD_iL_i^{\mathrm{T}}$，令 $L=\mathrm{diag}(L_i)$，则有

$$L^{-1}AL^{-T}=\begin{bmatrix} D_0 & \bar{B}_0 & & & \\ \bar{B}_0^{\mathrm{T}} & D_1 & \bar{B}_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \bar{B}_{p-3}^{\mathrm{T}} & D_{p-2} & \bar{B}_{p-2} \\ & & & \bar{B}_{p-2}^{\mathrm{T}} & D_{p-1} \end{bmatrix}$$

其中 $\bar{B}_i=L_i^{-1}B_iL_{i+1}^{-T}=B_iL_{i+1}^{-T}$，它是除最后一行外均为 0 的矩阵。由于 D_i 是对角矩阵，可以分别把 $\bar{B}_i$ 的最后一行除最后一个元素外均消为 0，记消去后的矩阵为 $\widetilde{D}_i$ 和 $\widetilde{B}_i$，从而 $\widetilde{D}_i$ 和 $\widetilde{B}_i$ 以及 $\widetilde{B}_i^{\mathrm{T}}$ 的最后一个元素构成一个新的小的三对角线性方程组。对于这个小的线性方程组可在一台机器上求解，然后把解传送到所有的处理机中，就可求出原问题的解。在这个方法中，首先要用到的是 LDL^{T} 分解，这个分解可由类似于上一节中的 $R^{\mathrm{T}}R$ 分解的方法得到。这里只需用串行的分解方法，故不再具体列出 LDL^{T} 的串行分解方法。记 B_i 的最后一行和 D_i 的对角元分别为 b_{ij} 和 d_{ij}，$i=0,\ldots,m-1$，则分裂算法如下：

算法 6.14:

(1) 计算 L_i，D_i，使 $A_i = L_i D_i L_i^{\mathrm{T}}$；
(2) 对矩阵做变换 $B_i = B_i L_{i+1}^{-T}$；
(3) 计算 $d_{i,m-1} = d_{i,m-1} - \sum_{j=0}^{m-2} b_{ij}^2 / d_{i+1,j}$
(4) 形成小的三对角线性方程组，并求解之；
(5) 求解整个问题。

这个算法具有很好的并行性，是求解这类问题非常有效的算法。通过简单的计算可以得出，该算法的并行计算复杂性比串行计算复杂性增加了一倍，虽然增加了并行性，但计算复杂性的增加降低了算法的效率。目前由于缺少更有效的三对角方程组的并行计算方法，该算法在求解此类问题时仍被广泛采用。由于它是分块在每个处理机上独立进行大量的运算，因此这个算法不难应用到块三对角线性方程组。

如果矩阵 L_i 的次对角线元素的积衰减得非常快，在计算 B_{i-1} 时，它的最后一行后面的元素将近似为 0，因此就不需要大量的计算，也不需要求解小的方程组。对于这种情况，当问题的规模较大时，算法的复杂性并没有增加一倍，只是增加了一个与处理机个数有关的常数，从而大幅度地提高并行求解效率。

6.5 经典迭代算法的并行化

数值代数方程组的求解方法，有直接法，也有迭代法。前面介绍了一些直接求解线性代数方程组的方法，这里简单介绍一下经典迭代求解线性代数方程组的方法。在下面介绍的方法中，假设矩阵的对角线元素都是非零的。

6.5.1 Jacobi 迭代法

考虑求解线性代数方程组

$$Ax = b \tag{6.10}$$

其中 A 是 $m \times m$ 矩阵，记 D、$-L$、$-U$ 分别是 A 的对角、严格下三角、严格上三角部分构成的矩阵，即 $A = D - L - U$。这时方程组 (6.10) 可以变为

$$Dx = b + (L + U)x \tag{6.11}$$

如果方程组 (6.11) 右边的 x 已知，由于 D 是对角矩阵，可以很容易求得左边的 x，这就是 Jacobi 迭代法的出发点。因此，对于给定的初值 $x^{(0)}$，Jacobi 迭代法如下：

$$x^{(k+1)} = D^{-1}(L + U)x^{(k)} + D^{-1}b \tag{6.12}$$

记 $G = D^{-1}(L + U) = I - D^{-1}A$，$g = D^{-1}b$。则每次迭代就是做矩阵向量乘，然后是向量加。亦即：

$$x^{+} = Gx + g \tag{6.13}$$

公式 (6.13) 的计算和前面介绍的同步并行计算方法是类似的，因此并行计算方法非常容易构造，这里就不再赘述。

6.5.2 Gauss-Seidel 迭代法

Gauss-Seidel 迭代法是逐个分量进行计算的一种方法，考虑线性代数方程组 (6.10) 的分量表示

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \qquad i = 1, \ldots, n \tag{6.14}$$

对于给定的初值 $x^{(0)}$，Gauss-Seidel 迭代法如下：

算法 6.15: (Gauss-Seidel 迭代算法)

- $k = 0$
- $x_1^{(k+1)} = (b_1 - \sum_{j=2}^{n} a_{1j} x_j^{(k)})/a_{11}$
- $x_2^{(k+1)} = (b_2 - a_{21} x_1^{(k+1)} - \sum_{j=3}^{n} a_{2j} x_j^{(k)})/a_{22}$
- ...
- $x_{n-1}^{(k+1)} = (b_{n-1} - \sum_{j=1}^{n-2} a_{n-1,j} x_j^{(k+1)} - a_{n-1,n} x_n^{(k)})/a_{n-1,n-1}$
- $x_n^{(k+1)} = (b_n - \sum_{j=1}^{n-1} a_{nj} x_j^{(k+1)})/a_{nn}$
- $\|x^{(k+1)} - x^{(k)}\|_2 < \epsilon \|x^{(k+1)} - x^{(0)}\|_2$? $k = k + 1$

从算法 6.15 的计算过程可以发现，每计算一个新的分量都需要前面所有新计算出来的分量的结果，这是一个严格的串行过程。那么，如何设计一个并行计算的方法呢？记 $s_i = \sum_{j=i+1}^{n} a_{ij} x_j^{(0)}$，$i = 1, \ldots, n-1$，$s_n = 0$。并行计算方法如下：

算法 6.16: (并行 Gauss-Seidel 迭代算法)

$k = 0$
for $i = 1$, n do
  $x_i^{(k+1)} = (b_i - s_i)/a_{ii}$, $s_i = 0$
  for $j = 1$, n, $j \neq i$ do
    $s_j = s_j + a_{ji} x_i^{(k+1)}$
  end{for}
end{for}
$\|x^{(k+1)} - x^{(k)}\|_2 < \epsilon \|x^{(k+1)} - x^{(0)}\|_2$? $k = k + 1$

在算法 6.16 中，每次并行计算 s_j，之后可以并行计算截止条件是否满足。这个并行计算方法与串行算法在计算量上是有些差别的。

6.6 异步并行迭代法

异步迭代算法在并行计算中起着重要的作用，因为此类算法不需要处理机之间的等待，使处理机的工作效率能够得到充分的发挥。这方面的研究工作早在 60 年代就已经开始，文献 [61] 中给出了线性迭代 $x = Bx + c$ 的收敛定理，当谱半径 $\rho(|B|) < 1$ 时，此迭代过程是异步迭代收敛的。继此之后，文献 [62] 给出了非线性迭代 $x = F(x)$ 的收敛定理，当 F 是 P– 收缩映射时，迭代过程是异步迭代收敛的。P– 收缩映射 (P–contraction) 的定义参见文献 [63]。

6.6.1 异步并行迭代法基础

首先引入关于 $x = F(x)$ 的异步迭代算法的定义，最后给出文献 [62] 中的一个结论。记 $|A|$ 为 A 的每个元素取绝对值的矩阵，$|x|$ 表示对向量 x 的分量取绝对值的向量，$A \geqslant 0$ 表示 A 的元素均大于或等于 0。$F(x)$ 的第 i 个分量记为 $f_i(x)$ 或 $f_i(x_1, \ldots, x_n)$，向量序列记为 $x^{(j)}$，$j = 0, 1, \ldots$，所有非负整数的集合记为 N。

定义 6.2: 设 F 是 $R^n \to R^n$ 的映射，则关于算子 F 和初始点 $x^{(0)}$ 的异步迭代是由下述递推关系定义的向量序列 $x^{(j)} \in R^n$，$j = 1, 2, \ldots$，

$$J = \{J_j | j = 1, 2, \ldots\}$$

$$S = \{(s_1^{(j)}, \ldots, s_n^{(j)}) \,|\, j = 1, 2, \ldots\}$$

$$x_i^{(j)} = \begin{cases} x_i^{(j-1)}, i \notin J_j \\ f_i(x_1^{(s_1^{(j)})}, \ldots, x_n^{(s_n^{(j)})}), i \in J_j \end{cases}$$

其中，J 是 $\{1, 2, \ldots, n\}$ 的非空子集构成的序列，S 是 N^n 中的一个序列。此外，对每个 $i = 1, \ldots, n$，J 和 S 满足如下三个条件：

(1) $s_i^{(j)} \leqslant j - 1$, $j = 1, 2, \ldots$;

(2) $s_i^{(j)}$ 作为 j 的函数趋于无穷大;

(3) i 在集合 J_j $(j = 1, 2, \ldots)$ 中出现无穷多次。

下边要用到文献 [62] 中的一个重要结果，这里以引理的形式给出。

引理 6.5: 假设 $|F(x) - F(y)| \leqslant A|x - y|$，其中 A 是非负矩阵并且 $\rho(A) < 1$。则迭代 $x = F(x)$ 是异步收敛的。

6.6.2 线性迭代的一般收敛性结果

在这一小节中，考虑求解线性方程组 $Ax = b$ 的一些迭代法的收敛性。对于线性迭代法，通常采用 A 的分裂形式，$A = B - C$，这时的迭代形式如下：

$$x = B^{-1}Cx + B^{-1}b \tag{6.15}$$

其中 B 是可逆的。由引理 6.5 可知，当 $\rho(|B^{-1}C|) < 1$ 时，上述的迭代过程是异步迭代收敛的。下面对 A 是 M 矩阵或对角占优矩阵的情况讨论其收敛性，这些结果容易推广到分块 M 矩阵或 H 矩阵，关于这些矩阵的定义可参见文献 [63]。下面不加证明地给出文献 [63] 中的一些结论。首先给出弱正则分裂的定义。

定义 6.3: 设 A，B 和 C 是实矩阵，如果 $A = B - C$，$B^{-1} \geqslant 0$ 和 $B^{-1}C \geqslant 0$，则称 $A = B - C$ 是 A 的弱正则分裂。

引理 6.6: 设 $A = B - C$ 是一弱正则分裂，则 $\rho(B^{-1}C) < 1$ 当且仅当 A^{-1} 存在并且 $A^{-1} \geqslant 0$。

引理 6.7: 设 B 和 C 是 $n \times n$ 阶矩阵，如果 $|C| \leqslant B$，则 $\rho(C) \leqslant \rho(B)$。

引理 6.8: 设 A 是严格或不可约对角占优矩阵，如果 $a_{ij} \leqslant 0$ $(i \neq j)$，且 $a_{ii} > 0$，则 A 是 M 矩阵。

下面不加证明地引用文献 [64] 中的两个结论。

定理 6.1: 设 $A = B - C$ 是 A 的弱正则分裂且 A 是 M 矩阵，则迭代形式 (6.15) 是异步迭代收敛的。

这个定理可应用到许多迭代法中，比如 Jacobi 和 Gauss-Seidel 型迭代法等。对于这种类型的迭代法，其矩阵 A 分解成 $A = D - L - U$，其中 D 是对角矩阵，L 和 U 分别是严格下三角矩阵和上三角矩阵。基于这种分裂形式的迭代过程的异步迭代收敛性可用下面的定理给出。

定理 6.2: 设 A 是严格或不可约对角占优矩阵,$A = B - C$, 其中 $B = D - \alpha L$，$C = (1 - \alpha)L + U$，$0 \leqslant \alpha \leqslant 1$，则迭代形式 (6.15) 是异步迭代收敛的。

这些结果在求解偏微分方程的差分离散的方程中非常有用。在区域分解计算中，采用异步迭代，信息的交换不需要等待，有助于提高并行处理的效率。

6.7 代数特征值问题的并行求解

代数特征值问题在科学与工程计算中非常重要。这一节重点介绍如何求解标准特征值问题。假设 A 是 $n \times n$ 阶实对称矩阵，即 $A \in R^n$，则标准特征值问题为：

$$Ax = \lambda x \tag{6.16}$$

称满足特征方程 (6.16) 的一对 (λ, x) 为矩阵 A 的特征对，其中 λ 称为矩阵 A 的特征值，x 称为矩阵 A 的特征向量。

6.7.1 对称三对角矩阵特征值问题

在特征值问题 (6.16) 中，考虑 A 是对称三对角矩阵的情形，即：

$$\begin{pmatrix} a_1 & b_1 & & & & \\ b_1 & a_2 & b_2 & & & \\ & b_2 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{n-2} & a_{n-1} & b_{n-1} \\ & & & & b_{n-1} & a_n \end{pmatrix} x = \lambda x \tag{6.17}$$

由于矩阵特征值是特征方程 $P_n(\lambda) = \det(A - \lambda I)$ 的根，因此，可以通过计算 $P_n(\lambda)$ 的根来求特征方程 (6.17) 的特征值。记 A_i 是矩阵 A 的左上角 i 阶主子式，则有如下的交错定理。

定理 6.3: 设所有的 $b_j \neq 0$，$j = 1, \ldots, n-1$。记 A_i 的特征值为 $\alpha_1 < \alpha_2 < \ldots < \alpha_i$，$A_{i+1}$ 的特征值为 $\beta_1 < \beta_2 < \ldots < \beta_i < \beta_{i+1}$，则 A_i 的特征值分隔 A_{i+1} 的特征值，即有 $\beta_1 < \alpha_1 < \beta_2 < \alpha_2 < \ldots < \beta_i < \alpha_i < \beta_{i+1}$。

由于矩阵 A 是对称三对角的，因此它的特征多项式是容易计算的，具有如下的递推形式：

$$\begin{cases} P_0(\lambda) = 1 \\ P_1(\lambda) = a_1 - \lambda \\ P_n(\lambda) = (a_n - \lambda)P_{n-1}(\lambda) - b_{n-1}^2 P_{n-2}(\lambda) \quad n = 2, \ldots \end{cases} \tag{6.18}$$

从数值计算的稳定性及方便性考虑，令 $Q_n(\lambda) = P_n(\lambda)/P_{n-1}(\lambda)$，则有：

$$\begin{cases} Q_1(\lambda) = a_1 - \lambda \\ Q_n(\lambda) = (a_n - \lambda) - b_{n-1}^2/Q_{n-1}(\lambda) \quad n = 2, \ldots \end{cases} \tag{6.19}$$

矩阵 A 的小于 α 的特征值的个数和关系式 (6.19) 中 $Q_i(\alpha)$ 小于 0 的个数相同，因此可以通过计算 $Q_i(a)$ 和 $Q_i(b)$ 来求一个给定区间 $[a,\ b)$ 内的特征值个数。

假设矩阵 A 的所有特征值在区间 $[\alpha_0,\ \alpha_n)$ 内，取 $\alpha = (\alpha_0 + \alpha_n)/2$，计算 $Q_i(\alpha)$ 小于 0 的个数，记这个数为 k，令 $\alpha_k = \alpha$，则可以得到两个新的小区间 $[\alpha_0,\ \alpha_k)$ 和 $[\alpha_k,\ \alpha_n)$。然后对每个小区间进行同样的对分，直到每个区间中都只包含一个特征值。假设区间 $[a,\ b)$ 内只包含矩阵 A 的 1 个特征值 λ，则有如下的计算该特征值的计算方法：

算法 6.17: (二分法)

$c = (a + b)/2$, if $|b - a| < \epsilon$, then stop

compute $Q_i(c)$ for all i

if $\lambda < c$, then $b = c$, otherwise $a = c$

对于给定的特征值 λ，其特征向量可以通过逆迭代来获得，迭代形式如下：

$$(A - \lambda I)x^{k+1} = x^k \tag{6.20}$$

对任意给定的无穷范数很小的初始值 x^0，即 $\|x^0\|_{\inf}$ 充分的小。一般情况下，迭代 1 至 2 次就可以得到所需要的特征向量。值得注意的是，方程 (6.20) 是奇异的，在进行求解的时候，需要选主元，最后一个主元可能是 0，这时候用 ϵ 来代替。由此计算特征值和特征向量的方法，是一个可以完全并行的计算方法。

6.7.2 Householder 变换

Householder 变换是一个特殊的正交变换，它具有如下的形式：

$$H = I - 2uu^{\mathrm{T}} \tag{6.21}$$

其中 $\|u\|_2 = 1$。容易验证矩阵 H 是一个正交矩阵。这个矩阵有什么好处呢？从 H 的表达式 (6.21) 可以看出，其形式非常简单，是

一个容易构造的正交矩阵。记 e_i 是单位矩阵 I 的第 i 列，则对于任何一个给定的向量 x，可以选择一个 Householder 变换，使得 $Hx = \alpha e_1$。下面介绍如何计算这个 Householder 变换。假设 u 是一个任意的向量，则 $H = I - 2uu^{\mathrm{T}}/\|u\|_2^2$ 是一个 Householder 变换。因为 $Hx = x - 2(u^{\mathrm{T}}x/\|u\|_2^2)u = \alpha e_1$，所以 $|\alpha| = \|x\|_2$。从线性代数的基础知识可知，u 一定是 x 和 e_1 的线性组合。因此，可以假设 $u = x + \beta e_1$，由此可得：

$$\frac{\beta^2 - \|x\|_2^2}{\|x\|_2^2 + 2\beta x_1 + \beta^2}x - \frac{2(\|x\|_2^2 + \beta x_1)}{\|x\|_2^2 + 2\beta x_1 + \beta^2}\beta e_1 = \alpha e_1 \tag{6.22}$$

为使等式 (6.22) 对任意的 x 成立，必有 $\beta^2 - \|x\|_2^2 = 0$。在计算过程中，为保证数值稳定性，取 $\beta = \mathrm{sign}(x_1)\|x\|_2$。由此可以得出，$\alpha = -\beta$。

Householder 变换在数值计算中是非常重要的，许多和正交矩阵分解有关的问题的计算，如矩阵 QR 分解、化对称矩阵为三对角矩阵等，都离不开它。

6.7.3 化对称矩阵为三对角矩阵

在这一节中，考虑如何将对称矩阵化为三对角矩阵。记

$$A = \begin{pmatrix} \alpha & u^{\mathrm{T}} \\ u & B \end{pmatrix} \tag{6.23}$$

由前述的讨论可知，存在一个 Householder 变换 H，使得 $Hu = \beta e_1$。令 $G = \begin{pmatrix} 1 & 0 \\ 0 & H \end{pmatrix}$，则矩阵 G 也是正交矩阵。且有：

$$G^{\mathrm{T}}AG = \begin{pmatrix} 1 & 0 \\ 0 & H \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} \alpha & u^{\mathrm{T}} \\ u & B \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & H \end{pmatrix} = \begin{pmatrix} \alpha & \beta e_1^{\mathrm{T}} \\ \beta e_1 & H^{\mathrm{T}}BH \end{pmatrix} \tag{6.24}$$

同样可以对矩阵 $H^{\mathrm{T}}BH$ 做和式 (6.24) 相同的操作，由此可以得出所需的三对角矩阵。在并行计算中，Householder 变换是计算一个向

量，在一个处理机中进行，并行主要体现在每次计算 $H^{\mathrm{T}}BH$。下面考虑如何计算 $H^{\mathrm{T}}BH$。假设 Householder 变换 $H = I - 2vv^{\mathrm{T}}/(v^{\mathrm{T}}v)$，则有：

$$\begin{aligned} H^{\mathrm{T}}BH &= (I - 2vv^{\mathrm{T}}/(v^{\mathrm{T}}v))B(I - 2vv^{\mathrm{T}}/(v^{\mathrm{T}}v)) \\ &= B - 2vv^{\mathrm{T}}B/(v^{\mathrm{T}}v) - 2Bvv^{\mathrm{T}}/(v^{\mathrm{T}}v) + 4(v^{\mathrm{T}}Bv)vv^{\mathrm{T}}/(v^{\mathrm{T}}v)^2 \end{aligned} \tag{6.25}$$

令 $\tau = 2/(v^{\mathrm{T}}v)$，$x = Bv$，$w = \dfrac{v^{\mathrm{T}}x}{2}v - \tau x$，则：

$$H^{\mathrm{T}}BH = B - vw^{\mathrm{T}} - wv^{\mathrm{T}} \tag{6.26}$$

经过变形的修正公式 (6.26) 比直接计算方法 (6.25) 节约了一个对称秩 1 修正。在这里，并行计算主要包括两个部分，一是计算 $x = Bv$，另一个是计算公式 (6.26)。由此可见并行计算过程是比较简单的，就不再赘述。

习　题

1. 假设矩阵 B 按列卷帘方式存放在 q 个处理机中，向量 u 和 v 存放在每个处理机中，给出并行计算 $A = B - uv^{\mathrm{T}} - vu^{\mathrm{T}}$ 的方法。

2. 假设矩阵 A 是三对角的，试给出求解方程组 $Ax = b$ 的并行 Jacobi 迭代方法。

第 7 章　FFT 算法与应用

在 1965 年，两位美国科学家 J. W. Cooley 和 J. W. Tukey 发明了一种有效计算富氏变换的方法，被称之为 FFT（Fast Fourier Transform，快速富氏变换），该算法在众多的科学与工程计算中起着至关重要的作用，是 20 世纪计算科学的重要贡献之一。在这一章里，重点介绍 FFT 的串行算法、并行实现、以及 FFT 的应用。FFT 考虑快速计算

$$y_k = \sum_{j=0}^{n-1} x_j e^{-\frac{2\pi i j k}{n}}, \qquad k = 0, 1, \ldots, n-1. \tag{7.1}$$

的问题，其中 $i^2 = -1$。记 $\omega(n) = e^{-\frac{2\pi i}{n}}$，则 $\omega(n)^k$ 是方程 $x^n = 1$ 的根。下面的性质显然成立：

(1) $(\omega(n)^k)^n = 1$

(2) $\omega(n)^2 = \omega(n/2)$

(3) $\omega(n)^{n/2} = -1$

记 $Y = (y_0, y_1, \ldots, y_{n-1})^{\mathrm{T}}$，$X = (x_0, x_1, \ldots, x_{n-1})^{\mathrm{T}}$，$\Omega_{kj} = \omega(n)^{kj}$，则式 (7.1) 的计算过程可以写成矩阵乘向量的形式 $Y = \Omega X$。因此，直接计算式 (7.1) 需要 $O(n^2)$ 个浮点运算。由于 $\omega(n)$ 具有特殊性，假设 $n = 2m$，式 (7.1) 可以分成如下的计算过程

$$\begin{cases} y_k = \sum_{j=0}^{m-1} x_{2j}\omega(m)^{kj} + \omega(n)^k \sum_{j=0}^{m-1} x_{2j+1}\omega(m)^{kj} \\ y_{k+m} = \sum_{j=0}^{m-1} x_{2j}\omega(m)^{kj} - \omega(n)^k \sum_{j=0}^{m-1} x_{2j+1}\omega(m)^{kj} \\ k = 0, 1, \ldots, m-1 \end{cases} \tag{7.2}$$

假设 T_n 是计算所有 y_k 的计算量，由式 (7.2) 有 $T_n = 2T_{n/2} + 3/2n$，也即 $T_n = 3/2n\log_2 n + n$。在这里讨论的计算方法中，假定数据的长度是 $n = 2^m$。

7.1 一维串行 FFT 算法

公式 (7.2) 是 FFT 的一种计算方法基础，可以通过递推的方式来完成其计算任务。在计算过程中，需要大量的数据移动，从而使得计算效率有所降低。为了有效实现 FFT 方法，需要对数据进行重排序，使得新的数据顺序适合于计算过程。以 $n = 8$ 为例，计算过程的数据依赖关系见图 7.1。 在图 7.1 中，左边的第一列是最后的结果，右边的第一列是原始数据。从图 7.1 可以看出，在进行 FFT 算法的执行过程中，需要对原始数据进行重排序，以利于计算。数据重新排列的规则是按位倒置（bit reverse）方式进行的，以 $n = 16$ 为例，按位倒置变换如表 7.1 所示。

假设 $n = 2^m$，所有 x_j 按照按位倒置规则进行重新排序，记为 y_j。令 $D(2k) = \mathrm{diag}(\omega(2k)^j)$，$j = 0, \ldots, k-1$，它是 k 阶对角矩阵。按照公式 (7.2)，在时间上进行大幅度减少（decimation in time，DIT）的 FFT 算法如下：

算法 7.1： FFT 计算方法

(1) 置 $s = 1$，$t = 1$，$l = n/2$

(2) 计算所有长度为 $2t$ 的变换 l 个，其中每个变换的形式为 $Y = \begin{pmatrix} I & D(2t) \\ I & -D(2t) \end{pmatrix} Y$

(3) 如果 $s < m$，置 $s = s + 1$，$t = 2t$，$l = l/2$，重复上一步计算。

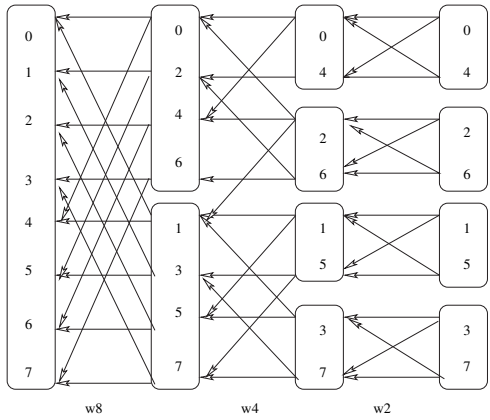


图 7.1 FFT 数据依赖关系

在这个算法中，其计算过程是从计算长度为 2 的一些变换开始，然后是长度为 4 的一些变换，最后得到长度为 2^m 的变换。每次的计算是非常简单的，比如已经有了 2 个长度为 2 的 FFT 的序列分别记为 u，v，则由它们产生的长度为 4 的序列 z 的计算公式如下：

$$\begin{cases} z_0 &= u_0 + \omega(4)^0 v_0 \\ z_1 &= u_1 + \omega(4)^1 v_1 \\ z_2 &= u_0 - \omega(4)^0 v_0 \\ z_3 &= u_1 - \omega(4)^1 v_1 \end{cases} \tag{7.3}$$

代码 7.1: 串行 FFT 算法的实现

文件名: `code/fft/fft1d.f`

```
1 ************************************************************************
2 * This is a subroutine for FFT transform program                       *
3 * Made by Dr. Xue-bin Chi                                              *
4 * Date: Aug.18, 2005                                                   *
```

表 7.1 按位倒置变换

原始顺序	第一次	第二次	第三次	十进制
0000	0000	0000	0000	0
0001	1000	1000	1000	8
0010	0001	0100	0100	4
0011	1001	1100	1100	12
0100	0010	0001	0010	2
0101	1010	1001	1010	10
0110	0011	0101	0110	6
0111	1011	1101	1110	14
1000	0100	0010	0001	1
1001	1100	1010	1001	9
1010	0101	0110	0101	5
1011	1101	1110	1101	13
1100	0110	0011	0011	3
1101	1110	1011	1011	11
1110	0111	0111	0111	7
1111	1111	1111	1111	15

```
5  * Supercomputing Center                                          *
6  * Computer Network Information Center, CAS                       *
7  *                                                               *
8  * Deal with the length N=2**m.                                  *
9  * x is an input array for FFT                                   *
10 * y is an output array for FFT                                  *
11 * ip is an output array for bit reverse order                   *
12 * w is an working space for saving exp(-2pijk/N)                *
13 *****************************************************************
14       subroutine fft1d( m, x, y, w, ip, f )
15       integer m, ip(*), f
16       complex*16 x(*), y(*), w(*)
```

```
      integer n, i, j, k, l, it, iw
      complex*16 s

      call bitreverse( m, ip )          !
      n = 2**m
      do 10 i = 1, n
        y(i) = x(ip(i)+1)
 10   continue                          !

      if ( m .eq. 0 ) then
        return
      endif

      it = n / 2
      n = 1
      iw = 0

      call oneroots( m, w, f )          !
      do 100 j=1, m
        l = 0
        do 90 k=1, it                   !
          do 80 i=1, n
            s = w(iw+i)*y(l+n+i)
            y(l+n+i) = y(l+i) - s
            y(l+i) = y(l+i) + s
 80       continue
          l = l + 2*n
 90     continue                         !
        iw = iw + n
        n = 2*n
        it = it / 2
100   continue

      if ( f .eq. -1 ) then             !
        n = 2**m
        s = 1.0/n
```

```
53          do 120 i=1, n
54            y(i)=s*y(i)
55  120     continue
56        endif                              !
57
58        return
59        end
```

在这个程序中，行 20–24 按照按位倒置方式对数据进行重新排序，行 34 产生 FFT 变换所需的一系列 $\omega(k)^j$，行 37–44 执行算法 7.1 的第二步，行 50–56 是逆变换需要对数据进行的缩放。

下面介绍计算 FFT 的另一种方法，称之为在频率上大幅度减少（decimation in frequency，DIF）的 FFT 算法。从公式 (7.1) 出发，其计算过程可以按照如下方式进行

$$\begin{cases} y_k &= \sum_{j=0}^{m-1} x_j\omega(n)^{jk} + \sum_{j=m}^{n-1} x_j\omega(n)^{jk} \\ &= \sum_{j=0}^{m-1} x_j\omega(n)^{jk} + \sum_{j=0}^{m-1} x_{j+m}\omega(n)^{mk+jk} \\ &= \sum_{j=0}^{m-1}(x_j + \omega(n)^{mk}x_{j+m})\omega(n)^{jk} \\ &= \sum_{j=0}^{m-1}(x_j + (-1)^k x_{j+m})\omega(n)^{jk} \\ & k = 0, 1, \ldots, n-1 \end{cases} \tag{7.4}$$

由此可以得出

$$\begin{cases} y_{2k} &= \sum_{j=0}^{m-1}(x_j + x_{j+m})\omega(m)^{jk} \\ y_{2k+1} &= \sum_{j=0}^{m-1}(x_j - x_{j+m})\omega(n)^j\omega(m)^{jk} \\ & k = 0, 1, \ldots, m-1 \end{cases} \tag{7.5}$$

和 DIT 方法相同，DIF 方法同样将一个大的变换转化成小的变换，从而节约计算量，达到快速计算的目的。在这里，计算是从大到小，在同一水平上的各个小的变换之间没有任何关系，可以完全独立地

进行计算。DIT 方法是将小的 FFT 合成一个大的 FFT，而 DIF 方法是每次将大的 FFT 化成小的相互独立的 FFT。图 7.2 给出了当 $n = 8$ 时的 FFT 计算过程。

7.2 二维串行 FFT 算法

$$\begin{cases} y_{k_x k_y} & = \sum_{j_x=0}^{n_x-1} \sum_{j_y=0}^{n_y-1} x_{j_x j_y} e^{-\frac{2\pi i j_x k_x}{n_x}} e^{-\frac{2\pi i j_y k_y}{n_y}} \\ & = \sum_{j_y=0}^{n_y-1} (\sum_{j_x=0}^{n_x-1} x_{j_x j_y} e^{-\frac{2\pi i j_x k_x}{n_x}}) e^{-\frac{2\pi i j_y k_y}{n_y}} \\ & = \sum_{j_y=0}^{n_y-1} z_{k_x j_y} e^{-\frac{2\pi i j_y k_y}{n_y}} \\ & k_x = 0, 1, \ldots, n_x - 1, k_y = 0, 1, \ldots, n_y - 1 \end{cases} \tag{7.6}$$

从公式 (7.6) 不难得出，$y_{k_x k_y}$ 的计算过程可以由两个方向的一维 FFT 来完成。在二维 FFT 的计算过程中，如果通过一维 FFT 来实现，那么没有新的算法需要进行讨论。但是，由于二维 FFT 所处理的是矩阵数据，具有其固有的特点，可以采用一种向量计算的方式来完成二维的 FFT 计算，从而提高计算性能。这里用到的向量运算

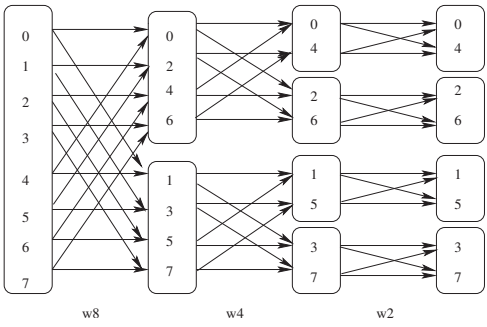


图 7.2 DIF FFT 计算过程

形式为 $y = ax + y$，其中 x 和 y 是向量，a 是标量。在基本线性代数子程序包 BLAS 中，`xAXPY` 子程序可以完成 $y = ax + y$ 的计算任务。但是在 FFT 算法中，其主要运算为 $y = y + ax$ 和 $x = y - ax$，因此有数据覆盖的问题。这里引进一个临时向量 $t = ax$，从而使其计算变为 $x = y - t$ 和 $y = y + t$，该计算方式可以解决 FFT 计算过程中的数据覆盖问题。如果不增加额外的存储开销，上述计算需要增加额外的计算来完成。在 BLAS 中还有一个子程序 `xSCAL`，用于计算 $x = ax$。因此计算 $y = y + ax$ 和 $x = y - ax$ 可以按照如下方式进行：

(1) 计算 $x = -\dfrac{1}{a}y + x$ 即 `call xAXPY(n,-1/a,y,1,x,1)`

(2) 计算 $x = -ax$ 即 `call xSCAL(n,-a,x,1)`

(3) 计算 $y = -\dfrac{1}{2}x + y$ 即 `call xAXPY(n,1/2,x,1,y,1)`

(4) 计算 $y = 2y$ 即 `call xSCAL(n,2,y,1)`

在二维 FFT 的计算公式 (7.6) 中，Z 矩阵的行可以如此计算。同样，在计算 Y 矩阵时，其每一列也可以这样来计算。由于在计算二维 FFT 时，其计算可以表示成一维的 FFT，因此重要的是如何对数据进行合理的安排。在以上的分析与讨论中，给出了两种在计算二维 FFT 时的实现方法。对于更高维的 FFT，也可以用类似的方法来计算，因此这里就不再赘述。下面将重点阐述如何并行计算 FFT。

7.3 并行 FFT 算法

在这一节中，给出针对一维 FFT 的并行计算方法，并对其加以分析推广到二维 FFT 的并行计算中。为讨论方便，假设下述条件成立：

假设 7.1: 数据长度和处理机个数满足:

(1) 数据的长度为 $n = 2^m$;

(2) 处理机个数 $p = 2^q$;

(3) $m \geqslant 2q$。

最后一个假设中,要求数据长度大于或等于处理机个数的平方。由于并行计算处理的是大规模的问题,因此该假设并不苛刻。

这里再次强调,并行计算中最重要的问题是数据的划分,算法设计是根据数据在处理机中的存放形式确定的。在这里讨论 FFT 的并行计算方法,所考虑的数据是分块存放在各个处理机中的,亦即公式 (7.1) 中的向量 x 和 y 是分块方式存放在处理机中的。记处理机为 P_i, $i = 0, \ldots, p-1$,在每个处理机中存放的 x 和 y 的部分为 X_i 和 Y_i,其中 X_i 和 Y_i 的长度为 $s = n/p$。令 $B = \{b_0, b_1, \ldots, b_{n-1}\}$ 为 $N = \{0, 1, \ldots, n-1\}$ 的按位倒置序列,将 B 按顺序平均分成 p 块,记为 B_i, $i = 0, \ldots, p-1$,则 $|B_i| = s$。

引理 7.1: 令 $v_{ij} = |V_{ij} = \{b : b \in B_i, b/s = j\}|$, $i, j = 0, \ldots, p-1$, n 和 p 满足假设 7.1,则有

(1) $v_{ij} = s/p$;

(2) 如果 $b_k/s = j$, $b_{k+p}/s = j$;

(3) B_i/s 的前面 p 个元素是 $\{0, 1, \ldots, p-1\}$ 的按位倒置序列;

(4) B_i 中 $b_{ij}/s = k$ 的序列是长度为 s/p 的按位倒置序列经过放大和平移获得的。

这个引理的证明并不困难，这里举例说明该引理所表达的内容。以 $n = 8$，$p = 2$ 为例，这时

$$B = \{0,4,2,6,1,5,3,7\}, \quad B_0 = \{0,4,2,6\}, \quad B_1 = \{1,5,3,7\}$$
$$s = 4, \quad v_{01} = |V_{01} = \{4,6\}| = s/p = 2, \quad b_1/s = 1 = b_{1+2}/s$$

有了这个引理，在处理机中进行数据交换就非常方便。

在算法 7.1 中，数据已经进行了按位倒置重新排序。从 FFT 的计算方法中可以看出，其计算是可以并行完成的。在进行计算时，需要进行按位倒置重新排序，因此也就需要对每个处理机中的数据进行交换。记 B_i 中的序列为 $\{b_{ij}\}_{j=0}^{s-1}$，如果 $k = b_{ij}/s$，则 P_i 中的 x_j 需要与 P_k 中的数据进行交换。事实上，由于按位倒置重新排序是非常有规律的，在 P_k 中需要与 P_i 进行交换的数据是一一对应的，因此只要第一个需要交换的数据位置确定之后，就可以进行处理机之间的数据交换。

算法 7.2: 并行 FFT 算法

(1) 计算长度为 p 和 n/p^2 的按位倒置序列；

(2) 进行处理机内部的数据交换和外部的数据交换；

(3) 在每个处理机中计算长度为 s 的 FFT 变换；

(4) 将这些长度为 s 的 FFT 变换合成为长度为 n 的 FFT 变换；

代码 7.2: 并行 FFT 算法的 MPI 实现程序

文件名: `code/fft/mpifft1d.f`

```
1 ************************************************************************
2 * This is a subroutine for FFT transform program                       *
3 * Made by Dr. Xue-bin Chi                                              *
4 * Date: Sept. 12, 2005                                                 *
```

```
5  * Supercomputing Center                                              *
6  * Computer Network Information Center, CAS                           *
7  *                                                                    *
8  * Deal with the length N=2**m.                                       *
9  * x is an input array for FFT                                        *
10 * y is an output array for FFT                                       *
11 * iwork is a working space at least having the length of             *
12 *       2**(q+q) + 2**q + 2**(m-2*q)                                 *
13 * work is an working space for saving exp(-2pijk/N) & as a temporary  *
14 * space for data transfer having at least 2**m+2**(m-q)-1            *
15 **********************************************************************
16       subroutine mpifft1d( m, q, comm, iam, x, y, iwork, work, f )
17       include 'mpif.h'
18
19       integer m, q, comm, iam, iwork(*), f
20       complex*16 x(*), y(*), work(*)
21       integer n, p, ibr, imap, ipb, cmpl16, ierr, brdt, nsr,
22      &        cnt, lng, str, mst, s, i, tw, stat(mpi_status_size),
23      &        it, iw, l, j, k, n1, n2, siw, js, ks
24       complex*16 w
25
26       tw = 2**m
27       p = 2**q
28       s = 2**(m-q)
29       call zcopy( s, x, 1, y, 1 )
30       imap = 1
31       ipb = imap + p*p
32       ibr = ipb + p
33       call mapping( q, iwork(imap), p )         !
34       call bitreverse( q, iwork(ipb) )
35       call bitreverse( m-2*q, iwork(ibr) )
36       call oneroots( m, work, f )
37
38       lng = 1
39       str = p
40       cnt = 2**(m-2*q)
```

```
*data exchange among innerprocessor
      do 20 j=0, cnt-1                          !
        k = iwork(ibr+j)
        if ( k .gt. j ) then
          js = j*str
          ks = k*str
          do 10 i=1, p
            w = y(js+i)
            y(js+i) = y(ks+i)
            y(ks+i) = w
 10       continue
        endif
 20   continue                                  !
      call mpi_type_contiguous( 2, mpi_double_precision, cmpl16, ierr ) !
      call mpi_type_commit( cmpl16, ierr )
* bit reverse order data type
      call mpi_type_vector( cnt, lng, str, cmpl16, brdt, ierr )
      call mpi_type_commit( brdt, ierr )        !
      ibr = 1 + iam*p
* data communication among processors
      do 50 i=1, p-1                            !
        nsr = iwork(ibr+i)
        mst = iwork(ipb+nsr)+1
        call zcopy( cnt, y(mst), str, work(tw), 1 )
        if ( iam .lt. nsr ) then
          call mpi_send(work(tw), cnt, cmpl16, nsr, 1, comm, ierr )
          call mpi_recv(y(mst), 1, brdt, nsr, 1, comm, stat, ierr )
        else
          call mpi_recv(y(mst), 1, brdt, nsr, 1, comm, stat, ierr )
          call mpi_send(work(tw), cnt, cmpl16, nsr, 1, comm, ierr )
        endif
 50   continue                                  !

      it = s / 2
      n = 1
      iw = 0
```

```
77
78        do 100 j=1, m-q                          !
79          l = 0
80          do 90 k=1, it
81            do 80 i=1, n
82              w = work(iw+i)*y(l+n+i)
83              y(l+n+i) = y(l+i) - w
84              y(l+i) = y(l+i) + w
85   80         continue
86            l = l + 2*n
87   90       continue
88          iw = iw + n
89          n = 2*n
90          it = it / 2
91  100     continue                                 !
92
93        n1 = 1
94        do 110 j=1, q                            !
95          n2 = n1*2
96          k = mod( iam, n2 )
97          if ( k .lt. n1 ) then
98            nsr = iam + n1
99            siw = k*s
100           call mpi_recv( work(tw), s, cmpl16, nsr, 1, comm, stat, ierr )
101           call mpi_send( y, s, cmpl16, nsr, 1, comm, ierr )
102           do 103 i=1, s
103             y(i) = y(i) + work(iw+siw+i)*work(tw+i-1)
104  103        continue
105         else
106           k = k - n1
107           nsr = iam - n1
108           siw = k*s
109           call zcopy( s, y, 1, work(tw), 1 )
110           call mpi_send( work(tw), s, cmpl16, nsr, 1, comm, ierr )
111           call mpi_recv( y, s, cmpl16, nsr, 1, comm, stat, ierr )
112           do 107 i=1, s
```

```
113             y(i) = y(i) - work(iw+siw+i)*work(tw+i-1)
114 107       continue
115         endif
116         iw = iw + n
117         n = 2*n
118         n1 = n2
119 110   continue                                !
120
121       call mpi_type_free( brdt, ierr )
122       call mpi_type_free( cmpl16, ierr )
123
124       if ( f .eq. -1 ) then
125         n = tw
126         w = 1.0/n
127         do 120 i=1, s
128           y(i)=w*y(i)
129 120     continue
130       endif
131
132       return
133       end
```

在这个并行程序中，行 33 产生每次数据交换的处理机对，其第 j 列对应 P_{j-1}，从第 2 行开始，表示每次数据与哪个处理机进行交换。行 42–53 将要与其他处理机进行交换的数据按照按位倒置方式进行交换，行 54–58 定义 MPI 程序的两个新数据类型，行 61–72 交换处理机之间的数据，以形成 FFT 算法所需要的数据，行 78–91 在每个处理机中做长度为 n/p 的 FFT 变换，行 94–119 合成整体 FFT 变换。

对于二维 FFT 变换，假设处理机个数为 $2^{q_0} \times 2^{q_1}$，矩阵的阶数为 $2^{m_0} \times 2^{m_1}$。将矩阵按块存放在处理机 P_{ij} 中，亦即输入矩阵 X 和

输出矩阵 Y 的分块形式为：

$$X = \begin{pmatrix} X_{00} & X_{01} & \cdots & X_{0,2^{q_1}-1} \\ X_{10} & X_{11} & \cdots & X_{1,2^{q_1}-1} \\ \vdots & \vdots & \vdots & \vdots \\ X_{2^{q_0}-1,0} & X_{2^{q_0}-1,1} & \cdots & X_{2^{q_0}-1,2^{q_1}-1} \end{pmatrix} \tag{7.7}$$

矩阵 Y 与 X 的分块方式是相同的，其中 X_{ij} 和 Y_{ij} 是存放在处理机 P_{ij} 中的 $2^{(m_0-q_0)} \times 2^{(m_1-q_1)}$ 矩阵。因此，只要 m_0 和 q_0、m_1 和 q_1 满足一维 FFT 的假设要求，就可以在 x 和 y 方向逐一进行并行的一维 FFT 变换，这里就不再赘述。

7.4 FFT 应用示例

FFT 算法在众多的领域中有着广泛的应用，这一节介绍应用 FFT 计算多项式的乘积和循环矩阵方程组的求解。从这两个例子可以看到，FFT 是一种非常有效的方法，可以极大地缩短许多问题的计算时间。

7.4.1 多项式相乘

假设要计算多项式 $f = \sum_{i=0}^{m} a_i x^i$ 和 $g = \sum_{i=0}^{n} b_i x^i$ 的乘积，亦即计算 $h = fg$。通常情况下 h 是一个 $k = m + n$ 阶多项式，这里假定 a_m 和 b_n 是非零的。因此计算 h 就是计算所有的系数 c_i 使得

$$h = \sum_{i=0}^{k} c_i x^i \tag{7.8}$$

众所周知，直接计算所有的系数 c_i 需要的计算量为 $O(mn)$，这里使用 FFT 可以减少计算量，使其计算复杂性为 $O((m+n)\log_2(m+n))$。

一个多项式的系数表示，如 (7.8) 是通常习惯的一种多项式的表示方法。还有一种数值表示方式，比如多项式 f 可以由 $\{(x_i, f_i)\}_{i=0}^{m}$

表示，其中 x_i 是互不相同的，且 $f_i = f(x_i)$。由系数表示方式变成数值表示方式是非常简单的，只需计算出 $f(x_i)$。从数值表示方式转化成系数表示方式虽然并不复杂，但需要通过如下的方法来达到目的。

记

$$L_i(x) = (x - x_0)\cdots(x - x_{i-1})(x - x_{i+1})\cdots(x - x_m) = \prod_{j=0, j\neq i}^{m}(x - x_j)$$

$L_i(x)$ 是一个 m 阶多项式。那么，对于给定的数值表示 $\{(x_i, f_i)\}_{i=0}^m$，定义 m 阶多项式 F 为

$$F = \sum_{i=0}^{m} \frac{L_i(x)}{L_i(x_i)} f_i \tag{7.9}$$

由 $L_i(x)$ 的特殊性可知，$F(x_i) = f(x_i)$，所以 $F = f$。因此，由关系式 (7.9) 可以得出多项式 f 的系数表示。

用 FFT 快速计算 $h = fg$，是通过计算 h 在 $k+1$ 个点的数值得到其数值表示方式，然后将数值表示方式转变成系数表示方式。计算一个 $f(x_i)$ 需要 $2m$ 次计算，因此，如果计算所有的 $f(x_i)$，$i = 0, 1, \ldots, m$ 需要 $2m(m+1)$ 次计算。也就是说，如果没有对 $\{x_i\}_{i=0}^m$ 进行选择，将系数表示转变成数值表示需要计算量为 $2m(m+1)$。如果是这样，对计算多项式 h 没有任何好处。由于只要选择的 $\{x_i\}_{i=0}^m$ 是互不相同的，多项式的两种表示就是等价的。为了能够使用快速方法将系数表示变成数值表示，记 $\omega = e^{\frac{2\pi i}{m+1}}$，取 $x_j = \omega^j$，则

$$f_j = f(x_j) = \sum_{l=0}^{m} a_l x_j^l = \sum_{l=0}^{m} a_l \omega^{jl} \tag{7.10}$$

上式相当于对 a 做长度为 $m+1$ 的 FFT 变换。如果已知 $\{(x_i, f_i)\}_{i=0}^m$，则可以由 FFT 的逆变换得到 a，从而实现用 $O(m \log_2 m)$ 计算量来完成两种表示的转换。对于计算多项式 h 的，只需分别计算出 $\{f_j\}_{j=0}^k$

和 $\{g_j\}_{j=0}^{k}$，然后计算 $\{f_j g_j\}_{j=0}^{k}$ 的逆 FFT 就可以得到 h 的系数表示方式。

7.4.2 循环矩阵方程组的求解

在科学与工程计算中，经常会遇到一类特殊的线性代数方程组，就是系数矩阵是循环矩阵，它是一种特殊的 Toeplitz 阵。以一个 $n = 6$ 阶的循环矩阵为例，它具有以下形式 [52]

$$C(w) = \begin{pmatrix} w_0 & w_5 & w_4 & w_3 & w_2 & w_1 \\ w_1 & w_0 & w_5 & w_4 & w_3 & w_2 \\ w_2 & w_1 & w_0 & w_5 & w_4 & w_3 \\ w_3 & w_2 & w_1 & w_0 & w_5 & w_4 \\ w_4 & w_3 & w_2 & w_1 & w_0 & w_5 \\ w_5 & w_4 & w_3 & w_2 & w_1 & w_0 \end{pmatrix} \tag{7.11}$$

这是由一个 6 维向量产生的循环矩阵的基本形式。对于一般的由 n 维向量 w 产生的循环矩阵 $C(w) = (c_{ij})$，$c_{ij} = w_{i-j \mod n}$。

记 n 阶 FFT 矩阵为 F_n，即 $F_n = (\omega^{kl})_{k,l=0}^{n-1}$，令 $u = F_n w$，则有 F_n 的第 k 行与 $C(w)$ 的第 l 列的内积为

$$\begin{aligned} &(F_n)_k (C(w))_l \\ &= \sum_{j=0}^{n-1} \omega^{kj} w_{j-l \mod n} = \sum_{j=0}^{l-1} \omega^{kj} w_{n-l+j} + \sum_{j=l}^{n-1} \omega^{kj} w_{j-l} \\ &= \sum_{j=0}^{l-1} \omega^{kj} w_{n-l+j} + \sum_{j=0}^{n-1} \omega^{k(j+l)} w_j - \sum_{j=n-l}^{n-1} \omega^{k(j+l)} w_j \\ &= \sum_{j=n-l}^{n-1} \omega^{k(j+l-n)} w_j + \omega^{kl} u_k - \sum_{j=n-l}^{n-1} \omega^{k(j+l)} w_j = \omega^{kl} u_k \end{aligned} \tag{7.12}$$

所以有 $F_n C(w) = \mathrm{diag}(u) F_n$。据此可知，一个循环矩阵可以用 FFT 矩阵进行对角化。如果要求解的方程组为 $C(w)x = b$，用 FFT 求解

的方法可以归纳如下

(1) 计算 w 的 FFT 变换，即 $u = F_n w$；

(2) 计算 b 的 FFT 变换，即 $a = F_n b$；

(3) 求解 $\mathrm{diag}(u)y = a$，即 $y = \mathrm{diag}(u)^{-1}a$；

(4) 计算 y 的 FFT 逆变换，即 $x = F_n^{-1}y$。

因此，求解循环矩阵方程组可以在 $O(n\log_2 n)$ 时间内完成。如果是计算循环矩阵和向量的乘积，也同样可以用 FFT 来快速计算，其计算过程如同求解方程组 $C(w)x = b$。

第 8 章　二维 Poisson 方程

本章介绍一个采用 5 点差分格式、结合点 Jacobi 迭代求解二维 Poisson 方程的 MPI Fortran 程序实例。这个算法本身比较简单，效率也很低，但是它包含了规则网格上基于区域分解的偏微分方程计算 MPI 程序的典型结构和通信模式，非常有代表性。本章介绍的程序实例稍加修便可以用于许多其他类似问题的计算。此外，该程序亦不难推广到三维问题以及非定常问题 (主要适用于显式格式及某些隐式格式)。

本章将首先简单介绍二维区域上 Poisson 方程的 5 点差分离散以及如何用 Jacobi 迭代来求解所导出的线性方程组。在此基础上给出基于区域分解的并行算法和 MPI 并行程序 Fortran 代码。同时，为该算法建立并行模型对算法的并行效率进行分析。最后，从几个不同的角度对 MPI 并行程序进行改进。

考虑定义在二维规则区域上的 Poisson 方程：

$$\begin{cases} -\Delta u(x,y) = f(x,y) \quad (x,y) \in \Omega = (0,W) \times (0,H) \\ u(x,y)|_{\partial\Omega} = g(x,y) \end{cases} \tag{8.1}$$

其中，$f(x,y)$ 和 $g(x,y)$ 为已知函数，分别定义在区域 Ω 的内部和边界。

沿坐标轴 x 和 y 方向，分别取步长

$$h_x = \frac{W}{\texttt{IM}}, \qquad h_y = \frac{H}{\texttt{JM}} \tag{8.2}$$

将区域 Ω 离散成规模为 $\texttt{IM} \times \texttt{JM}$ 的网格，其中 IM 和 JM 分别为沿坐标轴 x 和 y 方向的网格单元个数。

假设方程 (8.1) 的离散解 $u(x,y)$ 定义在所有网格结点上，且用如下未知量表示

$$\begin{cases} u_{i,j} = u(i \times h_x, j \times h_y) & 1 \leqslant i \leqslant \mathtt{IM}-1, \quad 1 \leqslant j \leqslant \mathtt{JM}-1 \\ u_{i,j} = g_{i,j} = g(i \times h_x, j \times h_y) & i = 0 \text{ 或 } i = \mathtt{IM} \text{ 或 } j = 0 \text{ 或 } j = \mathtt{JM} \end{cases} \tag{8.3}$$

用二阶中心差商近似导数：

$$\begin{cases} u_{xx}(i \times h_x, j \times h_y) \approx \dfrac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} \\ u_{yy}(i \times h_x, j \times h_y) \approx \dfrac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} \end{cases} \tag{8.4}$$

并记

$$f_{i,j} = f(i \times h_x, j \times h_y) \tag{8.5}$$

将以上公式代入方程 (8.1)，便得到了它的 5 点差分离散。而问题则转化为求解稀疏线性代数方程组：

$$2(h_x^2 + h_y^2)u_{i,j} - h_y^2(u_{i-1,j} + u_{i+1,j}) - h_x^2(u_{i,j-1} + u_{i,j+1}) = h_x^2 h_y^2 f_{i,j}, \quad 1 \leqslant i \leqslant \mathtt{IM}-1, \quad 1 \leqslant j \leqslant \mathtt{JM}-1 \tag{8.6}$$

该方程组包含 $(\mathtt{IM}-1) \times (\mathtt{JM}-1)$ 个未知量 $u_{i,j}$，$i = 1, \ldots, \mathtt{IM}-1$，$j = 1, \ldots, \mathtt{JM}-1$。

这里，采用众所周知的 Jacobi 点迭代算法求解方程组 (8.6)。从任意一个初始近似解

$$u_{i,j}^0, \quad i = 1, \ldots, \mathtt{IM}-1, \quad j = 1, \ldots, \mathtt{JM}-1$$

出发，迭代计算

$$\begin{cases} u_{i,j}^k = \dfrac{h_x^2 h_y^2 f_{i,j} + h_y^2(u_{i-1,j}^{k-1} + u_{i+1,j}^{k-1}) + h_x^2(u_{i,j-1}^{k-1} + u_{i,j+1}^{k-1})}{2(h_x^2 + h_y^2)} \\ i = 1, \ldots, \mathtt{IM}-1, \quad j = 1, \ldots, \mathtt{JM}-1 \end{cases}$$

$k = 1, 2, \ldots$，直到近似解 $u_{i,j}^k$ 满足误差要求。

在程序实例中取 $f(x, y) = -4$，方程 (8.1) 的解析解为 $u(x, y) = x^2 + y^2$，此时离散方程和原方程的解是一样的。由于离散方程的精确解已知，因此程序中直接比较近似解与精确解之间的误差来判断近似解是否满足误差要求，当不知道离散方程的精确解时，可以计算近似解的余量来判断是否达到收敛要求。初始近似解取为 $u_{i,j}^0 = 0$，$i = 1, \ldots, \mathtt{IM} - 1$，$j = 1, \ldots, \mathtt{JM} - 1$。

8.1 并行算法设计

设计求解方程 (8.1) 的 MPI 并行算法必须考虑两个关键问题：

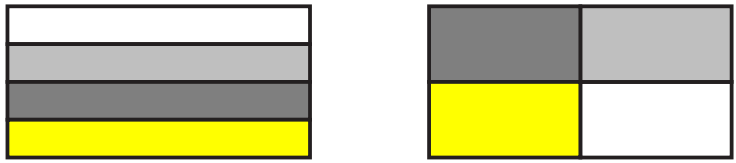
第一，选择恰当的区域分解策略，将区域 Ω 分解成多个子区域，分配给不同的进程，并保证进程间的负载平衡和最小的消息传递通信开销。通常有两种方式：

(1) 沿 x 方向或 y 方向的一维条分解策略，图 8.1(a) 显示了沿 y 方向划分的情况；

(2) 沿两个方向的二维块分解策略，如图 8.1(b) 所示。显然，如果某个方向的进程个数等于 1，则二维块分解策略就退化为一维条分解策略。无论哪种方式，都应该尽量保证每个子区域包含的网格结点个数相等，因为这样才能保证进程间的负载平衡。

第二，选择合适的通信数据结构。由式 (8.6) 可知，在任意网格结点上，执行 Jacobi 点迭代需要知道该结点上、下、左、右四个相邻结点上的近似解。因此，在每次 Jacobi 迭代之前，每个进程必须与其相邻的进程交换边界结点上的近似解。

为了描述通信数据结构，不妨设方程近似解定义在网格单元的中心。图 8.2 给出了一个 3×3 的二维块区域分解，其中，各个子区域被分配给不同的进程，各个进程负责求解该子区域的近似解。具



(a) 一维条分解 (b) 二维块分解

图 8.1 两种区域分解策略

体地，进程 5 将向其相邻的四个进程 (进程 2、进程 4、进程 6 和进程 8) 输出 “●” 标示的网格单元的近似解；同时，从这四个进程接收用 “○” 标示的网格单元上的近似解。因此，如何管理这些沿区域分解边界交换的网格单元上的量，将会直接影响到 MPI 程序的并行性能。图 8.3 给出了一个比较有效的办法，就是沿各个子区域的边界，向外增加一个宽度为 1 的辅助网格单元，用于存储相邻子区域在这些网格单元上的近似解。

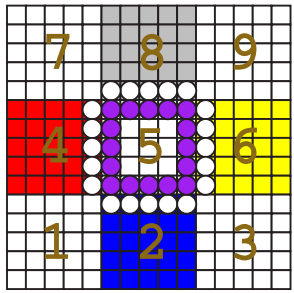


图 8.2 3×3 的二维块区域分解

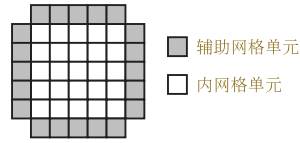


图 8.3 辅助网格单元示意图

类似地，同样的数据结构也适应于近似解定义在网格结点上的情形，这里不再讨论。

8.2 MPI 并行程序设计

下面，基于以上二维块区域分解策略和通信数据结构，给出求解差分方程 (8.6) 的 MPI 并行程序的 Fortran 代码。其中，近似解定义在网格结点上。为简单起见，这里假设进程数 $p = \texttt{NPX} \times \texttt{NPY}$，NPX 和 NPY 分别是沿 x 方向和 y 方向的进程个数，网格单元个数 IM 和 JM 能分别被 NPX 和 NPY 整除，子区域的网格规模为 IML × JML，其中 IML = IM/NPX，JML = JM/NPY，进程按自然序排列 (先沿 x 方向，后沿 y 方向)。

代码 8.1： 点 Jacobi 迭代 MPI 并行程序：二维块分解策略。

文件名: code/poisson/poisson0.f

```
! Poisson 方程求解: 使用阻塞通信 (可能死锁)。作者: 莫则尧
      INCLUDE  'mpif.h'
      PARAMETER(DW=2.0, DH=3.0)  ! 问题求解区域沿 X、Y 方向的大小
      PARAMETER(IM=30, JM=60)    ! 沿 X、Y 方向的全局网格规模
      PARAMETER(NPX=1,  NPY=1)   ! 沿 X、Y 方向的进程个数
      PARAMETER(IML=IM/NPX, JML=JM/NPY)
        ! 各进程沿 X、Y 方向的局部网格规模
      REAL  U(0:IML+1, 0:JML+1)  ! 定义在网格结点的近似解
      REAL  US(0:IML+1, 0:JML+1) ! 定义在网格结点的精确解
      REAL  U0(IML, JML)         ! Jacobi 迭代辅助变量
      REAL  F(IML, JML)          ! 函数 f(x,y) 在网格结点上的值
      INTEGER  NPROC             ! mpirun 启动的进程个数, 必须等于 NPX*NPY
      INTEGER  MYRANK,MYLEFT,MYRIGHT,MYUPPER,MYLOWER
                                 ! 各进程自身的进程号, 4 个相邻进程的进程号
      INTEGER  MEPX,MEPY         ! 各进程自身的进程号沿 X、Y 方向的坐标
      REAL  XST,YST              ! 各进程拥有的子区域沿 X、Y 方向的起始坐标
      REAL  HX, HY               ! 沿 X、Y 方向的网格离散步长
```

```
      REAL  HX2,HY2,HXY2,RHXY
      INTEGER  IST,IEND,JST,JEND
                    ! 各进程沿 X、Y 方向的内部网格结点的起始和终止坐标
      INTEGER  HTYPE, VTYPE
                    ! MPI 用户自定义数据类型, 表示各进程沿 X、Y 方向
                    ! 与相邻进程交换的数据单元
      INTEGER  STATUS(MPI_STATUS_SIZE)  !
      DOUBLE PRECISION T0, T1
! In-line functions
      solution(x,y)=x*x+y*x    ! 解析解
      rhs(x,y)=-4.0             ! Poisson 方程源项 (右端项)
! 程序可执行语句开始
      CALL MPI_Init(IERR)
      CALL MPI_Comm_size(MPI_COMM_WORLD,NPROC,IERR)
      IF (NPROC.NE.NPX*NPY.OR.MOD(IM,NPX).NE.0.OR.MOD(JM,NPY).NE.0) THEN
         PRINT *, '+++ mpirun -np xxx error OR grid scale error, ',
     &           'exit out +++'
         CALL MPI_Finalize(IERR)
         STOP
      ENDIF
! 按自然序 (先沿 X 方向, 后沿 Y 方向) 确定各进程自身及其 4 个相邻进程的进程号
      CALL MPI_Comm_rank(MPI_COMM_WORLD,MYRANK,IERR)
      MYLEFT  = MYRANK - 1
      IF (MOD(MYRANK,NPX).EQ.0)   MYLEFT=MPI_PROC_NULL
      MYRIGHT = MYRANK + 1
      IF (MOD(MYRIGHT,NPX).EQ.0)  MYRIGHT=MPI_PROC_NULL
      MYUPPER = MYRANK + NPX
      IF (MYUPPER.GE.NPROC)       MYUPPER=MPI_PROC_NULL
      MYLOWER = MYRANK - NPX
      IF (MYLOWER.LT.0)           MYLOWER=MPI_PROC_NULL
      MEPY=MYRANK/NPX
      MEPX=MYRANK-MEPY*NPX
!                     对应二维 NPYxNPX Cartesian 行主序坐标为 (MEPY,MEPX).
! 基本变量赋值, 确定各进程负责的子区域
      HX =DW/IM
      HX2=HX*HX
```

```
      HY =DH/JM
      HY2=HY*HY
      HXY2=HX2*HY2
      RHXY=0.5/(HX2+HY2)
      DX=HX2*RHXY
      DY=HY2*RHXY
      DD=RHXY*HXY2
      XST=MEPX*DW/NPX
      YST=MEPY*DH/NPY
      IST=1
      IEND=IML
      IF (MEPX.EQ.NPX-1) IEND=IEND-1   ! 最右边的区域 X 方向少一个点
      JST=1
      JEND=JML
      IF (MEPY.EQ.NPY-1) JEND=JEND-1   ! 最上边的区域 Y 方向少一个点
! 数据类型定义
      CALL MPI_Type_contiguous(IEND-IST+1, MPI_REAL, HTYPE, IERR)
      CALL MPI_Type_commit(HTYPE, IERR)
                ! 沿 X 方向的连续 IEND-IST+1 个 MPI_REAL 数据单元,
                ! 可用于表示该进程与其上、下进程交换的数据单元
      CALL MPI_Type_vector(JEND-JST+1, 1, IML+2, MPI_REAL, VTYPE, IERR)
      CALL MPI_Type_commit(VTYPE, IERR)
                ! 沿 Y 方向的连续 JEND-JST+1 个 MPI_REAL 数据单元,
                ! 可用于表示该进程与其左、右进程交换的数据单元
! 初始化
      DO J=JST-1, JEND+1
      DO I=IST-1, IEND+1
         xx=(I+MEPX*IML)*HX           ! xx=XST+I*HX
         yy=(J+MEPY*JML)*HY           ! yy=YST+J*HY
         IF (I.GE.IST.AND.I.LE.IEND .AND. J.GE.JST.AND.J.LE.JEND) THEN
            U(I,J)  = 0.0             ! 近似解赋初值
            US(I,J) = solution(xx,yy) ! 解析解
            F(I,J)  = DD*rhs(xx,yy)   ! 右端项
         ELSE IF ((I.EQ.IST-1  .AND. MEPX.EQ.0) .OR.
     &            (J.EQ.JST-1  .AND. MEPY.EQ.0) .OR.
     &            (I.EQ.IEND+1 .AND. MEPX.EQ.NPX-1) .OR.
```

```
    &           (J.EQ.JEND+1 .AND. MEPY.EQ.NPY-1)) THEN
          U(I,J) = solution(xx,yy)  ! 边界值
        ENDIF
      ENDDO
      ENDDO
! Jacobi 迭代求解
      NITER=0
      T0 = MPI_Wtime()
100   CONTINUE
      NITER=NITER+1
! 交换定义在辅助网格结点上的近似解
      CALL MPI_Send(U(1,1),      1, VTYPE, MYLEFT,  NITER+100, !
     &              MPI_COMM_WORLD,IERR)                    ! 发送左边界
      CALL MPI_Send(U(IEND,1),   1, VTYPE, MYRIGHT, NITER+100,
     &              MPI_COMM_WORLD,IERR)                    ! 发送右边界
      CALL MPI_Send(U(1,1),      1, HTYPE, MYLOWER, NITER+100,
     &              MPI_COMM_WORLD,IERR)                    ! 发送下边界
      CALL MPI_Send(U(1,JEND),   1, HTYPE, MYUPPER, NITER+100,
     &              MPI_COMM_WORLD,IERR)                    ! 发送上边界
      CALL MPI_Recv(U(IEND+1,1), 1, VTYPE, MYRIGHT, NITER+100,
     &              MPI_COMM_WORLD,STATUS,IERR)             ! 接收右边界
      CALL MPI_Recv(U(0,1),      1, VTYPE, MYLEFT,  NITER+100,
     &              MPI_COMM_WORLD, STATUS,IERR)            ! 接收左边界
      CALL MPI_Recv(U(1,JEND+1), 1, HTYPE, MYUPPER, NITER+100,
     &              MPI_COMM_WORLD, STATUS,IERR)            ! 接收上边界
      CALL MPI_Recv(U(1,0),      1, HTYPE, MYLOWER, NITER+100,
     &              MPI_COMM_WORLD, STATUS, IERR)           ! 接收下边界
      DO J=JST,JEND     !
      DO I=IST,IEND
        U0(I,J)=F(I,J)+DX*(U(I,J-1)+U(I,J+1))+DY*(U(I-1,J)+U(I+1,J))
      ENDDO
      ENDDO             !
! 计算与精确解间的误差
      ERR=0.0
      DO J=JST,JEND
      DO I=IST,IEND
```

```
126         U(I,J)=U0(I,J)
127         ERR=MAX(ERR, ABS(U(I,J)-US(I,J))) ! 用L^∞模以使误差与NP无关
128       ENDDO
129       ENDDO
130       ERR0=ERR
131       CALL MPI_Allreduce(ERR0,ERR,1,MPI_REAL,MPI_MAX,
132      &                   MPI_COMM_WORLD,IERR)
133       IF (MYRANK.EQ.0 .AND. MOD(NITER,100).EQ.0) THEN
134         PRINT *, 'NITER = ', NITER, '    ERR = ', ERR
135       ENDIF
136       IF (ERR.GT.1.E-3)  THEN     ! 收敛性判断
137         GOTO 100                  ! 没有收敛, 进入下次迭代
138       ENDIF
139       T1 = MPI_Wtime()
140       IF (MYRANK.EQ.0) THEN
141         PRINT *, ' !!! Successfully converged after ',
142      &           NITER, ' iterations'
143         PRINT *, ' !!! error = ', ERR, '   wtime = ', T1 - T0
144       ENDIF
145 ! 输出近似解 (略)
146       CALL MPI_Finalize(IERR)
147       END
```

上例中固定了该 MPI 程序产生的进程个数 NP = NPY*NPX，这样做主要是为了方便将各进程的数组大小声明成仅为相应串行程序的 1/(NPY*NPX)，从而使得原来串行程序在单机上由于内存不够而无法计算的问题通过多机并行计算成为可能[1]。当然，这样做也带来一些不便，例如它要求 MPI 运行命令 “mpirun -np xxx” 中的参数 xxx 等于 NPY*NPX，并且参数 NPY 和 NPX 被改变后，必须重新编译该程序。

[1]Fortran 77 中没有动态内存分配功能，习题 1 中提供了一些克服这一限制的方法。

8.3 并行效率分析

本节对 Jacobi 迭代的计算量、通信量进行统计，分析算法的并行效率。

首先统计程序的浮点计算时间。代码 8.1 的主要计算量是 117–121 行的 Jacobi 迭代循环，每步循环需要 6 次浮点计算 (4 次加法，2 次乘法，其中可能的优化可以参看习题 4)。此外，在计算近似解误差的循环中，平均每个网格点需要 3 次浮点计算 (1 次浮点减法，1 次取绝对值，1 次取 max)。因此，每个进程中每步迭代的总浮点计算量近似等于：

$$9 \times \texttt{IML} \times \texttt{JML}$$

假设每个处理机上运行一个进程，而处理机完成一次浮点运算的时间为 T_0，则每个进程的总计算时间为：

$$T_{\text{cpu}} \approx 9 \times T_0 \times \texttt{IML} \times \texttt{JML}$$

下面统计程序的通信时间。每步迭代中的通信分两部分，第一部分通信是相邻子区域间交换辅助网格点上的近似解。为简单起见，假设一次通信的时间包括通信开销 (延迟) 和数据传输时间 (带宽)，并且发送数据和接收数据的时间一样，其中通信开销只依赖于通信次数，数据传输时间只依赖于收发的数据量，则各进程中每步迭代总通信时间的最大者为 (假设总进程数 p 大于 2)：

$$T_{\texttt{comm}} + T_{\texttt{lat}} \approx \begin{cases} 4 \times T_1 \times \texttt{JM} + 4 \times T_2, & \texttt{NPY} = 1 \\ 4 \times T_1 \times \texttt{IM} + 4 \times T_2, & \texttt{NPX} = 1 \\ 4 \times T_1 \times (\texttt{IML} + \texttt{JML}) + 8 \times T_2, & \texttt{NPX} > 1, \texttt{NPY} > 1 \end{cases} \tag{8.7}$$

其中 T_1 为传送一个浮点数需要的平均时间 (通信带宽的倒数)，T_2 为一次通信的开销 (通信延迟)。上式表明，如果采用一维条划分的

区域划分方式，每个进程的通信量是一个与进程数无关的常数；如果采用二维块划分的区域划分方式，则通信次数是一维条划分时的 2 倍，而通信量依赖于进程数，并且随进程数的增加而减少。不难看出，如果固定总的处理器数目和问题规模，则使得通信量最小的最优进程网格划分是使得 IML 和 JML 的值尽可能接近的划分，这就是通常所说的划分处理器网格时应该尽量使得子区域接近“正方形”。为简化讨论，这里仅考虑式 (8.7) 最后一种情况。

第二部分通信是计算全局误差时的归约操作。这类操作通常采用树型算法，所需要的时间为：

$$T_{\texttt{reduce}} = C \times \log p$$

其中 C 为常数，p 为总进程数。

由于 Jacobi 迭代中没有因数据相关而引起的空闲等待，因而并行程序的运行时间为：

$$\begin{aligned} T_{\text{并行}} &= T_{\text{cpu}} + T_{\text{comm}} + T_{\text{lat}} + T_{\text{reduce}} \\ &\approx 9 \times T_0 \times \texttt{IML} \times \texttt{JML} + 4 \times T_1 \times (\texttt{IML} + \texttt{JML}) \\ &\quad + 8 \times T_2 + C \times \log p \end{aligned}$$

显然，串行程序的运行时间为：

$$T_{\text{串行}} = 9 \times T_0 \times (\texttt{IM} - 1) \times (\texttt{JM} - 1) \approx 9 \times T_0 \times \texttt{IM} \times \texttt{JM}$$

因此，并行加速比为：

$$\begin{aligned} S &= T_{\text{串行}} / T_{\text{并行}} \\ &\approx \frac{9 \times T_0 \times \texttt{IM} \times \texttt{JM}}{9 \times T_0 \times \texttt{IML} \times \texttt{JML} + 4 \times T_1 \times (\texttt{IML} + \texttt{JML}) + 8 \times T_2 + C \times \log p} \end{aligned}$$

并行效率为 (注意 IML = IM/NPX，JML = JM/NPY)：

$$
\begin{aligned}
E &= S/p \\
&= \frac{9 \times T_0 \times \mathtt{IM} \times \mathtt{JM}}{9 \times T_0 \times \mathtt{IML} \times \mathtt{JML} + 4 \times T_1 \times (\mathtt{IML} + \mathtt{JML}) + 8 \times T_2 + C \times \log p} \\
&\quad \times \frac{1}{\mathtt{NPX} \times \mathtt{NPY}} \\
&= \frac{9 \times T_0 \times \mathtt{IML} \times \mathtt{JML}}{9 \times T_0 \times \mathtt{IML} \times \mathtt{JML} + 4 \times T_1 \times (\mathtt{IML} + \mathtt{JML}) + 8 \times T_2 + C \times \log p}
\end{aligned}
$$

上式中 T_0，T_1，T_2 和 C 均为常数。令：

$$
\delta = 4 \times T_1 \times \frac{\mathtt{IML} + \mathtt{JML}}{\mathtt{IML} \times \mathtt{JML}} + \frac{8 \times T_2 + C \times \log p}{\mathtt{IML} \times \mathtt{JML}} \tag{8.8}
$$

则：

$$
E = \frac{1}{1 + \delta/T_0} \tag{8.9}
$$

由于 IML + JML 相当于子区域边界上的网格点数的一半，IML × JML 相当于子区域内部的网格点数，因而通常称 (IML + JML)/(IML × JML) 为子区域的“面体比”[2]。显然面体比随子区域的增大而减小。从式 (8.8) 和式 (8.9) 易知，子区域的面体比越小，并行效率越高。因此，当进程数目固定时，并行效率将随子区域问题规模的增加而增加。

8.4 MPI 并行程序的改进

在代码 8.1 中，交换定义在辅助网格结点近似解的 8 条消息传递语句 (101–116 行) 是该 MPI 程序的关键。但是，由 MPI 标准可知，它们是不安全的，因为在某些并行机上，当消息较长时，可能由于 MPI 系统缓存区大小的限制，而导致执行该 MPI 程序的进程死

[2]名词“面体比”来源于三维问题，出于习惯对二维问题依然沿用这一叫法。

锁 (参看 196 页代码 3.1)。这里，可以将它们替换成如下的非阻塞通信函数。

代码 8.2: 改进一：非阻塞通信 (源程序见文件 poisson1.f)。

```
! 用下一行替换 poisson0.f 第 24 行
      INTEGER  REQ(8), STATUS(MPI_STATUS_SIZE,8)
      ... ... ... ... ... ... (略)
! 用下述内容替换 poisson0.f 101-116 行
      CALL MPI_Isend(U(1,1),      1, VTYPE, MYLEFT,  NITER+100,
     &               MPI_COMM_WORLD,REQ(1),IERR)           ! 发送左边界
      CALL MPI_Isend(U(IEND,1),   1, VTYPE, MYRIGHT, NITER+100,
     &               MPI_COMM_WORLD,REQ(2),IERR)           ! 发送右边界
      CALL MPI_Isend(U(1,1),      1, HTYPE, MYLOWER, NITER+100,
     &               MPI_COMM_WORLD,REQ(3),IERR)           ! 发送下边界
      CALL MPI_Isend(U(1,JEND),   1, HTYPE, MYUPPER, NITER+100,
     &               MPI_COMM_WORLD,REQ(4),IERR)           ! 发送上边界
      CALL MPI_Irecv(U(IEND+1,1), 1, VTYPE, MYRIGHT, NITER+100,
     &               MPI_COMM_WORLD, REQ(5),IERR)          ! 接收右边界
      CALL MPI_Irecv(U(0,1),      1, VTYPE, MYLEFT,  NITER+100,
     &               MPI_COMM_WORLD, REQ(6),IERR)          ! 接收左边界
      CALL MPI_Irecv(U(1,JEND+1), 1, HTYPE, MYUPPER, NITER+100,
     &               MPI_COMM_WORLD, REQ(7),IERR)          ! 接收上边界
      CALL MPI_Irecv(U(1,0),      1, HTYPE, MYLOWER, NITER+100,
     &               MPI_COMM_WORLD, REQ(8),IERR)          ! 接收下边界
      CALL MPI_Waitall(8,REQ,STATUS,IERR)     ! 阻塞式等待消息传递的结束
      ... ... ... ... ... ... (略)
```

在代码 8.1 中，将 117–121 行的循环分裂成两个部分，其中一个部分需要辅助网格点上的近似解，而另一个部分不需要辅助网格点上的近似解。这样，为了改进该 MPI 程序的并行性能，可以将后一个部分的计算与代码 8.2 的非阻塞消息传递重叠起来，从而达到屏蔽网络延迟的目的。具体改进如下。

代码 8.3: 改进二：重叠通信与计算 (源程序见文件 poisson2.f)。

```
     ... ... ... ... ... ... (略)
! 用下述内容替换 poisson1.f 118-122 行
      DO J=JST+1,JEND-1
      DO I=IST+1,IEND-1
         U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
     &                          +HY2*(U(I-1,J)+U(I+1,J)))
      ENDDO
      ENDDO
      CALL MPI_Waitall(8,REQ,STATUS,IERR)     ! 阻塞式等待消息传递的结束
      DO J=JST, JEND, JEND-JST
      DO I=IST, IEND
         U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
     &                          +HY2*(U(I-1,J)+U(I+1,J)))
      ENDDO
      ENDDO
      DO J=JST, JEND
      DO I=IST, IEND, IEND-IST
         U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
     &                          +HY2*(U(I-1,J)+U(I+1,J)))
      ENDDO
      ENDDO
     ... ... ... ... ... ... (略)
```

在代码 8.1–8.3 中，各进程按自然序 (先 x 后 y) 确定与它相邻的 4 个进程的进程号 (MYLEFT，MYRIGHT，MYLOWER，MYUPPER)，以及它自己所处的行主序位置 (MEPY，MEPX)。实际上，这些进程按区域分解策略可以很自然地映射到 NPY*NPX 的二维 Cartesian 拓扑结构 (参看 3.2.8)，而 (MEPY，MEPX) 就是各进程在该拓扑结构中的坐标。因此，可以从通信器 MPI_COMM_WORLD 出发，建立二维 Cartesian 拓扑结构，从而方便地确定各进程的相邻关系，并使得之后的所有 MPI 消息传递均基于该拓扑结构进行。

代码 8.4: 改进三：二维 Cartesian 拓扑结构 (源程序见文件 poisson3.f)。

```
      ... ... ... ... ... ... (略)
! 在 poisson[012].f 程序头(变量声明部分)加入下面二行
      INTEGER COMM, DIMS(2),COORD(2)
      LOGICAL PERIOD(2),REORDER
      ... ... ... ... ... ... (略)
! 用下述内容替换 poisson[012].f 38-49 行
      DIMS(1)=NPY            ! 拓扑结构中Y方向的进程个数
      DIMS(2)=NPX            ! 拓扑结构中X方向的进程个数
      PERIOD(1)=.FALSE.      ! 沿Y方向, 拓扑结构非周期连接
      PERIOD(2)=.FALSE.      ! 沿X方向, 拓扑结构非周期连接
      REORDER=.TRUE.         ! 在新通信器中, 允许进程重新排序
      CALL MPI_Cart_create(MPI_COMM_WORLD, 2, DIMS, PERIOD, REORDER,
     &                     COMM, IERR)
      CALL MPI_Comm_rank(COMM,MYRANK,IERR)
      CALL MPI_Cart_coords(COMM,MYRANK,2,COORD,IERR)
      MEPY=COORD(1)
      MEPX=COORD(2)
      CALL MPI_Cart_shift(COMM, 0, 1, MYLOWER, MYUPPER, IERR)   ! Y方向
      CALL MPI_Cart_shift(COMM, 1, 1, MYLEFT,  MYRIGHT, IERR)   ! X方向
      ... ... ... ... ... ... (略)
```

代码 8.1–8.4 中忽略了近似解的输出。这里采用 3.2.9 中介绍的 MPI 并行 I/O 函数实现近似解的并行输出，要求输出的近似解按自然序排列，且包含物理边界结点。

代码 8.5: 改进四：并行 I/O (源程序见文件 poisson4)。该程序使用了独立文件指针聚合型输出函数 `MPI_File_write_all` (参看表 3.3)。

```
      ... ... ... ... ... ... (略)
! 在程序头(变量声明部分)加入下面内容
      INTEGER FH, FILETYPE, MEMTYPE, GSIZE(2), LSIZE(2), START(2)
! 注意: 从下面三种形式的变量声明中根据所使用的MPI系统选择一个正确的
```

```
! (可以参考文件 mpiof.h 或 mpif.h 中 MPI_OFFSET_KIND 的定义)
!     INTEGER(kind=MPI_OFFSET_KIND) OFFSET    ! 适用于 Fortran 90/95
!     INTEGER*8 OFFSET                        ! 适用于 64 位系统
      INTEGER*4 OFFSET                        ! 适用于某些 32 位系统
      ... ... ... ... ... ... (略)
! 在标有“输出近似解(略)”处 (倒数第三行) 加入下述内容
      GSIZE(1)=IM+1
      GSIZE(2)=JM+1
      LSIZE(1)=IEND-IST+1
      IF (MEPX.EQ.0) LSIZE(1)=LSIZE(1)+1
      IF (MEPX.EQ.NPX-1) LSIZE(1)=LSIZE(1)+1
      LSIZE(2)=JEND-JST+1
      IF (MEPY.EQ.0) LSIZE(2)=LSIZE(2)+1
      IF (MEPY.EQ.NPY-1) LSIZE(2)=LSIZE(2)+1
      START(1)=IML*MEPX
      IF (MEPX.NE.0) START(1)=START(1)+1
      START(2)=JML*MEPY
      IF (MEPY.NE.0) START(2)=START(2)+1
!  定义局部子数组数据类型
      CALL MPI_Type_create_subarray(2, GSIZE, LSIZE, START,
     &          MPI_ORDER_FORTRAN, MPI_REAL, FILETYPE, IERR)
      CALL MPI_Type_commit(FILETYPE, IERR)
! 打开二进制文件
      CALL MPI_File_open(COMM,'result.dat',
     &          MPI_MODE_CREATE + MPI_MODE_WRONLY,
     &          MPI_INFO_NULL, FH, IERR)
      OFFSET=0          ! 注意使用正确的变量类型 (INTEGER*4 或 INTEGER*8)
                        ! (参考文件 mpif.h 中 MPI_OFFSET_KIND 的定义)
      CALL MPI_File_set_view(FH, OFFSET, MPI_REAL, FILETYPE,
     &          'native', MPI_INFO_NULL, IERR)
! 定义数据类型, 描述子数组在内存中的分布
      GSIZE(1)=IML+2
      GSIZE(2)=JML+2
      START(1)=1
      IF (MEPX.EQ.0) START(1)=0
      START(2)=1
```

```
      IF (MEPY.EQ.0) START(2)=0
      CALL MPI_Type_create_subarray(2, GSIZE, LSIZE, START,
     &          MPI_ORDER_FORTRAN, MPI_REAL, MEMTYPE, IERR)
      CALL MPI_Type_commit(MEMTYPE, IERR)
! 输出近似解(含物理边界结点)
      CALL MPI_File_write_all(FH, U, 1, MEMTYPE, STATUS, IERR)
      CALL MPI_File_close(FH, IERR)
      ... ... ... ... ... ... (略)
```

至此，代码 8.2–8.5 分别从非阻塞通信、重叠通信与计算、拓扑结构和并行 I/O 四个方面，利用相应的 MPI 函数，依次改进了代码 8.1 中 MPI 程序的功能和并行性能。通过该应用示例，读者可以较好地将所介绍的 MPI 函数联系在一起，解决实际问题。

习　题

1. 试用下面两种方式修改代码 8.1，使得改变进程数目或问题规模后不必重新编译：

(1) 根据结点机的物理内存容量固定各个数组大小；

(2) 利用一个 C 语言编写的函数动态分配内存。

2. 代码 8.1 中使用了 L^∞ 模计算近似解的误差，从并行计算和舍入误差的角度讲使用 L^∞ 模和 L^2 模有什么区别?

3. 修改代码 8.1，使其能够处理 NPX 不是 IM 的倍数或 NPY 不是 JM 的倍数的情况。

4. 代码 8.1 第 117–121 行的 Jaboci 迭代循环中，假设 $h_x = h_y$ (即 DX = DY)，试改写代码，将平均每个网格点的浮点工作量降为 1 个乘法、3 个加法。

5. 使用不同处理机数目、区域划分及问题规模运行代码 8.1，计算相应的并行效率和加速比，并分析所得到的性能结果。

6. 通过将代码 8.1 中的 `MPI_Send` 和 `MPI_Recv` 进行适当配对，然后用 `MPI_Sendrecv` 代替是否可以避免通信死锁？用这样的方式避免通信死锁与代码 8.2 相比有何优劣？

7. 在代码 8.1–8.5 中用了一排辅助网格单元。通过增加虚拟网格点的宽度可以提高通信粒度。例如，如果使用两排辅助网格单元，则可以每两次迭代交换一次子区域边界附近的近似解，代价是子区域间增加了少量的重复计算。试修改代码 8.4 或代码 8.5 中的程序，在程序中增加一个参数 `BW`，它代表辅助网格单元的宽度 (`BW` $\geqslant 1$)，比较不同 `BW` 的值对程序性能的影响。

8. 修改代码 8.5，将 Jacobi 迭代改为红黑顺序的 Gauss–Seidel 迭代。

9. 统计 Jacobi 迭代的浮点运算次数，修改代码 8.1–8.4，在程序结束时打印出实际达到的 Mflops 值，并根据处理机的峰值性能计算程序的实际效率。

10. 在本章程序的基础上，设计三维 Poisson 方程 Jacobi 迭代求解的 MPI 并行程序。

第 9 章　二维热传导方程

本章中以矩形区域上的二维热传导方程为例，介绍并行算法设计中另一类重要方法：流水线方法，特别是如何通过分块技术达到并行度与通信粒度之间的有效平衡。流水线方法在计算机体系结构设计、优化编译等领域是一项非常重要的技术，是现代计算机系统设计中提高处理能力的最主要的手段之一。在许多问题的并行算法设计中，它同样扮演着重要的角色。

考虑长方形区域 $\Omega = (0, W) \times (0, H)$ 上的二维热传导方程:

$$\begin{cases} \dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2}, & (x, y) \in \Omega, \quad t > 0 \\ u = u^0, & t = 0 \\ u = g, & (x, y) \in \partial\Omega, \quad t \geqslant 0 \end{cases} \tag{9.1}$$

其中，$u = u(x, y, t)$ 为未知函数，$u^0 = u^0(x, y)$ 和 $g = g(x, y, t)$ 为已知函数，分别定义在区域 Ω 的内部和边界。

9.1　空间离散与区域划分

方程 (9.1) 的空间离散与第 8 章 Poisson 方程一样。沿坐标轴 x 和 y 方向，分别取步长

$$h_x = \frac{W}{\texttt{IM}}, \quad h_y = \frac{H}{\texttt{JM}} \tag{9.2}$$

将区域 Ω 离散成规模为 $\texttt{IM} \times \texttt{JM}$ 的网格，其中 IM 和 JM 分别为沿坐标轴 x 和 y 方向的网格单元个数。假设所有函数均定义在网格结点上，并对任意函数 f，用 $f_{i,j}$ 表示 $f(ih_x, jh_y)$。用二阶中心差商近似

空间导数后，得到下述常微分方程组：

$$\begin{cases} \dfrac{\mathrm{d}u_{i,j}}{\mathrm{d}t} = -\dfrac{2u_{i,j} - u_{i-1,j} - u_{i+1,j}}{h_x^2} - \dfrac{2u_{i,j} - u_{i,j-1} - u_{i,j+1}}{h_y^2}, \\ u_{i,j}|_{t=0} = u_{i,j}^0, \quad 1 \leqslant i \leqslant \mathtt{IM}-1, \quad 1 \leqslant j \leqslant \mathtt{JM}-1 \\ u_{i,j} = g_{i,j}, \quad i = 0 \text{ 或 } i = \mathtt{IM} \text{ 或 } j = 0 \text{ 或 } j = \mathtt{JM} \end{cases} \tag{9.3}$$

在非定常问题的并行算法设计中，由于时间方向上的数据相关性，任务划分通常在空间方向进行。这里采用与第 8 章 Poisson 方程中完全一样的子区域划分方式：假设使用 p 个进程并行计算，则将计算区域沿两个空间方向分成 $\mathtt{NPX}\times\mathtt{NPY}$ 个子区域，其中 $\mathtt{NPX}\times\mathtt{NPY} = p$，所采用的数据结构和通信过程也完全类似，详参看 8.2。

9.2 时间离散：显式格式

选取时间步长 $\Delta t > 0$，并记 $u_{i,j}^n = u_{i,j}|_{t=n\Delta t}$，$g_{i,j}^n = g_{i,j}|_{t=n\Delta t}$。用一阶向前 Euler 格式离散方程 (9.3) 得到：

$$\begin{cases} \dfrac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} = -\dfrac{2u_{i,j}^n - u_{i-1,j}^n - u_{i+1,j}^n}{h_x^2} - \dfrac{2u_{i,j}^n - u_{i,j-1}^n - u_{i,j+1}^n}{h_y^2}, \\ \qquad 1 \leqslant i \leqslant \mathtt{IM}-1, \quad 1 \leqslant j \leqslant \mathtt{JM}-1, \quad n \geqslant 0 \\ u_{i,j}^{n+1} = g_{i,j}^{n+1}, \ \ i = 0 \text{ 或 } i = \mathtt{IM} \text{ 或 } j = 0 \text{ 或 } j = \mathtt{JM}, \ \ n \geqslant 0 \end{cases} \tag{9.4}$$

格式 (9.4) 的稳定性条件为 $\dfrac{\Delta t}{h_x^2} + \dfrac{\Delta t}{h_y^2} < \dfrac{1}{2}$ [67]。

容易看出，方程 (9.4) 的计算过程与求解 Poisson 方程的 Jacobi 迭代过程是完全类似的。只要将第 8 章中求解 Poisson 方程的程序稍加改动，便可用于计算显式格式离散的热传导方程。

代码 9.1：规则区域上的二维热传导方程：显式 Euler 格式。

文件名：`code/heat/heat1.f`

```
! 二维热传导方程：显式Euler格式 (基于莫则尧的 poisson1.f)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      INCLUDE  'mpif.h'
      PARAMETER(DW=2.D0,DH=3.D0)  ! 问题求解区域沿 X、Y 方向的大小
      PARAMETER(DT=.0008D0)       ! 时间步长，要求满足：Δt(1/h_x^2 + 1/h_y^2) < 1/2
      PARAMETER(IM=30, JM=60)     ! 沿 X、Y 方向的全局网格规模
      PARAMETER(NPX=1, NPY=1)     ! 沿 X、Y 方向的进程个数
      PARAMETER(IML=IM/NPX, JML=JM/NPY)
        ! 各进程沿 X、Y 方向的局部网格规模，仅为全局网格规模的 1/(NPX*NPY)
      DIMENSION U (0:IML+1,0:JML+1) ! 当前时间层的近似解
      DIMENSION U0(0:IML+1,0:JML+1) ! 前一时间层的近似解
      DOUBLE PRECISION KX, KY     ! Δt/h_x^2 和 Δt/h_y^2
      DOUBLE PRECISION T0, T1     ! 用于统计运行时间
      INTEGER  NPROC              ! mpirun 启动的进程个数，必须等于 NPX*NPY
      INTEGER  MYRANK, MYLEFT, MYRIGHT, MYUPPER, MYLOWER
                                  ! 各进程自身的进程号，4 个相邻进程的进程号
      INTEGER  MEPX,MEPY          ! 各进程自身的进程号沿 X、Y 方向的坐标
      INTEGER  IST,IEND,JST,JEND
                     ! 各进程沿 X、Y 方向的内部网格结点的起始和终止坐标
      INTEGER  HTYPE, VTYPE
                     ! MPI 用户自定义数据类型，表示各进程沿 X、Y 方向
                     ! 与相邻进程交换的数据单元
      INTEGER  REQ(8), STATUS(MPI_STATUS_SIZE,8)
! Constants
      DATA TWO/2.D0/, ZERO/0.D0/
! In-line functions
      solution(x,y,t) = EXP(-t-t)*SIN(x)*COS(y) ! 解析解：e^{-2t} sin x cos y
! 程序可执行语句开始
      CALL MPI_Init(IERR)
      CALL MPI_Comm_size(MPI_COMM_WORLD,NPROC,IERR)
      IF (NPROC.NE.NPX*NPY.OR.MOD(IM,NPX).NE.0.OR.MOD(JM,NPY).NE.0) THEN
         PRINT *, '+++ Incorrect parameters, abort +++'
```

```
33          CALL MPI_Finalize(IERR)
34          STOP
35        ENDIF
36  ! 按自然序 (先沿 X 方向, 后沿 Y 方向) 确定各进程自身及其 4 个相邻进程的进程号
37        CALL MPI_Comm_rank(MPI_COMM_WORLD,MYRANK,IERR)
38        MYLEFT  = MYRANK - 1
39        IF (MOD(MYRANK,NPX).EQ.0)   MYLEFT=MPI_PROC_NULL
40        MYRIGHT = MYRANK + 1
41        IF (MOD(MYRIGHT,NPX).EQ.0)  MYRIGHT=MPI_PROC_NULL
42        MYUPPER = MYRANK + NPX
43        IF (MYUPPER.GE.NPROC)       MYUPPER=MPI_PROC_NULL
44        MYLOWER = MYRANK - NPX
45        IF (MYLOWER.LT.0)           MYLOWER=MPI_PROC_NULL
46        MEPY=MYRANK/NPX
47        MEPX=MYRANK-MEPY*NPX
48  ! 基本变量赋值, 确定各进程负责的子区域
49        HX = DW/IM                  ! X 方向网格步长 h_x
50        KX = DT/(HX*HX)             ! Δt/h_x^2
51        HY = DH/JM                  ! Y 方向网格步长 h_y
52        KY = DT/(HY*HY)             ! Δt/h_y^2
53  ! 各子区域负责计算的范围
54        IST=1
55        IEND=IML
56        IF (MEPX.EQ.NPX-1) IEND=IEND-1   ! 最右边的区域 X 方向少一个点
57        JST=1
58        JEND=JML
59        IF (MEPY.EQ.NPY-1) JEND=JEND-1   ! 最上边的区域 Y 方向少一个点
60  ! 初始条件
61        DO J=JST-1, JEND+1
62          yy=(J+MEPY*JML)*HY
63          DO I=IST-1, IEND+1
64            xx=(I+MEPX*IML)*HX
65            U(I,J)=solution(xx,yy,ZERO) ! 初始解
66          ENDDO
67        ENDDO
68  ! 数据类型定义
```

```
      CALL MPI_Type_contiguous(IEND-IST+1, MPI_DOUBLE_PRECISION,
     &                         HTYPE, IERR)
      CALL MPI_Type_commit(HTYPE, IERR)
                ! 沿 X 方向的连续 IEND-IST+1 个 MPI_DOUBLE_PRECISION 数据单元,
                ! 可用于表示该进程与其上、下进程交换的数据单元
      CALL MPI_Type_vector(JEND-JST+1, 1, IML+2, MPI_DOUBLE_PRECISION,
     &                     VTYPE, IERR)
      CALL MPI_Type_commit(VTYPE, IERR)
                ! 沿 Y 方向的连续 JEND-JST+1 个 MPI_DOUBLE_PRECISION 数据单元,
                ! 可用于表示该进程与其左、右进程交换的数据单元
! 时间推进
      NT=0
      T0 = MPI_Wtime()
100   CONTINUE   ! 主循环
      NT=NT+1
      T=NT*DT
! 拷贝 U -> U0
      DO J=JST-1,JEND+1
      DO I=IST-1,IEND+1
         U0(I,J)=U(I,J)
      ENDDO
      ENDDO
! 边界条件
      IF (MEPX.EQ.0) THEN
         xx = ZERO
         DO J=JST,JEND
            yy=(J+MEPY*JML)*HY
            U(0,J)=solution(xx,yy,T)
         ENDDO
      ENDIF
      IF (MEPX.EQ.NPX-1) THEN
         xx = DW
         DO J=JST,JEND
            yy=(J+MEPY*JML)*HY
            U(IEND+1,J)=solution(xx,yy,T)
         ENDDO
```

```
105       ENDIF
106       IF (MEPY.EQ.0) THEN
107          yy = ZERO
108          DO I=IST,IEND
109             xx=(I+MEPX*IML)*HX
110             U(I,0)=solution(xx,yy,T)
111          ENDDO
112       ENDIF
113       IF (MEPY.EQ.NPY-1) THEN
114          yy = DH
115          DO I=IST,IEND
116             xx=(I+MEPX*IML)*HX
117             U(I,JEND+1)=solution(xx,yy,T)
118          ENDDO
119       ENDIF
120 ! 显式Euler格式推进
121       DO J=JST,JEND
122       DO I=IST,IEND
123          U(I,J)=U0(I,J)
124      &             - KX * (TWO*U0(I,J) - U0(I-1,J) - U0(I+1,J))
125      &             - KY * (TWO*U0(I,J) - U0(I,J-1) - U0(I,J+1))
126       ENDDO
127       ENDDO
128 ! 交换定义在辅助网格结点上的近似解
129       CALL MPI_Isend(U(1,1),      1, VTYPE, MYLEFT,  NT+100,
130      &               MPI_COMM_WORLD,REQ(1),IERR)          ! 发送左边界
131       CALL MPI_Isend(U(IEND,1),   1, VTYPE, MYRIGHT, NT+100,
132      &               MPI_COMM_WORLD,REQ(2),IERR)          ! 发送右边界
133       CALL MPI_Isend(U(1,1),      1, HTYPE, MYLOWER, NT+100,
134      &               MPI_COMM_WORLD,REQ(3),IERR)          ! 发送下边界
135       CALL MPI_Isend(U(1,JEND),   1, HTYPE, MYUPPER, NT+100,
136      &               MPI_COMM_WORLD,REQ(4),IERR)          ! 发送上边界
137       CALL MPI_Irecv(U(IEND+1,1), 1, VTYPE, MYRIGHT, NT+100,
138      &               MPI_COMM_WORLD, REQ(5),IERR)         ! 接收右边界
139       CALL MPI_Irecv(U(0,1),      1, VTYPE, MYLEFT,  NT+100,
140      &               MPI_COMM_WORLD, REQ(6),IERR)         ! 接收左边界
```

```
141       CALL MPI_Irecv(U(1,JEND+1), 1, HTYPE, MYUPPER, NT+100,
142      &               MPI_COMM_WORLD, REQ(7),IERR)          ! 接收上边界
143       CALL MPI_Irecv(U(1,0),      1, HTYPE, MYLOWER, NT+100,
144      &               MPI_COMM_WORLD, REQ(8),IERR)          ! 接收下边界
145       CALL MPI_Waitall(8,REQ,STATUS,IERR)     ! 阻塞式等待消息传递的结束
146       T1 = MPI_Wtime()
147       IF (MYRANK.EQ.0) PRINT *, 'T=', T, '   wtime=', T1 - T0
148       IF (T.LT.1.0) GOTO 100
149  ! 计算与精确解间的误差
150       ERR0=ZERO
151       DO J=JST, JEND
152          yy=(J+MEPY*JML)*HY
153          DO I=IST, IEND
154             xx=(I+MEPX*IML)*HX
155             ERR0=MAX(ERR0,ABS(U(I,J)-solution(xx,yy,T)))
156          ENDDO
157       ENDDO
158       CALL MPI_Reduce(ERR0, ERR, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0,
159      &                MPI_COMM_WORLD, IERR)
160       IF (MYRANK.EQ.0) THEN
161          PRINT *, 'Error: ', ERR
162          PRINT *, 'Wall time: ', T1 - T0
163       ENDIF
164       CALL MPI_Finalize(IERR)
165       STOP
166       END
```

9.3 时间离散：隐式/半隐式格式

由于计算稳定性的要求，采用显式格式计算对时间步长有着严格的限制，因而实际计算中更经常使用的是隐式格式。例如，采用一

阶向后 Euler 格式离散方程 (9.3) 得到下面的方程：

$$\begin{cases} \dfrac{u_{i,j}^{n+1}-u_{i,j}^{n}}{\Delta t} = -\dfrac{2u_{i,j}^{n+1}-u_{i-1,j}^{n+1}-u_{i+1,j}^{n+1}}{h_x^2} \\ \qquad\qquad -\dfrac{2u_{i,j}^{n+1}-u_{i,j-1}^{n+1}-u_{i,j+1}^{n+1}}{h_y^2}, \\ \qquad 1\leqslant i\leqslant \mathtt{IM}-1,\quad 1\leqslant j\leqslant \mathtt{JM}-1,\quad n\geqslant 0 \\ u_{i,j}^{n+1}=g_{i,j}^{n+1},\ i=0 \text{ 或 } i=\mathtt{IM} \text{ 或 } j=0 \text{ 或 } j=\mathtt{JM},\ n\geqslant 0 \end{cases} \tag{9.5}$$

采用隐式格式计算时，每个时间步需要解一个关于 $u_{i,j}^{n+1}$ 的线性代数方程组，可以采用某种迭代法并结合预条件技术求解。对于这种情况，最好选用一个现有的并行解法器包，例如 PETSc。另一个选择是使用 PETSc 的时间步进器 TS，441 页“一维热传导方程的求解”中给出了一个一维问题的程序实例，这个例子不难推广到本章所讨论的二维乃至更高维的问题。

另外一类处理隐式格式并行计算问题的方法是由周毓麟等发展的并行差分格式，它们通过将显隐格式相结合，即在子区域内部采用隐式格式、而在子区域边界附近采用显式格式的方法来改善计算格式的稳定性以便使用大的时间步长计算，同时避免线性方程组并行求解的问题。有关这方面的工作可参看 [5]。

9.4 时间离散：ADI 方法

交替方向隐式方法 (Alternating Direction Implicit，简称 ADI 方法) 通过对算子进行分裂，可以将高维问题的计算转化为一组一维

问题的计算。以 Peaceman–Rachford 格式为例 [22]：

$$\frac{\tilde{u}_{i,j}^{n+\frac{1}{2}}-u_{i,j}^{n}}{\Delta t}=\frac{1}{2}\left(\frac{-2\tilde{u}_{i,j}^{n+\frac{1}{2}}+\tilde{u}_{i-1,j}^{n+\frac{1}{2}}+\tilde{u}_{i+1,j}^{n+\frac{1}{2}}}{h_x^2}+\frac{-2u_{i,j}^{n}+u_{i,j-1}^{n}+u_{i,j+1}^{n}}{h_y^2}\right) \tag{9.6a}$$

$$\frac{u_{i,j}^{n+1}-\tilde{u}_{i,j}^{n+\frac{1}{2}}}{\Delta t}=\frac{1}{2}\left(\frac{-2u_{i,j}^{n+1}+u_{i,j-1}^{n+1}+u_{i,j+1}^{n+1}}{h_y^2}+\frac{-2\tilde{u}_{i,j}^{n+\frac{1}{2}}+\tilde{u}_{i-1,j}^{n+\frac{1}{2}}+\tilde{u}_{i+1,j}^{n+\frac{1}{2}}}{h_x^2}\right) \tag{9.6b}$$

它具有二阶精度，并且是无条件稳定的。

格式 (9.6) 中，每个时间步的计算分两步进行，第一步通过方程 (9.6a) 计算出一个中间解 $\tilde{u}^{n+\frac{1}{2}}$，第二步通过方程 (9.6b) 计算出新时间层的解 u^{n+1}。第一步计算中需要求解一族沿 x 方向的三对角线性方程组，而在第二步计算中，则需要求解一族沿 y 方向的三对角线性方程组。在串行计算机上，格式 (9.6) 是非常有效的，因为三对角线性方程组的求解很容易高效地用追赶法 [49, 67] 实现。但在这里，由于计算区域在空间方向被分解成若干个子区域，相应地三对角线性方程组的解向量及右端项分布在不同进程中，因此需要考虑如何在多个进程中并行求解这些三对角线性方程组。如果直接对这些三对角线性方程组用追赶法求解，则计算过程在这些进程中是串行进行的，每个进程只有等它前面的进程中的计算完成后才能开始计算。多年来，虽然已经发展了许多三对角线性方程组的并行解法 (参看 6.4)，但这些方法的计算量一般大于追赶法，从而降低了计算效率。另一个做法是交替地对数据重新进行分布，在求解方程 (9.6a) 时只在 y 方向划分子区域 (即取 `NPX` $=1$，`NPY` $=p$) 而在求解方程 (9.6b) 时则只在 x 方向划分子区域 (即取 `NPX` $=p$，`NPY` $=1$)，这样在每个空间方向上求解三对角线性方程组时可以直接采用串行

算法，它的缺点是计算过程中需要对整个数据场反复在进程间进行复杂的迁移，产生大数据量的通信，对并行系统的通信带宽要求很高。

9.5 分块流水线方法

本节介绍如何采用流水线方法 [23] 求解方程 (9.6a) 和 (9.6b)，并利用分块技术来平衡算法的并行度与通信粒度。在目前的分布式内存并行计算机上，分块流水线方法是解决该类数据相关性问题非常有效的方法，其思想亦适用于许多其他问题的并行算法设计。

9.5.1 模型问题

为便于算法的描述，采用如下形式的递推关系计算做为模型问题：

$$a_{0,j} \text{ 给定}, \quad a_{i,j} := F(a_{i-1,j}), \quad i = 1, 2, \ldots, n, \quad j = 1, \ldots, m \tag{9.7}$$

上述计算中，沿 j 方向的计算是互相独立的，而沿 i 方向的计算则存在着向前依赖关系 (递推关系)，无法独立进行，它对应于求解方程 (9.6a) 或 (9.6b) 的一次追赶或回代过程。事实上，这里所描述的算法当 j 方向上存在某些依赖关系 (例如自然顺序 Gauss–Seidel 迭代) 时仍然是有效的。

假设使用 p 个进程计算公式 (9.7)。如果数据的划分仅沿 j 方向进行，则计算是完全并行的。不失一般性，假设数据划分仅沿 i 方向进行，即假设 $a_{i,j}$ 被分成 p 段，$\{a_{i,j} \,|\, i = n_k, n_k+1, \ldots, n_{k+1}-1,\, j = 1, \ldots, m\}$ 存储在进程 p_k 中，$k = 0, \ldots, p-1$，这里 $1 = n_0 < n_1 < \cdots < n_p = n+1$。

在给定上述数据划分的情况下，式 (9.7) 的计算表面上看似乎难以并行，因为每个进程必须得到前一个进程的计算结果后才能开始

计算。但事实上，只要适当安排公式 (9.7) 中的通信与计算顺序，计算过程便可以在 p 个进程中以流水线的方式并发进行，这就是流水线算法，它由算法 9.1 给出。

算法 9.1: 计算递推关系的流水线算法。

```
do j = 1, m
   if (k>0) 从 p_{k-1} 接收 a(n_k-1, j)
   do i = n_k , n_{k+1}-1
      a(i,j) = F(a(i-1,j))
   enddo
   if (k<p-1) 发送 a(n_{k+1}-1, j) 至 p_{k+1}
enddo
```

其中 k 代表当前进程号。这里采用伪 Fortran 语言对算法进行描述，以便于与程序实例对应。

为了便于理解算法 9.1，表 9.1 和图 9.1 给出了当 $p = 3$ 时的计算过程。这里，为描述简单起见，假设进程间任务的划分是均衡的，即每个进程计算一次内层循环所需要的时间是一样的，该时间即为表中的一拍，并且所有进程的工作量也是一样的，即 $n_1 - n_0 = n_2 - n_1 = n_3 - n_2$，$p_0$、$p_1$ 和 p_2 代表三个进程。该算法可以被看成数据流水线，流水线长度为 p，数据长度为 m，因而完成全部计算需要的时间为 $m + p - 1$ 拍，它的并行度好并且实现起来非常简单，缺点是通信粒度太小，通信延迟对并行效率有很大的影响，因而实际中采用的不多。

利用分块技术可以有效克服流水线方法中通信粒度小、通信次数多的缺点。只要对算法 9.1 循环的顺序稍加改动，将 m 个递推关系式分成组，各进程每次处理完一组递推关系式的计算后再进行通信，便得到了分块流水线算法，即算法 9.2。

算法 9.2: 计算递推关系的分块流水线算法。

表 9.1 递推关系式的流水线计算流程

	p_0	p_1	p_2
第 1 拍	计算 $a_{i,1} := F(a_{i-1,1})$，$i = 1, \ldots, n_1 - 1$，发送 $a_{n_1-1,1}$ 给 p_1	等待接收 $a_{n_1-1,1}$	等待接收 $a_{n_2-1,1}$
第 2 拍	计算 $a_{i,2} := F(a_{i-1,2})$，$i = 1, \ldots, n_1 - 1$，发送 $a_{n_1-1,2}$ 给 p_1	计算 $a_{i,1} := F(a_{i-1,1})$，$i = n_1, \ldots, n_2 - 1$，发送 $a_{n_2-1,1}$ 给 p_2	等待接收 $a_{n_2-1,1}$
第 3 拍	计算 $a_{i,3} := F(a_{i-1,3})$，$i = 1, \ldots, n_1 - 1$，发送 $a_{n_1-1,3}$ 给 p_1	计算 $a_{i,2} := F(a_{i-1,2})$，$i = n_1, \ldots, n_2 - 1$，发送 $a_{n_2-1,2}$ 给 p_2	计算 $a_{i,1} := F(a_{i-1,1})$，$i = n_2, \ldots, n$
⋮	⋮	⋮	⋮
第 m 拍	计算 $a_{i,m} := F(a_{i-1,m})$，$i = 1, \ldots, n_1 - 1$，发送 $a_{n_1-1,m}$ 给 p_1	计算 $a_{i,m-1} := F(a_{i-1,m-1})$，$i = n_1, \ldots, n_2 - 1$，发送 $a_{n_2-1,m-1}$ 给 p_2	计算 $a_{i,m-2} := F(a_{i-1,m-2})$，$i = n_2, \ldots, n$
第 $m+1$ 拍	空闲	计算 $a_{i,m} := F(a_{i-1,m})$，$i = n_1, \ldots, n_2 - 1$，发送 $a_{n_2-1,m}$ 给 p_2	计算 $a_{i,m-1} := F(a_{i-1,m-1})$，$i = n_2, \ldots, n$
第 $m+2$ 拍	空闲	空闲	计算 $a_{i,m} := F(a_{i-1,m})$，$i = n_2, \ldots, n$

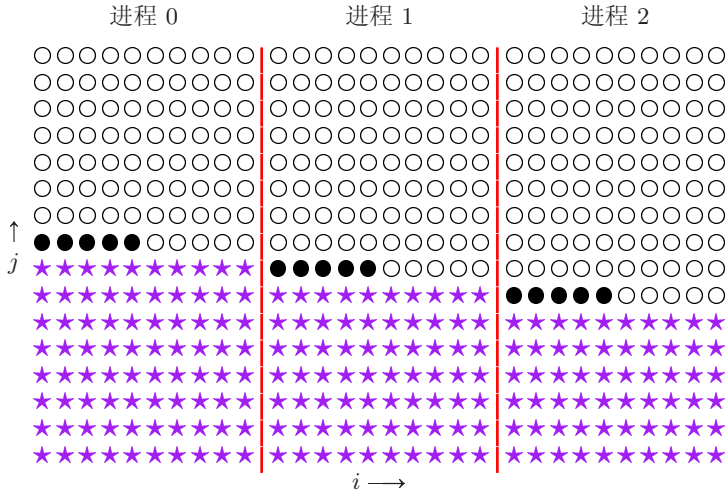


图 9.1 “★”代表已经完成的部分，
“●”代表正在计算的部分，
“○”代表尚未计算的部分。
流水线方法计算流程示意图

```
do j0 = 1, m, b
   j1 = min(m, j0+b-1)
   if (k>0) 从 pk-1 接收 a(nk-1, j0⋯j1)
   do j = j0, j1
      do i = nk, nk+1-1
        a(i,j) = F(a(i-1,j))
      enddo
   enddo
   if (k<p-1) 发送 a(nk+1-1, j0⋯j1) 至 pk+1
enddo
```

其中 k 为进程号，$b \geqslant 1$ 为组 (块) 的大小。

容易看出，分块算法流水线算法的数据长度，即通信次数，由 m 变成了 $l = \lceil m/b \rceil$。b 较大时，数据长度短，算法的并行度差但通信粒度大。b 较小时，数据长度长，算法的并行度较好但通信粒度较小。当 $m \gg p$ 时，可以通过选择合适的 b 值来获得较好的并行性能。此外，如果式 (9.7) 的计算需要重复进行 (如迭代法)，则总数据长度将变为重复计算的次数乘以 l。

9.5.2 模型问题的并行效率分析

这里为上节所给出的分块流水线算法建立一个简单的并行模型以分析它的并行效率及算法中一些参数的作用。

采用分块流水线算法完成全部计算需要 $l + p - 1$ 拍，其中 $l = \lfloor m/b \rfloor$ 为数据长度，流水线长度等于进程数 p (参看 9.5.1)。流水线计算过程中每拍所花费的时间可分解为:

$$T_{\text{cpu}} + T_{\text{comm}} + T_{\text{lat}}$$

其中：

$$\begin{cases} T_{\text{cpu}} = \dfrac{n}{p}\dfrac{m}{l}T_0, & \text{每拍计算时间} \\ T_{\text{comm}} = \sigma_p \dfrac{m}{l}T_1, & \text{每拍数据传送时间} \\ T_{\text{lat}} = \sigma_p T_2, & \text{每拍通信延迟} \end{cases}$$

其中 T_0 为计算一次 $F(\cdot,\cdot)$ 所需要的时间，对应于 CPU 计算速度；T_1 为传递一个浮点数所需要的时间，对应于通信带宽的倒数；T_2 为完成一次发送或接收所需要的通信开销，对应于通信延迟。σ_p 为依赖于 p 的常数：$p = 1$ 时 $\sigma_p = 0$，$p = 2$ 时 $\sigma_p = 1$，$p > 2$ 时 $\sigma_p = 2$。从

而完成全部计算需要的时间为：

$$\begin{aligned} T &= (l+p-1)(T_{\text{cpu}}+T_{\text{comm}}+T_{\text{lat}}) \\ &= (l+p-1)\big[(n/p)(m/l)T_0+\sigma_p(m/l)T_1+\sigma_p T_2\big] \\ &= \big[(p-1)T_0nm/p+\sigma_p(p-1)T_1m\big]/l \\ &\quad +\sigma_p T_2 l+T_0nm/p+\sigma_p T_1 m+\sigma_p T_2(p-1) \end{aligned} \tag{9.8}$$

为了突出数据长度对并行效率的影响，可将公式 (9.8) 写成如下形式：

$$T=\alpha/l+\beta\cdot l+\gamma \tag{9.9}$$

其中：

$$\begin{aligned} \alpha &= (p-1)T_0nm/p+\sigma_p(p-1)T_1m, \\ \beta &= \sigma_p T_2, \\ \gamma &= T_0nm/p+\sigma_p T_1 m+\sigma_p T_2(p-1) \end{aligned}$$

公式 (9.9) 恰当地刻划了流水线方法中的数据长度对并行效率的影响。容易算出使得总计算时间最短的最优数据长度为：

$$l_{\text{opt}}=\sqrt{\alpha/\beta}$$

相应计算时间为：

$$T_{\text{opt}}=2\sqrt{\alpha\beta}+\gamma$$

图 9.2 给出了分块流水线算法中当进程数和问题总规模固定时计算时间 T 随数据长度 l 的典型变化曲线。$l=1$ 和 $l=m$ 分别代表两个极端情况，前者对应的计算过程是串行的，而后者对应着不分块的流水线算法。

应该指出上述公式给出的最优数据长度公式是在简化模型下得到的，实际情况则要复杂得多。例如，不同的分块大小会改变计算过程中的访存模式，从而影响到处理机浮点性能乃至通信性能发挥的好坏。

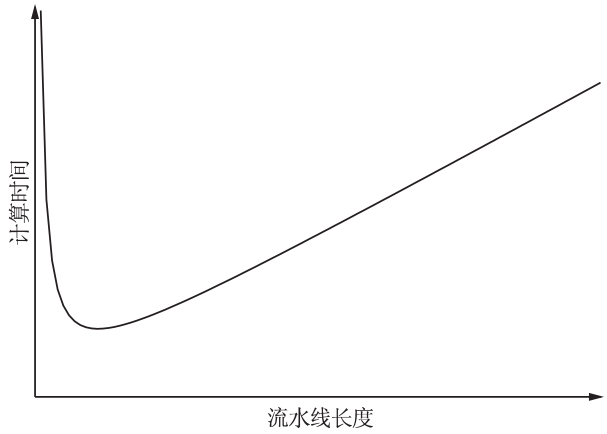


图 9.2 分块流水线方法计算时间随流水线长度的变化曲线

9.5.3 二维热传导方程的分块流水线算法程序实例

本节给出二维热传导方程分块流水线算法的程序实例。程序中 NBX 和 NBY 分别为沿 x 方向和沿 y 方向的流水线算法的分块大小，它们对应于算法 9.2 中的 b。为了便于实现，记 $k_x = \Delta t/h_x^2$，$k_y = \Delta t/h_y^2$，则方程 (9.6) 可写成如下形式：

$$\begin{aligned} &2(1+k_x)\tilde{u}_{i,j}^{n+\frac{1}{2}} - k_x\tilde{u}_{i-1,j}^{n+\frac{1}{2}} - k_x\tilde{u}_{i+1,j}^{n+\frac{1}{2}} \\ &\quad = 2(1-k_y)u_{i,j}^n + k_y u_{i,j-1}^n + k_y u_{i,j+1}^n \end{aligned} \tag{9.10a}$$

$$\begin{aligned} &2(1+k_y)u_{i,j}^{n+1} - k_y u_{i,j-1}^{n+1} - k_y u_{i,j+1}^{n+1} \\ &\quad = 2(1-k_x)\tilde{u}_{i,j}^{n+\frac{1}{2}} + k_x\tilde{u}_{i-1,j}^{n+\frac{1}{2}} + k_x\tilde{u}_{i+1,j}^{n+\frac{1}{2}} \end{aligned} \tag{9.10b}$$

为了求解方程 (9.10a)，需要中间解 $\tilde{u}^{n+\frac{1}{2}}$ 在 $i = 0$ (即 $x = 0$) 和 $i =$ IM (即 $x = W$) 处的边界条件，将方程 (9.10a) 与 (9.10b) 相减

得：

$$\begin{aligned} 4\tilde{u}_{i,j}^{n+\frac{1}{2}} = {} & 2(1-k_y)u_{i,j}^{n} + k_y u_{i,j-1}^{n} + k_y u_{i,j+1}^{n} + \\ & 2(1+k_y)u_{i,j}^{n+1} - k_y u_{i,j-1}^{n+1} - k_y u_{i,j+1}^{n+1} \end{aligned} \tag{9.11}$$

程序中利用公式 (9.11) 来确定 $\tilde{u}_{i,j}^{n+\frac{1}{2}}$ 在 $i=0$ 和 $i=$ `IM` 处的值。

计算过程中，需要在两个空间方向交替求解下面形式的三对角线性方程组：

$$bv_{k-1} + av_k + bv_{k+1} = c_k, \quad k = 1, \ldots, N$$

其中 v_0 和 v_{N+1} 为已知值。对方程 (9.10a) 而言，

$$\begin{aligned} a &= 2(1+k_x) \\ b &= -k_x \\ c_k &= 2(1-k_y)u_{k,j}^{n} + k_y u_{k,j-1}^{n} + k_y u_{k,j+1}^{n} \\ N &= \texttt{IM-1} \end{aligned}$$

而对方程 (9.10b) 而言，

$$\begin{aligned} a &= 2(1+k_y) \\ b &= -k_y \\ c_k &= 2(1-k_x)\tilde{u}_{i,k}^{n+\frac{1}{2}} + k_x \tilde{u}_{i-1,k}^{n+\frac{1}{2}} + k_x \tilde{u}_{i+1,k}^{n+\frac{1}{2}} \\ N &= \texttt{JM-1} \end{aligned}$$

这些方程的矩阵形式如下：

$$AX = B$$

其中：

$$A=\begin{pmatrix} a & b & & & \\ b & a & b & & \\ & \ddots & \ddots & \ddots & \\ & & b & a & b \\ & & & b & a \end{pmatrix},\quad X=\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{N-1} \\ v_N \end{pmatrix},\quad B=\begin{pmatrix} c_1-bv_0 \\ c_2 \\ \vdots \\ c_{N-1} \\ c_N-bv_{N+1} \end{pmatrix}$$

由于系数矩阵 A 不随时间变化，可预先对它进行 LU 分解，并存储 LU 分解后的形式，以减少时间推进过程中的计算量：

$$A=LU=\begin{pmatrix} 1 & & & & \\ l_1 & 1 & & & \\ & \ddots & \ddots & & \\ & & l_{N-2} & 1 & \\ & & & l_{N-1} & 1 \end{pmatrix}\begin{pmatrix} d_1 & u_1 & & & \\ & d_2 & u_2 & & \\ & & \ddots & \ddots & \\ & & & d_{N-1} & u_{N-1} \\ & & & & d_N \end{pmatrix}$$

其中 d_k，l_k，u_k 的计算公式如下：

$$\begin{cases} d_1=a, \\ u_k=b,\quad l_k=b/d_k,\quad d_{k+1}=a-l_ku_k, \\ k=1,\ldots,N-1 \end{cases}$$

程序中实际存储的是矩阵 U 的对角线元素 $\{d_1,d_2,\ldots,d_N\}$ 的倒数 $\{1/d_1,1/d_2,\ldots,1/d_N\}$，这样可以将除法运算转变为乘法运算，有助于改进程序性能。

另外，程序中用于边界通信的数据类型 `HTYPE` 和 `VTYPE` 与代码 9.1 中的定义有所不同。以 `VTYPE` 为例，因为在分块流水线计算过程中需要发送沿 y 方向的一条边界线上不同长度的数据段，因此

将它定义成

$$\Big\{\big(\texttt{MPI\_DOUBLE\_PRECISION},\,0\big),\\ \big(\texttt{MPI\_UB},\,(\texttt{IML+1})\times\texttt{extent}(\texttt{MPI\_DOUBLE\_PRECISION})\big)\Big\}$$

其中 IML+1 为数组在 x 方向的维数，这样定义的 VTYPE 只包含一个数，但它的域是数组第一维的大小 (注意 Fortran 数组的第一维在内存中是连续排列的)，正好是沿 y 方向的相邻两个数之间的位移，这样实际通信时只要直接在 MPI_Send 或 MPI_Recv 中用 count 参数指定数据个数即可。HTYPE 则简单地定义为 MPI_DOUBLE_PRECISION，因为沿 x 方向数据是连续存放的。

一个有意思的现象是，代码 9.2 在时间推进过程中没有进行全局同步，因此时间推进时有可能在不同时间层之间形成另一个层次的流水线，有兴趣的读者可以自行分析一下代码 9.2 的详细运行过程。

最后要指出的是，在测试代码 9.2 时，由于二维问题计算粒度太小，难以看出分块大小对并行性能的影响，如果网络较慢的话，并行效率也不一定好，测试并行效率时应该适当将网格规模取大些，例如取 2048×2048。事实上，本章介绍的分块流水线方法用于求解三维问题时更加有效，二维问题由于受本身计算规模的限制，并行计算的意义并不大，这里选用二维问题做为例子主要是为了突出算法的主要思想、方便算法的描述、以及减少程序实例中的代码长度。

代码 9.2: 规则区域上的二维热传导方程：ADI 格式，分块流水线算法。

文件名: code/heat/heat2.f

```
1 ! 二维热传导方程：Peaceman-Rachford 格式，分块流水线算法
2       IMPLICIT DOUBLE PRECISION (A-H,O-Z)
3       INCLUDE  'mpif.h'
```

```
      PARAMETER(DW=2.D0, DH=3.D0) ! 问题求解区域沿 X、Y 方向的大小
      PARAMETER(DT=.05D0)         ! 时间步长
      PARAMETER(IM=50, JM=100)    ! 沿 X、Y 方向的全局网格规模
      PARAMETER(NPX=1, NPY=1)     ! 沿 X、Y 方向的进程个数
      PARAMETER(IML=IM/NPX, JML=JM/NPY)
        ! 各进程沿 X、Y 方向的局部网格规模, 仅为全局网格规模的 1/(NPX*NPY)
      DIMENSION U (0:IML+1,0:JML+1) ! 当前时间层的近似解
      DIMENSION U0(0:IML+1,0:JML+1) ! 中间变量
      DOUBLE PRECISION KX, KY     ! Δt/h_x^2 和 Δt/h_y^2
      DOUBLE PRECISION T0, T1     ! 用于统计运行时间
      INTEGER  NPROC              ! mpirun 启动的进程个数, 必须等于 NPX*NPY
      INTEGER  MYRANK, MYLEFT, MYRIGHT, MYUPPER, MYLOWER
                                  ! 各进程自身的进程号, 4 个相邻进程的进程号
      INTEGER  MEPX,MEPY          ! 各进程自身的进程号沿 X、Y 方向的坐标
      INTEGER  IST,IEND,JST,JEND
                    ! 各进程沿 X、Y 方向的内部网格结点的起始和终止坐标
      INTEGER  HTYPE, VTYPE
                    ! MPI 用户自定义数据类型, 表示各进程沿 X、Y 方向
                    ! 与相邻进程交换的数据单元
! 用于求解三对角线性方程组的变量
      PARAMETER(NBX=50, NBY=50)   ! 分块流水线方法沿 X、Y 方向分块大小
      DOUBLE PRECISION LX(0:IML-1),DX(IML),UX(IML) ! 沿 X 方向的 LU 分解系数
      DOUBLE PRECISION LY(0:JML-1),DY(JML),UY(JML) ! 沿 Y 方向的 LU 分解系数
      INTEGER  LENS(2), DISPS(2), TYPES(2) ! 用于定义 VTYPE 的辅助数组
                    ! 注: 某些 64 位机器上可能需要将 DISPS 声明成 INTEGER*8
      INTEGER  STATUS(MPI_STATUS_SIZE)
! Constants
      DATA ONE/1.D0/, TWO/2.D0/, ZERO/0.D0/, HALF/.5D0/
      DATA LENS/1,1/, TYPES/MPI_DOUBLE_PRECISION, MPI_UB/
! In-line functions
      solution(x,y,t) = EXP(-t-t)*SIN(x)*COS(y) ! 解析解: e^{-2t} sin x cos y
! 程序可执行语句开始
      CALL MPI_Init(IERR)
      CALL MPI_Comm_size(MPI_COMM_WORLD,NPROC,IERR)
      IF (NPROC.NE.NPX*NPY.OR.MOD(IM,NPX).NE.0.OR.MOD(JM,NPY).NE.0) THEN
```

```
        PRINT *, '+++ Incorrect parameters, abort +++'
        CALL MPI_Finalize(IERR)
        STOP
      ENDIF
! 按自然序 (先沿 X 方向, 后沿 Y 方向) 确定各进程自身及其 4 个相邻进程的进程号
      CALL MPI_Comm_rank(MPI_COMM_WORLD,MYRANK,IERR)
      MYLEFT  = MYRANK - 1
      IF (MOD(MYRANK,NPX).EQ.0)   MYLEFT=MPI_PROC_NULL
      MYRIGHT = MYRANK + 1
      IF (MOD(MYRIGHT,NPX).EQ.0)  MYRIGHT=MPI_PROC_NULL
      MYUPPER = MYRANK + NPX
      IF (MYUPPER.GE.NPROC)       MYUPPER=MPI_PROC_NULL
      MYLOWER = MYRANK - NPX
      IF (MYLOWER.LT.0)           MYLOWER=MPI_PROC_NULL
      MEPY=MYRANK/NPX
      MEPX=MYRANK-MEPY*NPX
! 基本变量赋值, 确定各进程负责的子区域
      HX = DW/IM                ! X 方向网格步长 h_x
      KX = DT/(HX*HX)           ! Δt/h_x^2
      HY = DH/JM                ! Y 方向网格步长 h_y
      KY = DT/(HY*HY)           ! Δt/h_y^2
! 各子区域负责计算的范围
      IST=1
      IEND=IML
      IF (MEPX.EQ.NPX-1) IEND=IEND-1   ! 最右边的区域 X 方向少一个点
      JST=1
      JEND=JML
      IF (MEPY.EQ.NPY-1) JEND=JEND-1   ! 最上边的区域 Y 方向少一个点
! 初始条件 (注意我们需要区域角点处的值)
      DO J=JST-1, JEND+1
        yy=(J+MEPY*JML)*HY
        DO I=IST-1, IEND+1
          xx=(I+MEPX*IML)*HX
          U(I,J)=solution(xx,yy,ZERO) ! 初始解
        ENDDO
      ENDDO
```

```
! X 方向三对角矩阵的 LU 分解，各处理器独立计算自己需要的那部分系数
! (跳过前面 MEPX*IML 个不属于自己的系数)
      AX = TWO * (ONE + KX)
      BX = -KX
      DD = ONE / AX
      DO I = 1, MEPX*IML
         DU = BX
         DL = BX * DD
         DD = ONE / (AX - DL * DU)
      ENDDO
      IF (MEPX.EQ.0) THEN
         LX(0) = BX
      ELSE
         LX(0) = DL
      ENDIF
      DX(1) = DD
      DO I = IST, IEND-1
         UX(I) = BX
         LX(I) = BX * DX(I)
         DX(I+1) = ONE / (AX - LX(I) * UX(I))
      ENDDO
      UX(IEND) = BX
! Y 方向三对角矩阵的 LU 分解，各处理器独立计算自己需要的那部分系数
! (跳过前面 MEPY*JML 个不属于自己的系数)
      AY = TWO * (ONE + KY)
      BY = -KY
      DD = ONE / AY
      DO J = 1, MEPY*JML
         DU = BY
         DL = BY * DD
         DD = ONE / (AY - DL * DU)
      ENDDO
      IF (MEPY.EQ.0) THEN
         LY(0) = BY
      ELSE
         LY(0) = DL
```

```
      ENDIF
      DY(1) = DD
      DO J = JST, JEND-1
         UY(J) = BY
         LY(J) = BY * DY(J)
         DY(J+1) = ONE / (AY - LY(J) * UY(J))
      ENDDO
      UY(JEND) = BY
! 数据类型定义
      HTYPE=MPI_DOUBLE_PRECISION
                ! HTYPE 用于发送沿 X 方向一条线上的一段数据
      DISPS(1) = 0
      CALL MPI_Type_extent(MPI_DOUBLE_PRECISION, DISPS(2), IERR)
      DISPS(2) = DISPS(2) * (IML+2)
      CALL MPI_Type_struct(2, LENS, DISPS, TYPES, VTYPE, IERR)
      CALL MPI_Type_commit(VTYPE, IERR)
                ! VTYPE 用于发送沿 Y 方向一条线上的一段数据
! 时间推进
      NT=0
      T0 = MPI_Wtime()
100   CONTINUE   ! 主循环
      NT=NT+1
      T=NT*DT
!---- X 方向求解: 方程 (9.10a)
      DO J=JST,JEND
      DO I=IST,IEND
         U0(I,J)=TWO*(U(I,J)-KY*(U(I,J)-HALF*(U(I,J-1)+U(I,J+1)))) ! 右端项
      ENDDO
      ENDDO
!     X 方向边界条件
      IF (MEPX.EQ.0) THEN
!        中间解 u^{n+1/2} 的边界条件前半部分 (保存于 U0)
         I=IST-1
         DO J=JST,JEND
            U0(I,J)=U(I,J)-KY*(U(I,J)-HALF*(U(I,J-1)+U(I,J+1)))
```

```
146          ENDDO
147  !       u^{n+1} 的边界条件
148          xx = ZERO
149          DO J=JST-1,JEND+1
150             yy=(J+MEPY*JML)*HY
151             U(I,J)=solution(xx,yy,T)
152          ENDDO
153  !       中间解 ũ^{n+1/2} 的边界条件后半部分
154          DO J=JST,JEND
155             U0(I,J)=HALF*(U0(I,J) +
156       &                   U(I,J)+KY*(U(I,J)-HALF*(U(I,J-1)+U(I,J+1))))
157          ENDDO
158       ENDIF
159       IF (MEPX.EQ.NPX-1) THEN
160  !       中间解 ũ^{n+1/2} 的边界条件前半部分 (存在 U0 中)
161          I=IEND+1
162          DO J=JST,JEND
163             U0(I,J)=U(I,J)-KY*(U(I,J)-HALF*(U(I,J-1)+U(I,J+1)))
164          ENDDO
165  !       u^{n+1} 的边界条件
166          xx = DW
167          DO J=JST-1,JEND+1
168             yy=(J+MEPY*JML)*HY
169             U(I,J)=solution(xx,yy,T)
170          ENDDO
171  !       中间解 ũ^{n+1/2} 的边界条件后半部分
172          DO J=JST,JEND
173             U0(I,J)=HALF*(U0(I,J) +
174       &                   U(I,J)+KY*(U(I,J)-HALF*(U(I,J-1)+U(I,J+1))))
175          ENDDO
176       ENDIF
177  !    下三角求解
178       DO JJ = JST, JEND, NBY
179          JE = MIN(JEND, JJ+NBY-1)
180          CALL MPI_Recv(U0(IST-1,JJ), JE-JJ+1, VTYPE, MYLEFT, 11,
```

```
181      &                MPI_COMM_WORLD, STATUS, IERR)
182         DO J = JJ, JE
183         DO I = IST, IEND
184            U0(I,J) = U0(I,J) - LX(I-1) * U0(I-1,J)
185         ENDDO
186         ENDDO
187         CALL MPI_Send(U0(IEND,JJ),  JE-JJ+1, VTYPE, MYRIGHT, 11,
188      &                MPI_COMM_WORLD, IERR)
189       ENDDO
190 !     上三角求解
191       DO JJ = JST, JEND, NBY
192          JE = MIN(JEND, JJ+NBY-1)
193          CALL MPI_Recv(U0(IEND+1,JJ), JE-JJ+1, VTYPE, MYRIGHT, 22,
194      &                MPI_COMM_WORLD, STATUS, IERR)
195          DO J = JJ, JE
196          DO I = IEND, 1, -1
197             U0(I,J) = (U0(I,J) - UX(I) * U0(I+1,J)) * DX(I)
198          ENDDO
199          ENDDO
200          CALL MPI_Send(U0(IST,JJ),    JE-JJ+1, VTYPE, MYLEFT, 22,
201      &                MPI_COMM_WORLD, IERR)
202       ENDDO
203 !     沿 X 方向交换定义在辅助网格结点上的近似解
204       CALL MPI_Sendrecv(U0(IEND,1), JEND-JST+1, VTYPE, MYRIGHT, 33,
205      &                  U0(0,1),    JEND-JST+1, VTYPE, MYLEFT,  33,
206      &                  MPI_COMM_WORLD, STATUS, IERR)
207 !---- Y 方向求解: 方程 (9.10b)
208       DO J=JST,JEND
209       DO I=IST,IEND
210          U(I,J)=TWO*(U0(I,J)-KX*(U0(I,J)-HALF*(U0(I-1,J)+U0(I+1,J)))) ! 右端项
211       ENDDO
212       ENDDO
213 !     Y 方向边界条件
214       IF (MEPY.EQ.0) THEN
215          J=JST-1
216          yy = ZERO
```

```
        DO I=IST,IEND
           xx=(I+MEPX*IML)*HX
           U(I,J)=solution(xx,yy,T)
        ENDDO
      ENDIF
      IF (MEPY.EQ.NPY-1) THEN
        J=JEND+1
        yy = DH
        DO I=IST,IEND
           xx=(I+MEPX*IML)*HX
           U(I,J)=solution(xx,yy,T)
        ENDDO
      ENDIF
!     下三角求解
      DO II = IST, IEND, NBX
        IE = MIN(IEND, II+NBX-1)
        CALL MPI_Recv(U(II,JST-1), IE-II+1, HTYPE, MYLOWER, 44,
     &                MPI_COMM_WORLD, STATUS, IERR)
        DO J = JST, JEND
        DO I = II, IE
           U(I,J) = U(I,J) - LY(J-1) * U(I,J-1)
        ENDDO
        ENDDO
        CALL MPI_Send(U(II,JEND),  IE-II+1, HTYPE, MYUPPER, 44,
     &                MPI_COMM_WORLD, IERR)
      ENDDO
!     上三角求解
      DO II = IST, IEND, NBX
        IE = MIN(IEND, II+NBX-1)
        CALL MPI_Recv(U(II,JEND+1), IE-II+1, HTYPE, MYUPPER, 55,
     &                MPI_COMM_WORLD, STATUS, IERR)
        DO J = JEND, 1, -1
        DO I = II, IE
           U(I,J) = (U(I,J) - UY(J) * U(I,J+1)) * DY(J)
        ENDDO
        ENDDO
```

```
253         CALL MPI_Send(U(II,JST),    IE-II+1, HTYPE, MYLOWER, 55,
254      &                MPI_COMM_WORLD, IERR)
255       ENDDO
256 !     沿 Y 方向交换定义在辅助网格结点上的近似解
257       CALL MPI_Sendrecv(U(1,JEND),  IEND-IST+1, HTYPE, MYUPPER, 66,
258      &                  U(1,0),     IEND-IST+1, HTYPE, MYLOWER, 66,
259      &                  MPI_COMM_WORLD, STATUS, IERR)
260 ! 注: 沿 X 方向辅助网格结点上的近似解没有更新 (下一个时间层的计算不需要它)
261       T1 = MPI_Wtime()
262       IF (MYRANK.EQ.0) PRINT *, 'T=', T, '   wtime=', T1 - T0
263       IF (T.LT.1.0) GOTO 100
264 ! 计算与精确解间的误差
265       ERR0=ZERO
266       DO J=JST, JEND
267          yy=(J+MEPY*JML)*HY
268          DO I=IST, IEND
269             xx=(I+MEPX*IML)*HX
270             ERR0=MAX(ERR0,ABS(U(I,J)-solution(xx,yy,T)))
271          ENDDO
272       ENDDO
273       CALL MPI_Reduce(ERR0, ERR, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0,
274      &                MPI_COMM_WORLD, IERR)
275       IF (MYRANK.EQ.0) THEN
276          PRINT *, 'Error: ', ERR
277          PRINT *, 'Wall time: ', T1 - T0
278       ENDIF
279       CALL MPI_Finalize(IERR)
280       STOP
281       END
```

习　题

1. 运行代码 9.1 并统计分析不同问题规模、进程数与进程划分、不同并行平台上的并行效率。

2. 参照代码 8.2 中的优化方法改造代码 9.1，使得相邻子区域边界上的通信与子区域内部的计算推进重叠进行，测试新代码的并行效率。

3. 统计代码 9.1 中的浮点运算次数，修改代码，在程序结束时打印出实际达到的 Mflops 值，并根据处理机的峰值性能计算程序的实际效率。

4. 考虑用自然顺序 Gauss–Seidel 迭代求解离散 Poisson 方程 (8.6)：

$$\begin{cases} u_{i,j}^k = \dfrac{h_x^2 h_y^2 f_{i,j} + h_y^2 (u_{i-1,j}^k + u_{i+1,j}^{k-1}) + h_x^2 (u_{i,j-1}^k + u_{i,j+1}^{k-1})}{2(h_x^2 + h_y^2)}, \\ i = 1, \ldots, \mathtt{IM} - 1, \quad j = 1, \ldots, \mathtt{JM} - 1 \end{cases}$$

给出它的分块流水线并行算法，并编写 MPI 并行程序。

5. 编译、运行代码 9.2，测试不同进程规模、问题规模下流水线算法的最优分块大小。

6. 统计代码 9.2 中的浮点运算次数，修改代码，在程序结束时打印出实际达到的 Mflops 值，并根据处理机的峰值性能计算程序的实际效率。

7. 设计三维热传导方程 ADI 格式的分块流水线算法。

第 3 部分

附　　录

附录 A　并行程序开发工具与高性能程序库

本附录介绍几个高性能计算的基础开源软件。限于篇幅，这里仅限于简单介绍每个软件的功能、特点和基本用法，使读者对它们有一个基本的了解，在实际应用中能够有效地利用它们来完成特定的工作。本附录的内容主要依据相关软件的使用手册编写，细节内容请在具体使用时参阅它们所附带的资料。

A.1　BLAS

BLAS (Basic Linear Algebra Subroutines) 是一组高质量的基本向量、矩阵运算子程序。最早的 BLAS 是一组 Fortran 函数和子程序，后来又发展了其他语言接口，包括 C、Java 等。BLAS 的官方网址在

`http://www.netlib.org/blas/`

国内镜像为

`http://netlib.amss.ac.cn/blas/`。

由于 BLAS 涉及最基本的向量、矩阵运算，因此在程序中合理地调用 BLAS 子程序，并且在不同平台上选用经过特殊优化的 BLAS 库可以大大提高程序的性能。BLAS 的主要贡献是将高性能代数计算程序的开发同针对特定机器的性能优化独立开来：代数算法程序的开发者只需要运用适当的分块技术将计算过程变成矩阵、向量的基本运算并调用相应的 BLAS 子程序而不必考虑与计算机体系结构相关的性能优化问题 (后者往往是非常繁杂的)，而针对不同平台的优化 BLAS 库的开发则由计算机厂商和专业开发人员来完成。这一模式大提高了高性能代数程序的开发效率。线性代数软件包如 LAPACK、ScaLAPACK 等都是基于这一思想设计的。

对于 BLAS 库，现在有多种不同的优化实现，适用于 Intel/Linux 平台的主要有以下几种：

BLAS 参考实现

这是一组标准 Fortran 子程序，可以从 BLAS 的主页下载：`http://www.netlib.org/blas/index.html`；

ATLAS 库 (Automatically Tuned Linear Algebra Software)

它可以在不同平台上自动生成优化的 BLAS 库，其主页为 `http://math-atlas.sourceforge.net/`；

Goto 库

Kazushige Goto 开发的一套高性能 BLAS 库，其主页为 `http://www.cs.utexas.edu/users/flame/goto/`；

MKL 库 (Math Kernel Library)

Intel 为自己的 CPU 专门优化的基本数学运算库，其中包含 BLAS 库，其主页为 `http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm`。

前三种库可以免费下载，而 Intel MKL 库是商业软件，对于商业应用需要购买，而非商业应用可以免费使用。

BLAS 从结构上分成三个层次：Level 1 BLAS、Level 2 BLAS 和 Level 3 BLAS。其中 Level 1 BLAS 涉及向量和向量、向量和标量间的运算，Level 2 BLAS 涉及向量和矩阵间的运算，Level 3 BLAS 则涉及矩阵和矩阵间的运算。此外，还有一个辅助子程序 `XERBLA` 用于错误信息的打印。通常在优化 BLAS 库中，层次越高的子程序性能改善越大，例如，许多平台上优化 BLAS 库中的矩阵乘子程序 `DGEMM` 相对于它的标准 Fortran 版本 `dgemm.f` 的性能提升可接近 10 倍甚至

更多。因此，使用 BLAS 库的一个基本原则是：尽可能地使用 Level 3 BLAS 中的子程序，其次是 Level 2 BLAS 中的子程序。

BLAS 支持四种浮点数格式：单精度实数 (REAL)、双精度实数 (DOUBLE PRECISION)、单精度复数 (COMPLEX) 和双精度复数 (DOUBLE COMPLEX 或 COMPLEX*16)。BLAS 的子程序名中首字母表示浮点数类型：“S” 为单精度实数、“D” 为双精度实数、“C” 为单精度复数、“Z” 为双精度复数。下面的介绍中将主要以双精度实数子程序为例介绍，其他类型的子程序只需要将子程序名中的首字母 “D” 相应地换成 “S”、“C” 或 “Z” 即可。

A.1.1 Level 1 BLAS

Level 1 BLAS 包含一组标量与向量、向量与向量运算的子程序 [24]。

- 向量内积与模

$$x^{\mathrm{T}}y, \quad x^{\mathrm{H}}y, \quad \|x\|_2, \quad \|x\|_1, \quad \dots$$

相关的子程序有 DDOT，DDOTU，DDOTC，DNRM2，DASUM 等。

- 向量、标量运算

$$x := \alpha x, \quad y := x, \quad x \text{ 与 } y \text{ 交换}, \quad y := \alpha x + y$$

相关的子程序有 DSCAL，DCOPY，DSWAP，DAXPY。

- 平面旋转变换

相关的子程序有 DROT，DROTG，DROTM 和 DROTMG。

关于这些子程序的详细说明请参看有关文档或它们的 Fortran 源程序。

A.1.2 Level 2 BLAS

Level 2 BLAS 包含下面几类矩阵、向量运算子程序 [25]：

矩阵乘向量

有下面几种形式

$$y := \alpha Ax + \beta y, \quad y := \alpha A^{\mathrm{T}}x + \beta y, \quad y := \alpha \bar{A}^{\mathrm{T}}x + \beta y$$

其中 α 和 β 代表标量，x 和 y 代表向量，A 代表矩阵。A 可以是普通矩阵、对称 (Hermitian) 矩阵、带状矩阵或 (上或下) 三角矩阵。

秩 1、秩 2 修正

有下面几种形式

$$A := \alpha xy^{\mathrm{T}} + A, \quad A := \alpha x\bar{y}^{\mathrm{T}} + A,$$
$$H := \alpha x\bar{x}^{\mathrm{T}} + H, \quad H := \alpha x\bar{y}^{\mathrm{T}} + \bar{\alpha}y\bar{x}^{\mathrm{T}} + H$$

其中 H 代表 Hermitian 矩阵。

三角方程组求解

有下面几种形式

$$x := T^{-1}x, \quad x := T^{-T}x, \quad x := \bar{T}^{-T}x$$

其中 T 代表非奇异三角矩阵。

Level 2 BLAS 子程序名称中最后一个或两个字母表示运算类型：“MV” 表示矩阵乘向量 (Matrix 乘 Vector)，“R” 表示秩 1 修正，“R2” 表示秩 2 修正，“SV” 表示解线性方程组；中间两个字母表示矩阵类型：“GE” 表示普通矩阵，“GB” 表示普通带状矩阵，“HE” 表示 Hermitian 矩阵，“SY” 表示对称矩阵，“HP” 表示压缩存储的 Hermitian 矩阵，“SP” 表示压缩存储的对称矩阵，“HB” 表示带状 Hermitian

矩阵，“SB” 表示带状对称矩阵，“TR” 表示三角矩阵，“TP” 表示压缩存储的三角矩阵，“TB” 表示带状三角矩阵。例如，子程序 DGEMV 计算普通矩阵乘以向量，而子程序 DTRSV 求解普通三角线性方程组。

对于对称或 Hermitian 矩阵而言，BLAS 只使用它们的上三角或下三角部分，具体由参数 UPLO 指定。BLAS 允许两种存储格式，第一种格式按普通形式存储在一个二维数组中，第二种格式称为“压缩存储”格式，它将矩阵的行或列压缩存储在一个一维数组中：如果存储的是上三角部分 (UPLO = 'U')，则顺序存储矩阵的列，如果存储的是下三角部分 (UPLO = 'L')，则顺序存储矩阵的行。此外，由于 Hermitian 矩阵的对角线元素总是实数，BLAS 不使用它们的虚部。

对于三角矩阵，参数 UPLO 指定是上三角还是下三角矩阵。在压缩存储格式中三角矩阵总是按列依次存储。

其他存储格式 (如带状矩阵) 以及各子程序的详细说明请自行参看有关文档或它们的 Fortran 源程序。

A.1.3 Level 3 BLAS

Level 3 BLAS 由下面几类矩阵运算子程序构成 [26]：

矩阵乘积

包括下面几种形式：

$$C := \alpha AB + \beta C, \quad C := \alpha A^{\mathrm{T}} B + \beta C,$$
$$C := \alpha AB^{\mathrm{T}} + \beta C, \quad C := \alpha A^{\mathrm{T}} B^{\mathrm{T}} + \beta C$$

其中 α 和 β 为标量，A，B，C 为矩阵。

对称矩阵秩 k、秩 $2k$ 修正

有下面几类运算：

$$C := \alpha AA^{\mathrm{T}} + \beta C, \quad C := \alpha A^{\mathrm{T}}A + \beta C,$$
$$C := \alpha AB^{\mathrm{T}} + \alpha BA^{\mathrm{T}} + \beta C,$$
$$C := \alpha A^{\mathrm{T}}B + \alpha B^{\mathrm{T}}A + \beta C$$

其中 C 为对称矩阵。

矩阵与三角矩阵的乘积

有下面几类运算：

$$B := \alpha TB, \quad B := \alpha T^{\mathrm{T}}B, \quad B := \alpha BT, \quad B := \alpha BT^{\mathrm{T}}$$

其中 T 为三角矩阵。

求解含多个右端项的三角线性方程组

有下面几类运算：

$$B := \alpha T^{-1}B, \quad B := \alpha T^{-T}B, \quad B := \alpha BT^{-1}, \quad B := \alpha BT^{-T}$$

此外，上述包含矩阵转置的运算中对复矩阵还提供了相应的共轭转置运算，例如 $C := \alpha AA^{\mathrm{H}} + \beta C$。

Level 3 BLAS 子程序的命名规则与 Level 2 BLAS 类似：子程序名的最后几个字母用于表明矩阵运算的类型，“MM” 表示矩阵乘积 (Matrix 乘 Matrix)，“RK” 表示秩 k 修正，“R2K” 表示秩 $2k$ 修正，“SM” 表示解 (三角) 方程；中间两个字母表示矩阵的类型：“GE” 表示普通矩阵，“SY” 表示对称矩阵，“HE” 表示 Hermitian 矩阵，“TR” 表示三角矩阵。例如，DGEMM 表示普通矩阵乘积，而 DSYMM 表示对称矩阵乘积。

关于这些子程序的更详细的信息请自行参看有关文档或者它们的 Fortran 源程序。

A.2 LAPACK

LAPACK (Linear Algebra PACKage) 是由 Argonne 国家实验室、Courant 研究院和 NAG (Numerical Algorithms Group) 公司联合开发完成的线性代数函数库。LAPACK V1.0 发布于 1992 年 2 月，自 1994 年 9 月 V2.0 版发布以来受到广泛关注。1999 年 6 月发布了 V3.0 版，之后同年 10 月和 2000 年 5 月又分别发布了更新的 V3.0 版。LAPACK 的网址在 `http://www.netlib.org/lapack/`，国内镜像为 `http://netlib.amss.ac.cn/lapack/`。

LAPACK 包含了求解科学与工程计算中最常见的数值线性代数计算问题，如线性方程组、线性最小二乘问题、特征值问题和奇异值问题等。LAPACK 还可以实现矩阵分解和条件数估计等相关计算。

LAPACK 项目的最初目标是在共享存储向量并行计算机上高效地使用 EISPACK 和 LINPACK。由于 LINPACK 和 EISPACK 忽视了微处理器的多层存储结构的特点，以向量操作的形式调用 Level 1 BLAS 中的子程序完成基本运算，使得 Cache 利用率很低，处理器大部分时间花在从内存中存取数据而不是进行浮点运算，因而效率低下。LAPACK 利用分块技术解决了这个问题。其思想是对矩阵进行分块，通过分块将许多操作转换为矩阵运算，主要是矩阵乘法，这些运算调用 Level 3 BLAS 中的高效子程序来完成。此外，把原本较大的工作分为若干较小的部分也有助于提高 Cache 命中率，进一步改善程序的执行效率。移植 LAPACK 时，只要适当调整分块参数，便能使它的许多子程序的实际处理性能接近处理机的峰值性能。

A.2.1 LAPACK 软件包组成

1. 程序分类

在 LAPACK 软件包中，其子程序可以分为三类。它们是：

(1) 驱动程序 (driver routines)：用于解决一个完整问题，例如线性

方程组求解，QR 分解，或求一个实对称矩阵的特征值等。

(2) 计算程序 (computional routines)：也叫作简单 LAPACK 子程序，用以完成一个特定的计算任务，例如一个 $m \times m$ 矩阵的 LU 分解，或把一个普通实矩阵化简为上 Hessenberg 型。

(3) 辅助程序 (auxiliary routines)：是被驱动程序和计算程序调用的子程序。这些程序主要完成对子块的操作和一些常用的底层计算。例如生成初等 Householder 矩阵和计算矩阵范数等。

图 A.1 给出了 LAPACK 软件包的组成结构，其中 `SRC` 是存放源程序代码的目录。LAPACK 软件包中 `TESTING` 子目录下的 `LIN` 子目录存放测试线性系统求解程序正确性的源代码，`EIG` 子目录存放测试特征值问题求解程序正确性的源代码，而 `MAGTEN` 子目录存放生成测试矩阵的源代码。`TIMING` 子目录下的 `LIN` 子目录存放测试线性系统求解程序性能的源代码，而 `EIG` 子目录存放测试特征值问题求解程序性能的源代码。`BLAS` 子目录下的 `SRC` 子目录存放 BLAS 程序的源代码，`TESTING` 子目录存放测试 BLAS 程序正确性的源代码。`INSTALL` 子目录存放安装 LAPACK 软件包所需的 `Makefile`，`make.inc.*` 等文件。

2. 数据类型和精度

除了少数例外，LAPACK 对实数和复数数据类型提供相同的功能。例如对应于求解系数矩阵为实对称矩阵的线性方程组，LAPACK 亦提供程序求解系数矩阵为 Hermitian 矩阵和复型对称矩阵的线性方程组。然而 LAPACK 不提供相当于求解实对称三对角矩阵特征值的复型数据程序，因为 Hermitian 矩阵总是可以规约成实对称三对角矩阵。只要有可能，实型和复型对应的程序的源代码将尽量保持对应。从精度上来说，LAPACK 对所有的程序都提供单精度和双精度两个版本。双精度复型的程序需要机器的 Fortran 77 编译器对

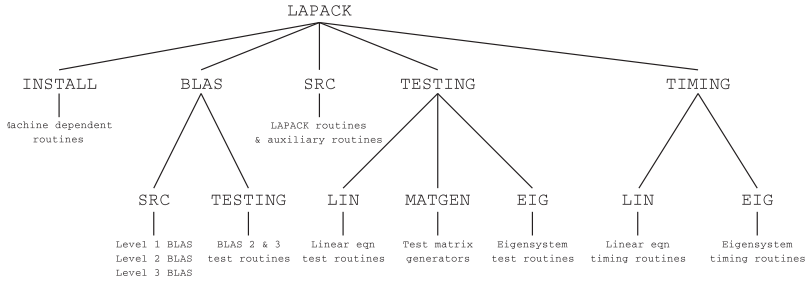


图 A.1 LAPACK 软件包目录结构

COMPLEX*16 数据类型的支持，而这种类型在大多数提供双精度计算的机器上都能支持。

3. 命名规则

LAPACK 的所有驱动和计算程序名称都具有 XYYZZZ 的形式，其中的第一个字母 X 表示程序的数据类型，如下表所示：

X 的取值	所表示的数据类型
S	单精度实型
D	双精度实型
C	单精度复型
Z	双精度复型

当提及某个程序而不管其数据类型时，通常将第一个字母用 “x” 代替。因此，xGETRF 将代表 SGETRF，DGETRF，CGETRF 和 ZGETRF 中的任一个或全部。

接下来的两个字母，YY，表示程序所操作的矩阵类型。这两个字母编码的大多数对实型和复型都适用，但少量的编码只适用于其中一种数据类型。因此当泛指在不同类型的矩阵上完成相同代数操作

的一类程序时，用 “xyy” 代替第一，第二和第三个字母。所以，xyyTRF 指所有进行三角分解的程序。

子程序命名形式的最后三个字母 ZZZ 表示该程序所完成的计算任务。例如，DGEBRD 表示该程序把一个双精度的普通实矩阵化成一个双对角阵。

辅助程序名除第 2 个和第 3 个字母通常是 “LA” 外，与驱动程序和计算程序的命名规则相似，例如 SLASCL，CLARFG。但是有两类例外：

(1) 辅助程序中非分块计算的程序与相应的分块算法的程序名相似，不同点只是前者最后一个字符是 “2”。比如 DGETF2 是与分块三角分解 DGETRF 对应的不分块三角分解程序；

(2) 一些被认为是 BLAS 功能扩充的程序与 BLAS 的命名规则相似，比如 CROT，CSYR 等。

A.2.2 LAPACK 程序文档

每个 LAPACK 程序的文档包含如下内容：

(1) SUBROUTINE 或 FUNCTION 声明以及紧随其后的用以说明参数类型和大小的变量说明部分；

(2) 程序功能的说明；

(3) 按参数序列顺序的参数描述部分；

(4) (可选) 代码段落的说明；

(5) (可选) 内部参数说明。

A.2.3 LAPACK 参数设计

1. 参数顺序

LAPACK 程序中的参数按如下顺序出现：

(1) 选项参数，

(2) 问题规模参数，

(3) 输入数组或标量参数，有些可能被结果覆盖，

(4) 输出数组或标量参数，

(5) 工作数组及相应的规模参数，

(6) 返回信息 (INFO)。

值得注意的是并非每种参数都在一个子程序中出现。

2. 参数说明

下面的例子展示了 LAPACK 中参数的说明格式。

TRANS	(输入) 字符型；“'N'”表示对原矩阵进行操作，“'T'”表示对原矩阵的转置进行操作。
M	(输入) 整型；矩阵 A 的行数；M 或 -M 等于 A 的行数。
A	(输入/输出) 双精度实型；维数为 (LDA, N)；输入为 $A_{m\times n}$；如果 M>=0，输出时 A 被程序 DGEQRF 返回的 QR 分解覆盖，如果 M<0，输出时 A 被程序 DGELQF 返回的 LQ 分解覆盖。
LDA	(输入) 整型；矩阵 A 第一维的大小；LDA>=M。

INFO (输出) 整型；0 表示成功，<0 表示第 -INFO 个参数有违法值，>0 则表示 U(INFO，INFO) 的值为零，即三角分解可以完成但上三角矩阵 U 是奇异的，因而不能用以求解线性方程组。

每个参数的描述按如下顺序排列：

(1) 参数的分类：(输入)，(输出)，(输入/输出)，(输入或输出)，(工作数组)，(工作数组/输出)；

(2) 参数数据类型；

(3) 若参数是数组，其大小；

(4) 该参数所需的或将要提供的数据的描述，或二者都有。对于后者，在描述中以 “On entry” 和 “On exit” 开头；

(5) 若该参数是输入的标量，对其值的限制 (如上例中的 “LDA>=M”)。

3. 选项参数

一些选项的参数是 CHARACTER*1 型的，这样的选项在 LAPACK 中有 SIDE，TRANS，UPLO，DIAG。

SIDE 在调用时的用法如下：

输入值	含义
'L'	在矩阵的左边乘以一个对称或三角阵
'R'	在矩阵的右边乘以一个对称或三角阵

TRANS 在调用时的用法如下：

输入值	含义
'N'	对原矩阵进行操作
'T'	对原矩阵的转置进行操作
'C'	对原矩阵的共轭转置进行操作

UPLO 参数用于 Hermitian、对称和三角阵的情形，用于指示对矩阵的上或下三角进行操作，如下所示：

输入值	含义
'U'	上三角
'L'	下三角

DIAG 在对三角阵进行操作的程序中用于指明该三角阵是否对角线元素为 1，如下所示：

输入值	含义
'U'	单位三角阵
'N'	非单位三角阵

当 DIAG 被指定为 'U' 值时，对应的对角元素将不被引用。

以上参数也可以用小写字母，但任何其他值都是非法的。为了增强程序的可读性，程序员也可以使用更长的字符串，但只有第一个字符有效。例如：

```
CALL DPOTRS('Upper', ...)
```

4. 工作数组

很多 LAPACK 程序需要一个或多个工作数组作为参数。这种数组的名字一般是 WORK，有时可以是 IWORK 和 RWORK，以区别不同的数据类型。紧随其后的是声明工作数组大小的 LWORK、LIWORK 或 LRWORK 参数。工作数组的第一个元素总是返回为了完成计算任务所需的最小空间。若用户提供的工作数组不够大，则程序会赋值给 INFO 并把正确的数组大小存在 WORK(1) 中，最后调用 XERBLA 程序报错。因此建议用户最好每次都检查程序返回的 INFO 值。

若用户对于该工作组需要多大的空间有疑虑，不妨将 LWORK 设为 -1，然后进行调用，再把 WORK(1) 中返回的值作为 LWORK 的正确值。把 LWORK 设为 -1 不会引发任何错误信息，而是被作为一个查询请求来处理。

5. 错误处理

所有程序都返回错误指示信息 INFO，告诉用户计算成功或失败。因而推荐用户每次调用都检查返回的 INFO 值。INFO 值的定义如下所示：

INFO = 0 成功完成计算任务；

INFO < 0 一个或多个参数有错，无法进行计算；

INFO > 0 在计算过程中失败。

如果使用标准的 XERBLA 程序，当程序出错时，LAPACK 将打印一条错误信息，并且当 INFO<0 时终止程序运行，所以 LAPACK 的所有函数通常不会返回 INFO<0。但是，对于非标准的 XERBLA，这种情况则有可能发生。

A.2.4 LAPACK 使用示例

1. 解普通线性方程组

本节演示 LAPACK 计算子程序 DGETRF 和 DGETRS 的调用方法。方程的系数矩阵 A 和右端项矩阵 B 分别按下面的公式初始化：

$$A_{ij} = \sum_{k=1}^{\min(i,j)} (i+j), \qquad B_{ij} = \sum_{k=1}^{\min(i,j)} (1+j)/(i+k)$$

初始化完成后，调用 DGETRF 子程序对该矩阵进行 LU 分解。之后，调用 DGETRS 求解该方程组。

代码 A.1: 解普通线性方程组。

文件名: code/lapack/lapack1.f

```
1       PROGRAM TEST
2  * .. Scalar Arguments ..
3       INTEGER INFO, LDA, LDB, N, NRHS
4       PARAMETER ( N = 500, NRHS = 20, LDA = N, LDB = N )
```

```
* .. Array Arguments ..
      INTEGER IPIV( N )
      DOUBLE PRECISION A( LDA, N ), B( LDB, NRHS )
* .. External Subroutines ..
      EXTERNAL DGETRF, DGETRS
* .. Intrinsic Functions ..
      INTRINSIC MAX

* .. Executable Statements ..
*     Get the value of matrix
*     Matrix values are L = min(i,j), A_ij = sum_{1<=k<=L}(i+j)
      CALL INITMTRA(N, N, A, LDA)
* Compute the LU factorization of A
      CALL DGETRF( N, N, A, LDA, IPIV, INFO )
      IF( INFO.EQ.0 ) THEN
*        Generate the right hand side of linear equations
*        Matrix values are L = min(i,j), B_ij = sum_{1<=k<=L}(1+j)/(i+k)
         CALL INITMTRB(N, NRHS, B, LDB)
*        Solve the system A*X = B, overwriting B with X
         CALL DGETRS( 'No transpose', N, NRHS, A, LDA, IPIV, B, LDB,
     &                INFO )
      END IF
      STOP
      END

************************************************************************
* 初始化矩阵的子程序
************************************************************************

      SUBROUTINE INITMTRA( M, N, A, LDA )
* ..Scalar Arguments..
      INTEGER M, N, LDA
* ..Array Arguments..
      DOUBLE PRECISION A(LDA,*)
* ..Intrinsic Functions..
      INTRINSIC MIN
```

```
41 * ..Local Arguments..
42       INTEGER I, J, K
43
44       DO 30 J = 1, N
45       DO 20 I = 1, M
46          A(I,J) = 0.0
47          DO 10 K = 1, MIN(I,J)
48             A(I,J) = A(I,J) + I + J
49    10    CONTINUE
50    20 CONTINUE
51    30 CONTINUE
52       RETURN
53       END
54
55 ************************************************************************
56
57       SUBROUTINE INITMTRB( M, N, B, LDB )
58 * ..Scalar Arguments..
59       INTEGER M, N, LDB
60 * ..Array Arguments..
61       DOUBLE PRECISION B(LDB,*)
62 * ..Intrinsic Functions..
63       INTRINSIC MIN
64 * ..Local Arguments..
65       INTEGER I, J, K
66
67       DO 30 J = 1, N
68       DO 20 I = 1, M
69          B(I,J) = 0.0
70          DO 10 K = 1, MIN(I,J)
71             B(I,J) = B(I,J) + (1 + J) / (I + K)
72    10    CONTINUE
73    20 CONTINUE
74    30 CONTINUE
75       RETURN
```

```
76      END
```

2. QR **类型矩阵分解**

本节演示 LAPACK 计算子程序 `DGEQRF` 的调用方法。矩阵 `A` 按下面的公式初始化：

$$A_{ij} = \begin{cases} 4 & i = j = 1 \\ 5 & i = j \neq 1 \\ 3 & i = j + 1 \\ 0 & \text{其他} \end{cases}$$

完成初始化后调用 `DGEQRF` 子程序对该矩阵进行 QR 分解。

代码 A.2: QR 类型矩阵分解。

文件名: `code/lapack/lapack2.f`

```
1       PROGRAM TESTQRF
2   * .. Scalar Arguments ..
3       INTEGER INFO, LDA, LWORK, M, N, MN
4       PARAMETER (M = 500, N = 500, LDA = N)
5       PARAMETER (LWORK = N*256, MN = M)
6   * .. Array Arguments ..
7       DOUBLE PRECISION A(LDA, N), TAU(MN), WORK(LWORK)
8   * .. External Subroutines ..
9       EXTERNAL DGEQRF
10
11  * .. Executable Statements ..
12  * Get the value of matrix A
13      CALL INITMTRA(M, N, A, LDA)
14  * Compute QR factorization of A
15      CALL DGEQRF(M, N, A, LDA, TAU, WORK, LWORK, INFO)
16      STOP
17      END
```

```
18
19 ************************************************************************
20 * 初始化矩阵的子程序
21 ************************************************************************
22
23       SUBROUTINE INITMTRA(M, N, A, LDA)
24 * ..Scalar Arguments..
25       INTEGER M, N, LDA
26       DOUBLE PRECISION ZERO, THR, FOUR, FIVE
27       PARAMETER( ZERO = 0.0D0, THR = 3.0D0, FOUR = 4.0D0, FIVE = 5.0D0 )
28 * ..Array Arguments..
29       DOUBLE PRECISION A(LDA, *)
30 * ..Local Arguments..
31       INTEGER I, J
32
33       DO 20 J=1, N
34       DO 10 I=1, M
35          IF( I .EQ. J .AND. I .EQ. 1 )THEN
36             A(I, J) = FOUR
37          ELSE IF( I .EQ. J .AND. I .NE. 1 ) THEN
38             A(I, J) = FIVE
39          ELSE IF( I .EQ. J+1 ) THEN
40             A(I, J) = THR
41          ELSE
42             A(I, J) = ZERO
43          END IF
44    10 CONTINUE
45    20 CONTINUE
46       RETURN
47       END
```

3. 上 Hessenberg 矩阵化简

本节演示 LAPACK 计算子程序 DGEHRD。矩阵 A 的初始化与 QR 类型矩阵分解相同。完成初始化后调用 DGEHRD 子程序将该矩阵化

简为上 Hessenberg 矩阵。

代码 A.3: 上 Hessenberg 矩阵化简。

文件名: code/lapack/lapack3.f

```
      PROGRAM TESTBRD
* .. Scalar Arguments ..
      INTEGER ILO, IHI, INFO, LDA, LWORK, N
      PARAMETER ( N = 500, LDA = N, ILO = 1, IHI = N, LWORK = N*256 )
* .. Array Arguments ..
      DOUBLE PRECISION A( LDA, N ), TAU (N-1), WORK(LWORK)
* .. External Subroutines ..
      EXTERNAL DGEHRD

* .. Executable Statements ..
* Get the value of matrix A
      CALL INITMTRA(M, N, A, LDA)
* Reduce to upper Hessenberg form
      CALL DGEHRD(N, ILO, IHI, A, LDA, TAU, WORK, LWORK, INFO)
      STOP
      END

*************************************************************************
* 初始化矩阵的子程序 (同 QR 分解，略)
*************************************************************************
```

4. 三对角矩阵化简

本节演示 LAPACK 计算子程序 DSYTRD。矩阵 A 的初始化与 QR 类型矩阵分解相同。完成初始化后调用 DSYTRD 子程序将该矩阵化简为三对角矩阵。

代码 A.4: 三对角矩阵化简。

文件名: code/lapack/lapack4.f

```
      PROGRAM TESTTRD
```

```
* .. Scalar Arguments ..
      CHARACTER*1 UPLO
      INTEGER INFO, LDA, LWORK, N
      PARAMETER ( UPLO = 'U', N = 500, LDA = N, LWORK = N*256 )
* .. Array Arguments ..
      DOUBLE PRECISION A(LDA, N), D(N), E(N-1), TAU(N-1), WORK(LWORK)
* .. External Subroutines ..
      EXTERNAL DSYTRD

* .. Executable Statements ..
* Get the value of matrix A
      CALL INITMTRA(M, N, A, LDA)
* Call DSYTRD to reduce symmetric matrix to tridiagonal form
      CALL DSYTRD(UPOL, N, A, LDA, D, E, TAU, WORK, LWORK, INFO)
      STOP
      END

**********************************************************************
* 初始化矩阵的子程序 (同 QR 分解，略)
**********************************************************************
```

5. 对称特征值问题

本节演示 LAPACK 计算子程序 DSTEQR 和 DSTERF。矩阵 A 按下面公式初始化：

$$\begin{cases} A_{ii} = 1/(i+1) & i = 1, \ldots, N \\ A_{ij} = A_{ji} = 1/(i+j) & i = j+1 \text{ 或 } i = j-1 \\ A_{ij} = 0 & \text{其他} \end{cases}$$

程序中主对角线元素存储在数组 D 中，次对角线元素存储在数组 E 中。完成初始化后首先调用 DSTEQR 子程序求该矩阵的特征值，然后调用 DSTERF 求解特征向量。

代码 A.5: 对称特征值问题。

文件名: code/lapack/lapack5.f

```
      PROGRAM TESTEIG
* .. Scalar Arguments ..
      CHARACTER*1 JOBZ
      INTEGER INFO, LDZ, N, NN
      PARAMETER ( JOBZ = 'V', N = 500, LDZ = N+1, NN = 2*N-2 )
* .. Array Arguments ..
      DOUBLE PRECISION D( N ), E( N ), WORK( NN ), Z( LDZ, N )
* .. Parameters ..
      DOUBLE PRECISION ZERO, ONE
      PARAMETER ( ZERO = 0.0D0, ONE = 1.0D0 )
* .. Local Scalars ..
      LOGICAL WANTZ
      INTEGER IMAX, ISCALE
      DOUBLE PRECISION BIGNUM, EPS, RMAX, RMIN, SAFMIN,
     &                 SIGMA, SMLNUM, TNRM
* .. External Functions ..
      LOGICAL LSAME
      DOUBLE PRECISION DLAMCH, DLANST
      EXTERNAL LSAME, DLAMCH, DLANST
* .. External Subroutines ..
      EXTERNAL DSCAL, DSTEQR, DSTERF
* .. Intrinsic Functions ..
      INTRINSIC SQRT

* .. Executable Statements ..
      WANTZ = LSAME( JOBZ, 'V' )
* Quick return if possible
      IF( N.EQ.0 ) RETURN
      IF( N.EQ.1 ) THEN
         IF( WANTZ ) Z( 1, 1 ) = ONE
         RETURN
      END IF
* Get machine constants
```

```
      SAFMIN = DLAMCH( 'Safe minimum' )
      EPS = DLAMCH( 'Precision' )
      SMLNUM = SAFMIN / EPS
      BIGNUM = ONE / SMLNUM
      RMIN = SQRT( SMLNUM )
      RMAX = SQRT( BIGNUM )
* Get the value of matrix A
      CALL INITMTRA( N, D, E )
* Scale matrix to allowable range, if necessary
      ISCALE = 0
      TNRM = DLANST( 'M', N, D, E )
      IF( TNRM.GT.ZERO .AND. TNRM.LT.RMIN ) THEN
         ISCALE = 1
         SIGMA = RMIN / TNRM
      ELSE IF( TNRM.GT.RMAX ) THEN
         ISCALE = 1
         SIGMA = RMAX / TNRM
      END IF
      IF( ISCALE.EQ.1 ) THEN
         CALL DSCAL( N, SIGMA, D, 1 )
         CALL DSCAL( N-1, SIGMA, E( 1 ), 1 )
      END IF
* For eigenvalues only, call DSTERF. For eigenvalues and
* eigenvectors, call DSTEQR
      IF( .NOT.WANTZ ) THEN
         CALL DSTERF( N, D, E, INFO )
      ELSE
         CALL DSTEQR( 'I', N, D, E, Z, LDZ, WORK, INFO )
      END IF
* If matrix was scaled, then rescale eigenvalues appropriately
      IF( ISCALE.EQ.1 ) THEN
         IF( INFO.EQ.0 ) THEN
            IMAX = N
         ELSE
            IMAX = INFO - 1
         END IF
```

```
70          CALL DSCAL( IMAX, ONE / SIGMA, D, 1 )
71        END IF
72        STOP
73        END
74
75  ************************************************************************
76  * 初始化矩阵的子程序
77  ************************************************************************
78
79        SUBROUTINE INITMTRA( N, D, E )
80  * ..Scalar Arguments..
81        INTEGER N
82  * ..Array Arguments..
83        DOUBLE PRECISION D( N ), E( N )
84  * ..Local Arguments..
85        INTEGER I, ONE
86        PARAMETER ( ONE = 1 )
87  * .. Intrinsic Functions ..
88        INTRINSIC DBLE
89
90        DO 10 I = ONE, N
91           D(I) = DBLE(ONE)/(DBLE(ONE)+I)
92     10 CONTINUE
93        DO 20 I = ONE, N-1
94           E(I) = DBLE(ONE)/(I+J)
95     20 CONTINUE
96        RETURN
97        END
```

A.3 ScaLAPACK

ScaLAPACK (Scalable LAPACK) 是美国能源部 DOE2000 支持开发的 20 多个 ACTS 工具箱之一，由 Oak Ridge 国家实验室、加州大学 Berkeley 分校和 Illinois 大学等联合开发。它是 LAPACK 在分布

式存储环境中的扩展，主要运行在基于分布式存储和消息传递机制的 MIMD 计算机以及支持 PVM 或 MPI 的机群上。LAPACK 是适用于向量超级计算机、共享式存储并行计算机和各种单机上的线性代数运算程序包，ScaLAPACK 实现了其功能的一个子集。ScaLAPACK 的名称来源于 Scalable Linear Algebra PACKage 或 Scalable LAPACK 的缩写。ScaLAPACK 被设计为具有高效率、可移植性、可扩展性、可靠性、灵活性和易用性。

ScaLAPACK 可以求解线性方程组、线性最小二乘问题、特征值和奇异值问题。同时它也可以处理许多相关问题，如矩阵分解和估计条件数。它只适用于稠密矩阵和带状矩阵，对于普通稀疏矩阵不适用。与 LAPACK 类似，ScaLAPACK 也是基于块划分算法以减少进程间的通信。ScaLAPACK 的基本组成部分是 PBLAS (并行 BLAS) 和 BLACS。PBLAS 是 1、2、3 级 BLAS 的分布式存储版本；BLACS 则负责实现并行线性代数计算中常用的通信。在 ScaLAPACK 程序中，多数通信发生在 PBLAS 中，所以 ScaLAPACK 的顶层软件源代码看起来和 LAPACK 相似。

ScaLAPACK 使用基于 SPMD 模型的 Fortran 77 编程，采用显式消息传递进行通信。PBLAS 和 BLACS 使用 C 语言编程，但是具有 Fortran 77 接口。ScaLAPACK 程序提供四个版本，分别对应于单精度和双精度、实数和复数计算。

ScaLAPACK 和 LAPACK 已经成为进行线性代数运算的事实标准而被研究领域和工业界广泛使用，目前已经被集成到许多商业软件包中，包括 NAG Parallel Library，IBM Parallel ESSL，Cray LIBSCI，VNI IMSL Numerical Library，以及 Fujitsu，HP/Convex，Hitachi 和 NEC 计算机的软件库。截止到本书成稿时，ScaLAPACK 的最新版本为 1.7 版，于 2001 年 8 月发布。ScaLAPACK 的网站为

`http://www.netlib.org/scalapack`

国内镜像为

http://netlib.amss.ac.cn/scalapack

A.3.1 ScaLAPACK 体系结构

1. 软件组成

ScaLAPACK 软件的层次结构如图 A.2 所示。虚线下部标注为本地，本地组件在一个进程中被调用，其参数只存储在一个进程中。虚线上部标注为全局，全局组件是同步的并行程序，其参数，包括矩阵和向量，分布在多个进程中。

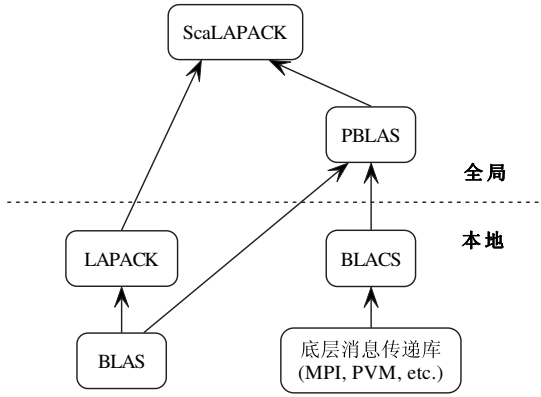


图 A.2 ScaLAPACK 软件的层次结构

LAPACK 线性代数程序库。关于 LAPACK 的介绍可参看 A.2。

BLAS 基本线性代数子程序库。BLAS 的一个重要目标是为基本线性代数计算提供可移植层。关于 BLAS 的介绍可参看 A.1。

PBLAS 并行 BLAS，执行消息传递并且其接口与 BLAS 尽可能相似。它简化了 ScaLAPACK 的设计，使得 ScaLAPACK 的代码与相应的 LAPACK 代码相当接近，有时甚至几乎一样。

BLACS 基本线性代数通信子程序库，是为线性代数设计的消息传递库。计算模型由一个一维或二维进程网格构成，每个进程存储矩阵和向量的一些片段。BLACS 包括点对点通信程序和聚合通信程序，也有构造、改变和查询进程网格的程序。同时，BLACS 具有上下文 (context) 的概念，对应于 MPI 中的通信器 (communicator)，它提供了分隔消息传递域的能力。BLACS 的重要目标是为通信提供专用于线性代数的可移植层。

2. 程序等级

与 LAPACK 类似，ScaLAPACK 中的程序也分为三个大类：

(1) 驱动程序 (driver routines)：用于求解标准类型的问题，例如，求解线性方程组和计算实对称矩阵的特征值。每个驱动程序调用一系列计算程序，并在可能时对程序进行全局和本地输入错误检查。

(2) 计算程序 (computational routines)：用于执行特定的计算任务，例如 LU 分解或将实对称矩阵化简为三对角形式。对程序进行全局和本地输入错误检查。作为整体，计算程序比驱动程序可以承担的任务范围更广。

(3) 辅助程序 (auxiliary routines)：又分为两类：一类是负责完成矩阵、向量分块后子块计算的程序，特别是实现了不分块版本算法的程序；另一类是执行一些常用底层计算的程序，例如缩放矩阵、计算矩阵范数、或生成初等 Householder 矩阵，将来它们会被考虑加入到 PBLAS 中。一般来说，在辅助程序中不进行输入错误检查，其例外是对于等同于第 2 级计算程序的辅助程序进行本地输入错误检查。

3. 命名规则

对于与 LAPACK 相对应的 ScaLAPACK 程序的名称，只是简单地在 LAPACK 名称前面加一个 “P”。因此 Fortran 77 的要求被放松 (破坏) 了，即允许子程序名称长度多于 6 个字符，在特定的 TOOLS 程序名称中还允许出现下划线 “_”。

与 LAPACK 相似，所有驱动程序和计算程序都有 PXYYZZZ 形式的名称，其中有些驱动程序的第七个字母为空。第二个字母 X 表示数据类型；后两个字母 YY 指明矩阵 (或最重要矩阵) 的数据类型；最后三个字母 ZZZ 指明执行的计算。辅助程序的名字遵从相似的规则，只是第三和第四个字母 YY 通常是 LA。

A.3.2 ScaLAPACK 程序介绍

1. 驱动程序：线性方程组求解

求解如下形式的线性方程组：

$$Ax = b \tag{A.1}$$

其中 A 是系数矩阵，b 是右端向量，x 是解向量。如果有多个右端向量，则可以写成

$$AX = B \tag{A.2}$$

其中 B 的列是独立的右端向量，X 的列是相应的解向量。

ScaLAPACK 提供了两类驱动程序用以求解线性方程组：简单驱动（名字以 SV 结尾）和专家驱动 (名字以 SVX 结尾)。简单驱动通过分解 A 求解方程组 $AX = B$，并用 X 覆盖 B。专家驱动还有一些额外的功能，例如：求解 $A^{\mathrm{T}}X = B$ 或 $A^{\mathrm{H}}X = B$；估计 A 的条件数，检查近奇异性，检查主元增长；改进解，计算向前和向后误差范围；在 A 中元素的数量级差别很大时，采用平衡方法来降低其条件数。

为了利用矩阵 A 的特性和存储方案，ScaLAPACK 提供了不同的驱动程序。它们几乎覆盖了线性方程组求解计算的全部功能，除了很少用到的矩阵求逆。

目前，ScaLAPACK 对于涉及带状和三对角矩阵的线性方程组只提供了简单驱动。应当注意的是在这些驱动中使用的带状和三对角分解得到的结果与 LAPACK 中同样分解得到的结果不同。为了并行的目的，ScaLAPACK 在矩阵中进行了额外的置换。

2. 驱动程序：线性最小二乘问题

求解线性最小二乘问题 (LLS)：

$$\min_x \|b - Ax\|_2 \tag{A.3}$$

其中 A 是一个 $m \times n$ 维矩阵，b 是一个给定的 m 维向量，x 是 n 维解向量。

驱动程序 `PxGELS` 在求解时假设 A 是满秩的，即 $\mathrm{rank}(A) = \min(m, n)$，它使用 A 的 QR 或 LU 分解进行计算，并且也可以求解关于 A^{T} 或 A^{H} 的问题。当 $m < n$ 时解是不唯一的，此时程序计算范数最小的解。

所有驱动程序都允许在同一次调用中处理多个右端向量 b 和相应的解 x，将这些向量分别存储为矩阵 B 和 X 的列。方程 (A.3) 对于每个右边向量独立求解，这与找到一个矩阵 X 使 $\|B - AX\|_2$ 最小是不同的。

3. 驱动程序：标准特征值和奇异值问题

对称特征值问题 (SEP) 是要找到特征值 λ 和相应的特征向量 $z \neq 0$，满足 $Az = \lambda z$，其中 A 是实对称矩阵或复 Hermitian 矩阵，对于这种情况 λ 是实数。当所有特征值和特征向量被计算出之后，可以写出 A 的经典谱分解 $A = Z\Lambda Z^{\mathrm{T}}$，其中 Λ 是以特征值为对角元素的对角矩阵，Z 为正交矩阵 (或酉矩阵)，它的列为特征向量。

对于对称或 Hermitian 特征值问题，ScaLAPACK 提供了简单驱动和专家驱动两类程序。简单驱动只能计算所有的特征值和特征向量，而专家驱动则可以选择要计算的特征值的子集和相应的特征向量。

一个 $m \times n$ 维矩阵 A 的奇异值分解 (SVD) 由如下公式给出：

$$A = U\Sigma V^{\mathrm{T}}, \text{(在复数情况下是} A = U\Sigma V^{\mathrm{H}}\text{)} \tag{A.4}$$

其中 U 和 V 是正交 (酉) 阵，Σ 是 $m \times n$ 的对角矩阵，具有实对角元素 σ_i，满足 $\sigma_1 \geqslant \sigma_2 \geqslant \ldots \geqslant \sigma_{\min(m,n)} \geqslant 0$。$\sigma_i$ 是 A 的奇异值，U、V 的前 $\min(m,n)$ 列分别是 A 的左、右奇异向量。奇异值和奇异向量满足

$$Av_i = \sigma_i u_i \text{ 以及 } A^{\mathrm{T}}u_i = \sigma_i v_i\text{(或者 } A^{\mathrm{H}}u_i = \sigma_i v_i\text{)} \tag{A.5}$$

其中 u_i 和 v_i 分别是 U 和 V 的第 i 列。

4. 驱动程序：广义对称正定特征值问题 (GSEP)

ScaLAPACK 提供了一个专家驱动对于以下类型的问题计算所有特征值和特征向量：

$$Az = \lambda Bz \tag{A.6}$$

$$ABz = \lambda z \tag{A.7}$$

$$BAz = \lambda z \tag{A.8}$$

其中 A 和 B 是对称矩阵或 Hermitian 矩阵并且 B 是正定的。对于上述所有问题特征值 λ 均为实数。

5. 计算程序：线性方程组

求解公式 (A.1) 和 (A.2) 中的联立线性方程组，并假设 A 为 n 阶方阵，但是有些单独的程序允许 A 不是方阵。ScaLAPACK 在可

能的情况下为每种类型矩阵的每个不同存储方案都提供了不同的程序。

6. 计算程序：正交分解和最小二乘问题

ScaLAPACK 提供了许多程序用于分解普通 $m \times n$ 维矩阵 A 为正交矩阵 (对于复数是酉矩阵) 和三角矩阵 (可能是梯形阵) 的乘积，包括 QR 分解、LQ 分解、RQ 分解、QL 分解、RZ 分解等。正交矩阵或酉阵的一个重要特性是它们不改变向量的二范数：$\|x\|_2 = \|Qx\|_2$，因此有助于保持计算的数值稳定性，因为它们不放大舍入误差。正交分解用于线性最小二乘问题的求解，它们也可用于在求解特征值或奇异值问题时将矩阵分解。

7. 计算程序：广义正交分解

ScaLAPACK 提供了两个程序分别求解 $n \times m$ 维矩阵 A 和 $n \times p$ 维矩阵 B 的广义 QR 分解 (GQR) 以及 $m \times n$ 维矩阵 A 和 $p \times n$ 维矩阵 B 的广义 RQ 分解 (GRQ)，它们分别由一对分解给出

$$A = QR \quad 和 \quad B = QTZ \tag{A.9}$$

$$A = RQ \quad 和 \quad B = ZTQ \tag{A.10}$$

其中 Q 和 Z 分别为 $n \times n$ 维和 $p \times p$ 维的正交矩阵（或酉矩阵，如果 A 和 B 为复矩阵）。

8. 计算程序：对称特征值问题

A 是 $n \times n$ 维实对称或复 Hermitian 矩阵。如果 $Az = \lambda z$，λ 为特征值，非零列向量 z 则是相应的特征向量。不论 A 是实对称矩阵还是复 Hermitian 矩阵，λ 总是实数。对称特征问题程序的基本任务是计算给定矩阵 A 的 λ 的值和 (可选的) 相应特征向量 z。

9. 计算程序：非对称特征值问题

A 是 $n \times n$ 维方阵。如果 $Av = \lambda v$，则标量 λ 是特征值，非零列向量 v 是相应的右特征向量，而满足 $u^{\mathrm{H}}A = \lambda u^{\mathrm{H}}$ 的非零列向量 u 是左特征向量。该程序的基本任务是计算给定矩阵 A 的所有 n 个特征值，如果需要，也可以计算相应的右特征向量 v 和左特征向量 u。另一个基本任务是计算矩阵 A 的 Schur 分解。

10. 计算程序：奇异值分解

A 为 $m \times n$ 普通实矩阵。A 的奇异值分解 (SVD) 为，$A = U\Sigma V^{\mathrm{T}}$，其中 U 和 V 为正定阵，$\Sigma = \mathrm{diag}\{\sigma_1, \ldots, \sigma_r\}$，$r = \min(m, n)$，$\sigma_1 \geqslant \ldots \geqslant \sigma_r \geqslant 0$。如果 A 为复数，其奇异值分解为 $A = U\Sigma V^{\mathrm{H}}$，其中 U 和 V 为酉阵，Σ 和前面一样具有实对角元素。σ_i 称为奇异值，V 的前 r 列为右奇异向量，V 的前 r 列为左奇异向量。

11. 计算程序：广义对称正定特征值问题

计算广义特征值问题 $Az = \lambda Bz$，$ABz = \lambda z$，以及 $BAz = \lambda z$，其中 A 和 B 是实对称矩阵或复 Hermitian 矩阵，B 是正定的。这些问题中的每一个都可以用 B 的 Cholesky 分解化简为 $Cy = \lambda y$ 的标准对称特征值问题。

12. 数据分布

ScaLAPACK 要求所有全局数据 (向量或矩阵) 在调用 ScaLAPACK 程序前被分布到相关进程上。应用程序的数据在并行计算机的分级存储器结构中的布局对于决定并行代码的性能和可扩展性是非常关键的。

ScaLAPACK 采用块划分算法，并且尽可能使用面向块的第 3 级 BLAS 矩阵—矩阵运算来实现。这种方法可使浮点操作和内存访问的比例最大化，并尽可能重用存储在分级存储器结构的最高级存

储器中的数据，从而降低了数据在进程间传送的频率，及每次通信的启动开销 (延迟)。

用于求解稠密线性系统和特征值问题的 ScaLAPACK 程序假设所有全局数据都已经使用一维或二维块轮转数据分布方式分布到相关进程上。这种分布正是 ScaLAPACK 使用块划分算法的自然表达。用于求解窄带状线性系统和三对角系统的 ScaLAPACK 程序假设所有全局数据都已经使用一维块数据分布方式分布到进程上。

A.3.3 ScaLAPACK 安装

完整的 ScaLAPACK 软件包可以通过其在 Netlib 上的主页免费获得，也可以通过 WWW 或匿名 FTP 得到。ScaLAPACK 的参考源代码可以通过以下网址：

`http://www.netlib.org/scalapack/scalapack.tgz`

或国内镜像：

`http://netlibamss.ac.cn/scalapack/scalapack.tgz`

获得。用户可以通过编译参考源代码获得自己 ScaLAPACK 程序库。编译的方法将在稍后介绍。

同时，在 Netlib 也有一些已经编译好的 ScaLAPACK 库，分别适用于不同的计算机平台，包括 Cray T3E、Intel Paragon、IBM SP-2、SGI Origin 2000、DEC ALPHA、HP 9000、以及 Intel/Linux 等等。这些预编译好的库可以通过以下网址获得：

`http://www.netlib.org/scalapack/archives/`

`http://netlib.amss.ac.cn/scalapack/archives/`

安装 ScaLAPACK，要求系统中已经安装了 BLAS 和 BLACS 库。其中 BLAS 的安装可以参考 A.1，而 BLACS 则可以从它的主页

`http://www.netlib.org/blacs/index.html`

或国内镜像

`http://netlib.amss.ac.cn/blacs/index.html`

下载针对不同平台的源代码或者已经编译好的库。

获取了参考源代码之后，安装 ScaLAPACK 主要有以下四个步骤：

(1) 解压缩源代码打包文件 `scalapack.tgz`；

(2) 编辑文件 `SLmake.inc`，指定 MPI、BLAS、BLACS 各个库的位置；

(3) 编辑顶层的 `Makefile` 文件，然后键入 `make` 命令进行编译；

(4) 运行测试文件集。

1. 解压缩源代码

在源代码的打包文件中包括了 ScaLAPACK 的源文件、PBLAS 的源文件以及它们的测试文件集。可以用以下命令解压缩打包文件：

```
gunzip -c scalapack.tgz | tar xvf -
```

解压缩之后所有的文件都放入 `SCALAPACK` 目录中，产生的目录结构如图 A.3 所示：

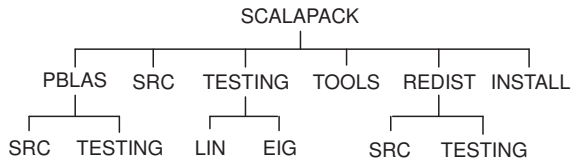


图 A.3 ScaLAPACK 软件的目录

ScaLAPACK 的源代码在 `SRC` 目录中，`PBLAS` 目录下是 PBLAS 的源代码和测试文件集，`TESTING` 目录下是 ScaLAPACK 的测试文件集。

2. 编辑 SLmake.inc 文件

在 SCALAPACK/INSTALL 目录中包含特定平台的 SLmake.inc 文件示例，包括 Intel i860、IBM SP、Cray T3E、SGI Origin、以及使用 MPI 和 PVM 的各种工作站和微机机群。首先将其中相应平台的示例文件拷贝到 SCALAPACK/SLmake.inc 文件中，例如在 Linux 平台上编译 ScaLAPACK，使用如下的命令：

```
$ cp INSTALL/SLmake.LINUX SLmake.inc
```

然后编辑 SLmake.inc，指定以下一些内容：

(1) 指定 ScaLAPACK 顶层目录所在的完整路径，称为 HOME。

(2) 确定需要安装库的平台。

(3) 指定所使用的编译器，链接器，编译、链接选项，库文件打包选项：CC，F77，NOOPT，CCFLAGS，F77FLAGS，LOADER，LOADFLAGS，ARCH，ARCHFLAGS 和 RANLIB。

(4) 指定调试和预处理选项：BLACSDBGLVL 和 CDEFS。BLACSDBGLVL 可以取 0 和 1，它表示 BLACS 的调试级别。CDEFS 可以是 -DAdd_，-DNoChange 或 -DUPCASE。

(5) 指定所需要的库文件的位置，包括：BLACS、MPI 或 PVM 以及 BLAS。

3. 编辑顶层 Makefile 并编译

在顶层目录中已经包括了编译建立库文件和所有测试执行文件的 Makefile 文件，这个文件一般不需要编辑。在编辑好 SLmake.inc 文件后，想要编译生成 ScaLAPACK 库文件，只需要简单地执行以下两个命令：

```
make
```

如果 SLmake.inc 文件中的各项都定义正确，经过几分钟的运行之后，库文件 libscalapack.a 就出现在顶层目录中了。

如果希望建立测试执行文件，可以使用

```
make exe
```

完成后，所有的测试执行文件存放在 TESTING 目录下。

如果只希望建立部分的库或测试执行文件，可以编辑 Makefile 文件，改变 lib 或 exe 的定义。要指定希望使用的数据类型，需要改变 PRECISIONS 的定义。在默认情况下，它的定义如下

```
PRECISIONS = single double complex complex16
```

它表示对于所有数据类型都进行编译，其中，single 指定单精度实型，double 指定双精度实型，complex 指定单精度复型，complex16 指定双精度复型。

当所有测试执行文件都运行完成之后，可以使用

```
make clean
```

移除全部目标文件和测试执行文件。也可以使用 make cleanlib 和 make cleanexe 分别移除库的目标文件或测试程序的目标文件和执行文件。

4. 运行测试文件集

在 ScaLAPACK 的源程序包中带有三类测试程序：PBLAS 测试、REDIST 测试、以及 ScaLAPACK 测试。它们可以用来检查生成的 ScaLAPACK 库的正确性及其性能。每个测试都有一个输入文件，用以指定矩阵规模、分块大小、进程网格尺寸等参数。PBLAS 测试包括了对于全部 3 级 PBLAS 程序的检查和计时；REDIST 测试是对于矩阵在任意进程网格间使用任意分块大小进行二维块轮转重分布的

检查；一共有 18 个 ScaLAPACK 测试程序分别进行 LU、Cholesky、带状 LU、带状 Cholesky、普通三对角、带状三对角、QR（RQ、LQ、QL、QP 和 TZ）、线性最小二乘、上 Hessenberg 化简、三对角化简、双对角化简、矩阵求逆、对称特征值问题、广义对称特征值问题、非对称特征值问题和奇异值问题的计算。

A.3.4 ScaLAPACK 编程指南

1. 调用 ScaLAPACK 程序

在用户自己的程序中调用 ScaLAPACK 程序需要四个基本步骤：

(1) 初始化进程网格

一台抽象的并行计算机的 P 个进程可以表示为一维数组，通常将这个一维数组映射到二维矩形网格可以更方便地表示算法，这个网格称为进程网格。在程序的开始用户需要初始化进程网格，得到默认的系统上下文。用户也可以查询进程网格以识别每个进程的坐标。

调用 ScaLAPACK TOOLS 程序 SL_INIT 初始化进程网格。这个程序使用进程行优先排序初始化一个 $P_r \times P_c$（在源代码中以 NPROW × NPCOL 表示）进程网格，得到默认的系统上下文。用户可以通过调用 BLACS_GRIDINFO 查询进程网格以识别每个进程的坐标 (MYROW, MYCOL)。

完成这项任务的典型代码如下：

```
CALL BLACS_GET( -1, 0, ICTXT )
CALL BLACS_GRIDINIT( ICTXT, 'Row-major', NPROW, NPCOL )
CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
```

其中，BLACS_GET 获得默认上下文 ICTXT；BLACS_GRIDINIT 定

义进程网格，'Row-major' 说明网格是按照行优先的顺序排列的；BLACS_GRIDINFO 获得本进程在进程网格中的位置信息。

(2) 将矩阵分布到进程网格上

在调用 ScaLAPACK 程序之前所有全局矩阵必须分布在进程网格上。执行数据分布是用户的责任。每个要分布在进程网格上的全局矩阵都必须分配一个数组描述符，数组描述符通过调用 ScaLAPACK TOOLS 的程序 DESCINIT 可以很简单地初始化，它必须在调用 ScaLAPACK 程序之前设置。

矩阵的数组描述符用以下代码分配：

```
 CALL DESCINIT( DESCA, M, N, MB, NB, RSRC, CSRC, ICTXT,
$               MXLLDA, INFO )
 CALL DESCINIT( DESCB, N, NRHS, NB, NBRHS, RSRC, CSRC,
$               ICTXT, MXLLDB, INFO )
 CALL DESCINIT( DESCX, N, NRHS, NB, NB, 0, 0, ICTXT,
$               MAX(1, NP), INFO )
```

数组描述符各项的详细描述可以在 [30] 中找到。

之后可以使用如下的代码调用从数据文件中读入矩阵的数据并将其分布到进程网格上：

```
 CALL PDLAREAD( 'SCAEXMAT.dat', MEM(IPA), DESCA, 0, 0,
$               MEM(IPW) )
 CALL PDLAREAD( 'SCAEXRHS.dat', MEM(IPB), DESCB, 0, 0,
$               MEM(IPW) )
```

(3) 调用 ScaLAPACK 程序

当数据正确分布到进程网格上之后，用户就可以调用相应的驱动、计算、辅助程序进行需要的计算。各个程序的调用参数的详细解释可以在 ScaLAPACK 源代码的注释中找到。

下面的例子调用求解线性方程组 $AX = B$ 的简单驱动程序 PDGESV：

```
      CALL PDGESV( N, NRHS, MEM(IPA), 1, 1, DESCA, MEM(IPPIV),
     $             MEM(IPB), 1, 1, DESCB, INFO )
```

(4) 释放进程网络

在进程网格上执行完计算后，通过调用 BLACS_GRIDEXIT 释放进程网格。当所有计算完成后，程序通过调用 BLACS_EXIT 退出。

完成这个步骤的典型代码是：

```
      CALL BLACS_GRIDEXIT( ICTXT )
      CALL BLACS_EXIT( 0 )
```

具体的程序调用请参阅《ScaLAPACK 用户手册》[30]。

在编译链接程序时，需要指定 libscalapack.a 文件的位置，确保程序可以被正确链接。

2. 使用 ScaLAPACK 获得高性能的原则

ScaLAPACK 被设计为在分布式存储计算机上运行。一般来说，分布式存储计算机包括高效的消息传递系统、进程到处理机的一对一映射、成组调度程序和精心链接的通信网络。为了在分布式存储计算机上获得高性能，在编写和运行 ScaLAPACK 程序时，需要遵循以下一些原则：

(1) 使用正确数量的进程

根据经验，使用的进程数量要与所计算的问题规模相适应，[30] 中的建议为进程数量 NP 可以由经验公式

$$\mathtt{NP} = \mathtt{M} \times \mathtt{N}/1000000 \tag{A.11}$$

大致估计，即使得每个进程中本地子矩阵的规模大约为 1000×1000。需要指出的是，这个建议是在 1996–1997 年间基于 ScaLAPACK 1.4 版本的测试而做出的，现在随着处理器速度的提高和存储器的加大，本地子矩阵的规模也需要根据使用 ScaLAPACK 测试程序得到的结果做出相应调整。需要注意的是本地子矩阵不要过小而占用过多的进程使得通信时间加长，也不要将本地子矩阵划分得过大而超过物理存储器的容量。

(2) 使用高效的数据分布

进行数据分布时，使用分块尺寸 `MB` = `NB` = 64，当然这个参数是与使用的 BLAS 库相关的，不同的 BLAS 库在不同的环境中有不同的最佳的分块，在 [34] 中得到的最优分块是 Intel MKL 库的分块大小为 64 的倍数，ATLAS 生成的 BLAS 库的分块大小为 40 的倍数。

进程网格通常要尽量接近正方形。进程网格尺寸由两个参数决定，P 表示水平方向进程数量，Q 表示垂直方向进程数量。一般情况下需要使 $P = Q$ 或者 P 略小于 Q。

(3) 使用对于特定平台优化的 BLAS 和 BLACS 库

BLAS 和 BLACS 是 ScaLAPACK 的两个基本组件，它们的性能决定了 ScaLAPACK 的效率，因此要选择在特定平台上效率最高的 BLAS 和 BLACS。

除了以上这些大的原则之外，还有一些其他的问题需要考虑，包括：保证每结点的带宽，网络的延迟不能过大，使用同构的工作站或微机机群，正在使用的处理机上不要有其他任务，每个处理机上运行的进程数不要多于 CPU 数。其中有些问题是普通用户无法控制的。

A.4 FFTW

FFTW (The Fastest Fourier Transform in the West) 是一个免费的快速富氏变换 (FFT) 软件包，开发者是麻省理工学院的 Matteo Frigo 和 Steven G. Johnson，可从站点 `http://www.fftw.org`下载。该软件包用 C 语言开发，其核心技术 (编码生成器) 采用面向对象设计技术和面向对象语言 Caml 编写。FFTW 能自动适应系统硬件，因而可移植性很强，用户无须对系统干预太多。它能计算一维和多维离散傅立叶变换（Discrete Fourier Transform），其数据类型可以是实型、复型或半复型。该软件通过方案 (plan) 和执行器 (executor) 与用户进行接口，内部结构及其复杂性对用户透明，速度快。内部编译器、代码生成器利用 AST（Abstract Syntax Tree）在运行时生成适合所运行的机器的代码并自我优化。FFTW 为指定的变换生成一个方案，通过执行方案完成变换。它的运算性能远远领先于许多其它 FFT 软件，受到越来越多的科学研究和工程计算工作者的青睐。这里介绍的是 FFTW 3.0.1 版的基本用法，与 FFTW 2 相比，FFTW 3 在调用接口上发生了比较大的变化。除非特别提及，本书中 “FFTW” 一律指 FFTW 3[1]。所用到的材料来源于 `http://www.fftw.org/fftw3_doc/`，目的是给读者提供一个快速入门参考，详细说明请自行参考原始文档。

FFTW 实现了多种类型的变换，包括复型变换、实型变换、sin 变换、cos 变换和 Hartley 变换。在调用接口方面，FFTW 提供了基本接口和高级接口，后者提供了对 FFTW 更精细的控制。对于 Fortran 程序，FFTW 亦提供了相应的 Fortran 接口。这里仅以复型和实型变换函数为例给出 FFTW 的基本 C 接口函数。

[1]FFTW 2 支持共享存储多线程并行和分布式存储 MPI 并行，而 FFTW 3 目前只支持共享存储多线程并行，因此使用 MPI 并行的程序可能仍然需调要用 FFTW 2 的 MPI 函数。

FFTW 默认使用 double 型数据，并将复数定义为如下类型：

```
typedef double fftw_complex[2];
#define c_re(c) ((c)[0])
#define c_im(c) ((c)[1])
```

如果使用符合 C99 标准的 C 编译器，并且在 FFTW 的头文件之前包含了 complex.h 文件，则 fftw_complex 将被定义为 C 的默认复数类型 (如 complex<double>，这种情况下可以直接在代码中使用复数表达式)。

FFTW 的基本使用包括三个步骤：为特定的变换创建方案、(反复) 执行方案完成变换和释放方案。典型调用过程如下 (以三维复变换为例)：

```
#include <fftw3.h>
...
{
    fftw_complex *in, *out;     /* 变换数组 */
    fftw_plan p;                /* 方案 */
    ...
    /* 申请内存及创建方案 */
    in = fftw_malloc(sizeof(fftw_complex) * nx * ny * nz);
    out = fftw_malloc(sizeof(fftw_complex) * nx * ny * nz);
    p = fftw_plan_dft_3d(nx, ny, nz, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
    ...
    /* 实施变换 (可反复执行) */
    fftw_execute(p);
    ...
    /* 释放方案、数组 */
    fftw_destroy_plan(p);
    fftw_free(in);
    fftw_free(out);
}
```

注意这里调用 fftw_malloc 函数为数组申请存储空间，它与 malloc

的区别是前者会根据 FFTW 的要求在需要时调整数组的对界。

A.4.1 复型变换

计算如下公式：

(1) 向前变换：

$$y[i_1,i_2,\ldots,i_d]=\sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1}\cdots\sum_{j_d=0}^{n_d-1}x[j_1,j_2,\ldots,j_d]\omega_1^{-i_1j_1}\omega_2^{-i_2j_2}\cdots\omega_d^{-i_dj_d} \tag{A.12}$$

(2) 向后变换：

$$y[i_1,i_2,\ldots,i_d]=\sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1}\cdots\sum_{j_d=0}^{n_d-1}x[j_1,j_2,\ldots,j_d]\omega_1^{i_1j_1}\omega_2^{i_2j_2}\cdots\omega_d^{i_dj_d} \tag{A.13}$$

其中 x 是 d 维复型数组，其元素为 $x[j_1,j_2,\ldots,j_d]$，$\omega_s=e^{2\pi\sqrt{-1}/n_s}$，$0\leqslant j_s<n_s$，$s\in\{1,2,\ldots,d\}$。$y$ 与 x 的结构一样。

先做一次向前变换，然后再做一次向后变换相当于将数组乘以 $\prod_{s=1}^{d}n_s$。

主要函数如下：

```
fftw_plan fftw_plan_dft_1d(int n,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_2d(int nx, int ny,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_3d(int nx, int ny, int nz,
                           fftw_complex *in, fftw_complex *out,
```

```
                          int sign, unsigned flags);
fftw_plan fftw_plan_dft   (int rank, const int *n,
                          fftw_complex *in, fftw_complex *out,
                          int sign, unsigned flags);
```

前三个函数分别用于一、二和三维变换，参数 nx (或 n)、ny 和 nz 分别给出 x、y 和 z 方向的变换大小。最后一个函数，fftw_plan_dft，用于任意维变换，参数 rank 给出维数，相当于式 (A.12) 和式 (A.13) 中的 d，而数组 n[] 则给出每维上的变换大小：n_i = n[i]，$i = 1, 2, \ldots, d$。

参数 in 和 out 分别给出输入和输出数组，它们相当于式 (A.12) 和式 (A.13) 中的 x 和 y。除非使用了 FFTW_ESTIMATE 标志，否则这些数组的内容在创建方案时会被破坏。

参数 sign 可以是 FFTW_FORWARD (+1) 或 FFTW_BACKWARD (-1)，用于指定向前还是向后变换。

参数 flags 包含一组按位表示的标志用于指定变换算法细节，常用的有下面一些标志，不同的标志可用 "|" 运算符组合起来使用：

(1) FFTW_ESTIMATE 不通过试算，直接选用一个相对合理的方案，变换性能不一定是最优的。使用该标志创建方案时不会破坏输入/输出数组 in/out 的内容。

(2) FFTW_MEASURE 通过进行几次试算产生一个优化的方案，它是默认的方案创建方式，创建方案的过程会花费一些时间，通常数秒钟。

(3) FFTW_PATIENT 与 FFTW_MEASURE 类似，但测试更多的可能性以求产生一个更好的方案，代价是创建方案需要的时间更长。

(4) FFTW_EXHAUSTIVE 比 FFTW_PATIENT 做更多的测试以求进一步优化变换方案，包括对许多通常并不一定会有效的算法进行测试。

(5) FFTW_DESTROY_INPUT 在执行变换时允许破坏输入数组 in 的内容 (有助于改善变换性能)。

(6) FFTW_PRESERVE_INPUT 在执行变换时不允许破坏输入数组 in 的内容。

函数 fftw_plan_dft_c2r 的默认行为是 FFTW_DESTROY_INPUT，其余函数的默认行为则是 FFTW_PRESERVE_INPUT。对于高维变换，函数 fftw_plan_dft_c2r 中不允许使用标志 FFTW_PRESERVE_INPUT，如果使用了这个标志，则方案创建将会失败 (函数返回空指针 NULL)。

A.4.2 实型变换

FFTW 的实型变换指输入数组为实型数组的情形，变换公式与复型变换一样。对于一维变换而言，假设 $x_0, \ldots, x_{n-1}$ 是一个实型数组，$y_0, \ldots, y_{n-1}$ 是它的 (正向) 离散傅里叶变换结果，则数组 y 满足条件：$y_i = \mathrm{conj}(y_{n-i})$，$0 \leqslant i < n$，这样的数组称为 Hermitian 数组。为了节省内存空间，实型变换函数中 Hermitian 数组采用长度为 $\lfloor n/2 \rfloor + 1$ 的复型数组存储。

高维 Hermitian 数组满足 $y_{i_1,\ldots,i_d} = \mathrm{conj}(y_{n_1-i_1,n_2-i_2,\ldots,n_d-i_d})$，$0 \leqslant i_k < n_k$，$k = 1, 2, \ldots, d$，这里假设数组在所有维上都是周期的，即 $y_{\ldots,n_k,\ldots} = y_{\ldots,0,\ldots}$。FFTW 中高维 Hermitian 数组存储在一个长度为 $n_1 \times \ldots \times n_{d-1} \times (\lfloor n_d/2 \rfloor + 1)$ 的复型数组中，数组最后一维的大小大约是变换大小的一半。

与复型变换不同的是，FFTW 的正向和反向实型变换创建方案时采用不同的函数。这些函数除了比相应的复型变换函数少一个参数 sign 外，其余参数形式和顺序是一样的，但是它们的输入和输出数组一个是实型数组、另一个是 Hermitian 数组。这些变换函数包括：

正向变换

```
fftw_plan fftw_plan_dft_r2c_1d(int n, double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c_2d(int nx, int ny, double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c_3d(int nx, int ny, int nz,
                               double *in, fftw_complex *out, unsigned flags);
fftw_plan fftw_plan_dft_r2c(int rank, const int *n,
                            double *in, fftw_complex *out, unsigned flags);
```

反向变换

```
fftw_plan fftw_plan_dft_c2r_1d(int n, fftw_complex *in, double *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_c2r_2d(int nx, int ny, fftw_complex *in, double *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_c2r_3d(int nx, int ny, int nz,
                               fftw_complex *in, double *out, unsigned flags);
fftw_plan fftw_plan_dft_c2r(int rank, const int *n,
                            fftw_complex *in, double *out, unsigned flags);
```

A.4.3 并行 FFTW

在共享内存型 (SMP) 并行计算机上，FFTW 支持多线程并行。用户通过下面几个函数来建立、控制多线程并行：

```
int fftw_init_threads(void);
void fftw_plan_with_nthreads(int nthreads);
void fftw_cleanup_threads(void);
```

其中函数 `fftw_init_threads` 用在所有多线程并行的变换函数之前，其作用是初始化多线程并行；函数 `fftw_plan_with_nthreads` 指定使用的线程数，调用它后，新创建的方案将使用所指定的线程数；函数 `fftw_cleanup_threads` 用在所有多线程变换完成之后，以释放 FFTW 的多线程函数占用的资源。

FFTW 2 中提供了适合于分布式存储并行系统的 MPI 版本，而

FFTW 3 中尚未实现这些功能。如果需要 MPI 并行，目前可以考虑使用 FFTW 2.1.5，也可以通过在部分空间方向上划分数据、并对数据在处理器间进行一次转置来调用 FFTW 3 的串行变换函数实现。

A.4.4 FFTW 计算实例

本节给出一个 1 维复型快速傅里叶变换的完整代码实例。

输入时间序列 $X \in C^n$，通过一维离散傅里叶变换得到输出频谱序列 $Y \in C^n$，即：

$$Y_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \omega^{-jk} X_k, \quad j = 0, 1, \ldots, n-1$$

其中 $\omega = e^{2\pi\sqrt{-1}/n}$。将上式写成矩阵形式为

$$Y = WX$$

其中 $W = (\omega^{jk})$ 为 $n \times n$ 阶循环方阵。直接采用矩阵乘向量运算，上式的浮点计算量为 $O(n^2)$。如果运用 FFT 算法计算上式，则浮点计算量降至 $O(n \log n)$。

代码 A.6: FFTW 程序实例。

文件名: `code/fftw/fftw-1d.c`

```
1  #include "fftw3.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5
6  #define N       4
7  #define REAL    0
8  #define IMAG    1
9  #define PI      3.1415926535898
10
```

```
11 int main (void)
12 {
13     fftw_complex *in, *out;
14     fftw_plan p;
15     double constants[N] = {10, 2.1, 4.7, 1.3};
16     double f;
17     int i, j;
18
19     /* Allocate memory for the arrays */
20     in = fftw_malloc(sizeof(fftw_complex) * N);
21     out = fftw_malloc(sizeof(fftw_complex) * N);
22
23     if ((in == NULL) || (out == NULL)) {
24         printf ("Error: insufficient available memory\n");
25     }
26     else {
27         /* Create the FFTW execution plan */
28         p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
29
30         /* Initialize the input data */
31         for (i = 0; i < N; i++) { /* All sampling points */
32             in[i][REAL] = constants[0];
33             in[i][IMAG] = 0;
34             for (j = 1; j < N; j++) { /* All frequencies */
35                 in[i][REAL] += constants[j] * cos(j * i * 2 * PI / (double)N);
36                 in[i][IMAG] += constants[j] * sin(j * i * 2 * PI / (double)N);
37             }
38         }
39
40         /* Execute plan */
41         fftw_execute(p);
42
43         /* Destroy plan */
44         fftw_destroy_plan(p);
45
46         /* Display results */
```

```
47          printf ("Constants[] = {");
48          for (i = 0; i < N; i++)
49              printf("%lf%s", constants[i], (i == N-1) ? "}\n" : ", ");
50
51          printf ("Input[][REAL] = {");
52          for (i = 0; i < N; i++)
53              printf("%lf%s", in[i][REAL], (i == N-1) ? "}\n" : ", ");
54
55          printf ("Output[][REAL] = {");
56          for (i = 0; i < N; i++)
57              printf("%lf%s", out[i][REAL], (i == N-1) ? "}\n" : ", ");
58
59          /* Scale output */
60          f = 1.0/sqrt((double)N);
61          for (i = 0; i < N; i++)
62              out[i][REAL] *= f;
63
64          /* Display final results */
65          printf ("Scaled[][REAL] = {");
66          for (i = 0; i < N; i++)
67              printf("%lf%s", out[i][REAL], (i == N-1) ? "}\n" : ", ");
68      }
69
70      /* Free allocated memory */
71      if (in != NULL) fftw_free(in);
72      if (out != NULL) fftw_free(out);
73
74      return 0;
75  }
```

A.5 PETSc

PETSc (Portable，Extensible Toolkit for Scientific Computation) 是美国能源部 DOE2000 支持开发的 20 多个 ACTS 工具箱之一，由

Argonne 国家实验室开发的可移植可扩展科学计算工具箱，主要用于在分布式存储环境高效求解偏微分方程组及相关问题。PETSc 所有消息传递通信均采用 MPI 标准实现。

PETSc 用 C 语言开发，遵循面向对象设计的基本特征，用户基于 PETSc 对象可以灵活开发应用程序。目前，PETSc 支持 Fortran 77/90、C 和 C++ 编写的串行和并行代码。

PETSc 是一系列软件和库的集合，三个基本组件 SLES、SNES 和 TS 本身基于 BLAS、LAPACK 和 MPI 等库实现，同时为 TAO、ADIC/ADIFOR、MATLAB、ESI 等工具提供数据接口或互操作功能，并具有极好的可扩展性能。PETSc 为用户提供了丰富的 Krylov 子空间迭代方法和预条件子，并提供错误检测、性能统计和图形打印等功能。

如今，越来越多的应用程序在 PETSc 环境上开发，并逐渐显示出 PETSc 在高效求解大规模数值模拟问题方面的优势和威力。

PETSc 的网站是：`http://www.mcs.anl.gov/petsc`。

A.5.1 PETSc 的系统结构

不同于其他微分/代数方程求解器，PETSc 为用户提供了一个通用的高层应用程序开发平台。基于 PETSc 提供的大量对象和解法库，用户可以灵活地开发自己的应用程序，还可随意添加和完善某些功能，如为线性方程求解提供预条件子、为非线性问题的牛顿迭代求解提供雅可比矩阵、为许多数值应用软件和数学库提供接口等。图 A.4 描述了 PETSc 在实现层次上的抽象。这里做简要说明。

应用程序

用户在 PETSc 环境下基于 PETSc 对象和算法库编写的串行或并行应用程序。尽管 PETSc 完全在 MPI 上实现，但 PETSc 程序具有固定的框架结构，包括初始化、空间释放和运行结束等环境语句。

PDE 求解器

用户基于 PETSc 的三个基本算法库 (TS、SNES 和 SLES) 构建的偏微方程求解器。但它却不是 PETSc 的基本组件。

TS (Time Stepping)

时间步进积分器，用于求解常微分方程 (ODE)，或依赖时间的空间离散化后的偏微分方程 (PDE)。对于非时间演化或稳态方程，PETSc 提供了伪时间步进积分器。TS 积分器最终依赖线性求解器 SLES 和非线性求解器 SNES 来实现。PETSc 为 PVODE 库提供了接口。另外，TS 的用法非常简单方便。

SNES (Nonlinear Solver)

非线性求解器，为大规模非线性问题提供高效的非精确或拟牛顿迭代解法。SNES 调用线性求解器 SLES，并采用线搜索和信赖域方法实现。SNES 依赖于雅可比矩阵求解，PETSc 既支持用户提供的有限差分程序，同时又为用户提供了 ADIC 等自动微分软件生成的微分程序接口。

SLES (Linear Solver)

线性求解器，求解大规模线性方程组，它是 PETSc 的核心部分。PETSc 几乎提供了所有求解线性方程组的高效求解器，既有串行求解也有并行求解，既有直接法求解也有迭代法求解。对于大规模线性方程组，PETSc 提供了大量基于 Krylov 子空间方法和各种预条件子的成熟而有效的迭代方法，以及其它通用程序和用户程序的接口。

KSP (Krylov Subspace)

Krylov 子空间方法，包括 Richardson 方法、共轭梯度法 (CG 和 BiCG)、广义最小残差法 (GMRES)、最小二乘 QR 分解 (LSQR) 等。

PC (Preconditioner)

预条件子，包括雅可比、分块雅可比、SOR / SSOR、不完全 Cholesky 分解、不完全 LU 分解、加性 Schwartz，多重网格等方法。

DRAW

应用程序的性能分析和结果显示。

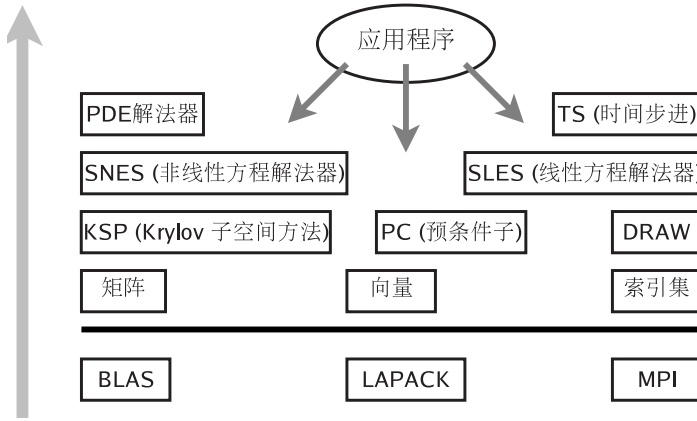


图 A.4 PETSc 实现的层次结构

A.5.2 PETSc 的基本特色

通过 PETSc 开发应用程序往往具有相当的难度。一方面，它需要用户本身具有较高的数值计算方法方面的专业知识和并行计算方法方面的编程技巧。另一方面，总的来说 PETSc 只是一个高级的应用程序开发环境，它为许多软件 (库) 和用户程序提供接口，用户只有充分熟悉和利用现有的软件资源和数学库的基础上才有可能开发出高效的应用程序。尽管如此，PETSc 仍然因为其具有其他软件不

可比拟的优点吸引着越来越多的用户用它来开发应用程序。下面一一介绍和分析 PETSc 的这些优点或特色。

计算能力 PETSc 为用户提供了丰富的算法和库资源。三个求解器（SLES、SNES 和 TS）构成了 PETSc 的核心组件。PETSc 不仅为中小规模线性方程组的求解提供了高效的直接方法，还为大规模 (稀疏) 线性方程组的迭代求解提供了多种 Krylov 子空间方法和多种预条件子。

兼容性 一方面，PETSc 具有很强的兼容能力，可在不同体系结构和不同操作系统环境高效运行。另一方面，PETSc 本身基于高性能的线性代数库 (BLAS 和 LAPACK) 和 MPI 消息传递环境实现，同时又充分吸收和融入了其他优秀软件的优点，如无结构网格区域分解和雅可比矩阵求解等方面的功能。

可扩展性 PETSc 的可扩展性主要体现在三个方面：计算性能的并行可扩展性、功能的可扩展性和计算能力的可扩展性。无论是在计算时间还是在浮点性能方面，PETSc 提供的范例程序都有良好的线性加速性能。面向对象的良好设计风格使得 PETSc 具有良好的功能扩展能力。作为一个高级应用程序开发平台，PETSc 特别适合于用来开发大型应用程序。

抽象数据类型 PETSc 基于面向对象技术实现，具有所有面向对象软件的可移植性、可继承性和可扩展性等基本程序特征。PETSc 的向量、矩阵等基本数据对象完全采用抽象数据类型实现，尽量对用户屏蔽数据对象的区域分解和存储等细节。所有 PETSc 格点数据对象的划分、初始化和存取等基本操作都由 DA 对象 (Distributed Array，即分布式数组) 来管理和相应 PETSc 库函数来实现。用户基于 PETSc 对象可以灵活开发其应用程

序。PETSc 对象和组件为构造大规模应用程序奠定了一个良好的基础。

输出能力 PETSc 具有良好的性能分析和图形输出功能，同时还具有高可用性，并具有很强的错误诊断能力。

总之，PETSc 在无论是在计算能力、设计风格还是在可兼容性和可扩展性等方面都显示出极大的优越性。PETSc 不但为科学与工程计算领域的科学家和工程师提供了强大的 (大规模) 偏微方程求解工具，而且也为高效算法设计提供了一个丰富的试验平台和计算环境。它使得算法的扩展和应用程序的个性化实现都更为容易。另外，PETSc 的这种设计风格增强了代码的重用性和编程的灵活性。

A.5.3 PETSc 的基本功能

这里主要介绍 PETSc 的三个核心组件及性能分析和接口功能。三个核心组件包括线性方程组求解器 (SLES)、非线性方程组求解器 (SNES) 和时间步进积分器 (TS)。

1. 线性方程组求解器

线性方程组求解器 (SLES) 构成了 PETSc 最核心的部分。它不仅是 PDE 方程求解器的基本内核，而且也是实现 PETSc 的其他两个核心组件 SNES 和 TS 的必不可少的部分。SLES 求解线性方程组

$$Ax = b \tag{A.14}$$

其中解算子 A 是 $n \times n$ 维非奇异矩阵，b 是 n 维右端向量，x 为 n 维解向量。SLES 在线性方程组求解环境的创建、Krylov 子空间方法和预条件子 (PC) 的选择、收敛性判据、LU 直接求解等方面为用户提供了强大的功能和灵活的操作方式。

2. 非线性方程组求解器

非线性方程组求解器 (SNES) 基于牛顿迭代法 (线搜索和信赖域方法)，在线性方程组求解器 (SLES) 的基础上实现。雅可比矩阵的求解是 SNES 求解器的重要组成部分。SNES 求解以下形式的非线性方程组

$$F(x) = 0 \tag{A.15}$$

其中算子 F 是 $\mathbb{R}^n \to \mathbb{R}^n$ 的函数。

3. 时间步进积分器

时间步进积分器 (TS) 用于求解 ODE 方程，或依赖时间的空间离散化后的 PDE 方程。TS 主要求解如下时间依赖问题

$$u_t = F(u, t) \tag{A.16}$$

其中 u 为有限维解向量，上式通常为运用有限差分或有限元方法对 PDE 进行离散后得到的常微分方程组。对于非时间演化或稳态方程，PETSc 提供了伪时间步进积分器。TS 积分器最终依赖线性方程组求解器 (SLES) 和非线性方程组求解器 (SNES) 来实现。PETSc 还提供了与 PVODE 求解器的接口。

4. PETSc 的其他功能

PETSc 不仅能为应用程序提供完整的计算时间、存储代价和浮点性能方面的信息，还能为计算的每个阶段和每个程序对象提供详细的信息。这些信息对于程序调试与优化、算法分析与比较等都有非常大的帮助。

5. PETSc 与其他软件

PETSc 可扩展性的另一个方面还表现在它为非常广泛的一类数值软件和数学库提供了非常方便的程序接口，主要包括以下几种类型：

(1) 线性代数求解器，如 AMG，BlockSolve95，DSCPACK，hypre，ILUTP，LUSOL，SPAI，SPOOLES，SuperLU，SuperLU_Dist；

(2) 最优化软件，如 TAO，Veltisto；

(3) 离散化和网格生成和优化工具包，如 Overture，SAMRAI，SUMAA3d；

(4) 常微分方程求解器，如 PVODE；

(5) 其他，如 MATLAB，ParMETIS。

感兴趣的读者可自行参阅相关软件的手册。

A.5.4 PETSc 计算实例

本节讨论基于 PETSc 的三个核心求解器 SLES、SNES 和 TS 如何开发不同的应用程序，它们分别与一个或多个不同的 PETSc 范例程序相对应。PETSc 范例程序不仅给用户提供了大量的编程模版，同时也给用户选择合适的数值方法提供了很好的参考。这些典型的范例程序包括二维 Poisson 方程、二维 Bratu 方程和一维热传导方程的数值求解。

1. 二维泊松方程的求解

(1) 问题描述及其离散化

二维 Poisson 方程是典型的椭圆型方程，其初边值问题在物理与工程领域具有重要的应用。考虑长方形区域 $\Omega = (0,a) \times (0,b)$ 上的二维 Poisson 方程

$$\begin{cases} -\Delta u = f, & (x,y) \in \Omega \\ u = g, & (x,y) \in \partial\Omega \end{cases} \tag{A.17}$$

如果 $f=0$，方程 (A.17) 就退化为二维 Laplace 方程。采用均匀网格及五点有限差分格式将方程 (A.17) 离散化，并取 $h_x=a/n$，$h_y=b/m$ 为网格步长，$x_i=ih_x$，$y_j=jh_y$，$i=0,1,\ldots,n$，$j=0,1,\ldots,m$。则差分方程为

$$\begin{cases} \dfrac{2u_{i,j}-u_{i+1,j}-u_{i-1,j}}{h_x^2}+\dfrac{2u_{i,j}-u_{i,j+1}-u_{i,j-1}}{h_y^2}=f_{i,j} \\ u_{i,0}=g_{i,0},\quad u_{i,m}=g_{i,m},\quad u_{0,j}=g_{0,j},\quad u_{n,j}=g_{n,j} \\ \qquad i=1,2,....,n-1,\quad j=1,2,....,m-1 \end{cases} \tag{A.18}$$

令 $d=1/(2/h_x^2+2/h_y^2)$，$d_x=d/h_x^2$，$d_y=d/h_y^2$，则上式可改写为

$$\begin{cases} u_{i,j}-d_x(u_{i+1,j}+u_{i-1,j})-d_y(u_{i,j+1}+u_{i,j-1})=df_{i,j} \\ u_{i,0}=g_{i,0},\quad u_{i,m}=g_{i,m},\quad u_{0,j}=g_{0,j},\quad u_{n,j}=g_{n,j} \\ \qquad i=1,2,....,n-1,\quad j=1,2,....,m-1 \end{cases} \tag{A.19}$$

这是一个 $(n-1)\times(m-1)$ 维的大型稀疏线性方程组。为简单起见，这里取网格步长 $h_x=h_y$，并适当排列、整理以上方程组，则式 (A.19) 可写成如下分块形式：

$$\begin{bmatrix} A & I & & & \\ I & A & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & A & I \\ & & & I & A \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_{m-2} \\ \mathbf{u}_{m-1} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_{m-2} \\ \mathbf{b}_{m-1} \end{bmatrix} \tag{A.20}$$

其中 I 为 $n-1$ 阶单位矩阵，A 为 $n-1$ 阶三对角对称矩阵，

$$A=\begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ & & & 1 & -4 \end{bmatrix},$$

$$\mathbf{u}_j=\begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \vdots \\ u_{n-2,j} \\ u_{n-1,j} \end{bmatrix}, \quad \mathbf{b}_j=-4d\begin{bmatrix} f_{1,j} \\ f_{2,j} \\ \vdots \\ f_{n-2,j} \\ f_{n-1,j} \end{bmatrix}$$

图 A.5 显示了使用 PETSc 运行选项 `-mat_view_draw` 后打印的网格规模为 36×36 的稀疏矩阵结构，它与方程 (A.20) 中的系数矩阵相对应。

(2) SLES 求解器中的主要参数设置

PETSc 在运行参数列表中提供了许多可选功能来完成 SLES 求解器的使用，其中与求解本例有关的选项有 (参考 SLES 范例 `ex2`)：

`-m/-n`: x 和 y 方向网格点数目

`-ksp_type`: 选择 Krylov 子空间迭代方法的类型，包括 Richardson 方法、切比雪夫方法、共轭梯度方法 (CG)、广义最小残量法 (GMRES)、双共轭梯度方法 (BiCG)、双共轭梯度平方法 (BCGS) 等。

`-ksp_rtol/-ksp_atol`: 设置解向量的相对误差和绝对误差。

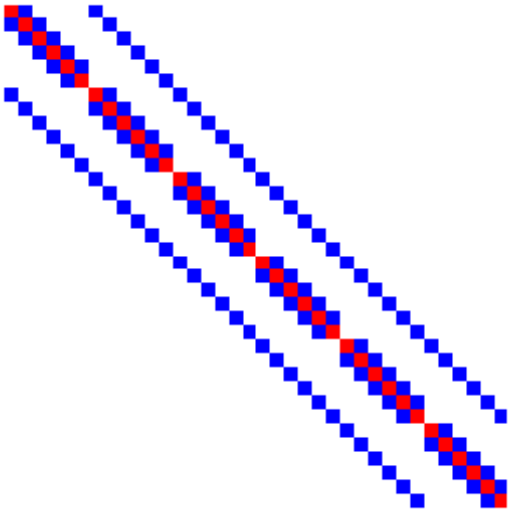


图 A.5 网格规模为 36×36 的稀疏矩阵结构：二维 Poisson 方程

`-pc_type`：选择预条件子的类型，包括雅可比矩阵、分块雅可比矩阵、加性 Schwartz 方法 (ASM)、多重网格方法等。

(3) SLES 求解器中的主要计算流程

- 运行参数的设置：包括网格点数目、解向量的相对误差或绝对误差、范数类型、迭代方法和预条件子等。
- 算子矩阵的填充：对于本例中求解的线性问题，需要填充的矩阵就是方程 (A.20) 中的系数矩阵 A。它是一个五对角的稀疏矩阵，通常可采用 PETSc 提供的稀疏行压缩格式 (AIJ)。
- 设置右端向量和解向量的初值。

- 启动 SLES 求解器。
- 结束 SLES 求解器并释放所有存储空间。

(4) 稀疏矩阵的赋值

稀疏矩阵的赋值关键是为函数 `MatCreatMPIAIJ` (并行稀疏行压缩格式) 或 `MatCreatMPIBAIJ` (并行稀疏矩阵的分块压缩格式) 提供四个参数：主对角分块每行非零元素的最大数目和每行非零元素的数目列表，非主对角分块每行非零元素的最大数目和每行非零元素的数目列表。根据方程 (A.20) 中表示的系数矩阵，每个进程划分中的主对角分块和非主对角分块每行非零元素的最大数目分别为 5 和 1。对于第一个进程划分，每行主对角非零元素的数目列表为：

$$
\begin{gathered}
3,4,\ldots,4 \\
5,5,\ldots,5 \\
\cdots \\
4,4,\ldots,3
\end{gathered}
$$

而每行非主对角非零元素的数目列表为

$$
\begin{gathered}
0,0,\ldots,0 \\
0,0,\ldots,0 \\
\cdots \\
1,1,\ldots,1
\end{gathered}
$$

其余进程划分可类似分析。当引用函数 `MatCreatMPIAIJ` 创建了一个稀疏矩阵结构后，用户就可以按通常的方式逐行对矩阵进行填充了。

(5) Krylov 子空间迭代方法和预条件子的选择

Krylov 子空间迭代方法是求解大型稀疏线性方程组的一类有效算法，其收敛速度依赖于矩阵特征值或奇异值的分布。如果选取一

个好的预条件子，各种 Krylov 子空间迭代方法在计算效率上通常没有多大差别，但实践中如何识别并构造一个合适的预条件子却甚为困难。

对本例 (即 SLES 范例 `ex2`) 的大量测试结果表明，在 PETSc 提供的所有 Krylov 子空间迭代方法中，共轭梯度方法 (CG) 是求解该问题最有效的迭代方法之一，它通常具有最小的计算时间和存储需求，而使用块雅可比矩阵预条件子则能成倍地加速其计算收敛性能。

2. 二维 Bratu 问题的求解

(1) 问题描述及其离散化

考虑区域 $\Omega = (0,1) \times (0,1)$ 上的二维 Bratu 方程

$$\begin{cases} -\nabla^2 u - \lambda e^u = 0, & (x,y) \in \Omega \\ u = 0, & (x,y) \in \partial\Omega \end{cases} \tag{A.21}$$

其离散化过程与 Poisson 方程的离散化完全类似，采用均匀网格及五点有限差分格式将方程 (A.21) 离散，并取 $h_x = 1/n$，$h_y = 1/m$ 为网格步长，$x_i = ih_x$，$y_j = jh_y$，$i = 0,1,\ldots,n$，$j = 0,1,\ldots,m$。则差分方程为

$$\begin{cases} \dfrac{2u_{i,j} - u_{i+1,j} - u_{i-1,j}}{h_x^2} + \dfrac{2u_{i,j} - u_{i,j+1} - u_{i,j-1}}{h_y^2} - \\ \qquad \lambda e^{u_{i,j}} = 0 \\ u_{i,0} = 0, \quad u_{i,m} = 0, \quad u_{0,j} = 0, \quad u_{n,j} = 0 \\ \qquad i = 1,2,....,n-1, \quad j = 1,2,....,m-1 \end{cases} \tag{A.22}$$

令 $\alpha = h_x/h_y$，$\beta = -\lambda h_x h_y$，

$$f_{i,j} = \left[(\alpha + \frac{1}{\alpha})u_{i,j} + \beta e^{u_{i,j}}\right] - \frac{1}{\alpha}(u_{i+1,j} + u_{i-1,j}) - \alpha(u_{i,j+1} + u_{i,j-1}) \tag{A.23}$$

将上式改写成向量形式

$$F(U) = 0 \tag{A.24}$$

其中 $F = [F_1, F_2, \ldots, F_{n-1}]$，$U = [u_{1,1}, u_{2,1}, \ldots, u_{n-1,m-1}]$，$F_j = [f_{1,j}, f_{2,j}, \ldots, f_{n-1,j}]$，$1 \leqslant j \leqslant m-1$。显然这里 F 为非线性函数，在应用牛顿迭代法来求解方程 (A.24) 的过程中需要求解其雅可比矩阵，用来形成每步迭代的线性方程组的解算子。根据式 (A.23)，雅可

比矩阵可简单表示为

$$\frac{\partial^{\mathrm{T}} F_j}{\partial U} = \begin{bmatrix} 1 & & & & & & & & & & & & \\ & -\frac{1}{\alpha} & & & & & & & & & & & \\ & \vdots & & -\frac{1}{\alpha} & & & & & & & & & \\ & \vdots & & \vdots & & \ddots & & & & & & & \\ & -\alpha & & \vdots & & & & -\frac{1}{\alpha} & & & & & \\ \alpha+\frac{1}{\alpha}+\beta e^{u_{2,j}} & & -\alpha & & & & & \vdots & & & & & \\ & -\alpha & & \alpha+\frac{1}{\alpha}+\beta e^{u_{3,j}} & & \ddots & & \vdots & & & & & \\ & \vdots & & -\alpha & & \ddots & & -\alpha & & & & & \\ & \vdots & & \vdots & & \ddots & \alpha+\frac{1}{\alpha}+\beta e^{u_{n-1,j}} & & & & & & \\ & \vdots & & \vdots & & & & -\alpha & & & & & \\ & -\frac{1}{\alpha} & & \vdots & & & & \vdots & & & & & \\ & & & -\frac{1}{\alpha} & & & & \vdots & & & & & \\ & & & & & \ddots & & \vdots & & & & & \\ & & & & & & & -\frac{1}{\alpha} & & & & & \\ & & & & & & & & & & & & 1 \end{bmatrix} \tag{A.25}$$

其中 $j = 1, \ldots, m-1$，分块雅可比矩阵维数为 $(m-1)\times(n-1)\times(n-1)$。图 A.6 显示了网格规模为 36×36 的雅可比稀疏矩阵结构。

(2) SNES 求解器中的主要参数设置

PETSc 在运行参数列表中提供了丰富的可选功能来完成 SNES 求解器和 DA 对象的使用，其中与求解本例有关的选项有 (参考

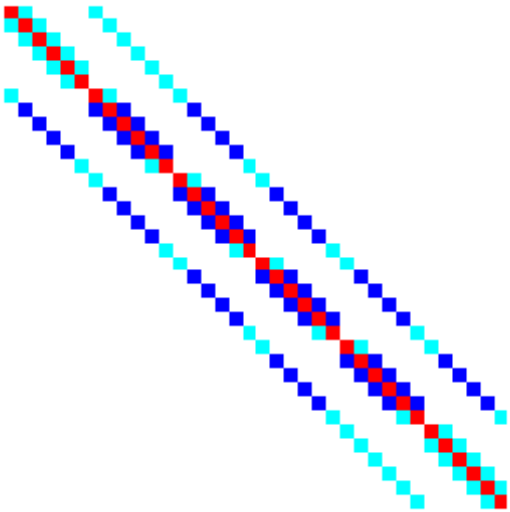


图 A.6 网格规模为 36×36 的稀疏雅可比矩阵结构：二维 Bratu 方程

SNES 范例 ex5)：

-da_grid_x/-da_grid_y：设置 x 和 y 方向网格点数目

-fd_jacobian：选择有限差分方法求解雅可比矩阵

-adic_jacobian：选择自动微分方法求解雅可比矩阵

-adicmf_jacobian：选择“无矩阵”格式和自动微分方法求解雅可比矩阵

-snes_type：选择牛顿迭代类型，包括一维线搜索和信赖域方法

-snes_rtol/-snes_atol：设置解向量的相对误差和绝对误差

`-snes_max_it`：设置最大牛顿迭代步数

`-ksp_type`：选择 Krylov 子空间迭代方法的类型

`-pc_type`：选择预条件子的类型

`-ksp_rtol/-ksp_atol`：设置每步求解线性方程组解向量的相对误差和绝对误差

`-ksp_max_it`：设置每步求解线性方程组的最大迭代步数

(3) SNES 求解器中的主要计算流程

- 运行参数的设置：包括网格点数目、求解雅可比矩阵的方法、牛顿迭代类型、牛顿迭代的精度和最大迭代步数、Krylov 子空间迭代方法和预条件子、线性方程组求解的精度和最大迭代步数等。
- 创建 DA 向量对象：包括创建函数对象、解向量和雅可比矩阵对象。
- 非线性函数和雅可比矩阵: 见式 (A.24) 和式 (A.25)。
- 设置向量的边值：见式 (A.22)，均置零。
- 启动 SNES 求解器：主要计算流程见图 A.7。
- 打印输出结果和性能统计。
- 结束 SNES 求解器并释放所有存储空间。

(4) 雅可比矩阵的着色和求解

PETSc 提供了三种有效的方法来求解雅可比矩阵，包括有限差分方法、自动微分方法和“无矩阵”方法。这里仅做简单介绍。

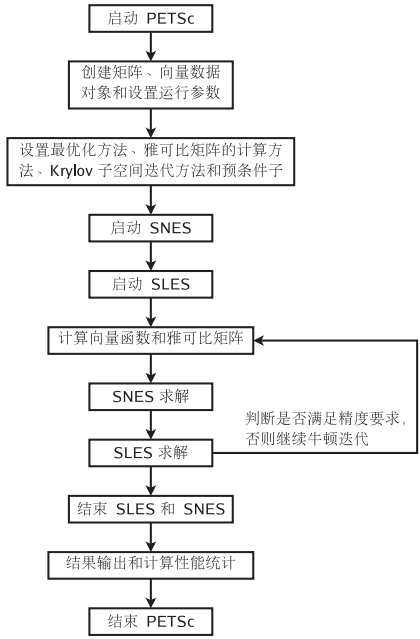


图 A.7 非线性求解器 (SNES) 的主要计算流程

有限差分方法 传统有限差分方法求解函数导数由于因数值方法带来的截断误差和因机器有效精度的影响，具有计算代价高和精度低的缺点。事实上，选择一个合适的变量增量值本身就是一个非常棘手的问题。

自动微分方法 自动微分方法在无截断误差意义上分析求解函数的导数，具有计算时间代价小、精度高和可靠性好等优点。切线性模式和伴随模式分别为与之相对应的最基本的代码实现形式，它们在精确求解函数梯度和雅可比矩阵方面分别具有不同的计算时间和空间存储代价。伴随模式在求解函数梯度方面具有理

想的计算时间代价，但其存储代价往往随问题的计算规模线性增长。

“无矩阵”方法 事实上，在使用牛顿迭代方法求解非线性方程组的过程中，只需求解雅可比矩阵与向量的乘积，因此许多矩阵运算本身可以借助向量操作来实现。然而，PETSc 目前尚未为“无矩阵”方法提供任何预条件子。

无论采用哪种方法来求解雅可比矩阵，都须首先求解其稀疏结构。当雅可比矩阵各元素取绝对值后的任意两行或两列相互“正交”时，通过适当设置初始输入向量后，雅可比矩阵的这两行或两列的所有元素都可以在一次微分函数或差分近似中同时求出。所有这样的初始输入向量经顺序排列后就形成初始输入矩阵。而如何根据雅可比矩阵的稀疏结构来求解初始输入矩阵，就是通常意义上的雅可比矩阵着色问题 (参考文献 [43, 44])。

(5) 迭代方法和预条件子的选择

此外，由于牛顿迭代的每一步都要求解一个线性方程组，用户还须为 SNES 中的迭代求解器 (KSP) 提供合适的迭代方法和预条件子。其中 PETSc 默认取广义最小残量法 (GMRES) 和零级优化的不完全 LU 分解预条件子。关于不同迭代方法和预条件子的选取，用户可以参考 429 页“二维泊松方程的求解”，这里不再赘述。

3. 一维热传导方程的求解

(1) 问题描述及其离散化

热传导方程是最简单的抛物型方程，其初边值问题在物理与工程领域具有广泛的应用。考虑区域 $\Omega=(0,1)$ 上的一维热传导方程

$$\begin{cases} u_t = u_{xx}, & x\in\Omega, 0<t\leqslant T \\ u(0,x)=\sin 6\pi x+3\sin 2\pi x, & x\in\Omega \\ u(t,0)=u(t,1)=0, & 0\leqslant t\leqslant T \end{cases} \tag{A.26}$$

这是一个二阶线性方程，解析解取为 $u(t,x)=e^{-36\pi^2 t}\sin 6\pi x+3e^{-4\pi^2 t}\sin 2\pi x$。将方程 (A.26) 等距离散化，取网格步长 $h=1/m$，$x_i=ih$，$i=0,1,\ldots,m$，就得到如下形式的时间依赖问题

$$\begin{cases} (u_j)_t=\dfrac{u_{j+1}-2u_j+u_{j-1}}{h^2} \\ u_j(0)=\sin 6\pi jh+3\sin 2\pi jh \\ u_0(t)=u_m(t)=0 \end{cases} \tag{A.27}$$

将上式改写成向量形式

$$U_t=AU \tag{A.28}$$

其中 $U=[u_1(t),u_2(t),\ldots,u_{m-1}(t)]$，系数矩阵

$$A=\frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}$$

显然系数矩阵 A 不依赖于时间，时间积分的每一步中只须重新设置右端向量。对于时间参数的离散化，TS 求解器在实现过程中可以分别采用显式向前 Euler 方法和隐式向后 Euler 方法。如果采用隐式向后 Euler 方法，离散化后可得

$$\begin{cases} \dfrac{u_j^{i+1}-u_j^i}{\Delta t^i} = \dfrac{u_{j+1}^{i+1}-2u_j^{i+1}+u_{j-1}^{i+1}}{h^2} & \\ u_j^0 = \sin(6\pi jh)+3\sin(2\pi jh), & j=0,1,....,m \\ u_0^i = u_m^i = 0, & i=0,1,....,n \end{cases} \tag{A.29}$$

取固定时间步长 $\tau = T/n$，并写成向量形式

$$[(1+2r)I - rC]\,U_h^{k+1} = U_h^k \tag{A.30}$$

其中 $r=\tau/h^2$，$U_h^k=[u_1^k,u_2^k,\ldots,u_{m-1}^k]$，而矩阵

$$C = \frac{1}{h^2}\begin{bmatrix} 0 & 1 & & & \\ 1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 0 & 1 \\ & & & 1 & 0 \end{bmatrix}$$

这是一个三对角线性方程组，可以利用 SLES 求解器的迭代方法求解。在选择时间步长和空间步长时，一定要注意为了保持计算过程的稳定性，不同离散格式对它们的限制是不同的。

(2) TS 求解器中的主要参数设置

PETSc 在运行参数列表中提供了许多可选功能来完成 TS 求解器的使用，其中与用向后 Euler 方法求解热传导方程有关的有：

`-m`: 网格点数目

`-ts_type`: 选择积分类型，有向前 Euler (`euler`)、向后 Euler (`beuler`) 和拟时间步进积分法 (`pseudo`)

`-ts_max_time`: 最终时间

`-ts_max_steps`: 最大时间积分步数

`-time_dependent_rhs`: 选择右端项为时间依赖项

`-TSSetInitialTimeStep`: 设置初始时间和时间步长

`-ksp_type`: 选择 Krylov 子空间迭代方法的类型

`-pc_type`: 选择预条件子的类型

(3) TS 求解器中的主要计算流程

- 运行参数的设置：包括积分类型、网格点数目、初始和最终时间、时间步长、最大时间积分步数、迭代方法和预条件子等。
- 设置求解矩阵：本例中求解线性问题，需要填充的线性算子就是式 (A.28) 中的系数矩阵 A。它是一个三对角的稀疏矩阵，用户可采用 PETSc 提供的稀疏行矩阵填充结构 (AIJ)。
- 向量初值的设置：设置求解向量初始时刻的值。
- 启动 TS 求解器：主要计算流程见图 A.8。
- 打印输出结果和性能统计。
- 结束 TS 求解器并释放所有存储空间。

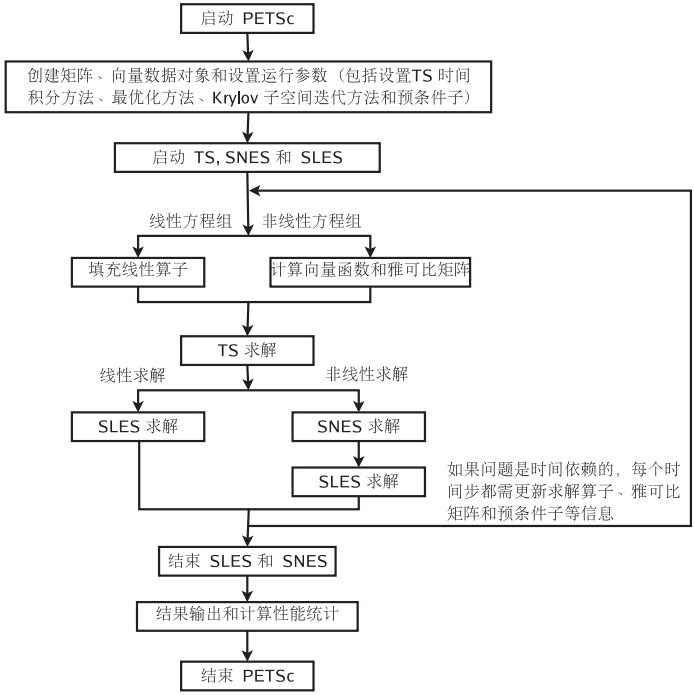


图 A.8 时间步进积分器 TS 的主要求解流程：隐式 Euler 方法

(4) 迭代方法和预条件子的选择

如果采用向后 Euler 方法，用户还须为 SLES 中的迭代求解器 (KSP) 提供合适的迭代方法和预条件子，其中 PETSc 默认选取广义最小残量法 (GMRES) 和零级优化的不完全 LU 分解预条件子。关于不同迭代方法和预条件子的选取，用户可以参考 429 页“二维泊松方程的求解”，这里不再赘述。

A.5.5 PETSc 小结

PETSc 为用户提供了一个高层的 PDE 应用程序开发平台，它由三个核心求解器，即线性求解器 SLES、非线性求解器 SNES 和时间步进积分器 TS 组成。PETSc 不仅提供丰富的 Krylov 子空间迭代方法和预条件子，还具有完善的性能统计、对象性能分析和图形可视化能力。用户基于 PETSc 的基本对象和库函数可以灵活地开发应用程序，同时也可以向 PETSc 添加新算法和预条件子等功能。此外，PETSc 还为许多软件和库提供了方便的接口。

PETSc 提供的分布式存储对象 (DA) 向用户屏蔽了物理存储的具体实现方式和细节，这种被尽可能追求的代数抽象让用户设计自己的数据结构和应用程序更为方便，客观上也降低了并行程序开发的难度。这种抽象的一个负面因素就是：远离具体存储实现的用户从理论上难以通过组织应用程序来获得程序的最优计算性能。另外，庞大的 PETSc 对象一方面占用系统的资源过大而影响了程序性能的提高，另一方面也增加了用户学习和应用其编程的难度。

PETSc 采用面向对象的程序技术开发，这使之具有现代软件可扩展性和可移植性好的程序风格，但同时也增加了那些仅具有 C 或 Fortran 程序开发经验的科技人员学习和应用的难度。不过，PETSc 提供的许多标准范例极大地方便了应用程序的开发。PETSc 还为许多专业软件提供了方便的接口，它们扩展了 PETSc 的应用范围和功能。

PETSc 除了学习难度较大之外，还要求几乎所有 PETSc 应用程序都必须在其提供的程序框架 (数据结构) 上开发，这有悖于人们习惯在用户程序框架下引用其他软件库和函数的传统思路。

大量测试结果表明：PETSc 应用程序具有良好的并行可扩展性能，但实际达到的浮点计算性能仍然比较低。例如，在深腾 6800 系统上，求解大规模稀疏线性方程组的 SLES 程序其单机浮点性能仅

为 80–120Mflops，大致为单机系统峰值性能的 4% ~ 8% 左右；使用一维线搜索和信赖域方法求解大规模非线性方程组的 SNES 程序其单机浮点性能分别为 30–70Mflops 和 25–40Mflops，占系统峰值性能的百分比更低；而求解大规模时间积分方程的 TS 程序其单机浮点性能仅达到 20–70Mflops。相对于 SLES 求解器而言，SNES 与 TS 求解器的单机浮点性能要更低一些。

不可否认，在高端应用程序开发平台软件的开发方面，PETSc 为新一代数值软件工具树立了一个优秀典范。面向对象的程序设计技术使得 PETSc 的所有对象和库，都具有标准化的程序接口和高度统一的程序设计风格和功能方面的巨大可扩展能力。

附录 B　MPI 参考手册

本附录给出主要 MPI 函数的参考手册。为了节省篇幅，这里仅列出 MPI 函数的 C 接口，Fortran 接口子程序的参数与 C 接口函数的参数完全类似。除 MPI_Wtime 和 MPI_Wtick 外，Fortran 接口子程序比 C 接口函数在最后多出一个整型参数，用于返回错误码。

所列出的 MPI 函数和变量是按照它们的类别组织的。为方便查找特定的函数，B.1 中给出了一个 MPI 的函数、变量名称按字母顺序排列的索引。

本附录的内容主要参考文献 [18, 19] 以及 MPICH 的部分在线手册编写而成。

B.1　MPI 函数、变量速查表

本节中出现在变量或函数名右上角的数字和后边的页码分别代表它们在参考手册中的序号及所在页号，主要为了方便它们的查找，没有其他含义。

B.2 MPI 预定义的变量及类型

B.2.1 C 语言 MPI 原始数据类型

C 语言中表示 MPI 数据类型的变量类型是 MPI_Datatype。

1. 基本数据类型

1	MPI_CHAR	对应于 char。
2	MPI_BYTE	对应于 unsigned char。
3	MPI_SHORT	对应于 short。
4	MPI_INT	对应于 int。
5	MPI_LONG	对应于 long。
6	MPI_FLOAT	对应于 float。
7	MPI_DOUBLE	对应于 double。
8	MPI_UNSIGNED_CHAR	对应于 unsigned char。
9	MPI_UNSIGNED_SHORT	对应于 unsigned short。
10	MPI_UNSIGNED	对应于 unsigned int。
11	MPI_UNSIGNED_LONG	对应于 unsigned long。
12	MPI_LONG_DOUBLE	对应于 long double (有的系统不支持)。
13	MPI_LONG_LONG_INT	对应于 long long (有的系统不支持)。

2. 特殊数据类型

14	MPI_PACKED	MPI_Pack 和 MPI_Unpack 函数用的打包类型。
15	MPI_UB	用于在 MPI_Type_struct 函数中设定数据类型的上界。
16	MPI_LB	用于在 MPI_Type_struct 函数中设定数据类型的下界。

3. MPI_MAXLOC 和 MPI_MINLOC 中使用的数据类型

17	MPI_FLOAT_INT	对应于 struct {float,int}。
18	MPI_LONG_INT	对应于 struct {long,int}。
19	MPI_DOUBLE_INT	对应于 struct {double,int}。
20	MPI_SHORT_INT	对应于 struct {short,int}。
21	MPI_2INT	对应于 struct {int,int}。
22	MPI_LONG_DOUBLE_INT	对应于 struct {long double,int}。

B.2.2 Fortran 77 语言 MPI 原始数据类型

Fortran 语言中表示 MPI 数据类型的变量类型是 INTEGER。

1. 基本数据类型

23	MPI_REAL	对应于 REAL。
24	MPI_INTEGER	对应于 INTEGER。
25	MPI_LOGICAL	对应于 LOGICAL。
26	MPI_DOUBLE_PRECISION	对应于 DOUBLE PRECISION。
27	MPI_COMPLEX	对应于 COMPLEX。
28	MPI_DOUBLE_COMPLEX	对应于 COMPLEX*16 或 COMPLEX*32。
29	MPI_CHARACTER	对应于 CHARACTER*1。

2. 其他数据类型

这些数据类型不一定在所有系统中都支持。

30	MPI_INTEGER1	对应于 INTEGER*1。
31	MPI_INTEGER2	对应于 INTEGER*2。
32	MPI_INTEGER4	对应于 INTEGER*4。
33	MPI_REAL4	对应于 REAL*4。
34	MPI_REAL8	对应于 REAL*8。

3. MPI_MAXLOC 和 MPI_MINLOC 中使用的数据类型

35	MPI_2INTEGER	对应于 INTEGER BUF(2)。
36	MPI_2REAL	对应于 REAL BUF(2)。
37	MPI_2DOUBLE_PRECISION	对应于 DOUBLE PRECISION BUF(2)。
38	MPI_2COMPLEX	对应于 COMPLEX BUF(2)。
39	MPI_2DOUBLE_COMPLEX	对应于 COMPLEX*16 BUF(2)。

B.2.3 预定义的通信器与进程组

MPI 的通信器和进程组在 C 语言中的变量类型分别为 MPI_Comm 和 MPI_Group，它们在 Fortran 语言中都用 INTEGER 表示。

40	MPI_COMM_WORLD	包含所有进程的通信器。
41	MPI_COMM_SELF	只包含本进程的通信器。
42	MPI_GROUP_EMPTY	空进程组 (不包含任何进程)。

1. 通信器或进程组的比较结果

43	MPI_IDENT	表示两个通信器或进程组完全一样。
44	MPI_CONGRUENT	表示两个通信器包含的进程组一样 (参看 MPI_Comm_compare)。
45	MPI_SIMILAR	表示两个通信器或进程组中的进程集合一样，但进程排序不同。
46	MPI_UNEQUAL	表示两个通信器或进程组不相同。

B.2.4 用于归约函数的预定义的二目运算

MPI 用于进行归约运算的函数有 MPI_Reduce、MPI_Allreduce、MPI_Reduce_scatter 和 MPI_Scan，它们所使用的二目运算在 C 中的类型为 MPI_Op，在 Fortran 中的类型为 INTEGER。

MPI_MAX	两个操作数中较大的一个。
MPI_MIN	两个操作数中较小的一个。
MPI_SUM	两个操作数之和。
MPI_PROD	两个操作数之积。
MPI_LAND	两个操作数的逻辑与 (logical and)。
MPI_BAND	两个操作数的按位与 (bitwise and)。
MPI_LOR	两个操作数的逻辑或 (logical or)。
MPI_BOR	两个操作数的按位或 (bitwise or)。
MPI_LXOR	两个操作数的逻辑异或 (logical xor)。
MPI_BXOR	两个操作数的按位异或 (bitwise xor)。
MPI_MINLOC	两对操作数中较小一个的值和位置。
MPI_MAXLOC	两对操作数中较大一个的值和位置。

B.2.5 C 变量类型及预定义函数

MPI_Status	存储通信状态的变量 (参看 B.2.9)。
MPI_Aint	存放地址或位移的变量。
MPI_Datatype	数据类型变量。
MPI_Request	通信请求变量。
MPI_Comm	通信器变量。
MPI_Group	进程组变量。
MPI_Op	归约操作的二目运算操作句柄。
MPI_User_function	聚合通信中的自定义函数 (参看 MPI_Op_create)。
MPI_Copy_function	通信器属性复制函数 (参看 MPI_Keyval_create)。

68	MPI_Delete_function	通信器属性删除函数 (参看 MPI_Keyval_create)。
69	MPI_NULL_COPY_FN	预定义的属性拷贝函数。
70	MPI_NULL_DELETE_FN	预定义的属性删除函数。
71	MPI_DUP_FN	预定义的属性复制函数。
72	MPI_Errhandler	错误处理函数句柄。
73	MPI_Handler_function	错误处理函数 (参看 MPI_Errhandler_create)。
74	MPI_ERRORS_ARE_FATAL	预定义的错误处理函数：发生错误则立即退出 (默认行为)。
75	MPI_ERRORS_RETURN	预定义的错误处理函数：发生错误时返回错误码，程序继续运行。

B.2.6 空对象

76	MPI_COMM_NULL	空通信器。
77	MPI_OP_NULL	空操作。
78	MPI_GROUP_NULL	空进程组。
79	MPI_DATATYPE_NULL	空数据类型。
80	MPI_REQUEST_NULL	空请求 (空回执)。
81	MPI_ERRHANDLER_NULL	空错误处理过程。

B.2.7 MPI 常量

82	MPI_TAG_UB	最大标签值 (不小于 $2^{16}-1$)。
83	MPI_HOST	该变量给出主机所在的进程号 (如果有主机的话)。
84	MPI_IO	具有输入、输出能力的进程号。
85	MPI_WTIME_IS_GLOBAL	代表 MPI_Wtime 函数返回的时间是否是全局同步的。
86	MPI_MAX_PROCESSOR_NAME	给出 MPI_Get_processor_name 返回的处理器名称最大长度。

87	MPI_MAX_ERROR_STRING	给出 MPI_Error_string 返回的错误信息的最大长度。
88	MPI_UNDEFINED	被许多 MPI 函数用于表示未知或未定义的整数值。
89	MPI_UNDEFINED_RANK	未定义的进程号。
90	MPI_KEYVAL_INVALID	用于表示非法或未定义的 keyvalue。
91	MPI_BSEND_OVERHEAD	给出 MPI_Bsend 附加的额外数据长度。
92	MPI_PROC_NULL	空进程，与空进程进行通信相当于空操作。
93	MPI_ANY_SOURCE	接收操作中用于表示从任何源地址接收。
94	MPI_ANY_TAG	接收操作中用于表示接收任何标签的消息。
95	MPI_BOTTOM	表示 MPI 地址空间的基底地址(参看 MPI_Address)。

B.2.8 进程拓扑结构

96	MPI_GRAPH	图结构。
97	MPI_CART	笛卡尔结构。

B.2.9 通信状态信息

C 语言中，MPI 利用结构 MPI_Status 来返回消息传递的完成情况。该结构中包含下述成员可供调用程序查询：

```
typedef struct {
    ... ...
    int MPI_SOURCE;     /* 消息源地址 */
    int MPI_TAG;        /* 消息标签 */
    int MPI_ERROR;      /* 错误码 */
    ... ...
} MPI_Status;
```

而在 Fortran 77 中则利用一个长度为 MPI_STATUS_SIZE 的整型数组来返回消息完成情况 (称为状态数组)，MPI_SOURCE、MPI_TAG 和 MPI_ERROR 表示状态数组中的位置。例如，假设 status 是一个状态

数组，则 status(MPI_SOURCE) 为消息源地址，status(MPI_TAG) 为消息标签，status(MPI_ERROR) 为错误码。

98	MPI_STATUS_SIZE	Fortran 接口中存储通信状态的数组长度。
99	MPI_SOURCE	通信状态数组中存放消息源地址的位置。
100	MPI_TAG	通信状态数组中存放消息源标签的位置。
101	MPI_ERROR	通信状态数组中存放错误码的位置。

B.2.10 错误码

102	MPI_SUCCESS	操作成功。
103	MPI_ERR_BUFFER	非法缓冲区指针。
104	MPI_ERR_COUNT	非法个数。
105	MPI_ERR_TYPE	非法数据类型。
106	MPI_ERR_TAG	非法消息标签。
107	MPI_ERR_COMM	非法通信器。
108	MPI_ERR_RANK	非法进程号。
109	MPI_ERR_ROOT	非法根进程。
110	MPI_ERR_GROUP	非法进程组。
111	MPI_ERR_OP	非法归约运算操作。
112	MPI_ERR_TOPOLOGY	非法进程拓扑结构。
113	MPI_ERR_DIMS	非法维数。
114	MPI_ERR_ARG	非法参数。
115	MPI_ERR_UNKNOWN	未知错误。
116	MPI_ERR_TRUNCATE	接收数据时消息被截断。
117	MPI_ERR_OTHER	其他错误，错误信息可通过 MPI_Error_string 获得。
118	MPI_ERR_INTERN	内部错误。
119	MPI_ERR_IN_STATUS	错误码在状态变量的 MPI_ERROR 元素中 (参看 MPI_Status)。
120	MPI_ERR_PENDING	有尚未完成的请求。
121	MPI_ERR_REQUEST	非法请求 (MPI_Request)。

122 MPI_ERR_LASTCODE 该值位于错误码列表的最后。

B.2.11 MPI–2 用于文件输入、输出的常量与类型

123 MPI_File MPI 文件句柄的变量类型。
124 MPI_Info 自定义文件输入、输出附加信息的变量类型。
125 MPI_Offset 用于文件位移的变量类型。
126 MPI_INFO_NULL 表示空 MPI_Info 对象的常量。
127 MPI_FILE_NULL 表示空 (无效) 文件句柄。
128 MPI_ERR_NO_SUCH_FILE 错误码，表示文件不存在。
129 MPI_DISPLACEMENT_CURRENT 代表文件当前位移的常量。
130 MPI_MODE_RDONLY MPI 文件访问模式：只读。
131 MPI_MODE_RDWR MPI 文件访问模式：读写。
132 MPI_MODE_WRONLY MPI 文件访问模式：只写。
133 MPI_MODE_CREATE MPI 文件访问模式：如果文件不存在则创建一个新文件。
134 MPI_MODE_EXCL MPI 文件访问模式：创建文件时若文件存在则创建失败。
135 MPI_MODE_APPEND MPI 文件访问模式：打开后将文件指针置于文件结尾处。
136 MPI_MODE_DELETE_ON_CLOSE MPI 文件访问模式：关闭文件后将其删除。
137 MPI_MODE_UNIQUE_OPEN MPI 文件访问模式：只有当前程序访问该文件。
138 MPI_MODE_SEQUENTIAL MPI 文件访问模式：只对文件进行顺序读写。
139 MPI_SEEK_SET 将文件指针设为指定值。
140 MPI_SEEK_CUR 将文件指针加上指定值。
141 MPI_SEEK_END 将文件指针设为文件长度减去指定值。
142 MPI_ORDER_C 数组元素按 C 数组的顺序排列。
143 MPI_ORDER_FORTRAN 数组元素按 Fortran 数组的顺序排列。

B.3 初始化、退出与错误处理函数

144 `int MPI_Init(int *argc, char ***argv)`

初始化 MPI 系统。通常它应该是第一个被调用的 MPI 函数。除 `MPI_Initialized` 外，其他所有 MPI 函数仅在调用了该函数后才可以被调用。`argc` 和 `argv` 分别是命令行参数的个数和参数数组的指针 (通过 C 的 `main` 函数得到)，必须将它们如实传递给 MPI 系统。MPI 系统通过它们得到所需的参数，并且会将 MPI 系统专用的参数删除而仅留下供用户程序使用的参数。

参看 `MPI_Initialized`，`MPI_Finalize`，`MPI_Abort`。

145 `int MPI_Initialized(int *flag)`

用于检查 MPI 系统是否已经初始化。如果已经调用过 `MPI_Init` 则返回值 `flag != 0`，否则返回值 `flag == 0`。这是唯一可以在 `MPI_Init` 之前调用的函数。

参看 `MPI_Init`。

146 `int MPI_Finalize(void)`

退出 MPI 系统。所有 MPI 进程在正常退出前都必须调用该函数。它是 MPI 程序中最后一个被调用的 MPI 函数。调用 `MPI_Finalize` 后不允许再调用任何 MPI 函数。调用该函数前应该确认所有的 (非阻塞型) 通信均已完成。

参看 `MPI_Init`，`MPI_Initialized`，`MPI_Abort`。

147 `int MPI_Abort(MPI_Comm comm, int errorcode)`

调用该函数时表明因为出现了某种致命错误而希望立即终止 MPI 程序的执行。MPI 系统会尽量设法终止通信器 `comm` 中的所有进程。在 UNIX 系统环境中，`errorcode` 被作为进程的退出码 (exit code) 返回给操作系统。

参看 `MPI_Init` 和 `MPI_Finalize`。

148
```
int MPI_Get_processor_name(char *name, int *resultlen)
```

该函数返回运行本进程的处理器名称。参数 name 应该提供不少于 MPI_MAX_PROCESSOR_NAME 个字节的存储空间用于存放处理器名称。

149
```
int MPI_Errhandler_create(MPI_Handler_function *function,
                          MPI_Errhandler *errhandler)
```

注册异常处理函数。参数 function 为异常处理函数，errhandler 返回一个可用于 MPI_Errhandler_set 的句柄。function 应该是一个如下形式的 C 函数：

```
void function(MPI_Comm *comm, int *errcode, ...)
```

其中 comm 是与之相关联的通信器，errcode 为错误码 (它们都是输入参数，使用地址是为了方便编写 Fortran 异常处理函数)。其余参数与 MPI 的具体实现有关。

参看 MPI_Errhandler_free 和 MPI_Errhandler_set 等。

150
```
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
```

释放一个已注册的异常处理函数。

参看 MPI_Errhandler_create 和 MPI_Errhandler_set 等。

151
```
int MPI_Errhandler_set(MPI_Comm comm,
                       MPI_Errhandler errhandler)
```

为通信器 comm 指定异常处理函数。其中，参数 errhandler 给出异常处理函数，它可以是通过 MPI_Errhandler_create 注册的自定义异常处理函数，也可以是 MPI 预定义的异常处理函数。MPI 预定义的异常处理函数有：MPI_ERRORS_ARE_FATAL 和 MPI_ERRORS_RETURN。

参看 MPI_Errhandler_create 和 MPI_Errhandler_free 等。

152
```
int MPI_Errhandler_get(MPI_Comm comm,
                       MPI_Errhandler *errhandler)
```

获取通信器 comm 的异常处理函数，errhandler 返回异常处理函数的句柄。

参看 MPI_Errhandler_create 和 MPI_Errhandler_free 等。

```
153 int MPI_Error_string(int errorcode, char *string,
                        int *resultlen)
```

获取指定错误码的错误信息 (字符串)。string 的长度必须不小于 MPI_MAX_ERROR_STRING。

```
154 int MPI_Error_class(int errorcode, int *errorclass)
```

获取指定错误码的错误类。

B.4 点对点通信函数

B.4.1 阻塞型通信函数

```
155 int MPI_Send(void *buf, int count, MPI_Datatype datatype,
                int dest, int tag, MPI_Comm comm)
```

阻塞型消息发送。参数 buf 为发送缓冲区；count 为发送的数据个数；datatype 为发送的数据类型；dest 为消息的目的地址 (进程号)，其取值范围为 0 到 np − 1 间的整数 (np 代表通信器 comm 中的进程数) 或 MPI_PROC_NULL；tag 为消息标签，其取值范围为 0 到 MPI_TAG_UB 间的整数；comm 为通信器。

```
156 int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
                int source, int tag, MPI_Comm comm,
                MPI_Status *status)
```

阻塞型消息接收。参数 buf 为接收缓冲区；count 为数据个数，它是接收数据长度的上限，具体接收到的数据长度可通过调用 MPI_Get_count 函数得到；datatype 为接收的数据类型；source 为消息源地址 (进程号)，其取值范围为 0 到 np − 1 间的整数 (np 代表通信器 comm 中的进程数)，或 MPI_ANY_SOURCE，或 MPI_PROC_NULL；tag 为消息标签，其取值范围为 0 到 MPI_TAG_UB 间的整数或 MPI_ANY_TAG；comm 为通信器；status 返回接收状态，参看 MPI_Status。

157
```
int MPI_Sendrecv(void *sendbuf, int sendcnt,
                 MPI_Datatype sendtype, int dest,
                 int sendtag, void *recvbuf, int recvcnt,
                 MPI_Datatype recvtype, int source,
                 int recvtag, MPI_Comm comm,
                 MPI_Status *status)
```

该函数将一次发送调用和一次接收调用合并进行，它使得 MPI 程序更为简洁。更重要的是，它能够避免阻塞型通信函数由于消息收发配对不好而引起的程序死锁。使用该函数的一个限制是 `sendbuf` 和 `recvbuf` 必须指向不同的缓冲区。各参数与 `MPI_Send` 和 `MPI_Recv` 中的参数相对应。

158
```
int MPI_Sendrecv_replace(void *buf, int count,
                         MPI_Datatype datatype, int dest,
                         int sendtag, int source,
                         int recvtag, MPI_Comm comm,
                         MPI_Status *status)
```

该函数的功能与 `MPI_Sendrecv` 类似，但收发使用同一缓冲区。

159
```
int MPI_Bsend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
```

阻塞型缓冲模式消息发送，各参数的含义与 `MPI_Send` 同。该函数将 `buf` 中的数据拷贝到事先指定的缓冲区中然后立即返回，实际发送由 MPI 系统在后台进行。在调用该函数前必须调用 `MPI_Buffer_attach` 函数来指定发送缓冲区。各参数的含义与 `MPI_Send` 函数相同。

160
```
int MPI_Buffer_attach(void *buffer, int size)
```

指定 `buffer` 做为 `MPI_Bsend` 使用的缓冲区，缓冲区的最大长度为 `size` 字节。MPI 只允许提交一个供 `MPI_Bsend` 使用的缓冲区，应该保证该缓冲区足够容下用 `MPI_Bsend` 发送的消息 (每个消息所需的缓冲区长度通常等于消息中数据的长度加上一个常量，可以调用函数 `MPI_Type_size` 来查询一个消息实际需要的缓冲区长度)。

```
161 int MPI_Buffer_detach(void *bufferptr, int *size)
```

撤销在此之前通过 MPI_Buffer_attach 函数提交的缓冲区。在调用 MPI_Buffer_detach 前不应该修改缓冲区中的内容，以免破坏正在发送的消息。该函数的返回表明所有缓冲模式的消息发送均已完成。注意该函数中 bufferptr 和 size 均为输出参数。其中 bufferptr 返回所撤销的缓冲区的地址，而 size 则返回所撤销的缓冲区的长度。

```
162 int MPI_Probe(int source, int tag, MPI_Comm comm,
                 MPI_Status *status)
```

该函数等待一个符合条件的消息到达然后返回。参数 source、tag 和 comm 的含义与 MPI_Recv 函数相同，status 返回所到达的消息的有关信息。

```
163 int MPI_Ssend(void *buf, int count, MPI_Datatype datatype,
                 int dest, int tag, MPI_Comm comm)
```

略。

```
164 int MPI_Rsend(void *buf, int count, MPI_Datatype datatype,
                 int dest, int tag, MPI_Comm comm)
```

略。

B.4.2 非阻塞型通信函数

```
165 int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
                 int dest, int tag, MPI_Comm comm,
                 MPI_Request *request)
```

非阻塞型消息发送函数。该函数递交一个消息发送请求，要求 MPI 系统在后台完成消息的发送。MPI 系统为该发送创建一个请求并将请求的句柄通过 request 变量返回给调用它的进程，供随后查询/等待消息发送完成时用。其余参数的含义与 MPI_Send 函数相同。

166
```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

非阻塞型消息接收函数。该函数递交一个消息接收请求，要求 MPI 系统在后台完成消息的接收。MPI 系统为该接收创建一个请求并将请求的句柄通过 request 变量返回给调用它的进程，供随后查询/等待消息接收的完成时用。其余参数的含义与 MPI_Recv 函数相同。

167
```
int MPI_Ibsend(void *buf, int count,
               MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request)
```

略。

168
```
int MPI_Irsend(void *buf, int count,
               MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request)
```

略。

169
```
int MPI_Issend(void *buf, int count,
               MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request)
```

略。

170
```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

检测指定的通信请求，不论通信是否完成都立即返回。如果通信已经完成则在 flag 中返回 !0，(此时 status 中包含关于所完成的通信的信息，对于接收请求，status 返回的内容与 MPI_Recv 返回的一样；对于发送请求，status 返回的内容不确定)，相应的通信请求被释放，request 被置成 MPI_REQUEST_NULL。如果通信尚未完成则在 flag 中返回 0。

171
```
int MPI_Testall(int count,
                MPI_Request array_of_requests[],
                int *flag, MPI_Status array_of_statuses[])
```

检测数组 array_of_requests 中的 count 个请求是否已全部完成。如果已经全部完成则在 flag 中返回 !0，否则在 flag 中返回 0。当函数

返回值等于 MPI_ERR_IN_STATUS 时表明部分通信请求处理出错，此时调用程序可检查 array_of_statuses 中每个元素的 MPI_ERROR 成员的值来得到出错的通信请求的错误码。

172
```
int MPI_Testany(int count,
                MPI_Request array_of_requests[],
                int *index, int *flag, MPI_Status *status)
```

检测数组 array_of_requests 中的 count 个请求中任何一个的完成。返回时，index 包含已完成的通信请求在数组 array_of_requests 中的位置，其他参数的含义与 MPI_Wait 和 MPI_Test 函数相同。

173
```
int MPI_Testsome(int incount,
                 MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[])
```

检测数组 array_of_requests 中的 incount 个请求中是否已经部分完成。outcount 中返回的是成功完成的通信请求个数 (返回 0 表示所有通信请求都尚未完成)，array_of_indices 的前 outcount 个元素给出已完成的通信请求在数组 array_of_requests 及 array_of_statuses 中的位置。当函数返回值等于 MPI_ERR_IN_STATUS 时表明部分通信请求处理出错，此时调用程序可检查 array_of_statuses 中每个元素的 MPI_ERROR 成员的值来得到出错的通信请求的错误码。

174
```
int MPI_Test_cancelled(MPI_Status *status, int *flag)
```

检测一个通信请求是否已被取消，如果该通信请求已被取消，则 flag 中返回的值不等于 0，否则返回 0。

175
```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

该函数等待指定的通信请求完成然后返回。成功返回时，status 中包含关于所完成的通信的信息 (对于接收请求，status 返回的内容与 MPI_Recv 返回的一样；对于发送请求，status 的返回的值不确定)，相应的通信请求被释放，request 被置成 MPI_REQUEST_NULL。

176
```
int MPI_Waitall(int count,
                MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[])
```

等待数组 array_of_requests 中的 count 个请求全部完成然后返回。当函数返回值等于 MPI_ERR_IN_STATUS 时表明部分通信请求处理出错，此时调用程序可检查 array_of_statuses 中每个元素的 MPI_ERROR 成员的值来得到出错的通信请求的错误码。

177
```
int MPI_Waitany(int count,
                MPI_Request array_of_requests[],
                int *index, MPI_Status *status)
```

等待数组 array_of_requests 中的 count 个请求中任何一个的完成，然后返回。成功返回时，参数 index 中包含已经完成的通信请求在数组 array_of_requests 中的位置。

178
```
int MPI_Waitsome(int incount,
                 MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[])
```

等待数组 array_of_requests 中的 incount 个请求中部分请求的完成然后返回。outcount 返回成功完成的通信请求个数 (返回 0 表示所有通信请求都尚未完成)，array_of_indices 的前 outcount 个元素给出已完成的通信请求在数组 array_of_requests 及 array_of_statuses 中的位置。当函数返回值等于 MPI_ERR_IN_STATUS 时表明部分通信请求处理出错，此时调用程序可检查 array_of_statuses 中每个元素的 MPI_ERROR 成员的值来得到出错的通信请求的错误码。

179
```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status)
```

该函数用于探测符合条件的消息是否已经到达。如果有符合条件的消息到达则 flag 中返回 !0，否则 flag 中返回 0。当有符合条件的消息时，status 中包含关于该消息的信息。参数 tag 和 source 的含义与 MPI_Recv 函数相同。

180
```
int MPI_Cancel(MPI_Request *request)
```

取消一个尚未完成的通信请求，它在 MPI 系统中设置一个取消该通信请求的标志然后立即返回，实际取消操作由 MPI 系统在后台完成。调用 MPI_Cancel 后，仍需调用 MPI_Wait，MPI_Test，或 MPI_Request_free 等函数来完成并释放该通信请求。

181
```
int MPI_Request_free(MPI_Request *request)
```

该函数释放指定的通信请求及所占用的资源。如果与该通信请求相关联的通信尚未完成，则它会先等待通信的完成。若操作成功则 request 的值被置成 MPI_REQUEST_NULL。

B.4.3 持久通信函数

182
```
int MPI_Send_init(void *buf, int count,
                  MPI_Datatype datatype, int dest,
                  int tag, MPI_Comm comm,
                  MPI_Request *request)
```

创建一个非阻塞型持久消息发送请求。该函数并不开始实际消息的发送，而只是创建一个请求句柄，通过 request 参数返回给调用程序，留待以后实际消息发送时用。用 MPI_Send_init 创建的持久通信请求可通过反复调用 MPI_Start 或 MPI_Startall 来完成多次消息发送。其余参数的含义与 MPI_Send 函数相同。

183
```
int MPI_Recv_init(void *buf, int count,
                  MPI_Datatype datatype, int source,
                  int tag, MPI_Comm comm,
                  MPI_Request *request)
```

创建一个非阻塞型持久消息接收请求。该函数并不开始实际消息的接收，而只是创建一个请求句柄，通过 request 参数返回给调用程序，留待以后实际消息接收时用。用 MPI_Recv_init 创建的持久接收可通过反复调用 MPI_Start 或 MPI_Startall 来完成多次消息接收。其余参数的含义与 MPI_Recv 函数相同。

184
```
int MPI_Start(MPI_Request *request)
```

启动基于持久通信请求的通信。每次调用 MPI_Start 相当于调用一次相应的非阻塞型通信函数 (MPI_Isend 和 MPI_Irecv)。随后调用程序还需要调用 MPI_Wait 函数来等待通信的完成。

185
```
int MPI_Startall(int count,
                 MPI_Request array_of_requests[])
```

MPI_Startall 的作用与 MPI_Start 类似，但它可以一次启动由数组 array_of_requests 指定的 count 个通信。

186
```
int MPI_Bsend_init(void *buf, int count,
                   MPI_Datatype datatype, int dest,
                   int tag, MPI_Comm comm,
                   MPI_Request *request)
```

创建一个缓冲式持久消息发送请求。该函数并不开始实际消息的发送，而只是创建一个请求句柄并返回给调用程序，留待以后实际消息发送时用。用 MPI_Bsend_init 创建的持久通信请求可通过反复调用 MPI_Start 或 MPI_Startall 来完成多次消息发送。

参看 MPI_Send_init。

187
```
int MPI_Rsend_init(void *buf, int count,
                   MPI_Datatype datatype, int dest,
                   int tag, MPI_Comm comm,
                   MPI_Request *request)
```

创建一个就绪式持久消息发送请求，该函数并不开始实际消息的发送，而只是创建一个请求句柄并返回给调用程序，留待以后实际消息发送时用。用 MPI_Rsend_init 创建的持久通信请求可通过反复调用 MPI_Start 或 MPI_Startall 来完成多次消息发送。

参看 MPI_Send_init。

188
```
int MPI_Ssend_init(void *buf, int count,
                   MPI_Datatype datatype, int dest,
                   int tag, MPI_Comm comm,
                   MPI_Request *request)
```

创建一个同步式持久消息发送请求，该函数并不开始实际消息的发送，而只是创建一个请求句柄并返回给调用程序，留待以后实际消息发送时用。用 MPI_Ssend_init 创建的持久通信请求可通过反复调用 MPI_Start 或 MPI_Startall 来完成多次消息发送。

参看 MPI_Send_init。

B.5 数据类型与打包函数

189
```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

数据类型创建函数：新数据类型 newtype 由 count 个老数据类型 oldtype 按域 (extent) 连续存放构成。

参看 MPI_Type_commit 和 MPI_Type_free。

190
```
int MPI_Type_vector(int count, int blocklen, int stride,
                    MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

数据类型创建函数：新数据类型 newtype 由 count 个数据块构成，每个数据块由 blocklen 个连续存放、类型为 oldtype 的数据构成，相邻两个数据块的位移相差 stride × extent(oldtype) 个字节。

参看 MPI_Type_commit 和 MPI_Type_free。

191
```
int MPI_Type_hvector(int count, int blocklen,
                     MPI_Aint stride,
                     MPI_Datatype oldtype,
                     MPI_Datatype *newtype)
```

数据类型创建函数：新数据类型 newtype 由 count 个数据块构成，每个数据块由 blocklen 个连续存放、类型为 oldtype 的数据构成，相邻两个数据块的位移相差 stride 个字节。该函数与 MPI_Type_vector 的

区别在于：stride 在后者中以 oldtype 的域为单位，而在前者中以字节为单位。

参看 MPI_Type_commit 和 MPI_Type_free。

```
192 int MPI_Type_indexed(int count, int blocklens[],
                        int indices[], MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

数据类型创建函数：新数据类型 newtype 由 count 个数据块构成，数据块 i 由 blocklens[i] 个连续存放、类型为 oldtype 的数据构成，其字节位移为 indices[i] × extent(oldtype)。该函数与 MPI_Type_vector 的区别是数据块的长度可以不同，数据块间还可以不等距。

参看 MPI_Type_commit 和 MPI_Type_free。

```
193 int MPI_Type_hindexed(int count, int blocklens[],
                         MPI_Aint indices[],
                         MPI_Datatype oldtype,
                         MPI_Datatype *newtype)
```

数据类型创建函数：新数据类型 newtype 由 count 个数据块构成，数据块 i 由 blocklens[i] 个连续存放、类型为 oldtype 的数据构成，其字节位移为 indices[i]。该函数与 MPI_Type_indexed 的区别在于 indices 在前者中以字节为单位，而在后者中以老数据类型的域为单位。

参看 MPI_Type_commit 和 MPI_Type_free。

```
194 int MPI_Type_struct(int count, int blocklens[],
                       MPI_Aint indices[],
                       MPI_Datatype oldtypes[],
                       MPI_Datatype *newtype)
```

数据类型创建函数：新数据类型 newtype 由 count 个数据块构成，数据块 i 由 blocklens[i] 个连续存放、类型为 oldtypes[i] 的数据构成，其字节位移为 indices[i]。

这是 MPI 中最一般的类型构造函数。它与 MPI_Type_hindexed 的区别在于各数据块可以由不同的数据类型构成。

参看 MPI_Type_commit 和 MPI_Type_free。

195
```
int MPI_Type_commit(MPI_Datatype *datatype)
```

提交数据类型。所有新创建的数据类型在首次用于消息传递前必须进行提交。新数据类型提交后就可以和 MPI 原始数据类型完全一样地在消息传递中使用。如果一个数据类型仅被用于创建其他数据类型的中间步骤而并不直接在消息传递中使用，则不必将它提交，一旦基于它的其他数据类型创建完毕即可立即将其释放。

196
```
int MPI_Type_free(MPI_Datatype *datatype)
```

释放指定的数据类型。当一个数据类型不再需要时应该将它释放以便释放其所占用的内存。函数返回时将 datatype 置成 MPI_DATATYPE_NULL。释放一个数据类型不会影响正在进行的使用该数据类型的通信，也不会影响基于它创建的其他数据类型。

197
```
int MPI_Pack(void *inbuf, int incount,
             MPI_Datatype datatype, void *outbuf,
             int outsize, int *position, MPI_Comm comm)
```

该函数将缓冲区 inbuf 中的 incount 个类型为 datatype 的数据进行打包，打包后的数据放在缓冲区 outbuf 中，outsize 给出的是 outbuf 的总长度 (字节数，供函数检查打包缓冲区是否越界用)，comm 是发送打包数据将使用的通信器，position 用于保存打包缓冲区的当前位置，第一次调用 MPI_Pack 前调用程序应将 position 置为 0，随后 MPI_Pack 将自动修改它，使得它总是指向打包缓冲区中尚未使用部分的起始位置，每次调用 MPI_Pack 后 position 的值实际上就是已打包的数据的总长度。

参看 MPI_Unpack 和 MPI_Pack_size。

198
```
int MPI_Unpack(void *inbuf, int insize, int *position,
               void *outbuf, int outcount,
               MPI_Datatype datatype, MPI_Comm comm)
```

该函数进行数据拆包操作，它正好是 MPI_Pack 的逆操作，用打包方式发送的消息接收方必须进行拆包。该函数从 inbuf 中拆包 outcount 个类型为 datatype 的数据到 outbuf 中。函数中各参数的含义与 MPI_Pack 类似，只不过这里的 inbuf 和 insize 对应于 MPI_Pack 中的 outbuf 和

outsize，而 outbuf 和 outcount 则对应于 MPI_Pack 中的 inbuf 和 incount。

参看 MPI_Pack 和 MPI_Pack_size。

199
```
int MPI_Pack_size(int incount, MPI_Datatype datatype,
                  MPI_Comm comm, int *size)
```

由于 MPI 打包时会在用户数据中加入一些附加信息，因此打包后的数据大小不等于用户数据的实际大小。该函数用于计算数据打包后的大小，通常用于估计所需的打包缓冲区长度，它在 size 中返回 incount 个类型为 datatype 的数据在通信器 comm 中打包后的数据长度。

参看 MPI_Pack 和 MPI_Unpack。

200
```
int MPI_Address(void *location, MPI_Aint *address)
```

返回指定变量的“绝对”地址，可在构造数据类型时用于计算位移。该函数在 Fortran 77 中特别有用，因为 Fortran 77 没有提供通用的获取变量地址的手段。为便于 Fortran 77 代码使用 MPI_Address 函数返回的地址，MPI 定义了一个常量 MPI_BOTTOM，相当于绝对地址 0，因此，调用

```
CALL MPI_Address(BUFF,ADDRESS,IERR)
```

后，MPI_BOTTOM(ADDRESS) 与 BUFF 代表着同一个内存地址。

参看 MPI_BOTTOM。

201
```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

返回指定数据类型的大小，即其中所包含的实际数据的字节数。

参看 MPI_Type_extent，MPI_Type_lb，MPI_Type_ub。

202
```
int MPI_Type_extent(MPI_Datatype datatype,
                    MPI_Aint *extent)
```

返回指定数据类型的域。

参看 MPI_Type_size，MPI_Type_lb，MPI_Type_ub。

203
```
int MPI_Type_lb(MPI_Datatype datatype,
                MPI_Aint *displacement)
```

返回指定数据类型的下界。

参看 MPI_Type_size，MPI_Type_extent，MPI_Type_ub。

```
204 int MPI_Type_ub(MPI_Datatype datatype,
                   MPI_Aint *displacement)
```

返回指定数据类型的上界。

参看 MPI_Type_size，MPI_Type_extent，MPI_Type_lb。

```
205 int MPI_Get_count(MPI_Status *status,
                     MPI_Datatype datatype, int *count)
```

该函数在 count 中返回消息中实际数据的长度 (数据类型个数)。

```
206 int MPI_Get_elements(MPI_Status *status,
                        MPI_Datatype datatype, int *elements)
```

该函数与 MPI_Get_count 类似，但它返回消息中所包含的 MPI 原始数据类型个数。所返回的 count 值如果不等于 MPI_UNDEFINED 的话，则应该是函数 MPI_Get_count 所返回的 count 值的倍数。

```
207 int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)
```

(MPI–2 函数) 复制数据类型。

B.6 同步与聚合通信函数

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

该函数用于进程的同步。调用该函数后进程将等待直到通信器 comm 中的所有进程都调用了该函数才返回。

```
209 int MPI_Bcast(void *buffer, int count,
                MPI_Datatype datatype, int root,
                MPI_Comm comm)
```

广播。通信器 comm 中进程号为 root 的进程 (称为根进程) 将自己 buffer 中的内容发送给通信器中所有其他进程。参数 buffer、count 和 datatype 的含义与点对点通信函数 (如 MPI_Send 和 MPI_Recv) 相同。

210
```
int MPI_Gather(void *sendbuf, int sendcnt,
               MPI_Datatype sendtype, void *recvbuf,
               int recvcnt, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

收集相同长度的数据块。以 root 为根进程，所有进程 (包括根进程自己) 将 sendbuf 中的数据块发送给根进程，根进程将这些数据块按进程号的顺序依次放到 recvbuf 中。发送和接收的数据类型与长度必须相配，即发送和接收使用的数据类型必须具有相同的类型序列。参数 recvbuf，recvcnt 和 recvtype 仅对根进程有意义。

需要特别注意的是，在根进程中，参数 recvcnt 指分别从每个进程接收的数据长度，而不是从所有进程接收的数据长度之和。因此，当 sendtype 等于 recvtype 时，sendcnt 应该等于 recvcnt。

211
```
int MPI_Allgather(void *sendbuf, int sendcnt,
                  MPI_Datatype sendtype, void *recvbuf,
                  int recvcnt, MPI_Datatype recvtype,
                  MPI_Comm comm)
```

MPI_Allgather 与 MPI_Gather 类似，区别是所有进程同时将数据收集到 recvbuf 中，因此称为数据全收集。MPI_Allgather 相当于依次以 comm 中的每个进程为根进程调用普通数据收集函数 MPI_Gather，或者以任一进程为根进程调用一次普通收集，紧接着再对收集到的数据进行一次广播。

212
```
int MPI_Gatherv(void *sendbuf, int sendcnt,
                MPI_Datatype sendtype, void *recvbuf,
                int *recvcnts, int *displs,
                MPI_Datatype recvtype, int root,
                MPI_Comm comm)
```

收集不同长度的数据块。与 MPI_Gather 类似，但允许每个进程发送的数据块长度不同，并且根进程可以任意排放数据块在 recvbuf 中的位置。recvbuf，recvtype，recvcnts 和 displs 仅对根进程有意义。数组 recvcnts 和 displs 的元素个数等于进程数，用于指定从每个进程接收的数据块长度和它们在 recvbuf 中的位移，均以 recvtype 为单位。

213
```
int MPI_Allgatherv(void *sendbuf, int sendcnt,
                   MPI_Datatype sendtype, void *recvbuf,
                   int *recvcnts, int *displs,
                   MPI_Datatype recvtype, MPI_Comm comm)
```

不同长度数据块的全收集。参数与 MPI_Gatherv 类似。它等价于依次以 comm 中的每个进程为根进程调用 MPI_Gatherv，或是以任一进程为根进程调用一次普通收集，紧接着再对收集到的数据进行一次广播。

214
```
int MPI_Scatter(void *sendbuf, int sendcnt,
                MPI_Datatype sendtype, void *recvbuf,
                int recvcnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

散发相同长度数据块。根进程 root 将自己的 sendbuf 中的 np 个连续存放的数据块按进程号的顺序依次分发到 comm 的各个进程 (包括根进程自己) 的 recvbuf 中，这里 np 代表 comm 中的进程数。sendcnt 和 sendtype 给出 sendbuf 中每个数据块的大小和类型，recvcnt 和 recvtype 给出 recvbuf 的大小和类型，其中参数 sendbuf、sendcnt 和 sendtype 仅对根进程有意义。

需要特别注意的是，在根进程中，参数 sendcnt 指分别发送给每个进程的数据长度，而不是发送给所有进程的数据长度之和。因此，当 recvtype 等于 sendtype 时，recvcnt 应该等于 sendcnt。

215
```
int MPI_Scatterv(void *sendbuf, int *sendcnts,
                 int *displs, MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt,
                 MPI_Datatype recvtype, int root,
                 MPI_Comm comm)
```

散发不同长度的数据块。与 MPI_Scatter 类似，但允许 sendbuf 中每个数据块的长度不同并且可以按任意的顺序排放。sendbuf、sendtype、sendcnts 和 displs 仅对根进程有意义。数组 sendcnts 和 displs 的元素个数等于 comm 中的进程数，它们分别给出发送给每个进程的数据长度和位移，均以 sendtype 为单位。

216
```
int MPI_Alltoall(void *sendbuf, int sendcnt,
                 MPI_Datatype sendtype, void *recvbuf,
                 int recvcnt, MPI_Datatype recvtype,
                 MPI_Comm comm)
```

相同长度数据块的全收集散发：进程 i 将 sendbuf 中的第 j 块数据发送到进程 j 的 recvbuf 中的第 i 个位置，$i, j = 0, \ldots,$ np $- 1$ (np 代表 comm 中的进程数)。sendbuf 和 recvbuf 均由 np 个连续的数据块构成，每个数据块的长度/类型分别为 sendcnt/sendtype 和 recvcnt/recvtype。该操作相当于将数据在进程间进行一次转置。例如，假设一个二维数组按行分块存储在各进程中，则调用该函数可很容易地将它变成按列分块存储在各进程中。

217
```
int MPI_Alltoallv(void *sendbuf, int *sendcnts,
                  int *sdispls, MPI_Datatype sendtype,
                  void *recvbuf, int *recvcnts,
                  int *rdispls, MPI_Datatype recvtype,
                  MPI_Comm comm)
```

不同长度数据块的全收集散发。与 MPI_Alltoall 类似，但每个数据块的长度可以不等，并且不要求连续存放。各个参数的含义可参考函数 MPI_Alltoall，MPI_Scatterv 和 MPI_Gatherv。

218
```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
```

归约操作。假设 comm 中的进程数为 np，则该函数相当于在根进程 (root) 中计算：

```
for (k = 0; k < count; k++) {
    recvbuf[k] = 0;
    for (i = 0; i < np; i++) {
       recvbuf[k] = recvbuf[k] op (sendbuf[k] of process i)
    }
}
```

这里 op 是归约操作的二目运算。

参看附录 B.2.4 中的 MPI_Op_create，MPI_Op_free。

219
```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op,
                  MPI_Comm comm)
```

全归约。它与普通归约函数 MPI_Reduce 的操作类似，但所有进程将同时获得归约运算的结果。它除了比 MPI_Reduce 少一个 root 参数外，其余参数及含义与后者一样。MPI_Allreduce 相当于在 MPI_Reduce 后马上再调用 MPI_Bcast 广播归约结果。

参看附录 B.2.4 中的 MPI_Reduce 和 MPI_Op_create。

220
```
int MPI_Reduce_scatter(void *sendbuf, void *recvbuf,
                       int *recvcnts,
                       MPI_Datatype datatype, MPI_Op op,
                       MPI_Comm comm)
```

归约散发。该函数的作用相当于首先进行一次

$$\texttt{count} = \sum_{i=0}^{\texttt{np}-1} \texttt{recvcnts}[i]$$

的归约操作，然后再对归约结果进行散发操作，散发给第 i 个进程的数据块长度为 recvcnts[i]，这里 np 代表 comm 中的进程数。其他参数的含义与 MPI_Reduce 一样。

参看 B.2.4 和 MPI_Reduce。

221
```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm)
```

前缀归约 (或扫描归约)。与普通全归约 MPI_Allreduce 类似，但各进程依次得到部分归约的结果。确切地，进程 i 的 recvbuf 中包含前 $i+1$ 个进程的归约运算结果，$i = 0, \ldots, \texttt{np} - 1$，np 代表 comm 中的进程数。各参数的含义与 MPI_Allreduce 一样。

参看 B.2.4 和 MPI_Reduce。

222
```
int MPI_Op_create(MPI_User_function *func, int commute,
                  MPI_Op *op)
```

创建自定义的用于归约操作的二目运算。func 参数是用于完成该运算的函数的指针，commute 说明所定义的运算是否满足交换律 (commute 为非 0 表示满足交换律)。op 返回所创建的二目运算的指针，类型为 MPI_Op。一个运算创建后便和 MPI 预定义的运算一样，可以用在所有归约 (包括前缀归约) 函数中。负责完成二目运算的函数 func 应该具有如下形式的接口：

```
void func(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype);
```

其中 invec 和 inoutvec 指向参与运算的操作数 (operand)，函数返回时 inoutvec 中应该包含运算的结果；len 给出 invec 和 inoutvec 中的元素个数，它对应于函数 MPI_Reduce 中的 count；datatype 给出操作数的数据类型，它对应于函数 MPI_Reduce 中的 datatype。直观地说，函数 func 必须负责完成如下操作：

```
for (i = 0; i < *len; i++) {
    inoutvec[i] = invec[i] op inoutvec[i]
}
```

参看附录 B.2.4 中的 MPI_Reduce 和 MPI_Op_free。

223
```
int MPI_Op_free(MPI_Op *op)
```

释放自定义的归约操作二目运算。

B.7 进程组与通信器操作

B.7.1 进程组操作

224
```
int MPI_Group_size(MPI_Group group, int *size)
```

返回指定进程组的大小 (包含的进程个数)。与 MPI_Comm_size 函数类似。

225
```
int MPI_Group_rank(MPI_Group group, int *rank)
```

返回进程在指定进程组中的进程号。与 MPI_Comm_rank 函数类似.

226
```
int MPI_Group_compare(MPI_Group group1, MPI_Group group2,
                      int *result)
```

比较两个进程组并在 result 中返回比较结果。如果两个进程组包含的进程一样、各进程的进程号也一样则结果为 MPI_IDENT，如果两个进程组包含的进程一样但进程号不同则结果为 MPI_SIMILAR，否则结果为 MPI_UNEQUAL。

227
```
int MPI_Group_difference(MPI_Group group1,
                         MPI_Group group2,
                         MPI_Group *group_out)
```

group_out 中返回的新进程组由属于 group1 但不属于 group2 的进程构成，进程号按 group1 中的顺序进行编排。

228
```
int MPI_Group_intersection(MPI_Group group1,
                           MPI_Group group2,
                           MPI_Group *group_out)
```

group_out 中返回由 group1 与 group2 中的进程的交集构成的进程组。新进程组中进程号按 group1 中的顺序进行编排。

229
```
int MPI_Group_incl(MPI_Group group, int n, int *ranks,
                   MPI_Group *group_out)
```

新进程组 group_out 由老进程组 group 中的部分进程构成，这些进程的进程号由数组 ranks 给出，n 是数组 ranks 的元素个数。新进程组中的进程号亦由进程在 ranks 数组中的顺序决定，即进程组 group_out 中进程号为 i 的进程在老进程组 group 中的进程号为 ranks[i]。该函数也可用来对进程组中的进程进行重新排序。当参数 n $= 0$ 时将创建一个空进程组 MPI_GROUP_EMPTY。

230
```
int MPI_Group_excl(MPI_Group group, int n, int *ranks,
                   MPI_Group *newgroup)
```

该函数将进程组 group 的进程集合减去一个子集而得到一个新进程组 newgroup，减去的进程的进程号由数组 ranks 给出，n 是 ranks 中的进程数。新进程组的进程号保持进程在老进程组中顺序。

231
```
int MPI_Group_range_incl(MPI_Group group, int n,
                         int ranges[][3],
                         MPI_Group *newgroup)
```

将由一组进程号范围给出的进程集合组成一个新进程组。每个进程号范围通过一个三元数对 (起始进程号，终止进程号，步长) 描述，n 为进程号范围的个数。确切地说，构成新进程组的进程集合由老进程组中进程号属于下述集合的进程组成：

$$\{r \mid r = \texttt{ranges}[i][0] + k \cdot \texttt{ranges}[i][2],$$
$$k = 0, \ldots, \lfloor \frac{\texttt{ranges}[i][1] - \texttt{ranges}[i][0]}{\texttt{ranges}[i][2]} \rfloor, \quad i = 0, \ldots, \texttt{n-1}\}$$

上式中所有计算出的 r 必须互不相同，否则调用出错。新进程组中进程号按上式中先 k 再 i 的顺序编排。

232
```
int MPI_Group_range_excl(MPI_Group group, int n,
                         int ranges[][3],
                         MPI_Group *newgroup)
```

从老进程组中减去由一组进程号范围指定的进程而得到一个新进程组。该函数的结果正好是函数 MPI_Group_range_incl 的补集。各项参数的含义与 MPI_Group_range_incl 相同。

233
```
int MPI_Group_translate_ranks(MPI_Group group_a, int n,
                              int *ranks_a,
                              MPI_Group group_b,
                              int *ranks_b)
```

返回进程组 group_a 中的一组进程在进程组 group_b 中的进程号。数组 ranks_a 列出 group_a 中的进程号，而数组 ranks_b 则给出相应的进程在 group_b 中的进程号，n 为数组 ranks_a 和 ranks_b 中的进程个数。

```
234 int MPI_Group_union(MPI_Group group1, MPI_Group group2,
                       MPI_Group *group_out)
```

group_out 中返回由 group1 与 group2 的并集构成的进程组。新进程组中进程号的分配原则是先对属于 group1 的进程按 group1 中的顺序编号，再对属于 group2 但不属于 group1 的进程按 group2 中的顺序编号。

```
235 int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

在 group 参数中返回指定通信器包含的进程组。

```
236 int MPI_Group_free(MPI_Group *group)
```

释放指定的进程组。函数返回时会将 group 置成 MPI_GROUP_NULL 以防止释放后被误用。实际上，该函数只是将该进程组加上释放标志。只有基于该进程组的所有通信器均被释放后才会实际将其释放。

B.7.2 域内通信器操作

```
237 int MPI_Comm_size(MPI_Comm comm, int *size)
```

在 size 中返回指定通信器中的进程数。

```
238 int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

在 rank 中返回本进程在指定通信器中的进程号。

```
239 int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2,
                       int *result)
```

比较两个 (域内) 通信器并在 result 中返回比较结果。如果两个通信器代表同一个通信域，则结果为 MPI_IDENT，表示它们实际上是同一个通信器；如果两个通信器不代表同一通信域，但它们的进程组相同，即它们包含的进程相同并且进程号也相同，则结果为 MPI_CONGRUENT；如果两个通信器包含的进程相同但进程号不同，则结果为 MPI_SIMILAR；否则结果为 MPI_UNEQUAL。

240
```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                    MPI_Comm *comm_out)
```

创建一个包含指定进程组 group 的新通信器 comm_out。这个函数并不将 comm 的属性传递给 comm_out，而是为 comm_out 建立一个新的上下文。返回时，属于进程组 group 的进程中 comm_out 等于新通信器的句柄，而不属于进程组 group 的进程中 comm_out 则等于 MPI_COMM_NULL。

241
```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *comm_out)
```

生成一个与 comm 具有完全相同属性的新通信器 comm_out。注意，新通信器 comm_out 与老通信器 comm 代表着不同的通信域，因此在它们之间不能进行通信操作，即用 comm 发送的消息不能用 comm_out 来接收，反之亦然。该函数通常用在并行库中：在库函数的开头将调用程序提供的通信器参数复制产生一个新通信器，库函数中使用新通信器进行通信，而在库函数返回前将新通信器释放，这样可以确保库函数中的通信不会与程序的其他通信相互干扰。

242
```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *comm_out)
```

该函数按照由参数 color 给出的颜色将通信器中的进程分组，所有具有相同颜色的进程构成一个新通信器。新通信器中进程号按参数 key 值的大小排序，两个进程的 key 值相同时则按它们在老通信器 comm 中的进程号排序。返回时，comm_out 等于进程所属的新通信器的句柄。color 必须是非负整数或 MPI_UNDEFINED。如果 color = MPI_UNDEFINED，则表示进程不属于任何新通信器，返回时 comm_out = MPI_COMM_NULL。

243
```
int MPI_Comm_free(MPI_Comm *comm)
```

释放指定的通信器。函数返回时会将 comm 置成 MPI_COMM_NULL 以防止释放后被误用。实际上该函数只是将通信器加上释放标志。当所有引用该通信的操作全部完成后才会实际将其释放。

```
244 int MPI_Keyval_create(MPI_Copy_function *copy_fn,
                         MPI_Delete_function *delete_fn,
                         int *keyval, void *extra_state)
```

创建新的通信器属性 (attribute key) (略)。

```
245 int MPI_Keyval_free(int *keyval)
```

释放通信器属性 (略)。

```
246 int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

删除通信器中的指定属性 (略)。

```
247 int MPI_Attr_get(MPI_Comm comm, int keyval,
                    void *attr_value, int *flag)
```

获取通信器的指定属性的值 (略)。

```
248 int MPI_Attr_put(MPI_Comm comm, int keyval,
                    void *attr_value)
```

设定通信器的指定属性的值 (略)。

B.7.3 进程拓扑结构

```
249 int MPI_Topo_test(MPI_Comm comm, int *top_type)
```

查询拓扑结构类型。如果 `comm` 具有笛卡尔拓扑结构，则在 `top_type` 中返回 `MPI_CART`，如果 `comm` 具有图拓扑结构，则在 `top_type` 中返回 `MPI_GRAPH`，否则在 `top_type` 中返回 `MPI_UNDEFINED`。

1. 笛卡尔拓扑结构

```
250 int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                       int *dims, int *periods, int reorder,
                       MPI_Comm *comm_cart)
```

该函数从通信器 `comm_old` 出发创建一个具有笛卡尔拓扑结构的新通信器 `comm_cart`。`ndims` 给出进程网格的维数；数组 `dims` 给出每维中的进程数；数组 `periods` 则说明进程在各个维上的联接是否具有周期性，即该维中第一个进程与最后一个进程是否相联，周期的笛卡尔拓

扑结构也称为环面 (torus) 结构，periods[i] $\neq 0$ 表明第 i 维是周期的，否则则是非周期的；reorder 指明是否允许在新通信器 comm_cart 中对进程进行重新排序，在某些并行计算机上，根据处理机的物理联接方式及所要求的进程拓扑结构对进程重新排序有助于提高并行程序的性能。comm_cart 中各维进程数之积必须不大于 comm_old 中的进程数，即 $\prod_{i=0}^{\texttt{ndims}-1}$ dims[i] $\leqslant$ np，这里 np 代表 comm_old 中的进程数。如果 $\prod_{i=0}^{\texttt{ndims}-1}$ dims[i] $<$ np，则一些进程将不属于 comm_cart，这些进程的 comm_cart 参数返回 MPI_COMM_NULL。

参看 MPI_Dims_create。

251
```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

(辅助函数) 该函数当给定总进程数及维数时自动计算各维的进程数，使得它们的乘积等于总进程数，并且各维上的进程数尽量接近。确切地说，给定 nnodes 和 ndims，函数计算正整数 dims[i]，$i = 0, \ldots,$ ndims $- 1$，使得 $\prod_{i=0}^{\texttt{ndims}-1}$ dims[i] = nnodes 并且 dims[i] 的值尽量接近。该函数要求输入时 dims 中元素的值为非负整数，并且它仅修改 dims 中输入值为 0 的元素。因此调用程序可以指定一些维上的进程数而仅要求计算其他维上的进程数。

这个函数的局限是没有考虑实际数据在各维上的大小。例如在二维区域分解计算中，假如进程数为 4，计算网络为 100×400，则理想的进程拓扑结构应为 1×4，而用该函数计算出的结果是 2×2。

参看 MPI_Cart_create。

252
```
int MPI_Cart_map(MPI_Comm comm_old, int ndims, int *dims,
                 int *periods, int *newrank)
```

该函数在 newrank 中返回给定笛卡尔拓扑结构下当前进程的建议编号。参数 ndims，dims 和 periods 的含义与函数 MPI_Cart_create 中相同。

参看 MPI_Cart_create。

253
```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

在 ndims 中返回通信器 comm 的笛卡尔拓扑结构的维数。

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

给定一个进程在通信器 comm 中的笛卡尔坐标 coords，该函数在 rank 中返回进程在 comm 中的进程号。对于具有周期性的维，coords 中对应的坐标值允许“越界”，即小于 0 或大于等于相应维上的进程数。

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
                    int *coords)
```

给定一个进程在通信器 comm 中的进程号 rank，该函数在 coords 中返回进程在 comm 中的笛卡尔坐标；maxdims 给出数组 coords 的最大长度。

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,
                 int *periods, int *coords)
```

返回通信器 comm 的笛卡尔拓扑结构的详细信息。数组 dims、periods 和 coords 分别返回各维的进程数、是否周期及当前进程的笛卡尔坐标；参数 maxdims 给出数组 dims、periods 和 coords 的最大长度。

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
                   int displ, int *source, int *dest)
```

计算在具有笛卡尔拓扑结构的通信器 comm 一个给定维上进行数据平移 (如用 MPI_Sendrecv 将一块数据发送给该维上后面一个进程，同时接收从该维上前面一个进程发送来的数据) 的目的地址和源地址。输入参数 direction 是进行数据平移的维号 ($0 \leqslant$ direction $<$ ndims)；disp 给出数据移动的“步长”(绝对值) 和“方向”(正负号)；输出参数 rank_source 和 rank_dest 分别是平移操作的源地址和目的地址。

假设指定的维上的进程数为 d，当前进程在该维上的坐标为 i，源进程 rank_source 在该维上的坐标为 i_s，目的进程 rank_dest 在该维上的坐标为 i_d，如果该维是周期的，则：

$$i_s = i - \texttt{disp} \mod d$$

$$i_d = i + \texttt{disp} \mod d$$

否则：

$$i_s = \begin{cases} i - \texttt{disp}, & \texttt{if } 0 \leqslant i - \texttt{disp} < d \\ \texttt{MPI\_PROC\_NULL}, & \text{otherwise} \end{cases}$$

$$i_d = \begin{cases} i + \texttt{disp}, & \texttt{if } 0 \leqslant i + \texttt{disp} < d \\ \texttt{MPI\_PROC\_NULL}, & \text{otherwise} \end{cases}$$

```
258 int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
                    MPI_Comm *comm_new)
```

该函数将一个具有笛卡尔拓扑结构的通信器 comm 指定的维抽取出来，构成一个具有低维笛卡尔结构的新通信器 comm_new。数组 remain_dims 的元素用来指定哪些维被包含在新通信器中，remain_dims[i] $\neq 0$ 表示新通信器包含第 i 维，否则表示新通信器不包含第 i 维。

2. 图拓扑结构

```
259 int MPI_Graph_create(MPI_Comm comm_old, int nnodes,
                        int *index, int *edges, int reorder,
                        MPI_Comm *comm_graph)
```

从通信器 comm_old 出发，创建一个具有指定图拓扑结构的新通信器 comm_graph。新通信器的拓扑结构图通过参数 nnodes、index 和 edges 描述：nnodes 是图的结点数 (如果 nnodes 小于通信器 comm_old 的进程数，则一些进程将不属于新通信器 comm_graph，这些进程中参数 comm_graph 的返回值将为 MPI_COMM_NULL)，index[i] ($i = 0, \ldots,$ nnodes $- 1$) 给出结点 $0, \ldots, i$ 的邻居数之和，edges 则顺序给出所有结点的邻居的进程号。如果用 Neighbor(i) 表示第 i 个结点的邻居的进程号集合，则：

$$\texttt{Neighbor}(0) = \{\texttt{edges}[j] \mid 0 \leqslant j < \texttt{index}[0]\}$$

$$\texttt{Neighbor}(i) = \{\texttt{edges}[j] \mid \texttt{index}[i-1] \leqslant j < \texttt{index}[i]\}$$

$$i = 1, \ldots, \texttt{nnodes} - 1$$

参数 reorder 指示是否允许在新通信器中对进程进行重新编号 (与函数 MPI_Cart_create 类似)。

260
```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes,
                      int *nedges)
```

该函数在参数 nnodes 中返回通信器 comm 的拓扑结构图的结点数(等于 comm 中的进程数)，在参数 nedges 中返回通信器 comm 的拓扑结构图的边数。

261
```
int MPI_Graph_get(MPI_Comm comm, int maxindex,
                  int maxedges, int *index, int *edges)
```

该函数返回通信器 comm 的拓扑结构图中的 index 和 edges 数组。参数 maxindex 和 maxedges 分别限定数组 index 和 edges 的最大长度。

参看 MPI_Graph_create。

262
```
int MPI_Graph_map(MPI_Comm comm_old, int nnodes,
                  int *index, int *edges, int *newrank)
```

该函数在 newrank 中返回给定图结构下当前进程的建议编号。参数 nnodes、index 和 edges 的含义与函数 MPI_Graph_create 中相同。

参看 MPI_Graph_create。

263
```
int MPI_Graph_neighbors(MPI_Comm comm, int rank,
                        int maxneighbors, int *neighbors)
```

在数组 neighbors 中返回与通信器 comm 中的进程 rank 相邻的进程的进程号，maxneighbors 限定数组 neighbors 的最大长度。

264
```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank,
                              int *nneighbors)
```

返回指定进程的邻居数。

B.7.4 域间通信器操作

265
```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

略。

266
```
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
```

略。

267
```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
```

略。

268
```
int MPI_Intercomm_create(MPI_Comm local_comm,
                         int local_leader,
                         MPI_Comm peer_comm,
                         int remote_leader, int tag,
                         MPI_Comm *comm_out)
```

略。

269
```
int MPI_Intercomm_merge(MPI_Comm comm, int high,
                        MPI_Comm *comm_out)
```

略。

B.8 时间函数

270
```
double MPI_Wtime(void)
```

返回当前墙钟时间，以从某一固定时刻算起的秒数为单位。时钟精度可通过函数 MPI_Wtick 查询。在 C 接口中，这是唯一两个返回双精度值而非整型错误码的 MPI 函数之一。在 Fortran 77 接口中，这是唯一两个 FUNCTION 形式而非 SUBROUTINE 形式的 MPI Fortran 接口之一。

参看 MPI_Wtick。

271
```
double MPI_Wtick(void)
```

该函数给出 MPI_Wtime 函数的时钟精度，以秒为单位。例如，假设 MPI_Wtime 使用的硬件时钟频率为 1/1000 秒，则 MPI_Wtick 的返回值为 10^{-3}，表示用函数 MPI_Wtime 得到的时间精度为千分之一秒。在 C 中，这是唯一两个返回双精度值而非整型错误码的 MPI 函数之一。在 Fortran 77 中，这是唯一两个 FUNCTION 形式而非 SUBROUTINE 形式的 MPI Fortran 接口之一。

参看 MPI_Wtime。

B.9 MPI–2 文件输入、输出函数

272
```
int MPI_File_open(MPI_Comm comm, char *filename,
                  int amode, MPI_Info info, MPI_File *fh)
```

打开一个 MPI 文件。文件成功打开后，在参数 fh 中返回文件的句柄，供以后对该文件进行操作用；comm 指定打开文件的通信器，所有属于 comm 的进程必须同时调用该函数；filename 是打开的文件名，comm 中所有进程提供的文件名必须代表同一个文件；amode 给出文件的打开模式，MPI 定义了一组形为 MPI_MODE_XXXX 的访问模式，它们大部分的含义与 C 语言中普通文件的访问模式 (如 O_RDONLY，O_CREAT 等) 类似，参看 B.2.11，这些模式可以用二进制“或”运算进行叠加 (C 中为“|”，Fortran 77 中可以用“+”代替“或”，只要同一模式不出现两次以上)，comm 中所有进程必须提供同样的 amode 参数；输入参数 INFO (类型为 MPI_Info) 提供给 MPI 系统一些附加提示信息，它由 MPI 的具体实现定义，这里不做介绍。调用时可用常数 MPI_INFO_NULL 代替它，表示没有提示信息。

273
```
int MPI_File_close(MPI_File *fh)
```

关闭 MPI 文件。文件关闭完成后，文件句柄被释放，fh 被置成 MPI_FILE_NULL。调用程序应该确保调用该函数前所有与该文件有关的操作请求均已完成。这是一个聚合型函数，进程组中所有进程必须同时调用并且提供同样的参数。

274
```
int MPI_File_delete(char *filename, MPI_Info info)
```

删除指定文件。如果文件不存在，则返回 MPI_ERR_NO_SUCH_FILE 错误。要删除的文件通常应该是没打开的或已关闭的。

275
```
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
```

将指定文件的长度 (指从文件开头到文件结尾的字节数) 设成 size。如果当前文件长度大于 size，则文件将被截断成 size 字节。如果当前文件长度小于 size，则文件大小被设为指定长度 (此时并不一定为该文件

分配实际存储空间)。这是一个聚合型函数，进程组中所有进程必须同时调用并且提供同样的参数。

276
```
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
```

为文件预留空间。如果当前文件长度大于等于 size，则该函数不起任何作用。否则它将文件长度调整到 size 指定的大小，并且强制操作系统为文件分配好存储空间。这是一个聚合型函数，进程组中所有进程必须同时调用并且提供相同的参数。

277
```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
```

查询文件长度。在参数 size 中返回指定文件的当前长度。

278
```
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
```

查询打开文件的进程组。在参数 group 中返回与文件句柄 fh 相关联 (即打开该文件) 的进程组。调用程序应该负责在不再需要该进程组句柄时将其释放。

279
```
int MPI_File_get_amode(MPI_File fh, int *amode)
```

查询文件访问模式。在参数 amode 中返回文件句柄 fh 所对应的文件的访问模式。

280
```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
                      MPI_Datatype etype,
                      MPI_Datatype filetype,
                      char *datarep, MPI_Info info)
```

将文件视窗的起始位置设为 disp (从文件开头以字节为单位计算)，基本单元类型设为 etype，文件单元类型设为 filetype；参数 datarep 给出文件中的数据表示格式，一般用 “native” 即可；参数 info 用来重新指定附加信息。这是一个聚合型函数，进程组中所有进程必须同时调用。不同进程可以提供不同的 disp、filetype 和 info 参数，但必须提供相同的 datarep 参数和具有相同域的 etype 参数。如果文件打开时使用了模式 MPI_MODE_SEQUENTIAL，则 disp 只能取为 MPI_DISPLACEMENT_CURRENT (文件的当前位置)。

281
```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp,
                      MPI_Datatype *etype,
                      MPI_Datatype *filetype,
                      char *datarep)
```

获取文件当前视窗。参看 `MPI_File_set_view`。

282
```
int MPI_File_get_type_extent(MPI_File fh,
                             MPI_Datatype datatype,
                             MPI_Aint *extent)
```

查询一个 (内存中的) 数据类型在文件中的域 (当文件的数据表示格式不等于 `native` 时，数据类型在文件中的域可能与它在内存中的域不同)。

283
```
int MPI_Register_datarep(char *datarep,
     MPI_Datarep_conversion_function *read_conversion_fn,
     MPI_Datarep_conversion_function *write_conversion_fn,
     MPI_Datarep_extent_function *dtype_file_extent_fn,
     void *extra_state)
```

略。

284
```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

略。

285
```
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
```

略。

286
```
int MPI_File_read(MPI_File fh, void *buf, int count,
                  MPI_Datatype datatype,
                  MPI_Status *status)
```

独立指针阻塞型读。`fh` 为文件句柄，`buf`、`count` 和 `datatype` 分别为数据缓冲区地址、个数和类型，`status` 返回操作结果状态。

287
```
int MPI_File_write(MPI_File fh, void *buf, int count,
                   MPI_Datatype datatype,
                   MPI_Status *status)
```

独立指针阻塞型写。`fh` 为文件句柄，`buf`、`count` 和 `datatype` 分别为数据缓冲区地址、个数和类型，`status` 返回操作结果状态。

```
int MPI_File_read_all(MPI_File fh, void *buf, int count,
                      MPI_Datatype datatype,
                      MPI_Status *status)
```

独立指针阻塞型聚合式读。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

```
int MPI_File_write_all(MPI_File fh, void *buf, int count,
                       MPI_Datatype datatype,
                       MPI_Status *status)
```

独立指针阻塞型聚合式写。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

```
int MPI_File_read_all_begin(MPI_File fh, void *buf,
                            int count,
                            MPI_Datatype datatype)
```

独立指针阻塞型聚合式读开始。

fh 为文件句柄，buf、count 和 datatype 分别给出数据缓冲区地址、个数和类型。

```
int MPI_File_read_all_end(MPI_File fh, void *buf,
                          MPI_Status *status)
```

独立指针阻塞型聚合式读结束。fh 为文件句柄，buf 为数据缓冲区地址，status 返回操作结果状态。

```
int MPI_File_write_all_begin(MPI_File fh, void *buf,
                             int count,
                             MPI_Datatype datatype)
```

独立指针阻塞型聚合式写开始。

fh 为文件句柄，buf、count 和 datatype 分别给出数据缓冲区地址、个数和类型。

```
int MPI_File_write_all_end(MPI_File fh, void *buf,
                           MPI_Status *status)
```

独立指针阻塞型聚合式写结束。fh 为文件句柄，buf 为数据缓冲区地址，status 返回操作结果状态。

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset,
                     void *buf, int count,
                     MPI_Datatype datatype,
                     MPI_Status *status)
```

显式位移阻塞型读。fh 为文件句柄，offset 给出位移，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset,
                      void *buf, int count,
                      MPI_Datatype datatype,
                      MPI_Status *status)
```

显式位移阻塞型写。fh 为文件句柄，offset 给出位移，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

```
int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset,
                         void *buf, int count,
                         MPI_Datatype datatype,
                         MPI_Status *status)
```

显式位移阻塞型聚合式读。

fh 为文件句柄，offset 为位移，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

```
int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset,
                          void *buf, int count,
                          MPI_Datatype datatype,
                          MPI_Status *status)
```

显式位移阻塞型聚合式写。

fh 为文件句柄，offset 为位移，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

298
```
int MPI_File_read_at_all_begin(MPI_File fh,
                               MPI_Offset offset,
                               void *buf, int count,
                               MPI_Datatype datatype)
```

显式位移阻塞型聚合式读开始。fh 为文件句柄，offset 给出位移，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型。

299
```
int MPI_File_read_at_all_end(MPI_File fh, void *buf,
                             MPI_Status *status)
```

显式位移阻塞型聚合式读结束。fh 为文件句柄，buf 为数据缓冲区地址，status 返回操作结果状态。

300
```
int MPI_File_write_at_all_begin(MPI_File fh,
                                MPI_Offset offset,
                                void *buf, int count,
                                MPI_Datatype datatype)
```

显式位移阻塞型聚合式写开始。fh 为文件句柄，offset 给出位移，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型。

301
```
int MPI_File_write_at_all_end(MPI_File fh, void *buf,
                              MPI_Status *status)
```

显式位移阻塞型聚合式写结束。fh 为文件句柄，buf 为数据缓冲区地址，status 返回操作结果状态。

302
```
int MPI_File_iread(MPI_File fh, void *buf, int count,
                   MPI_Datatype datatype,
                   MPI_Request *request)
```

独立指针非阻塞型读。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，request 返回一个操作请求，与非阻塞型通信函数返回的操作请求作用一样，使用该请求可随后调用 MPI_Wait、MPI_Test 等函数等待文件操作的完成。

303
```
int MPI_File_iwrite(MPI_File fh, void *buf, int count,
                    MPI_Datatype datatype,
                    MPI_Request *request)
```

独立指针非阻塞型写。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，request 返回一个操作请求，与非阻塞型通信函数返回的操作请求作用一样，使用该请求可随后调用 MPI_Wait、MPI_Test 等函数等待文件操作的完成。

304
```
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset,
                      void *buf, int count,
                      MPI_Datatype datatype,
                      MPI_Request *request)
```

显式位移非阻塞型读。fh 为文件句柄，offset 给出位移，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，request 返回一个操作请求，与非阻塞型通信函数返回的操作请求作用一样，使用该请求可随后调用 MPI_Wait、MPI_Test 等函数等待文件操作的完成。

305
```
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset,
                       void *buf, int count,
                       MPI_Datatype datatype,
                       MPI_Request *request)
```

显式位移非阻塞型写。fh 为文件句柄，offset 给出位移，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，request 返回一个操作请求，与非阻塞型通信函数返回的操作请求作用一样，使用该请求可随后调用 MPI_Wait、MPI_Test 等函数等待文件操作的完成。

306
```
int MPI_File_read_ordered(MPI_File fh, void *buf,
                          int count,
                          MPI_Datatype datatype,
                          MPI_Status *status)
```

共享指针阻塞型聚合式读。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

```
int MPI_File_write_ordered(MPI_File fh, void *buf,
                           int count,
                           MPI_Datatype datatype,
                           MPI_Status *status)
```

共享指针阻塞型聚合式写。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

```
int MPI_File_read_ordered_begin(MPI_File fh, void *buf,
                                int count,
                                MPI_Datatype datatype)
```

共享指针阻塞型聚合式读开始，各进程依照进程号的顺序读相应的数据块。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型。

```
int MPI_File_read_ordered_end(MPI_File fh, void *buf,
                              MPI_Status *status)
```

共享指针阻塞型聚合式读结束。fh 为文件句柄，buf 为数据缓冲区地址，status 返回操作结果状态。

```
int MPI_File_write_ordered_begin(MPI_File fh, void *buf,
                                 int count,
                                 MPI_Datatype datatype)
```

共享指针阻塞型聚合式写开始，各进程写入的数据块在文件中按进程号排列。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型。

```
int MPI_File_write_ordered_end(MPI_File fh, void *buf,
                               MPI_Status *status)
```

共享指针阻塞型聚合式写结束。fh 为文件句柄，buf 为数据缓冲区地址，status 返回操作结果状态。

```
int MPI_File_read_shared(MPI_File fh, void *buf,
                         int count, MPI_Datatype datatype,
                         MPI_Status *status)
```

共享指针阻塞型非聚合式读。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

```
313 int MPI_File_write_shared(MPI_File fh, void *buf,
                             int count,
                             MPI_Datatype datatype,
                             MPI_Status *status)
```

共享指针阻塞型非聚合式写。fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，status 返回操作结果状态。

```
314 int MPI_File_iread_shared(MPI_File fh, void *buf,
                             int count,
                             MPI_Datatype datatype,
                             MPI_Request *request)
```

共享指针非阻塞型非聚合式读。

fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，request 返回请求句柄。

```
315 int MPI_File_iwrite_shared(MPI_File fh, void *buf,
                              int count,
                              MPI_Datatype datatype,
                              MPI_Request *request)
```

共享指针非阻塞型非聚合式写。

fh 为文件句柄，buf、count 和 datatype 分别为数据缓冲区地址、个数和类型，request 返回请求句柄。

```
316 int MPI_File_seek(MPI_File fh, MPI_Offset offset,
                     int whence)
```

设定独立文件指针。参数 whence 可取 MPI_SEEK_SET、MPI_SEEK_CUR 或 MPI_SEEK_END。该函数与 C 语言中的 fseek 函数类似。

```
317 int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
```

查询独立文件指针，在参数 offset 中返回独立文件指针的当前值。

318
```
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset,
                         int whence)
```

设定共享文件指针。参数 whence 取为 MPI_SEEK_SET、MPI_SEEK_CUR 或 MPI_SEEK_END。该函数与 C 语言中的 fseek 函数类似，但它是一个聚合型函数，进程组中所有进程必须同时调用并且提供相同的参数。

319
```
int MPI_File_get_position_shared(MPI_File fh,
                                 MPI_Offset *offset)
```

查询共享文件指针，在参数 offset 中返回共享文件指针的当前值。

320
```
int MPI_File_get_byte_offset(MPI_File fh,
                             MPI_Offset offset,
                             MPI_Offset *disp)
```

计算文件指针在文件中的绝对地址。该函数将以 etype 为单位相对于当前文件视窗的文件指针位移值 (offset) 换算成以字节为单位从文件开头计算的绝对地址 (disp)。

321
```
int MPI_File_set_atomicity(MPI_File fh, int flag)
```

该函数设定是否需要保证打开文件的进程组中的进程对该文件的访问的原子性 (atomicity)。当 flag != 0 时，MPI 系统将保证文件访问的原子性从而保证属于 (与该文件相关联的) 同一进程组的进程对该文件访问的相容性。而当 flag == 0 时，MPI 不保证对文件访问的原子性，需要用户程序通过其他途径来避免因不同进程对文件的访问冲突而导致不可预料的结果。这是一个聚合型函数，进程组中所有进程必须同时调用并且提供相同的参数。

参看 MPI_File_get_atomicity。

322
```
int MPI_File_get_atomicity(MPI_File fh, int *flag)
```

查询文件访问原子性的当前设定，在参数 flag 中返回 atomicity 的当前值。

参看 MPI_File_set_atomicity。

323
```
int MPI_File_sync(MPI_File fh)
```

文件读写与存储设备间的同步。该函数确保将调用它的进程新近写入指定文件的数据写入存储设备。如果指定文件在存储设备中的内容已被其他进程改变，则它确保调用它的进程随后读该文件时得到的是改变后的数据。调用该函数时不能有尚未完成的对指定文件的非阻塞型或分裂型读写操作。注意，如果打开指定文件的进程组中一个进程往文件中写入一组数据，另一个进程希望从文件的同一位置读到这组数据，则可能需要调用两次 `MPI_File_sync`，并在两次调用间进行一次同步 (`MPI_Barrier`)。第一次调用 `MPI_File_sync` 可以确保第一个进程写的数据被写入存储设备，而第二次调用则可以确保新写入存储设备的数据被另一个进程读到。这是一个聚合型函数，进程组中所有进程必须同时调用并且提供相同的参数。

324
```
int MPI_Type_create_subarray( int ndims,
                              int *array_of_sizes,
                              int *array_of_subsizes,
                              int *array_of_starts,
                              int order,
                              MPI_Datatype oldtype,
                              MPI_Datatype *newtype)
```

构造数据类型的辅助函数，主要用于分布式数组的读写操作。它创建一个“子数组”数据类型，即描述一个 n 维 (全局) 数组中的一个 n 维子数组。所创建的新数据类型的域对应于全局数组，位移由子数组的元素在全局数组中的位置确定。参数 `ndims` 给出维数；`array_of_sizes[`i`]`、`array_of_subsizes[`i`]` 和 `array_of_starts[`i`]` 分别给出全局数组第 i 维的大小、子数组第 i 维的大小和子数组第 i 维在全局数组中的起始位置 (不论 C 还是 Fortran 均用 0 表示从全局数组的第一个元素开始)；参数 `order` 给出数组元素排列顺序，`MPI_ORDER_C` 表示数组元素按 C 数组的顺序排列，`MPI_ORDER_FORTRAN` 表示数组元素按 Fortran 数组的顺序排列；`oldtype` 给出数组元素的数据类型；`newtype` 返回所创建的子数组数据类型。子数组各维的大小必须大于 0 并且小于或等于全局数组相应维的大小。子数组的起始位置可以是全局数组中的任何位置，但必须确保子

数组被包含在全局数组中，否则函数调用将出错。如果数据类型 `oldtype` 是可移植数据类型，则新数据类型 `newtype` 也是可移植数据类型。

```
int MPI_File_get_errhandler(MPI_File fh,
                            MPI_Errhandler *errhandler)
```

略。

```
int MPI_File_set_errhandler(MPI_File fh,
                            MPI_Errhandler errhandler)
```

略。

参考文献

[1] J Dongarra, I Foster, G Fox, W Gropp, K Kennedy, L Torczon, A White. Sourcebook of Parallel Computing, Elsevier Science. 2003
中译本：莫则尧, 陈军, 曹小林等. 并行计算综论. 北京：电子工业出版社. 2005

[2] 李晓梅, 莫则尧, 文尚猛, 窦勇. 面向拓扑结构的并行算法设计与分析. 长沙：国防科技大学出版社. 1996

[3] 李晓梅, 蒋增荣. 同步并行算法. 长沙：湖南科学技术出版社. 1992

[4] 莫则尧. 实用的并行程序性能分析方法. 数值计算与计算机应用, 2000, 21(4):266–275

[5] 张宝琳, 谷同详, 莫则尧. 数值并行计算原理与方法. 北京：国防工业出版社. 1999

[6] V Kumar, A Gupta, *et al.* Introduction to Parallel Computing: Design and Analysis of Algorithm. In Redwood City: Benjamin/Cummings Publishing Company, Inc.. 1994

[7] Sun X H, D T Rover. Scalability of Parallel Algorithm-machine Combinations. IEEE Trans. on Parallel and Distributed Systems, 1994, 5(6):599-613

[8] OpenMP. `http://www.openmp.org/`

[9] TOP 500 list. `http://www.top500.org`

[10] TOP 500 Report for June 2005. `http://www.top500.org/lists/2005/06/Top500-Report-0605.pdf`

[11] 中国 TOP 100 排名. `http://www.samss.org.cn/TOP100/`

[12] D C Culler, J P Singh, A Gupta. Parallel Computer Architecture: A Hardware/Software Approach. In San Francisco: Morgan Kaufmann Publishers, Inc.. 1996

[13] Myrinet. `www.myrinet.org`

[14] Quadrics. www.quadrics.org

[15] InfiniBand. www.infinibandta.org

[16] MPICH — A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/

[17] LAM/MPI Parallel Computing. http://www.lam-mpi.org/

[18] MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[19] MPI–2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

[20] 莫则尧, 袁国兴. 消息传递并行编程环境 MPI. 北京：科学出版社. 2001

[21] 都志辉. 高性能计算并行编程技术 — MPI 并行程序设计. 北京：清华大学出版社. 2001

[22] D W Peaceman, H H Rachford. The Numerical Solution of Parabolic and Elliptic Differential Equations. J. SIAM, 1955, 3:28–41

[23] 张林波. 关于采用流水线方式进行一簇递推关系式的并行计算. 数值计算与计算机应用, 1999, 20(3):184–191

[24] C L Lawson, R J Hanson, D Kincaid, F T Krogh. Basic Linear Algebra Subprograms for FORTRAN Usage. ACM Trans. Math. Soft., 1979, 5:308–323

[25] J J Dongarra, J Du Croz, S Hammarling, R Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. ACM Trans. Math. Soft., 1998, 14:1–17

[26] J J Dongarra, J Du Croz, I S Duff, S Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Soft., 1990, 16:1–17

[27] E Anderson, Z Bai, C Bischof, S Blackford, J Demmel, J Dongarra, J Du Croz, A Greenbaum, S Hammarling, A McKennry, D Sorensen. LAPACK Users' Guide (3rd edition). In Philadelphia: Society for Industrial and Applied Mathematics. 1999

[28] LAPACK Frequently Asked Questions (FAQ). http://www.netlib.org/lapack/faq.html

[29] J Demmel, J Dongarra, J Du Croz, A Greenbaunm, S Hammarling, D Sorensen, Prospectus for the Development of a Linear Algebra Library for High-Performance Computers. LAPACK Working Note 01. 1987.

[30] L S Blackford, J Choi, A Cleary, E D'Azevedo, J Demmel, I Dhillon, J Dongarra, S Hammarling, G Henry, A Petitet, K Stanley, D Walker, R C Whaley. ScaLAPACK Users' Guide. SIAM, Philadelphia, PA. May 1997

[31] ScaLAPACK Homepage Reviewed in December 2004. http://www.netlib.org/scalapack/scalapack_home.html

[32] ScaLAPACK Frequently Asked Questions (FAQ) Reviewed in December 2004. http://www.netlib.org/scalapack/faq.html

[33] 晏益慧. 可扩展线性代数计算软件包 ScaLAPACK 及其实现. 第 8 届全国并行计算大会论文集, 258–262. 大连：大连理工大学出版社. 2004

[34] 罗水华, 杨广文, 张林波, 石威, 郑维民. 并行集群系统的 Linpack 性能测试分析. 数值计算与计算机应用, 2003, 24(4):285–292

[35] 陈江等. ScaLAPACK: 可扩展线性代数库. 超级计算通讯, 2004, 2(4)

[36] M Frigo, S G Johnson. FFTW User's Manual. http://www.fftw.org

[37] M Frigo, S G Johnson. The Fastest Fourier Transform in the West. http://www.fftw.org

[38] M Frigo, A Fast Fourier Transform Compiler. http://www.fftw.org

[39] M Frigo, S G Johnson. FFTW: An Adaptive Software Architecture for the FFT. http://www.fftw.org

[40] 潘文杰. 傅里叶分析及其应用. 北京：北京大学出版社. 2000

[41] S Balay *et. al.* PETSc 2.0 Users Manual. Technical Report ANL–95/11–Revision 2.1.3. Argonne National Laboratory. 2003

[42] M R Hestenes, E Stiefel. Method of Conjugate Gradients for Solving Linear Systems. J. Res. Nat. Bur. Stand., 1952, (49):pp409–436

[43] A Griewank. Evaluating Derivatives. SIAM, 2000

[44] 程强, 王斌, 马再忠. 自动微分转换系统及其应用. 数值计算与计算机应用, 2003, 4(3)

[45] 陈国良. 并行计算 — 结构 · 算法 · 编程. 北京：高等教育出版社. 2003

[46] 孙家昶, 张林波, 迟学斌, 汪道柳. 网络并行计算与分布式编程环境. 北京：科学出版社. 1996

[47] TOP 500 2004. http://www.top500.org/

[48] 冯圣中. 并行计算基础知识. 手稿. 2004

[49] 李庆扬, 王能超, 易大义. 数值分析 (第 4 版). 北京：清华大学出版社. 2001

[50] J M Ortega. Introduction to Parallel and Vector Solution of Linear Systems. In New York: Plenum Press. 1988

[51] J Dongarra, I Duff, D Soresen, H van der Vorst. Solving Linear Systems on Vector and Shared Memory Computers. SIAM, 1991

[52] G Golub, C van Loan. Matrix Computations. In Baltimore: The Johns Hopkins University Press, 1983
中译本: 廉庆荣, 邓健新, 刘秀兰译. 矩阵计算. 大连：大连理工大学出版社. 1988

[53] 迟学斌. 在具有局部内存与共享主存的并行机上并行求解线性方程组. 计算数学, 1995, 17(2)

[54] Li G Y, and T F Coleman. A Parallel Triangular Solver for a Hypercube Multiprocessor. 1986, TR 86–787, Cornell University

[55] 迟学斌. Transputer 上 Cholesky 分解的并行实现. 计算数学, 1993, 15(3)

[56] J M Delosme, I C F Ipsen. Positive Definite Systems with Hyperbolic Rotations. Linear Algebra Appl., 1986, 77:75–111

[57] D H Lawrie, A H Sameh. The Computation and Communication Com-

plexity of a Parallel Banded System Solver. ACM Trans. Math. Soft., 1984, 10:185–195

[58] 迟学斌. Transputer 上线性系统的并行求解. 中国计算机用户, 1991, (10)

[59] 陈景良. 并行数值方法. 北京：清华大学出版社. 1983

[60] 张宝琳, 袁国兴, 刘兴平, 陈劲. 偏微分方程并行有限差分方法. 北京：科学出版社. 1994

[61] D Chazan, W Miranker. Chaotic Relazation. J. Lin. Alg. Appl., 1969, 2:199–222

[62] G Baudet. Asynchronous Iterative Methods for Multiprocessors. J. ACM, 1978, 25:226–244

[63] J M Ortega, W C Rheinboldt. Iterative Solution of Nonlinear Equations in Several Variables. In New York: Academic Press. 1970

[64] 迟学斌. 线性方程组的异步迭代法. 计算数学, 1992, 14(3)

[65] 张林波. 利用 m4 宏语言进行 Fortran 77 循环展开. 数值计算与计算机应用, 1998, 19(1):49–63

[66] Intel VTune Performance Analysers. `http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm`

[67] 周铁，徐树方，张平文，李铁军. 计算方法. 北京：清华大学出版社. 2006

MPI 函数、变量索引

T

U

W

名词索引