高性能并行计算

迟学斌　中国科学院计算机网络信息中心
chi@sccas.cn, chi@sc.cnic.cn

http://lssc.cc.ac.cn/
http://www.sccas.cn/
http://www.scgrid.cn/
http://www.cngrid.org/

2005 年 4 月 6 日

目 录

教学要求

1. **教学目的：** 通过该课程的学习，使同学们了解和掌握并行计算机的发展、并行算法的基本概念、并行程序设计方法和并行实现环境，为从事并行计算研究和使用超级计算机系统奠定基础。
2. **内容简介：** 介绍并行计算机的发展，当今并行计算机的主流发展方向；并行计算的基本概念，并行算法的基本类型及设计技术；矩阵并行计算问题，重点介绍并行矩阵乘法，线性代数方程组并行求解方法，代数特征值问题的并行求解，经典迭代算法的并行化；并行程序设计技术，介绍并行程序的特点和实现技术；并行实现环境 MPI，介绍 MPI 过程管理函数，掌握用 MPI 编写并行程序的方法；MPI 点对点通讯函数，全局通讯函数，全局操作函数，进程组的操作；并行程序实现实例，剖析矩阵并行乘法的实现和求解方程组的实现。
3. **基础知识：** FORTRAN/C 语言，计算方法。
4. **参考资料：** 陈国良，并行计算 - 结构、算法、编程，高等教育出版社，2003 年 8 月。

第一部分

并行计算基础

第一章　预备知识

§1.1　并行计算的目标和内容

目标：求解大规模问题和复杂系统。早期并行计算以加速求解问题为目的，这来源于单处理机的计算速度受到物理上的限制，光速是其上限。随着计算在科学研究和实际应用中发挥越来越大的作用，人们对计算已经产生了依赖，将数值模拟作为许多决策的依据。现在人们已经习惯将计算作为科学研究的第三种手段，和传统的科学研究的理论方法和实验方法并列。

自 90 年代以来，并行计算得以空前的飞速发展，一方面，由于单处理机的计算速度不断提高，并行计算机的体系结构趋于成熟，数据传输网络的标准化和传输速率的大幅提升，使得并行计算机的研制周期能够从几年到几个月，为研制并行计算机系统创造了有利条件。另一方面，推动并行计算发展的主要动力来自于国际上的一些重要研究计划[1, 2]。

美国HPCC 计划：科学和工程计算需要能够提供 1TFLOPS 计算能力、1TB 内存容量、1TB/s 的 I/O 带宽，也就是 3T 性能目标。美国为了保持其在高性能计算与计算机通信领域的领先地位，在 1993 年，由科学、工程、技术联邦协调理事会向国会提交了“重大挑战项目：高性能计算与通信”的报告，也就是被称为 HPCC 计划的报告，即美国总统科学战略项目，其目的是通过加强研究与开发解决一批重要的科学与技术挑战问题。该项目由四部分组成：

1) 高性能计算机系统（HPCS），内容包括今后几代计算机系统的研究、系统设计工具、先进的典型系统及原有系统的评价等；
2) 先进软件技术与算法（ASTA），内容有巨大挑战问题的软件支撑、新算法设计、软件分支与工具、计算技术及高性能计算研究中心等；
3) 国家科研与教育网（NREN），内容有中接站及 10 亿位级传输的研究与开发；
4) 基本研究与人类资源（BRHR），内容有基础研究、培训、教育及课程教材。

HPCC 计划中近期要解决的“巨大挑战”问题有：

1) 磁记录技术。要在一平方厘米的磁盘表面上压缩记录 10 亿位数据；
2) 新药研制。特别是防治癌症与艾滋病新药的研制；
3) 高速城市交通。新型低噪音飞机的研制，空气动力学的计算；
4) 催化剂设计。改变至今为止多数催化剂靠经验设计的习惯，转向计算机辅助设计，主要分析这些复杂系统的大规模量子化学模型；
5) 燃料燃烧原理。通过化学动力学计算，揭示流体力学的作用，研制新型发动机；
6) 海洋模型模拟。对海洋活动与大气流的热交换进行整体海洋模拟；

7) 臭氧层空洞。研究控制臭氧消耗过程的化学和动力学机制；

8) 数字解剖。如三维 CT 扫描图象处理，人脑主题模型，三维生物结构与四维时间结构；

9) 空气污染。计算模拟能提供有效控制污染传播的途径，揭示其物理与化学机理；

10) 蛋白质结构设计。使用计算机模拟，对蛋白质组成的三维结构进行研究；

11) 图象理解。实时绘制图象或动画；

12) 密码破译技术。破译长位数的密码，主要是寻找一个大数的两个素因子。

美国ASCI 计划：全面禁止核试验条约签订后，对核武器的研制只能通过在实验室的数值模拟来完成。1996 年 6 月由美国能源部提出了“加速战略计算创新”计划，也即 ASCI 计划项目。提出通过数值模拟来评估核武器的性能、安全性、可靠性、更新等。要求数值模拟达到高分辨率、高逼真度、三维、全物理、全系统的规模和能力。该计划被认为是与当年曼哈顿计划等同的一个巨大的挑战，它不仅需要自然科学家的参与，而且也需要与计算机等工业界的合作，提供保障 ASCI 计划中的应用所需的计算机平台。为此，美国三大核武器实验室（Lawrence Livermore、Los Alamos、Sandia 国家实验室）分别向三大计算机公司（Intel、IBM、SGI/Cray 公司）预订了峰值浮点运算速度超过 1TFLOPS 的并行计算机。目前已经在这些实验室投入使用的并行计算机系统，其峰值浮点运算速度已经超过了 50TFLOPS。

并行计算的内容涵盖非常广，因此很难能够对其进行全面描述，在这里将重点介绍并行计算机的体系结构，编程语言，并行算法，并行实现技术，并行应用软件等。

§1.2 并行计算机发展历程

§1.2.1 计算机系统发展简史

计算机的起源可以追溯到欧洲文艺复兴时期 [4]。16-17 世纪的思想解放和社会大变革，大大促进了自然科学技术的发展，其中制造一台能帮助人进行计算的机器，就是最耀眼的思想火花之一。

1614 年，苏格兰人 John Napier 发表了关于可以计算四则运算和方根运算的精巧装置的论文。1642 年，法国数学家 Pascal 发明能进行八位计算的计算尺。1848 年，英国数学家 George Boole 创立二进制代数学。1880 年美国普查人工用了 7 年的时间进行统计，而 1890 年，Herman Hollerith 用穿孔卡片存储数据，并设计了机器，仅仅用了 6 个周就得出了准确的数据（62622250 人）。1896 年，Herman Hollerith 创办了 IBM 公司的前身。

这些” 计算机”，都是基于机械运行方式，还没有计算机的灵魂：逻辑运算。而在这之后，随着电子技术的飞速发展，计算机开始了质的转变。

1943 年到 1959 年时期的计算机通常被称作第一代计算机。使用真空电子管，所有的

程序都是用机器码编写，使用穿孔卡片。1946 年，John W. Mauchly 和 J. Presper Eckert 负责研制的 ENIAC (Electronic Numerical Integrator and Computer) 是第一台真正意义上的数字电子计算机。重 30 吨，18000 个电子管，功率 25 千瓦。主要用于弹道计算和氢弹研制。

1949 年，科学杂志大胆预测" 未来的计算机不会超过 1.5 吨。" 真空管时代的计算机尽管已经步入了现代计算机的范畴，但其体积之大、能耗之高、故障之多、价格之贵大大制约了它的普及应用。直到 1947 年，Bell 实验室的 William B. Shockley、John Bardeen 和 Walter H. Brattain. 发明了晶体管，电子计算机才找到了腾飞的起点，开辟了电子时代新纪元。

1959 年到 1964 年间设计的计算机一般被称为第二代计算机。大量采用了晶体管和印刷电路。计算机体积不断缩小，功能不断增强，可以运行 FORTRAN 和 COBOL，接收英文字符命令。出现大量应用软件。尽管晶体管的采用大大缩小了计算机的体积、降低了其价格，减少了故障。但离人们的要求仍差很远，而且各行业对计算机也产生了较大的需求，生产更强、更轻便、更便宜的机器成了当务之急，而集成电路的发明，不仅仅使体积得以减小，更使速度加快，故障减少。

1958 年，在 Robert Noyce（INTEL 公司的创始人）的领导下，继发明了集成电路后，又推出了微处理器。

1964 年到 1972 年的计算机一般被称为第三代计算机。大量使用集成电路，典型的机型是 IBM360 系列。1972 年以后的计算机习惯上被称为第四代计算机。基于大规模集成电路，及后来的超大规模集成电路。计算机功能更强，体积更小。在这之前，计算机技术主要集中在大型机和小型机领域发展，但随着超大规模集成电路和微处理器技术的进步，计算机进入寻常百姓家的技术障碍已突破。特别是从 INTEL 发布其面向个人机的微处理器 8080 的同时，互联网技术、多媒体技术也得到了空前的发展，计算机真正开始改变人们的生活。

1976 年，Cray-1，第一台商用超级计算机问世。集成了 20 万个晶体管，每秒进行 1.5 亿次浮点运算。今天，INTEL 已经推出主频超过 3.0Ghz 的微处理器。

与整个人类的发展历程相比、与传统科学技术相比，计算机的历史才刚刚开始书写，我们正置身其中，感受其日新月异的变化，被计算机大潮裹挟着丝毫不得停歇。

§1.2.2 并行计算机发展简述

40 年代开始的现代计算机发展历程可以分为两个明显的发展时代：串行计算时代、并行计算时代。每一个计算时代都从体系结构发展开始，接着是系统软件（特别是编译器与操作系统）、应用软件，最后随着问题求解环境的发展而达到顶峰。创建和使用并行计算机的主要原因是因为并行计算机是解决单处理器速度瓶颈的最好方法之一。

并行计算机是由一组处理单元组成的，这组处理单元通过相互之间的通信与协作，以更快的速度共同完成一项大规模的计算任务。因此，并行计算机的两个最主要的组成部分是计算节点和节点间的通信与协作机制。并行计算机体系结构的发展也主要体现在计算节点性能的提高以及节点间通信技术的改进两方面。

60 年代初期，由于晶体管以及磁芯存储器的出现，处理单元变得越来越小，存储器也更加小巧和廉价。这些技术发展的结果导致了并行计算机的出现，这一时期的并行计算机多是规模不大的共享存储多处理器系统，即所谓大型主机（Mainframe）。IBM360 是这一时期的典型代表。

到了 60 年代末期，同一个处理器开始设置多个功能相同的功能单元，流水线技术也出现了。与单纯提高时钟频率相比，这些并行特性在处理器内部的应用大大提高了并行计算机系统的性能。伊利诺依大学和 Burroughs 公司此时开始实施 IlliacIV 计划，研制一台 64 个 CPU 的 SIMD 主机系统，它涉及到硬件技术、体系结构、I/O 设备、操作系统、程序设计语言直至应用程序在内的众多研究课题。不过，当一台规模大大缩小了的 16CPU 系统终于在 1975 年面世时，整个计算机界已经发生了巨大变化。

首先是存储系统概念的革新，提出虚拟存储和缓存的思想。IBM360/85 系统与 360/91 是属于同一系列的两个机型，360/91 的主频高于 360/85，所选用的内存速度也较快，并且采用了动态调度的指令流水线；但是，360/85 的整体性能却高于 360/91，唯一的原因就是前者采用了缓存技术，而后者则没有。

其次是半导体存储器开始代替磁芯存储器。最初，半导体存储器只是在某些机器被用作缓存，而 CDC7600 则率先全面采用这种体积更小、速度更快、可以直接寻址的半导体存储器，磁芯存储器从此退出了历史舞台。与此同时，集成电路也出现了，并迅速应用到了计算机中。元器件技术的这两大革命性突破，使得 IlliacIV 的设计者们在底层硬件以及并行体系结构方面提出的种种改进都大为逊色。

1976 年 CRAY-1 问世以后，向量计算机从此牢牢地控制着整个高性能计算机市场 15 年。CRAY-1 对所使用的逻辑电路进行了精心的设计，采用了我们如今称为 RISC 的精简指令集，还引入了向量寄存器，以完成向量运算。这一系列全新技术手段的使用，使 CRAY-1 的主频达到了 80MHz。

微处理器随着机器的字长从 4 位、8 位、16 位一直增加到 32 位，其性能也随之显著提高。正是因为看到了微处理器的这种潜力，卡内基 - 梅隆大学开始在当时流行的 DEC PDP11 小型计算机的基础上研制成功一台由 16 个 PDP11/40 处理机通过交叉开关与 16 个共享存储器模块相连接而成的共享存储多处理器系统 C.mmp。

从 80 年代开始，微处理器技术一直在高速前进。稍后又出现了非常适合于 SMP 方式的总线协议，而伯克利加州大学则对总线协议进行了扩展，提出了 Cache 一致性问题的处理方案。从此，C.mmp 开创出的共享存储多处理器之路越走越宽；现在，这种体系

结构已经基本上统治了服务器和桌面工作站市场。

同一时期，基于消息传递机制的并行计算机也开始不断涌现。80 年代中期，加州理工成功地将 64 个 i8086/i8087 处理器通过超立方体互连结构连结起来。此后，便先后出现了 Intel iPSC 系列、INMOS Transputer 系列，Intel Paragon 以及 IBM SP 的前身 Vulcan 等基于消息传递机制的并行计算机。

80 年代末到 90 年代初，共享存储器方式的大规模并行计算机又获得了新的发展。IBM 将大量早期 RISC 微处理器通过蝶形互连网络连结起来。人们开始考虑如何才能在实现共享存储器缓存一致的同时，使系统具有一定的可扩展性（Scalability）。90 年代初期，斯坦福大学提出了 DASH 计划，它通过维护一个保存有每一缓存块位置信息的目录结构来实现分布式共享存储器的缓存一致性。后来，IEEE 在此基础上提出了缓存一致性协议的标准。

90 年代以来，主要的几种体系结构开始走向融合。属于数据并行类型的 CM-5 除大量采用商品化的微处理器以外，也允许用户层的程序传递一些简单的消息；CRAY T3D 是一台 NUMA 结构的共享存储型并行计算机，但是它也提供了全局同步机制、消息队列机制，并采取了一些减少消息传递延迟的技术。

随着商品化微处理器、网络设备的发展，以及 MPI/PVM 等并行编程标准的发布，机群架构的并行计算机出现。IBM SP2 系列机群系统就是其中的典型代表。在这些系统中，各个节点采用的都是标准的商品化计算机，它们之间通过高速网络连接起来。

今天，越来越多的并行计算机系统采用商品化的微处理器加上商品化的互连网络构造，这种分布存储的并行计算机系统称为机群。国内几乎所有的高性能计算机厂商都生产这种具有极高性能价格比的高性能计算机，并行计算机就进入了一个新的时代，并行计算的应用达到了前所未有的广度和深度。

§1.3 目前世界高性能计算机的状况

并行计算机随着微处理芯片的发展，已经进入了一个新时代。目前并行计算机的性能已经接近 100TFLOPS，1000TFLOPS 的并行计算机正在规划之中。我国并行计算机的研制已经走在世界前列，正在研制生产 100TFLOPS 的巨型计算机系统。2003 年由联想公司生产的深腾 6800 在 2003 年 11 月世界TOP500 排名中位列第 14 名，2004 年曙光公司生产的曙光 4000A 在 2004 年 6 月的世界TOP500 排名中位列第 10 名，这是我国公开发布的高性能计算机在世界TOP500 中首次进入前十名，这标志着我国在并行计算机系统的研制和生产中已经赶上了国际先进水平，为提高我国的科学研究水平奠定了物质基础。表 1.1 是按照 2004 年 11 月公布的世界 TOP500 排名榜得出的 [3]。

从TOP500 的前 10 名表 1.1 来看，美国仍然是超级计算机的最大拥有者。按照世

表 1.1: 世界 TOP10 并行计算机

Rank	site country/year	computer processor/manufacturer	$R_{\max}$ R_{peak}
1	IBM/DOE United States/2004	BlueGene/L DD2 beta-System, 0.7GHz PowerPC 32768 / IBM	70720 91750
2	NASA/Ames United States/2004	SGI Altix 1.5GHz, Voltaire Infiniband 10160 / SGI	51870 60960
3	Earth Sim. Center Japan/2002	Earth-Simulator 5120 / NEC	35860 40960
4	Barcelona SC Spain/2004	eServer BladeCenter 2.2GHz PowerPC, Myrinet 3564 / IBM	20530 1363
5	LLNL United States/2004	Thunder Itanium2 Tiger4 1.4GHz, Quadrics 4096 / California Digital Corp.	19940 22938
6	LANL United States/2004	ASCI Q - AlphaServer SC45, 1.25GHz 8192 / HP	13880 20480
7	Virginia Tech United States/2004	System X Dual 2.3GHz Apple XServe, Infiniband 2200 / Self-made	12250 20240
8	IBM - Rochester United States/2004	BlueGene/L DD1 Prototype 0.5GHz PowerPC 8192 / IBM/ LLNL	11680 16384
9	NAVOCEANO United States	eServer pSeries 655 1.7GHz Power4+ 2944 / IBM	10310 20019
10	NCSA United States/2003	Tungsten PowerEdge1750 Xeon 3.06GHz, Myrinet 2500 / Dell	9819 15300

界TOP500 的统计数据来分析，美国在计算能力上占有近全世界的一半，在TOP500 中的所有计算机中拥有的数量超过 50%。

表 1.2: 世界TOP500 并行计算机体系结构分析

	count	share	$R_{\max}$ sum	R_{peak} sum	procrssor sum
Cluster	294	58.8	612373	1101130	209421
Constellations	106	21.2	122640	192482	44016
MPP	100	20	392468	578794	155192

从表 1.2 可以看出，目前世界高性能计算机的主流结构是机群。

§1.4 可扩展的并行计算机体系结构

根据指令流和数据流的不同，通常把计算机系统分为四类 [1, 2]:

- 单指令流单数据流（SISD）
- 单指令流多数据流（SIMD）
- 多指令流单数据流（MISD）
- 多指令流多数据流（MIMD）

并行计算机系统除少量早期的、专用的SIMD 系统外，绝大部分为MIMD 系统。目前主要的并行计算机系统有五种：

- 并行向量机（PVP，Parallel Vector Processor）；
- 对称多处理机（SMP,Symmetric Multiprocessor）；
- 大规模并行处理机（MPP，Massively Parallel Processor）；
- 机群（Cluster）；
- 分布式共享存储多处理机（DSM，Distributied Shared Memory）。

这五类计算机系统代表了当今世界并行计算机的主要体系结构，下面我们简单介绍一下SMP、DSM、MPP 和机群并行计算机系统。

§1.4.1 对称多处理机系统

图 1.1 是对称多处理机系统的简单结构，它由处理单元、高速缓存、总线或交叉开关、共享内存以及 I/O 等组成。

SMP **具有如下特征：**

- 对称共享存储：系统中的任何处理机均可直接访问任何内存模块的存储单元和 I/O 模块连接的 I/O 设备，且访问的延迟、带宽和访问成功率是一致的。所有内存模块

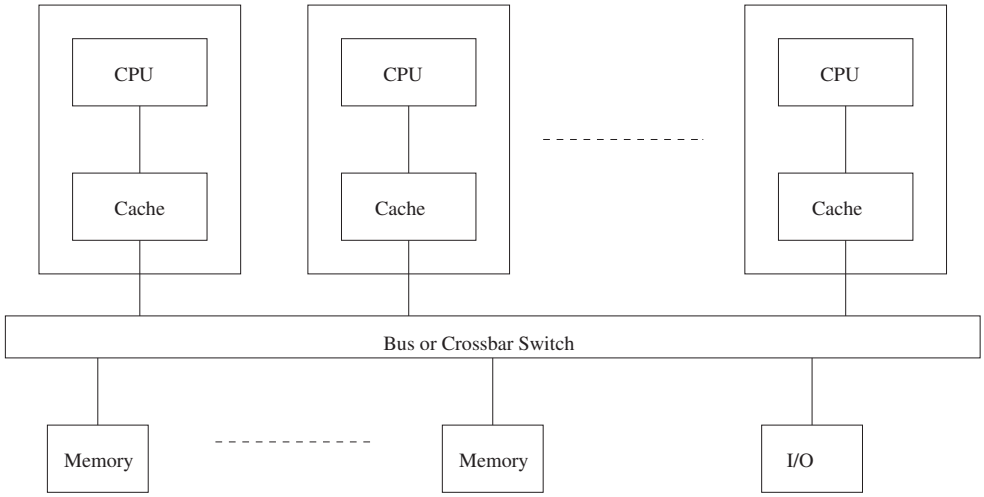


图 1.1: 对称多处理机系统

的地址单元是统一编码的，各个处理机之间的地位相同。操作系统可以运行在任意一个处理机上。

- 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，它根据各个处理机的负载情况，动态分配各个处理机的负载，并保持每个处理机的负载均衡。
- 局部高速缓存及其数据一致性：每个处理机均有自己的高速缓存，它们可以拥有独立的局部数据，但是这些数据必须保持与存储器中的数据是一致的。
- 低通信延迟：各个进程根据操作系统提供的读/写操作，通过共享数据缓存区来完成处理机之间的通信，其延迟通常远小于网络通信的延迟。
- 共享总线的带宽：所有处理机共享同一个总线带宽，完成对内存模块的数据和 I/O 设备的访问。
- 支持消息传递、共享存储模式的并行程序设计。

SMP **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全部瘫痪。
- 可扩展性差：由于所有处理机共享同一个总线带宽，而总线带宽每 3 年才增加 2 倍，跟不上处理机速度和内存容量的发展步伐。因此，SMP 并行计算机系统的处理机个数一般少于 64 个，也就只能提供每秒数百亿次的浮点运算性能。

SMP **的典型代表：**

- SGI Power Challenge XL 系列并行计算机（32 个 MIPS R10000 微处理器）
- COMPAQ Alphaserver 84005/440（12 个 Alpha 21264 微处理器）
- HP HP9000/T600（12 个 HP PA9000 微处理器）
- IBM RS6000/R40（8 个 RS6000 微处理器）

§1.4.2 分布共享存储处理机系统

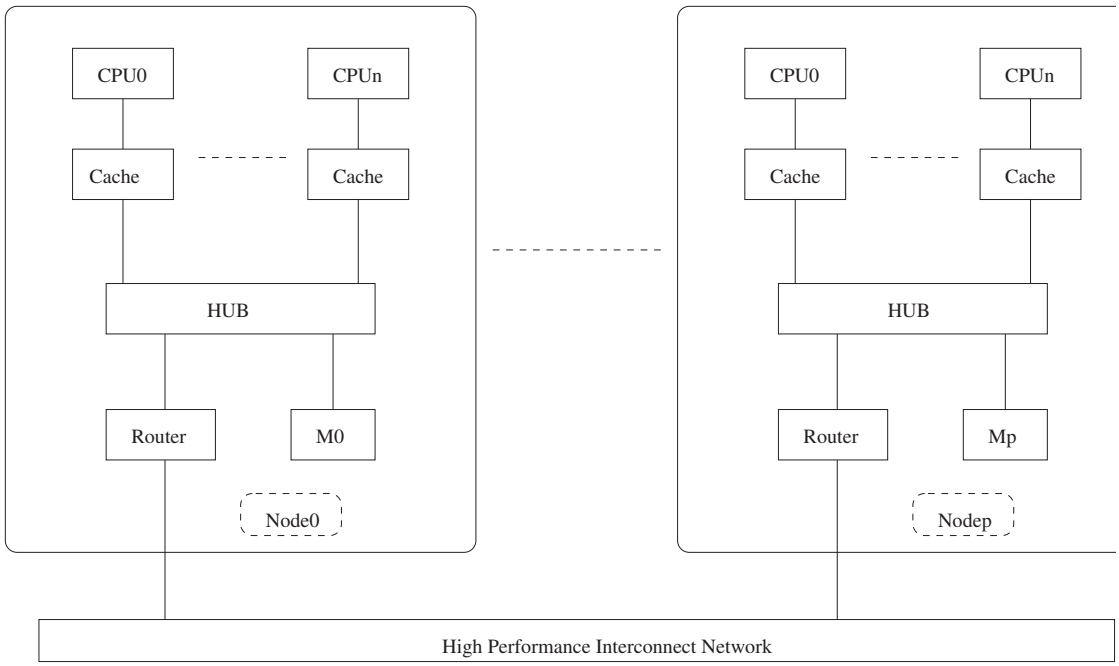


图 1.2: 分布共享存储处理机系统

图 1.2 是分布共享存储处理机系统，DSM 较好地改善了 SMP 的可扩展能力，是目前高性能计算机的主流发展方向之一。

DSM **的特点**：

- 并行计算机以节点为单位：每个节点由一个或多个 CPU 组成，每个 CPU 拥有自己的局部高速缓存 (Cache)，并共享局部存储器和 I/O 设备，所有节点通过高性能网络互联。
- 物理上分布存储：内存模块局部在各节点中，并通过高性能网络相互连接，避免了 SMP 访存总线的带宽瓶颈，增强了并行计算机系统的可扩展能力。
- 单一的内存地址空间：尽管内存模块分布在各个节点，但是所有这些内存模块都由硬件进行了统一编址，并通过互联网络联接形成了并行计算机的共享存储器。各个节点既可以直接访问局部内存单元，又可以访问其它节点的局部存储单元。
- 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 倍左右。
- 单一的操作系统映像：类似于SMP，在DSM 并行计算机中，用户只看到一个操作系统，它可以根据各个节点的负载情况，动态地分配进程。
- 基于高速缓存的数据一致性：通常采用基于目录的告诉缓存一致性协议来保证各节

点的局部高速缓存数据与存储器中的数据是一致的。同时，我们称这种DSM 并行计算机结构为 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节点间的访问延迟很小，通信带宽可以扩展。例如，目前最具代表性的DSM 并行计算机 SGI Origin 3000，它的双向点对点带宽可达 3.2GB/秒，而延迟小于 1 个微秒。
- 可扩展性高：DSM 并行计算机可扩展到上千个节点，能提供每秒数万亿次的浮点运算性能。
- 支持消息传递、共享存储并行程序设计。

DSM **的典型代表：**

- SGI Origin 2000、3000、3800；
- SGI Altix。

§1.4.3 大规模并行计算机系统

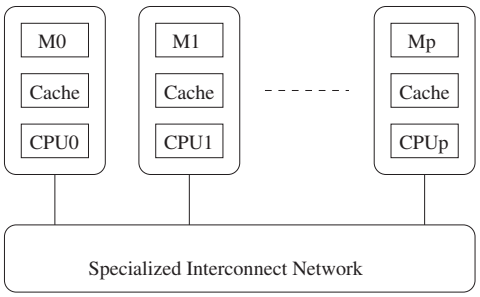


图 1.3: 大规模并行计算机系统

图 1.3 是一个大规模并行计算机系统的简单结构，它是并行计算机发展过程中的主力，现在已经发展到由上万个处理机构成一个系统，随着并行计算机的发展，几十万个处理机的超大规模系统也会在不久的将来问世。

MPP **的特点：**

- 节点数量多，成千上万，这些节点由局部网卡通过高性能互联网络连接。
- 每个节点都相对独立，并拥有一个或多个微处理机。这些微处理机都有局部高速缓存，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。
- MPP 的各个节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，由它来调度当前系统中有效的计算节点来执行该作业。同时，MPP 系统也允许用户登录到指定的节点，或到某些特定的节点上运行作业。
- 各个节点上的内存模块是相互独立的，且不存在全局内存单元的统一硬件编址。一般情况下，各个节点只能直接访问自身的局部内存模块。如果需要直接访问其它节点的内存模块，则必须有操作系统提供特殊的软件支持。

MPP 的典型代表：

- ICT Dawning 1000（32 个处理机）；
- IBM ASCI White（8192 个处理机）；
- Intel ASCI Red（9632 个处理机）；
- Cray T3E（1084 个处理机）

§1.4.4 机群系统

- Linux 机群系统已成为最流行的高性能计算平台，在高性能计算机中占有越来越大的比重
- 系统规模可从单机、少数几台联网的微机直到包括上千个结点的大规模并行系统
- 既可作为廉价的并行程序调试环境，也可设计成真正的高性能并行机
- 普及并行计算必不可少的工具
- 用于高性能计算的机群系统在结构上、使用的软件工具上通常有别于用于提供网络、数据库服务的机群 (后者亦称为服务器集群)
- *TOP500 Supercomputer Sites*
- *Cluster@TOP500*
- 参考资料: 用关键字 “cluster howto” 在网上搜索相关材料 ...

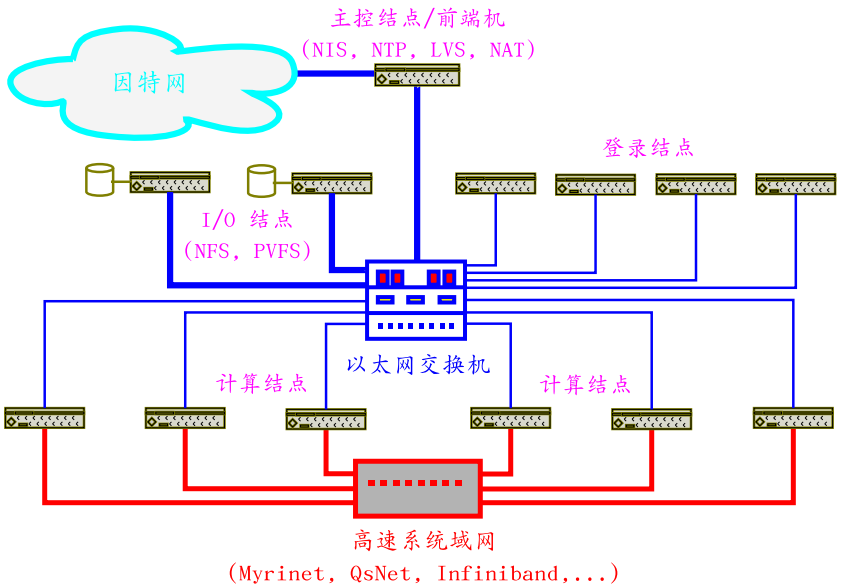


图 1.4: 典型 Linux 机群系统

§1.4.4.1 构建 Linux 机群的要素

- 单台或联网的多台微机或服务器

- Linux 系统: RedHat, Debian, SuSE, Mandrake, ...
- (可选) 高速内联网络: 千兆以太网，*Myrinet*, *QsNet*, *Dolphin SCI*, *Infiniband*, ...
- 编译系统: gcc/g++/g77, *PGI*, *Intel*, ...
- MPI 系统: *MPICH*, *LAM-MPI*, ...
- 网络文件系统: NFS, *PVFS*, *Lustre* ...
- 资源管理与作业调度: *PBS*, *Condor*, *LSF*, ...
- 数学库:
  BLAS *MKL*, *ATLAS*, *Kazushige Goto's BLAS* (推荐)
  FFTW *http://www.fftw.org/*
  LAPACK *http://www.netlib.org/lapack/*
  ScaLAPACK *http://www.netlib.org/scalapack/*
  ...
- 其它工具:
  PETSc *http://www-unix.mcs.anl.gov/petsc/petsc-2/*
  UG *http://cox.iwr.uni-heidelberg.de/~ug/*
  ...

§1.4.4.2 专用并行机群

指专门建造的用于并行计算的机群。

- 通常单独形成一个局域网，再通过一个网关连接到 Internet
- 通常使用内部 IP 地址 (如 192.168.0.x), 对外部而言只有网关是可见的
- 可根据需要设立一至数台服务器，分别承担网关、时钟同步 (NTP)、NIS/LDAP、网络文件系统 (NFS)、资源管理、用户登录、作业调度等服务
- 通过 IP 伪装 (*IP Masquerading*) 或网络地址转换 (*NAT*, Network Address Translation) 使得内部结点能够直接访问 Internet (ipchains 或 iptables)
- 利用*LVS* (Linux Virtual Server) 将外部用户分配到登录结点
- 大型机群通常采用安装在机柜中的机架式或刀片式服务器，并有专门配备的 UPS 及空调

§1.5 国内外超级计算中心状况

§1.5.1 美国超级计算中心简介

- San Diego **超级计算中心** (SDSC)成立于 1985 年，通过大力发展计算科学和高性能计算，推动国际科学和工程探索。随着这一传统宗旨逐渐步入先进计算设施（cyberinfrastructure）

的时代，SDSC 已经成为科学、企业、学术的战略资源，担负着数据管理、网格计算、生物信息、地理信息、高端计算以及其它科学工程学科的领导地位。SDSC 的使命是延伸科学成就，向社会提供高性能硬件技术、集成软件技术以及深度的跨学科专业技能。

- **美国国家超级计算应用中心** (NCSA)创建于 1986 年 1 月。NCSA 在高性能计算、网络、存储和数据分析领域具有较高国际声誉。它被一贯认为是创新性的科学工程系统和软件的先锋。NCSA 的宗旨是与可种研究领域合作，创建先进计算设施（cyberinfrastructure），探索新的科学发现。它的专业特长是应用最先进的计算机，构建融合软件应用、可视化工具、数据挖掘和分析工具的工作系统。这些创新系统将成为未来先进计算设施的核心，把分散系统连通成为单一、无缝的资源。NCSA 是 National Science Foundation 的 TeraGrid 计划的关键合作伙伴，该计划投资了一亿美元，帮助研究人员远程访问最快速的、顶级的超级计算机、可视化工具、应用软件、传感器、仪器以及大型存储设备。
- Pittsburgh **超级计算中心** 是 Carnegie Mellon 大学、Pittsburgh 大学以及 Westinghouse Electric 公司共同组建的。它成立于 1986 年，得到多个联邦部门以及 Pennsylvania 周政府和私人企业共同体的支持。其使命是向国家社会提供一流的计算资源，支持科学工程领域主要问题的解决；推动计算科学、计算技术以及国家信息设施的发展；向研究人员传授高性能技术以及它们的使用；帮助私人部门，开发高性能计算软件，促进他们的竞争优势。
- ...

§1.5.2 中国大陆超级计算中心简介

- **国家高性能计算中心（北京）** 是国家科委在曙光 1000 大规模并行计算机研制成功之后，于 1995 年 5 月批准成立的第一个高性能计算中心。
- **国家高性能计算中心（合肥）** 是国家科委于 1995 年 9 月 5 日批准成立的” 中国合肥高性能计算中心” 的前身，1998 年 5 月 18 日国家科技部将其易名为现名。中心隶属于安徽省科技厅、中国科学技术大学、中国科学院合肥分院和中国科学技术大学高等研究院。
- **国家高性能计算中心（成都）** 是 1 9 9 7 年 1 1 月 1 1 日在西南交通大学正式挂牌成立的。
- **国家高性能计算中心（武汉）** 成立于 1997 年 12 月 20 日，是经国家科委批准，由国家科技部、国家教育部、湖北省科技厅、武汉市科委和华中科技大学共同出资支持的。
- **国家高性能计算中心（上海）** 成立于 1998 年 4 月，位于上海复旦大学，由国家科技

部，上海市科委和复旦大学合资组建。

- **国家高性能计算中心（杭州）**于 2001 年 4 月 1 日，依托浙江省基因组信息学重点实验室、浙江大学基因组信息学研究所在杭州成立。
- **国家高性能计算中心（西安）** 于 2002 年 4 月 13 日在西安交通大学揭牌。被称为”中国西部 IT 航母” 信息化平台的启动，标志着我国西部的信息化建设步入全国先进行列。
- **山东大学高性能计算中心** 作为国家”211 工程” 首批重点建设的高等学校之一，山东大学近年来一直致力于提高高新技术研究水平。2004 年，山东大学投资数百万元，组建成立了该中心。
- **天津高性能计算中心** 于 2004 年 8 月 23 日在南开大学成立。旨在为天津培养高性能计算人才，推动大规模科学与工程计算的发展，抢占综合科技实力竞争的制高点。
- **北京应用物理与计算数学研究所 - 高性能计算中心** 拥有超级并行计算机以及上百台高性能计算工作站，是国内规模最大的超级计算中心之一。中心与所外建立了较广泛的学术交流与合作关系，承担多项国家自然科学基金课题，国家重大基础研究项目课题以及国家高技术研究项目。中心的主要任务是为研究所的高性能计算提供相关的科学与技术支持，主要包括：
  1. 高性能并行计算机环境的建设与管理
  2. 高性能并行计算软件与理论研究
  3. 大规模科学计算可视化软件与理论研究
  4. 大规模数据管理软件与理论研究
  5. 计算机网络技术的开发与应用
  6. 虚拟现实技术研究
- **上海超级计算中心**成立与 2000 年底，是上海市信息化建设的一个重要基础设施。该中心以上海市的科研教育和工业发展为基础，为汽车碰撞模拟、船舶设计等提供高性能计算能力。目前拥有国内 TOP100 排名第一的超级计算机曙光 4000A。
- **中国科学院的超级计算中心** 成立于 1996 年，为中国科学院的科学研究提供计算环境，是中国科学院信息化建设的重要基础设施之一。
  1. 硬件环境：
     A 5 万亿次面向网格的超级计算机系统深腾 6800，包括 265 个四路结点机，主频为 1.3GHz 的安腾 2 处理芯片（其中 1024 个处理机用于计算），内存总容量为 2.6TB，磁盘存储总容量为 80TB（其中，光纤盘阵容量 61TB）。高速连接网络为 QsNet(Quadrics 公司产品)，点对点通信带宽大于每秒 300MB，延迟时间小于 7 微秒。
     B SGI Onyx 350，32 个 MIPS R16000 处理器，主频 600MHZ；峰值 384 亿次浮点

运算；I/O 带宽：1.07GB/s（实际），2.4GB/s（峰值）；存储带宽：3.2GB/s（实际），3.2GB/s（峰值）；内存：32GB；硬盘：657GB；图形模块：IP 图形卡，128M 显存，纹理内存可达 104M，48 位 RGBA。

C 曙光 2000-II 超级服务器：峰值运算数为每秒 1117 亿次浮点运算；内存总容量 46GB；硬盘总容量 628GB，其中专用硬盘 405GB，共享硬盘 223GB；处理机总数为 164 个；节点总数为 82 个。

2. 主要研究方向与应用：超级计算中心是为院内外科研单位提供超级计算服务的支撑单位，主要从事并行计算的研究、实现及应用服务，并为大规模复杂技术和商业应用提供可能的解决方案。中心的宗旨是面向科学院乃至社会提供尽可能强的高性能计算能力和技术支持，深腾 6800 正式注册用户数已达 112 个，其中院内用户 109 个，院外用户 3 个。

第二章　基础并行算法

并行算法是适合在并行计算机上实现的算法。一个“好的”并行算法应该能够充分发挥并行计算机多处理机的计算能力。

§2.1　并行计算基本概念

定义 2.1.1 粒度: 是各个处理机可独立并行执行的任务大小的度量。大粒度反映可并行执行的运算量大，亦称为粗粒度。指令级并行等则是小粒度并行，亦称为细粒度。

定义 *2.1.2* 加速比*:* 串行执行时间为 T_s，使用 q 个处理机并行执行的时间为 $T_p(q)$，则加速比为

$$S_p(q) = \frac{T_s}{T_p(q)} \tag{2.1}$$

定义 *2.1.3* 效率*:* 设 q 个处理机的加速比为 $S_p(q)$，则并行算法的效率

$$E_p(q) = \frac{S_p(q)}{q} \tag{2.2}$$

定义 *2.1.4* 性能*:* 求接一个问题的计算量为 W，执行时间为 T，则性能 *(FLOP/s)* 为

$$Perf = \frac{W}{T} \tag{2.3}$$

在 *80* 年代，使用 *FLOP/s* 为单位，*90* 年代，使用 *MFLOP/s* 和 *GFLOP/s*，*21* 世纪普遍使用 *GFLOP/s* 和 *TFLOP/s*，目前也逐渐开始使用 *PFLOP/s*。

引理 *2.1.1* *Amdahl* 定律，对已给定的一个计算问题，假设串行所占的百分比为 α，则使用 q 个处理机的并行加速比为

$$S_p(q) = \frac{1}{\alpha + (1-\alpha)/q} \tag{2.4}$$

Amdahl 定律表明，当 q 增大时，$S_p(q)$ 也增大。但是，它是有上界的。也就是说，无论使用多少处理机，加速的倍数不是能超过 $1/\alpha$。

引理 *2.1.2* *Gustafson* 公式，假设在每个处理机中，串行部分的百分比为 α，则使用 q 个处理机的并行加速比为

$$S_p(q) = \alpha + q \times (1-\alpha) \tag{2.5}$$

§2.2 并行算法设计基本原则

并行算法是并行计算的基础，与实现技术相结合，为高效率使用并行计算机提供解决方案。其基本原则简述如下:

1. 与体系结构相结合;

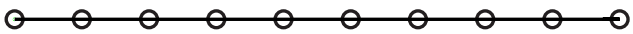


图 2.1: 线性结构

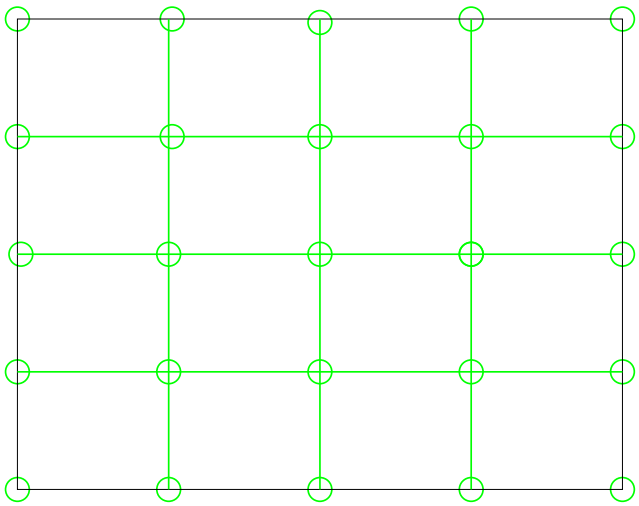


图 2.2: 2 维网格结构

2. 具有可扩展性; 并行算法是否是随处理机个数增加而能够线性或近似线性的加速，这是评价一个并行算法是否有效的重要标志之一。也就是说，如果一个并行算法的加速比是 $S_p(q) = O(q)$ 或者 $S_p(q) = O(q/(1+\log(q)))$，则可以称为具有可扩展性的并行算法。
3. 粗粒度; 通常情况下，粒度越大越好。这是因为在每个处理机中有很多需要计算的工作任务，如此可以充分发挥多处理机的作用。并行加速比对细粒度问题一般情况下是不会很高的，这也是为什么并行计算需要求解大规模问题的原因所在。
4. 减少通信; 一个高效率的并行算法，通信是至关重要的。提高性能的关键是减少通信量和通信次数，其中通信次数通常情况下是决定因素。
5. 优化性能: 一个算法是否有效，不仅依赖于理论分析的结果，也和在实现的过程中采用的技术息息相关。性能主要看单处理机能够发挥计算能力的百分比，然后是并行效率。

影响并行算法效率的因素可能很多，但是这里所给出的几条是主要因素。因此，在算法设计的过程中，如果能够将上述 5 条加以仔细考虑，就能够取得非常好的效果。

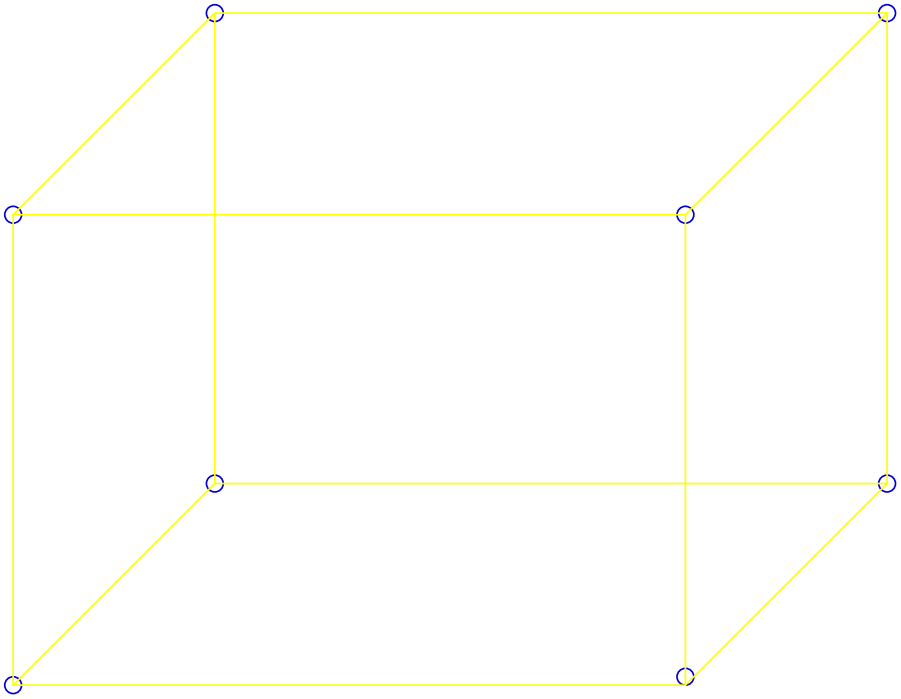


图 2.3: 超立方体结构

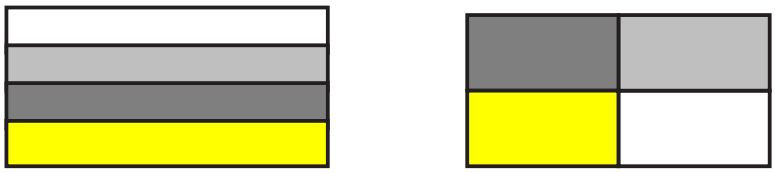
§2.3 区域分解方法

顾名思义，区域分解方法是将区域进行分解的一种方法，早期应用于求解椭圆型偏微分方程的一种方法。假设求解的是矩型区域上的 *Lapalace* 问题，描述如下:

$$\begin{cases} -\Delta u(x,y) = f(x,y) \quad (x,y) \in \Omega = (0,a) \times (0,b) \\ u(x,y)|_{\partial\Omega} = g(x,y) \end{cases} \tag{2.6}$$

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。

图 *2.4* 是非重叠的区域分解，这种分解将离散化后的方程化成一些独立的小规模问题和一个与每个小问题关联的全局问题，此种分解给出的方法是子结构方法。



(a) 一维条分解 (b) 二维块分解

图 2.4: 两种区域分解策略

图 *2.5* 是带重叠的区域分解，记阴影部分的区域为 Ω_1，虚线至右边的区域为 Ω_2。假

设初始解为 u^0，则可将求解方程 *2.6* 转化为如下的两个小问题求解:

$$\begin{cases} -\Delta u(x,y) = f(x,y) \quad (x,y) \in \Omega_1 \\ u(x,y)|_{\partial\Omega_1} = u^0 \end{cases} \tag{2.7}$$

$$\begin{cases} -\Delta u(x,y) = f(x,y) \quad (x,y) \in \Omega_2 \\ u(x,y)|_{\partial\Omega_2} = u^0 \end{cases} \tag{2.8}$$

对于每个子问题的解，需要进行延拓，以便获得全局解的一个近似。假设方程 *2.7* 的解为 u_1^1，方程 *2.8* 的解为 u_2^1，则可由如下的方式延拓它们的解为全局解:

$$\begin{cases} u_1^1 = u_1^1 \in \Omega_1 \\ u_1^1 = u^0 \in \Omega - \Omega_1 \end{cases} \tag{2.9}$$

$$\begin{cases} u_2^1 = u_2^1 \in \Omega_2 \\ u_2^1 = u^0 \in \Omega - \Omega_2 \end{cases} \tag{2.10}$$

则新的近似解为 $u^1 = 1/2(u_1^1 + u_2^1)$。

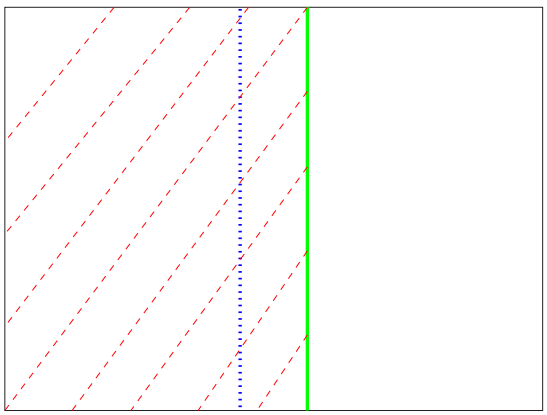


图 2.5: 带重叠的区域分解

§2.4 功能分解方法

功能分解是将不同功能组成的问题，按照其功能进行分解的一种手段，其目的是逐一解决不同功能的问题，从而获得整个问题的解。这里我们使用 *Newton* 法求解非线性方程 *2.11* 为示例，简单说明功能分解的方法。

假设 $F(x)$ 是 $D \subset R^n$ 到 D 的一个映射，要求解 x^*，使得 x^* 是方程 *2.11* 的一个解。

$$F(x) = 0 \tag{2.11}$$

记 $F(x)$ 的 *Jacobi* 矩阵为 $G(x) = F'(x)$，对给定的初始值 $x^{(0)}$，则 *Newton* 迭代法如下:

$$x^{(k+1)} = x^{(k)} - G^{-1}(x^{(k)})F(x^{(k)}), \qquad k = 0, \ldots, \tag{2.12}$$

在这个求解过程中，需要用到函数值和导数值。因此，如果一部分处理机负责计算函数值，另一部分处理机负责计算导数值，就可以得到一种并行计算方法。这时候计算是按照功能进行分配的，也既是所谓的功能分解。

在迭代公式 *2.12* 中，需要求矩阵的逆，通常在计算过程中是将其转化为求解线性代数方程组。即求解:

$$G(x^{(k)})\delta^{(k)} = -F(x^{(k)}) \tag{2.13}$$

其中 $\delta^{(k)} = x^{(k+1)} - x^{(k)}$。

§2.5 流水线技术

流水线技术是并行计算中一个非常有效的、常用的手段，在并行计算中起着非常重要的作用。这里以求解一簇递推问题为例加以说明。

假设要求解的问题是:

$$a_{0,j} \text{ 给定}, \quad a_{i,j} := F(a_{i-1,j}), \quad i = 1, \ldots, n, \quad j = 1, \ldots, m \tag{2.14}$$

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。事实上，当 j 方向上存在某些依赖关系 *(*例如 *5* 点差分格式的 *Gauss–Seidel* 迭代*)* 时仍然是有效的。当数据在处理器中的划分仅沿 j 方向进行时，*(2.14)* 中的计算是完全并行的。在实际问题的计算中，如二维或三维 *PDE* 的数值解，由于数据划分一般在所有空间方向上同时进行，并且计算通常也在各个空间方向上交替进行，因此，需要考虑当沿 i 方向进行数据划分时 *(2.14)* 的并行算法。不失一般性，我们假设数据划分仅沿 i 方向进行，即假设在有 q 个处理器的分布式系统上，$a_{i,j}$ 被分成 q 段，$\{a_{i,j},\ i = n_k, \ldots, n_{k+1} - 1\}$ 存储在处理器 P_k 上，$k = 0, \ldots, q-1$, 这里 $1 = n_0 < n_1 < \cdots < n_q = n + 1$。

表 *2.1* 给出了流水线方法当 $q = 3$ 时的计算流程*:*

这个计算过程也可以用图 *2.6* 表示，在这个图中，相同颜色的部分是可以并行计算的。

	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 $a_{i,m} := F(a_{i-1,m})$, $i = n_2, \ldots, n$

表 2.1: 递推关系式的流水线计算流程

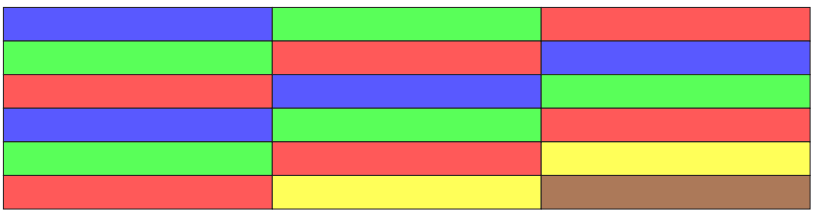


图 2.6: 流水线计算示意图

§2.6 分而治之方法

这里我们以一个简单的求和问题为例，说明什么是分而治之方法。假设在 $q = 2^3$ 个处理机上计算:

$$s = \sum_{i=0}^{n-1} a_i \tag{2.15}$$

假设在 *(2.15)* 中的 $n = 2^m \gg q$，记 $s_{000}(0:n)$ 为 s。则可以将计算 $s_{000}(0:n)$ 分解为两个小的求和问题，即:

$$\begin{cases} s_{000}(0:n) = s_{000}(0:n/2) + s_{100}(n/2:n/2) \\ s_{000}(0:n/2) = s_{000}(0:n/4) + s_{010}(n/4:n/4) \\ s_{100}(n/2:n/2) = s_{100}(n/2:n/4) + s_{110}(3n/4:n/4) \\ s_{000}(0:n/4) = s_{000}(0:n/8) + s_{001}(n/8:n/8) \\ s_{010}(n/4:n/4) = s_{010}(n/4:n/8) + s_{011}(3n/8:n/8) \\ s_{100}(n/2:n/4) = s_{100}(n/2:n/8) + s_{101}(5n/8:n/8) \\ s_{110}(3n/4:n/4) = s_{110}(3n/4:n/8) + s_{111}(7n/8:n/8) \end{cases} \tag{2.16}$$

其中 $s_{xxx}(k:l) = \sum_{i=k}^{k+l-1} a_i$ 是在处理机 P_{xxx} 中完成的计算，xxx 是二进制数。这里的计算可以用图 *2.7* 简单的描述，在图中只简单的给出处理机号。在图 *2.7* 中，从上至下

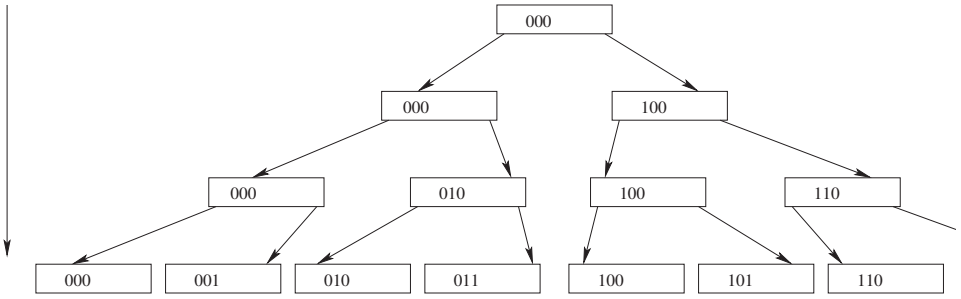


图 2.7: 分而治之计算求和示意图

是分解的过程，从下至上是求部分和的过程。这就是有分有治的一个简单过程，也既是一种分而治之方法。分而治之方法在并行计算中起着举足轻重的作用，将在今后的算法设计中经常遇到。

§2.7 同步并行算法

在这一节中，以计算线性迭代问题 (给定 $x^{(0)}$) *(2.17)* 为例，对同步并行算法加以说

明。

$$x^{(k+1)} = b + Ax^{(k)} \tag{2.17}$$

这里 A 是 $n \times n$ 矩阵，b、$x^{(k)}$ 是 n 维向量。假设矩阵在处理机是按照行分块存储的，图 *2.8* 是以 *4* 个处理机为例的划分。在每个处理机中计算出 $x^{(k+1)}$ 的一部分，下一次计

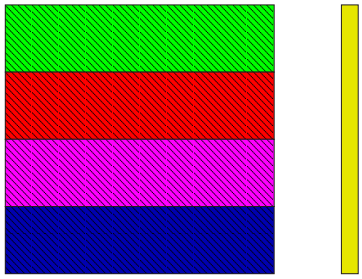


图 2.8: 矩阵向量乘划分图

算的时候需要将每个处理机计算出的部分 $x^{(k+1)}$ 送到其它的处理机中，进行新一步的计算。在这个问题中，不难看出，为了计算下一步的新 $x^{(k+1)}$，必须等到所有处理机都计算完部分 $x^{(k+1)}$ 之后才能进行，这是一个典型的需要同步的算法。许多并行算法是这一类的算法，尤其是从串行算法并行化得到的大部分算法属于这一类。象前述的流水线算法、分而治之算法等，它们在某种意义上也是同步并行算法。

§2.8 异步并行算法

异步并行算法是与同步并行算法相对的，同步算法在某一时刻需要与其它的处理机进行数据交换，然后才能继续执行。异步并行算法进行数据交换不需要严格确定在某一时刻，每个处理机按照预定的计算任务持续执行，但通常需要在一定的时候必须进行一次数据交换，以保证算法的正确性。关于这个算法的实际例子将在矩阵并行计算中进行介绍。

§2.9 作业

1. 从加速比的定义出发，证明 *Amdahl* 定律和 *Gustafson* 公式。
2. 假设 $m \times n$ 矩阵 A 是按列分块存储在每个处理机中的，x 存储在处理机 P_0 中，假设使用 *4* 个处理机，即 $A = [A_0, A_1, A_2, A_3]$，其中 A_i 存放在处理机 P_i 中。试给出并行计算 $y = Ax$ 的方法。

第二部分

并行算法设计与实现

第三章　矩阵并行计算

在科学与工程计算的许多问题中, 矩阵运算, 特别是矩阵相乘、求解线性方程组和矩阵特征值问题是最基本的内核. 随着*MPP* 并行计算机和网络并行环境的不断发展, 为了充分发挥分布式并行计算机的功效, 有必要对计算方法进行深入的研究. 这里着重考虑矩阵乘积和求解线性方程组的多种并行算法, 其向量化算法 [6, 7] 本书将不作介绍. 代数特征值问题及其相关问题的计算方法在文献 *[8]* 中有较详细的讨论, 这里将不作介绍. 为了讨论方便起见, 先作一些约定, 假设有 p 个处理机, P_j 表示第 j 个处理机, P_{myid} 表示当前运行程序的处理机, *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, 本章给出的算法都是在 P_{myid} 处理机上的. 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, 计算结果的好坏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. 我们结合算法研究给出在不同并行机上的一些重要例子来表明程序设计思想, 并采用程序方式描述算法, 以利于机器实现.

关于通信模式, 数据传输方式有许多不同的假设, 这样的假设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, 通常考虑其为: $T_c = \alpha + \beta \times N$, 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, 重要的问题是矩阵在处理机中的存放方式, 通常考虑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, 矩阵存放方式如下:

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \\ A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} \tag{3.1}$$

$$\left(\begin{array}{cccc|cccc} A_{00} & A_{02} & A_{04} & A_{06} & A_{01} & A_{03} & A_{05} & A_{07} \\ A_{30} & A_{32} & A_{34} & A_{36} & A_{31} & A_{33} & A_{35} & A_{37} \\ A_{60} & A_{62} & A_{64} & A_{66} & A_{61} & A_{63} & A_{65} & A_{67} \\ \hline A_{10} & A_{12} & A_{14} & A_{16} & A_{11} & A_{13} & A_{15} & A_{17} \\ A_{40} & A_{42} & A_{44} & A_{46} & A_{41} & A_{43} & A_{45} & A_{47} \\ A_{70} & A_{72} & A_{74} & A_{76} & A_{71} & A_{73} & A_{75} & A_{77} \\ \hline A_{20} & A_{22} & A_{24} & A_{26} & A_{21} & A_{23} & A_{25} & A_{27} \\ A_{50} & A_{52} & A_{54} & A_{56} & A_{51} & A_{53} & A_{55} & A_{57} \end{array}\right) \tag{3.2}$$

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 $A = U\Sigma V^T$

§3.1 并行矩阵乘法

矩阵乘积在实际应用中是经常要用到的, 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. 本节要考虑的计算问题是

$$C = A \times B \tag{3.3}$$

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 $n = \bar{n} \times p$, 下面考虑基于对矩阵 A 和 B 的不同划分的并行计算方法.

§3.1.1 串行矩阵乘法

例 3.1.1 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 3.1.2 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
```

§3.1.2 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:

$$A = \begin{bmatrix} A_0^T & A_1^T & \dots & A_{p-1}^T \end{bmatrix}^T, \quad B = \begin{bmatrix} B_0 & B_1 & \dots & B_{p-1} \end{bmatrix} \tag{3.4}$$

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 $C_{i,j}$, $j = 0, \dots, p-1$ 存放在 P_i 中, 这种存放方式使数据在处理机中不重复. 在本节所考虑的算法中, 都使用这种数据的存放方式. 由于使用 p 个处理机, 每次每台处理机计算出一个 $C_{i,j}$, 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 3.1.1

for $i = 0$ *to* $p-1$ *do*
  $l \equiv i$+*myid mod* p
  $C_l = A \times B$, *mp1* $\equiv$ *myid*+1 *mod* p, *mm1* $\equiv$ *myid*-1 *mod* p
  if $i \neq p-1$, *send*(B, *mm1*), *recv*(B, *mp1*)
end{*for*}

在这个算法中, $C_l = C_{myid,l}$, $A = A_{myid}$, 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 表示在算法 *3.1.1* 中数据的交换量, CA_1 表示在算法 *3.1.1* 中的计算量, 则有 $DTA_1 = 2 \times k \times (n - \bar{n})$, $CA_1 = m \times k \times n/p$.

§3.1.3 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 *(3.4)*, B 的划分如下:

$$B = \begin{bmatrix} B_0^T & B_1^T & \dots & B_{p-1}^T \end{bmatrix}^T \tag{3.5}$$

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

$$A_i = \begin{bmatrix} A_{i0} & A_{i1} & \dots & A_{i,p-1} \end{bmatrix}$$

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 *3.1.2* 相同, 在结点 P_{myid} 上的计算过程可归纳为算法 *3.1.2*.

算法 *3.1.2*

for $i = 0$ *to* $p-1$ *do*
  $l \equiv i$+*myid mod* p
  $C = C + A_l \times B$, *mp1* $\equiv$ *myid+1 mod* p, *mm1* $\equiv$ *myid-1 mod* p
  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 *3.1.1* 相同, 所不同的只是计算 C 的方式, 其中 $A_l = A_{myid,l}$.

§3.1.4 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 *(3.4)* 相同, A 划分为如下形式:

$$A = \begin{bmatrix} A_0 & A_1 & \ldots & A_{p-1} \end{bmatrix} \tag{3.6}$$

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 $B_i = \begin{bmatrix} B_{i0}^T & B_{i1}^T & \ldots & B_{i,p-1}^T \end{bmatrix}^T$, 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 *3.1.3*

for $i = 0$ *to* $p-1$ *do*
  $l \equiv i$+*myid mod* p
  $C = C + A \times B_l$, *mp1* $\equiv$ *myid+1 mod* p, *mm1* $\equiv$ *myid-1 mod* p
  if $i \neq p-1$, *send(A, mm1), recv(A, mp1)*
end$\{for\}$

算法 *3.1.3* 的计算量与算法 *3.1.1* 和算法 *3.1.2* 是相同的, 算法 *3.1.3* 的数据交换量是 $DTA_3 = 2 \times m \times (k - \bar{k})$. 当 $m \neq n$ 时, $DTA_1 \neq DTA_3$. 两种算法数据交换量的大小

是由 m 和 n 决定的, 即当 $m \leq n$ 时, $DTA_3 \leq DTA_1$. 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.

§3.1.5 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 *(3.6)* 相同, B 的划分与式 *(3.5)* 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 $B_i = \begin{bmatrix} B_{i0} & B_{i1} & \dots & B_{i,p-1} \end{bmatrix}$ 则有

$$C_j = \sum_{i=0}^{p-1} A_i \times B_{ij}$$

因此, 我们就可以给出如下的算法:

算法 *3.1.4*

$C = A \times B_{myid}$
for $i = 1$ *to* $p-1$ *do*
  $l \equiv i + myid \ mod \ p$, $k \equiv myid - imodp$
  $T = A \times B_l$
  send(T, l), recv(T, k)
  $C = C + T$
end$\{for\}$

这里给出的算法是简洁易懂, 其通信量 $DTA_4 = 2 \times m \times (n - \bar{n})$. 如果采用按行分块方式计算 C, 算法 *3.1.4* 也同样适合, 且通讯量也是不变的, 因此选择何种方式计算 C 可根据需要而定.

§3.1.6 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

$$Q_{ij} = \begin{cases} I_n, & j \equiv i + 1 \ mod \ m \\ 0_n, & \text{其它情况} \end{cases}$$

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 $D_A^{(l)} = \mathrm{diag}(D_i^{(l)}) = \mathrm{diag}(A_{i,i+l \bmod m})$, 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 $P_{myid} = P_{myrow,mycol}$, 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 *3.1.5*

$C = 0$
for $i = 0$ *to* $m - 1$ *do*
  $k \equiv$ *myrow+i mod m; mp1* $\equiv$ *mycol+1 mod m; mm1* $\equiv$ *mycol-1 mod m;*
  if mycol=k then
    send(A, (myrow, mp1)); copy(A, tmpA);
  else
    recv(tmpA, (myrow, mm1));
    if $k \neq$ *mp1, send(tmpA, (myrow, mp1));*
  end$\{if\}$
  $C = C$+*tmpA* $\times B$; *mp1* $\equiv$ *myrow+1 mod m; mm1* $\equiv$ *myrow-1 mod m;*
  if $i \neq m - 1$ *then*
    send(B, (mm1, mycol)); recv(B, (mp1, 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 $DTA_5 = m \times 2 \times n^2 + (m-1) \times 2 \times n^2 = 2(2m-1)n^2 = 4m^2n^2/\sqrt{p} - 2m^2n^2/p$. 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 $p \geq 4$ 时, *Cannon* 算法具有优越性.

§3.2 线性代数方程组并行求解方法

线性方程组是许多重要问题的核心, 因此有效地求解线性方程组在科学与工程计算中是非常重要的. 并行计算机的问世, 使求解问题的速度和解题规模大幅度地提高, 同

时也使计算方法产生了变化. 在传统的串行机上, *Linpack* 是求解线性方程组的有效软件包. 然而在并行机上求解此问题, 就需要设计出适合于该机的并行算法, 算法的优劣会对并行机的效率产生很大的影响. 这里考虑的问题是

$$Ax = b$$

这里的任务可以分为两方面, 一方面是并行计算矩阵 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 $AQ = LU$. 另一方面是并行求解三角形方程组, 即, 求解方程组 $Ly = b$ 和 $Ux = y$. 下面我们描述有关的算法.

§3.2.1 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, 进行列消去, 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 *3.2.1*

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 *3.2.1* 中, 主要计算工作量是修正矩阵 A, 即做 $a_{ik} - a_{ij} \times a_{jk}$, 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. 在多处理机上 LU 分解的重要工作是使载荷尽可能的平衡, 我们采用卷帘 *(wrap)* 存储方式在各处理机上分配矩阵 A, 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 $i \times p + myid$ 列, 矩阵在处理机中的存放方式为:

$$\left[\begin{array}{ccc|ccc|cc|c} A_{00} & A_{0p} & \cdots & A_{01} & A_{0,p+1} & \cdots & A_{02} & A_{0,p+2} & \cdots & \cdots \\ A_{10} & A_{1p} & \cdots & A_{11} & A_{1,p+1} & \cdots & A_{12} & A_{1,p+2} & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{n-1,0} & A_{n-1,p} & \cdots & A_{n-1,1} & A_{n-1,p+1} & \cdots & A_{n-1,2} & A_{n-1,p+2} & \cdots & \cdots \end{array}\right] \tag{3.7}$$

下面给出在 P_{myid} 上的结点描述.

算法 *3.2.2*

icol= 0
for $j = 0$ *to* $n-2$ *do*
  if myid=j *mod p then*
    find l: $|a_{l,icol}| = \max\{|a_{i,icol}|, i = j, \ldots, n-1\}$
    if $l \neq j$, *swap* $a_{j,icol}$ *and* $a_{l,icol}$
    if $a_{j,icol} = 0$, A *is singular and kill all processes*
    $a_{i,icol} = a_{i,icol}/a_{j,icol}$, $i = j+1, \ldots, n-1$
    $f_{i-j-1} = a_{i,icol}$, $i = j+1, \ldots, n-1$
    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 p, send(l, myid+1) and send(f, myid+1)*
  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 *3.2.2* 是在分布式并行计算机上做 LU 分解的有效方法之一, 我们在国产并行机上做了许多实验, 效果很好. 如果采用分块卷帘方式存储, 增大了算法的粒度, 效果更好.

§3.2.2 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, 我们不妨仅讨论解下三角方程组 $Lx = b$. 三角方程组的并行求解对有效的并行求解线性方程组是不可缺少的, 它的并行效果的好坏对求解整个问题有直接的影响, 这里给出两种并行实现方法. 首先给出一个串行算法.

算法 *3.2.3*

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. 正如我们所描述的, 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它处理机中, 其通信次数是很多的, 这对于消息传递型并行系统是不太适合的. 但是对于有共享存储的系统是可以采用这种计算方案的. 像前一节中做 LU 分解时一样, 需要引进共享变量 $flag$ 和 gb, 这时在共享存储系统上的算法可描述成如下形式:

算法 *3.2.4*

$k = 0$
$for\ i = 0\ to\ n-1\ do$
  $if\ myid \equiv i\ mod\ p\ then$
    $gb_i = b_k/l_{ki}$, $flag_i = 1$, $k+1 \to k$
  $end\{if\}$
  $if\ flag_i \neq 1$, $wait$
  $for\ j = k\ to\ m-1\ do$
    $b_j = b_j - l_{ji} \times gb_i$
  $end\{for\}$
$end\{for\}$

这是一种非常自然的并行实现方法, 但是为减少等待, 可以采用修正 b 的一个分量就计算下一个解的方式, 这样其它的处理机在做下一次修正时就已经有了要用的新值, 而不需要等待. 这些都是算法与程序设计时应该注意的技巧, 反复试验就会有所体会. 在大部分现有的共享存储并行系统上, 都提供了同步机制, 在算法中条件判断的等待也可以由系统提供的库函数来实现, 即在此同步. 但对采用异步方式计算下一个新解的算法就不适合了, 这时就应用我们在算法中给出的条件判断的等待来实现. 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, 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. 如图是使用 *3* 个处理机求解下三角线性代数方程组的并行计算过程:

$$
\begin{array}{l}
*\\
*\ *\\
*\ *\ *\\
\diamond\ *\ *\ *\\
\diamond\ \diamond\ *\ *\ *\\
\diamond\ \diamond\ \diamond\ *\ *\ *\\
\diamond\ \diamond\ \diamond\ \diamond\ *\ *\ *\\
\diamond\ \diamond\ \diamond\ \diamond\ \diamond\ *\ *\ *\\
\diamond\ \diamond\ \diamond\ \diamond\ \diamond\ \diamond\ *\ *\ *\\
\diamond\ \diamond\ \diamond\ \diamond\ \diamond\ \diamond\ \diamond\ *\ *\ *\\
\diamond\ \diamond\ \diamond\ \diamond\ \diamond\ \diamond\ \diamond\ \diamond\ *\ *\ *
\end{array}
\tag{3.8}
$$

其算法的具体描述形式如下:

算法 ***3.2.5***

$k = 0$

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 $i = 0$ *to* $n-1$ *do*

if $i > 0$, *recv(v,* $i-1$ *mod p)*
$x_k = (u_i + v_0)/l_{ik}$
$v_j = v_{j+1} + u_{i+1+j} - l_{i+1+j,k} \times x_k$, $j = 0, \ldots, p-3$
$v_{p-2} = u_{i+p-1} - l_{i+p-1,k} \times x_k$
send(v, $i+1$ *mod p)*
$u_j = u_j - l_{jk} \times x_k$, $j = i+p, \ldots, n-1$
$k+1 \to k$
end$\{for\}$

关于这个算法的详细讨论可参见文献 *[10].*

§3.3 对称正定线性方程组的并行解法

对于对称正定矩阵 A 的 LU 分解, 我们采用*Cholesky* 分解, 也即 $A = R^T R$, 其中 R 是上三角矩阵. 由于三角方程组的求解在前一节中已经给出, 因此这里着重考虑对称正定矩阵的*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, 提出的并行算法. 另一个方法是用双曲旋转变换的方式来做 *Cholesky* 分解. 下面就分别介绍这两种算法.

§3.3.1 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 [11]. 它是在分布式多处理机系统上求 *Cholesky* 分解的有效算法. 首先从串行算法出发

算法 ***3.3.1***

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 存放在与矩阵 A 相对应的下三角位置, 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 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 *2*

$for\ i=0\ to\ m-1\ do$
$k=i\times p+myid,\ l=k-p+1$
$if\ k>0\ then,\ recieve\ G\ from\ P_{myid-1}$
$for\ j=0\ to\ p-2\ do$
$a_{i,j+l}=(a_{i,j+l}-\sum_{t=0}^{j+l-1}a_{it}\times g_{jt})/g_{j,j+l}$
$F_j=G_{j+1}$
$end\{for\}$
$a_{ik}=a_{ik}-\sum_{t=0}^{k-1}a_{it}\times a_{it}$
$a_{ik}=\sqrt{a_{ik}},\ F_{p-2}=A_i$
$Send\ F\ to\ P_{myid+1}$
$for\ e=i+1\ to\ m-1\ do$
$for\ j=0\ to\ p-2\ do$
$a_{e,j+l}=(a_{e,j+l}-\sum_{t=0}^{j+l-1}a_{et}\times g_{jt})/g_{j,j+l}$
$a_{ek}=(a_{ek}-\sum_{t=0}^{k-1}a_{et}\times a_{it})/a_{ik}$
$end\{for\}$
$end\{for\}$

这个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. 理论分析与数值计算结果可参见文献 *[11]*.

§3.3.2 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:

$$H=\begin{bmatrix} cosh\phi & sinh\phi \\ sinh\phi & cosh\phi \end{bmatrix}$$

在我们的算法中, 使用下面的形式:

$$H=(1-\rho^2)^{-\frac{1}{2}}\begin{bmatrix} 1 & -\rho \\ -\rho & 1 \end{bmatrix}$$

其中 $\rho = tanh(-\phi)$. 通过这个变换把矩阵 A 化成 R^TR, 首先我们从它的基本理论开始.

假设 $A = D + U^T + U$, 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

$$A = V^TV - W^TW$$

从而有下面的关系式成立:

$$\begin{bmatrix} R^T & 0 \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} V^T & W^T \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} V \\ W \end{bmatrix}$$

其中 I 是 $n \times n$ 单位矩阵, 为做双曲变换 *Cholesky* 分解, 要用到下面的一些定义与引理.

定义 *3.3.1* 如果一个 $2m \times 2m$ 矩阵 Θ 满足下面的关系式:

$$\Theta^T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \Theta = \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix}$$

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 $A = R^TR$, 因此主要任务就是寻找伪正交矩阵 Q. 下面不加证明地列出文献 *[12]* 中的引理.

引理 *3.3.1* 如果 R 和 S 都是 $n \times n$ 上三角矩阵, 使得 $R^TR - S^TS$ 对称正定, 则 R 是可逆的, 并满足:

$$| s_{kk}r_{kk}^{-1} | < 1, \qquad 1 \le k \le n$$

引理 *3.3.2* 如果 R 和 S 都是 $n \times n$ 上三角矩阵, 使得 $R^TR - S^TS$ 对称正定, 并令 $\rho_k \equiv s_{kk}r_{kk}^{-1}, \quad 1 \le k \le n$, $\hat{Q} = \widetilde{Q}^{(n)}\widetilde{Q}^{(n-1)} \cdots \widetilde{Q}^{(1)}$, 其中 $\widetilde{Q}^{(k)}$ 的元素定义如下:

$$\tilde{q}_{ij}^{(k)} = \begin{cases} 1, & i = j \ne k \text{ 或 } i = j \ne n+k \\ (1-\rho_k^2)^{-\frac{1}{2}}, & i = j = k \text{ 或 } i = j = n+k \\ -(1-\rho_k^2)^{-\frac{1}{2}}\rho_k, & (i,j) = (k, n+k) \text{ 或 } (i,j) = (n+k, k) \\ 0, & \text{其它} \end{cases}$$

此外, 如果 $\begin{bmatrix} \widetilde{R} \\ \widetilde{S} \end{bmatrix} = \hat{Q} \begin{bmatrix} R \\ S \end{bmatrix}$, 则 $\hat{Q}$ 是伪正交矩阵, 并且 $\widetilde{R}$ 是上三角矩阵, $\widetilde{S}$ 是严格上三角矩阵.

从 $\widetilde{Q}^{(k)}$ 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 $\widetilde{Q}^{(k)} \begin{bmatrix} R \\ S \end{bmatrix}$ 的第 k 行和第 $k+n$ 行是相同的. 由此我们说 $\widetilde{Q}^{(k)}$ 是一个旋转变换.

假设 Q 是 $n \times n$ 循环置换矩阵, 其中 $p_{1,n} = 1$ 和 $p_{i,i-1} = 1, \quad 2 \le i \le n$. 根据引理 *2*, 双曲变换 *Cholesky* 分解的算法可描述成如下:

算法 *3.3.2*

$V^0 = V$, $W^0 = W$, $A = V^T V - W^T W$

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 *3.3.2* 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 *3.3.2* 中的循环变量 i 只需到 β 就可计算出 A 的 *Cholesky* 分解, 从而减少了计算时间, 详细的讨论请参见文献 *[12]*. 由于计算每个 H_k 是相互独立的, 因此易于并行计算, 这里就不再详述了.

§3.3.3 修正的双曲变换 Cholesky 分解

在算法 *3* 中 $\hat{Q}^{(i)}$ 是 $\widetilde{Q}^{(k)}$ 的乘积, 而每个 $\widetilde{Q}^{(k)}$ 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. 我们的目的是在计算 $\overline{M} = HM$ 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, 为此可以选择 $\rho = m_{21} m_{11}^{-1}$. 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 $G = \bar{K}^{-1} H K$, 其中 $\bar{K}$ 是 2×2 对角矩阵. 如果 $\bar{B} = GB$, 则 $\bar{M} = \bar{K}\bar{B}$. 这里通过适当选取 $\bar{K}$, 以达到减少运算次数和开方运算的目的. 在这里我们并不直接计算 $\bar{K}$, 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 3.3.3 假设 $\alpha = \frac{L_2}{L_1}$, $\beta = \frac{b_{21}}{b_{11}}$. 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 的定义我们知道 $\rho = m_{21}m_{11}^{-1} = \frac{K_2 b_{21}}{K_1 b_{11}}$. 因此就有

$$HK = (1-\rho^2)^{-\frac{1}{2}} \begin{bmatrix} K_1 & -\frac{K_2^2 b_{21}}{K_1 b_{11}} \\ -\frac{K_2 b_{21}}{b_{11}} & K_2 \end{bmatrix} = (1-\alpha\beta^2)^{-\frac{1}{2}} K \begin{bmatrix} 1 & -\alpha\beta \\ -\beta & 1 \end{bmatrix}$$

引理得证.

引理 3.3.4 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 $R^T ER - S^T FS$ 是正定的, 并假设 $\alpha_k = \frac{F_k}{E_k}$, $\beta_k = \frac{s_{kk}}{r_{kk}}$, 如果

$$\begin{bmatrix} \widetilde{R} \\ \widetilde{S} \end{bmatrix} = \hat{Q} \begin{bmatrix} R \\ S \end{bmatrix}$$

其中 $\hat{Q} = \widetilde{Q}^{(n)}\widetilde{Q}^{(n-1)} \cdots \widetilde{Q}^{(1)}$,$\widetilde{Q}^{(k)}$, E 和 F 的元素是如下定义的:

$$\tilde{q}_{ij}^{(k)} = \begin{cases} 1, & i = j \\ -\alpha_k\beta_k, & i = k, j = n+k \\ -\beta_k, & i = n+k, j = k \\ 0, & \text{其它} \end{cases}$$

和

$$\widetilde{E}_k = \frac{E_k}{1-\alpha_k\beta_k^2}, \qquad \widetilde{F}_k = \frac{F_k}{1-\alpha_k\beta_k^2}, \qquad 1 \le k \le n$$

则 $\widetilde{R}^T\widetilde{E}\widetilde{R} - \widetilde{S}^T\widetilde{F}\widetilde{S} = R^T ER - S^T FS$, 并且 $\widetilde{R}$ 是上三角矩阵, $\widetilde{S}$ 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 $A = V^T EV - W^T FW$, 则修正的双曲 *Cholesky* 分解算法可被描述成如下形式:

算法 3.3.3

$V^0 = V$, $W^0 = W$, $E^0 = E$, $F^0 = F$

for $i = 0$ *to* $n-1$ *do*

$$\begin{bmatrix} V^{i+1} \\ W^{i+1} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix} \hat{Q}^{(i)} \begin{bmatrix} V^i \\ W^i \end{bmatrix}$$

$$E_k^{i+1} = \frac{E_k^i}{1-\alpha_k^i {\beta_k^i}^2}, \qquad \widetilde{F}_k^{i+1} = \frac{F_k^i}{1-\alpha_k^i {\beta_k^i}^2}, \qquad 1 \le k \le n$$

$$F_1^{i+1} = \widetilde{F}_n^{i+1}, \quad F_{k+1}^{i+1} = \widetilde{F}_k^{i+1}, \qquad 1 \le k \le n-1$$

end { *for* }

这里 $\hat{Q}^{(i)}$ 的定义与引理 *3.3.4* 中 $\hat{Q}$ 的定义相同. 这个算法与算法 *3.3.2* 一样都是易于并行实现的.

§3.4 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起着非常重要的作用, 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 *[13]–[14]*. 这里着重基于区域分解的分裂方法, 以便为进一步的并行计算打下基础. 在这里要求解的问题是 $Ax = d$, 其中

$$A = T(c,a,b) = \begin{bmatrix} a_0 & b_0 & & & \\ c_1 & a_1 & b_1 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-2} & a_{n-2} & b_{n-2} \\ & & & c_{n-1} & a_{n-1} \end{bmatrix}$$

对于一般性的三对角线性代数方程组，并行求解的方法也已经有许多工作。这里重点考虑对称正定三对角线性方程组的解法, 它基于对矩阵的分块, 故此称之为分裂法 [*16, 14*]. 其分解形式为如下的块三对角矩阵:

$$A = \begin{bmatrix} A_0 & B_0 & & & \\ B_0^T & A_1 & B_1 & & \\ & \ddots & \ddots & \ddots & \\ & & B_{p-3}^T & A_{p-2} & B_{p-2} \\ & & & B_{p-2}^T & A_{p-1} \end{bmatrix}$$

其中 B_i 只有左下角的一个元素不为零,A_i 是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 是单位下三角阵. 假设 $A_i = L_i D_i L_i^T$, 令 $L = \mathrm{diag}(L_i)$, 则有

$$L^{-1}AL^{-T} = \begin{bmatrix} D_0 & \bar{B}_0 & & & \\ \bar{B}_0^T & D_1 & \bar{B}_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \bar{B}_{p-3}^T & D_{p-2} & \bar{B}_{p-2} \\ & & & \bar{B}_{p-2}^T & D_{p-1} \end{bmatrix}$$

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 $\widetilde{B}_i^T$ 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来求解, 把这个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^TR 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. 综上所述, 我们可给出下述算法:

算法 *3.4.1*

(1) 计算 L_i, D_i, 使 $A_i = L_iD_iL_i^T$;

(2) 对矩阵做变换, 并对 B_i 做前述的消去;

(3) 形成小的三对角线性方程组, 并求解之;

(4) 求解整个问题.

这个算法具有很好的并行性, 是目前被认为求解这类问题最有效的算法. 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.

§3.5 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。前面我们介绍了一些直接求解线性代数方程组的方法，这里将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。

§3.5.1 Jacobi 迭代法

考虑求解线性代数方程组

$$Ax = b \tag{3.9}$$

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 *3.9* 可以变为

$$Dx = b + (L + U)x \tag{3.10}$$

如果方程组 *3.10* 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:

$$x^{(k+1)} = D^{-1}(L+U)x^{(k)} + D^{-1}b \tag{3.11}$$

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:

$$x^{+} = Gx + g \tag{3.12}$$

公式 *3.12* 的计算和前面在介绍同步并行计算方法的时候是类似的，因此并行计算方法是非常容易得到的，这里就不再赘述。

§3.5.2 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 *3.9* 的分量表示

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \qquad i = 1, \ldots, n \tag{3.13}$$

对于给定的初值 $x^{(0)}$，*Gauss-Seidel* 迭代法如下:

算法 *3.5.1* *(Gauss-Seidel 迭代算法)*

- $k = 0$
- $x_1^{(k+1)} = (b_1 - \sum_{j=2}^{n} a_{1j}x_j^{(k)})/a_{11}$
- $x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k+1)} - \sum_{j=3}^{n} a_{2j}x_j^{(k)})/a_{22}$
- $\ldots$
- $x_{n-1}^{(k+1)} = (b_{n-1} - \sum_{j=1}^{n-2} a_{n-1,j}x_j^{(k+1)} - a_{n-1,n}x_n^{(k)})/a_{n-1,n-1}$
- $x_n^{(k+1)} = (b_n - \sum_{j=1}^{n-1} a_{nj}x_j^{(k+1)})/a_{nn}$
- $\|x^{(k+1)} - x^{(k)}\|_2 < \epsilon\|x^{(k+1)} - x^{(0)}\|_2$? $k = k+1$

从算法 *3.5.1* 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 *3.5.2* *(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 *3.5.2* 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。

§3.6 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 *[17]* 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 *[18]* 给出了非线性迭代 $x=Fx$ 的收敛定理，当 F 是 P- 收缩映射时，此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 *[19]*.

§3.6.1 异步并行迭代法基础

首先引入关于 $x=Fx$ 的异步迭代算法的定义，最后给出文献 *[18]* 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，$A\geq 0$ 表示 A 的元素均大于等于 0. Fx 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 *3.6.1* 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) $s_i^{(j)}\leq j-1$，$j=1,2,\ldots$；

(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 *[18]* 中的一个重要结果，故此我们以引理的形式给出.

引理 *3.6.1* 假设 $|Fx - Fy| \leq A|x - y|$, 其中 A 是非负矩阵并且 $\rho(A) < 1$. 则迭代 $x = Fx$ 是异步收敛的.

§3.6.2 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:

$$x = B^{-1}Cx + B^{-1}b \tag{3.14}$$

其中 B 是可逆的. 由引理 *3.6.1* 可知, 当 $\rho(|B^{-1}C|) < 1$ 时, 上述的迭代过程是异步迭代收敛的. 下面我们就对 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 *[19]*. 下面不加证明地给出文献 *[19]* 中的一些结论. 首先给出弱正则分裂的定义.

定义 *3.6.2* 设 A, B 和 C 是实矩阵, 如果 $A = B - C$, $B^{-1} \geq 0$ 和 $B^{-1}C \geq 0$, 则称 $A = B - C$ 是 A 的弱正则分裂.

引理 *3.6.2* 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 $A^{-1} \geq 0$.

引理 *3.6.3* 设 B 和 C 是 $n \times n$ 阶矩阵, 如果 $|C| \leq B$, 则 $\rho(C) \leq \rho(B)$.

引理 *3.6.4* 设 A 是严格或不可约对角占优矩阵, 如果 $a_{ij} \leq 0$ $(i \neq j)$, 且 $a_{ii} > 0$, 则 A 是 M 矩阵.

下面我们不加证明地引用文献 *[20]* 中的两个结论.

定理 *3.6.1* 设 $A = B - C$ 是 A 的弱正则分裂且 A 是 M 矩阵, 则迭代形式 *(3.14)* 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 *3.6.2* 设 A 是严格或不可约对角占优矩阵,$A = B - C$, 其中 $B = D - \alpha L$, $C = (1 - \alpha)L + U$, $0 \leq \alpha \leq 1$, 则迭代形式 *(3.14)* 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有很大的作用, 因为在采用区域分解法的时候, 信息的交换可以不需要等待, 所以在某些情况下, 可以提高并行处理机的效率.

§3.7 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是非常重要的，在这一节中，我们将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 $A \in R^n$。则标准特征值问题是:

$$Ax = \lambda x \tag{3.15}$$

称满足特征方程 *3.15* 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。

§3.7.1 对称三对角矩阵特征值问题

在特征值问题 *3.15* 中，考虑 A 是对称三对角矩阵的特征值问题，即:

$$\begin{pmatrix} a_1 & b_1 & & & & \\ b_1 & a_2 & b_2 & & & \\ & b_2 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{n-2} & a_{n-1} & b_{n-1} \\ & & & & b_{n-1} & a_n \end{pmatrix} x = \lambda x \tag{3.16}$$

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 *3.16* 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 *3.7.1* 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:

$$\begin{cases} P_0(\lambda) = 1 \\ P_1(\lambda) = a_1 - \lambda \\ P_n(\lambda) = (a_n - \lambda)P_{n-1}(\lambda) - b_{n-1}^2 P_{n-2}(\lambda) \quad n = 2, \ldots \end{cases} \tag{3.17}$$

从数值计算的稳定性及方便考虑，令 $Q_n(\lambda) = P_n(\lambda)/P_{n-1}(\lambda)$，则有:

$$\begin{cases} Q_1(\lambda) = a_1 - \lambda \\ Q_n(\lambda) = (a_n - \lambda) - b_{n-1}^2/Q_{n-1}(\lambda) \quad n = 2, \ldots \end{cases} \tag{3.18}$$

矩阵 A 的、小于 α 的特征值的个数和关系式 *3.18* 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 $\alpha_k = \alpha$，则可以得到新的两个小区间 $[\alpha_0,\ \alpha_k)$ 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 *3.7.1* *(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:

$$(A - \lambda I)x^{k+1} = x^k \tag{3.19}$$

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 *3.19* 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。

§3.7.2 Householder 变换

Householder 变换是一个特殊的正交变换，它具有如下的形式:

$$H = I - 2uu^T \tag{3.20}$$

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 *3.20* 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 $Hx = \alpha e_1$。下面我们介绍如何计算这个 *Householder* 变换，假设 u 是一个任意的向量，则 $H = I - 2uu^T/\|u\|_2^2$ 是一个 *Householder* 变换。为使 $Hx = x - 2(u^Tx/\|u\|_2^2)u = \alpha e_1$，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:

$$\frac{\beta^2 - \|x\|_2^2}{\|x\|_2^2 + 2\beta x_1 + \beta^2}x - \frac{2(\|x\|_2^2 + \beta x_1)}{\|x\|_2^2 + 2\beta x_1 + \beta^2}\beta e_1 = \alpha e_1 \tag{3.21}$$

为使等式 *3.21* 对任意的 x 成立，则必有 $\beta^2 - \|x\|_2^2 = 0$。在计算过程中，为保证数值稳定性，取 $\beta = sign(x_1)\|x\|_2$。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都离不开这个变换。比如矩阵 QR 分解、化对称矩阵为三对角矩阵等。

§3.7.3 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

$$A = \begin{pmatrix} \alpha & u^T \\ u & B \end{pmatrix} \tag{3.22}$$

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：

$$G^T A G = \begin{pmatrix} 1 & 0 \\ 0 & H \end{pmatrix}^T \begin{pmatrix} \alpha & u^T \\ u & B \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & H \end{pmatrix} = \begin{pmatrix} \alpha & \beta e_1^T \\ \beta e_1 & H^T B H \end{pmatrix} \tag{3.23}$$

同样我们可以对矩阵 $H^T B H$ 做和式 *3.23* 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 $H^T B H$。下面考虑如何计算 $H^T B H$，假设 *Householder* 变换 $H = I - 2vv^T/(v^T v)$，则有：

$$\begin{aligned} H^T B H &= (I - 2vv^T/(v^T v))B(I - 2vv^T/(v^T v)) \\ &= B - 2vv^T B/(v^T v) - 2Bvv^T/(v^T v) + 4(v^T B v)vv^T/(v^T v)^2 \end{aligned} \tag{3.24}$$

令 $\tau = 2/(v^T v)$，$x = Bv$，$w = \frac{v^T x}{2} v - \tau x$，则：

$$H^T B H = B - vw^T - wv^T \tag{3.25}$$

经过变形的修正公式 *3.25* 比直接计算方法 *3.24* 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 *3.25*。由此可见并行计算过程是比较容易的，就不再赘述。

§3.8 作业

1. 假设矩阵 B 是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 $A = B - uv^T - vu^T$ 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。

第三部分

并行实现

第四章　并行程序设计

并行程序设计是并行软件开发的基础之一，针对不同的并行计算机，以及不同的并行实现平台，其实现方式是不同的。在这一部分，简单介绍并行实现过程所需的基本技术。

§4.1　并行编程模式的主要类型

并行编程模式主要有如下的三种类型:

- 主从模式 *(Master-slave)*: 有一个主进程，其它为从进程。在这种模式中，主进程一般负责整个并行程序的数据控制，从进程负责对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。一般情况下，从进程之间不发生数据交换，数据的交换过程是通过主进程来完成的。
- 对称模式 *(SPMD)*: 在这种编程模式中，没有哪个进程是主进程，每个进程的地位是相同的。然而，在并行实现过程中，我们总是要在这些进程中选择一个进行输入输出的进程，它扮演的角色和主进程类似。
- 多程序模式 *(MPMD)*: 在每个处理机上执行的程序可能是不同的，在某些处理机上可能执行相同的程序。

§4.2　并行程序的基本特点

并行程序和串行程序没有很大的差别，是为了实现并行算法在并行计算机上的执行。主要包括三个部分:

- 进入并行环境: 这部分是要让系统知道此程序是并行程序，启动并行计算环境。在这个过程中，产生并行程序所需要的各种环境变量。
- 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。
- 退出并行环境: 通知并行计算系统，从这里开始，不再使用并行计算环境。一般来说，只要退出并行计算环境，意味着将结束程序的运行。

§4.3　并行程序的实现技术

并行程序在实现过程中，有其特有的技术，是产生高效率并行程序的基础。这里以目前流行的并行程序模式 *SPMD* 为例，简单介绍这类程序在实现过程中需要掌握的基本技

术。

- 进程控制: 在 *SPMD* 并行程序的编写过程中，因为只有一份程序，每个处理机上执行的是相同的程序。因此，对于每个进程来说，需要知道自己是属于哪个进程，从而来确定该进程需要完成的任务。进程控制是并行程序的重要组成部分，所有的数据处理和交换过程都离不开进程标识，是在实现过程中必须时刻要牢记的。
- 数据交换: 在 *SPMD* 模式中，数据交换是其主要特征。进程之间的协同工作，信息沟通等都离不开数据交换。如何合理地实现数据交换，是提高并行计算程序性能的关键之一。对于一个给定的并行计算方法，为了高效率的实现，需要仔细分析数据依赖关系，尽可能减少不必要的数据交换，同时也要尽最大努力使数据交换在最少的次数内完成。
- 面向对象化: 建立自己的通讯库，使程序具有更加直观的可读性。比如说，在处理机中要进行矩阵传输，那就需要定制一个子程序来完成这项任务。

第五章　消息传递编程接口 MPI

§5.1　MPI 简介

MPI 是英文 *Message Passing Interface* 的缩写, 是基于消息传递编写并行程序的一种用户界面. 消息传递是目前并行计算机上广泛使用的一种程序设计模式, 特别是对分布式存储的可扩展的并行计算机 *SPCs (Scalable Parallel Computers)* 和工作站机群 *NOWs (Networks of Workstations)* 或 *COWs (Clusters of Workstations)*. 尽管还有很多其它的程序实现方式, 但是过程之间的通信采用消息传递已经是一种共识. 在 *MPI* 和 *PVM* 问世以前, 并行程序设计与并行计算机系统是密切相关的, 对不同的并行计算机就要编写不同的并行程序, 给并行程序设计和应用带来了许多麻烦, 广大并行计算机的用户迫切需要一些通用的消息传递用户界面, 使并行程序具有和串行程序一样的可移植性.

在过去的 *4* 年中, 国际上确定了 *MPI* 为消息传递用户界面标准, 自从 *1994* 年 *6* 月推出 *MPI* 以来, 它已被广泛接受和使用, 目前国际上推出的所有并行计算机都支持 *MPI* 和 *PVM*. 对于使用 *SPCs* 的用户来说, 编写 *SPMD* 并行程序使用 *MPI* 可能更为方便. 在使用 *MPI* 函数与常数时要注意, *MPI* 的所有函数与常数均以 `MPI_` 开头. 在 *C* 程序中, 所有常数的定义除下划线外一律由大写字母组成, 在函数和数据类型定义中, 接 `MPI_` 之后的第一个字母大写, 其余全部为小写字母, 在所有函数调用之后都将返回一个错误信息码, 对于 *Fortran* 程序, *MPI* 函数全部以过程方式调用, 其错误码以哑元参数返回. 此外, *MPI* 是按进程组 *(Process Group)* 方式工作的, 所有 *MPI* 程序在开始时均被认为是在通信子 `MPI_COMM_WORLD` 所拥有的进程组中工作, 之后用户可以根据自己的需要, 建立其它的进程组. 此外还需注意的是, 所有 *MPI* 的通信一定要在通信子 *(communicator)* 中进行. 此处我们选择了使用 *MPI* 编程中常用的函数作介绍, 并结合并行计算的实际需要, 给出了大量的 *Fortran* 程序示例, 以便增强对 *MPI* 函数的理解, 关于 *MPI* 的详细介绍可参见文献 *[5]*.

§5.2　MPI 程序实例

在这一节中, 给出一个计算 π 的并行程序。从微积分的知识可知

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \tag{5.1}$$

这里我们采用梯形计算公式，求 5.1 的数值积分。假设将区间分成 n 等份，$h = 1/n$，记 $x_i = ih$，$i = 0, 1, \ldots, n$，即有：

$$\int_0^1 \frac{4}{1+x^2}dx = \sum_{i=0}^{n-1}[\frac{h}{2}(\frac{4}{1+x_i^2} + \frac{4}{1+x_{i+1}^2}) - \frac{h^3}{12}(\frac{4}{1+\xi_i^2})''] \tag{5.2}$$

对于给定的精度 ϵ，可以确定 $N = \lceil\sqrt{4/3\epsilon}\rceil$。即当 $n \geq N$ 时，有：

$$|\int_0^1 \frac{4}{1+x^2}dx - \sum_{i=0}^{n-1}[\frac{h}{2}(\frac{4}{1+x_i^2} + \frac{4}{1+x_{i+1}^2})]| < \epsilon \tag{5.3}$$

例 *5.2.1* 并行计算 π 值

```
      program computing_pi
      include 'mpif.h'

      integer iam, np, comm, ierr
      integer n, i, num, is, ie
      real*8 pi, h, eps, xi, s

      call mpi_init(ierr)
      call mpi_comm_dup(mpi_comm_world, comm, ierr)
      call mpi_comm_rank(comm, iam, ierr)
      call mpi_comm_size(comm, np, ierr)
      print *, 'Process ', iam, ' of ', np, ' is running!'

      if(iam .eq. 0) then
        write(*, *) 'Number of digits(1-16)= '
        read(*, *) num
      endif

      call mpi_bcast(num, 1, mpi_integer, 0, comm, ierr)
      eps = 1
      do 10 i=1, num
        eps = eps * 0.1
10    continue

      n = sqrt(4.0/(3.0*eps))
      h = 1.0/n
      num = n/np
      if(iam .eq. 0) then
        s = 2.0
        xi = 0
        is = 0
```

```
        ie = num
      elseif(iam .eq. np-1) then
        s = 1.0
        is = iam*num
        ie = n - 1
        xi = is * h
      else
        s = 0.0
        is = iam*num
        ie = is + num
        xi = is * h
      endif

      do 20 i=is+1, ie
        xi = xi + h
        s = s + 4.0/(1.0+xi*xi)
20    continue

      call mpi_reduce(s, pi, 1, mpi_double_precision, mpi_sum,
     &                0, comm, ierr)

      if(iam .eq. 0) then
        pi = h*pi

        write(*, *) 'The pi = ', pi
      endif
      call mpi_finalize(ierr)

      end
```

第六章　MPI 并行环境管理函数

在 *MPI* 并行程序中, 必不可少的二个函数是 MPI_Init( ) 和 MPI_Finalize( ). 它们的详细定义如下:

MPI_INIT	
C	int MPI_Init(int *argc, char ***argv)
F	MPI_INIT(IERROR)
	INTEGER IERROR

由于 *ANSI C* 的主程序 *main* 接受参数 argc 和 argv, 如此使用 MPI_Init 可将这些参数传送到每个进程中. 在 *Fortran* 程序中, MPI_INIT 只返回一个错误码. 这个函数初始化 *MPI* 并行程序的执行环境, 它必须在调用所有其它 *MPI* 函数 *(*除 MPI_INITIALIZED*)* 之前被调用, 并且在一个 *MPI* 程序中, 只能被调用一次.

MPI_FINALIZE	
C	int MPI_Finalize(void)
F	MPI_FINALIZE(IERROR)
	INTEGER IERROR

这个函数清除 *MPI* 环境的所有状态. 即一但它被调用, 所有 *MPI* 函数都不能再调用, 其中包括 MPI_INIT.

在一个程序中, 如果不清楚是否已经调用了 MPI_INIT, 可以使用 MPI_INITIALIZED 来检查, 它是唯一的可以在调用 MPI_INIT 之前使用的函数.

MPI_INITIALIZED	
C	int MPI_Initialized(int flag)
F	MPI_INITIALIZED(FLAG, IERROR)
	LOGICAL FLAG
	INTEGER IERROR

参数说明	
OUT	FLAG, 如果 MPI_INIT 被调用, 返回值为 TRUE, 否则为 FALSE.

另外在 *MPI* 中还提供了一个用来检查出何种错误的函数 MPI_ERROR_STRING, 它将给出错误信息.

MPI_ERROR_STRING

C	`int MPI_Error_string( int errorcode, char *string, int *len )`

F	`MPI_ERROR_STRING( ERRORCODE, STRING, LEN, IERROR )`
	`INTEGER ERRORCODE, LEN, IERROR`
	`CHARACTER*(*) STRING`

参数说明

IN	ERRORCODE,	由 *MPI* 函数返回的错误码.
OUT	STRING,	对应 ERRORCODE 的错误信息.
OUT	LEN,	错误信息 STRING 的长度.

在使用此函数时, 应注意 STRING 的长度最小应为 MPI_MAX_ERROR_STRING.

第七章　MPI 进程控制函数

在这一章中, 介绍与进程组有关的一些基本函数, 其中包括如何建立进程组和通信子, 灵活使用这些函数在实际程序设计中会带来巨大方便.

§7.1　MPI 进程组操作函数

MPI_COMM_GROUP

C	int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
F	MPI_COMM_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR

参数说明

IN	COMM,	通信子.
OUT	GROUP,	对应 COMM 的进程组.

MPI_COMM_GROUP 这个函数是用来建立一个通信子对应的新进程组, 之后就可以对此进程组进行需要的操作.

MPI_GROUP_FREE

C	int MPI_group_free(MPI_Group *group)
F	MPI_GROUP_FREE(GROUP, IERROR) INTEGER GROUP, IERROR

参数说明

INOUT	GROUP,	释放进程组并返回 MPI_GROUP_NULL.

当 MPI_GROUP_FREE 被调用之后, 任何关于此进程组的操作将视为无效.

MPI_GROUP_SIZE

C	int MPI_Group_size(MPI_Group group, int *size)
F	MPI_GROUP_SIZE(GROUP, SIZE, IERROR) INTEGER GROUP, SIZE, IERROR

参数说明

IN	GROUP,	进程组.
OUT	SIZE,	进程组中的进程个数.

如果进程组是 MPI_GROUP_EMPTY, 则返回值 SIZE 为 0.

MPI_GROUP_RANK

C	int MPI_Group_rank(MPI_Group group, int *rank)
F	MPI_GROUP_RANK(GROUP, RANK, IERROR) INTEGER GROUP, RANK, IERROR

参数说明

IN	GROUP,	进程组.
OUT	RANK,	进程在进程组中的编号.

如果进程不是进程组中的成员, 则返回值 RANK 为 MPI_UNDEFINED.

MPI_GROUP_TRANSLATE_RANKS

C	int MPI_Group_translate_ranks(MPI_Group group1,int n,int *ranks1, MPI_Group group2, int *ranks2)
F	MPI_GROUP_TRANSLATE_RANKS(GROUP1,N,RANKS1,GROUP2,RANKS2,IERROR) INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR

参数说明

IN	GROUP1,	进程组 *1*.
IN	N,	RANKS1 和 RANKS2 中数组元素个数.
IN	RANKS1,	进程组 *1* 中有效编号组成的数组.
IN	GROUP2,	进程组 *2*.
OUT	RANKS2,	RANKS1 中的元素在进程组 *2* 中的对应编号.

如果属于进程组 *1* 的某个进程可以在 RANKS1 中找到, 而这个进程不属于进程组 *2*, 则在 RANKS2 中对应 RANKS1 的位置返回值为 MPI_UNDEFINED.

MPI_GROUP_INCL

C	int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
F	MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR) INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

参数说明

IN	GROUP,	进程组.
IN	N,	RANKS 数组中元素的个数和新进程组的大小.
IN	RANKS,	将在新进程组中出现的旧进程组中的编号.
OUT	NEWGROUP,	由 RANKS 定义的顺序导出的新进程组.

MPI_GROUP_EXCL

```
C  int MPI_Group_excl( MPI_Group group, int n, int *ranks,
                       MPI_Group *newgroup)
F  MPI_GROUP_EXCL( GROUP, N, RANKS, NEWGROUP, IERROR )
   INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

参数说明

IN	GROUP,	进程组.
IN	N,	RANKS 数组中元素的个数.
IN	RANKS,	在新进程组中不出现的旧进程组中的编号.
OUT	NEWGROUP,	旧进程组中不在 RANKS 里的元素组成的新进程组.

MPI_GROUP_UNION

```
C  int MPI_Group_union( MPI_Group group1, MPI_Group group2,
                        MPI_Group *newgroup)
F  MPI_GROUP_UNION( GROUP1, GROUP2, NEWGROUP, IERROR )
   INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

参数说明

IN	GROUP1,	进程组 *1*.
IN	GROUP2,	进程组 *2*.
OUT	NEWGROUP,	进程组 *1* 和进程组 *2* 的并.

MPI_GROUP_INTERSECTION

```
C  int MPI_Group_intersection( MPI_Group group1, MPI_Group group2,
                               MPI_Group *newgroup)
F  MPI_GROUP_INTERSECTION( GROUP1, GROUP2, NEWGROUP, IERROR )
   INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

参数说明		
IN	GROUP1,	进程组 *1*.
IN	GROUP2,	进程组 *2*.
OUT	NEWGROUP,	进程组 *1* 和进程组 *2* 的交.

MPI_GROUP_DIFFERENCE

C	int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
F	MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

参数说明		
IN	GROUP1,	进程组 *1*.
IN	GROUP2,	进程组 *2*.
OUT	NEWGROUP,	进程组 *1* 和进程组 *2* 的差.

以上关于进程组操作的函数功能, 和数学中集合运算相同.

§7.2 MPI 通信子操作

下面介绍的关于通信子的操作和上述关于进程组操作十分类似.

MPI_COMM_SIZE

C	int MPI_Comm_size(MPI_Comm comm, int *size)
F	MPI_COMM_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR

参数说明		
IN	COMM,	通信子.
OUT	SIZE,	通信子中的进程个数.

MPI_COMM_RANK

C	int MPI_Comm_rank(MPI_Comm comm, int *rank)
F	MPI_COMM_RANK(COMM, RANK, IERROR) INTEGER COMM, RANK, IERROR

参数说明		
IN	COMM,	通信子.
OUT	RANK,	通信子中的进程编号.

MPI_COMM_DUP

C	int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
F	MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
	INTEGER COMM, NEWCOMM, IERROR

参数说明		
IN	COMM,	通信子.
OUT	NEWCOMM,	COMM 通信子的复制.

MPI_COMM_CREATE

C	int MPI_Comm_create(MPI_Comm comm,MPI_Group group,MPI_Comm *newcomm)
F	MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
	INTEGER COMM, GROUP, NEWCOMM, IERROR

参数说明		
IN	COMM,	通信子.
IN	GROUP,	通信子 COMM 的一个子集.
OUT	NEWCOMM,	对应 GROUP 的新通信子.

MPI_COMM_SPLIT

C	int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
F	MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
	INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

参数说明		
IN	COMM,	通信子.
IN	COLOR,	子集控制值.
IN	KEY,	子集中进程编号的顺序.
OUT	NEWCOMM,	由此产生的新通信子.

这个函数划分 COMM 所对应的进程组为不相交的子进程组, 每个子进程组有一个共同的值 COLOR, 就是说, 每个子进程组中包含 COLOR 相同的所有进程.

MPI_COMM_FREE

C	int MPI_Comm_free(MPI_Comm *comm)
F	MPI_COMM_FREE(COMM, IERROR) INTEGER COMM, IERROR

参数说明

INOUT	COMM, 通信子.

第八章　MPI 点到点通信函数

点到点通信是指一对进程之间的数据转换, 也就是说, 一边发送数据另一边接收数据. 点到点通信是 *MPI* 通信机制的基础, 它分为同步通信和异步通信二种机制.

§8.1　阻塞式通信函数

MPI_SEND

C	int MPI_Send(void* buf,int count,MPI_Datatype datatype,int dest, int tag, MPI_Comm comm)
F	MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

参数说明

IN	BUF,	所要发送消息数据的首地址.
IN	COUNT,	发送消息数组元素的个数.
IN	DATATYPE,	发送消息的数据类型.
IN	DEST,	接收消息的进程编号.
IN	TAG,	消息标签.
IN	COMM,	通信子.

这里 COUNT 是 BUF 的元素个数而不是字节数.

MPI_RECV

C	int MPI_Recv(void* buf,int count,MPI_Datatype datatype,int source, int tag, MPI_Comm comm, MPI_Status *status)
F	MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER COUNT,DATATYPE,SOURCE,TAG,COMM, STATUS(MPI_STATUS_SIZE),IERROR

参数说明		
OUT	BUF,	接收消息数据的首地址.
IN	COUNT,	接收消息数组元素的最大个数.
IN	DATATYPE,	接收消息的数据类型.
IN	SOURCE,	发送消息的进程编号.
IN	TAG,	消息标签.
IN	COMM,	通信子.
OUT	STATUS,	接收消息时返回的状态.

在这个函数中, 可以不指定 SOURCE 和 TAG, 而用 MPI_ANY_SOURCE 和 MPI_ANY_TAG 来代替, 用于接收任何进程发送的消息或任何编号的消息. 接收消息时返回的状态 STATUS 在 *C* 语言中是用结构定义的, 在 *Fortran* 中是用数组定义的, 其中包括 MPI_SOURCE, MPI_TAG 和 MPI_ERROR. 此外 STATUS 还包含接收消息元素的个数, 但它不是显式给出的, 需要用到后面给出的函数 MPI_GET_COUNT.

MPI 的基本数据类型定义与相应的 *Fortran* 和 *C* 的数据类型对照关系如下:

Fortran 程序中可使用的 *MPI* 数据类型

MPI datatype	*Fortran datatype*
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

C 程序中可使用的 *MPI* 数据类型

MPI datatype	*Fortran datatype*
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

这里 MPI_BYTE 和 MPI_PACKED 不对应 *Fortran* 或 *C* 中的任何数据类型, MPI_BYTE 是由一个字节组成的, 而 MPI_PACKED 将在后面介绍. 到目前为止, 我们已经可以编写一些简单的 *MPI* 程序, 下面给出一个 *MPI* 程序的例子, 我们结合这个例子说明一个 *MPI* 程序的组成部分. 重点需要掌握的 *MPI* 函数有初始化并行环境的函数 MPI_INIT, 得到本进程的编号函数 MPI_COMM_RANK, 得到全部进程个数的函数 MPI_COMM_SIZE, 退出 *MPI* 并行环境的函数 MPI_FINALIZE, 以及进行消息传递的二个函数 MPI_SEND 和 MPI_RECV.

例 *8.1.1* 假设一共有 p 个进程, 在进程编号为 myid ($\texttt{myid}=0,\ldots,p-1$) 的进程中有一个整数 m, 我们要把 m 传送到进程 $(myid+1)\ mod\ p$ 中.

```
      program ring
c c The header file mpif.h must be included when you use MPI
fuctions. c
      include 'mpif.h'
      integer myid, p, mycomm, ierr, m, status(mpi_status_size),
     &        next, front, mod, n
c c Create MPI parallel environment and get the necessary data. c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
c c Beginning the main parallel work on each process. c
      m = myid
      front = mod(p+myid-1, p)
      next = mod(myid+1, p)
```

```
c c Communication with each other. c
      if(myid .eq. 0) then
        call mpi_recv(n, 1, mpi_integer, front, 1, mycomm, status, ierr)
        call mpi_send(m, 1, mpi_integer, next, 1, mycomm, ierr)
        m = n
      else
        call mpi_send(m, 1, mpi_integer, next, 1, mycomm, ierr)
        call mpi_recv(m, 1, mpi_integer, front, 1, mycomm, status, ierr)
      endif
c c Ending of parallel work. c
      print *, 'The value of m is ', m, ' on Process ', myid
      call mpi_comm_free(mycomm, ierr)
c c Remove MPI parallel environment. c
      call mpi_finalize(ierr)
      end
```

MPI_GET_COUNT

C	int MPI_Get_count(MPI_Status status,MPI_Datatype datatype,int *count)
F	MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

参数说明

IN	STATUS,	接收消息时返回的状态.
IN	DATATYPE,	接收消息的数据类型.
OUT	COUNT,	接收消息数组元素的个数.

由于在接收消息时使用的是最大个数, 为了准确知道接收消息的个数, 就要使用此函数.

MPI_SENDRECV

C	int MPI_Sendrecv(void *sendbuf,int sendcount,MPI_Datatype sendtype, int dest,int sendtag,void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
F	MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT,RECVTYPE,SOURCE,RECVTAG,COMM,STATUS,IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

参数说明		
IN	SENDBUF,	所要发送消息数据的首地址.
IN	SENDCOUNT,	发送消息数组元素的个数.
IN	SENDTYPE,	发送消息的数据类型.
IN	DEST,	接收消息的进程编号.
IN	SENDTAG,	发送消息标签.
OUT	RECVBUF,	接收消息数据的首地址.
IN	RECVCOUNT,	接收消息数组元素的最大个数.
IN	RECVTYPE,	接收消息的数据类型.
IN	SOURCE,	发送消息的进程编号.
IN	RECVTAG,	接收消息标签.
IN	COMM,	通信子.
OUT	STATUS,	接收消息时返回的状态.

这是发送消息和接收消息组合在一起的一个函数, 好处是不用考虑先发送还是先接收消息. 在示例 *8.1.1* 中从通信开始到通信结束部分可用如下的一个函数来完成:

```
      call mpi_sendrecv(m, 1, mpi_integer, next, 1, n, 1, mpi_integer,
     &                  front, 1, mycomm, status, ierr)
      m = n
```

由此可见使用 MPI_SENDRECV 可以使程序简化, 更重要的是在通信过程中不需要考虑哪个进程先发送还是先接收, 从而可以避免消息传递过程中的死锁.

MPI_SENDRECV_REPLACE

C
```
int MPI_Sendrecv_replace(void *sendbuf,int count,MPI_Datatype datatype,
                         int dest,int sendtag,int source, int recvtag,
                         MPI_Comm comm, MPI_Status *status )
```

F
```
MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG,
                     SOURCE, RECVTAG, COMM, STATUS, IERROR )
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
        COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

参数说明		
OUT	BUF,	发送和接收消息数据的首地址.
IN	COUNT,	发送和接收消息数组元素的个数.
IN	DATATYPE,	发送和接收消息的数据类型.
IN	DEST,	接收消息的进程编号.
IN	SENDTAG,	发送消息标签.
IN	SOURCE,	发送消息的进程编号.
IN	RECVTAG,	接收消息标签.
IN	COMM,	通信子.
OUT	STATUS,	接收消息时返回的状态.

在这个函数中, 发送和接收使用同一个消息缓冲区 BUF. 因此示例 *8.1.1* 的通信部分可进一步改写成如下的形式:

```
      call mpi_sendrecv_replace(m, 1, mpi_integer, next, 1,
     &                          front, 1, mycomm, status, ierr)
```

在进行环形通信时, 使用 MPI_SENDRECV_REPLACE 是非常方便的. 为了更方便使用此函数, *MPI* 定义了一个空进程 MPI_PROC_NULL, 如果通信采用这个空进程将不起任何作用. 比如在示例 *8.1.1* 中, 如果我们不需要从第 $p-1$ 个进程向第 0 个进程传送 m, 则可用如下方式来实现:

```
      ......
      if(myid .eq. 0) front = MPI_PROC_NULL
      if(myid .eq. p-1) next = MPI_PROC_NULL
      call mpi_sendrecv_replace(m, 1, mpi_integer, next, 1,
     &                          front, 1, mycomm, status, ierr)
```

MPI_PROBE

C	int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status)
F	MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

参数说明		
IN	SOURCE,	发送消息进程的编号.
IN	TAG,	接收消息的标签.
IN	COMM,	通信子.
OUT	STATUS,	返回到达消息的状态.

此函数与下面给出的 MPI_IPROBE 的功能基本类似, 但是 MPI_PROBE 一定要等到消息才返回.

MPI_IPROBE

C	int MPI_Iprobe(int source, int tag, MPI_Comm comm, int* flag, MPI_Status *status)
F	MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR) LOGICAL FLAG INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

参数说明

IN	SOURCE,	发送消息进程的编号.
IN	TAG,	接收消息的标签.
IN	COMM,	通信子.
OUT	FLAG,	如果指定消息已经达到, FLAG 返回值为 TRUE.
OUT	STATUS,	返回到达消息的状态.

这个函数对由 SOURCE, TAG 和 COMM 确定的消息是否到达, 它都将立即返回. 这里 SOURCE 和 TAG 同 MPI_RECV 中一样, 可以用通配符 *(wildcard)* MPI_ANY_SOURCE 和 MPI_ANY_TAG 来代替.

§8.2 非阻塞式通信函数

非阻塞式通信函数是指在通信过程中, 不需要等待通信结束就返回, 通常这种通信过程交由计算机的后台来处理. 如果计算机系统提供硬件支持非阻塞式通信函数, 就可以使计算与通信在时间上的重叠, 从而提高并行计算的效率.

MPI_ISEND

C	int MPI_Isend(void* buf,int count,MPI_Datatype datatype,int dest, int tag, MPI_Comm comm, MPI_Request *request)
F	MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

参数说明

IN	BUF,	所要发送消息数据的首地址.
IN	COUNT,	发送消息数组元素的个数.
IN	DATATYPE,	发送消息的数据类型.
IN	DEST,	接收消息的进程编号.
IN	TAG,	消息标签.
IN	COMM,	通信子.
OUT	REQUEST,	请求句柄以备将来查询.

MPI_IRECV

C
```
int MPI_Irecv(void* buf,int count,MPI_Datatype datatype,int source,
              int tag, MPI_Comm comm, MPI_Request *request )
```

F
```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

参数说明

OUT	BUF,	接收消息数据的首地址.
IN	COUNT,	接收消息数组元素的个数.
IN	DATATYPE,	接收消息的数据类型.
IN	SOURCE,	发送消息的进程编号.
IN	TAG,	消息标签.
IN	COMM,	通信子.
OUT	REQUEST,	请求句柄以备将来查询.

MPI_ISEND 和 MPI_IRECV 不需要等待发送或接收消息完成就可执行其余的任务, 看发送或接收过程是否结束, 有下面的一些函数:

MPI_WAIT

C
```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
```

F
```
MPI_WAIT( REQUEST, STATUS, IERROR )
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

参数说明

INOUT	REQUEST,	请求句柄.
OUT	STATUS,	发送或接收消息的状态.

如果 REQUEST 所指的操作已经完成, MPI_WAIT 将结束等待状态.

MPI_TEST

C	int MPI_Test(MPI_Request *request, int* flag, MPI_Status *status)
F	MPI_TEST(REQUEST, FLAG, STATUS, IERROR) LOGICAL FLAG INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

参数说明

INOUT	REQUEST,	请求句柄.
OUT	FLAG,	REQUEST 所指的操作已经完成返回值为 TRUE.
OUT	STATUS,	发送或接收消息的状态.

以上这二个函数中 STATUS 对发送消息操作是没有定义的, 唯一可在发送消息操作中使用 STATUS 的是查询函数 MPI_TEST_CANCELLED.

MPI_REQUEST_FREE

C	int MPI_Request_free(MPI_Request *request)
F	MPI_REQUEST_FREE(REQUEST, IERROR) INTEGER REQUEST, IERROR

参数说明

INOUT	REQUEST,	请求句柄, 返回值为 MPI_REQUEST_NULL.

以上是对单个 REQUEST 进行查询的函数, 如果要对多个请求句柄进行查询, 有下面的一些函数可以使用.

MPI_WAITANY

C	int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)
F	MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR) INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERROR

参数说明

IN	COUNT,	请求句柄的个数.
INOUT	ARRAY_OF_REQUESTS,	请求句柄数组.
OUT	INDEX,	已经完成通信操作的句柄指标.
OUT	STATUS,	消息的状态.

这个函数当所有请求句柄中至少有一个已经完成通信操作, 就返回, 如果有多于一个请求句柄已经完成, MPI_WAITANY 将随机选择其中的一个并立即返回.

MPI_TESTANY

```
C  int MPI_Testany( int count, MPI_Request *array_of_requestS,
                    int *index, int *flag, MPI_Status *status )
F  MPI_TESTANY(COUNT,ARRAY_OF_REQUESTS,INDEX,FLAG,STATUS,IERROR)
   LOGICAL FLAG
   INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX,
           STATUS(MPI_STATUS_SIZE), IERROR
```

参数说明

IN	COUNT,	请求句柄的个数.
INOUT	ARRAY_OF_REQUESTS,	请求句柄数组.
OUT	INDEX,	已经完成通信操作的句柄指标.
OUT	FLAG,	如果有一个已经完成, 则 FLAG=TRUE.
OUT	STATUS, 消息的状态.	

这个函数无论有没有通信操作完成都将立即返回.

MPI_WAITALL

```
C  int MPI_Waitall( int count, MPI_Request *array_of_requests,
                    MPI_Status *array_of_statuses )
F  MPI_WAITALL(COUNT,ARRAY_OF_REQUESTS,ARRAY_OF_STATUSES,IERROR)
   INTEGER COUNT, ARRAY_OF_REQUESTS(*),
           ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
```

参数说明

IN	COUNT,	请求句柄的个数.
INOUT	ARRAY_OF_REQUESTS,	请求句柄数组.
INOUT	ARRAY_OF_STATUSES,	所有消息的状态数组.

这个函数当所有通信操作完成之后才返回, 否则将一直等待.

MPI_TESTALL

```
C  int MPI_Testall( int count, MPI_Request *array_of_requests,
                    int *flag, MPI_Status *array_of_statuses )
F  MPI_TESTALL(COUNT,ARRAY_OF_REQUESTS,FLAG,ARRAY_OF_STATUSES,IERROR)
   LOGICAL FLAG
   INTEGER COUNT, ARRAY_OF_REQUESTS(*),
           ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
```

参数说明

IN	COUNT,	请求句柄的个数.
INOUT	ARRAY_OF_REQUESTS,	请求句柄数组.
OUT	FLAG,	如果有一个没完成, 则 FLAG=FALSE.
INOUT	ARRAY_OF_STATUSES,	所有消息的状态数组.

这个函数无论所有通信操作是否完成都将立即返回.

MPI_CANCEL

```
C  int MPI_Cancel( MPI_Request *request )
F  MPI_CANCEL( REQUEST, IERROR )
   INTEGER REQUEST, IERROR
```

参数说明

INOUT	REQUEST,	请求句柄.

这个函数用来取消一个发送或接收操作, 其后要使用 MPI_WAIT 或 MPI_TEST 等函数, 因此使用此函数会损失许多效率. 但是要知道关于 REQUEST 请求句柄的操作是否已经被取消, 还要使用下面的检测函数:

MPI_TEST_CANCELLED

```
C  int MPI_Test_cancelled( MPI_Status *status, int *flag )
F  MPI_TEST_CANCELLED( STATUS, FLAG, IERROR )
   LOGICAL FLAG
   INTEGER REQUEST, IERROR
```

参数说明

IN	STATUS,	消息的状态.
OUT	FLAG,	如果已经取消, 则 FLAG=TRUE.

§8.3 特殊的点到点通信函数

MPI_SEND_INIT

C
```
int MPI_Send_init(void* buf,int count,MPI_Datatype datatype,int dest,
                  int tag, MPI_Comm comm, MPI_Request *request )
```

F
```
MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR )
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

参数说明

IN	BUF,	所要发送消息数据的首地址.
IN	COUNT,	发送消息数组元素的个数.
IN	DATATYPE,	发送消息的数据类型.
IN	DEST,	接收消息的进程编号.
IN	TAG,	消息标签.
IN	COMM,	通信子.
OUT	REQUEST,	请求句柄以备将来查询.

MPI_RECV_INIT

C
```
int MPI_Recv_init(void* buf,int count,MPI_Datatype datatype,int source,
                  int tag, MPI_Comm comm, MPI_Request *request )
```

F
```
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

参数说明

OUT	BUF,	接收消息数据的首地址.
IN	COUNT,	接收消息数组元素的个数.
IN	DATATYPE,	接收消息的数据类型.
IN	SOURCE,	发送消息的进程编号.
IN	TAG,	消息标签.
IN	COMM,	通信子.
OUT	REQUEST,	请求句柄以备将来查询.

MPI_SEND_INIT 和 MPI_RECV_INIT 是二个持久性通信函数, 它们的参数和前面讲到的异步通信中的发送与接收相同, 由此可以把通信以通道方式建立起来对应关系, 从而提高通信效率. 但是仅有这二个函数还不能够达到通信的目的, 它们还需要用 MPI_START 或 MPI_STARTALL 来激活.

MPI_START

C	int MPI_Start(MPI_Request *request)
F	MPI_START(REQUEST, IERROR)
	INTEGER REQUEST, IERROR

参数说明

INOUT	REQUEST,	请求句柄.

在使用 MPI_START 之后, MPI_SEND_INIT 和 MPI_RECV_INIT 就同异步发送与接收.

MPI_STARTALL

C	int MPI_Startall(int count, MPI_Request *array_of_request)
F	MPI_STARTALL(COUNT, ARRAY_OF_REQUEST, IERROR)
	INTEGER COUNT, ARRAY_OF_REQUEST, IERROR

参数说明

IN	COUNT,	需要激活的请求句柄个数.
INOUT	ARRAY_OF_REQUEST,	请求句柄数组.

例 *8.3.1* 在示例 *8.1.1* 中, 整数 m 只是在进程环中向前移动一次, 如果我们要移动多次, 可以使用下面的程序实现:

```
      program mring
      include 'mpif.h'
c
      integer iter
      parameter(iter = 3)
      integer myid, p, mycomm, ierr, m, status(mpi_status_size, 2),
     &        next, front, mod, n, i, requests(2)
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
```

```
c
      m = myid
      front = mod(p+myid-1, p)
      next = mod(myid+1, p)
      call mpi_send_init(m, 1, mpi_integer, next, 1, mycomm,
     &                   requests(1), ierr)
      call mpi_recv_init(n, 1, mpi_integer, front, 1, mycomm,
     &                   requests(2), ierr)
      do 10 i=1, iter
        call mpi_startall(2, requests, ierr)
        call mpi_waitall(2, requests, status, ierr)
        m = n
10    continue
      print *, 'The value of m is ', m, ' on Process ', myid
c
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end
```

§8.4 MPI 的通信模式

前面所提到的通信模式是标准通信模式, 它是由 *MPI* 实现中确定的, 可能选择的是缓冲区模式 *(buffered-mode)* 或同步模式 *(synchronous-mode)*, 此外 *MPI* 还提供了就绪模式 *(ready-mode)*. 这些模式仅对发送消息起作用, 无论用任何模式发送的消息都可用一个接收函数来完成接收消息. 在下面给出的与发送消息有关的函数, 其参数与对应标准通信模式时是完全一致的, 因此这里我们仅给出函数名字及简要说明.

- MPI_BSEND, 使用缓冲区发送消息, 只要缓冲区足够大, 而不管接收进程是否开始接收它都将立即返回, 也就是说, 在发送消息时是不需要等待的.
- MPI_SSEND, 它可以在没有接收信号时就开始发送, 但要等到接收完成之后才能结束通信操作, 好处是无需用户自己申请缓冲区.
- MPI_RSEND, 只当已经有接收信号时才开始发送消息, 否则将出现错误.
- MPI_IBSEND, 异步通信的 MPI_BSEND.
- MPI_ISSEND, 异步通信的 MPI_SSEND.
- MPI_IRSEND, 异步通信的 MPI_RSEND.
- MPI_BSEND_INIT, 建立自己缓冲区的 MPI_SEND_INIT.
- MPI_SSEND_INIT, 同步模式的 MPI_SEND_INIT.
- MPI_RSEND_INIT, 就绪模式的 MPI_SEND_INIT.

有关消息缓冲区的管理, 可以使用如下的二个函数:

MPI_BUFFER_ATTACH

C	`int MPI_Buffer_attach( void *buffer, int size )`

F	`MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR )`
	`<TYPE> BUFFER(*)`
	`INTEGER SIZE, IERROR`

参数说明

IN	BUFFER,	初始化缓冲区的首地址.
IN	SIZE,	缓冲区按字节的大小.

这个函数指明使用消息缓冲区模式通信时用哪一个缓冲区.

MPI_BUFFER_DETACH

C	`int MPI_Buffer_detach( void *buffer, int *size )`

F	`MPI_BUFFER_DETACH( BUFFER, SIZE, IERROR )`
	`<TYPE> BUFFER(*)`
	`INTEGER SIZE, IERROR`

参数说明

OUT	BUFFER,	要拆卸的缓冲区.
OUT	SIZE,	返回缓冲区的字节数.

第九章　MPI 用户自定义的数据类型与打包

§9.1　用户定义的数据类型

在 *MPI* 中用户定义的数据类型和 *C* 语言中的结构是非常类似的, 它包括数据类型、数据长度和数据起始位置, 这个新定义的数据类型同其它 *MPI* 定义的标准数据类型一样, 可以用来作为发送和接收消息的数据类型. 对于这个新定义的数据类型, 具体地说它的组成可以概括为如下的形象描述:

- 它包含了一系列的原始数据类型;
- 和一系列的称之为位移 *(displacement)* 的整数.

数据类型与位移是成对出现的, 这一系列的数据对就是称之为类型映射 *(type map)*. 用类型映射来表示一个新的数据类型为:

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

在这个新的数据类型中, 包括位移是 $disp_i$ 的原始数据类型 $type_i$, 其中 $i = 0, \ldots, n-1$. 如何构造新的数据类型我们将在以下进行详细介绍.

MPI_TYPE_CONTIGUOUS

C	`int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype )`
F	`MPI_TYPE_CONTIGUOUS( COUNT, OLDTYPE, NEWTYPE, IERROR )` `INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR`

参数说明

IN	COUNT,	重复放在一起的 OLDTYPE 的元素个数.
IN	OLDTYPE,	原始数据类型.
OUT	NEWTYPE,	新构造的数据类型.

这是构造数据类型中最简单的一种, 就是把原始数据类型重复 COUNT 次, 我们举个简单的例子来说明其含义. 假设 COUNT=5, OLDTYPE=INTEGER, 则调用此函数之后产生的新的数据类型 NEWTYPE 为 *5* 个整数.

MPI_TYPE_VECTOR

C	`int MPI_Type_vector( int count, int blocklength, int stride, MPI_Datatype oldtype,MPI_Datatype *newtype )`
F	`MPI_TYPE_VECTOR(COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR)` `INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR`

参数说明		
IN	COUNT,	OLDTYPE 的块的个数.
IN	BLOCKLENGTH,	每块的长度.
IN	STRIDE,	用 OLDTYPE 的元素度量的每块起始的间距.
IN	OLDTYPE,	原始数据类型.
OUT	NEWTYPE,	新构造的数据类型.

假设 COUNT=3, BLOCKLENGTH=2, STRIDE=3, OLDTYPE=INTEGER, 则调用此函数得到的新数据类型 NEWTYPE 为 *6* 个整数组成. 这 *6* 个整数是在原始数组中从开始取 *2* 个, 隔一个再取 *2* 个, 再隔一个再取 *2* 个构成的.

MPI_TYPE_HVECTOR

C	int MPI_Type_hvector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
F	MPI_TYPE_HVECTOR(COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

参数说明		
IN	COUNT,	OLDTYPE 的块的个数.
IN	BLOCKLENGTH,	每块的长度.
IN	STRIDE,	用字节数度量的每块起始的间距.
IN	OLDTYPE,	原始数据类型.
OUT	NEWTYPE,	新构造的数据类型.

这个函数要比 MPI_TYPE_VECTOR 更具普遍性, 但是它的每一块是固定不变的, 下面将介绍变块长度的数据类型如何构造.

MPI_TYPE_INDEXED

C	int MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)
F	MPI_TYPE_INDEXED(COUNT,ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR

参数说明

IN	COUNT,	OLDTYPE 的块的个数.
IN	ARRAY_OF_BLOCKLENGTHS,	每块长度数组.
IN	ARRAY_OF_DISPLACEMENTS,	用元素个数定义的每块的起始位移量数组.
IN	OLDTYPE,	原始数据类型.
OUT	NEWTYPE,	新构造的数据类型.

这里 ARRAY_OF_BLOCKLENGTHS 和 ARRAY_OF_DISPLACEMENTS 是 OLDTYPE 定义的元素个数度量的, 而 ARRAY_OF_DISPLACEMENTS 是一个相对值. 前述的 MPI_TYPE_VECTOR 函数是此函数的特例, 也即下面的二个调用产生相同的数据类型.

MPI_TYPE_VECTOR(COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR) 和

MPI_TYPE_INDEXED(COUNT,LENGTHS,DISPLACEMENTS,OLDTYPE,NEWTYPE,IERROR)

LENGTHS(i)=BLOCKLENGTH, DISPLACEMENTS(i)=i*STRIDE, i=0, ..., COUNT-1. 在这里我们给出的函数 MPI_TYPE_INDEXED 的位移量是由 OLDTYPE 定义的元素个数确定的, 下面将要给出的函数的位移量是由字节数确定的.

MPI_TYPE_HINDEXED

C
```
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
                      int *array_of_displacements,
                      MPI_Datatype oldtype, MPI_Datatype *newtype )
```

F
```
MPI_TYPE_HINDEXED(COUNT,ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS,
                  OLDTYPE, NEWTYPE, IERROR )
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
        OLDTYPE, NEWTYPE, IERROR
```

参数说明

IN	COUNT,	OLDTYPE 的块的个数.
IN	ARRAY_OF_BLOCKLENGTHS,	每块长度数组.
IN	ARRAY_OF_DISPLACEMENTS,	用字节数定义的每块的起始位移量数组.
IN	OLDTYPE,	原始数据类型.
OUT	NEWTYPE,	新构造的数据类型.

以上给出的关于数据类型构造函数都是单一的原始数据类型, 下面将给出一个最一般的构造新数据类型的函数.

MPI_TYPE_STRUCT

C
```
int MPI_Type_struct(int count, int *array_of_blocklengths,
                    int *array_of_displacements,
                    MPI_Datatype *array_of_types,MPI_Datatype *newtype)
```

F
```
MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
                ARRAY_OF_TYPES, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
        ARRAY_OF_TYPES(*), NEWTYPE, IERROR
```

参数说明

IN	COUNT,	块的个数.
IN	ARRAY_OF_BLOCKLENGTHS,	每块长度数组.
IN	ARRAY_OF_DISPLACEMENTS,	用字节数定义的每块的起始位移量数组.
IN	ARRAY_OF_TYPES,	原始数据类型数组.
OUT	NEWTYPE,	新构造的数据类型.

在所有 *MPI* 的构造数据类型中, MPI_TYPE_STRUCT 是使用最广泛的一个函数, 正确使用此函数在实际应用中是非常重要的. 到目前为此, 所有构造数据类型的函数都已经介绍了, 但是要使新构造的数据类型真正能够使用, 还需要有以下的一些函数配合才行, 因此关于如何使用这些数据类型构造函数的例子将在其它一些有关的函数介绍完之后给出.

MPI_TYPE_COMMIT

C
```
int MPI_Type_commit( MPI_Datatype *datatype )
```

F
```
MPI_TYPE_COMMIT( DATATYPE, IERROR )
INTEGER DATATYPE, IERROR
```

参数说明

INOUT	DATATYPE,	准备提交的数据类型.

在使用自定义的 *MPI* 数据类型之前, 必须调用 MPI_TYPE_COMMIT.

MPI_TYPE_FREE

C
```
int MPI_Type_free( MPI_Datatype *datatype )
```

F
```
MPI_TYPE_FREE( DATATYPE, IERROR )
INTEGER DATATYPE, IERROR
```

参数说明		
INOUT	DATATYPE,	释放不用的数据类型.

MPI_TYPE_EXTENT

```
C  int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
F  MPI_TYPE_EXTENT( DATATYPE, EXTENT, IERROR )
   INTEGER DATATYPE, EXTENT, IERROR
```

参数说明		
IN	DATATYPE,	数据类型.
OUT	EXTENT,	数据类型拓展后的字节数.

MPI_ADDRESS

```
C  int MPI_Address( void* location, MPI_Aint *address )
F  MPI_ADDRESS( LOCATION, ADDRESS, IERROR )
   <type> LOCATION(*)
   INTEGER ADDRESS, IERROR
```

参数说明		
IN	LOCATION,	在存储器中的变量表示.
OUT	ADDRESS,	变量在存储器中的地址.

为了说明如何构造数据类型和 MPI_TYPE_EXTENT 函数的含义以及 MPI_ADDRESS 的使用, 我们给出一个具体的例子加以解释.

例 9.1.1 假设在 *C* 语言中定义了一个结构, 其中包括一个双精度数和一个字符, 我们要定义这样结构的 *MPI* 数据类型.

```
#include <stdio.h> #include "mpi.h" typedef struct {
  double value; char str; } data;

void main(argc, argv) int argc; char **argv; {
  int p, myid, lens[3]={1, 1, 1}, i;
  MPI_Comm mycomm;  data tst[3];
  MPI_Datatype new, type[3] = {MPI_DOUBLE, MPI_CHAR, MPI_UB};
  MPI_Aint disp[3], size;  MPI_Status status;

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

```
  MPI_Comm_dup( MPI_COMM_WORLD, &mycomm);
  MPI_Comm_rank( mycomm, &myid);  MPI_Comm_size( mycomm, &p);

  if (myid == 0)
    for(i=0; i<3; i++) { tst[i].value = 1.0; tst[i].str='a'; }
  printf("\nThe Process %1d of %1d is running.\n", myid, p);
  MPI_Address(&tst[0], &disp[0]);  MPI_Address(&tst[0].str, &disp[1]);
  for (i=1; i>=0; i--)  disp[i] -= disp[0];
  MPI_Type_struct(3, lens, disp, type, &new);
  MPI_Type_commit(&new);
  MPI_Type_extent(new, &size);

  if(myid == 0) printf("\nThe Extent of Data Type is %d.\n", size);
  if(myid == 0) MPI_Ssend(tst, 3, new, 1, 1, mycomm);
  else if (myid == 1) MPI_Recv(tst, 3, new, 0, 1, mycomm, &status);
  MPI_Type_free(&new);
  if(myid == 1)
    printf("\\nThe values are \%f and \%c\n", tst[1].value, tst[1].str);
  MPI_Comm_free(&mycomm);
  MPI_Finalize();
}
```

在这个 *C* 程序中, 我们定义了一个新的 *MPI* 数据类型, 而且还使用它进行消息传递. 这里我们使用了 *MPI* 中提供的伪数据类型 *(pseudo datatype)* MPI_UB, 在 *MPI* 中还有一个伪数据类型 MPI_LB. 这样定义的 *MPI* 的新数据类型 *+new+* 在调用函数 MPI_TYPE_EXTENT 之后返回值与机器关于双精度数是按 *4* 位还是 *8* 位对齐有关, 如果是按 *4* 位对齐, 返回值为 *12*, 否则为 *16*. 但是如果我们使用下面的函数 MPI_TYPE_SIZE, 它的返回值是 *9*.

MPI_TYPE_SIZE

C	int MPI_Type_size(MPI_Datatype datatype, int *size)
Fortran	MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, SIZE, IERROR

参数说明

IN	DATATYPE,	数据类型.
OUT	SIZE,	数据类型的字节数.

MPI_GET_ELEMENTS

C	`int MPI_Get_elements(MPI_Status *status,MPI_Datatype datatype,int *count)`
F	`MPI_GET_ELEMENTS( STATUS, DATATYPE, COUNT, IERROR )` `INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR`

参数说明

IN	STATUS,	接收消息的状态.
IN	DATATYPE,	数据类型.
OUT	COUNT,	*MPI* 基本数据类型的个数.

该函数和 MPI_GET_COUNT 的含意是不同的, MPI_GET_COUNT 是接收到的数据类型个数, 而这里给出的 MPI_GET_ELEMENTS 得到的是 *MPI* 基本数据类型的个数. 如果这里的数据类型是 *MPI* 基本数据类型, 如 *C* 语言中的 MPI_INT *(Fortran* 中 MPI_INTEGER*)* 等, 则 MPI_GET_COUNT 和 MPI_GET_ELEMENTS 等价. 我们使用下面的一段程序来说明它们的差别.

```
......
call mpi_type_contiguous(2, mpi_integer, type, ierr)
call mpi_type_commit(type, ierr)
if(myid .eq. 0 .and. p .gt. 1) then
  call mpi_send(ia, 3, mpi_integer, 1, 1, mycomm, ierr)
elseif(myid .eq. 1) then
  call mpi_recv(ia, 2, type, 0, 1, mycomm, status, ierr)
  call mpi_get_count(status, type, count1, ierr)
  call mpi_get_elements(status, type, count2, ierr)
endif
......
```

一般来说, 在我们自己写的程序中很少会出现发送与接收时数据类型是不同的, 因此在大部分实际应用中, 只需会使用 MPI_GET_COUNT 就不会遇到任何困难. 在这一节中, 我们已经介绍了如何构造自定义的 *MPI* 数据类型, 通常在定义这些数据类型时使用的是相对位置移动量, 如果使用绝对地址来构造新的 *MPI* 数据类型, 在使用这个新的数据类型进行消息传递时, *MPI* 定义了一个称之为 MPI_BOTTOM 的消息缓冲区常数, 以供 *MPI* 的发送与接收函数使用, 具体的使用方式将在以后的例子中给出.

§9.2 MPI 的数据打包与拆包

这里的数据打包与拆包和 *PVM (Parallel Virtual Machine)* 有相同的性质, 在 *MPI* 的绝大多数应用中使用用户自定义的 *MPI* 数据类型就可以完成许多数据一起发送与接

收的目的. 但是对于复杂的情况使用数据打包可能会收到好处.

MPI_PACK

```
C  int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype,
                void* outbuf,int outsize,int *position,MPI_Comm comm)
F  MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION,
           COMM, IERROR )
   <type> INBUF(*), OUTBUF(*)
   INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

参数说明

IN	INBUF,	准备打在包中的输入缓冲区.
IN	INCOUNT,	输入元素的个数.
IN	DATATYPE,	数据类型.
OUT	OUTBUF,	打包缓冲区.
IN	OUTSIZE,	用字节数定义的打包缓冲区的大小.
INOUT	POSITION,	打包缓冲区的当前位置.
IN	COMM,	打包消息的通信子.

MPI_UNPACK

```
C  int MPI_Unpack(void* inbuf,int insize,int *position,void* outbuf,
                  int outcount, MPI_Datatype datatype, MPI_Comm comm)
F  MPI_UNPACK( INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE,
               COMM, IERROR )
   <type> INBUF(*), OUTBUF(*)
   INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
```

参数说明

IN	INBUF,	拆包缓冲区.
IN	INSIZE,	用字节数定义的拆包缓冲区的大小.
INOUT	POSITION,	打包缓冲区的当前位置.
OUT	OUTBUF,	输出数据缓冲区.
IN	OUTCOUNT,	输出数据缓冲区元素的个数.
IN	DATATYPE,	数据类型.
IN	COMM,	通信子.

采用这种打包方式进行消息传递时, 数据类型一定要使用 MPI_PACKED. 为了能够知道一组数据在打包时需要多大的缓冲区, *MPI* 提供了如下的函数:

MPI_PACK_SIZE

C	int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
F	MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR) INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR

参数说明

IN	INCOUNT,	准备打包的元素个数.
IN	DATATYPE,	数据类型.
IN	COMM,	通信子.
OUT	SIZE,	按字节数定义的打包需要的缓冲区大小.

此函数返回的值 SIZE 要比原始数据本身所占有的字节数要大, 这是因为在包中还有一些其它的信息. 以下我们给出使用 MPI_PACK 和 MPI_UNPACK 的一个例子, 来结束关于用户自定义的数据类型部分.

例 *9.2.1* 假设我们要把一个整数数组和一个双精度数组从第 0 个进程传送到第 1 个进程, 尽管我们可以使用用户自定义数据类型来达到一次传送的目的, 但我们也可用打包和拆包的方式实现上述要求. 具体实现如下:

```
      program pack
      include 'mpif.h'
c
      integer maxbuf, len
      parameter (maxbuf = 200, len = 10)
      integer myid, p, mycomm, ierr, status(mpi_status_size, 2),
     &        ia(len), count1, count2, i, pos
      real*8 a(len)
      character buf(maxbuf)
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
c
      if(myid .eq. 0) then
        do 10 i=1, len
```

```
          ia(i) = i
          a(i) = dble(i)
10      continue
       endif
       call mpi_pack_size(len, mpi_integer, mycomm, count1, ierr)
       call mpi_pack_size(len, mpi_double_precision, mycomm, count2, i)
       print *, 'The pack size of 10 integer and real8 are:',
     &         count1, count2
       pos = 1
       if(myid .eq. 0 .and. p .gt. 1) then
         call mpi_pack(ia, len, mpi_integer, buf, maxbuf, pos,
     &                 mycomm, ierr)
         call mpi_pack(a, len, mpi_double_precision, buf, maxbuf, pos,
     &                 mycomm, ierr)
         call mpi_send(buf, pos-1, mpi_packed, 1, 1, mycomm, ierr)
       elseif(myid .eq. 1) then
         call mpi_recv(buf, maxbuf, mpi_packed, 0, 1, mycomm,
     &                 status, ierr)
         call mpi_unpack(buf, maxbuf, pos, ia, len, mpi_integer,
     &                   mycomm, ierr)
         call mpi_unpack(buf, maxbuf, pos, a, len, mpi_double_precision,
     &                   mycomm, ierr)
         print *, 'The received values are: ', a(1), a(2), a(3)
       endif
       call mpi_comm_free(mycomm, ierr)
       call mpi_finalize(ierr)
       stop
       end
```

第十章　MPI 聚合通信

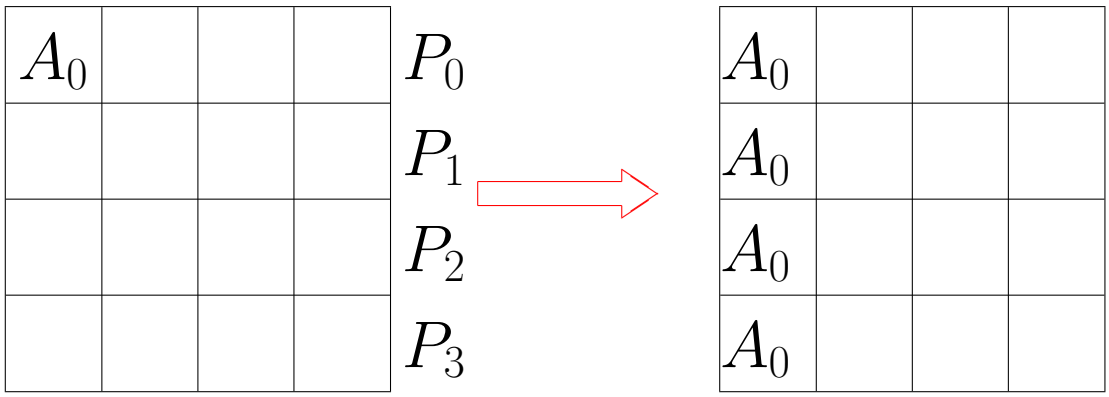


图 10.1: 广播函数的作用

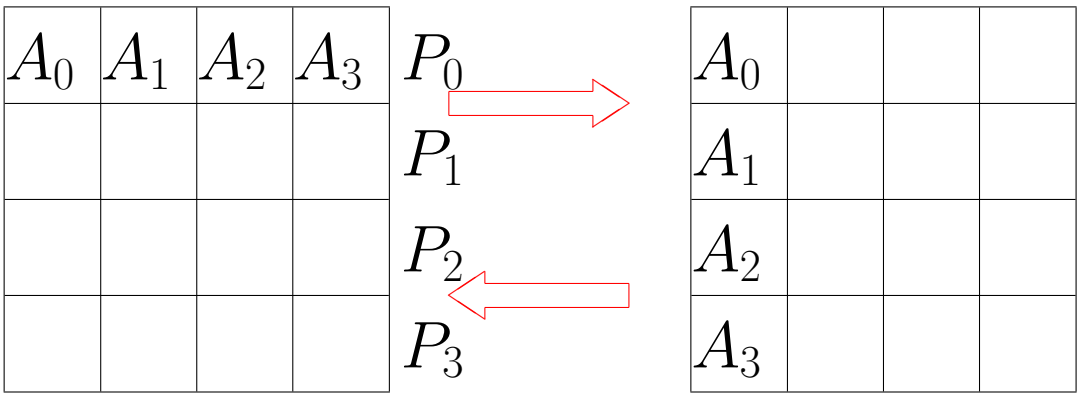


图 10.2: 散播与收集函数的作用

聚合通信 *(collective communication)* 是指多个进程 *(*通常大于 2*)* 之间的通信. 在这一节中, 介绍三方面的内容, 障碍同步 *(barrier synchronization)*、全局通信函数 *(global communication functions)* 和全局归约操作 *(global reduction operations)*. 以上的 4 个图从形式上给出了这几种聚合通讯所表达的内容.

§10.1　障碍同步

MPI_BARRIER

C	`int MPI_Barrier( MPI_Comm comm )`
F	`MPI_BARRIER( COMM, IERROR )`
	`INTEGER COMM, IERROR`

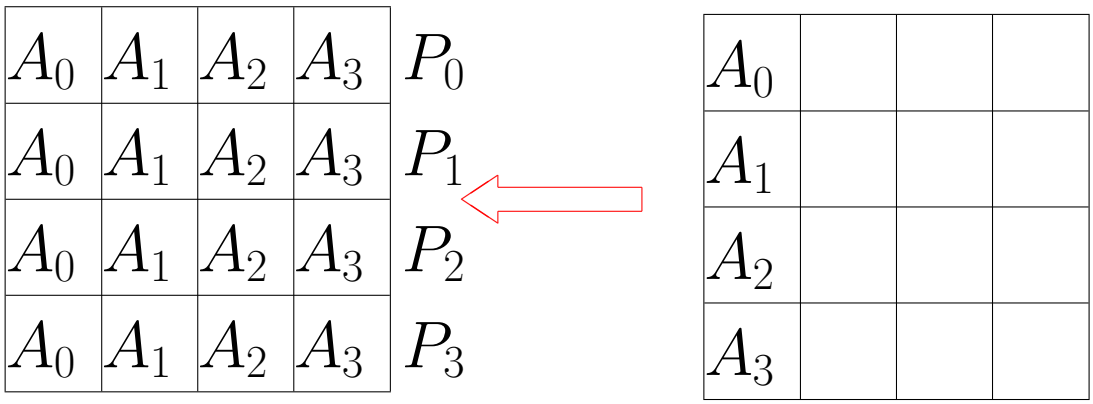


图 10.3: 每个进程都收集函数的作用

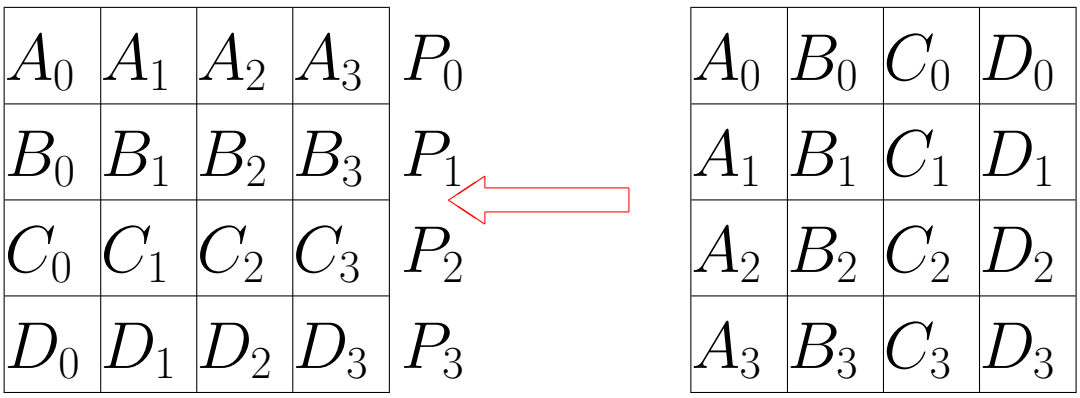


图 10.4: 全交换函数的作用

参数说明		
IN	COMM,	通信子.

这是 *MPI* 提供的唯一的一个同步函数, 当 COMM 中的所有进程都执行这个函数时才返回, 如果有一个进程没有执行此函数, 其余进程将处于等待状态. 在执行完这个函数之后, 所有进程将同时执行其后的任务.

§10.2 单点与多点通信函数

MPI_BCAST

C	int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
F	MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) <type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

参数说明		
INOUT	BUFFER,	缓冲区的首地址.
IN	COUNT,	缓冲区中元素的个数.
IN	DATATYPE,	缓冲区的数据类型.
IN	ROOT,	以它为源进行广播的进程编号.
IN	COMM,	通信子.

使用此函数时必须注意在 COMM 中的所有进程都执行此函数, 如果进程编号 ROOT, 则表示把此进程中的 BUFFER 的内容广播到 COMM 所有其它的进程中. 此函数是并行程序中经常出现的, 因此是个必须很好掌握的 *MPI* 通信函数.

MPI_GATHER

C	int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
F	MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

参数说明		
IN	SENDBUF,	发送缓冲区的首地址.
IN	SENDCOUNT,	要发送的元素的个数.
IN	SENDTYPE,	发送缓冲区的数据类型.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	RECVCOUNT,	接收每个进程中数据元素的个数.
IN	RECVTYPE,	接收缓冲区的数据类型.
IN	ROOT,	接收进程的编号.
IN	COMM,	通信子.

这是 *MPI* 提供的聚集 *(gather)* 函数, 其作用是把 COMM 中所有进程 *(*包括 ROOT*)* 的数据聚集到 ROOT 进程中, 并且自动按进程编号顺序存放在接收缓冲区 RECVBUF 中. 对于非 ROOT 进程, 忽略接收缓冲区 RECVBUF.

MPI_GATHERV

C
```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int *recvcounts, int *displs,
                MPI_Datatype recvtype, int root, MPI_Comm comm )
```

F
```
MPI_GATHERV( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
             DISPLS, RECVTYPE, ROOT, COMM, IERROR )
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE,
        ROOT, COMM, IERROR
```

参数说明		
IN	SENDBUF,	发送缓冲区的首地址.
IN	SENDCOUNT,	要发送的元素的个数.
IN	SENDTYPE,	发送缓冲区的数据类型.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	RECVCOUNTS,	接收每个进程数据元素的个数整数数组.
IN	DISPLS,	接收数据存放的位置数组.
IN	RECVTYPE,	接收缓冲区的数据类型.
IN	ROOT,	接收进程的编号.
IN	COMM,	通信子.

这个函数是 MPI_GATHER 的扩充, 它允许从不同的进程中接收不同长度的消息, 而且接收到的消息可以存放在接收缓冲区的不同位置, 因此使用要比 MPI_GATHER 更灵活. 下面我们给出一个例子来说明如何使用 *MPI* 的聚集函数:

例 10.2.1 假设在每个进程上都有一组数据，现要将它们收集到进程编号为 ROOT 的进程中，并按进程编号的顺序存放，则有如下的程序实现此功能.

```
      program gather
      include 'mpif.h'
c
      integer maxbuf, len, mp
      parameter (maxbuf = 200, len = 10, mp = 5)
      integer myid, p, mycomm, ierr, root, ia(len), iga(maxbuf), i,
     &        displs(mp), counts(mp)
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
c
      do 10 i=1, len
        ia(i) = i+myid*len
10    continue c
      do 20 i=1, p
        displs(i) = 20*i-20
        counts(i) = len
20    continue
      root = 0
c
      call mpi_gather(ia, len, mpi_integer, iga, len, mpi_integer,
     &                root, mycomm, ierr)
c
      if(myid .eq. root) then
        print *, 'The gather values are: ',
     &           iga(1), iga(len+1), iga(2*len+1)
      endif
c
      call mpi_gatherv(ia, len, mpi_integer, iga, counts, displs,
     &                 mpi_integer, root, mycomm, ierr)
c
      if(myid .eq. root) then
        print *, 'The gatherv values are: ',
     &           iga(2), iga(2*len+2), iga(4*len+2)
      endif
c
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
```

```
    stop
    end
```

这个程序的输出值应分别为: *1, 11, 21* 和 *2, 12, 22*. MPI_GATHER 接收到的数据在接收缓冲区中是连续存放的, 而 MPI_GATHERV 接收到的数据在接收缓冲区中是不连续存放的, 在我们这里给出的例子中, 由于 COUNTS 的所有分量是相同的, 可以看成是每隔步长为 *20* 的位置存放一个新接收的数据.

MPI_SCATTER

C
```
int MPI_Scatter( void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm )
```

F
```
MPI_SCATTER( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, IERROR )
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

参数说明

IN	SENDBUF,	发送缓冲区的首地址.
IN	SENDCOUNT,	发送到每个进程中的元素的个数.
IN	SENDTYPE,	发送缓冲区的数据类型.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	RECVCOUNT,	接收数据元素的个数.
IN	RECVTYPE,	接收缓冲区的数据类型.
IN	ROOT,	发送进程的编号.
IN	COMM,	通信子.

这是 *MPI* 提供的散布 *(scatter)* 函数, 此函数是 MPI_GATHER 的逆操作.

MPI_SCATTERV

C
```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
                 MPI_Datatype sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm )
```

F
```
MPI_SCATTERV( SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
              RECVTYPE, ROOT, COMM, IERROR )
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
        COMM, IERROR
```

参数说明

IN	SENDBUF,	发送缓冲区的首地址.
IN	SENDCOUNTS,	发送到每个进程中的元素的个数数组.
IN	DISPLS,	发送到每个进程中的数据起始位移.
IN	SENDTYPE,	发送缓冲区的数据类型.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	RECVCOUNT,	接收数据元素的个数.
IN	RECVTYPE,	接收缓冲区的数据类型.
IN	ROOT,	发送进程的编号.
IN	COMM,	通信子.

§10.3 多点与多点通信函数

MPI_ALLGATHER

C
```
int MPI_Allgather( void* sendbuf, int sendcount, MPI_Datatype sendtype,
                   void* recvbuf, int recvcount, MPI_Datatype recvtype,
                   MPI_Comm comm )
```

F
```
MPI_ALLGATHER( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
               RECVTYPE, COMM, IERROR )
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

参数说明

IN	SENDBUF,	发送缓冲区的首地址.
IN	SENDCOUNT,	要发送的元素的个数.
IN	SENDTYPE,	发送缓冲区的数据类型.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	RECVCOUNT,	接收每个进程中数据元素的个数.
IN	RECVTYPE,	接收缓冲区的数据类型.
IN	COMM,	通信子.

MPI_ALLGATHERV

C
```
int MPI_Allgatherv( void* sendbuf, int sendcount, MPI_Datatype sendtype,
                    void* recvbuf, int recvcount, MPI_Datatype recvtype,
                    MPI_Comm comm )
```

F
```
MPI_ALLGATHERV( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                RECVTYPE, COMM, IERROR )
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

参数说明

IN	SENDBUF,	发送缓冲区的首地址.
IN	SENDCOUNT,	要发送的元素的个数.
IN	SENDTYPE,	发送缓冲区的数据类型.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	RECVCOUNT,	接收每个进程中数据元素的个数.
IN	RECVTYPE,	接收缓冲区的数据类型.
IN	COMM,	通信子.

此二函数是每个进程都进行数据聚集操作, 而 MPI_GATHER 和 MPI_GATHERV 只是其中的一个进程进行数据的聚集. 这里每个进程都发送消息也都接收消息.

MPI_ALLTOALL

C
```
int MPI_Alltoall( void* sendbuf, int sendcount, MPI_Datatype sendtype,
                  void* recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm )
```

F
```
MPI_ALLTOALL( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
              RECVTYPE, COMM, IERROR )
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

参数说明

IN	SENDBUF,	发送缓冲区的首地址.
IN	SENDCOUNT,	要发送的元素的个数.
IN	SENDTYPE,	发送缓冲区的数据类型.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	RECVCOUNT,	接收每个进程中数据元素的个数.
IN	RECVTYPE,	接收缓冲区的数据类型.
IN	COMM,	通信子.

MPI_ALLTOALLV

C
```
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
                  MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
                  int *rdispls, MPI_Datatype recvtype, MPI_Comm comm )
```

F
```
MPI_ALLTOALLV( SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
               RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR )
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
        RECVTYPE, COMM, IERROR
```

参数说明

IN	SENDBUF,	发送缓冲区的首地址.
IN	SENDCOUNTS,	发送到每个进程中的元素的个数数组.
IN	SDISPLS,	发送到每个进程中的数据起始位移.
IN	SENDTYPE,	发送缓冲区的数据类型.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	RECVCOUNTS,	接收数据元素的个数数组.
IN	RDISPLS,	接收进程中存放数据的起始位移.
IN	RECVTYPE,	接收缓冲区的数据类型.
IN	COMM,	通信子.

这二个函数等价于在所有进程中的数据进行散布. 为了便于理解和正确使用此函数, 我们还是给出一个具体例子.

```
      program allall
      include 'mpif.h'
      integer maxbuf, len, mp
      parameter (maxbuf = 200, len = 10, mp = 5)
```

```
      integer myid, p, mycomm, ierr, igb(maxbuf), iga(maxbuf), i,
     &        sdispls(mp), scounts(mp), rdispls(mp), rcounts(mp)
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
      do 10 i=1, maxbuf
        iga(i) = i+maxbuf*myid
10    continue
      do 20 i=1, p
        sdispls(i) = 20*i-20
        rdispls(i) = 15*i-15
        scounts(i) = len
        rcounts(i) = len
20    continue
      call mpi_alltoall(iga, len, mpi_integer, igb, len, mpi_integer,
     &                  mycomm, ierr)
      print *, 'The alltoall values are: ',
     &         igb(1), igb(len+1), igb(2*len+1), ' on Proc. ', myid
      call mpi_alltoallv(iga, scounts, sdispls, mpi_integer, igb,
     &                   rcounts, rdispls, mpi_integer, mycomm, ierr)
      print *, 'The alltoallv values are: ',
     &         igb(1), igb(16), igb(31), ' on Proc. ', myid
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end
```

第十一章　MPI 全局归约操作

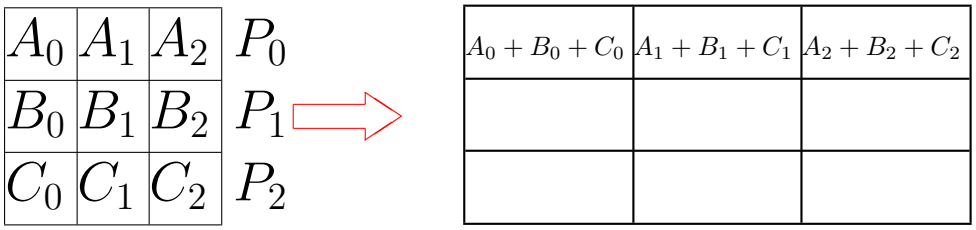


图 11.1: 一个进程进行归约操作

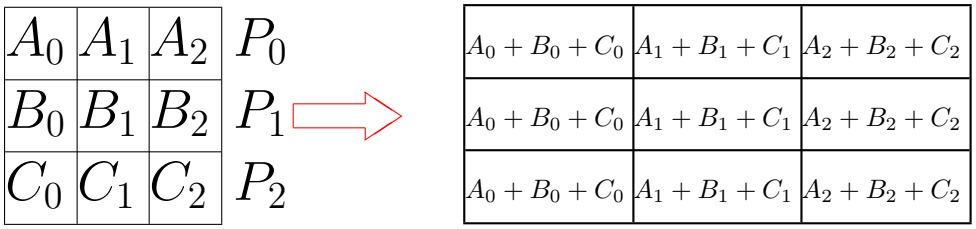


图 11.2: 所有进程进行归约操作

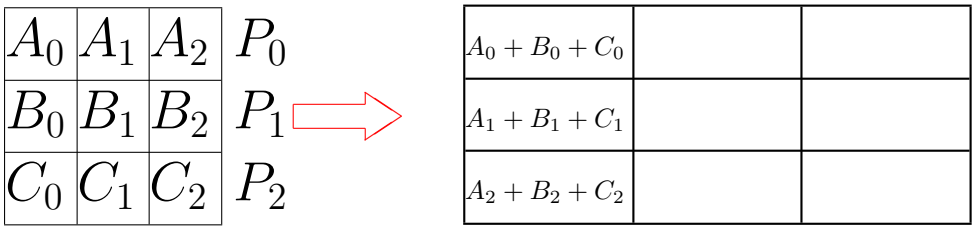


图 11.3: 进行归约分发操作

上述 4 个图是对归约操作的一形式描述.

MPI_REDUCE

C	`int MPI_Reduce(void* sendbuf, void* recvbuf, int count,` `MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm comm)`

F	`MPI_REDUCE( SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR )`
	`<type> SENDBUF(*), RECVBUF(*)`
	`INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR`

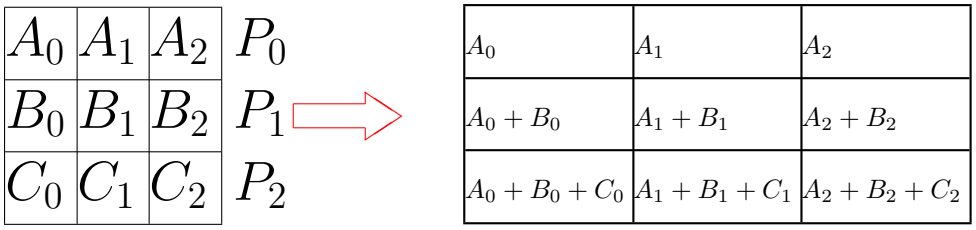


图 11.4: 前缀归约操作

参数说明		
IN	SENDBUF,	发送缓冲区的首地址.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	COUNT,	缓冲区中元素的个数.
IN	DATATYPE,	缓冲区的数据类型.
IN	OP,	何种归约操作.
IN	ROOT,	接收进程编号.
IN	COMM,	通信子.

这个函数是按分量进行归约操作的, 其运算由 OP 来确定, 最后的结果放在 ROOT 进程中, 其它进程中的 RECVBUF 不起作用. 在 *MPI* 中规定了一些允许的操作如下:

操作名	意义
MPI_MAX	求最大
MPI_MIN	求最小
MPI_SUM	求和
MPI_PROD	求积
MPI_LAND	逻辑与
MPI_BAND	按位与
MPI_LOR	逻辑或
MPI_BOR	按位或
MPI_LXOR	逻辑与或
MPI_BXOR	按位与或
MPI_MAXLOC	求最大和位置
MPI_MINLOC	求最小和位置

这些运算是有数据类型要求的, 首先对数据类型进行分类:

C integer:	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG
Fortran integer:	MPI_INTEGER
Floating point:	MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE
Logical:	MPI_LOGICAL
Complex:	MPI_COMPLEX
Byte :	MPI_BYTE

现在对每种操作允许的数据类型规定如下:

OP	允许的数据类型
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical
MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte

关于 MPI_MAXLOC 和 MPI_MINLOC, *MPI* 对 *Fortran* 程序和 *C* 程序使用的复合数据类型规定如下:

Fortran 程序

复合数据类型	类型描述
MPI_2REAL	*pair of* REALs
MPI_2DOUBLE_PRECISION	*pair of* DOUBLE PRECISIONs
MPI_2INTEGER	*pair of* INTEGERs

C 程序

复合数据类型	类型描述
MPI_FLOAT_INT	float *and* int
MPI_DOUBLE_INT	double *and* int
MPI_LONG_INT	long *and* int
MPI_SHORT_INT	short *and* int
MPI_LONG_DOUBLE_INT	long double *and* int
MPI_2INT	*pair of* ints

例 *11.0.1* 假设在每个处理机中有一个数, 我们要在这些数中找一个最大的, 并确定这个最大数在哪个处理机中, 则可用如下的程序:

```
      program reduce
      include 'mpif.h'
```

```
c
      integer myid, p, mycomm, ierr, m, n, root, pair(2), answer(2)
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
      root = 0
      m = myid
      call mpi_reduce(m, n, 1, mpi_integer, mpi_max, root, mycomm, ierr)
c
      if(myid .eq. root) print *, 'The maxmum value is ', n
      pair(1) = mod(myid + 1, p)
      pair(2) = myid
      call mpi_reduce(pair, answer, 1, mpi_2integer, mpi_maxloc, root,
     &                mycomm, ierr)
      if(myid .eq. root) print *, 'The maxmum value is ', answer(1),
     &                            ' on process ', answer(2)
c
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end
```

MPI_ALLREDUCE

C	`int MPI_Allreduce( void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )`

F	`MPI_ALLREDUCE( SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR )`
	`<type> SENDBUF(*), RECVBUF(*)`
	`INTEGER COUNT, DATATYPE, OP, COMM, IERROR`

参数说明

IN	SENDBUF,	发送缓冲区的首地址.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	COUNT,	缓冲区中元素的个数.
IN	DATATYPE,	缓冲区的数据类型.
IN	OP,	何种归约操作.
IN	COMM,	通信子.

此函数和 MPI_REDUCE 的意思是相同的, 只是最后结果在所有的进程中.

MPI_REDUCE_SCATTER

C	int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype,MPI_Op op,MPI_Comm comm)
F	MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

参数说明

IN	SENDBUF,	发送缓冲区的首地址.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	RECVCOUNTS,	元素个数的数组.
IN	DATATYPE,	缓冲区的数据类型.
IN	OP,	何种归约操作.
IN	COMM,	通信子.

此函数相当于先做 MPI_REDUCE, 然后在做 MPI_SCATTER.

例 *11.0.2* 设我们使用 p 个进程计算 $c = Ab$, 其中 A 是 m 阶矩阵, b 是 m- 维向量. 假设 $m = np$ (如不能整除把剩余部分放在最后一个进程中), 在每个进程中存放 A 的 n 列和对应的 b 的 n 个分量, 现要在每个进程中得到与 b 对应的 c 或是 c 的全部, 则我们可以用下面的程序实现:

```
      program redsct
      include 'mpif.h'
c
      integer lda, cols, maxnp
      parameter (lda = 100, cols = 100, maxnp = 5)
      integer myid, p, mycomm, ierr, m, n, counts(maxnp)
      real a(lda, cols), b(cols), c(lda), sum(lda)
      logical sct
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
      if (p .gt. maxnp) goto 999
      m = 100
      sct = .false.
c      sct = .true.
```

```
      call initab(m, myid, p, n, a, lda, b, sum)
      call locmv(m, n, a, lda, b, sum)
c if sct = true, call reduce_scatter, otherwize call allreduce
      if(sct) then
        call mpi_allgather(n, 1, mpi_integer, counts, 1,
     &                     mpi_integer, mycomm, ierr)
        call mpi_reduce_scatter(sum, c, counts, mpi_real, mpi_sum,
     &                          mycomm, ierr)
      else
        call mpi_allreduce(sum, c, m, mpi_real, mpi_sum, mycomm, ierr)
      endif
c
      print *, 'The values of c are ', c(1), c(2)
c 999   call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end

      subroutine initab(m, myid, np, k, a, lda, b, sum)
      integer m, myid, np, k, lda, i, j, l
      real a(lda, *), b(*), sum(*)
      do 10 i=1, m
10      sum(i) = 0.0
      k = m/np
      l = myid * k
      if(myid .eq. np-1) k = m-l
      do 20 i=1, k
20      b(i) = 1.0
      do 40 j=1, k
        do 40 i=1, m
40        a(i, j) = real(i+l+j)
      return
      end

      subroutine locmv(m, n, a, lda, b, c)
      integer m, n, lda, i, j
      real a(lda, *), b(*), c(*)
      do 20 j=1, n
        do 20 i=1, m
20        c(i) = c(i)+a(i, j)*b(j)
      return
      end
```

MPI_SCAN

C	int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
F	MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT(*), DATATYPE, OP, COMM, IERROR

参数说明

IN	SENDBUF,	发送缓冲区的首地址.
OUT	RECVBUF,	接收缓冲区的首地址.
IN	COUNT,	元素个数.
IN	DATATYPE,	缓冲区的数据类型.
IN	OP,	何种归约操作.
IN	COMM,	通信子.

此函数是 MPI_REDUCE 在不同进程中的重复使用, 进程 i 中得到的是在由编号为 $\{0,1,\ldots,i\}$ 的进程构成的进程组中使用 MPI_REDUCE 结果, 是一个并不常用的函数.

在 *MPI* 中提供了用户自定义的操作函数以满足不同需要, 它由如下的几个函数来实现:

MPI_OP_CREATE

C	int MPI_Op_create(MPI_User_function *function,int commute,MPI_Op *op)
F	MPI_OP_CREATE(FUNCTION, COMMUTE, OP, IERROR) EXTERNAL FUNCTION LOGICAL COMMUTE INTEGER OP, IERROR

参数说明

IN	FUNCTION,	用户定义的函数.
IN	COMMUTE,	等于 TRUE 是可交换的, 否则只是可结合的.
OUT	OP,	新的归约操作.

此函数定义了一个新的归约操作 OP.

MPI_OP_FREE

C	int MPI_Op_free(MPI_Op *op)
F	MPI_OP_FREE(OP, IERROR) INTEGER OP, IERROR

参数说明	
IN	OP, 归约操作.

此函数释放归约操作 OP. 在 *MPI* 中, 用户自定义操作的函数是有严格要求的, 其形式如下:

C	void user_function(void* invec, void* inoutvec, int *len, MPI_Datatype *datatype)
F	FUNCTION USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE) <datatype> INVEC(LEN), INOUTVEC(LEN) INTEGER LEN, DATATYPE

下面的例子是用自定义的求和函数来实现全局操作的:

```
      program userdef
      include 'mpif.h'
      integer len
      parameter (len = 100)
      integer myid, p, mycomm, ierr, m, root, myop
      real x(len), y(len)
      external userfunc
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
      m = 100
      root = 0
      call initx(m, myid, x)
      call mpi_op_create(userfunc, .true., myop, ierr)
      call mpi_reduce(x, y, m, mpi_real, myop, root, mycomm, ierr)
      call mpi_op_free(myop, ierr)
      if(myid .eq. root)
     &   print *, 'The values of answer are ', y(1), y(2), y(3)
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end

      subroutine initx(m, myid, x)
      integer m, myid, i
      real x(*)
      do 10 i=1, m
10      x(i) = real(i+myid)
```

```
      return
      end

      subroutine userfunc(x, y, m, mpi_real)
      integer m
      real x(*), y(*)
      do 10 i=1, m
10      y(i) = y(i)+x(i)
      return
      end
```

第十二章　HPL 程序实例剖析

参考文献

[1] 陈国良,《并行计算 – 结构 · 算法 · 编程》，高等教育出版社，2003。

[2] 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》，科学出版社，1996。

[3] http://www.top500.org/。

[4] 冯圣中，并行计算基础知识，手稿，2004。

[5] 莫则尧，袁国兴,《消息传递并行编程环境 MPI》，科学出版社，2001。

[6] J. M. Ortega, Introduction to Parallel and Vector Solution of Linear Systems, Plenum Press, 1988.

[7] J. Dongarra, I. Duff, D. Soresen and H. van der Vorst, Solving Linear Systems on Vector and Shared Memory Computers, SIAM, 1991.

[8] G. Golub and C. van Loan, Matrix Computation, The Johns Hopkins University Press, 1983. (中译本: 矩阵计算, 廉庆荣、邓健新、刘秀兰译, 大连理工大学出版社, 1988 年)

[9] 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 年第 17 卷第 2 期.

[10] G. Y. Li and T. F. Coleman, A Parallel Triangular Solver for a Hypercube Multiprocessor, TR 86–787, Cornell University, 1986.

[11] 迟学斌, Transputer 上 Cholesky 分解的并行实现, 计算数学,1993 年第 15 卷第 3 期.

[12] J. M. Delosme and I. C. F. Ipsen, Positive Definite Systems with Hyperbolic Rotations, Linear Algebra Appl., 77 (1986), 75–111.

[13] D. H. Lawrie and A. H. Sameh, The Computation and Communication Complexity of a Parallel Banded System Solver, ACM Trans. Math. Soft., 10 (1984), 185–195.

[14] 迟学斌, Transputer 上线性系统的并行求解, 中国计算机用户,1991 年第 10 期.

[15] 陈景良, 并行数值方法, 清华大学出版社,1983 年.

[16] 张宝琳、袁国兴、刘兴平、陈劲, 偏微分方程并行有限差分方法, 科学出版社,1994 年.

[17] D. Chazan and W. Miranker, Chaotic Relazation, J. Lin. Alg. Appl., 2 (1969), 199–222.

[18] G.Baudet, Asynchronous Iterative Methods for Multiprocessors, J. ACM, 25 (1978), 226–244.

[19] J. M. Ortega and W. C. Rheinboldt, Iterative Solution of Nonlinear Equations in Several Variables, Academic Press, 1970.

[20] 迟学斌, 线性方程组的异步迭代法, 计算数学,1992 年第 14 卷第 3 期.

附录一　并行程序开发工具与高性能程序库

§A.1　BLAS、LAPACK、ScaLAPACK

§A.2　FFTW

§A.3　PETSc

附录二　MPI 函数 reference

索引: *MPI* 函数，重要概念、名词

索 引