

XIAO XI CHUAN DI BING XING
BIAN CHENG HUAN JING

消息传递 并行编程环境

MPI

莫则尧 袁国兴 编著

11.11



科学出版社
Science Press

消息传递 并行编程环境

MPI

(TN-0350.0101)

责任编辑：陆新民 林 鹏 封面设计：王 浩

ISBN 7-03-009805-6



9 787030 098054 >

ISBN 7-03-009805-6 / TN · 350

定 价：26.00 元

消息传递并行编程环境 MPI

莫则尧 袁国兴 编著

科学出版社
2001

内 容 提 要

本书围绕消息传递并行程序设计,全面介绍了当前最流行的消息传递并行编程环境 MPI 1.0 版本,以及 MPI 2.0 版本中已在各类并行机上被普遍实现的部分,并简单介绍了 MPI 2.0 版本的许多新特征。

本书面向的读者是非计算机专业毕业的科学与工程计算科研人员,注重用简明易懂的语言,采用函数说明和程序实例相结合的方式来组织内容,并尽量将 MPI 的许多抽象概念具体化,使读者容易理解。由于消息传递并行程序设计的主要应用领域是科学与工程计算,因此,本书注重用 Fortran 语言结合具体应用实例来介绍,但对 C 语言也进行了详细的讨论。

从消息传递并行程序设计角度出发,本书吸取了众多国外 MPI 资料的内容,全面介绍了目前应用最广泛的消息传递并行编程环境 MPI。内容丰富,取材新颖,覆盖面广,可作为科学与工程计算部门科研人员设计消息传递并行程序的参考资料,也可作为高等院校计算数学、计算物理、计算化学、计算流体力学、计算材料、计算气象、计算生物、计算机及其相关专业的本科高年级学生和研究生的并行程序设计教学用书。

图书在版编目(CIP)数据

消息传递并行编程环境 MPI / 莫则尧, 袁国兴编著.

北京:科学出版社, 2001

ISBN 7-03-009805-6

I . 消… II . ①莫… ②袁… III . 并行程序语言~程序设计环境, MPI
IV . TP. 312

中国版本图书馆 CIP 数据核字(2001)第 066689 号

科 学 出 版 社 出 版

北京东黄城根北街 16 号

邮政编码: 100717

<http://www.sciencep.com>

深 海 印 刷 厂 印 刷

科学出版社发行 各地新华书店经销

*

2001 年 11 月第 一 版 开本: 787 × 1092 1/16

2001 年 11 月第 次印刷 印张: 12 3/4

印数: 1—2 500 字数: 290 000

定 价: 26.00 元

(如有印装质量问题, 我社负责调换(北燕))

前　　言

近年来,由于核禁试后加速战略计算创新(ASCI)计划的推动,世界上和我国高性能并行计算机取得了长足的发展,每秒数千亿次以上的高性能并行机对一般科研部门已经不再是一种奢望,以前许多无法求解和研究的问题现在已经成为可能。

随着并行机的发展,高性能并行计算已经在我国科学与工程计算部门和高等院校得到很好的推广,越来越多的青年科技工作者开始学习并行计算的知识。并行计算是一门交叉学科,涉及计算机、计算数学、计算物理、计算化学、计算流体力学、计算材料、计算气象、计算生物等学科,以及其他非数值应用领域,是高性能并行机成功应用的前提条件。

并行计算包含的内容较广,但大体可分为并行机体系结构、并行计算支撑环境、并行算法设计、并行程序设计和并行计算性能评价等五个方面。这五个部分相辅相存,缺一不可。其中,并行程序设计是在具体并行机上实现并行算法的关键,其复杂度远远超过了串行程序设计。一个好的并行算法,如果没有一个好的并行程序设计来具体实现,就无法得到推广应用。从 20 世纪 90 年代开始,国内并行计算专家已经出版了许多与并行计算相关的书籍,但是他们大多注重于并行机体系结构和并行算法设计,而很少进行并行程序设计的介绍。本书是对它们的一个很好补充。

目前,高性能并行机主要可分为对称多处理(SMP)共享存储并行机、分布共享存储(DSM)并行机、大规模并行机(MPP)和微机机群等四类。在这些并行机上,并行程序设计平台主要可分为消息传递、共享存储和数据并行三类,其中消息传递具有很好的可移植性,它能被所有这些类型的并行机所支持,而共享存储只能在 SMP 和 DSM 并行机中使用,数据并行只能在 SMP, DSM 和 MPP 并行机上使用。

消息传递并行编程环境 MPI(Message Passing Interface)是目前国际上最流行、可移植性和可扩展性很好的并行程序设计平台,并被当前流行的所有高性能并行机所支持。目前,国外关于 MPI 的文献和资料非常多,但这些资料大多是面向计算机专业毕业的科研人员编写的,不太适合非计算机专业毕业的科学与工程计算科研人员。在国内,全面介绍 MPI 的中文文献和资料却几乎是一个空白,这与我国对高性能并行计算的需求是不相匹配的。因此,在实际工作中,我们发现有必要针对我国非计算机专业毕业的科学与工程计算科研人员的特点,编写一本全面的、简明易懂的 MPI 并行编程参考书,它将有利于推动高性能并行计算在我国科学与工程计算部门的具体应用。

消息传递并行编程环境 MPI 1.0 版于 1994 年 6 月由全球工业、政府和科研部门联合推出,至今已经发展到 MPI 2.0 版。但是,目前流行的并行机一般只能支持 MPI 1.0 版本的全部和 MPI 2.0 版本的部分。本书面向非计算机专业毕业的科学与工程计算科研人员,以 MPI 1.0 版为基准,加上目前 MPI 2.0 版本中已经在各类并行机上被普遍实现的 MPI 并行 I/O 函数,采用函数说明与应用实例相结合的方法,由浅入深地介绍 MPI 并行编程的各个方面,最后还简单介绍了 MPI 2.0 版本的许多新的特征。同时,考虑到 Fortran 是目前在科学与工程计算部门应用最为广泛的编程语言,本书着重于用 Fortran

语言来介绍 MPI 的应用同时,对 C 语言,本书也进行了详细的讨论。

本书共分为十三章。其中,第一章介绍消息传递并行程序设计的基础知识,包括并行机体系统结构和操作系统概念,第二章介绍 MPI 并行编程的一些预备知识,第三章至第九章分别介绍点对点通信、自定义数据类型、聚合通信、进程通信器、进程拓扑结构、并行 I/O 和 MPI 系统环境管理,第十章讨论 MPI 与共享存储并行程序设计平台 OpenMP 的混合编程技术,第十一章给出一个求解二维 Laplace 方程的应用程序实例,它综合利用了前面各章介绍的 MPI 函数,第十二章介绍 MPI 2.0 标准的许多新特征,第十三章简单介绍 MPI 的应用现状和发展。此外,本书最后还附有 5 个附录,分别介绍 MPI 程序的编译和运行、MPI 系统的安装、MPI 网站、MPI 函数索引和 MPI 术语中-英文对照及索引。

中国科学院数学与系统科学研究所张林波研究员审阅了书稿,并提出了许多宝贵建议,在此表示衷心感谢。本书编写过程中,北京大学湍流国家重点实验室蔡庆东、北京应用物理与计算数学研究所刘兴平等老师提出了许多有益的建议,在此表示衷心感谢。

消息传递并行程序设计涉及的内容广泛,加上作者学识有限,写作时间仓促,书中错误和片面之处在所难免,恳请读者不吝批评指正。

莫则尧 袁国兴

2001 年 5 月

目 录

前言

第一章 消息传递并行程序设计基础	(1)
1.1 并行计算环境	(1)
1.1.1 并行机的发展动力	(1)
1.1.2 并行机体系结构	(2)
1.1.3 并行机软件环境	(6)
1.2 进程与进程间通信	(7)
1.2.1 进程	(7)
1.2.2 进程间通信	(7)
1.3 线程	(8)
1.4 并行编程环境	(9)
1.5 消息传递并行机模型	(10)
1.6 标准消息传递界面 MPI	(11)
第二章 MPI 预备知识	(12)
2.1 MPI 程序示例	(12)
2.2 MPI 并行程序设计流程图	(14)
2.3 MPI 并行编程模式	(16)
2.4 MPI 函数的分类	(17)
2.5 其他的预备知识	(18)
第三章 点对点通信	(19)
3.1 标准模式阻塞通信	(19)
3.1.1 消息发送/接收函数	(19)
3.1.2 一个示例:并行矩阵乘	(24)
3.1.3 消息发收函数	(25)
3.1.4 消息长度查询函数	(27)
3.1.5 空进程	(28)
3.2 数据类型的匹配与转换	(29)
3.2.1 数据类型匹配规则	(29)
3.2.2 数据转换	(30)
3.3 标准模式非阻塞通信	(30)
3.3.1 非阻塞消息发送/接收函数	(31)

3.3.2 通信请求完成函数	(33)
3.3.3 消息查询函数	(37)
3.4 有限缓存区资源对消息传递的影响	(39)
3.5 持久通信请求	(40)
3.6 通信请求的释放与取消	(43)
3.7 其他通信模式	(45)
3.7.1 阻塞式消息发送函数	(45)
3.7.2 非阻塞式消息发送函数	(49)
3.7.3 持久通信函数	(50)
第四章 自定义数据类型与数据封装	(52)
4.1 自定义数据类型	(52)
4.2 自定义数据类型的创建	(55)
4.3 自定义数据类型的应用	(63)
4.3.1 自定义数据类型的提交与释放	(63)
4.3.2 消息参数的进一步理解	(63)
4.3.3 数据类型匹配规则的进一步理解	(64)
4.3.4 数据类型查询函数	(64)
4.4 数据的封装与拆卸	(64)
第五章 聚合通信	(68)
5.1 同步通信函数	(70)
5.2 全局通信函数	(71)
5.2.1 消息广播函数	(71)
5.2.2 消息收集函数	(71)
5.2.3 基于向量的消息收集函数	(73)
5.2.4 消息分发函数	(74)
5.2.5 基于向量的消息分发函数	(76)
5.2.6 消息全收集函数	(77)
5.2.7 基于向量的消息全收集函数	(78)
5.2.8 消息全交换函数	(79)
5.2.9 基于向量的消息全交换函数	(80)
5.3 全局归约函数	(82)
5.3.1 归约函数	(83)
5.3.2 归约操作 MPI_MAXLOC 和 MPI_MINLOC	(85)
5.3.3 全归约函数	(86)
5.3.4 归约分发函数	(87)
5.3.5 并行前缀归约函数	(89)
5.3.6 自定义归约操作	(89)

第六章 进程通信器	(92)
6.1 进程组管理	(93)
6.1.1 进程组创建	(93)
6.1.2 进程组访问与比较	(98)
6.1.3 进程组释放	(100)
6.2 通信器管理	(100)
6.2.1 通信器创建	(100)
6.2.2 通信器访问与比较	(105)
6.2.3 通信器释放	(107)
6.2.4 通信器附加属性	(107)
6.3 域间通信器	(107)
6.3.1 域间通信器的创建与释放	(108)
6.3.2 域间通信器访问	(110)
第七章 进程拓扑结构	(111)
7.1 Cartesian 拓扑结构	(111)
7.1.1 Cartesian 拓扑结构创建	(111)
7.1.2 Cartesian 拓扑结构辅助函数	(113)
7.1.3 Cartesian 拓扑结构查询	(114)
7.1.4 Cartesian 拓扑结构分解	(117)
7.1.5 Cartesian 拓扑结构映射	(118)
7.2 图拓扑结构	(119)
7.2.1 图拓扑结构创建	(119)
7.2.2 图拓扑结构查询	(120)
7.2.3 图拓扑结构映射	(122)
7.3 拓扑结构类型查询	(122)
第八章 并行 I/O	(123)
8.1 串行 I/O	(123)
8.2 非 MPI 并行 I/O	(125)
8.3 MPI 并行 I/O: 并行访问不同的文件	(126)
8.4 MPI 并行 I/O: 并行访问同一个文件	(129)
8.4.1 简单的并行 I/O	(130)
8.4.2 显式偏移并行 I/O	(131)
8.4.3 非连续访问并行 I/O	(133)
8.4.4 聚合并行 I/O	(136)
8.4.5 分布存储数组的并行 I/O	(138)
8.5 非阻塞并行 I/O 与分裂聚合并行 I/O	(145)

8.6 共享文件指针	(147)
8.7 提示信息	(148)
8.8 并行 I/O 的数据一致性	(151)
8.9 文件的可移植性	(153)
第九章 MPI 系统环境管理	(155)
9.1 进入和退出 MPI 系统	(155)
9.2 获取墙上时间	(156)
9.3 MPI 系统常数的查询	(157)
9.4 MPI 异常及其处理	(158)
9.4.1 错误处理程序	(158)
9.4.2 信息码	(160)
第十章 MPI 与 OpenMP 的混合编程	(161)
10.1 MPI 进程的多线程执行	(161)
10.2 MPI 与 OpenMP 的混合编程	(162)
10.3 并行矩阵乘混合编程示例	(163)
第十一章 MPI 程序示例	(165)
11.1 并行算法设计	(165)
11.2 MPI 并行程序设计	(165)
11.3 MPI 并行程序的改进	(171)
第十二章 MPI 2.0 的新特征	(176)
12.1 单边通信	(176)
12.2 动态进程管理	(181)
第十三章 MPI 的现状与发展	(182)
附录	(183)
附录 A MPI 程序的编译和运行	(183)
附录 B MPICH 的安装	(184)
附录 C MPI 网站	(190)
附录 D MPI 函数索引	(191)
附录 E 术语中英文对照及索引	(194)

第一章 消息传递并行程序设计基础

本章主要讨论当前流行的并行程序设计平台所依赖的并行计算机硬件和软件环境,以及消息传递并行程序设计的基础理论知识。

1.1 并行计算环境

并行计算环境是并行算法设计和并行程序设计的基础,是并行计算发展的前提条件,它可以分为并行计算机硬件环境和软件环境两类。本节简要介绍当前流行的高性能并行计算机体系结构及其软件支持环境。

1.1.1 并行机的发展动力

自从 1972 年世界上第一台并行计算机 ILLIAC-IV 诞生以来,并行机已经经历了近 30 年的发展,它们对推动计算机技术的飞速发展和高性能计算在各个领域的应用做出了重要贡献。其中,应用问题的实际需求和微电子技术的革新是推动并行机不断发展的两个主要动力。

人类对计算机性能的需求是无止境的,在诸如预测模型的构造、工程设计和自动化、能源勘探、医学、军事、国家安全以及基础理论研究等领域中,都对计算提出了极具挑战性的需求。进入 20 世纪 90 年代以来,以美国为主的西方发达国家以应用需求为背景,提出了一系列的发展规划,极大地推动了高性能并行机的发展。其中,两个最主要的规划是美国于 1993 年提出的高性能计算与通信(HPCC: High Performance Computing and Communication)计划和 1996 年提出的加速战略计算创新(ASCI: Accelerated Strategic Computing Initiative)计划。

为了保持在高性能计算与通信领域中的世界领先地位,1993 年美国科学、工程、技术联邦协调理事会向国会提交了题为“重大挑战性项目:高性能计算与通信(HPCC)”的报告,简称 HPCC 计划,目的是研制能提供 3T 性能目标(1Tflops 计算能力、 1TB 内存容量和 1TB/s 的 I/O 带宽, $1\text{T} = 10^{12}$)的高性能并行机,解决科学与工程计算中的重大挑战性课题,其中包括新药设计、磁记录技术、高速民航、催化作用、燃料燃烧、海洋建模、臭氧耗损、数字解析、大气污染、蛋白质结构设计、图像理解与密码破译等。世界上第一台峰值速度超过 1Tflops 的高性能计算机是由 Intel 公司于 1996 年 12 月研制成功的。

“全面禁试条约”签定后,核武器的研究转到以实验室研究和数值模拟为基础。这样,1996 年 6 月,美国能源部联合美国三大核武器实验室(Los Alamos 国家实验室、Lawrence Livermore 国家实验室和 Sandia 国家实验室)共同提出了“加速战略计算创新(ASCI)计划”,提出通过数值模拟评估核武器的性能、安全性、可靠性、更新等,要求数值模拟达到高分辨率、高逼真度、三维、全物理、全系统的规模和能力。为此,三大实验室分别向美国三大公司(Intel, IBM 和 SGI)预定了峰值速度超过 1Tflops 的并行机,计划分四个阶段,分别

实现万亿次、10 万亿次、30 万亿次和 100 万亿次的高性能并行机。目前，前 2 个阶段已经初步实现，第 3 阶段即将实现。

与此同时，随着我国国民经济的快速发展，能源、气象、国家安全等各个应用部门对高性能计算的需求也越来越迫切，我国的并行机研制水平也得到了快速提高。

从 90 年代初期开始至今，得益于超大规模集成电路等微电子技术的革新，商用微处理器的性能几乎每年增长 1 倍，内存容量几乎每年增长 3~4 倍。为了追求最优的性能价格比，各并行机厂商纷纷改变研制方向，直接将多个高档商用微处理器通过高性能互联网络相互联接而构成高性能并行机。从 90 年代中期开始，由于网络通信技术的快速发展，高性能互联网络的拓扑结构和处理器间的距离也不再是影响并行机性能的重要关键因素。因此，高性能并行机的体系结构逐步趋于统一，从而进一步促进了操作系统、编译系统和并行编程等并行软件环境的统一。

1.1.2 并行机体系结构

对称多处理共享存储并行机(SMP:Symmetric Multi-Processing)、分布共享存储并行机(DSM:Distributed Shared Memory)、大规模并行机(MPP:Massively Parallel Processors)和微机机群(Beowulf PC-Cluster)构成了当前最流行的四类高性能并行机体系结构，它们均属于典型的多指令流多数据流(MIMD)机器。其中，MPP 又可以分为分布式存储大规模并行机(DM-MPP)和 SMP(或 DSM)大规模机群(SMP-MPP, DSM-MPP)两类。下面，我们逐一介绍这四类并行机的体系结构。

对称多处理共享存储并行机(SMP) 典型的对称多处理并行机(SMP)的体系结构如图 1.1 所示，它的基本组成单位是当前流行的商用微处理器，例如 SGI 公司的 MIPS R10000-R12000 系列、Compaq 公司的 Alpha 21264 系列、IBM 公司的 P3 系列、HP 公司的 PA9000 系列、Intel 公司的 Pentium-III 系列。为了缓和快速的处理器速度和较慢的内存访问速度之间的不匹配，每个处理器均配置 1MB-8MB 大小的局部高速缓存 Cache。所有处理器通过系统总线或交叉开关与多个内存模块和输入输出(I/O)模块相联接，所有内

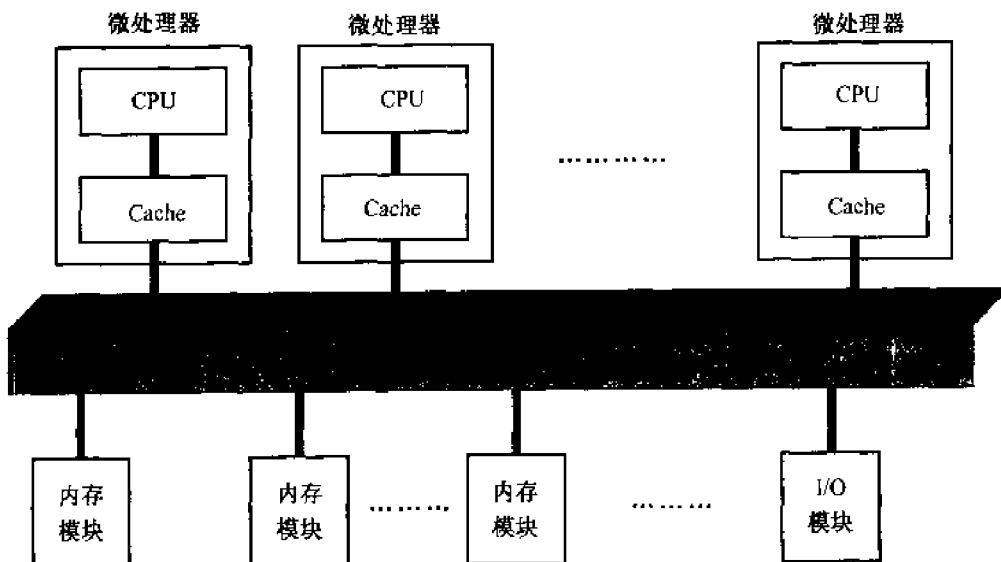


图 1.1 SMP 体系结构示意图

存模块构成 SMP 的内存地址空间,称之为 SMP 的共享存储器。

SMP 具有如下特征:(1)对称共享存储:系统中任何处理器均可直接访问任何存储模块中的存储单元和 I/O 模块联接的 I/O 设备,且访问的延迟、带宽和访问成功的概率是一致的。所有内存地址单元统一编址。各个处理器之间的地位等价,不存在任何特权处理器。操作系统可在任意处理器上运行。(2)单一的操作系统映像:全系统只有一个操作系统驻留在共享存储器中,它根据各个处理器的负载情况,动态地分配各个进程到各个处理器,并保持各处理器间的负载平衡。(3)局部高速缓存 Cache 及其数据一致性:每个处理器均配备局部 Cache,它们可以拥有独立的局部数据,但是这些数据必须保持与存储器内数据的一致。(4)低通信延迟:各个进程通过读/写操作系统提供的共享数据缓存区来完成处理器间的通信,其延迟通常小于网络通信的延迟。(5)共享总线带宽:所有处理器共享总线的带宽,完成对内存模块和 I/O 模块的访问。

同时,SMP 也具有如下一些问题:(1)欠可靠:总线、存储器或操作系统失效可导致系统崩溃。(2)可扩展性较差:由于所有处理器共享总线带宽,而总线带宽每 3 年才增加 2 倍,跟不上处理器速度和内存容量的发展步伐,因此,SMP 并行机的处理器个数一般少于 32 个,且只能提供每秒数百亿次的浮点运算功能。

SMP 并行机的典型代表有:SGI POWER Challenge XL 系列并行机(36 个 MIPS R10000 微处理器)、DEC Alphaserver 84005/440(4 个 Alpha 21264 个微处理器)、HP9000/T600(4 个 HP PA9000 微处理器)和 IBM RS6000/R40(16 个 RS6000 微处理器)。

分布共享存储并行机(DSM) 分布共享存储并行机 DSM 是对称多处理共享存储并行机 SMP 的扩展。它以结点为基本组成单位,每个结点包含多个微处理器、局部存储器和集线器(HUB)。每个微处理器拥有局部 Cache。微处理器、局部存储器和 I/O 设备接口通过 HUB 相互联接。同时,HUB 还与一个路由器联接,所有结点的路由器相互联接形成并行机的高性能互联网络。

DSM 的典型代表为 SGI 公司的 Origin 2000 和 Origin 3000 系列并行机,图 1.2 列出了 SGI Origin 2000 的体系结构。Origin 2000 可扩展到 8 个机柜,每个机柜含 8 个结点,结点是构成 Origin 2000 的基本单位,它包含:

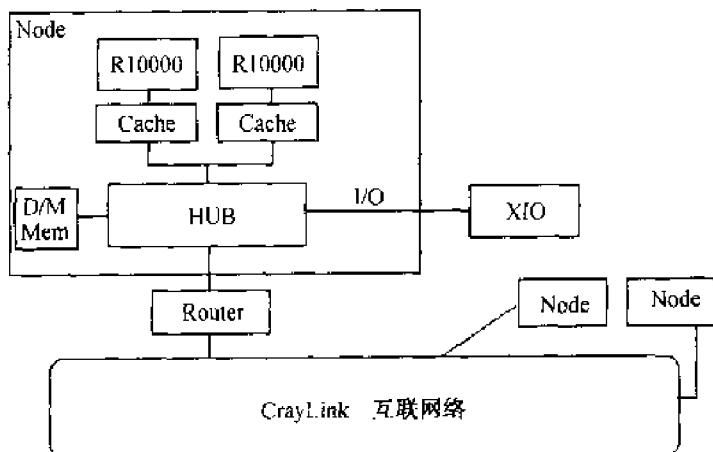


图 1.2 SGI Origin 2000 并行机体系结构示意图

- 1~2 个主频为 195MHz 或 250MHz 的 MIPS R10000 CPU, 每个 CPU 含 4MB 的二级 Cache;
- 内存 512MB~4GB, 分主存和目录内存两类, 后者主要用于保持结点间的 Cache 数据一致性;
- 集线器(HUB)含 4 个端口: CPU 端口、内存端口、XIO 端口和 CrayLink 互联网络端口, 采用交叉开关实现两个 CPU、内存、输入输出和互联网络路由器(router)之间的全互联, 分别提供 780MB/Sec, 780MB/Sec, 1.5GB/Sec, 1.5GB/Sec 的传送速率。

Origin 2000 的所有结点通过 CrayLink 高性能互联网络进行联接, 路由器是构成 CrayLink 的基本单位, 它包含 6 个端口, 内部采用交叉开关实现端口间的全互联, 具有 9.3GB/Sec 的峰值带宽。每个路由器的两个端口用于联接结点, 其余 4 个端口实现路由器间的互联, 形成互联网络拓扑结构。该 CrayLink 的半分带宽与结点个数成线性递增关系, 对任意两个结点, 至少能提供两条路径, 保证了结点间的高带宽、低延迟联接和互联网络的稳定性和容错能力。

相对于 SMP, DSM 具有如下主要特征:(1)物理上分布存储: 内存模块局部在各结点中, 并通过高性能互联网络相互联接, 避免了 SMP 访存总线的带宽瓶颈, 增强了并行机的可扩展能力。(2)单一的内存地址空间: 尽管内存模块分布在各个结点, 但是, 所有这些内存模块都由硬件进行了统一的编址, 并通过互联网络联接形成了并行机的共享存储器。各个结点即可以访问局部内存单元, 又可以访问其他结点的局部内存单元。为此, 引入两个概念: 如果某次内存访问的对象存在于结点自身的局部内存模块中, 则称该次内存访问为本地访问; 如果某次内存访问的对象存在于其他结点的局部内存模块中, 则称该次内存访问为远端访问。(3)非一致内存访问(NUMA)模式: 由于远端访问必须通过高性能互联网络, 而本地访问只需直接访问局部内存模块, 因此, 远端访问的延迟一般是本地访问延迟的 3 倍以上。(4)单一的操作系统映像: 类似于 SMP, 在 DSM 并行机中, 用户只看到一个操作系统, 它可以根据各结点的负载情况, 动态地分配进程。(5)基于 Cache 的数据一致性: 通常采用基于目录的 Cache 一致性协议来保证各结点的局部 Cache 数据与存储器中数据的一致性。同时, 我们也称这种 DSM 并行机结构为 CC-NUMA 结构。(6)低通信延迟与高通信带宽: 专用的高性能互联网络使结点间的延迟很小, 通信带宽可以扩展。例如, 目前最先进的 DSM 并行机 SGI Origin 3000 的双向点对点通信带宽可达 3.2GB/秒, 而延迟小于 1 个微秒。

DSM 并行机较好地改善了 SMP 并行机的可扩展能力。一般地, DSM 并行机可扩展到上百个结点, 能提供每秒数千亿次的浮点运算功能。例如, SGI Origin 2000 可以扩展到 64 个结点(128 个 CPU), 而 SGI Origin 3000 可以扩展到 256 个结点(512 个 CPU)。但是, 由于受 Cache 一致性要求和互联网络性能的限制, 当结点数目进一步增加时, DSM 并行机的性能也将大幅下降。

大规模并行机(MPP) 大规模并行机(MPP)一词的含义并不明确, 但通常是指数百个乃至数千个处理器组成的大规模并行机。例如, 当前位于 TOP 500 前列的并行机均属于这一类, 其中包括 IBM ASCI White(8192 个处理器), Intel ASCI Red(9632 个处理器), IBM ASCI Blue Pacific(5808 个处理器), SGI ASCI Blue Mountain(6144 个处理器), IBM

SP POWER3(1336个处理器),CRAY T3E1200(1084个处理器)等。按存储结构的不同,MPP又可以分为两类:一类是分布式存储大规模并行机(DM-MPP),另一类是由多台SMP或DSM并行机通过高性能互联网络相互联接的大规模机群(SMP-MPP或DSM-MPP)。

图1.3列出了MPP并行机的典型体系结构,它由数百个乃至数千个计算结点和I/O结点组成,这些结点由局部网卡(NIC)通过高性能互联网络相互联接而成。每个结点相对独立,并拥有一个或多个微处理器(P/C)。这些微处理器均配备有局部Cache,并通过局部总线或互联网络与局部内存模块和I/O设备相联接。

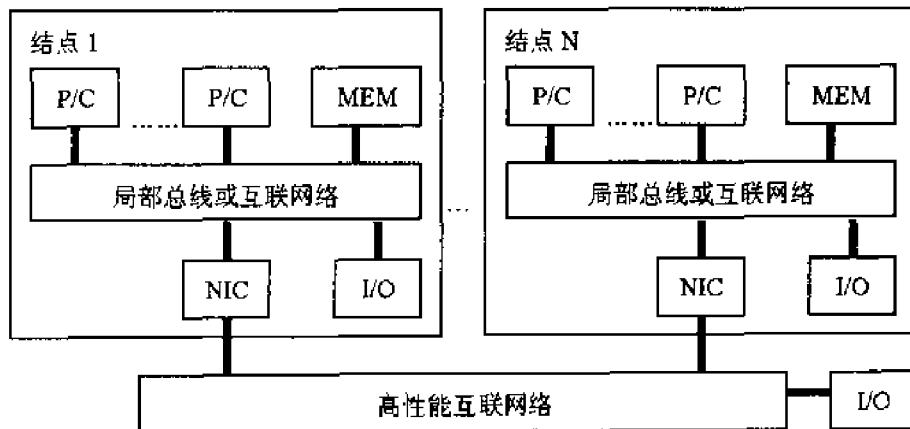


图1.3 MPP体系结构示意图

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

特别地,如果每个结点仅包含一个微处理器,则称该MPP为分布式存储并行机,90年代早期的MPP并行机均属于这一类,其中包括CRAY T3D,CRAY T3E,Intel Paragon,IBM SP-2,YH 3等;如果每个结点是一台SMP并行机,则称该MPP为基于SMP的大规模机群(SMP-MPP),当前位于Top500排名前列的多台MPP并行机均属于这一类,其中包括IBM ASCI White,Intel ASCI Red,IBM Blue Pacific等;如果每个结点是一台DSM并行机,则称该MPP为基于DSM的大规模机群(DSM-MPP),其典型代表为包含6144台处理器的ASCI Blue Mountain MPP并行机,它由48台Origin 2000构成,其中每台含128个微处理器。

目前,单纯的分布式存储MPP并行机已经退出了历史舞台,而SMP-MPP已成为当前国内外并行机研制的主流方向。

微机机群(Beowulf PC-Cluster) 随着商用微处理器性能的飞速发展,低延迟、高带宽商用网络交换机的出现,和Linux操作系统等自由软件的成熟,并行计算机不再是一个只有大型科研单位才能拥有的设备。例如,将128台当前市场上最高性能的Intel

Pentium-III/800MHz 的微机通过 6 个 24 端口的 100Mbps 的网络交换机进行联接, 即可构成浮点峰值性能在 1000 亿次左右的并行机, 而其成本不超过 200 万元人民币, 性能价格比远远高于以上提到的各类并行机, 国际上称该类自行研制的并行机为 Beowulf 机群。

尽管微机机群在通信性能、稳定性和使用方便等方面有待大幅度提高, 但是, 它们以其他并行机无法比拟的性能价格比, 近年来已经成为了高性能并行计算中的一支不可忽视的重要力量。目前, 在我国的各个大学和科研机构, 例如中国科学院、北京大学、清华大学、北京应用物理与计算数学研究所等, 微机机群也得到了快速发展和推广应用。特别地, 在 2000 年底的 Top 500 排名中, 美国 Sandia 国家重点实验室自行研制的机群 Cplant 排名第 84 位。

Beowulf 微机机群的体系结构如图 1.4 所示, 多台高性能微机通过商用网络交换机相互联接, 并拥有各自独立的操作系统、主板、内存、硬盘和其他 I/O 设备, 构成机群的计算结点。配置一台或多台文件服务器, 一方面管理机群计算结点共享的所有软件和用户计算资源, 另一方面充当机群与外部网络的联接桥梁, 外部科研网的用户只有通过文件服务器才能使用机群的计算资源。

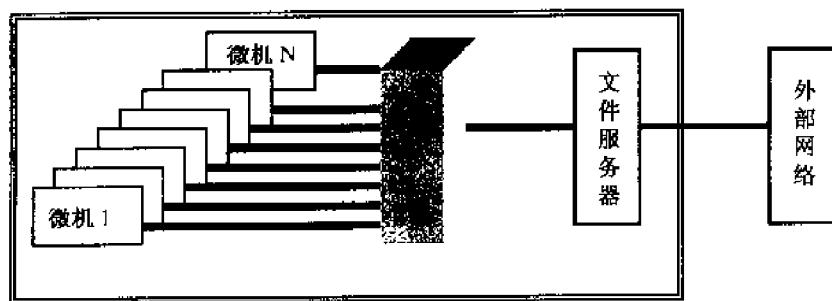


图 1.4 Beowulf 微机机群示意图

由于受商用交换机网络性能和操作系统功能的影响, Beowulf 微机机群的处理机规模一般限制在 100 台左右。但是, 如果将交换机替换成专用机群网络, 例如 GigaNet, Myrinet 等, 则它们的规模可以进一步扩大。因此, 在当前技术条件下, 微机机群一般可提供千亿次左右的浮点峰值功能。

1.1.3 并行机软件环境

UNIX 操作系统几乎是当前所有高性能并行机 (SMP, DSM, MPP, Beowulf PC-Cluster) 采用的标准操作系统, 其中包括 SGI 公司的 IRIX, COMPAQ 公司的 Tru64TM, HP 公司的 HPUX, IBM 公司的 AIX, SUN 公司的 Solaris 和自由软件 Linux 等。虽然各并行机厂商研制的 UNIX 操作系统的实现原理不尽相同, 但是, 它们给用户提供的基本 UNIX 操作系统界面大体是一致的。因此, 用户只要对 UNIX 操作系统有一定的了解, 就可以方便地使用以上介绍的各类并行机。

RedHat 是 Linux 操作系统中一个应用较为广泛的分支之一, 它能提供很好的机群管理功能。目前, 该系统的最新版本为 7.0, 属于自由软件, 可在 Linux 网站 (<http://www.redhat.com/>) 中自由下载, 或在市场上购买。

在程序设计语言方面, SMP, DSM 和 MPP 并行机一般均提供符合国际标准的

Fortran 77, Fortran 90, C/C++, HPF 等语言, 而机群系统一般免费提供 GNU Fortran 77, GNU C/C++ 等语言。特别地, 我们将在第 4 节介绍各类并行机支持的并行程序设计平台。

1.2 进程与进程间通信

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

1.2.1 进程

进程(process)可表示成四元组(P, C, D, S), 其中 P 是程序代码、 C 是进程的控制状态、 D 是进程的数据、 S 是进程的执行状态。任何进程总和程序联系在一起, 程序一旦在具体操作系统环境中投入运行, 就变成了进程。各个进程拥有独立的执行环境, 其中包括内存数据和指令地址空间、程序计数器、寄存器、栈空间、文件系统、I/O 设备等, 并在操作系统的控制、管理、保护和调度下, 在不同的时刻, 动态地申请和占有计算资源。特别地, 我们称进程的内存地址空间为该进程的局部内存空间。

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

任何进程, 在执行过程中, 均涉及以下几种状态:(1)非存在状态:进程依赖的程序还没有投入运行;(2)就绪状态:进程由其父进程(例如, 操作系统的内核进程和 Shell 进程, 或其他应用程序进程)调入并准备运行;(3)运行状态:进程占有 CPU 和其他必需的计算资源, 并执行指令;(4)挂起状态:由于 CPU 或其他必需的计算资源被其他进程占有, 或必需等待某类事件的发生, 进程转入挂起状态以后, 一旦条件满足, 由操作系统唤醒并转入就绪状态;(5)退出状态:进程正常结束或因异常退出而被废弃。

只对消息传递并行程序设计感兴趣的读者, 了解以上的进程概念就够了, 有关进程的详细定义, 将涉及到进程的影像、进程的执行模式、进程的现场活动、进程描述符和进程控制等诸多方面的知识, 有兴趣的读者请参考专门的操作系统专著。

1.2.2 进程间通信

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

中断。当然,位于不同处理机中的多个进程也拥有独立的执行环境,在各自操作系统的调度下,占有各自的计算机资源。

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

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

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

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

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

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

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

1.3 线 程

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

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

由于管理重量级进程的开销较大地影响了并行机性能的发挥,因此,该类进程不适合细粒度的共享存储并行程序设计。为了在共享存储环境下有效地开发应用程序的细粒度并行度,我们将一个进程分解为两个部分,其中一部分由其资源特征构成,称之为**进程**;另一部分由其执行特征构成,称之为**线程**,或者**轻量级进程**。具体地,如图 1.5 所示,进程的指令路径可以分解为并行的互不相关的多条子路径(用曲线表示),而每条子路径可由

一个线程来执行。因此,进程可由单个线程来执行,即通常所说的串行执行;同时,进程也可由多个线程来并行执行,此时,多个线程将共享该进程的所有资源特征,并可以使用不同的CPU,对不同的数据进行处理,从而达到提高进程执行速度的目的。



图 1.5 单进程多线程执行示意图,其中曲线表示执行的指令路径

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

有关线程的详细定义,请读者参考操作系统专著,本书这里不再深入讨论。

1.4 并行编程环境

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

表 1.1 三种并行编程环境主要特征一览表

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

(1)共享存储并行编程和数据并行编程基于细粒度并行,仅被 SMP, DSM 和 MPP 并行机所支持,移植性不如消息传递并行编程。但是,由于它们支持数据的共享存储,所以

并行编程的难度较小,但一般情形下,当处理机个数较多时,其并行性能明显不如消息传递编程。

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

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

1.5 消息传递并行机模型

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

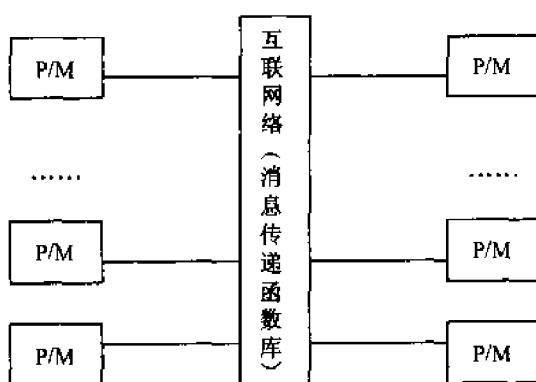


图 1.6 消息传递进程拓扑结构和并行机模型

图 1.6 给出了一个理想的消息传递进程拓扑结构。其中,“P”表示 MPI 进程,“M”表示每个进程的局部内存空间,多个“P/M”进程/内存模块通过互联网络相互联接,构成一个分布式存储的进程拓扑结构。在该结构中,各个进程之间可以直接通信,但是各个进程只能直接访问自身的局部内存空间,对其他进程的局部内存空间的访问只能调用消息传递函数,通过进程间通信才能实现。因此,该进程拓扑结构的核心是联接进程的互联网络,也就是消息传递标准函数库,而构成该函数库的所有函数就构成了用户面对的消息传递并行编程环境。

如果将图 1.6 的每个 P/M 模块替换成处理器,且规定每个处理器只能分配用户程序的一个进程,则所得的理想并行机模型就是消息传递并行机模型。不难看出,消息传递并行程序设计所依赖的并行机模型实际上属于典型的分布式存储并行机,且每台处理器只能分配用户程序的一个进程。基于该并行机模型,用户就可以自由地调用消息传递函数库中的函数来组织具体的并行程序设计,且程序研制成功后,便可以在任何支持该并行机模型隐含的进程拓扑结构的所有具体并行机上运行。

这里,有必要说明的是,消息传递分布式存储并行机模型和具体并行机体系结构没有必然的联系。无论将该模型映射到何种类型的并行机(SMP, DSM, MPP 或微机机群),用户面对的都是该模型隐含的进程拓扑结构,只是各类具体并行机实现的消息传递函数库的方式不同,但用户无需知道这些细节。例如,在共享存储 SMP, DSM 并行机中,消息传

递是通过共享数据缓存区来实现的；在 DM-MPP 并行机和微机机群中，消息传递是通过 Socket 网络通信来实现的；在 SMP-MPP, DSM-MPP 并行机中，消息传递在 SMP, DSM 并行机内部是通过共享数据缓存区实现的，而在 SMP, DSM 并行机之间是通过 Socket 网络通信来实现的。因此，无论哪种类型的并行机，呈现在消息传递并行程序设计用户面前的必然是图 1.6 所示的分布式存储并行机模型。

1.6 标准消息传递界面 MPI

1994 年 6 月，全球工业、政府和科研应用部门联合推出消息传递并行编程环境的标准用户界面（MPI），它将消息传递并行编程环境分解为两个部分，第一是构成该环境的所有消息传递函数的标准接口说明，它们是根据并行应用程序对消息传递功能的不同要求而制定的，不考虑该函数能否具体实现；第二是各并行机厂商提供的对这些函数的具体实现。这样，用户只需学习 MPI 库函数的标准接口，设计 MPI 并行程序，便可在支持 MPI 并行编程环境的具体并行机上执行该程序。通常意义上，我们所说的 MPI 系统就是指这些具有标准接口说明的消息传递函数所构成的函数库。

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

相对其他并行编程环境，MPI 具有许多优点：(1)具有很好的可移植性，几乎被所有并行环境支持；(2)具有很好的可扩展性，是目前高效率的大规模并行计算（数百个处理器）最可信赖的平台；(3)比其他消息传递系统好用；(4)有完备的异步通信功能；(5)有精确的定义，从而为并行软件产业的发展提供了必要的条件。

MPI 1.0 版于 1994 年推出，并同时获得了各并行机厂商的具体实现。MPI 2.0 版于 1998 年 10 月推出，它在 1.0 版的基础上，增加了如下的消息传递功能：(1)并行 I/O：允许多个进程同时读/写同一个文件；(2)线程安全：允许 MPI 进程的多个线程执行，即支持与 OpenMP 的混合并行编程；(3)动态进程管理：允许并行应用程序在执行过程中，动态地增加和删除进程个数；(4)单边通信：允许某个进程对其他进程的局部内存单元直接执行读/写访问，而不需要对方进程的显式干预；(5)并行应用程序之间的动态互操作：允许各个 MPI 并行应用程序之间动态地建立和删除消息传递通信通道。

目前，各类并行机，尤其是微机机群，只实现了 MPI 2.0 版的部分功能，本书的讨论将定位在目前已经实现的各类消息传递库函数，同时对其他没实现的函数也进行了简单的介绍，但不做详细讨论，感兴趣的读者请参考有关 MPI 2.0 版的资料。

第二章 MPI 预备知识

在第一章,我们讨论了消息传递并行程序设计所必需的一些基础理论知识,并简单介绍了标准消息传递界面 MPI,从本章开始,我们将全面介绍 MPI 的各类消息传递库函数。本章从一个简单的 MPI 程序例子入手,介绍 MPI 并行编程所必需的预备知识,其中包括 MPI 并行程序设计流程图、MPI 并行编程模式、MPI 函数的分类,以及其他预备知识。

2.1 MPI 程序示例

下面,我们用 Fortran 77 语言给出一个简单的 MPI 程序例子,它要求在参与并行计算的 P 个进程之间完成一次 Token-Ring 消息传递:初始,进程 0 将初始数据 data 赋值 1,并发送给进程 1;进程 1 接收到该值,并将其加 1,发送给进程 2;依次类推,最后,进程 $P-1$ 从进程 $P-2$ 中接收数据 data,同样将其加 1 后,发送给进程 0;最后,进程 0 从进程 $P-1$ 接收该数据 data,并判断该次 Token-Ring 消息传递是否正确(即 data 是否等于 P),并打印输出。

例 2.1 MPI 程序示例:Token-Ring 消息传递

```
!
implicit none
include "mpif.h"
integer status(MPI_STATUS_SIZE)
integer my_rank, p, source, dest, tag, data, ierr
!
call MPI_Init(ierr)
! 初始化应用程序,进入 MPI 系统,并初始化通信器 MPI_COMM_WORLD,
! 它将包含所有初始启动的进程,给各个进程编号,并允许这些进程之间可以相互通信。
call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
! 从通信器 MPI_COMM_WORLD 中获取本进程的序号 my_rank。
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
! 获取通信器 MPI_COMM_WORLD 包含的进程个数 p。
if(p.le.1) then
  Print *, " Error : number of spawned processes must be larger than 1"
  Stop
endif
tag = 50           ! 消息号赋值。
if (my_rank.eq.0) then
  data = 1         ! 数据赋初值。
```

```

dest = 1           ! 接收消息的进程序号。
!
call MPI_Send( data, 1, MPI_INTEGER, dest, tag, MPI_COMM_WORLD, ierr)
! 将包含数据 data、标号为 tag 的消息发送给进程 dest。
call MPI_Recv( data, 1, MPI_INTEGER, p-1, tag, MPI_COMM_WORLD, status, ierr)
! 从进程 p-1 中接收包含数据 data、标号为 tag 的消息。
else
source = my_rank-1 ! 发送消息的源进程序号。
call MPI_Recv( data, 1, MPI_INTEGER, source, tag,
               MPI_COMM_WORLD, status, ierr)
! 从进程 source 中接收包含数据 data、标号为 tag 的消息。
dest = my_rank + 1
if (dest .eq. p) dest = 0
data = data + 1    ! 数据 data 赋值。
call MPI_Send( data, 1, MPI_INTEGER, dest, tag,
               MPI_COMM_WORLD, ierr)
endif   ! 将包含数据 data、标号为 tag 的消息发送给进程 dest。
!
if ( my_rank.eq.0) then
  if ( data.eq.p) then
    print *, "Successfully Token-Ring Message-Passing with P=", p
  else
    print *, "Sorry, Token-Ring Message-Passing with P=", p, " Failed"
  endif
endif
endif
!
call MPI_Finalize(ierr)      ! 退出 MPI 系统。
end                         ! 应用程序结束。
!
```

参考附录 A 中关于 MPI 程序的编译与运行命令，在支持 MPI 并行编程环境的并行机上，编译运行该程序：

```

mpirun -np 1 编译形成的可执行文件名
mpirun -np 4 编译形成的可执行文件名
mpirun -np 8 编译形成的可执行文件名
mpirun -np 64 编译形成的可执行文件名

```

将分别启动 1, 4, 8, 64 个进程，所有进程将执行该程序编译形成的可执行代码，但按进程编号的不同，执行不同的指令路径，最终输出：

```

Error : number of spawned processes must be larger than 1
Successfully Token-Ring Message-Passing with P= 4
Successfully Token-Ring Message-Passing with P= 8
Successfully Token-Ring Message-Passing with P= 64

```

有关该例的各个语句的说明, 我们将在下节结合 MPI 并行程序设计流程图给出。

2.2 MPI 并行程序设计流程图

结合例 2.1, 本节介绍 MPI 并行程序设计的流程图。

在构成 MPI 程序的任何主程序和子程序中, 只要调用了 MPI 函数, 则该程序必须包含 MPI 系统头文件“mpif.h”(Fortran 77 语言)或“mpi.h”(C 语言)或“mpi++ .h”(C++ 语言), 这些文件包含了 MPI 程序编译所必需的 MPI 系统预先定义的常数、宏、数据类型和函数类型等。

MPI 系统提供给 Fortran 77, C, C++ 语言的是一个标准消息传递函数库。用户通过调用该库中的函数, 可组织各进程间的数据交换和同步通信。根据功能的不同, MPI 2.0 版本将这些函数划分为七类: 点对点通信函数、用户自定义数据类型函数、聚合通信函数、通信器函数、进程拓扑结构函数、并行 I/O 函数和 MPI 环境管理函数。

结合例 2.1, 图 2.1 用 Fortran 77 语言给出了一个用户通常使用的 MPI 并行程序设计流程图。该流程图表明, 在所有 MPI 函数中, 函数 MPI_Init 必须被首先调用, 且只能调用一次, 它的作用是初始化执行 MPI 程序的各个进程, 使之进入 MPI 系统。之后, 各个进程便可以调用其他 MPI 系统提供的任意函数。同时, 各个进程在结束前, 必须调用函数 MPI_Finalize 来通知 MPI 系统, 表明该进程将退出 MPI 系统。这两个函数, 再加上 MPI 系统提供的其他辅助函数, 例如获取程序运行时间、查询进程所属处理机的名称、处理 MPI 错误等, 构成了 MPI 系统的环境管理函数库, 我们将在第九章中详细介绍。

通过调用函数 MPI_Init, 各个进程将获得一个唯一的编号, 我们称之为进程的序号 (rank), 它是各个进程区别于其他进程的唯一标志。假设执行 MPI 程序的进程总数为 P , 则各个进程获得的序号将依次为 $0, 1, 2, \dots, P - 1$ 。进程的序号可用函数 MPI_Comm_rank 来获得, 其函数形式为:

C:	int MPI_Comm_rank (comm, rank)
	MPI_Comm comm
	int rank
Fortran:	MPI_Comm_rank(comm, rank, ierr)
	integer comm, rank, ierr

其中参数 comm 为输入参数, 称为**进程通信器**(communicator), 简称**通信器**, 可理解为一类进程的集合, 且在该集合内部, 各进程可以相互通信。任何 MPI 通信函数所执行的通信操作, 均必须基于某个通信器进行。函数 MPI_Init 调用后, MPI 系统将为所有执行 MPI 程序的进程提供一个已定义好的通信器 MPI_COMM_WORLD, 它将包含所有这些进程, 这些进程也可以基于该通信器自由地组织消息传递。有关 MPI 通信器的详细讨论, 例如通信器的复制、创建、释放、访问、属性和大小等, 我们将在第六章详细介绍。参数 rank 为输出参数, 返回函数调用进程的序号。参数 ierr 也为输出参数, 它将返回该次函数调用是否成功的标志, 我们称之为**信息码**。每个 MPI 函数调用返回时, 都将返回一个信息码。有关信息码的详细定义, 我们将在第 3.1 中给出。

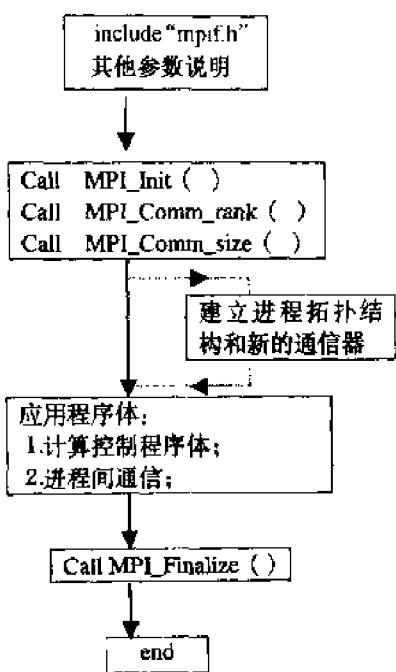


图 2.1 MPI 并行程序设计流程图

0	1	2	3
4	5	6	7
8	9	10	11

(a) 12 个进程

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

(b) 12 个进程的 Cartesion 拓扑结构

图 2.2 进程拓扑结构示意图

执行 MPI 程序的各个进程可以调用函数 `MPI_Comm_size(comm, p)` 来获得通信器 `comm` 包含的进程总数，并存储在变量 `p` 中。如果 `comm = MPI_COMM_WORLD`，则该次函数调用将返回执行该 MPI 程序的进程总数。

MPI 通信器包含两个重要属性：第一是 **进程组** (process group)，它是构成该通信器的所有进程的集合；第二是 **进程拓扑结构** (process topology)，它是构成进程组的所有进程之间的一种相互联接的虚拟拓扑结构，其目的是方便并行程序设计和提高并行计算性能。例如，假设某个应用问题定义在二维规则区域上，如图 2.2(a) 所示，并行程序设计时，我们沿着横向将区域划分成 4 个子区域，沿着纵向将区域划分成 3 个子区域，形成 12 个子区域，分配给 12 个不同的进程。尽管基于通信器 `MPI_COMM_WORLD`，这些进程可以相互通信。但是，在某些具体应用领域，例如二维流体力学数值模拟，大多数情况下，一个进程只需与相邻进程进行通信。例如，在图 2.2(a) 中，进程 5 只需与进程 {1, 4, 9, 6} 通信。这样，如何确定各个进程的相邻进程的序号将是一个繁琐和易错的工作。为此，MPI 系统提供了拓扑构造函数，可将这些进程映射到二维 Cartesion 坐标，如图 2.2(b) 所示。于是，每个进程均对应一个拓扑结构中的坐标，进程 (i, j) 的 4 个相邻进程就可以很快确定为 $\{(i-1, j), (i+1, j), (i, j-1), (i, j+1)\}$ 。有关进程组拓扑结构的创建、复制、释放等 MPI 函数，我们在第七章详细介绍。

MPI 程序经过初始化，并建立所需的通信器和进程拓扑结构后，就可进入一般的并行程序设计阶段了。在该阶段，各个进程可以根据自身序号的不同，选择执行 MPI 程序的不同指令路径。各个进程在运行过程中，可以调用 MPI 函数组织相互之间的消息传递，交换必需的信息，其中主要涉及到两种类型的通信操作：第一种是点对点通信，它可定

义为执行 MPI 程序的任意两个进程之间的直接通信,其中一个进程执行消息发送操作发送一个消息到另一个进程,而另一个进程执行消息接收操作接收发送来的消息;第二种是聚合通信,它是同一个通信器内所有进程均必须参与的通信,通常要求对各个进程的某个局部变量执行一个操作,例如求最大值、最小值、累加和等,得到一个新值,并将该值存储在某些指定的进程中。本书将在第三章和第五章分别介绍 MPI 系统提供的点对点通信函数和聚合通信函数。

MPI 系统提供特殊的数据结构来支持通信器、进程组、拓扑结构等概念,并称之为**MPI 对象**(object)。这些 MPI 对象的具体形状和大小,对用户是不透明的。在 MPI 程序中,称联接这些不透明对象的具体变量为**MPI 联接器**(handle)。用户可以通过联接器与 MPI 对象建立联接,并通过该联接,可以直接访问 MPI 对象,也可以通过联接器释放已经建立的联接。同时,用户还可以动态地创建和释放属于自己的 MPI 对象,例如通信器、进程组、进程拓扑结构等。不管怎样,MPI 对象和对象联接器均只能局部于各个进程内部,不能通过消息在进程间传递。

在例 2.1 中,根据各自序号的不同,进程 0, 进程 1…进程 $P - 1$ 执行了完全不同的指令路径。具体地,进程 0 首先调用函数 MPI_Send 向进程 1 发送一个包含数据 data 的消息,然后调用函数 MPI_Recv 从进程 $P - 1$ 接收一个消息,并将该消息包含的数据存储在变量 data 中,然后判断 data 是否正确,并打印数据结果,最后退出 MPI 系统;而进程 i 首先调用函数 MPI_Recv 从进程 $j = \text{MOD}(i - 1, P)$ 中接收一个消息,并将消息包含的数据加 1 后存储在变量 data 中,然后调用消息发送函数 MPI_Send 向进程 $j = \text{MOD}(i + 1, P)$ 发送一个包含变量 data 的消息,之后,就退出 MPI 系统。有关消息发送函数 MPI_Send 和消息接收函数 MPI_Recv 的各参数的具体含义,我们将在第三章详细介绍,这里不再讨论。

2.3 MPI 并行编程模式

在例 2.1 和图 2.1 的基础上,本节进一步介绍 MPI 系统支持的两种并行编程模式:第一种是单程序多数据流模式(SPMD);第二种是多程序多数据流模式(MPMD)。

SPMD 模式如图 2.3(a)所示, MPI 程序编译形成一个可执行代码后,由命令“mpirun -np N 可执行代码名”(参考附录 A)同时启动 N 个完全独立的进程,它们执行的是同一个 MPI 程序,但根据各自序号的不同,它们执行的是 MPI 程序的不同指令路径。各进程调用函数 MPI_Init 进入 MPI 系统之后,就可以调用 MPI 系统提供的函数来完成进程间消息传递。于是,所有进程通过消息传递可以相互协调地并行完成同一个任务。

如图 2.3(b)所示,与 SPMD 模式不同的是,MPMD 模式包含多个 MPI 程序,各个程序编译后形成不同的可执行代码。命令“mpirun -np N_1 可执行代码 1, -np N_2 可执行代码 2, …, -np N_K 可执行代码 K”(参考附录 A)同时启动 $N = N_1 + N_2 + \dots + N_K$ 个进程,其中 $N_j (j = 1, \dots, K)$ 个进程将执行可执行代码 j 。除此之外,MPMD 模式与 SPMD 模式是完全一致的。

MPI 并行程序设计采用何种并行编程模式,要视具体应用问题的特征而定,它们在并行程序设计的难度和并行计算性能上没有本质的差别。但是,在大多数情况下,为了降

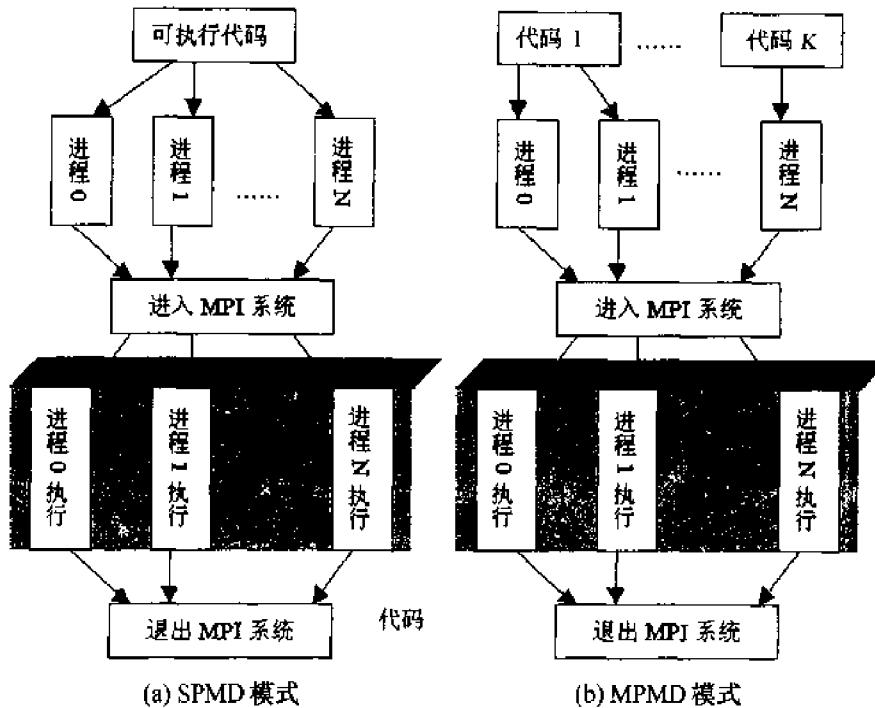


图 2.3 MPI 消息传递编程模式示意图

低使用和维护并行应用软件的复杂度,我们建议读者尽量采用 SPMD 模式。

MPI 1.0 版本规定,无论是 SPMD 模式,还是 MPMD 模式,所有启动的进程必须在命令 `mpirun` 中通过“`-np`”选项来显式指定(参考附录 A),且在执行过程中,任何进程均不能动态地产生或退出,否则将导致程序的运行失败。本书以后各章节的讨论将严格遵循这一规定。

如何映射执行 MPI 并行程序(SPMD 模式或 MPMD 模式)的各个进程到具体并行机的各个处理器,通常有两种方式:第一种是用户指定处理器运行特殊的进程,以获取最优的效率;第二种是用户将 MPI 程序提交给具体并行机的作业调度系统,由它来分配处理器。两种方式的具体命令形式,请参考附录 A 有关 MPI 程序运行的详细讨论。

此外,还有一种情形值得考虑,就是当并行机处理器个数少于启动的 MPI 进程个数时,这些进程能否运行呢?其实,执行 MPI 程序的各个进程可以在单处理器上运行,也可以在任意多台处理器上运行,它与处理器的个数无关。只是当处理器个数少于进程个数时,多个进程可能共享同一个处理器,降低并行计算性能,但不影响程序执行的正确性。因此,如果条件允许,我们可以在自己办公室的单机上编辑、调试 MPI 并行程序,待其正确运行后,再移植到具体并行机上。

2.4 MPI 函数的分类

按功能的不同,MPI 函数可以分为点对点通信函数、用户自定义数据类型函数、聚合通信函数、通信器函数、进程拓扑结构函数、并行 I/O 函数和 MPI 环境管理函数等七类。

本书以后的各章将逐一介绍各类函数。

在执行过程中,如果 MPI 函数只涉及到调用它的进程,而不需要进程间的相互通信,则我们称该 MPI 函数是**局部(local)**函数;如果 MPI 函数涉及到进程间的通信,需要多个进程之间的相互协调和同步,则我们称该 MPI 函数是**非局部(non-local)**函数。例如,例 2.1 中的 MPI 函数 MPI_Comm_rank 和 MPI_Comm_size 就是局部函数,而 MPI 函数 MPI_Init 和 MPI_Finalize 是非局部函数,它们需要多个进程之间的同步。

在执行 MPI 程序的各个进程调用 MPI 函数时,需要提供给该函数一些必要的计算资源,例如存储消息的数据缓存区,当函数返回时,存在两种不同结果:第一种是进程可以自由地修改这些计算资源的内容,它们不会对该函数的执行结果产生任何影响;第二种是进程对这些计算资源的修改将可能影响该函数的执行结果。MPI 系统称第一种类型为**阻塞式(blocking)**函数,它所执行的通信为**阻塞通信**,称第二种类型为**非阻塞式(non-blocking)**函数,它所执行的通信为**非阻塞通信**。例如,例 2.1 中的 MPI 函数 MPI_Send 和 MPI_Recv 是阻塞式函数,而语句为:

```
CALL MPI_SEND()
```

和语句:

```
CALL MPI_RECV()
```

执行的是阻塞通信。有关阻塞与非阻塞的含义,本书第三章将进行详细的讨论。

2.5 其他的预备知识

在介绍 MPI 函数时,本书将采用统一的书写格式,具体在第三章第一个函数 MPI_Send 的说明中介绍,且全书所有的 MPI 函数均将遵循这种说明格式。

在阅读本书的过程中,建议读者着重了解各类 MPI 函数的语义和用途,以及各类数据结构的特点,不必死记各个函数的具体形式。并行程序设计过程中,如果需要调用某个函数来完成某项任务,则可按任务的类型在本书寻找相应的函数名字,或者记录名字后,用联机命令 man 查询该函数的使用方式。特别地,查询 MPI 函数的命令格式为:

```
man MPI_Xxxx (MPI 函数名)
```

其中,函数名的字符 MPI 和下划线后的第一个字符必须大写,其余字符小写。例如,要查询函数 MPI_Init,则输入命令:

```
man MPI_Init
```

即可。

第三章 点对点通信

在第二章 2.2 节中,我们将点对点通信定义为执行 MPI 程序的任意两个进程之间的一次消息传递,其中一个进程执行消息发送操作,另一个进程执行消息接收操作。在 MPI 系统内部,完成一次消息传递,至少需要两个以上进程的共同参与,且其中至少存在一个进程发送消息,另一个进程接收消息。特别地,我们称发送消息的进程为**发送进程**,接收消息的进程为**接收进程**,并称在该次消息传递中,发送进程和接收进程是相互匹配的。

点对点通信是 MPI 通信的基础,可分为四种模式:

(1)**标准模式**(standard mode):执行 MPI 程序的进程可以自由地发送消息,而不必关心与之相匹配的接收进程的运行状态,甚至还可以发送不存在接收进程的消息。标准模式发送的消息,首先会被 MPI 系统缓存起来,然后通过网络传递给接收进程。

(2)**缓存模式**(buffered mode):执行 MPI 程序的进程显式地为 MPI 系统提供内存空间,用于缓存将要发送的消息,除此之外,缓存模式的含义与标准模式是一致的。

(3)**同步模式**(synchronous mode):类似于电话通信,当且仅当发送进程和接收进程建立联系后,消息才沿联接开始传递。

(4)**就绪模式**(ready mode):进程发送消息前,与之相匹配的接收进程必须已经提交了相匹配的消息接收操作,并正在等待该消息,否则,消息发送可能出错。

在第二章 2.4 节中,我们曾将 MPI 函数分为阻塞式和非阻塞式两类,类似地,以上四种模式也可以进一步分为阻塞式和非阻塞式两类,特别地,我们称阻塞式函数执行的通信操作为**阻塞通信**,而非阻塞式函数执行的通信操作为**非阻塞通信**。有关阻塞和非阻塞通信的内在含义,本章各节将会详细介绍。

标准模式是 MPI 点对点通信中最常使用的模式,因此,为了叙述简单,本章介绍的所有点对点通信函数,除非特别声明,均属于标准模式。本章 3.1 节介绍标准模式阻塞通信,3.2 节介绍 MPI 系统定义的进程间通信必须遵循的数据类型的匹配与转换规则,3.3 节介绍标准模式非阻塞通信,3.4 节讨论 MPI 系统内部有限的缓存区资源对消息传递的影响,3.5 节介绍持久通信,3.6 节介绍其他模式的通信函数。

3.1 标准模式阻塞通信

MPI 系统提供的阻塞式通信函数主要有消息发送函数 MPI_SEND, 消息接收函数 MPI_RECV, 消息发收函数 MPI_SENDRECV, 以及消息发收替换函数 MPI_SENDRECV_REPLACE。下面,我们一一介绍。

3.1.1 消息发送/接收函数

```
/* 阻塞式消息发送函数,执行一个标准模式的阻塞式消息发送通信操作。
```

```

MPI_SEND(buf, count, datatype, dest, tag, comm)
    IN      buf        消息发送缓存区的初始地址
    IN      count      消息发送缓存区包含的数据单元个数
    IN      datatype   数据单元类型
    IN      dest       接收该消息的进程序号
    IN      tag        消息标号
    IN      comm       通信器

C     int MPI_Send( void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
                  comm)

Fortran MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type>    BUF(*)
    INTEGER    COUNT, DATATYPE, DEST, TAG, COMM, IERROR

/* 消息接收函数, 执行一个标准模式的阻塞式消息接收通信操作。

MPI_RECV(buf, count, datatype, source, tag, comm, status)
    OUT     buf        消息接收缓存区初始地址
    IN      count      消息接收缓存区允许的最大数据单元个数
    IN      datatype   数据单元类型
    IN      source     发送该消息的进程序号
    IN      tag        消息号
    IN      comm       通信器
    OUT     status     接收返回状态信息

C     int MPI_Recv( void * buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
                  comm, MPI_Status * status)

Fortran MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type>    BUF(*)
    INTEGER    COUNT, DATATYPE, SOURCE, TAG, COMM,
               STATUS(MPI_STATUS_SIZE), IERROR

```

在详细讨论这两个函数之前, 我们介绍本书采用的函数说明格式。具体为:

- (1) 在说明的第一行, 给出函数的名字和它所执行的操作, 并在行首用“/*”标示。
- (2) 在说明的第二行, 给出独立于语言的函数说明格式, 即函数名和函数调用必需的重要参数, 并在随后的各行中逐一解释各参数含义。
- (3) 在参数解释行给出各个参数的解释, 由三个部分组成: 第一部分 IN 或 OUT 或 INOUT 分别表示该参数为输入参数、输出参数和既输入又输出的参数; 第二部分给出参数名字; 第三部分给出参数的具体含义。
- (4) 参数解释完毕, 给出函数在 C 语言中的调用格式, 并在行首用大写字符“C”标示, 同时, 各参数的数据类型也在参数前说明。
- (5) 最后, 给出函数在 Fortran 语言中的调用格式, 并在行首用单词“Fortran”标示, 并在随后的各行中列出各参数的数据类型。

此外, 敬请读者注意以下几点:

- (1) 本书提到的 Fortran 语言是指标准 Fortran 77 语言, 函数说明的数组下标从 1 开始; 本书提到的 C 语言是指标准 C 语言, 函数说明的数组下标从 0 开始; 本书不讨论 MPI 系统的 C++ 版本, 有兴趣的读者请参考 MPI 的相关资料。
- (2) 一个变量, 如果充当 MPI 对象的联接器, 则在 Fortran 语言中, 该变量必须被说明为整型变量, 而在 C 语言中, 该变量必须被说明为 MPI 对象所对应的数据类型变量。
- (3) 对 Fortran 语言, 所有 MPI 系统定义的函数和常数的名字, 大小写均可, 例如函数 MPI_Send 即可以写成 MPI_SEND, 也可以写成 mpi_send, 但是本书统一采用大写 MPI_SEND; 对 C 语言, MPI 系统要求 MPI 程序必须严格遵循函数说明的大小写格式。
- (4) 任何 MPI 函数执行完毕, 均将返回一个信息码, 标志该函数是否被成功地执行。具体在 C 语言中, 该信息码为函数执行返回的值; 在 Fortran 语言中, 信息码将被存储在函数调用的最后一个整型变量中, 例如函数 MPI_SEND 和函数 MPI_RECV 的参数 IERROR。表 3.1 列出了信息码的所有可能取值, 它们均是 MPI 系统内部定义的常数, 满足关系:

$$0 = \text{MPI_SUCCESS} < \text{MPI_ERR_BUFFER} < \dots \leq \text{MPI_ERR_LASTCODE}$$

表 3.1 MPI 函数调用返回信息表

序号	返回参数值	含义
①	MPI_SUCCESS	调用成功
③	MPI_ERR_COUNT	无效 COUNT 参数
⑤	MPI_ERR_TAG	无效的消息号
⑦	MPI_ERR_RANK	无效的进程序号
⑨	MPI_ERR_ROOT	无效的根进程
⑪	MPI_ERR_OP	无效的操作符
⑬	MPI_ERR_DIMS	无效的维数参数
⑯	MPI_ERR_UNKNOWN	不知道的错误
⑭	MPI_ERR_OTHER	不在表中的已知错
⑮	MPI_ERR_IN_STATUS	错误在 STATUS 中
⑰	MPI_ERR_LASTCODE	最近信息码
⑲	MPI_ERR_BUFFER	无效的缓存区指针
⑳	MPI_ERR_TYPE	无效的数据类型参数
㉑	MPI_ERR_COMM	无效的通信器
㉓	MPI_ERR_REQUEST	无效的通信请求
㉔	MPI_ERR_GROUP	无效的进程组
㉕	MPI_ERR_TOPOLOGY	无效的拓扑结构
㉖	MPI_ERR_ARG	无效的其他类参数
㉗	MPI_ERR_TRUNCATE	消息接收时出现了截断
㉘	MPI_ERR_INTERN	MPI 系统内部错误
㉙	MPI_ERR_PENDING	无效的通信提交请求

下面,我们讨论函数 MPI_SEND 和函数 MPI_RECV 的具体含义。函数 MPI_SEND 执行一个标准模式的阻塞式消息发送操作,它将消息发送缓存区包含的数据单元通过通信器 comm,发送给序号为 dest 的进程;函数 MPI_RECV 执行一个标准模式的阻塞式消息接收操作,它通过通信器 comm,从序号为 source 的进程接收一个消息,并将该消息包含的数据单元存储在消息接收缓存区中。其中,消息发送(接收)缓存区由参数 (buf, count, datatype)说明,表示从变量 buf 指定的起始地址开始,连续 count 个类型为 datatype 的数据单元所占的进程局部内存空间。对 Fortran 语言,buf 可以是程序说明的任何变量,或者数组的任何元素的名字,它代表该变量或元素的内存地址;对 C 语言,buf 必须是一个指向进程某个内存地址的指针变量。

参数 datatype 必须是 MPI 系统允许的数据类型,它可以是 MPI 系统预先定义的基本数据类型,也可以是执行 MPI 程序的各进程自己定义的新的 MPI 数据类型,称之为用户自定义数据类型或导出数据类型,我们将在第四章中详细介绍。MPI 系统预先定义的基本数据类型和 Fortran 与 C 语言定义的基本数据类型具有一一对应关系,具体在表 3.2 中列出。这种对应关系的主要意义在于,给定 MPI 程序的任何变量,在 MPI 通信函数中,可以采用与该变量数据类型相对应的 MPI 基本数据类型来说明。例如,对 Fortran 语言,类型为 INTEGER 的变量可以在 MPI 通信函数中用 MPI_INTEGER 来说明;对 C 语言,类型为 double 的变量可以在 MPI 通信函数中用 MPI_Double 来说明。

在表 3.2 中,MPI_BYTE 表示一个字节,MPI_PACKED 表示一种特殊的数据类型,我们将在第四章介绍。此外,不同的并行机可能为这些基本数据类型提供其他的名称,敬请读者注意。

表 3.2 MPI 系统基本数据类型与 C, Fortran 语言的基本数据类型对应关系

Fortran 77		C	
MPI	Fortran 77	MPI	C
MPI_INTEGER	INTEGER	MPI_CHAR	signed char
MPI_REAL	REAL	MPI_SHORT	signed short int
MPI_DOUBLE_PRECISION	DOUBLE PRECISION	MPI_INT	signed int
MPI_COMPLEX	COMPLEX	MPI_LONG	signed long int
MPI_LOGICAL	LOGICAL	MPI_UNSIGNED_CHAR	unsigned char
MPI_CHARACTER	CHARACTER	MPI_UNSIGNED_SHORT	unsigned short int
MPI_BYTE		MPI_UNSIGNED	unsigned int
MPI_PACKED		MPI_UNSIGNED_LONG	unsigned long int
		MPI_FLOAT	float
		MPI_DOUBLE	double
		MPI_LONG_DOUBLE	long double
		MPI_BYTE	
		MPI_PACKED	

在 MPI 系统中,消息可分为数据(data)和包装(envelope)两个部分。消息的包装由发送或接收该消息的进程序号(source, dest)、消息标号(tag)和通信器(comm)组成,其中,参数 source 和 dest 可用于区别不同的进程,tag 可用于区别来自或发送给同一进程的不同消息,comm 可用于区别该消息所属的通信器。如果通信器包含的进程个数为 P ,则 dest 必须取值于 $0, \dots, P-1$ 之间,tag 必须取值于 $0, \dots, UB$ 之间,其中 UB 为 MPI 系统所允许的最大消息号,由 MPI 常数 MPI_TAG_UB 定义,一般不小于 32767。函数 MPI_RECV 还提供了对参数 source 的通配值 MPI_ANY_SOURCE 和 tag 的通配值 MPI_ANY_TAG,分别表示可接收来自任意进程和任意标号的消息。由此,source 的有效取值范围为 $\{0, \dots, P-1\} \cup \{\text{MPI_ANY_SOURCE}\}$,tag 的有效取值范围为 $\{0, \dots, UB\} \cup \{\text{MPI_ANY_TAG}\}$ 。

消息的数据部分由消息发送缓存区包含的所有数据单元组成,其中,数据单元既可以指 Fortran 和 C 语言定义的基本数据类型,又可以指用户自定义的数据类型,数据单元个数指消息拥有的数据单元的个数,称之为消息长度,同时,消息包含的数据所占内存空间(以字节为单位)的大小称为消息大小。

函数 MPI_SEND 具有两种不同的语义:第一种是发送的消息将被拷贝给 MPI 系统,一旦 MPI 系统成功地缓存了该消息,函数就可以返回,而该消息在网络中的传递将由 MPI 系统来完成;第二种是 MPI 系统将不提供对消息的缓存,只有当接收该消息的进程执行相匹配的消息接收操作时,该消息才由 MPI 系统通过网络写入消息接收缓存区中,且直到消息被全部写入,函数 MPI_SEND 才返回。显然,由第二章 2.4 节的 MPI 函数分类可知,第一种语义隐含函数 MPI_SEND 是局部函数,而第二种语义隐含函数 MPI_SEND 是非局部函数。在具体并行机上,MPI 系统采用何种语义来具体实现函数 MPI_SEND,需要用户自己通过一些程序例子来测试。但是,在微机机群和分布式存储并行机上,MPI 系统一般均采用第一种语义;在共享存储并行机上,对短消息,MPI 系统采用第一种语义,而对长消息,MPI 系统采用第二种语义。

函数 MPI_RECV 的语义比较简单,它规定:如果接收的消息在 MPI 系统中已经存在,则立即接收该消息,并将其写入消息接收缓存区之后,函数返回;如果接收的消息还没有提交给 MPI 系统,也就是相匹配的消息发送操作还没有被执行,则必须阻塞等待该消息的发送,直到接收到该消息之后,函数返回。

函数 MPI_RECV 允许消息接收缓存区的长度 count 大于消息的实际长度,且消息接收后,缓存区中多余的内存空间将不会被覆盖,但是,如果消息的实际长度大于消息接收缓存区的长度,消息将被截断,产生溢出错,敬请读者注意。

对 Fortran 语言,函数 MPI_RECV 的参数 status 是长度为 MPI_STATUS_SIZE 的整型数组,其三个元素 status(MPI_SOURCE)、status(MPI_TAG) 和 status(MPI_ERROR) 分别表示消息的 source, tag 和返回信息码,如 status(MPI_SOURCE)=0 表示空进程;对 C 语言, status 则是类型为 MPI_Status 的结构变量,其三个域 status.MPI_SOURCE, status.MPI_TAG 和 status.MPI_ERROR 分别包含消息的 source, tag 和返回信息码。

在例 2.1 中,我们曾给出了应用函数 MPI_SEND 和函数 MPI_RECV 在各进程之间执行一次 Token-Ring 消息传递的 MPI 程序,这里,我们给出另一个例子,以加深对这

两个函数的理解。

例 3.1 消息发送和消息接收函数示例

```
REAL * 8      A(100,200)
INTEGER       NSTATUS(MPI_STATUS_SIZE)
CALL         MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
IF(MYRANK.EQ.1) THEN
    CALL MPI_SEND(A(20,120),40,MPI_DOUBLE_PRECISION,3,9999,
                  MPI_COMM_WORLD,IERR)
ELSE IF(MYRANK.EQ.3) THEN
    CALL MPI_RECV(A(50,180),200,MPI_DOUBLE_PRECISION,1,9999,
                  MPI_COMM_WORLD,NSTATUS,IERR)
ENDIF
```

在上例中, 进程 1 将二维数组 $A(100,200)$ 的连续 40 个双精度数 $A(20:59,120:120)$ 发送给进程 3; 进程 3 接收该消息, 并将这 40 个数存储在元素 $A(50:89,180:180)$ 中。

这里, 读者要注意以下几个方面:

- (1) Fortran 语言规定数组的元素按列连续存储, C 语言规定数组的元素按行连续存储。
- (2) 在函数 MPI_SEND 中, 消息发送缓存区的首地址是 $A(20,120)$, 表示消息将包含 $A(20,120)$ 之后的连续 40 个双精度数。
- (3) 在函数 MPI_RECV 中, 消息接收缓存区的首地址是 $A(50,180)$, 表示从该地址开始, 之后的连续 200 个双精度数据单元可能被接收的消息覆盖, 但在该例中, 消息的长度等于 $40 < 200$, 因此, 该消息能被正确接收并存储在地址 $A(50,180)$ 之后的 40 个双精度数据单元中。
- (4) 任何匹配的消息发送函数和消息接收函数必须基于同一个通信器发生, 且消息号必须相互匹配。在本例中, 通信器为 MPI_COMM_WORLD, 消息号等于 9999。

3.1.2 一个示例: 并行矩阵乘

假定 A 为 $M \times N$ 阶矩阵, B 为 $N \times L$ 阶矩阵, P 为进程的个数, M, N, L 均能被 P 整除。初始状态下, 矩阵 $A(M, N)$ 按行块、矩阵 $B(N, L)$ 按列块依次分布存储在各个进程中, 即子矩阵 $A(i \times M/P + 1 : (i+1) \times M/P, 1 : N)$ 和子矩阵 $B(1 : N, i \times L/P + 1 : (i+1) \times L/P)$ 存储在进程 i ($i = 0, \dots, P - 1$) 中。求 $M \times L$ 阶矩阵 C , 其中:

$$C(M, L) = A(M, N) \times B(N, L)$$

并将其按行块依次存储在各进程中。

例 3.2 并行矩阵乘

```
PARAMETER(MP=M/P, LP=L/P)          ! 子矩阵规模参数说明
REAL * 8     A(MP,N),B(N,LP),C(MP,L)  ! 子矩阵说明
```

```

INTEGER I,J,K,KK, MYRANK, MYLEFT, MYRIGHT, IC, IERR
INTEGER NSTATUS(MPI_STATUS_SIZE)

!
! 初始化,并行子矩阵赋初值(忽略)
!
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
MYLEFT = MYRANK - 1
MYRIGHT = MYRANK + 1
IF(MYLEFT.EQ.-1) MYLEFT = P - 1
IF(MYRIGHT.EQ.P) MYRIGHT = 0
IC = (MYRANK - 1) * LP
!
! 并行矩阵乘开始。
DO 100 K = 1, P
    IC = IC + LP           ! 当前计算的矩阵 C 的起始列。
    IF(IC.GE.L) IC = IC - L
    !
    ! 局部子矩阵乘。
    DO 10 J = 1, LP
        DO 10 I = 1, MP
            DO 10 KK = 1, N
                C(I, IC + J) = C(I, IC + J) + A(I, KK) * B(KK, J)
10             CONTINUE
            IF(K.EQ.P) GOTO 100
            !
            ! 消息传递,依次交换矩阵 B 的子块。
            CALL MPI_SEND(B, N * LP, MPI_DOUBLE_PRECISION, MYLEFT, K * 100,
                           MPI_COMM_WORLD, IERR)
            CALL MPI_RECV(B, N * LP, MPI_DOUBLE_PRECISION, MYRIGHT, K * 100,
                           MPI_COMM_WORLD, NSTATUS, IERR)
            !
100        CONTINUE
            !
            ! 正确性验证,程序退出(忽略)。
            !

```

3.1.3 消息发收函数

/* 消息发收函数,同时执行一个阻塞式消息发送和一个阻塞式消息接收通信操作。

```

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
              recvcount, recvtype, source, recvtag, comm, status)
IN      sendbuf      消息发送缓存区初始地址
IN      sendcount     发送的数据单元个数
IN      sendtype      发送的数据单元类型

```

IN	dest	接收进程的序号
IN	sendtag	发送消息号
OUT	recvbuf	消息接收缓存区初始地址
IN	recvcount	接收的数据单元个数
IN	recvtype	接收的数据单元类型
IN	source	发送进程的序号
IN	recvtag	接收消息号
IN	comm	通信器
OUT	status	接收返回状态信息

```
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)

Fortran 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
```

函数 MPI_SENDRECV 同时执行两个 MPI 函数，其中一个是阻塞式消息发送函数：

```
CALL MPI_SEND(sendbuf, sendcount, sendtype, dest, sendtag, comm, ierr)
```

另一个是阻塞式消息接收函数：

```
CALL MPI_RECV(recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierr)
```

当且仅当这两个函数执行完毕，函数 MPI_SENDRECV 才返回。其中，各参数的含义与函数 MPI_SEND 和函数 MPI_RECV 对应参数的含义一致。显然，为了避免数据的相互冲突，消息发送缓存区 (sendbuf, sendcount, sendtype) 和消息接收缓存区 (recvbuf, recvcount, recvtype) 必须互不相交。

函数 MPI_SENDRECV, MPI_SEND 和 MPI_RECV 相互兼容，即函数 MPI_SEND 发送的消息，可用函数 MPI_SENDRECV 接收；反之，函数 MPI_SENDRECV 发送的消息，也可用函数 MPI_RECV 来接收。

例 3.2 并行矩阵乘中的两条消息传递语句可用下面的语句替换：

```
CALL MPI_SENDRECV(B, N * LP, MPI_DOUBLE_PRECISION, MYLEFT, K * 100, D,
                  N * LP, MPI_DOUBLE_PRECISION, MYRIGHT, K * 100, MPI
                  _COMM_WORLD, NSIATUS, IERR)

DO J = 1, LP
  DO I = 1, N
    B(I,J) = D(I,J)
  ENDDO
ENDDO
```

其中，D 是另一个 $N \times LP$ 阶矩阵。

```

/* 消息发收替换函数,同时执行一个阻塞式消息发送和一个阻塞式消息接收
/* 通信操作,并用接收的消息覆盖缓存区中内容。

MPI _ SENDRECV _ REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)
    INOUT    buf      消息(发送/接收)缓存区初始地址
    IN       count    消息(发送/接收)缓存区内数据单元个数
    IN       datatype 数据单元类型
    IN       dest     接收进程的序号
    IN       sendtag  发送消息号
    IN       source   发送进程的序号
    IN       recvtag  接收消息号
    IN       comm     通信器
    OUT      status   接收返回状态信息

C      int MPI _ Sendrecv _ replace(void * buf, int count, MPI _ Datatype datatype, int dest, int sendtag, int
                                  source, int recvtag, MPI _ Comm comm, MPI _ Status * status)

Fortran 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

```

函数 MPI _ SENDRECV _ REPLACE 等价于执行一个阻塞式消息发送函数：

```
CALL MPI _ SEND(buf, count, datatype, dest, sendtag, comm, ierr)
```

然后执行一个阻塞式消息接收函数：

```
CALL MPI _ RECV(buf, count, datatype, source, recvtag, comm, status, ierr)
```

函数 MPI _ SENDRECV _ REPLACE 返回时,缓存区(buf, count, datatype)将包含函数接收的消息。

例 3.2 中的两条消息传递语句可用下面的语句替换：

```
CALL MPI _ SENDRECV _ REPLACE(B, N * LP, MPI _ DOUBLE _ PRECISION, MYLEFT, K *
                               100, MYRIGHT, K * 100, MPI _ COMM _ WORLD,
                               NSSTATUS, IERR)
```

3.1.4 消息长度查询函数

MPI 系统提供函数 MPI _ GET _ COUNT,可用于查询一个消息的长度,也就是消息包含的数据单元个数。

```
/* 消息长度查询函数,返回消息包含的数据单元个数。
```

```

MPI _ GET _ COUNT(status, datatype, count)
    IN      status   消息接收函数返回的状态信息
    IN      datatype 消息中数据单元类型
    OUT     count    消息包含的数据单元个数

```

```

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

```

函数 MPI_GET_COUNT 以消息接收函数 MPI_RECV 的输出参数 status 为输入参数, 统计接收消息中实际包含的类型为 datatype 的数据单元个数 count。例如, 在例 3.1 中, 可在语句:

```

CALL MPI_RECV(A(50,180),200,MPI_DOUBLE_PRECISION,1,9999,
              MPI_COMM_WORLD,NSTATUS,IERR)

```

之后, 插入函数 MPI_GET_COUNT 来获取消息包含的数据单元个数。

例 3.3 消息长度查询函数示例

```

CALL MPI_RECV(A(50,180),200,MPI_DOUBLE_PRECISION,1,9999,
              & MPI_COMM_WORLD,NSTATUS,IERR)
CALL MPI_GET_COUNT(NSTATUS,MPI_DOUBLE_PRECISION,IC,IERR)
      ! 将返回 IC=40。
或者 CALL MPI_GET_COUNT(NSTATUS,MPI_REAL,IC,IERR)
      ! 将返回 IC=80。
或者 CALL MPI_GET_COUNT(NSTATUS,MPI_BYTE,IC,IERR)
      ! 将返回 IC=320。

```

3.1.5 空进程

执行 MPI 程序的每个进程都有一个惟一的序号, 除此之外, MPI 系统还提供一个常数 MPI_PROC_NULL, 并将其定义为**空进程**(null process)的序号。如果用空进程充当消息发送函数的目标进程或消息接收函数的源进程时, 表示该函数执行的是一个空函数, 对程序的执行不产生任何影响。

MPI 提供空进程的主要目的是为了方便并行编程。例如, 利用空进程, 例 3.1 可以写为:

```

REAL*8 A(100,200)
INTEGER DEST,SOURCE,NSTATUS(MPI_STATUS_SIZE)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)
DEST = MPI_PROC_NULL
SOURCE = MPI_PROC_NULL
IF(MYRANK.EQ.1) DEST = 3
IF(MYRANK.EQ.3) SOURCE = 1
CALL MPI_SEND(A(20,120),40,MPI_DOUBLE_PRECISION,DEST,9999,MPI_COMM_
              WORLD,IERR)
CALL MPI_RECV(A(50,180),200,MPI_DOUBLE_PRECISION,SOURCE,9999,MPI_
              COMM_WORLD,NSTATUS,IERR)

```

在上例中, 当进程执行到函数 MPI_SEND 和函数 MPI_RECV 时, 如果它提供的

参数 DEST 或参数 SOURCE 等于 MPI _ PROC _ NULL, 则该函数等价于一个空函数, 不会对程序的执行产生任何影响。比较上例和例 3.1 不难发现, 该例的程序设计比较整洁。

3.2 数据类型的匹配与转换

3.2.1 数据类型匹配规则

两个进程之间的一次点对点通信可分为三个阶段:(1)消息从消息发送缓存区中拷贝给 MPI 系统, 并包装;(2)基于通信器, 消息从发送进程到达接收进程;(3)消息被接收, 并拷贝到消息接收缓存区中。MPI 系统规定, 每个阶段均必须遵守数据类型匹配规则:

- (1) 消息发送缓存区中变量的类型必须与消息发送函数说明的 MPI 数据类型匹配;
- (2) 消息发送函数说明的 MPI 数据类型必须与消息接收函数说明的 MPI 数据类型匹配;
- (3) 消息接收缓存区的变量类型必须与消息接收函数说明的 MPI 数据类型匹配。

为此, 我们必须从以下两个方面来考虑:

第一, 消息发送(或接收)缓存区的变量类型与消息发送(或接收)函数说明的 MPI 数据类型之间的匹配。它可以由 3.1 节表 3.2 列出的 Fortran 和 C 语言基本数据类型和 MPI 基本数据类型之间的一一对应关系来保证。例如, Fortran 变量类型 INTEGER 将对应 MPI 基本数据类型 MPI _ INTEGER, C 变量类型 int 将对应 MPI 基本数据类型 MPI _ Int。特别地, MPI _ BYTE 和 MPI _ PACKED 可用于匹配任何以字节为单位的内存空间, 而不管该空间所包含的变量属于何种类型。

第二, 消息发送函数说明的 MPI 数据类型与消息接收函数说明的 MPI 数据类型之间的匹配。它要求这两种类型一致。例如, 如果函数 MPI _ SEND 说明的数据类型 datatype = MPI _ REAL, 则与之相匹配的函数 MPI _ RECV 说明的数据类型 datatype 也必须是 MPI _ REAL, 但是 MPI _ PACKED 可用于匹配任意 MPI 数据类型(参考第四章)。

例 3.4 消息发送函数和消息接收函数之间正确的数据类型匹配。

```
Call MPI _ COMM _ RANK(comm, rank, ierr)
IF(rank . EQ. 0)THEN
    CALL MPI _ SEND(a(1), 10, MPI _ REAL, 1, tag, comm, ierr)
ELSE IF(rank . EQ. 1)THEN
    CALL MPI _ RECV(b(1), 15, MPI _ REAL, 0, tag, comm, status, ierr)
ENDIF
```

如果我们将上例中的消息接收函数改写为:

```
CALL MPI _ RECV(b(1), 40, MPI _ BYTE, 0, tag, comm, status, ierr)
```

则程序将出错, 因为这样将会使消息接收函数和消息发送函数说明的数据类型不匹配。若进一步将消息发送函数也改写为:

```
CALL MPI _ SEND(a(1), 40, MPI _ BYTE, 1, tag, comm, ierr)
```

则程序是正确的,且将产生与例 3.4 一致的消息传递结果。

3.2.2 数据转换

一台并行机如果由多台结构、性能均一致的处理机组成,则我们称之为同构型并行机,例如第一章中提到的各类并行机均属于同构型并行机。但是,目前也存在这样一类并行机,它由多台结构、性能不一致的处理机组成,甚至安装不有同操作系统,我们称之为异构型并行机,例如由多台性能不一致的微机构成的微机机群就是一台异构型并行机。

在异构型并行机上,进程间传输的数据有时需要进行适当的转换,具体可分为两类:第一类是数据类型的转换,例如将一个实型变量赋值给一个整型变量;第二类是数据表示的转换,例如将 32 位表示的变量赋值给一个 64 位表示的变量。如果 MPI 程序遵循了 3.2.1 节介绍的 3 个类型匹配规则,则 MPI 系统可保证任意两个进程之间的数据类型的转换。同是, MPI 系统能自动保证任意两个进程之间数据表示的转换。例如,在两台不同的处理机 A, B 上,实型数分别用 32 位和 64 位表示,进程 0 和进程 1 分别位于处理机 A 和处理机 B 上,假设进程 0 发送一个实型数据单元给进程 1,则 MPI 系统将保证进程 0 的数据(32 位表示)转变为一个 64 位表示的实型数据,存储在进程 1 的变量中。

在同构型并行机上,MPI 通信函数不需要进行数据类型和数据表示的转换,因为各台处理机结构是完全一致的。

3.3 标准模式非阻塞通信

在 3.1 节中,我们指出,阻塞式消息发送函数 MPI_SEND 具有两种不同的语义,但不管是何种语义,它可以进一步分解为提交(post-send)和完成(complete-send)两个阶段,其中,提交阶段初始化消息发送操作,当 MPI 系统开始从消息发送缓存区拷贝消息时,提交阶段的任务就已经完成;完成阶段确保 MPI 系统已经从消息发送缓存区成功地拷贝了消息,或者消息已经被成功地写入消息接收进程的消息接收缓存区中。为此,MPI 系统提供了特殊的函数,分别执行阻塞式消息发送函数 MPI_SEND 的提交和完成阶段,称之为消息发送提交函数和消息发送完成函数。于是,当进程发送消息时,它首先可以调用消息发送提交函数来提交该次消息发送操作,然后继续自己的计算工作;在进程执行计算工作的同时,MPI 系统可以并行地从消息发送缓存区中拷贝消息;最后,当进程要求更新消息发送缓存区时,可以调用消息发送完成函数来确保消息已经被成功地拷贝。这样,进程的计算就可以和消息的传递重叠起来,从而缩短了通信延迟对并行计算性能的影响。

显然,按 2.4 节和本章介绍的非阻塞通信的定义,消息发送提交函数属于非阻塞式函数,执行的是非阻塞通信,为了区别于阻塞式消息发送函数 MPI_SEND,我们称之为非阻塞消息发送函数。

同理,阻塞式消息接收函数也可以分解为提交和完成两个阶段,其中,提交阶段初始化消息接收操作后就返回,而不管该消息是否已经存在于 MPI 系统中;完成阶段确保消息被成功地拷贝到消息接收缓存区中。显然,阻塞式消息接收函数的分解也同样提供了通信与计算的重叠,可以缩小通信延迟对并行计算性能的影响。类似,MPI 系统也提供

特殊的函数,分别执行阻塞式消息接收函数 MPI_RECV 的提交和完成阶段,称之为消息接收提交函数和消息接收完成函数。显然,消息接收提交函数属于非阻塞式函数,执行的是非阻塞通信,因此我们也称之为非阻塞消息接收函数。

非阻塞消息发送函数和非阻塞消息接收函数提交的非阻塞消息发送和非阻塞消息接收通信操作,必须用它们各自对应的消息发送完成函数和消息接收完成函数来保证通信操作的正确完成,所有完成函数在 MPI 系统中统称为非阻塞通信完成函数。

非阻塞消息发送(或接收)函数可以和阻塞式消息发送(或接收)函数相互匹配,即非阻塞发送的消息可以用阻塞式消息接收函数接收,而阻塞发送的消息可以用非阻塞消息接收函数接收。

MPI 系统提供了一种特殊的数据结构:通信请求(request),用于联接非阻塞消息发送(或接收)函数与非阻塞通信完成函数,以确保非阻塞通信的提交和完成。MPI 系统规定:非阻塞消息发送(或接收)函数返回时,在参数中必须返回一个通信请求,当该通信请求作为非阻塞通信完成函数的入口参数时,表示该完成函数执行的是由该通信请求联接的非阻塞通信操作的完成,因此,MPI 系统也称非阻塞通信完成函数为通信请求完成函数,并称通信请求联接的非阻塞通信操作的完成为该通信请求的完成。

在 Fortran 语言中,充当通信请求的变量必须被说明为整型变量,而在 C 语言中,必须被说明为 MPI_Request 类型的变量。

基于通信请求的联接,非阻塞消息发送函数、非阻塞消息接收函数和通信请求完成函数就可以完成 MPI 系统定义的各类非阻塞通信操作。本节将逐一介绍这些函数。

3.3.1 非阻塞消息发送/接收函数

函数 MPI_ISEND 和函数 MPI_IRECV 分别提交一个标准模式的非阻塞消息发送操作和非阻塞消息接收操作,二者均返回一个通信请求。

```
/* 标准模式非阻塞消息发送函数,提交一个非阻塞的消息发送操作
 *
 MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
   IN      buf      消息发送缓存区初始地址
   IN      count    消息发送缓存区内单元个数
   IN      datatype 消息发送缓存区内数据单元类型
   IN      dest     接收进程的序号
   IN      tag      消息号
   IN      comm     通信器
   OUT     request  通信请求
C      int MPI_Isend( void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
                      comm, MPI_Request * request)
Fortran MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type>      BUF(*)
INTEGER      COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

/* 标准模式非阻塞消息接收函数,提交一个非阻塞的消息接收操作
 *
 MPI_IRECV(buf, count, datatype, source, tag, comm, request)
```

OUT	buf	消息接收缓存区初始地址
IN	count	消息接收缓存区内单元个数
IN	datatype	消息接收缓存区数据单元类型
IN	source	发送进程的序号
IN	tag	消息号
IN	comm	通信器
OUT	request	通信请求

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

Fortran MPI_RECV(BUF,COUNT,DATATYPE,SOURCE,TAG,COMM, REQUEST, IERROR)
 <type> BUF(*)
 INTERGER COUNT,DATATYPE,SOURCE,TAG,COMM, REQUEST, IERROR

函数 MPI_ISEND 返回时,表明 MPI 系统已经开始从消息发送缓存区中拷贝数据,但是并不隐含消息的拷贝已经完成。因此,在调用消息发送完成函数之前,对消息发送缓存区的任何更新都将是不安全的;即任何对消息发送缓存区的修改可能破坏正在发送的消息。

函数 MPI_RECV 的语义较复杂一些,它包含两个方面的含义:第一,如果接收的消息不在 MPI 系统中,则函数立即返回;第二,如果接收的消息已经存在于 MPI 系统中,则函数接收该消息,并将消息写入消息接收缓存区之后才返回。由此可知,在调用消息接收完成函数之前,对消息接收缓存区的任何访问也可能是不安全的。

例 3.5 非阻塞消息发送/接收函数示例:并行矩阵乘(参考例 3.2)

```

REAL * 8 D(N,LP)
INTEGER ISREQ,IRREQ ! 非阻塞消息发送/接收函数返回的通信请求。
!
! 并行矩阵乘开始。
DO 100 K=1, P
    ! 非阻塞发送矩阵 B 的子块。
    CALL MPI_ISEND(B,N * LP,MPI_DOUBLE_PRECISION,MYLEFT,K * 100,MPI_COMM
    _WORLD,ISREQ,IERR)
    ! 非阻塞接收矩阵 B 的子块到矩阵 D。
    CALL MPI_RECV(D,N * LP,MPI_DOUBLE_PRECISION,MYLEFT,K * 100,MPI_COMM
    _WORLD,IRREQ,IERR)
    IC = IC + LP
    IF(IC .GE. L) IC = IC - L
    ! 局部子矩阵乘。
    ! ..... C = A * B, 忽略 .....
    IF(K.EQ.P) GOTO 100
    CALL MPI_WAIT(IRREQ,NSTATUS,IERR)
    ! 阻塞等待与 IRREQ 联接的非阻塞消息接收函数的完成。
    CALL MPI_WAIT(ISREQ,NSTATUS,IERR)

```

```

! 阻塞等待与 ISREQ 联接的非阻塞消息发送函数的完成。
DO J = 1, LP
  DO I = 1, N
    B(I,J) = D(I,J)
  ENDDO
ENDDO
100 CONTINUE

```

在上例中, 函数 MPI_WAIT 是通信请求完成函数, 它的第一个参数是联接非阻塞通信操作的通信请求, 第 2 个参数是类似于函数 MPI_RECV 的接收返回状态信息。当且仅当第 1 个参数给出的通信请求联接的非阻塞通信操作安全完成之后, 函数 MPI_WAIT 才返回。我们在下节将讨论该函数。

3.3.2 通信请求完成函数

MPI 系统提供一系列的通信请求完成函数, 用于完成通信请求联接的非阻塞消息发送或消息接收通信操作。同时, MPI 系统也提供相应的通信请求查询函数, 用于查询某些非阻塞消息发送或接收通信操作是否已经安全完成。

```
/* 非阻塞通信完成函数, 确保与某个通信请求相联接的非阻塞通信操作的安全完成。
```

```

MPI_WAIT(request, status)
  INOUT request      通信请求
  OUT   status       接收返回状态信息
C     int MPI_Wait(MPI_Request *request, MPI_Status *status)
Fortran MPI_WAIT(REQUEST, STATUS, IERROR)
        INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

```

当且仅当通信请求 request 联接的非阻塞通信操作已完成, 函数 MPI_WAIT 才返回, 并自动将 request 的值改为 MPI 常数 MPI_REQUEST_NULL。类似于阻塞式消息接收函数 MPI_RECV, 数组 status 中将包含相应的接收返回状态信息。MPI_WAIT 的应用实例, 可参考例 3.5 的语句:

```

CALL MPI_WAIT(IREQ, NSTATUS, IERR)
CALL MPI_WAIT(ISREQ, NSTATUS, IERR)

```

```
/* 非阻塞通信完成函数, 确保至少存在一个与给定序列中某个通信请求相联接的
/* 非阻塞通信操作已经安全完成。
```

```

MPI_WAITANY(count, array_of_requests, index, status)
  IN   count          通信请求序列包含的通信请求个数
  INOUT array_of_requests  通信请求序列
  OUT  index          已完成的通信请求在序列中的位置
  OUT  status          接收返回状态信息
C     int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *

```

```

        status)
Fortran MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX,
           STATUS(MPI_STATUS_SIZE), IERROR

```

当且仅当序列 array_of_requests 包含的 count 个元素中, 至少存在一个通信请求, 它所联接的非阻塞通信操作已完成, 函数 MPI_WAITANY 才返回, 并输出该通信请求在序列中的位置 index。如果有多个通信请求相联接的非阻塞通信操作均已完成, 则 MPI_WAITANY 选择其中的任意一个返回, 并自动将 array_of_requests(index) 的值改为 MPI_REQUEST_NULL。类似, 数组 status 包含相应的接收返回状态信息。

```

/* 非阻塞通信完成函数, 确保与给定序列中所有通信请求各自联接的
/* 非阻塞通信操作的全部安全完成。

```

```

MPI_WAITALL(count, array_of_requests, array_of_statuses)
    IN      count          通信请求序列包含的通信请求个数
    INOUT   array_of_requests  通信请求序列
    OUT     array_of_statuses 接收返回状态信息序列
C     int MPI_Waitall(int count, MPI_Request *array_of_requests,
                     MPI_Status *array_of_statuses)
Fortran MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS,
                     ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR,
           ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *)

```

当且仅当序列 array_of_requests 包含的所有通信请求各自联接的非阻塞通信操作已经全部完成, 函数 MPI_WAITALL 才返回, 并自动将 array_of_requests 所有元素的值改为 MPI_REQUEST_NULL。此外, 序列 array_of_statuses 的每个元素将包含相应的接收返回状态信息。

例 3.6 非阻塞消息完成函数示例: 并行矩阵乘(参考例 3.5)

```

INTEGER IREQ(2)                                ! 通信请求序列。
INTEGER NSTATUS(MPI_STATUS_SIZE, 2)      ! 消息接收返回状态信息序列。
DO 100 K=1, P
    ! 非阻塞发送矩阵 B 的子块。
    CALL MPI_ISEND(B, N * LP, MPI_DOUBLE_PRECISION, MYLEFT, K * 100,
                   MPI_COMM_WORLD, IREQ(1), IERR)
    ! 非阻塞接收矩阵 B 的子块到矩阵 D。
    CALL MPI_IRECV(D, N * LP, MPI_DOUBLE_PRECISION, MYLEFT, K * 100,
                   MPI_COMM_WORLD, IREQ(2), IERR)
    ! 局部子矩阵乘…… C = A * B, 忽略……。
    IF(K.EQ.P) GOTO 100
    CALL MPI_WAITALL(2, IREQ, NSTATUS, IERR)

```

! 阻塞等待与序列 IREQ 各元素联接的非阻塞通信操作的全部完成。

!

! 局部子矩阵赋值……B=D……忽略。

100 CONTINUE

/* 非阻塞通信完成函数, 确保与给定序列中某些通信请求各自联接的
/* 非阻塞通信操作的安全完成。

MPI_WAITsome(incount, array_of_requests, outcount, array_of_indices,
array_of_statuses)

IN	incount	通信请求序列包含的通信请求个数
INOUT	array_of_requests	通信请求序列
OUT	outcount	序列中已完成的通信请求的个数
OUT	array_of_indices	完成的通信请求在序列中的位置
OUT	array_of_statuses	接收返回状态信息序列

C int MPI_Waitsome(int incount, MPI_Request * array_of_requests, int * outcount,
int * array_of_indices, MPI_Status * array_of_statuses)

Fortran MPI_WAITsome(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR)
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT,
ARRAY_OF_INDICES(*), IERROR,
ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *)

当且仅当序列 array_of_requests 包含的所有通信请求各自联接的非阻塞通信操作中, 存在一个以上的通信请求已经完成时, 函数 MPI_WAITsome 才返回。参数 outcount 返回联接的非阻塞通信操作已完成的通信请求个数, array_of_indices 记录这些请求在序列中的相应位置, 并自动将 array_of_requests 中对应元素的值改为 MPI_REQUEST_NULL。序列 array_of_statuses 的元素将包含对应序列位置的消息接收返回状态信息。

/* 非阻塞通信完成查询函数, 查询某个通信请求的联接的非阻塞通信操作
/* 是否已安全完成。

MPI_TEST(request, flag, status)

INOUT	request	通信请求
OUT	flag	true 表示该通信请求已完成
OUT	status	接收返回状态信息

C int MPI_Test(MPI_Request * request, int * flag, MPI_Status * status)

Fortran MPI_TEST(REQUEST, FLAG, STATUS, IERROR)

LOGICAL FLAG

INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

如果通信请求 request 联接的非阻塞通信操作已安全完成, 则返回 flag = true, 且

status 中包含相应的消息接收状态信息，同时自动将 request 的值改变为 MPI _ REQUEST _ NULL；否则，返回 flag = false，status 中不包含任何信息。

```
/* 非阻塞通信完成查询函数，查询是否至少存在一个与给定序列中某个通信请求  
/* 相联接的非阻塞通信操作已经安全完成。
```

```
MPI _ TESTANY(count, array _ of _ requests, index, flag, status)  
    IN      count          通信请求序列包含的通信请求个数  
    INOUT   array _ of _ requests 通信请求序列  
    OUT     index          已完成的通信请求在序列中的位置  
    OUT     flag           true 表示至少存在一个通信请求已完成  
    OUT     status         接收返回状态信息
```

```
C      int MPI _ Testany(int count, MPI _ Request * array _ of _ requests, int * index,  
                      int * flag, MPI _ Status * status)
```

```
Fortran MPI _ TESTANY(COUNT, ARRAY _ OF _ REQUESTS, INDEX, FLAG,  
                      STATUS, IERROR)
```

```
LOGICAL FLAG  
INTEGER COUNT, ARRAY _ OF _ REQUESTS(*), INDEX, IERROR,  
        STATUS(MPI _ STATUS _ SIZE)
```

如果序列 array _ of _ requests 包含的通信请求各自联接的非阻塞通信操作中至少存在一个通信请求已经完成，则返回 flag = true，并从中选择任意一个通信请求来完成。参数 index 记录该通信请求在序列中的位置，status 包含相应消息接收返回的状态信息，并自动将 array _ of _ requests(index) 的值改为 MPI _ REQUEST _ NULL；否则，返回 flag = false，status 中不包含任何信息。

```
/* 非阻塞通信完成查询函数，查询给定序列中所有通信请求各自联接的  
/* 非阻塞通信操作是否已经全部安全完成。
```

```
MPI _ TESTALL(count, array _ of _ requests, flag, array _ of _ statuses)  
    IN      count          通信请求序列包含的通信请求个数  
    INOUT   array _ of _ requests 通信请求序列  
    OUT     flag           true 表示所有通信请求已经全部完成  
    OUT     array _ of _ statuses 接收返回状态信息序列
```

```
C      int MPI _ Testall(int count, MPI _ Request * array _ of _ requests, int * flag,  
                        MPI _ Status * array _ of _ statuses)
```

```
Fortran MPI _ TESTALL(COUNT, ARRAY _ OF _ REQUESTS, FLAG,
```

```
                      ARRAY _ OF _ STATUSES, IERROR)
```

```
LOGICAL FLAG  
INTEGER COUNT, ARRAY _ OF _ REQUESTS(*), IERROR,  
        ARRAY _ OF _ STATUSES(MPI _ STATUS _ SIZE, *)
```

如果序列 array _ of _ requests 包含的所有通信请求各自联接的非阻塞通信操作已经

全部完成，则返回 flag = true, array _ of _ statuses 包含相应的接收状态信息，并自动将 array _ of _ requests 所有元素的值改为 MPI _ REQUEST _ NULL; 否则，返回 flag = false, array _ of _ statuses 中不包含任何信息(即为 0)。

```
/* 非阻塞通信完成查询函数, 查询与给定序列中某些通信请求各自联接的  
/* 非阻塞通信操作是否已经安全完成。
```

```
MPI _ TESTSOME(incount, array _ of _ requests, outcount, array _ of _ indices,  
                array _ of _ statuses)  
IN      incount          通信请求序列包含的通信请求个数  
INOUT   array _ of _ requests    通信请求序列  
OUT     outcount          序列中已完成的通信请求个数  
OUT     array _ of _ indices    已完成的通信请求在序列中的位置  
OUT     array _ of _ statuses   接收返回状态信息序列  
C      int MPI _ Testsome(int incount, MPI _ Request * array _ of _ requests,  
                           int * outcount, int * array _ of _ indices,  
                           MPI _ Status * array _ of _ statuses)  
Fortran MPI _ TESTSOME(INCOUNT, ARRAY _ OF _ REQUESTS, OUTCOUNT,  
                      ARRAY _ OF _ INDICES, ARRAY _ OF _ STATUSES, IERROR)  
INTEGER INCOUNT, ARRAY _ OF _ REQUESTS(*), OUTCOUNT,  
        ARRAY _ OF _ INDICES(*), IERROR,  
        ARRAY _ OF _ STATUSES(MPI _ STATUS _ SIZE, *)
```

如果序列 array _ of _ requests 包含的所有通信请求各自联接的非阻塞通信操作中，至少存在一个以上通信请求已经完成，则返回 flag = true, 参数 outcount 记录当前已完成的通信请求的个数, array _ of _ indices 记录通信请求在序列中的位置, 且自动将 array _ of _ requests 中对应位置上的元素的值改为 MPI _ REQUEST _ NULL, array _ of _ statuses 包含相应的消息接收状态信息。

3.3.3 消息查询函数

MPI 系统提供多个辅助函数，用于查询某个特定消息是否已经存在于 MPI 系统中，并利用这些函数返回的接收状态信息，确定是否进一步接收该消息，或以何种方式进行接收。

```
/* 阻塞式消息查询函数, 阻塞地查询某个特定消息是否已在 MPI 系统中。
```

```
MPI _ PROBE(source, tag, comm, status)  
IN      source    发送消息进程的序号  
IN      tag       消息号  
IN      comm      通信器  
OUT     status     接收返回状态信息  
C      int MPI _ Probe(int source, int tag, MPI _ Comm comm, MPI _ Status * status)  
Fortran MPI _ PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
```

```

INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
IERROR

```

函数 MPI_PROBE 查询某个特定消息(source, tag, comm)是否已在 MPI 系统中, 如果已经存在则函数返回, 参数 status 中将保留接收返回状态信息; 否则, 阻塞地等待发送该消息的进程提交该消息给通信器之后才返回。

/* 非阻塞式消息查询函数, 非阻塞地查询某个特定消息是否已在 MPI 系统中。

```

MPI_IPROBE(source, tag, comm, flag, status)
    IN      source          发送消息进程的序号
    IN      tag             消息号
    IN      comm            通信器
    OUT     flag            true 表示消息已在 MPI 系统中
    OUT     status          接收返回状态
C      int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
Fortran MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

```

函数 MPI_IPROBE 非阻塞地查询某个特定消息(source, tag, comm)是否已存在于 MPI 系统中: 如果已存在, 则返回 flag = true, status 中保留相应的接收返回状态信息; 否则, 返回 flag = false, status 中不包含任何信息。

例 3.7 利用消息查询函数等待一个消息

```

CALL MPI_COMM_RANK(COMM, RANK, IERR)
IF (RANK.EQ.0) THEN
    CALL MPI_SEND(I, 1, MPI_INTEGER, 2, 0, COMM, IERR)
ELSE IF( RANK.EQ.1) THEN
    CALL MPI_SEND(X, 1, MPI_REAL, 2, 0, COMM, IERR)
ELSE IF( RANK.EQ.2) THEN
    DO I=1,2
        CALL MPI_PROBE(MPI_ANY_SOURCE, 0, COMM, STATUS, IERR)
        IF( STATUS(MPI_SOURCE).EQ.0) THEN
            CALL MPI_RECV(I, 1, MPI_INTEGER, 0, 0, COMM, STATUS, IERR)
        ELSE
            CALL MPI_RECV(X, 1, MPI_REAL, 1, 0, COMM, STATUS, IERR)
        ENDIF
    ENDDO
ENDIF

```

在该例中, 进程 0 向进程 2 发送一个整型数, 进程 1 向进程 2 发送一个实型数, 进程 2 首先利用查询函数 MPI_PROBE 阻塞地等待来自其他进程的任何消息, 然后根据函数

MPI _ PROBE 返回的信息,采用不同的消息接收方式来接收不同类型的数据。

3.4 有限缓存区资源对消息传递的影响

在本章 3.1 节中,我们介绍了阻塞式消息发送函数 MPI _ SEND 和消息接收函数 MPI _ RECV 的含义,但没有介绍 MPI 系统内部有限的缓存区资源对这两个函数执行行为的影响。这里,利用前面各节介绍的阻塞式和非阻塞式通信函数,我们将仔细讨论 MPI 系统内部有限的缓存区资源可能带来的问题,以及避免办法。

例 3.8 MPI 消息缓存区大小对阻塞式消息发送/接收函数执行行为的影响

```
CALL MPI _ COMM _ RANK(COMM, RANK, IERR)
IF (RANK.EQ.0) THEN
    CALL MPI _ SEND(SENDBUF, COUNT, MPI _ REAL, 1, TAG, COMM, IERR)
    CALL MPI _ RECV(RECVBUF, COUNT, MPI _ REAL, 1, TAG, COMM, STATUS, IERR)
ELSE IF(RANK.EQ.1) THEN
    CALL MPI _ SEND(SENDBUF, COUNT, MPI _ REAL, 0, TAG, COMM, IERR)
    CALL MPI _ RECV(RECVBUF, COUNT, MPI _ REAL, 0, TAG, COMM, STATUS, IERR)
ENDIF
```

在例 3.8 中,进程 0 向进程 1 发送一个大小为 $SIZE = COUNT * SIZEOF(MPI _ REAL)$ 个字节的消息;同时,进程 1 也向进程 0 发送一个大小一致的消息。此时,程序的执行是不安全的,可能产生两种不同的结果:

- (1) 当消息的大小 SIZE 小于 MPI 系统为每个进程设置的最大消息缓存区 MPI _ BUFFER _ SIZE 时,例中的消息传递能够顺利完成,此时,MPICH 系统将为进程缓存发送的消息;
- (2) 当消息的大小 SIZE 大于 MPI 系统为每个进程设置的最大消息缓存区 MPI _ BUFFER _ SIZE 时,各进程执行函数 MPI _ SEND 将被阻塞,等待与之匹配的函数 MPI _ RECV 接收该消息。但是,进程 0 和进程 1 都将被阻塞,无法执行消息接收函数 MPI _ RECV。这样,程序的执行将会出现各个进程由于相互等待而无法继续运行的情况,我们称之为死锁(deadlock)。

给定一台具体并行机,如果它提供的 MPI 系统具有足够大的缓存区,则例 3.8 可以正确运行,但如果它提供的 MPI 系统没有足够的缓存区,则例 3.8 就可能被死锁。为此,MPICH 系统称类似于例 3.8 的 MPI 程序是不安全的(unsafe)。一个 MPI 程序,如果它是不安全的,则可移植性将受到很大的影响。因此,在实际应用中,我们应该尽量避免这种情况。那么如何来避免呢?目前,有如下的几种办法:

第一,交换进程 1 的消息发送/接收函数语句的位置,即:

```
IF (RANK.EQ.0) THEN
    CALL MPI _ SEND(SENDBUF, COUNT, MPI _ REAL, 1, TAG, COMM, IERR)
    CALL MPI _ RECV(RECVBUF, COUNT, MPI _ REAL, 1, TAG, COMM, STATUS, IERR)
ELSE IF(RANK.EQ.1) THEN
```

```

    CALL MPI_RECV(RECVBUF, COUNT, MPI_REAL, 0, TAG, COMM, STATUS, IERR)
    CALL MPI_SEND(SENDBUF, COUNT, MPI_REAL, 0, TAG, COMM, IERR)
ENDIF

```

第二,采用函数 MPI_SENDRECV,因为该函数同时执行一个阻塞式消息发送函数和一个阻塞式消息接收函数,因此不会存在次序上的依赖性。具体修改为:

```

IF (RANK.EQ.0) THEN
    CALL MPI_SENDRECV(SENDBUF, COUNT, MPI_REAL, 1, TAG,
                      RECVBUF, COUNT, MPI_REAL, 1, TAG, COMM, STATUS, IERR)
ELSE IF(RANK.EQ.1) THEN
    CALL MPI_SENDRECV(SENDBUF, COUNT, MPI_REAL, 0, TAG,
                      RECVBUF, COUNT, MPI_REAL, 0, TAG, COMM, STATUS, IERR)
ENDIF

```

第三,采用非阻塞消息发送函数,即:

```

IF (RANK.EQ.0) THEN
    CALL MPI_ISEND(SENDBUF, COUNT, MPI_REAL, 1, TAG, COMM, REQ, IERR)
    CALL MPI_RECV(RECVBUF, COUNT, MPI_REAL, 1, TAG, COMM, STATUS, IERR)
    CALL MPI_WAIT(REQ, STATUS, IERR)
ELSE IF(RANK.EQ.1) THEN
    CALL MPI_ISEND(SENDBUF, COUNT, MPI_REAL, 0, TAG, COMM, REQ, IERR)
    CALL MPI_RECV(RECVBUF, COUNT, MPI_REAL, 0, TAG, COMM, STATUS, IERR)
    CALL MPI_WAIT(REQ, STATUS, IERR)
ENDIF

```

第四,采用非阻塞消息接收函数,即:

```

IF (RANK.EQ.0) THEN
    CALL MPI_IRECV(RECVBUF, COUNT, MPI_REAL, 1, TAG, COMM, REQ, IERR)
    CALL MPI_SEND(SENDBUF, COUNT, MPI_REAL, 1, TAG, COMM, IERR)
    CALL MPI_WAIT(REQ, STATUS, IERR)
ELSE IF(RANK.EQ.1) THEN
    CALL MPI_IRECV(RECVBUF, COUNT, MPI_REAL, 0, TAG, COMM, REQ, IERR)
    CALL MPI_SEND(SENDBUF, COUNT, MPI_REAL, 0, TAG, COMM, IERR)
    CALL MPI_WAIT(REQ, STATUS, IERR)
ENDIF

```

第五,采用缓存区模式的阻塞式消息发送函数 MPI_BSEND,我们将在 3.7 节中介绍。

3.5 持久通信请求

考察例 3.2 中的这部分代码:

```

!
! 并行矩阵乘开始。
DO 100 K=1, P
.....
```

```

!
! 消息传递,依次交换矩阵 B 的子块
CALL MPI_SEND(B, N * LP, MPI_DOUBLE_PRECISION, MYLEFT, K * 100,
              MPI_COMM_WORLD, IERR)
CALL MPI_RECV(B, N * LP, MPI_DOUBLE_PRECISION, MYRIGHT, K * 100,
              MPI_COMM_WORLD, NSTATUS, IERR)
!
100 CONTINUE
!
```

可以看出,它要求一个循环的不同迭代,重复执行具有相同参数的消息发送和消息接收通信操作。其实,在这些函数的重复执行过程中,有一部分工作是相互重叠的。为此,MPI系统提供了持久的(persistent)通信功能,可用于增强此类重复通信操作的性能。具体是,在进入循环之前,进程首先调用函数MPI_SEND_INIT和函数MPI_RECV_INIT分别创建一个联接消息发送和消息接收的通信请求;然后,利用该持久通信请求重复地提交和完成消息的发送和接收,从而减少进程的通信开销。为了区别于3.3节非阻塞通信介绍的通信请求,我们称该类通信请求为持久通信请求,并称3.3节中的通信请求为非阻塞通信请求。

/* 消息发送持久通信请求创建函数,创建一个消息发送持久通信请求。

C	MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)
	IN buf 消息发送缓存区初始地址
	IN count 数据单元个数
	IN datatype 数据单元类型
	IN dest 接收进程的序号
	IN tag 消息号
	IN comm 通信器
	OUT request 持久通信请求
Fortran	MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
	<type> BUF(*)
	INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

/* 消息接收持久通信请求创建函数,创建一个消息接收持久通信请求。

C	MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)
	IN buf 消息接收缓存区初始地址
	IN count 数据单元个数
	IN datatype 数据单元类型
	IN source 发送进程的序号
	IN tag 消息号

IN	comm	通信器
OUT	request	持久通信请求

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

Fortran MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
 REQUEST, IERROR)

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

函数 MPI_SEND_INIT 和函数 MPI_RECV_INIT 分别创建一个标准模式的非阻塞消息发送和非阻塞消息接收持久通信请求，并将消息(buf, count, datatype, source/dest, tag, comm)捆绑到该请求中，只要以后涉及到该消息的发送或接收，则可调用函数 MPI_START 来启动该次非阻塞通信，将消息发送缓存区包含的数据发送出去或将消息接收到消息接收缓存区中。

请注意，捆绑到持久通信请求的消息，实际上是该消息的地址分布空间，而不是该消息在函数 MPI_SEND_INIT 调用之前的实际内容。也就是说，持久通信请求创建后，任何对消息发送或消息接收缓存区内数据号的修改，均将包含在下一次函数 MPI_START 启动的非阻塞消息发送或阻塞消息接收操作中。

/* 持久通信请求提交函数，提交一个持久通信请求。

 MPI_Start(request)
 INOUT request 持久通信请求
 C int MPI_Start(MPI_Request * request)
 Fortran MPI_START (REQUEST, IERROR)
 INTEGER REQUEST, IERROR

/* 持久通信请求提交函数，提交一个给定序列包含的所有持久通信请求。

 MPI_Startall(count, array_of_requests)
 IN count 持久通信序列包含的通信请求个数
 INOUT array_of_requests 持久通信请求序列
 C int MPI_Startall(int count, MPI_Request * array_of_requests)
 Fortran MPI_STARTALL (COUNT, ARRAY_OF_REQUESTS, IERROR)
 INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR

持久通信请求的提交隐含了它所联接的非阻塞消息发送或消息接收函数的提交，同理，持久通信请求的完成隐含了它所联接的非阻塞消息发送或消息接收函数的完成。持久通信请求的完成可与非阻塞通信请求一样，由 3.3.2 节介绍的各类非阻塞通信完成函数来进行。但是，与非阻塞通信请求不同的是，持久通信请求完成后，其变量的值不变，敬请读者注意。

例 3.9 持久通信请求示例: 并行矩阵乘(参考例 3.2)

```
INTEGER IREQ(2)                                ! 通信请求序列。
INTEGER NSTATUS(MPI_STATUS_SIZE, 2)            ! 消息接收返回状态信息。
!
! 创建持久通信请求。
CALL MPI_SEND_INIT(B, N * LP, MPI_DOUBLE_PRECISION, MYLEFT, 100,
                    MPI_COMM_WORLD, IREQ(1), IERR)
CALL MPI_RECV_INIT(D, N * LP, MPI_DOUBLE_PRECISION, MYRIGHT, 100,
                    MPI_COMM_WORLD, IREQ(2), IERR)
!
DO 100 K=1, P
    CALL STARTALL(2, IREQ, IERR)    ! 提交非阻塞消息发送/接收通信操作。
    ! 局部子矩阵乘…… C=A*B, 忽略 ……
    IF(K.EQ.P) GOTO 100
    CALL MPI_WAITALL(2, IREQ, NSTATUS, IERR)
        ! 完成序列 IREQ 中元素联接的非阻塞消息发送/接收通信操作, 且 IERR 元素的
        ! 值不变。
    !
    ! 局部子矩阵赋值…… B=D…… 忽略。
100 CONTINUE
```

3.6 通信请求的释放与取消

前面, 我们介绍了 MPI 系统的两种通信请求: 非阻塞通信请求和持久通信请求, 它们可用相同的函数来完成。其实, MPI 系统可以创建一个通信请求, 反之也可以释放或强行取消一个已存在的通信请求, 具体由如下函数来实现。

```
/* 通信请求取消函数, 在通信请求联接的非阻塞通信操作完成后, 释放该请求。
```

```
MPI_REQUEST_FREE(request)
    INOUT    request    通信请求
C     int MPI_Request_free(MPI_Request * request)
Fortran MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

函数 MPI_REQUEST_FREE 释放通信请求 `request` 和它所联接的非阻塞通信操作, 并将其值自动改为 `MPI_REQUEST_NULL`, 但不影响已提交的非阻塞通信操作, 也就是说, 如果 `request` 联接的某次非阻塞通信操作已提交, 但没完成, 则必须延放到该次操作安全完成后, 才能释放 `request`。

```

/* 通信请求强行取消函数, 强行取消一个通信请求, 而不管目前 MPI 系统中是否
/* 存在已提交但没完成的非阻塞通信。

    MPI_CANCEL(request)
        IN      request      通信请求
C      int MPI_Cancel(MPI_Request *request)
Fortran MPI_CANCEL(REQUEST, IERROR)
        INTEGER REQUEST, IERROR

```

函数 MPI_CANCEL 强行取消通信请求 request 和它所联接的非阻塞通信操作, 它与函数 MPI_REQUEST_FREE 的不同之处在于, 不管当前 MPI 系统内是否存在由该请求提交的、且还没有完成的非阻塞通信, 一律强行取消该通信请求, 并将其值自动改为 MPI_REQUEST_NULL。因此, 该函数可能使正在执行的由该通信请求联接的非阻塞通信变得不安全。

```

/* 通信请求查询函数, 查询一个通信请求是否已被取消。

    MPI_TEST_CANCELLED(status, flag)
        IN      status      接收返回状态信息
        OUT     flag       true 表示该请求已被取消
C      int MPI_Test_cancelled(MPI_Status *status, int *flag)
Fortran MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
        LOGICAL FLAG
        INTEGER STATUS(MPI_STATUS_SIZE), IERROR

```

函数 MPI_TEST_CANCELLED 用于查询某个通信请求是否已被成功取消。如果已取消, 则返回 flag = true ; 否则, 返回 flag = false 。

例 3.10 通信请求取消函数应用示例

```

CALL MPI_COMM_RANK(COMM, RANK, IERR)
IF (RANK.EQ.0)THEN
    CALL MPI_SEND(A, 1, MPI_CHAR, 1, TAG, COMM, IERR)
ELSE IF (RANK.EQ.1) THEN
    CALL MPI_IRecv(A, 1, MPI_CHAR, 0, TAG, COMM, IREQ, IERR)
    CALL MPI_CANCEL(IREQ, IERR)
    CALL MPI_WAIT(IREQ, STATUS, IERR)
    CALL MPI_TEST_CANCELLED(STATUS, FLAG, IERR)
    IF(FLAG) CALL MPI_RECV(A, 1, MPI_CHAR, 0, TAG, COMM, IREQ, IERR)
        ! 通信请求已经被成功地取消
ENDIF

```

在上例中, 进程 0 发送一个字符给进程 1, 进程 1 非阻塞接收该字符之后, 强行取消

非阻塞接收返回的通信请求，并等待通信请求 IREQ 的完成。这里，IREQ 有两种可能值，第一是 $\text{IREQ} = \text{MPI_REQUEST_NULL}$ ，表示该通信请求已经被成功地取消；第二是 $\text{IREQ} \neq \text{MPI_REQUEST_NULL}$ ，表示该通信请求还没有被成功地取消；最后，程序测试该通信请求是否已经被取消，如果已经被取消，则调用函数 MPI_RECV 阻塞地接收进程 0 发送来的字符。

3.7 其他通信模式

在本章前面各节，我们介绍了 MPI 系统定义的标准点对点通信函数，这里，我们再进一步介绍其他三类模式的点对点通信函数，它们是对标准模式的补充。

缓存模式(Buffered-mode**)** 是对标准模式消息发送函数的补充，它规定由进程向 MPI 系统提供缓存消息的内存空间。此时，消息发送函数可以被局部地提交和完成，与相匹配的消息接收函数是否存在没有任何关系。因此，基于缓存模式的消息发送函数是局部函数。

同步模式(Synchronous-mode) 是对标准模式消息发送函数的补充，它类似于电话的协议，规定：直到消息接收函数被提交，两个通信进程取得联接之后，消息才开始发送，且直到消息开始被接收，消息发送函数才能返回。显然，同步模式的消息发送函数属于非局部函数。

就绪模式(Ready-mode) 是标准模式消息发送函数的补充，它要求在提交消息发送函数之前，先提交与之相匹配的消息接收函数，且直到消息被写入消息接收缓存区后，消息发送函数才返回。如果消息发送函数被提交后，与之相匹配的消息接收函数还没有被提交，则消息的发送可能是不正确的。就绪模式的消息发送函数和消息接收函数之间的这种相互制约关系，必须由用户在组织并行程序设计时保证，MPI 系统将不会对此提供任何帮助。显然，就绪模式可用于取消进程间通信的同步机制，提高通信性能。

类似于标准模式的阻塞、非阻塞和持久通信，上述三种通信模式的消息发送函数也可分为阻塞、非阻塞和持久通信三类，并且，无论哪种类型的消息发送，均可以用前面介绍的阻塞或非阻塞消息接收函数来接收。为了便于区别，我们用前缀 B 表示缓存模式，S 表示同步模式，R 表示就绪模式，来分别讨论不同模式下的消息发送函数。除非特别说明，以下函数中的各参数含义与标准模式一致。

本节将重点介绍缓存模式的消息发送函数，至于其他两种模式，由于实际并行应用程序很少涉及它们，故本书只给出各个函数的说明与含义，而不再仔细讨论。有兴趣的读者，可以进一步查看 MPI 系统的联机文档，或其他参考文献。

3.7.1 阻塞式消息发送函数

/* 缓存模式消息发送函数，执行一个缓存模式的阻塞式消息发送操作。

MPI_BSEND(buf, count, datatype, dest, tag, comm)
IN buf 消息发送缓存区初始地址
IN count 消息发送缓存区包含的数据单元个数

IN	datatype	数据单元类型
IN	dest	接收进程的序号
IN	tag	消息号
IN	comm	通信器

C int MPI_Bsend(void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Fortran MPI_BSEND(BUF,COUNT,DATATYPE,DEST,TAG,COMM,IERROR)
 <type> BUF(*)
 INTEGER COUNT,DATATYPE,DEST,TAG,COMM,IERROR

由 3.4 节, 我们知道, 如果 MPI 系统为消息提供的缓存区不足以大, 则阻塞式消息发送函数 MPI_SEND 必须等到与之相匹配的消息接收函数接收该消息之后才能返回。函数 MPI_BSEND 利用函数 MPI_BUFFER_ATTACH 提交给 MPI 系统的应用程序内存空间来缓存消息, 较好地改进了函数 MPI_SEND 的功能。只要函数 MPI_BUFFER_ATTACH 提交足够大的内存空间, 则函数 MPI_BSEND 的执行将不会被阻塞地等待与之相匹配的消息接收函数的出现, 因此, 函数 MPI_BSEND 属于局部函数。

函数 MPI_BSEND 形式、语义和各参数含义与函数 MPI_SEND 是一致的, 唯一不同的是, 函数 MPI_BSEND 将使用函数 MPI_BUFFER_ATTACH 提交的应用程序内存空间, 而不是由 MPI 系统来缓存消息。

这里, 我们有必要详细介绍与函数 MPI_BSEND 密切相关的两个函数 MPI_BUFFER_ATTACH 和 MPI_BUFFER_DETACH。其中, 函数 MPI_BUFFER_ATTACH 可用于将应用程序的内存空间提交给 MPI 系统, 充当缓存模式消息发送函数 MPI_BSEND 的消息缓存区, 而函数 MPI_BUFFER_DETACH 可用于释放已经提交给 MPI 系统的消息缓存区, 使之返回到应用程序的内存空间。

/* 缓存区提交函数, 提交应用程序的内存空间给 MPI 系统,
/* 用于缓存以缓存模式发送的消息。

MPI_BUFFER_ATTACH(buffer, size)		
IN	buffer	缓存区初始地址
IN	size	缓存区大小(字节为单位)

C int MPI_Buffer_attach(void * buffer, int size)

Fortran MPI_BUFFER_ATTACH(BUFFER,SIZE,IERROR)
 <type> BUFFER(*)
 INTEGER SIZE,IERROR

/* 缓存区释放函数, 释放应用程序提交给 MPI 系统用于缓存消息的内存空间。

MPI_BUFFER_DETACH(buffer, size)		
IN	buffer	缓存区初始地址
IN	size	缓存区大小(字节为单位)

C int MPI_Buffer_detach(void * buffer, int size)

Fortran MPI_BUFFER_DETACH(BUFFER,SIZE,IERROR)

```

< type >      BUFFER( * )
INTEGER      SIZE, IERROR

```

例 3.11 缓存模式消息传递示例：并行矩阵乘(参考例 3.2)

```

CHARACTER BUF(8 * N * LP + 2 * MPI _ BSEND _ OVERHEAD)
! 初始化, 变量赋值, 并行子矩阵赋初值(忽略)。
! 提交应用程序内存空间给 MPI 系统, 充当消息缓存区。
CALL MPI _ BUFFER _ ATTACH(BUF, 8 * N * LP + 2 * MPI _ BSEND _ OVERHEAD, IERR)
! 并行矩阵乘开始。
DO 100 K = 1, P
! 局部子矩阵乘(忽略)。
! 消息传递, 交换矩阵 B 的子块。
    CALL MPI _ BSEND(B, N * LP, MPI _ DOUBLE _ PRECISION, MYLEFT, K * 100,
                      MPI _ COMM _ WORLD, IERR)
    CALL MPI _ RECV(B, N * LP, MPI _ DOUBLE _ PRECISION, MYRIGHT, K * 100,
                      MPI _ COMM _ WORLD, NSTATUS, IERR)
100 CONTINUE
! 将提交给 MPI 系统作消息缓存区的内存空间释放给应用程序。
CALL MPI _ BUFFER _ DETACH(BUF, 8 * N * LP + 2 * MPI _ BSEND _ OVERHEAD, IERR)
! 退出 MPI 系统(忽略)。

```

在上例中, 我们要求函数 MPI _ BUFFER _ ATTACH 提交的消息缓存区比实际将要发送的消息多 $2 \times \text{MPI_BSEND_OVERHEAD}$ 个字节 ($\text{MPI_BSEND_OVERHEAD}$ 一般为 512 个字节)。这些多余的空间将被 MPI 系统用于管理函数 MPI _ BUFFER _ ATTACH 提交的缓存区, 这是必需的, 敬请读者注意。当然, MPI 系统允许函数 MPI _ BUFFER _ ATTACH 提交更大的内存空间。

特别地, 在同一时刻, 一个进程只能定义一个缓存区, 也就是说, 进程在定义另一个缓存区时, 必须释放前一个已经定义的缓存区。

例 3.12 定义和释放消息缓存区示例

```

REAL * 8 A(100, 100), B(200, 200)
CHARACTER BUF1(100000), BUF2(400000)
CALL MPI _ BUFFER _ ATTACH(BUF1, 100000, IERR)
CALL MPI _ BSEND(A, 10000, MPI _ DOUBLE _ PRECISION, 1, TAG, COMM, IERR)
CALL MPI _ BUFFER _ DETACH(BUF1, 100000, IERR)
CALL MPI _ BUFFER _ ATTACH(BUF2, 400000, IERR)
CALL MPI _ BSEND(B, 40000, MPI _ DOUBLE _ PRECISION, 1, TAG, COMM, IERR)

```

在上例中, 第一次定义的缓存区的大小, 小于第二次将要发送的消息的大小, 因此, 有必要定义一个更大的新缓存区, 但在定义之前, 必须先释放前一次已经定义的缓存区。

下面, 我们给出 3.4 节中有关如何避免阻塞式通信中出现死锁问题的第五种办法, 就

是采用函数 MPI_BSEND 替换函数 MPI_SEND，具体为：

```
INTEGER SIZE
SIZE = COUNT * SIZEOF(MPR_REAL) + 2 * MPI_BSEND_OVERHEAD
CALL MPI_BUFFER_ATTACH(XXX, SIZE, IERR)
IF (RANK.EQ.0) THEN
    CALL MPI_BSEND(SENDBUF, COUNT, MPI_REAL, 1, TAG, COMM, IERR)
    CALL MPI_RECV(RECVBUF, COUNT, MPI_REAL, 1, TAG, COMM, STATUS, IERR)
ELSE IF(RANK.EQ.1) THEN
    CALL MPI_BSEND(SENDBUF, COUNT, MPI_REAL, 0, TAG, COMM, IERR)
    CALL MPI_RECV(RECVBUF, COUNT, MPI_REAL, 0, TAG, COMM, STATUS, IERR)
ENDIF
```

下面，我们列出同步模式和就绪模式的阻塞式消息发送函数。

/* 同步模式消息发送函数，执行一个同步模式的阻塞式消息发送操作。

```
MPI_SSEND(buf, count, datatype, dest, tag, comm)
    IN      buf          消息发送缓存区初始地址
    IN      count        消息发送缓存区内数据单元个数
    IN      datatype    数据单元类型
    IN      dest         接收进程的序号
    IN      tag          消息号
    IN      comm         通信器
C     int MPI_Ssend(void * buf, int count, MPI_Datatype datatype, int dest, int tag,
                  MPI_Comm comm)
Fortran MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type>    BUF(*)
    INTEGER    COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

/* 就绪模式消息发送函数，执行一个就绪模式的阻塞式消息发送操作。

```
MPI_RSEND(buf, count, datatype, dest, tag, comm)
    IN      buf          消息发送缓存区初始地址
    IN      count        消息发送缓存区数据单元个数
    IN      datatype    数据单元类型
    IN      dest         接收进程的序号
    IN      tag          消息号
    IN      comm         通信器
C     int MPI_Rsend(void * buf, int count, MPI_Datatype datatype, int dest, int tag,
                  MPI_Comm comm)
Fortran MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type>    BUF(*)
    INTEGER    COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

3.7.2 非阻塞式消息发送函数

/* 缓存模式非阻塞消息发送函数, 提交一个缓存模式的非阻塞消息发送操作。

```
MPI_ IBSEND(buf, count, datatype, dest, tag, comm, request)
    IN      buf          消息发送缓存区初始地址
    IN      count        消息发送缓存区数据单元个数
    IN      datatype    数据单元类型
    IN      dest         接收进程的序号
    IN      tag          消息号
    IN      comm         通信器
    OUT     request     通信请求

C      int MPI_ Ibsend(void * buf, int count, MPI_ Datatype datatype, int dest, int tag,
                  MPI_ Comm comm, MPI_ Request * request)

Fortran MPI_ IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
                     IERROR)
<type>   BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
         IERROR
```

函数 MPI_ IBSEND 与函数 MPI_ BSEND 的关系完全类似于函数 MPI_ ISEND 与函数 MPI_ SEND 的关系, 这里不再讨论。

/* 同步模式非阻塞消息发送函数, 提交一个同步模式的非阻塞消息发送操作。

```
MPI_ ISSEND(buf, count, datatype, dest, tag, comm, request)
    IN      buf          消息发送缓存区初始地址
    IN      count        消息发送缓存区内数据单元个数
    IN      datatype    数据单元类型
    IN      dest         接收进程的序号
    IN      tag          消息号
    IN      comm         通信器
    OUT     request     通信请求

C      int MPI_ Issend(void * buf, int count, MPI_ Datatype datatype, int dest, int tag,
                  MPI_ Comm comm, MPI_ Request * request)

Fortran MPI_ ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
                     IERROR)
<type>   BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
         IERROR
```

/* 就绪模式非阻塞消息发送函数, 提交一个就绪模式的非阻塞消息发送操作。

```
MPI_ IRSSEND(buf, count, datatype, dest, tag, comm, request)
```

IN	buf	消息发送缓存区初始地址
IN	count	消息发送缓存区内数据单元个数
IN	datatype	数据单元类型
IN	dest	接收进程的序号
IN	tag	消息号
IN	comm	通信器
OUT	request	通信请求

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

Fortran MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
 IERROR)
 <type> BUF(*)
 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
 IERROR

3.7.3 持久通信函数

/* 缓存模式持久通信函数, 创建一个缓存模式的消息发送持久通信请求。

		MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)
IN	buf	消息发送缓存区初始地址
IN	count	消息发送缓存区内数据单元个数
IN	datatype	数据单元类型
IN	dest	接收进程的序号
IN	tag	消息号
IN	comm	通信器
OUT	request	通信请求

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

Fortran MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
 REQUEST, IERROR)
 <type> BUF(*)
 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
 IERROR

· 函数 MPI_BSEND_INIT 与函数 MPI_IBSEND 的关系, 也完全类似于标准模式的函数 MPI_SEND_INIT 与函数 MPI_ISEND 的关系, 这里不再讨论。

/* 同步模式持久通信函数, 创建一个同步模式的消息发送持久通信请求。

		MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)
IN	buf	消息发送缓存区初始地址
IN	count	消息发送缓存区内数据单元个数
IN	datatype	数据单元类型

IN	dest	接收进程的序号
IN	tag	消息号
IN	comm	通信器
OUT	request	通信请求

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

 /* 就绪模式持久通信函数, 创建一个就绪模式的消息发送持久通信请求。

 MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)
 IN buf 消息发送缓存区初始地址
 IN count 消息发送缓存区内数据单元个数
 IN datatype 数据单元类型
 IN dest 接收进程的序号
 IN tag 消息号
 IN comm 通信器
 OUT request 通信请求
 C int MPI_Rsend_init(void * buf, int count, MPI_Datatype datatype, int dest,
 int tag, MPI_Comm comm, MPI_Request * request)
 Fortran MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
 REQUEST, IERROR)
 <type> BUF(*)
 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
 IERROR

通过上述函数创建的持久通信请求,与3.5节中介绍的持久通信请求一样,必须通过3.5节中介绍的函数MPI_START或函数MPI_ALLSTART来提交,并通过3.6节中介绍的函数MPI_REQUEST_FREE和函数MPI_CANCEL来释放。

第四章 自定义数据类型与数据封装

在第三章，我们介绍了 MPI 点对点通信函数，这些函数有一个共同的特点，就是要求发送和接收的数据单元属于同一种类型，且存储在连续的内存空间中。但实际应用问题通常要求发送类型不一致、存储位置不连续的多个数据单元，此时，我们只能调用 MPI 函数一个一个地发送。例如，假设进程 0 要求将实型数组 A(1000) 的 500 个下标为奇数的元素发送给进程 1，则程序代码只能写成：

```
IF(MYRANK.EQ.0) THEN
    DO I=1, 1000, 2
        CALL MPI_SEND(A(I), 1, MPI_REAL, 1, 9999, COMM, IERR)
    ENDDO
ELSE IF(MYRANK.EQ.1) THEN
    DO I=1, 1000, 2
        CALL MPI_RECV(A(I), 1, MPI_REAL, 1, 9999, COMM, STATUS, IERR)
    ENDDO
ENDIF
```

显然，这种程序设计方法非常繁琐，容易出错，且通信开销很大。为此，MPI 系统提供了两类特殊的函数：自定义数据类型函数和消息封装/拆卸函数。利用这些函数，用户可以定义一种新的数据类型，它包含将要发送出去的不同类型、存储在不同位置的多个数据单元，使原来的多次消息发送操作现在只需一次就可以完成，从而方便了并行程序的设计，提高了并行计算性能。

本章主要介绍 MPI 系统提供的自定义数据类型函数和消息封装/拆卸函数。

4.1 自定义数据类型

自定义数据类型(User-defined data type)也称为**导出数据类型**(Derived data type)，它是指基于表 3.2 中定义的 MPI 基本数据类型(例如 MPI_REAL, MPI_INTEGER, MPI_DOUBLE_PRECISION, MPI_CHARACTER 等)，通过说明一系列连续或非连续、相同或不同类型的数据单元在内存中的存储位置，应用程序自己定义的一种新的数据类型，它可以作为 MPI 函数中的数据类型参数参与消息传递。

MPI 系统规定，自定义数据类型由两个一一对应的序列构成，其中一个为**类型序列**，由多个 MPI 基本数据类型组成，在具体应用程序中，每个类型均对应一个具体的数据单元；另一个为**位置序列**，它包含类型序列的各个元素对应的具体数据单元在内存中的存储位置。具体而言，假设一个自定义数据类型包含 n 个 MPI 基本数据类型，则其类型序列和位置序列可表示成：

```
TypeSig = {type0, …, typen-1}
TypeDisp = {disp0, disp1, …, dispn-1}
```

分别称之为该数据类型的**类型特征**和**位置特征**,并称:

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

为数据类型的**类型图**(typemap)。其中, $type_i$ 为第 i 个 MPI 基本数据类型, $disp_i$ (字节为单位)为 $type_i$ 所对应的数据单元在内存中的存储位置。在实际应用程序中, 我们通常可设置 $disp_0 = 0$, $disp_i$ 为 $type_i$ 相对于 $type_0$ 的数据单元在内存中的存储位置。

MPI 系统规定, 自定义数据类型可以像 MPI 基本数据类型一样, 在所有 MPI 通信函数中作为数据类型参数使用。例如, `MPI_SEND(buf, 1, datatype, ...)` 和 `MPI_RECV(buf, 1, datatype, ...)` 中的参数 `datatype` 就可以是自定义数据类型, 此时, `datatype` 和 `buf` 定义了一个消息缓存区, 它由 n 个数据单元构成, 其中第 i 个单元的内存地址为 $buf + disp_i$, 类型为 $type_i$ 。

MPI 基本数据类型可以认为是自定义数据类型的特例, 其类型图为 $(datatype, 0)$ 。因此, 在本书以后所有章节的讨论中, 如果提到数据类型, 则它既可以是基本数据类型, 又可以是自定义数据类型。

关于数据类型, MPI 系统定义了如下的几个重要概念:

- (1) **数据类型大小**: 定义为该数据类型生成的数据单元占有的内存空间的实际大小(字节为单位)。例如, 一个整型、实型和双精度型数据类型的大小分别为 4, 4 和 8。
- (2) **数据单元对界大小**: 具体分两种情况: 第一, 对 MPI 基本数据类型, 它定义为该数据类型的大小; 第二, 对 MPI 自定义数据类型, 它定义为该数据类型的类型图包含的所有 MPI 基本数据类型对界大小的最大者。
- (3) **域(extent)**: 数据类型的下界下上界之差, 详见以下定义。

结合前面讨论的类型图 $\text{Typemap} = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$, 令 k_j 为类型 $type_j$ 的数据单元对界大小, 则:

$$\delta = \max_{j=0}^{n-1} |k_j|$$

就为该数据类型的数据单元对界大小, 令:

$$\begin{aligned} lb(\text{Typemap}) &= \min_i disp_i \\ ub(\text{Typemap}) &= \max_i (disp_i + \text{sizeof}(type_i)) + \epsilon \end{aligned}$$

分别定义为该数据类型的域的下界和上界, 而

$$\text{extent}(\text{Typemap}) = ub(\text{Typemap}) - lb(\text{Typemap})$$

就是该数据类型的域。其中, ϵ 是域 $\text{extent}(\text{Typemap})$ 为 δ 的最小倍数的增量, 下标 j 取值于 $\{0, \dots, n-1\}$ 之间。

由以上定义可知, 两个 MPI 数据类型的域和它们的大小之间有必然的联系。如果域相等, 则大小可能相等, 也可能不等; 反之亦然。例如, 假设类型图 $T1 = \{(double, 0), (char, 8)\}$, 则 $T1$ 的大小等于 9, 上界 $ub(T1) = 9 + \epsilon$, 下界 $lb(T1) = 0$, $\delta = \max\{8, 1\} = 8$, $\epsilon = 2 \times \delta - 9 = 16 - 9 = 7$, 故域 $\text{extent}(T1) = 16$; 类型图 $T2 = \{(double, 0), (char, 8), (char, 9), (char, 10)\}$, 则 $T2$ 的大小等于 11, 上界 $ub(T2) = 11 + \epsilon$, 下界 $lb(T1) = 0$, $\delta = \max\{1, 8\} = 8$, $\epsilon = 2 \times \delta - 11 = 16 - 11 = 5$, 故域 $\text{extent}(T2) = 16$ 。 $T1$ 和 $T2$ 域相等, 但大小不等。又如, 类型图 $T3 = T1$, $T4 = \{(double, 0), (char, 16)\}$, 则 $T3$ 和 $T4$ 大小相等, 但域不等。

省缺情况下, MPI 系统按以上方式定义一个数据类型的上界和下界,但是, MPI 系统也允许用户人为地强制定义一个数据类型的上界和下界。特别地, MPI 系统提供了两个特殊的数据类型 MPI _ LB 和 MPI _ UB, 称为伪数据类型(pseudo datatype)。它们的大小为 0, 并且当它们出现在一个数据类型的类型图中时,对该数据类型的实际数据内容不起任何作用。它们的作用是让用户可以人工指定一个数据类型的上下界。MPI 规定:如果一个数据类型的基本类型中包含有 MPI _ LB, 则其下界定义为:

$$lb(type) = \min_i \{disp_i | type_i = MPI_LB\}$$

同样,如果包含有 MPI _ UB, 则它的上界定义为:

$$ub(type) = \max_i \{disp_i | type_i = MPI_UB\}$$

例如,类型图 $\{(MPI_LB, -4), (MPI_UB, 20), (MPI_DOUBLE_PRE(ISION, 0), (MPI_INTEGER, 8), (MPI_BYFE, 12)\}$ 的下界为 -4, 上界为 20, 域为 24。

函数 MPI _ TYPE _ LB 和函数 MPI _ TYPE _ UB 以字节为单位, 分别返回 MPI 数据类型的域的下界和上界。

/* 数据类型域的下界查询函数, 获取某个数据类型域的下界。

```
MPI_TYPE_LB(datatype, lb)
    IN      datatype    数据类型
    OUT     lb          数据类型域的下界(字节为单位)
C      int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint lb)
Fortran MPI_TYPE_LB(DATATYPE, LB, IERROR)
    INTEGER DATATYPE, LB, IERROR
```

/* 数据类型域的上界查询函数, 获取某个数据类型域的上界。

```
MPI_TYPE_UB(datatype, ub)
    IN      datatype    数据类型
    OUT     ub          数据类型域的上界(字节为单位)
C      int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint ub)
Fortran MPI_TYPE_UB(DATATYPE, UB, IERROR)
    INTEGER DATATYPE, UB, IERROR
```

例如函数 MPI _ TYPE _ LB(T1)返回值 0, 函数 MPI _ TYPE _ UB(T2)返回值 16。下面两个函数以字节为单位, 分别返回 MPI 数据类型的域和大小。

/* 数据类型域查询函数, 获取某个数据类型的域。

```
MPI_TYPE_EXTENT(datatype, extent)
    IN      datatype    数据类型
    OUT     extent      数据类型的域(字节为单位)
C      int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint extent)
Fortran MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
    INTEGER DATATYPE, EXTENT, IERROR
```

```
/* 数据类型大小查询函数, 获取某个数据类型的大小。
```

```
MPI_TYPE_SIZE(datatype, size)
    IN      datatype   数据类型
    OUT     size        数据类型的大小(字节为单位)
C      int MPI_Type_size(MPI_Datatype datatype, int * size)
Fortran MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

例如, 函数 MPI_TYPE_EXTENT(T1,i) 和 MPI_TYPE_EXTENT(T2,i) 都返回 i=16, 而函数 MPI_TYPE_SIZE(T1,i) 和 MPI_TYPE_SIZE(T2,i) 将分别返回 i=9 和 i=11。

这里, 我们还要指出的是, MPI 自定义数据类型是一种特殊的数据结构, 属于 MPI 内部的不透明对象, 它可以由 MPI 系统提供的函数创建和释放, 当自定义数据类型的数据单元在进程之间进行传送时, 必须特别注意这些数据单元的数据类型是否匹配, 也就是说, 一个进程, 如果想接收某个自定义数据类型的数据单元, 则它首先必须自己调用 MPI 函数, 创建与之相匹配的数据类型, 然后才能正确地接收该数据单元。

4.2 自定义数据类型的创建

本节主要介绍 MPI 系统提供的创建自定义数据类型的函数。

```
/* 自定义数据类型创建函数, 在连续的内存空间上创建一个新的数据类型。
```

```
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
    IN      count       oldtype 数据单元个数
    IN      oldtype     原数据类型
    OUT     newtype     新数据类型
C      int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                           MPI_Datatype * newtype)
Fortran MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

函数 MPI_TYPE_CONTIGUOUS 是最简单的自定义数据类型创建函数。如图 4.1 所示, 它将 count=4 个类型为 oldtype(MPI 基本数据类型或用户自定义数据类型) 的数据复制到连续的位置, 形成新的自定义数据类型 newtype。其中, 图中方块的每个小格既可以代表一个字节, 也可以代表一个任意的 MPI 数据类型。

为了方便, 我们约定: 除非特别说明, 否则本节以后的数据类型创建函数示意图中方块和方块包含的每个小格的含义与图 4.1 描述的一致。

相对于上图, 假设 oldtype=MPI_INTEGER, 则可将函数调用形式写为:

```
CALL MPI_TYPE_CONTIGUOUS(4, MPI_INTEGER, NEWTYPE, IERR)
```



图 4.1 函数 MPI_TYPE_CONTIGUOUS 示意图

由此，发送整型数组 A(100)的前 4 个元素的函数调用可写成：

```
CALL MPI_SEND(A, 1, NEWTYPE, DEST, TAG, COMM, IERR)
```

它等价于函数调用：

```
CALL MPI_SEND(A, 4, MPI_INTEGER, DEST, TAG, COMM, IERR)
```

其中，DEST 为接收进程序号，TAG 为消息号，COMM 为通信器。

/ 自定义数据类型创建函数，创建一个新的数据类型，由间隔的多个数据块组成。*

	MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)	
IN	count	数据块个数
IN	blocklength	每个数据块包含的 oldtype 数据单元的个数
IN	stride	以数据单元为单位，连续两个数据块起始地址之间的间距
IN	oldtype	原数据类型
OUT	newtype	新数据类型

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

如图 4.2 所示，函数 MPI_TYPE_VECTOR 创建一个新的数据类型 newtype，它由 count = 3 个数据块组成，每个数据块包含 blocklength = 2 个类型为 oldtype 的数据单元，且连续数据块的起始地址之间相差 stride = 3 个 oldtype 数据单元。



图 4.2 函数 MPI_TYPE_VECTOR 示意图

相对于上图，假设 oldtype = MPI_INTEGER，则可将函数调用形式写为：

```
CALL MPI_TYPE_VECTOR(3, 2, 3, MPI_INTEGER, NEWTYPE, IERR)
```

由此，发送整型数组 A(100)的元素 A(1), A(2), A(4), A(5), A(7), A(8)的函数调用可写成：

```
CALL MPI_SEND(A, 1, NEWTYPE, DEST, TAG, COMM, IERR)
```

它等价于三次函数调用：

```

CALL MPI_SEND(A(1), 2, MPI_INTEGER, DEST, TAG, COMM, IERR)
CALL MPI_SEND(A(4), 2, MPI_INTEGER, DEST, TAG, COMM, IERR)
CALL MPI_SEND(A(7), 2, MPI_INTEGER, DEST, TAG, COMM, IERR)

```

其中, DEST 为接收进程序号, TAG 为消息号, COMM 为通信器。

/* 自定义数据类型创建函数, 创建一个新的数据类型, 由间隔的多个数据块组成。

```

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)
    IN      count      数据块个数
    IN      blocklength 每个数据块包含 oldtype 数据单元的个数
    IN      stride      以字节为单位, 连续两个数据块起始地址之间的间距
    IN      oldtype     原数据类型
    OUT     newtype     新数据类型
C      int MPI_Type_hvector(int count, int blocklength, int stride, MPI_Datatype
                           oldtype, MPI_Datatype * newtype)
Fortran MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
                         NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
         NEWTYPE, IERROR

```

函数 MPI_TYPE_HVECTOR 与函数 MPI_TYPE_VECTOR 的功能类似, 惟一不同的是, 连续两个数据块起始地址之间的间距不是以 oldtype 数据单元为单位, 而是以字节为单位, 具体如图 4.3 所示。在该图中, 假设每个小格代表一个字节。

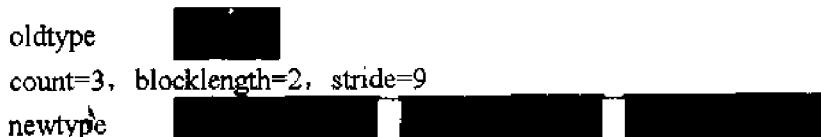


图 4.3 函数 MPI_TYPE_HVECTOR 示意图

相对于上图, 假设 oldtype=MPI_INTEGER, 则可将函数调用形式写为:

```
CALL MPI_TYPE_HVECTOR(3, 2, 9, MPI_INTEGER, NEWTYPE, IERR)
```

由此, 假设某个数据公用块/XXX/包含如下的数据单元:

```

INTEGER      A(2), B(2), C(2)
CHARACTER    D, E
COMMON /XXX/ A, D, B, E, C

```

则发送整型数组 A, B, C 的函数调用可写成:

```
CALL MPI_SEND(A, 1, NEWTYPE, DEST, TAG, COMM, IERR)
```

它等价于三次函数调用:

```

CALL MPI_SEND(A, 2, MPI_INTEGER, DEST, TAG, COMM, IERR)
CALL MPI_SEND(B, 2, MPI_INTEGER, DEST, TAG, COMM, IERR)
CALL MPI_SEND(C, 2, MPI_INTEGER, DEST, TAG, COMM, IERR)

```

其中, DEST 为接收进程序号, TAG 为消息号, COMM 为通信器。

例 4.1 矩阵转置

以行序创建一个描述矩阵各元素存储位置的数据类型, 发送包含该矩阵所有元素的消息, 以列序接收该消息, 完成矩阵转置。具体程序如下:

```
REAL      a(100,100), b(100,100)
INTEGER   row, xpose, sizeofreal, myrank, ierr
INTEGER   status(MPI_STATUS_SIZE)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
!
! 获取 MPI_REAL 的域。
CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
! 创建新的数据类型 row, 它包含 100 个相互间隔的实型数据单元, 对应矩阵的一行。
CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
!
! 创建新的数据类型 xpose, 它包含 100 个数据类型 row, 对应以行序排列的
! 所有矩阵元素。
CALL MPI_TYPE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr)
!
! 提交新产生的数据类型 xpose 给 MPI 系统。
CALL MPI_TYPE_COMMIT(xpose, ierr)
!
! 通过消息传递完成矩阵转置。
CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100 * 100, MPI_REAL,
                   myrank, 0, MPI_COMM_WORLD, status, ierr)
```

其中, 函数 MPI_TYPE_COMMIT 向 MPI 系统提交一个自定义数据类型, 可用于后面的消息传递操作。实际上, MPI 系统规定, 任何自定义数据类型创建之后, 只有当其被函数 MPI_TYPE_COMMIT 提交后, 才能用于其他的 MPI 通信函数。关于函数 MPI_TYPE_COMMIT, 我们将在下节讨论。

```
/* 自定义数据类型创建函数, 创建一个新的数据类型, 由任意间隔(以原数据类型
/* 为单位)的多个数据块组成。
```

```
MPI_TYPE_INDEXED(count, array_of_blocklengths,
                  array_of_displacements, oldtype, newtype)
IN    count          数据块个数
IN    array_of_blocklengths 数组, 含每个数据块拥有的数据单元个数
IN    array_of_displacements 数组, 含每个数据块的初始位置(oldtype 为单位)
IN    oldtype        原数据类型
OUT   newtype        新数据类型
C     int MPI_Type_indexed(int count, int *array_of_blocklengths, int
                           *array_of_displacements, MPI_Datatype oldtype,
                           MPI_Datatype *newtype)
```

```

Fortran MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
                         ARRAY_OF_DISPLACEMENTS, OLDTYPE,
                         NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
        ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
        NEWTYPE, IERROR

```

如图 4.4 所示, 函数 MPI_TYPE_INDEXED 创建的新的数据类型由 count=3 个数据块组成, 其中第 i 个数据块从位置 array-of-placements(i) 开始(数据单元为单位), 包含 array-of-blocklengths(i) 个类型为 oldtype 的数据单元。

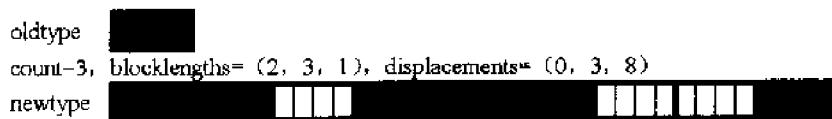


图 4.4 函数 MPI_TYPE_INDEXED 示意图

相对于上图, 假设 oldtype = MPI_REAL, 则可将函数调用形式写为:

```

CALL MPI_TYPE_INDEXED(3, blocklengths, displacements, MPI_REAL,
                      NEWTYPE, IERR)

```

由此, 发送实型数组 A(100)的第 1, 2, 4, 5, 6, 9 个元素的函数调用可写成:

```

CALL MPI_SEND(A, 1, NEWTYPE, DEST, TAG, COMM, IERR)

```

它等价于三次函数调用:

```

CALL MPI_SEND(A(1), 2, MPI_REAL, DEST, TAG, COMM, IERR)
CALL MPI_SEND(A(4), 3, MPI_REAL, DEST, TAG, COMM, IERR)
CALL MPI_SEND(A(9), 1, MPI_REAL, DEST, TAG, COMM, IERR)

```

其中, DEST 为接收进程号, TAG 为消息号, COMM 为通信器。

```

/* 自定义数据类型创建函数, 创建一个新的数据类型, 由任意间隔(字节为单位)
/* 的多个数据块组成。

```

```

MPI_TYPE_HINDEXED(count, array_of_blocklengths,
                   array_of_displacements, oldtype, newtype)

IN      count          数据块个数
IN      array_of_blocklengths 数组, 包含每个数据块拥有的数据单元个数
IN      array_of_displacements 数组, 包含每个数据块的初始位置(字节为单位)
IN      oldtype         原数据类型
OUT     newtype         新数据类型

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

```

```

Fortran MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
                           ARRAY_OF_DISPLACEMENTS, OLDTYPE,
                           NEWTYPE, IERROR)

```

```

        NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS( * ), ARRAY_OF_
DISPLACEMENTS( * ), OLDTYPE, NEWTYPE, IERROR

```

除了数组 array-of-displacements 中的元素是以字节为单位外, 函数 MPI_TYPE_HINDEXED 与函数 MPI_TYPE_INDEXED 的含义完全一致, 具体如图 4.5 所示。其中, 我们假设每个小格表示一个字节。

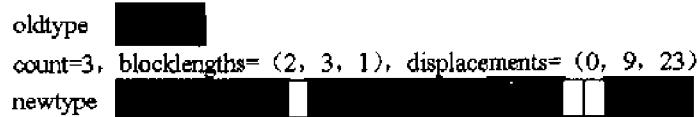


图 4.5 MPI_TYPE_HINDEXED 示意图

相对于上图, 假设 oldtype=MPI_REAL, 则可将函数调用形式写为:

```

CALL MPI_TYPE_HINDEXED(3, blocklengths, displacements, MPI_REAL,
                       NEWTYPE, IERR)

```

由此, 假设某个数据公用块/XXX/包含如下的数据单元:

```

REAL          A(2), B(3), C
CHARACTER D, E(2)
COMMON /XXX/ A, D, B, E, C

```

则发送实型数组 A, B 和数据 C 的函数调用可写成:

```

CALL MPI_SEND(A, 1, NEWTYPE, DEST, TAG, COMM, IERR)

```

它等价于三次函数调用:

```

CALL MPI_SEND(A, 2, MPI_REAL, DEST, TAG, COMM, IERR)
CALL MPI_SEND(B, 3, MPI_REAL, DEST, TAG, COMM, IERR)
CALL MPI_SEND(C, 1, MPI_REAL, DEST, TAG, COMM, IERR)

```

其中, DEST 为接收进程号, TAG 为消息号, COMM 为通信器。

/* 自定义数据类型创建函数, 创建一个新的数据类型, 由多种不同数据类型组成。

```

MPI_TYPE_STRUCT(count, array_of_blocklengths,
                 array_of_displacements, array_of_types, newtype)

IN   count                  数据块个数
IN   array_of_blocklengths    数组, 包含每个数据块拥有的数据单元个数
IN   array_of_displacements  数组, 包含每个数据块的初始位置(字节为单位)
IN   array_of_types           数组, 包含每个数据块拥有的数据单元类型
OUT  newtype                新数据类型

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

Fortran 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

```

函数 MPI_TYPE_STRUCT 是适应面最广的自定义数据类型创建函数, 它扩展了函数 MPI_TYPE_HINDEXED 的功能, 允许每个数据块复制不同的数据类型。如图4.6 所示, 新的数据类型由 3 个数据块组成, 其中第 i 个数据块包含 2 个类型为 array_of_types(i) 的 blocklengths(i) 个数据单元, 并从位置 displacements(i) 开始存储。

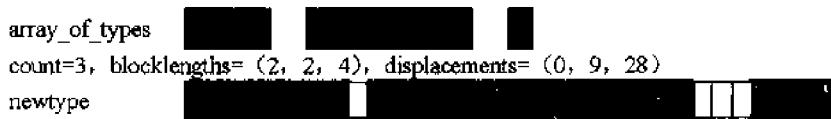


图 4.6 MPI_TYPE_STRUCT 示意图

相对于上图, 假设 array_of_types = (MPI_REAL, MPI_DOUBLE_PRECISION, MPI_BYTE), 则可将函数调用形式写为:

```

CALL MPI_TYPE_STRUCT(3, blocklengths, displacements,
                     array_of_types, NEWTYPE, IERR)

```

例 4.2 函数 MPI_TYPE_STRUCT 的应用示例

```

CHARACTER      CLASS
REAL*8         D(6)
INTEGER        A(8)
INTEGER TYPE(3), DISP(3), LEN(3), ADDR(3), NEWTYPE
!
TYPE(1)=MPI_CHARACTER
TYPE(2)=MPI_DOUBLE_PRECISION
TYPE(3)=MPI_INTEGER
LEN(1)=1
LEN(2)=6
LEN(3)=8
CALL MPI_ADDRESS(CLASS, ADDR(1), IERR)
! 获取变量 CLASS 的内存地址。
DISP(1)=0
CALL MPI_ADDRESS(D, ADDR(2), IERR)
! 获取变量 D(1)的内存地址。
DISP(2)=ADDR(2)-ADDR(1)
! 相对于变量 CLASS 的内存地址, 变量 D(1)的位移。
CALL MPI_ADDRESS(A, ADDR(3), IERR)
! 获取变量 A(1)的内存地址。

```

```

DISP(3) = ADDR(3) - ADDR(1)
    ! 相对于变量 CLASS 的内存地址, 变量 A(1)的位移。
!
CALL MPI_TYPE_STRUCT(3, LEN, DISP, TYPE, NEWTYPE, IERR)
    ! 定义新的 MPI 数据类型。
CALL MPI_TYPE_COMMIT(newtype, ierr)
    ! 提交新产生的数据类型 NEWTYPE 给 MPI 系统。
CALL MPI_SEND(CLASS, 1, NEWTYPE, DEST, TAG, COMM, IERR)
    ! 将包含变量 CLASS, 双精度型数组 D(6)和整型数组 A(8)的消息发送出去。

```

在上例中, 函数 MPI_ADDRESS 以字节为单位, 返回一个变量的内存地址, 具体定义如下:

/* 内存地址查询函数, 获取给定变量在内存空间的绝对地址。

MPI_ADDRESS(location, address)	
IN location	变量
OUT address	内存地址

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

对 Fortran 语言, 应用函数 MPI_ADDRESS 时必须注意:

- (1) 如果并行机具有 64 位的内存地址, 例如 SGI Origin 2000, 则变量 ADDRESS 必须被显式地说明为 8 个字节的整型变量, 否则可能返回错误的地址;
- (2) 如果并行机具有 32 位的内存地址, 例如微机机群, 则变量 ADDRESS 可被说明为 4 个字节的整型变量。

对 C 语言, 我们不必关心这个问题, 因为变量 address 的类型必须说明为 MPI_Aint, 对 64 位地址空间, 类型 MPI_Aint 的大小为 8 个字节, 而对 32 位地址空间, 类型 MPI_Aint 的大小为 4 个字节。

应用函数 MPI_ADDRESS 获得变量的绝对地址后, 可用该地址作为参数, 通过函数 MPI_TYPE_STRUCT 来创建新的自定义数据类型。为此, MPI 系统提供一个被称之为零地址的常数 MPI_BOTTOM, 它代表内存地址空间的起始地址, 一般可用于辅助发送或接收由绝对地址创建的自定义数据类型。应用零地址, 例 4.2 可得到简化。

例 4.3 函数 MPI_ADDRESS 与常数 MPI_BOTTOM 的应用示例

```

CALL MPI_ADDRESS(CLASS, ADDR(1), IERR)
    ! 获取变量 CLASS 的内存地址。
CALL MPI_ADDRESS(D, ADDR(2), IERR)
    ! 获取变量 D(1)的内存地址。
CALL MPI_ADDRESS(A, ADDR(3), IERR)

```

```

! 获取变量 A(1)的内存地址。
CALL MPI_TYPE_STRUCT(3, LEN, ADDR, TYPE, NEWTYPE, IERR)
    ! 定义新的 MPI 数据类型。
CALL MPI_TYPE_COMMIT(newtype, ierr)
    ! 提交新产生的数据类型 NEWTYPE 给 MPI 系统。
CALL MPI_SEND(MPI_BOTTOM, 1, NEWTYPE, DEST, TAG, COMM, IERR)
    ! 将包含变量 CLASS, 双精度数组 D(6)和整型数组 A(8)的消息发送出去。

```

4.3 自定义数据类型的应用

一个自定义数据类型创建之后,必须由专门的 MPI 函数提交给 MPI 系统后,才能应用于 MPI 通信函数中。同时,创建后的自定义数据类型可以使用专门的 MPI 函数来释放,一旦释放,该数据类型就不再有效。下面,我们分别讨论自定义数据类型的提交和释放,以及应用自定义数据类型时要注意的问题。

4.3.1 自定义数据类型的提交与释放

/* 自定义数据类型提交函数,提交一个已定义的自定义数据类型给 MPI 系统。

```

MPI_TYPE_COMMIT(datatype)
    INOUT      datatype      被提交的自定义数据类型
C      int MPI_Type_commit(MPI_Datatype *datatype)
Fortran MPI_TYPE_COMMIT(DATATYPE, IERROR)
        INTEGER DATATYPE, IERROR

```

自定义数据类型 datatype 被函数 MPI_TYPE_COMMIT 提交后,将被植入 MPI 系统的内部,与基本数据类型一样,可被 MPI 通信函数重复使用,以传递相同或不同内存空间上不同时刻的数据。

/* 自定义数据类型释放函数, MPI 系统释放一个已定义的自定义数据类型

```

MPI_TYPE_FREE(datatype)
    INOUT      datatype      被释放的自定义数据类型
C      int MPI_Type_free(MPI_Datatype *datatype)
Fortran MPI_TYPE_FREE(DATATYPE, IERROR)
        INTEGER DATATYPE, IERROR

```

4.3.2 消息参数的进一步理解

大量的 MPI 通信函数均涉及到数据单元的个数,例如,函数 MPI_SEND(buf, count, datatype, ...)中的参数 count。这里,我们对参数 count 需要进一步的理解:如果 count > 1,则表示连续的 count 个类型为 datatype 的数据单元将被包含在消息中发送出去,且相邻数据单元的内存位置的起始地址之差等于该 datatype 的域,敬请读者注意。特别地,若 datatype 为 MPI 基本数据类型,则连续 count 个数据单元就为存储在连续内存空间上

的 count 个类型为 datatype 的变量。因此, 函数 MPI _ SEND(buf, count, datatype, ……) 等价于:

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype, ierr)
MPI_TYPE_COMMIT(newtype, ierr)
MPI_SEND(buf, 1, newtype, dest, tag, comm, ierr)
MPI_TYPE_FREE(newtype, ierr)
```

4.3.3 数据类型匹配规则的进一步理解

下面, 基于类型图, 我们进一步理解 MPI 系统的数据类型匹配规则。

假设消息发送函数 MPI _ SEND(buf, count, datatype, dest, tag, comm) 被执行, 且 datatype 具有类型图:

$$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$

和域 extent, 则该消息包含 $n \times count$ 个基本数据单元 $\{(i, j), i = 0, \dots, count - 1; j = 0, \dots, n - 1\}$, 其中, 单元 (i, j) 的内存地址为 $addr_i = buf + extent * i + disp_j$, 类型为 $type_j$ 。如果调用该函数的进程在内存地址 $addr_i$ 处存放的数据单元类型也是 $type_j$, 则称该函数是类型匹配的。这样的数据类型匹配规则也同样适用于其他的 MPI 通信函数。

4.3.4 数据类型查询函数

如果一个基于自定义数据类型的消息被接收, 例如:

```
MPI_RECV(buf, count, datatype, src, tag, comm, status)
```

则 3.1.6 节介绍的函数:

```
MPI_GET_COUNT(status, datatype, count)
```

将返回消息包含的类型为 datatype 的数据单元的个数。更进一步, 如果函数 MPI _ GET _ COUNT 返回 $count = k$, 而每个 datatype 又包含 n 个 MPI 基本数据类型单元, 则该消息总共包含 $n \times k$ 个 MPI 基本数据类型单元, 这个统计数字可由函数 MPI _ GET _ ELEMENTS 来得到。

```
/* MPI 基本数据类型单元个数查询函数, 查询消息包含的 MPI 基本数据类型数据。
/* 单元的个数。
```

```
MPI_GET_ELEMENTS(status, datatype, count)
    IN      status      消息接收函数返回的接收状态信息
    IN      datatype   数据单元类型
    OUT     count       消息中包含的 MPI 基本数据类型单元的个数
C      int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype,
                           int *count)
Fortran MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT,
                  IERROR
```

4.4 数据的封装与拆卸

尽管 MPI 系统提供了诸多自定义数据类型创建函数, 以简化并行程序设计和提高并

行计算性能,但是,这些函数比较适合于固定通信模式的应用问题,即同一种类型的消息被重复利用多次。而对某些应用,消息的组成是动态变化的,一个定义好的消息可能仅被发送或接收一次。此时,如果仍然利用上面的函数,则每次消息传递开始前,均必须调用这些函数创建一个新的数据类型,并提交给 MPI 系统,而该次消息传递完毕,数据类型就不再有用,只能申请释放。这样,自定义数据类型的利用率太低,将会影响并行计算性能,同时也给并行程序设计带来不便。

为此,MPI 系统提供了函数 MPI _ PACK, 用户可以利用它将多个不同类型和数量的数据单元封装到一个应用程序显式提供的内存空间(缓存区)中,形成一个数据单元,其类型为 MPI _ PACKED(参考表 3.2),它可以像其他 MPI 数据类型一样,充当 MPI 通信函数的数据类型参数。因此,通过该函数,我们可以将大量零碎的数据封装在一个消息中,参与消息传递,提高并行计算性能。

/* 数据封装函数,封装应用程序数据单元到应用程序提供的缓存区中。

MPI _ PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)	
IN inbuf	包含应用程序数据单元的输入缓存区的起始地址
IN incount	输入数据单元个数
IN datatype	输入数据单元类型
OUT outbuf	输出缓存区起始地址
IN outsize	输出缓存区大小(字节为单位)
INOUT position	输出缓存区当前位置指针(字节为单位)
IN comm	通信器

C int MPI _ Pack(void * inbuf, int incount, MPI _ Datatype datatype, void * outbuf,
 int outsize, int * position, MPI _ Comm comm)
Fortran MPI _ PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
 POSITION, COMM, IERROR)
 <type> INBUF(*), OUTBUF(*)
 INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM,
 IERROR

函数 MPI _ PACK 从位置 position(字节为单位)开始, 将应用程序的输入缓存区 (inbuf, incount, datatype, comm) 中的数据单元封装到应用程序显式提供的缓存区 (outbuf, outsize) 中, 且自动赋值 position = position + extent(datatype) × incount。其中, 输入缓存区可为函数 MPI _ SEND 所允许的任何消息缓存区, 输出缓存区为包含 outsize 个字节的连续内存空间, 且两个缓存区互不相交。

反之,MPI 系统提供函数 MPI _ UNPACK, 它是函数 MPI _ PACK 的逆函数, 可拆卸一个类型为 MPI _ PACKED 的消息, 并将消息包含的零碎数据单元存储到不同的内存空间。

/* 消息拆卸函数,从应用程序显式提供的输入缓存区中拆卸数据单元到
/* 各输出缓存区。

MPI _ UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

IN	inbuf	包含应用程序数据单元的输入缓存区起始址
IN	insize	输入缓存区大小(字节为单位)
INOUT	position	被拆卸的数据单元在输入缓存区的起始位置(字节为单位)
OUT	outbuf	输出缓存区起始地址
IN	outcount	被拆卸的数据单元个数
IN	datatype	数据单元类型
IN	comm	通信器

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

其实,在 MPI 系统中,任何消息均可用数据类型 MPI_PACKED 来接收,然后调用函数 MPI_UNPACK 拆卸消息中包含的数据单元到各消息接收缓存区中,且每次拆卸完毕,自动赋值 $position = position + extent(datatype) \times outcount$ 。

此外,MPI 系统还提供辅助函数 MPI_PACK_SIZE,它可以查询某个消息所包含的所有数据单元的实际大小,辅助应用程序在调用函数 MPI_PACK 封装该消息之前,得知它所需要的内存空间的大小。

/* 消息大小查询函数,获取某个消息包含的所有数据单元的大小。

MPI_PACK_SIZE(incount, datatype, comm, size)		
IN	incount	消息包含的数据单元个数
IN	datatype	消息包含的数据单元类型
IN	comm	通信器
OUT	size	消息大小(字节)

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

MPI_PACK_SIZE 返回封装某个消息所必需的内存空间大小,其中,该消息包含 incount 个类型为 datatype 的数据,从而可知 $size = incount \times sizeof(datatype)$ 。

例 4.4 数据封装与拆卸函数应用示例(参考例 4.2)

CHARACTER	CLASS
REAL * 8	D(6)
INTEGER	A(8)

```

CHARACTER    BUF(1000)
INTEGER      POSITION, NSTATUS(MPI_STATUS_SIZE), SIZE
!
CALL MPI_COMM_RANK(COMM, MYRANK, IERR)
IF (MYRANK.EQ.0) THEN
  POSITION=0
  CALL MPI_PACK(CLASS, 1, MPI_CHARACTER, BUF, 1000,
                POSITION, COMM, IERR)      ! POSITION = 1
  CALL MPI_PACK(D, 6, MPI_DOUBLE_PRECISION, BUF, 1000,
                POSITION, COMM, IERR)      ! POSITION = 49
  CALL MPI_PACK(A, 8, MPI_INTEGER, BUF, 1000,
                POSITION, COMM, IERR)      ! POSITION = 81
  CALL MPI_SEND(BUF, 1, MPI_PACKED, 1, 5555, COMM, IERR)
ELSE IF(MYRANK.EQ.1) THEN
  CALL MPI_RECV(BUF, 1, MPI_PACKED, 0, 5555, COMM, NSTATUS, IERR)
  POSITION=0
  CALL MPI_UNPACK(BUF, 1000, POSITION, CLASS, 1, MPI_CHARACTER,
                  COMM, IERR)      ! POSITION = 1
  CALL MPI_UNPACK(BUF, 1000, POSITION, D, 6, MPI_DOUBLE_PRECISION,
                  COMM, IERR)      ! POSITION = 49
  CALL MPI_UNPACK(BUF, 1000, POSITION, A, 8, MPI_INTEGER,
                  COMM, IERR)      ! POSITION = 81
ENDIF

```

第五章 聚合通信

MPI 系统称通信器包含的所有进程构成的集合为**进程组**, 其中, 每个进程称为**进程组成员**, 且对应一个唯一的序号。在第三章, 我们介绍了点对点通信, 它们的执行只需要两个进程的参与, 其中一个进程发送消息, 另一个进程接收消息。本章介绍 MPI 系统定义的另一类重要函数: 聚合通信函数(collective communication), 它们的执行需要属于同一个进程组的所有进程的共同参与。MPI 系统提供以下三类聚合通信函数, 以实现不同的功能:

- (1) 同步通信函数(barrier synchronization): 所有进程组成员在某个程序语句上同步。
- (2) 全局通信函数(global communication): 如图 5.1 所示, 它又可分为:
 - 消息广播(broadcast): 消息从一个进程传送到所有进程;
 - 消息收集(gather): 将所有进程的消息片段集中到某个进程;
 - 消息分发(scatter): 按进程个数, 将一个消息分发为多个片段, 并依次发送给各个进程;
 - 消息全收集(allgather): 消息收集完毕, 还将结果广播给各个进程;
 - 消息全交换(all-to-all, complete exchange): 所有进程组成员各自并行地执行一个消息收集或消息分发全局通信函数。
- (3) 全局归约函数(global reduction): 每个进程均提供一个局部变量, 然后对这些局部变量执行某种归约操作, 例如求和、最大值、最小值等, 得到一个新值, 并将该值存储在某个或所有进程中。更进一步, 全局归约函数又可分为归约结果存储在一个进程、归约结果存储在所有进程、归约结果分发到各个进程, 以及并行前缀归约等四种情形。

聚合通信函数的执行需要所有进程组成员的共同参与, 但允许各个进程执行不同的操作。例如:(1)消息广播函数规定, 某个进程必须发送消息, 而其他进程只需接收消息;(2)消息收集函数规定, 每个进程都向其中某个进程发送一个消息, 由该进程一一接收这些消息, 并将结果存储在其局部缓存区中;(3)消息归约函数规定, 所有进程对各自提供的局部变量执行某个操作, 得到新的结果, 该结果只存储在其中的某一个进程中。这些都表明, 在全局通信函数的执行过程中, 可能存在这样一个进程, 它将向所有进程发送或从所有进程接收消息, 且将结果存储在其自身的局部缓存区中, MPI 系统称这样的进程为**根进程(root)**。

对 MPI 系统提供的聚合通信函数, 有如下几点限制:

- (1) 所有进程都必须参与函数调用;
- (2) 进程发送和接收的消息包含的数据单元类型必须相互匹配;
- (3) 无论是发送消息的进程, 还是接收消息的进程, 执行的都是阻塞式通信;
- (4) 在同一进程中, 消息发送缓存区和消息接收缓存区的内存空间必须互不重叠;

(5) 函数中的数据类型, 即可以是基本数据类型, 也可以是自定义数据类型。
本章共分为 3 节, 依次介绍同步通信函数、全局通信函数和全局归约函数。

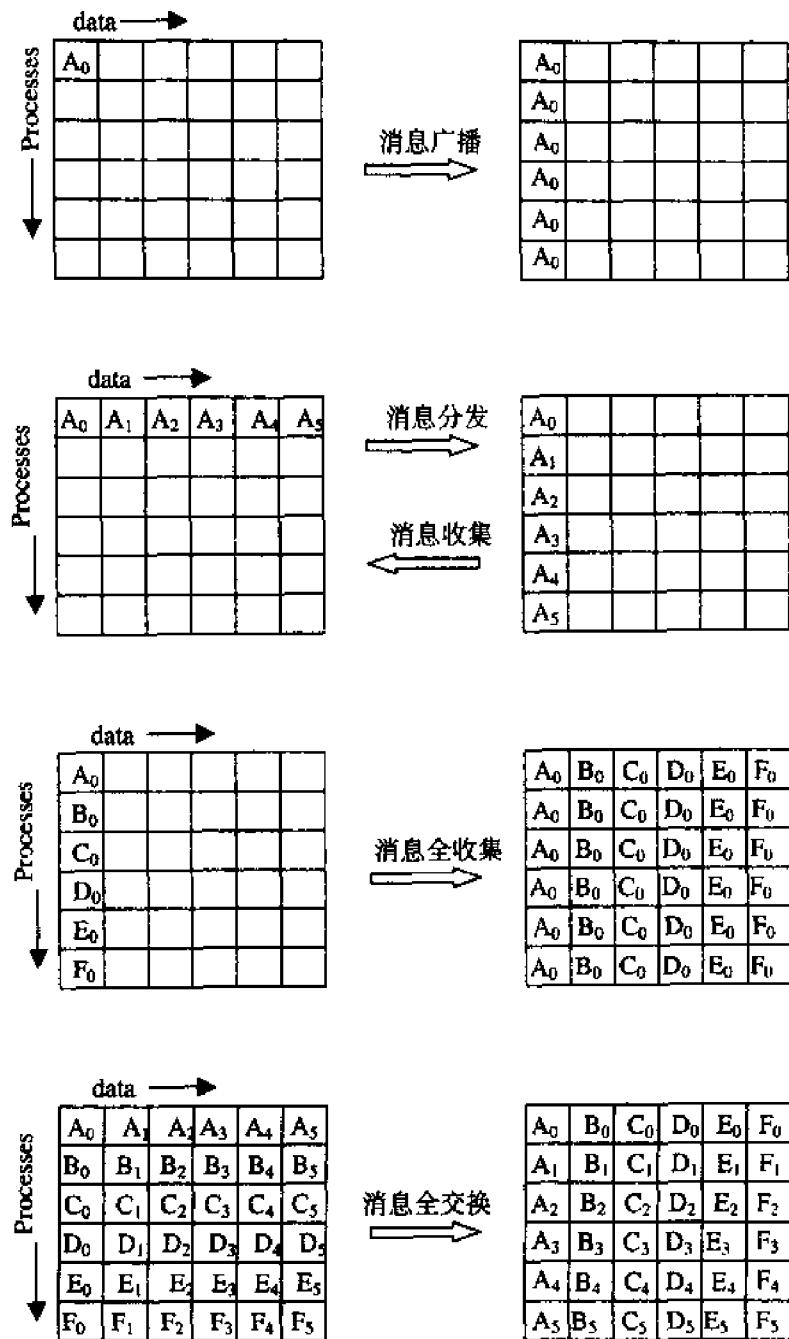


图 5.1 MPI 全局通信函数示意图, 其中, 横向表示每个进程拥有的
数据单元, 纵向表示各个进程

5.1 同步通信函数

```
/* 进程同步通信函数，在程序调用该函数的语句上，进程组各成员之间执行一个同步  
/* 操作。
```

```
MPI_BARRIER(comm)  
IN      comm      通信器  
C      int MPI_BARRIER(MPI_Comm comm)  
Fortran MPI_BARRIER(COMM, IERROR)  
      INTEGER COMM, IERROR
```

在程序的调用语句上，函数 MPI_BARRIER 阻塞式等待所有进程，它规定：当且仅当属于同一通信器 comm 的进程组所有成员，均已到达该程序语句后，各进程才能继续执行；否则，先到达的进程必须空闲地等待其他未到达的进程。另外，我们称程序调用函数 MPI_BARRIER 的语句为该程序的同步点。

例 5.1 同步通信示例

```
.....  
CALL MPI_COMM_RANK(COMM, RANK, IERR)  
IF(MYRANK.EQ.0) THEN  
    CALL WORK0(.....)  
ELSE  
    CALL WORK1(.....)  
ENDIF  
! 设置同步点，直到所有进程完成各自任务，并执行到该程序点后，  
! 各进程的执行才继续。  
CALL MPI_BARRIER(COMM, IERR)  
CALL WORK2(.....)
```

如果我们在上例中插入另一个同步点，即：

```
CALL MPI_COMM_RANK(COMM, RANK, IERR)  
IF(MYRANK.EQ.0) THEN  
    CALL WORK0(.....)  
    CALL MPI_BARRIER(COMM, IERR)  
ELSE  
    CALL WORK1(.....)  
ENDIF
```

则程序执行后，进程 0 将出现死锁，因为它永远也不可能等到其他进程到达该同步点，敬请用户注意。

5.2 全局通信函数

5.2.1 消息广播函数

/* 消息广播函数，在进程组各成员中执行一个消息广播操作。

```
MPI_BCAST(buffer, count, datatype, root, comm)
  INOUT    buffer      待广播的消息缓存区初始地址
  IN       count       待广播的消息缓存区内数据单元个数
  IN       datatype   待广播的消息缓存区内数据单元类型
  IN       root        根进程的序号
  IN       comm        通信器

C     int MPI_Bcast(void * buffer, int count, MPI_Datatype datatype, int root,
                  MPI_Comm comm)

Fortran MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type>    BUFFER(*)
INTEGER    COUNT, DATATYPE, ROOT, COMM, IERROR
```

函数 MPI_BCAST 将根进程 root 提供的消息缓存区 (buffer, count, datatype) 中的数据单元，广播给所有进程组成员。在根进程 root 中，该消息缓存区充当消息发送缓存区，而在其他进程中，则充当消息接收缓存区。函数 MPI_BCAST 要求，每个进程提供的参数 root 和通信器 comm 必须完全一致，否则程序的执行将导致进程的死锁；同时，它也要求所有进程提供的数据类型参数 datatype 必须具有相同的类型图，否则可能丢失数据。

函数 MPI_BCAST 返回时，根进程的消息将被拷贝到所有进程，且存放在消息接收缓存区 (buffer, count, datatype) 中。

例 5.2 消息广播函数应用示例

```
REAL A(100)
CALL MPI_COMM_RANK(COMM, RANK, IERR)
IF(RANK.EQ.0) THEN
    READ(*,*) (A(I), I=1, 10) ! 进程 0 从标准输入读入 10 个实型数
ENDIF
! 进程 0 将读入的 10 个实型数 A(1:10) 广播到所有其他进程
CALL MPI_BCAST(A, 10, MPI_REAL, 0, COMM, IERR)
! 每个进程的局部数据 A(1:10) 将拥有同样的值。
```

5.2.2 消息收集函数

/* 收集聚合通信函数，在进程组各成员中执行一个消息收集操作。

```
MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
            root, comm)
```

IN	sendbuf	各进程提供的消息发送缓存区初始地址
IN	sendcount	各进程提供的消息发送缓存区内数据单元个数
IN	sendtype	各进程提供的消息发送缓存区内数据单元类型
OUT	recvbuf	根进程提供的消息接收缓存区初始地址
IN	recvcount	根进程从每个进程接收的数据单元个数
IN	recvtype	根进程从每个进程接收的数据单元类型
IN	root	根进程的序号
IN	comm	通信器

```

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

Fortran MPI_GATHER (SEDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT,
                     RECVTYPE, ROOT, COMM, IERROR)
< type >    SEDBUF(*), RECVBUF(*)
INTEGER     SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE,
            ROOT, COMM, IERROR

```

函数 MPI_GATHER 规定：各进程将包含在各自消息缓存区 (sendbuf, sendcount, sendtype) 中具有相同类型图的数据单元发送给根进程；根进程接收这些消息，并按由小到大的进程序号，将这些消息依次存储在缓存区 (recvbuf, n × recvcount, recvtype) 中，其中 n 为该通信器 comm 包含的进程个数，recvcount 指根进程从其他每个进程接收的类型为 recvtype 的数据单元个数。显然，消息接收缓存区只对根进程有意义。

类似于消息广播函数 MPI_BCAST，函数 MPI_GATHER 要求每个进程提供相同的根进程序号 root 和通信器 comm，且参数 sendcount 与 sendtype 定义的类型图和参数 recvcount 与 recvtype 定义的类型图必须匹配。

其实，函数 MPI_GATHER 等价于：进程组每个成员均执行一个消息发送操作。

```
CALL MPI_Send(sendbuf, sendcount, sendtype, root, ...)
```

而根进程执行 n 个消息接收操作。

```

do i = 0, n - 1
    call MPI_RECV(recvbuf + i * recvcount * extent(recvtype), recvcount, recvtype, i, ...)
enddo

```

如图 5.2 所示，假设每个进程向根进程发送 100 个整型数，根进程接收它们，并存储在连续的内存空间，具体程序在例 5.3 中给出。



图 5.2 函数 MPI_GATHER 示意图

例 5.3 收集聚合通信函数应用示例

```
INTEGER A(100), RBUF(10000), SIZE, ROOT, RTYPE
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
IF (100 * SIZE.GT.10000) THEN
    PRINT *, "NOT ENOUGH RECEIVING BUF"
    CALL MPI_FINALIZE(IERR)
    STOP
ELSE
    CALL MPI_TYPE_CONTIGUOUS(100, MPI_INTEGER, RTYPE, IERR)
    ! 定义新的数据类型 RTYPE, 由 100 个连续的整型数据单元组成
    CALL MPI_TYPE_COMMIT(RTYPE, IERR)
    ROOT=0
    CALL MPI_GATHER(A, 100, MPI_INTEGER, RBUF, 1, RTYPE, ROOT, COMM, IERR)
    ! 从各进程收集 100 个整型数, 存放在数组 RBUF 中。
ENDIF
```

5.2.3 基于向量的消息收集函数

/* 向量收集聚合通信函数, 在进程组各成员中执行一个基于向量的消息收集操作。

```
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm)
IN      sendbuf      各进程提供的消息发送缓存区初始地址
IN      sendcount    各进程提供的消息发送缓存区内数据单元个数
IN      sendtype     各进程提供的消息发送缓存区内数据单元类型
OUT     recvbuf      根进程提供的消息接收缓存区初始地址
IN      recvcounts   数组, 包含从各进程接收的数据单元个数
IN      displs       数组, 包含存储各进程数据单元的初始地址
                  (以 recvtype 为单位)
IN      recvtype     消息接收缓存区内数据单元类型
IN      root         根进程的序号
IN      comm         通信器
C      int MPI_Gatherv(void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int *
                      recvcounts, int * displs, int MPI_Datatype
                      recvtype, int root, MPI_Comm comm)
Fortran MPI_GATHERV ( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
                      DISPLS, RECVTYPE, ROOT, COMM,
                      IERROR)
<type>    SENDBUF(*), RECVBUF(*)
INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNTS(*),
DISPLS(*), RECVTYPE, ROOT, COMM, IERROR
```

函数 MPI_GATHERV 扩展了函数 MPI_GATHER 的功能，允许各进程发送大小不一致的消息。参数 recvbuf, recvcounts 和 displs 只对根进程有效，其中，recvcounts(i) 表示根进程从第 i 个进程接收的消息所包含的类型为 recvtype 的数据单元个数，displs(i) 表示这些数据将从位置 $recvbuf + displs(i) * extent(recvtype)$ 开始连续存储。

函数 MPI_GATHERV 其余参数的含义与函数 MPI_GATHER 一致，这里不再讨论。

例 5.4 函数 MPI_GATHERV 应用示例

进程 i 向进程 0 发送 $100 - i$ 个整型数，进程 0 从各个进程接收消息，但每隔 100 个整型数依次存储这些消息。具体程序如下：

```
INTEGER A(100), RBUF(10000), SIZE, ROOT, RTYPE, RANK
INTEGER RECS(100), DISP(100)
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
CALL MPI_COMM_RANK(COMM, RANK, IERR)
IF (100 * SIZE .GT. 10000) THEN
    PRINT *, "NOT ENOUGH RECEIVING BUF"
    CALL MPI_FINALIZE(IERR)
    STOP
ELSE
    ROOT = 0
    IF(RANK .EQ. 0) THEN
        DO I = 1, SIZE
            RECS(I) = 101 - I
            DISP(I) = (I - 1) * 100
        ENDDO
    ENDIF
    CALL MPI_GATHERV(A, 100 - RANK, MPI_INTEGER, RBUF, RECS, DISP,
                      MPI_INTEGER, ROOT, COMM, IERR)
    ! 根进程从各进程收集长度不一致的多个整型数,
    ! 并按指定位置存储在数组 RBUF 中。
ENDIF
```

5.2.4 消息分发函数

/* 分发聚合通信函数，在进程组各成员中执行一个消息分发操作。

```
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm)
IN      sendbuf      根进程消息发送缓存区初始地址
IN      sendcount     根进程间各进程发送的数据单元个数
IN      sendtype      根进程消息发送缓存区内数据单元类型
OUT     recvbuf       各进程提供的消息接收缓存区初始地址
IN      recvcount     各进程从根进程接收的数据单元个数
```

```

IN      recvtype    各进程从根进程接收的数据单元类型
IN      root        根进程的序号
IN      comm        通信器
C      int MPI_Scatter(void * sendbuf, int sendcount, MPI_Datatype sendtype, void *
                      recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                      MPI_Comm comm)
Fortran MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                     RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
< type > SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
        ROOT, COMM, IERROR

```

函数 MPI_SCATTER 是函数 MPI_GATHER 的逆函数, 它等价于: 根进程执行 n 个消息发送操作:

```

do i = 1, n
    call MPI_Send(sendbuf + i * sendcount * extent(sendtype), sendcount, sendtype, i, ...),
enddo

```

其余进程执行一个消息接收操作:

```
call MPI_RECV(recvbuf, recvcount, recvtype, root, ...)
```

也就是说, 根进程将消息等分成 n 段, 并将第 i 段传送给第 i 个进程。参数 sendcount 与 sendtype 定义的类型图必须和参数 recvcount 与 recvtype 定义的类型图相匹配, 发送和接收的消息长度必须一致。其中, 消息发送缓存区只对根进程有意义。

如图 5.3 所示, 根进程将消息发送缓存区内数据单元等分成多段, 每段包含 100 个整型数, 依次发送给各个进程。具体程序在例 5.5 中给出。



图 5.3 MPI_SCATTER 示意图

例 5.5 分发聚合通信函数应用示例

```

INTEGER A(100), SBUF(10000), SIZE, ROOT, STYPE
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
IF (100 * SIZE .GT. 10000) THEN
    PRINT *, "NOT ENOUGH SENDING BUF"
    CALL MPI_FINALIZE(IERR)
    STOP
ELSE
    CALL MPI_TYPE_CONTIGUOUS(100, MPI_INTEGER, STYPE, IERR)
    ! 定义新的数据类型 STYPE, 由 100 个连续的整型数据单元组成

```

```

    CALL MPI_TYPE_COMMIT(STYPE, IERR)
    ROOT = 0
    CALL MPI_SCATTER(SBUF, 1, STYPE, A, 100, MPI_INTEGER,
                      ROOT, COMM, IERR)
    ! 各进程从根进程接收 100 个整型数, 存放在数组 A 中。
ENDIF

```

5.2.5 基于向量的消息分发函数

/* 向量分发聚合通信函数, 在进程组各成员中执行一个基于向量的消息分发操作。

```

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)
    IN      sendbuf      根进程提供的消息发送缓存区初始地址
    IN      sendcounts   数组, 含发送给各进程的消息包含的数据单元个数
    IN      displs       数组, 含发送给各进程的消息的初始地址(数据单元为单位)
    IN      sendtype     发送给各进程的消息包含的数据单元类型
    OUT     recvbuf      各进程提供的消息接收缓存区初始地址
    IN      recvcount    各进程从根进程接收的数据单元个数
    IN      recvtype     各进程从根进程接收的数据单元类型
    IN      root         根进程的序号
    IN      comm         通信器

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)

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

```

函数 MPI_SCATTERV 是函数 MPI_SCATTER 的推广, 也是函数 MPI_GATHERV 的逆函数, 它允许根进程从消息发送缓存区的任意指定位置开始, 向各个进程发送任意多个连续的 sendtype 型数据单元。其中, sendcounts(i) 表示根进程发送给第 i 个进程的数据单元个数, displs(i) 表示这些数据单元在消息发送缓存区内的起始地址。

例 5.6 函数 MPI_SCATTERV 应用示例

进程 0 按由小到大的进程序号, 每隔 100 个整型数, 向进程 i 发送 $100 - i$ 个整型数; 各个进程接收这些分发的数据, 存储在各自的局部数组中。具体程序如下:

```

INTEGER A(100), RBUF(10000), SIZE, ROOT, RANK
INTEGER SNDS(100), DISP(100)
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
CALL MPI_COMM_RANK(COMM, RANK, IERR)
IF(100 * SIZE.GT.10000) THEN

```

```

PRINT *, "NOT ENOUGH RECEIVING BUF"
CALL MPI_FINALIZE(IERR)
STOP
ELSE
    ROOT = 0
    IF(RANK.EQ.0) THEN
        DO I = 1,SIZE
            SNDS(I) = 101 - I
            DISP(I) = (I - 1) * 100
        ENDDO
    ENDIF
    CALL MPI_SCATTERV(RBUF,SNDS,DISP, MPI_INTEGER,A,100 - RANK,
                       MPI_INTEGER,ROOT,COMM,IERR)
ENDIF

```

5.2.6 消息全收集函数

/* 全收集聚合通信函数，在进程组各成员中执行一个全收集聚合通信操作。

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount,	
recvtype, comm)	
IN sendbuf	各进程提供的消息发送缓存区初始地址
IN sendcount	各进程提供的消息发送缓存区内数据单元个数
IN sendtype	各进程提供的消息发送缓存区内数据单元类型
OUT recvbuf	各进程提供的消息接收缓存区初始地址
IN recvcount	各进程从其他进程接收的数据单元个数
IN recvtype	各进程从其他进程接收的数据单元类型
IN comm	通信器

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

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

函数 MPI_ALLGATHER 是函数 MPI_GATHER 的推广，它要求将收集的结果存储在各个进程提供的消息接收缓存区中，也就是各个进程将获得完全一致的收集结果。除此之外，各参数含义和函数 MPI_GATHER 定义的一致。

例 5.7 全收集聚合通信函数 MPI_ALLGATHER 应用示例

每个进程从其他各个进程收集 100 个整型数据，按由小到大的进程序号，依次存储这

些数据到消息接收缓存区中。具体程序如下：

```
INTEGER A(100),RBUF(10000),SIZE,RTYPE
CALL MPI_COMM_SIZE(COMM,SIZE,IERR)
IF (100 * SIZE.GT.10000) THEN
    PRINT *, "NOT ENOUGH RECEIVING BUF"
    CALL MPI_FINALIZE(IERR)
    STOP
ELSE
    CALL MPI_TYPE_CONTIGUOUS(100,MPI_INTEGER,RTYPE,IERR)
    ! 定义新的数据类型 RTYPE, 由 100 个连续的整型数据单元组成。
    CALL MPI_TYPE_COMMIT(RTYPE,IERR)
    CALL MPI_ALLGATHER(A,100,MPI_INTEGER,RBUF,1,RTYPE,
                        COMM,IERR)
    ! 从各进程收集 100 个整型数, 存放在数组 RBUF 中。
ENDIF
```

在上例中, 函数 MPI_ALLGATHER 执行后, 每个进程的局部数组 RBUF 将拥有完全一致的前 $100 * \text{SIZE}$ 个元素; 而在例 5.3 中, 函数 MPI_GATHER 执行后, 仅进程 0 的局部数组 RBUF 的前 $100 * \text{SIZE}$ 个元素被改变。

5.2.7 基于向量的消息全收集函数

/* 向量全收集聚合通信函数, 在进程组各成员中执行一个基于向量的全收集聚合通信操作。

```
MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                 displs, recvtype, comm)
IN      sendbuf      各进程提供的消息发送缓存区初始地址
IN      sendcount    各进程提供的消息发送缓存区内数据单元个数
IN      sendtype     各进程提供的消息发送缓存区内数据单元类型
OUT     recvbuf      各进程提供的消息接收缓存区初始地址
IN      recvcounts   数组, 含从各进程接收的数据单元个数
IN      displs       数组, 含接收的数据单元的初始存储地址(数据单元为单位)
IN      recvtype     从各进程接收的数据单元类型
IN      comm         通信器
C      int MPI_Allgatherv(void * sendbuf, int sendcount, MPI_Datatype sendtype,
                         void * recvbuf, int * recvcounts, int * displs,
                         MPI_Datatype recvtype, MPI_Comm comm)
Fortran MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                       RECVCOUNTS, DISPLS, RECVTYPE, COMM,
                       IERROR)
<type>   SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*),
```

```
DISPLS( * ), RECVTYPE, COMM, IERROR
```

函数 MPI_ALLGATHERV 是函数 MPI_ALLGATHER, 或函数 MPI_GATHERV 的推广, 它允许每个进程提供的消息包含不同个数和不同类型的数据单元, 且允许每个进程将它从各个进程收集的数据从任意指定的位置开始连续存储。其中, recvcounts(i) 表示该进程将从第 i 个进程收集类型为 recvtype 的数据单元个数, 并且, 这些数据将从位置 displs(i) 开始连续存储在消息接收缓存区中。

例 5.8 函数 MPI_ALLGATHERV 的应用示例

各个进程从进程 i 收集 100 - i 个整型数, 并按由小到大的进程序号, 每隔 100 个整型数依次存储在消息接收缓存区中。具体程序如下:

```
INTEGER A(100), RBUF(10000), SIZE, RTYPE, RANK
INTEGER RECS(100), DISP(100)
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
CALL MPI_COMM_RANK(COMM, RANK, IERR)
IF (100 * SIZE.GT.10000) THEN
    PRINT *, "NOT ENOUGH RECEIVING BUF"
    CALL MPI_FINALIZE(IERR)
    STOP
ELSE
    DO I=1,SIZE
        RECS(I)=101-I
        DISP(I)=(I-1)*100
    ENDDO
    CALL MPI_ALLGATHERV(A, 100-RANK, MPI_INTEGER, RBUF, RECS, DISP,
                         MPI_INTEGER, COMM, IERR)
ENDIF
```

5.2.8 消息全交换函数

/* 全交换聚合通信函数, 在进程组各成员中执行一个全交换聚合通信操作。

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
               comm)
IN      sendbuf      各进程提供的消息发送缓存区初始地址
IN      sendcount     各进程向其他进程发送的数据单元个数
IN      sendtype      各进程向其他进程发送的数据单元类型
OUT     recvbuf       各进程提供的消息接收缓存区初始地址
IN      recvcount     各进程从其他进程接收的数据单元个数
IN      recvtype      各进程从其他进程接收的数据单元类型
IN      comm          通信器
C      int MPI_Alltoall(void * sendbuf, int sendcount, MPI_Datatype sendtype,
                      void * recvbuf, int recvcount, MPI_Datatype recvtype,
```

```

        MPI_Comm comm)
Fortran MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                      REVCOUNT, RECVTYPE, COMM, IERROR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE,
        COMM, IERROR

```

函数 MPI_ALLTOALL 等价于：首先，每个进程均执行一个分发聚合通信操作 MPI_SCATTER(sendbuf, sendcount, sendtype, …)，即将消息发送缓存区内所有数据按进程个数等分，按由小到大的进程序号，依次分发给各个进程；然后，每个进程均执行一个收集聚合通信操作 MPI_GATHER(…, recvbuf, recvcount, recvtype, …)，即从各个进程收集被分发给自身的数据，也同样按由小到大的进程序号，依次存储在消息接收缓存区中。显然，每个进程提供的参数 sendcount 与 sendtype 定义的类型图，必须和参数 recvcount 与 recvtype 定义的类型图匹配，否则，该函数将出错。

例 5.9 全交换聚合通信函数 MPI_ALLTOALL 应用示例

按由小到大的进程序号，每个进程从消息发送缓存区中向其他各个进程分发 100 个整型数，并从其他各个进程收集 100 个整型数，存储到消息接收缓存区中。具体程序如下：

```

INTEGER A(10000), RBUF(10000), SIZE, RTYPE
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
IF (100 * SIZE .GT. 10000) THEN
    PRINT *, "NOT ENOUGH RECEIVING BUF"
    CALL MPI_FINALIZE(IERR)
    STOP
ELSE
    CALL MPI_TYPE_CONTIGUOUS(100, MPI_INTEGER, RTYPE, IERR)
    CALL MPI_TYPE_COMMIT(RTYPE, IERR)
    CALL MPI_ALLTOALL(A, 100, MPI_INTEGER, RBUF, 1, RTYPE,
                      COMM, IERR)
ENDIF

```

5.2.9 基于向量的消息全交换函数

```

/* 向量全交换聚合通信函数，在进程组各成员中执行一个基于向量的全交换聚合通信
/* 操作。

```

```

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype,
               comm)
IN      sendbuf      各进程提供的消息发送缓存区初始地址
IN      sendcounts   数组，含向各进程发送的数据单元个数
IN      sdispls     数组，含向各进程发送的数据的初始存储地址(单元为单位)

```

IN	sendtype	向各进程发送的数据单元类型
OUT	recvbuf	各进程提供的消息接收缓存区初始地址
IN	recvcounts	数组,含从各进程接收的数据单元个数
IN	rdispls	数组,含从各进程接收的数据的初始存储地址(单元为单位)
IN	recvtype	从各进程接收的数据单元类型
IN	comm	通信器

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)

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

函数 MPI_ALLTOALLV 是函数 MPI_ALLTOALL 的扩展, 它等价于: 每个进程首先执行一个基于向量的分发聚合通信操作 MPI_SCATTERV, 然后执行一个基于向量的收集聚合通信操作 MPI_GATHERV。其中, 元素 sendcounts(i) 表示分发给第 i 个进程的消息包含的数据单元个数, sdispls(i) 表示这些数据在消息发送缓存区的首地址; 元素 recvcounts(j) 表示从第 j 个进程接收的消息包含的数据单元个数, rdispls(j) 表示这些数据在消息接收缓存区中的首地址。

例 5.10 函数 MPI_ALLTOALLV 的应用示例

按由小到大的进程序号, 每隔 100 个整型数, 进程 i 依次给进程 j 分发 $100 - i - j$ 个整型数, 且从进程 j 收集 $100 - i - j$ 个整型数。具体程序如下:

```

INTEGER A(10000), RBUF(10000), SIZE, RANK
INTEGER SNDS(100), SDISP(100), RECS(100), RDISP(100)
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
CALL MPI_COMM_RANK(COMM, RANK, IERR)
IF (100 * SIZE.GT.10000) THEN
  PRINT *, "NOT ENOUGH RECEIVING BUF"
  CALL MPI_FINALIZE(IERR)
  STOP
ELSE
  DO I=1,SIZE
    SNDS(I)=101-I-RANK
    RECS(I)=101-I-RANK
    SDISP(I)=(I-1)*100
    RDISP(I)=(I-1)*100
  END DO
END IF

```

```

ENDDO
CALL MPI_ALLTOALLV(A, SNDS, SDISP, MPI_INTEGER, RBUF, RECS, DISP,
MPI_INTEGER, COMM, IERR)
ENDIF

```

5.3 全局归约函数

全局归约(global reduction)函数是指:执行 MPI 程序的每个进程均提供一个局部变量,并对这些局部变量执行某种归约操作,例如求累加和、最大值、最小值等,得到一个新值之后,将该值存储在某个或所有进程中。因此,全局归约要求进程组所有成员的共同参与,是聚合通信的一种特殊情形。

如图 5.4 所示,全局归约函数可以分为四类:

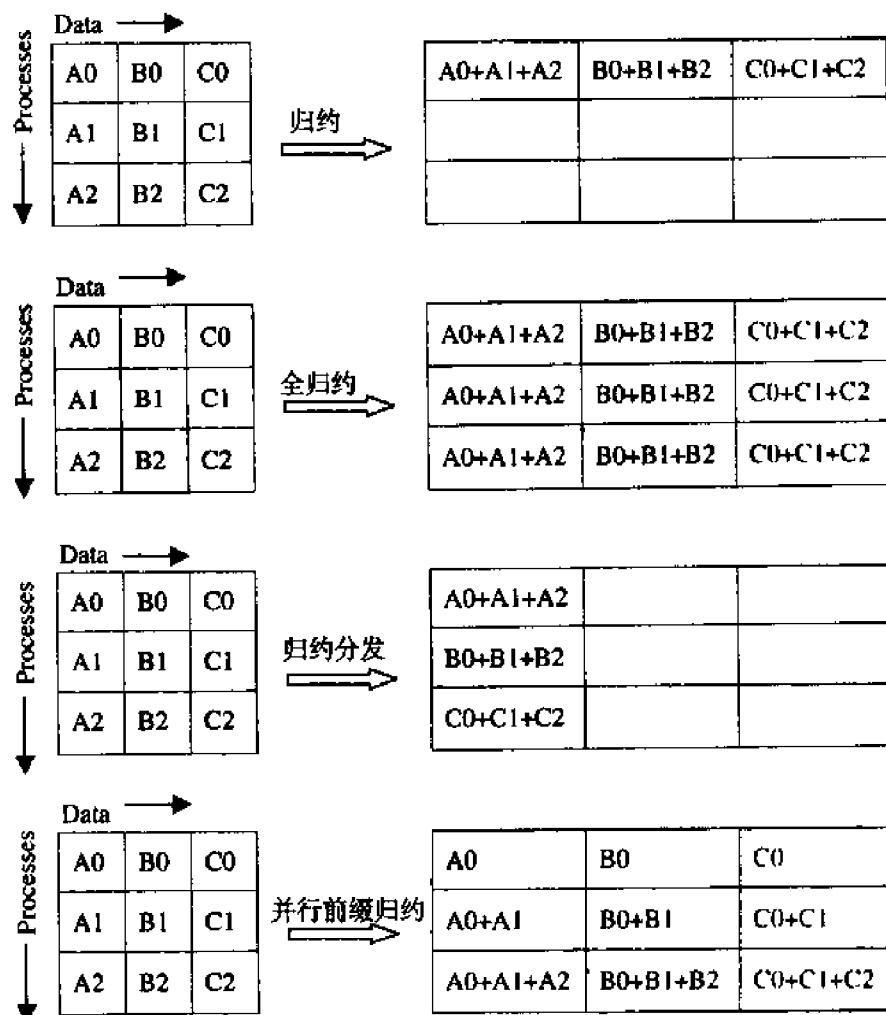


图 5.4 四类全局归约函数示意图。其中,横向表示各个进程提供的局部变量,纵向表示各个进程,归约操作为求累加和

- (1) 归约(reduce)函数 MPI _ REDUCE; 将各进程提供的局部变量归约, 归约结果只存储在根进程中;
- (2) 全归约(all-reduce)函数 MPI _ ALLREDUCE; 将各进程提供的局部变量归约, 将归约结果存储在所有进程中;
- (3) 归约分发(reduce-scatter)函数 MPI _ REDUCE _ SCATTER; 将各进程提供的局部变量归约, 将归约结果分发到各进程中;
- (4) 并行前缀归约(scan)函数 MPI _ SCAN; 对各进程提供的局部变量, 按由小到大的进程序号, 执行一个并行前缀归约, 并将结果存储在各进程中。

5.3.1 归约函数

/* 归约聚合通信函数, 对各个进程提供的局部变量执行一个全局归约操作, 并将结果
/* 存储在根进程的消息接收缓存区中。

```

MPI _ REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
    IN      sendbuf      各进程提供的消息发送缓存区初始地址
    OUT     recvbuf      各进程提供的消息接收缓存区初始地址
    IN      count        各进程提供的待归约的数据单元个数
    IN      datatype     各进程提供的待归约的数据单元类型
    IN      op           归约操作
    IN      root         根进程序号
    IN      comm         通信器

C      int MPI _ Reduce(void * sendbuf, void * recvbuf, int count, MPI _ Datatype
                      datatype, MPI _ Op op, int root, MPI _ Comm comm)

Fortran MPI _ REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT,
                      COMM, IERROR)
< type >   SENDBUF(*), RECVBUF(*)
INTEGER     COUNT, DATATYPE, OP, ROOT, COMM, IERROR

```

函数 MPI _ REDUCE 针对每个进程提供的包含在消息发送缓存区 (sendbuf, count, datatype) 中的数据单元, 按进程序号逐一执行 op 归约操作, 并将结果存储在根进程 root 的消息接收缓存区 (recvbuf, count, datatype) 中。其中, 各进程提供的参数 op, root 和 comm 必须相同, 而参数 count 和 datatype 定义的类型图必须匹配。这里, 我们要注意到两种情况:(1)如果每个进程只提供一个数据单元, 即 count = 1, 则形成的结果也仅为一个数据单元;(2)如果每个进程提供一个数组, 即 count > 1, 则归约操作对数组的对应元素逐一进行, 得到的结果也是一个长度为 count 的数组。

函数 MPI _ REDUCE 规定: 消息发送缓存区和消息接收缓存区必须互不重叠, 且提供的数据类型必须与提供的归约操作匹配(具体匹配规则在下面讨论)。在函数 MPI _ REDUCE 中, 消息接收缓存区仅对根进程有意义, 敬请读者注意。

表 5.1 列出了 MPI 系统预先定义的所有归约操作, 它们均满足结合律和交换律。MPI 系统规定, 在函数 MPI _ REDUCE 中, 各个进程提交的数据类型必须和归约操作相

互匹配。具体为：最大值 MPI_MAX、最小值 MPI_MIN 匹配整型变量、浮点型变量；累加和 MPI_SUM 匹配整型变量、浮点型变量和复型变量；逻辑与 MPI_BAND、逻辑或 MPI_LOR、逻辑异或 MPI_LXOR 匹配整型变量和逻辑型变量；按位与 MPI_BAND、按位或 MPI_BOR、按位异或 MPI_BXOR 匹配整型变量和字节变量；内积 MPI_PROD 匹配整型变量、浮点型变量和复型变量，并求由各个进程提供的局部变量组成的向量的内积；MPI_MAXLOC 和 MPI_MINLOC 匹配的数据类型比较复杂，我们在下一小节讨论。

表 5.1 MPI 系统定义的归约操作

OP	含 义	OP	含 义
MPI_MAX	最大值	MPI_MIN	最小值
MPI_SUM	累加和	MPI_PROD	内积
MPI_BAND	逻辑与	MPI_BAND	按位与
MPI_LOR	逻辑或	MPI_BOR	按位或
MPI_LXOR	逻辑异或	MPI_BXOR	按位异或
MPI_MAXLOC	最大值及相关信息	MPI_MINLOC	最小值及相关信息

例 5.11 函数 MPI_REDUCE 应用示例

求分布存储在各个进程的两个向量的内积，并将结果存储在进程 0 中。

```

SUBROUTINE PAR_BLASI (m, a, b, c, comm)
  REAL      a(m), b(m)      ! 该进程分布存储的两个向量的 m 个局部元素
  REAL      c                  ! 两个向量的内积，仅对进程 0 有意义。
  REAL      sum
  INTEGER m, comm, i, ierr
  !
  ! 求两个向量分布在该进程的 m 个元素的局部内积。
  sum = 0.0
  DO i = 1, m
    sum = sum + a(i) * b(i)
  ENDDO
  !
  ! 对各个进程的局部内积 sum 执行全局归约累加和操作，
  ! 并将结果存储在进程 0 的变量 c 中。
  CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
  !
  RETURN

```

例 5.12 函数 MPI_REDUCE 应用示例

已知一个分布存储在各进程的向量和一个按行块分配在各进程的矩阵，求该向量与

矩阵的乘积，并存储在进程 0 中。

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m)          ! 该进程分布存储向量的 m 个局部元素。
REAL b(m, n)        ! 该进程分布存储矩阵的 m 行，每行包含 n 个元素。
REAL c(n)
! 向量与矩阵的乘积: c(1:n) = a(1:M) × b(1:M, 1:n)，其中 M=m×p, p 为进程个数。
REAL sum(n)         ! 各进程中向量与矩阵的局部乘积。
INTEGER m, n, comm, i, j, ierr
!
! 各进程求向量与矩阵的局部乘积。
DO j = 1, n
    sum(j) = 0.0
    DO i = 1, m
        sum(j) = sum(j) + a(i) * b(i, j)
    ENDDO
ENDDO
!
! 全局累加和归约求向量与矩阵的乘积，并将结果存储在进程 0 中。
CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
!
RETURN
```

5.3.2 归约操作 MPI_MAXLOC 和 MPI_MINLOC

归约操作 MPI_MAXLOC 和 MPI_MINLOC，对各进程提供的局部变量分别求全局最大值和最小值，并返回附属于该值的某个信息，例如拥有该值的进程的序号。因此，它们的操作数必须包含两个变量，其中一个是为了求最大值和最小值的变量，另一个是附属于该变量的信息，该信息可为整型变量，也可为浮点型变量。

具体地，归约操作 MPI_MAXLOC 可定义为：

$$\begin{pmatrix} u \\ i \end{pmatrix} \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

其中 $w = \max(u, v)$ ，且：

$$k = \begin{cases} i & u > v \\ \min(i, j) & u = v \\ j & u < v \end{cases}$$

其中，变量 u, v 为待比较变量，变量 i, j 分别为附属于变量 u, v 的信息。将以上定义的变量 u, v 的最小值比较改为最大值比较，则可以类似地定义操作 MPI_MINLOC。

由于归约操作 MPI_MAXLOC 和 MPI_MINLOC 的操作数必须是两个变量，因此，MPI 系统为它们定义了专门的数据类型，具体在表 5.2 列出的。

表 5.2 与操作 MPI_MAXLOC 与 MPI_MINLOC 匹配的数据类型

C		FORTRAN	
数据类型	描述	数据类型	描述
MPI_FLOAT_INT	float 与 int	MPI_2REAL	一对 REAL
MPI_LONG_DOUBLE_INT	long double 与 int	MPI_2DOUBLE_PRECISION	一对 DOUBLE PRECISION
MPI_LONG_INT	long 与 int	MPI_2INTEGER	一对 INTEGER
MPI_2INT	一对 int		
MPI_SHORT_INT	short 与 int		
MPI_DOUBLE_INT	double 与 int		

例 5.13 操作 MPI_MAXLOC 应用示例

每个进程提供 30 个局部双精度数, 对它们执行全局归约最大值比较操作, 并获得拥有最大值的进程的序号, 将结果存储在进程 0 中。

```

DOUBLE PRECISION ain(30)      ! 各进程提供的局部 30 个双精度型数据。
DOUBLE PRECISION aout(30)     ! 最大值归约结果数组。
INTEGER ind(30)              ! 拥有各个最大值的进程的序号。
DOUBLE PRECISION in(2,30),    out(2,30)
                                ! 操作数, 定义为 - 对双精度型数据。

INTEGER i, myrank, root, ierr;
!
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr);
DO i = 1, 30
    in(1,i) = ain(i)
    in(2,i) = myrank
ENDDO
! 执行归约操作 MPI_MAXLOC
CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION,
                MPI_MAXLOC, root, comm, ierr);
!
IF (myrank.EQ.root) THEN
    DO i = 1, 30
        aout(i) = out(1,i)      ! 输出比较获得的各个最大值。
        ind(i) = out(2,i)       ! 输出拥有最大值的进程序号。
    ENDDO
ENDIF

```

5.3.3 全归约函数

```

/* 全归约聚合通信函数, 对各个进程提供的局部变量执行一个全局归约操作, 并将
/* 结果存储在各进程提供的消息接收缓存区中。

```

```
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)
```

IN	sendbuf	各进程提供的消息发送缓存区初始地址
OUT	recvbuf	各进程提供的消息接收缓存区初始地址
IN	count	各进程提供的消息发送缓存区内数据单元个数
IN	datatype	各进程提供的消息发送缓存区内数据单元类型
IN	op	归约操作
IN	comm	通信器

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

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

类似于函数 MPI_REDUCE, 全归约函数 MPI_ALLREDUCE 可对各进程提供的局部变量执行各种归约操作, 但不同的是, 它要求将归约结果存储在所有进程的消息接收缓存区中。

例 5.14 函数 MPI_ALLREDUCE 应用示例

参考例 5.12, 要求将向量与矩阵的乘积存储在所有进程中。

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m)                 ! 该进程分布存储向量的 m 个局部元素。
REAL b(m, n)             ! 该进程分布存储矩阵的 m 行, 每行包含 n 个元素。
REAL c(n)
! 向量与矩阵的乘积: c(1:n) = a(1:M) × b(1:M, 1:n), 其中 M = m × p, p 为进程个数。
REAL sum(n)             ! 各进程中向量与矩阵的局部乘积。
INTEGER m, n, comm, i, j, ierr
!
! 各进程求向量与矩阵的局部乘积。
DO j=1, n
    sum(j) = 0.0
    DO i=1, m
        sum(j) = sum(j) + a(i) * b(i, j)
    ENDDO
ENDDO
!
! 全局累加和归约求向量与矩阵的乘积, 并将结果存储在所有进程中。
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
!
RETURN
```

5.3.4 归约分发函数

```
/* 归约分发聚合通信函数, 对各个进程提供的局部变量执行一个全局归约操作,
/* 并将结果分发到各进程提供的消息接收缓存区中。
```

```
MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)
```

IN	sendbuf	各进程提供的消息发送缓存区初始地址
OUT	recvbuf	各进程提供的消息接收缓存区初始地址
IN	recvcounts	整型数组(见下面解释)
IN	datatype	各进程提供的数据单元类型
IN	op	归约操作
IN	comm	通信器

函数 MPI_REDUCE_SCATTER 等价于：首先，在各进程提供的消息发送缓存区 (sendbuf, count, datatype) 包含的数据单元中执行归约操作 op，其中 $count = \sum_i recvcounts(i)$, p 为进程个数, 表示提供的数据单元个数；然后，将归约结果分成 n 个连续的段，其中第 i 段包含 $recvcounts(i)$ 个数据单元，并存储在第 i 个进程的消息接收缓存区中。

例 5.15 函数 MPI_REDUCE_SCATTER 应用示例

参考例 5.12，要求将向量与矩阵的乘积分发到所有进程中。

5.3.5 并行前缀归约函数

/* 并行前缀归约聚合通信函数, 对各个进程提供的局部变量执行一个并行前缀归约,
/* 并将结果分别存储在各进程提供的消息接收缓存区中。

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)
IN sendbuf 各进程提供的消息发送缓存区初始地址
OUT recvbuf 各进程提供的消息接收缓存区初始地址
IN count 各进程提供的数据单元个数
IN datatype 各进程提供的数据单元类型
IN op 归约操作
IN comm 通信器

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

Fortran MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR

函数 MPI_SCAN 对各进程提供的消息发送缓存区 (sendbuf, count, datatype) 内数据单元, 按由小到大的进程序号与操作 op, 执行并行前缀计算, 其中第 i 个进程将获得进程 0 到进程 i 的归约结果, 具体各参数的含义与函数 MPI_REDUCE 一致。

例 5.16 函数 MPI_SCAN 应用示例

各个进程提供一个局部整型变量, 然后对这些变量执行并行前缀累加和归约。具体程序可写成:

```
INTEGER A ! 各进程提供的局部变量
INTEGER C ! 各进程获得的并行前缀归约结果
CALL MPI_SCAN(A, C, 1, MPI_INTEGER, MPI_SUM, COMM, IERR) ! 并行前缀归约
```

表 5.3 列出了 6 个进程时的情形, 其中, 第 2 行列出了各个进程提供的局部变量, 第 3 行列出了相应的并行前缀累加和归约结果。

表 5.3 并行前缀累加和归约示意表

进程	0	1	2	3	4	5
A	2	3	1	5	12	2
C	2	5	6	1	13	15

5.3.6 自定义归约操作

除了表 5.1 定义的归约操作, MPI 系统还提供特殊的函数, 辅助用户创建满足特殊应用需求的归约操作, 以方便并行程序设计和提高并行计算的性能。

```

/* 归约操作创建函数, 创建一个归约操作。

MPI_OP_CREATE(function, commute, op)
    IN      function      外部函数, 用于定义特殊的归约操作
    IN      commute       若 true, 则归约操作可交换
    OUT     op            归约操作

C     int MPI_Op_create(MPI_User_function *function, int commute,
                      MPI_Op *op)

Fortran MPI_OP_CREATE(FUNCTION, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL,   COMMUTE
    INTEGER     OP, IERROR

```

基于外部函数 function, 函数 MPI_OP_CREATE 创建一个新的归约操作 op, 可充当上述四个归约函数 MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER 和 MPI_SCAN 的归约操作参数, 称之为自定义归约操作。MPI 系统规定, 自定义归约操作必须是可结合的, 如果参数 commute = true, 则该操作还是可交换的。

函数 function 为自定义的外部函数, 其一般形式为:

```

typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
                               MPI_Datatype *datatype)
FUNCTION USER_FUNCTION(INVEC(*), INOUTVEC(*), LEN,
                      DATATYPE)
<type>      INVEC(LEN), INOUTVEC (LEN)
INTEGER      LEN, DATATYPE

```

其中, 参数 datatype 为上述四类归约函数允许的数据类型, len 为数组 invec 和数组 inoutvec 的长度, 数组 invec 和数组 inoutvec 分别是由操作数组成的序列, 函数 function 将输出:

$$\text{inoutvec}(i) = \text{invec}(i) \oplus \text{inoutvec}(i) \quad i=1, \dots, \text{len}$$

其中, 符号 \oplus 为函数定义的归约操作。

函数 MPI_OP_CREATE 可用于定义一个自定义归约操作, 反之, 函数 MPI_OP_FREE 可用于释放一个已经定义好的自定义归约操作。

```

/* 自定义归约操作释放函数, 释放一个自定义归约操作。

```

```

MPI_OP_FREE(op)
    IN      op            自定义归约操作

C     int MPI_Op_free(MPI_Op *op)

Fortran MPI_OP_FREE(OP, IERROR)
    INTEGER OP, IERROR

```

例 5.17 自定义归约操作应用示例

每个进程提供长度为 3 的局部实型数组, 要求所有进程对第 1 个元素执行最大值全

归约,对第 2 个元素执行最小值全归约,对第 3 个元素执行累加和全归约。有两种实现办法:(1)简单地调用 3 次全归约函数 MPI_ALLREDUCE,但这样会引起多次全局同步,导致并行计算性能的下降;(2)利用函数 MPI_OP_CREATE 创建一个新的归约操作,然后调用全归约函数来完成。下面,我们给出第 2 种方法的具体程序实现方法:

```
REAL A(3),B(3)
INTEGER USEROP
EXTERNAL      FUNC
CALL MPI_OP_CREATE(FUNC,.TRUE.,USEROP,IERR)
! 创建自定义归约操作 USEROP,它满足交换律。
CALL MPI_ALLREDUCE(A,B,3,MPI_REAL,USEROP,COMM,IERR)
! 使用自定义归约操作 USEROP 的全归约。
CALL MPI_OP_FREE(USEROP,IERR)
! 释放自定义归约操作 USEROP。
.....
END
!
! 定义归约操作 USEROP
FUNCTION      FUNC(INV,OUTV,LEN,TYPE,IERR)
INTEGER        LEN, TYPE
REAL          INV(LEN),OUTV(LEN)
!
OUTV(1) = MAX (INV(1),OUTV(1))
OUTV(2) = MIN (INV(2),OUTV(2))
OUTV(3) = INV(3)+OUTV(3)
!
RETURN
END
!
```

第六章 进程通信器

为了帮助用户理解 MPI 系统内部的通信机制,本书第二章曾简单地定义了进程组和通信器,本章在此基础上,进一步介绍与它们相关的各方面内容。

进程组 (process group) 是 MPI 系统内部一类有序进程的集合,且每个进程均具有唯一的序号。进程组属于 MPI 对象,在 C 语言中,其数据类型为 MPI_Group;在 Fortran 语言中,其数据类型为整型变量。进程组可以通过 MPI 函数进行创建和释放,不同进程组之间还可以相互比较。

MPI 系统规定,任何点对点通信和聚合通信均必须基于某个通信域 (communication domain) 进行,而通信域是指相互之间能进行点对点通信的所有进程的集合。通信器是建立在进程组上的具有多种属性的 MPI 对象,是进程通信域的具体表现形式。每个通信器均属于一个通信域,在该域内,任意进程之间可以进行点对点通信,反之,一个通信域可由一个或多个通信器来表示。

如果某个通信域只涉及同一进程组内部成员之间的通信,则称该通信域内发生的通信为**域内通信** (intracommunication),且称相应通信器为**域内通信器** (intracommunicator),例如前面各章例子程序中提到的各类通信,均为域内通信;如果某个通信域涉及到两个不同进程组成员之间的通信,则称该类通信域内发生的通信为**域间通信** (intercommunication),且称相应通信器为**域间通信器** (intercommunicator)。

任何域内通信器均具有两个固定属性:一个是它所包含的**进程组**,另一个是描述该进程组内各成员联接状态的拓扑结构(见第七章);而域间通信器只定义了两个不相交进程组之间的点对点通信域,它只有一个固定属性,就是构成它的两个进程组,但不存在任何进程间联接的拓扑结构,也不支持任何聚合通信函数。在本书的讨论中,除非特别说明,否则通信器均是指域内通信器。

特别地,MPI 系统初始化函数 MPI_INIT 为每个执行 MPI 程序的进程创建两个通信器:第一个是 MPI_COMM_WORLD,称之为**全局通信器**,它包含由命令 mpirun 初始启动的所有进程,且基于该通信器,各个进程之间可以组织任何形式的点对点通信和聚合通信;第二个是为每个进程创建的**局部通信器** MPI_COMM_SELF,它是仅包含进程自身的通信器。

基于这两个初始创建的通信器,MPI 系统允许用户通过特殊的函数创建新的通信器,以方便并行程序设计和提高并行计算性能。而且,MPI 系统还提供多个辅助函数,允许用户访问、比较和释放新的通信器,以及它所包含的进程组。

本章第 1 节和第 2 节分别介绍有关进程组和域内通信器的 MPI 函数,第 3 节介绍域间通信器。

6.1 进程组管理

进程组的管理涉及进程组的创建、访问、比较和释放，它们均可以由各个进程独立完成，属于局部函数，不涉及进程间的通信。

6.1.1 进程组创建

MPI 系统规定，进程组的创建属于局部函数：一个进程可独立地创建多个进程组，甚至创建不包含其自身的进程组，且进程组的创建不涉及其他进程的状态，也不需要任何进程间的通信。

MPI 系统提供两个进程组常数 MPI_GROUP_EMPTY 和 MPI_GROUP_NULL，分别代表不包含任何进程的空进程组和无效的进程组。同时，类似于通信器 MPI_COMM_WORLD 和通信器 MPI_COMM_SELF，MPI 系统也提供两个初始的进程组，分别附属于这两个通信器，可通过如下函数获得。

```
/* 初始进程组获取函数，获取某个通信器包含的进程组。
```

```
MPI_COMM_GROUP (comm, group)
    IN      comm      通信器
    Out     group     通信器包含的进程组
C      int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)
Fortran MPI_COMM_GROUP (COMM, GROUP, IERROR)
        INTEGER COMM, GROUP, IERROR
```

由此，我们可以通过如下的函数调用：

```
INTEGER INIT_GROUP_GLOBAL
CALL MPI_COMM_GROUP (MPI_COMM_WORLD, INIT_GROUP_GLOBAL, IERR)
来获得 MPI 系统提供的初始进程组 INIT_GROUP_GLOBAL，它包含由命令 mpirun 启动的所有进程，以及赋予各个进程的唯一序号；同时，通过如下的函数调用：
```

```
INTEGER INIT_GROUP_SELF
CALL MPI_COMM_GROUP (MPI_COMM_SELF, INIT_GROUP_SELF, IERR)
来获得 MPI 系统提供的初始进程组 INIT_GROUP_SELF，它仅包含调用进程本身。
```

MPI 系统只支持从某个或某些已经存在的进程组出发，创建一个新的进程组，它不支持凭空地创建一个新的进程组。由此，前面介绍的两个进程组 INIT_GROUP_GLOBAL 和 INIT_GROUP_SELF 非常有用，因为任何非空、有效的进程组的创建均必须从它们出发。下面，我们逐一介绍 MPI 系统提供的进程组创建函数。

```
/* 进程组并集创建函数，基于两个进程组成员的并集，创建一个新的进程组。
```

```
MPI_GROUP_UNION (group1, group2, newgroup)
    IN      group1      进程组 1
    IN      group2      进程组 2
```

```

          OUT      newgroup    新进程组
C      int MPI_Group_union (MPI_Group group1, MPI_Group group2, MPI_Group * newgroup)
Fortran MPI_GROUP_UNION (GROUP1, GROUP2, NEWGROUP, IERROR)
        INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

/* 进程组交集创建函数, 基于两个进程组成员的交集, 创建一个新的进程组。

   MPI_GROUP_INTERSECTION (group1, group2, newgroup)
     IN      group1    进程组 1
     IN      group2    进程组 2
     OUT      newgroup  新进程组
C      int MPI_Group_intersection (MPI_Group group1, MPI_Group group2,
                                     MPI_Group * newgroup)
Fortran MPI_GROUP_INTERSECTION (GROUP1, GROUP2, NEWGROUP, IERROR)
        INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

/* 进程组差集创建函数, 基于两个进程组成员的差集, 创建一个新的进程组。

   MPI_GROUP_DIFFERENCE (group1, group2, newgroup)
     IN      group1    进程组 1
     IN      group2    进程组 2
     OUT      newgroup  新进程组
C      int MPI_Group_difference (MPI_Group group1, MPI_Group group2, MPI_Group * newgroup)
Fortran MPI_GROUP_DIFFERENCE (GROUP1, GROUP2, NEWGROUP, IERROR)
        INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

```

以上三个进程组创建函数, 分别具有如下含义:

- MPI_GROUP_UNION: 进程组 group1 的所有成员, 再加上进程组 group2 中不属于进程组 group1 的所有进程, 组成新进程组 newgroup。新进程组中的进程重新排序, 先按由小到大的次序排 group1 中的进程, 接着按由小到大的次序排不属于 group1 中的 group2 中的进程;
- MPI_GROUP_INTERSECTION: 进程组 group1 中属于进程组 group2 的所有进程, 组成新进程组, 并以它们在进程组 group1 中的序号, 按由小到大的次序重新排序;
- MPI_GROUP_DIFFERENCE: 进程组 group1 中不属于进程组 group2 的所有进程, 组成新进程组, 并以它们在进程组 group1 中的序号, 按由小到大的次序重新排序。

例 6.1 进程组创建函数应用示例

```

INTEGER GROUP1      ! 进程组 GROUP1, 包含进程{a, b, c, d}。
INTEGER GROUP2      ! 进程组 GROUP2, 包含进程{d, a, e}。
INTEGER NEWGROUP1, NEWGROUP2, NEWGROUP3 ! 新进程组。
INTEGER RANK

```

```

CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
IF(RANK.EQ.0)THEN
    CALL MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP1, IERR)
    ! 并集创建新进程组 NEWGROUP1, 包含进程{a, b, c, d, e}。
ELSE IF( RANK.EQ.1)THEN
    CALL MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP2, IERR)
    ! 交集创建新进程组 NEWGROUP2, 包含进程{a, d}。
ELSE IF(RANK.EQ.2)THEN
    CALL MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP3, IERR)
    ! 差集创建新进程组 NEWGROUP3, 包含进程{b, c}。
ENDIF

```

在上例中,三个进程分别独立地创建了三个不同的进程组,可知,进程组的创建属于局部函数,它不涉及任何进程之间的通信。

/* 进程组子集创建函数,基于某个进程组成员的子集,创建一个新的进程组。

```

MPI_GROUP_INCL (group, n, ranks, newgroup)
    IN      group      进程组
    IN      n          新进程组包含的进程个数
    IN      ranks     新进程组包含的进程在 group 中的序号组成的数组
    OUT     newgroup   新进程组
C     int MPI_Group_incl (MPI_Group group, int n, int * ranks, MPI_Group * newgroup)
Fortran MPI_GROUP_INCL (GROUP, N, RANKS, NEWGROUP, IERROR)
        INTEGER GROUP, N, RANKS (*), NEWGROUP, IERROR

```

函数 MPI_GROUP_INCL 创建新进程组 newgroup,由原进程组 group 内数组 ranks 包含的 n 个进程组成,且保持它们在进程组 group 中的次序;若 n=0,则返回空进程组 newgroup=MPI_GROUP_EMPTY。

/* 进程组补集创建函数,基于某个进程组成员的补集,创建一个新的进程组。

```

MPI_GROUP_EXCL (group, n, ranks, newgroup)
    IN      group      进程组
    IN      n          进程组 group 中不属于新进程组的进程个数
    IN      ranks     进程组 group 中不属于新进程组的进程序号组成的数组
    OUT     newgroup   新进程组
C     int MPI_Group_excl (MPI_Group group, int n, int * ranks, MPI_Group * newgroup)
Fortran MPI_GROUP_EXCL (GROUP, N, RANKS, NEWGROUP, IERROR)
        INTEGER GROUP, N, RANKS (*), NEWGROUP, IERROR

```

函数 MPI_GROUP_EXCL 创建新进程组 newgroup,由原进程组 group 内删除数组 ranks 包含的 n 个进程后剩余的进程组成,且保持它们在 group 中的次序。

例 6.2 进程组创建函数应用示例

```
INTEGER GROUP    ! 进程组 GROUP, 包含进程{a, b, c, d, e, f, g, h, i}。
INTEGER NEWGROUP  ! 新进程组。
INTEGER RANK, RANKS(4)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
RANKS(1)=0
RANKS(2)=3
RANKS(3)=7
RANKS(4)=4
IF(RANK.EQ.0)THEN
    CALL MPI_GROUP_INCL(GROUP, 4, RANKS, NEWGROUP, IERR)
    ! 子集创建新进程组 NEWGROUP, 包含 4 个进程{a, d, e, h}。
ELSE IF(RANK.EQ.1)THEN
    CALL MPI_GROUP_INCL(GROUP, 0, RANKS, NEWGROUP, IERR)
    ! 子集创建新进程组 NEWGROUP, 为空进程组 MPI_GROUP_EMPTY。
ELSE IF(RANK.EQ.2)THEN
    CALL MPI_GROUP_EXCL(GROUP, 4, RANKS, NEWGROUP, IERR)
    ! 补集创建新进程组 NEWGROUP, 包含 5 个进程{b, c, f, g, i}。
ENDIF

/* 进程组子集创建函数, 基于某个数组包含的多个子集进程组,
/* 创建一个包含所有这些子集的新进程组。


MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
    IN      group      进程组
    IN      n          进程组子集的个数
    IN      ranges     二维数组, 其每 3 个元素说明一个进程组子集
    OUT     newgroup   由进程组所有子集组成的新进程组
C      int MPI_Group_range_incl (MPI_Group group, int n, int ranges[][3],
                                MPI_Group *newgroup)
Fortran MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3, N), NEWGROUP, IERROR
```

函数 MPI_GROUP_RANGE_INCL 基于二维数组 ranges(3, n) 所说明的进程组 group 中的 n 个子集, 来创建新进程组 newgroup。对 Fortran 语言, 假设数组 ranges 包含如下元素:

```
{ (first1, 1), (last1, 1), (stride1, 1) }
{ (first2, 2), (last2, 2), (stride2, 2) }
...
{ (firstn, n), (lastn, n), (striden, n) }
```

分别定义了 n 个子集, 由如下序号的进程组成:

子集 1 = {first₁, first₁ + stride₁, …, first₁ + ⌊(last₁ - first₁) / stride₁⌋ × stride₁}

子集 2 = {first₂, first₂ + stride₂, …, first₂ + ⌊(last₂ - first₂) / stride₂⌋ × stride₂}

.....

子集 n = {first_n, first_n + stride_n, …, first_n + ⌊(last_n - first_n) / stride_n⌋ × stride_n}

其中, first_i, last_i 均为进程的序号, 而 stride_i 为正整数。于是, 新进程组 newgroup 由以上 n 个子集包含的所有进程组成, 且保持它们在子集中说明的次序不变。特别地, 各个子集说明的进程必须互不相同, 否则程序调用将出错。

```
/* 进程组补集创建函数, 基于某个数组包含的进程组多个子集,
/* 创建一个由不属于所有这些子集的进程构成的新进程组。
```

```
MPI_GROUP_RANGE_EXCL (group, n, ranges, newgroup)
    IN      group      进程组
    IN      n          进程组子集个数
    IN      ranges     一维数组, 其每 3 个元素说明一个进程组子集
    OUT     newgroup   由不属于所有这些子集的进程构成的新进程组
C      int MPI_Group_range_excl (MPI_Group group, int n, int ranges[][3],
                                MPI_Group *newgroup)
Fortran MPI_GROUP_RANGE_EXCL (GROUP, N, RANGES, NEWGROUP, IERROR)
      INTEGER GROUP, N, RANGES (3, N), NEWGROUP, IERROR
```

在进程组 group 中, 函数 MPI_GROUP_RANGE_EXCL 删除由数组 ranges 说明的进程, 并将所有剩余的进程, 按它们在 group 中的序号, 重新排序, 组成新的进程组 newgroup。

例 6.3 进程组创建函数应用示例

```
INTEGER GROUP
! 进程组 GROUP, 包含进程{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p}。
INTEGER NEWGROUP ! 新进程组。
INTEGER RANK, RANGE (3, 4)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
RANGE (1, 1) = 1
RANGE (2, 1) = 5
RANGE (3, 1) = 2
! 进程组子集 1, 包含进程{b, d, f}。
RANGE (1, 2) = 6
RANGE (2, 2) = 14
RANGE (3, 2) = 3
! 进程组子集 2, 包含进程{g, j, m}。
RANGE (1, 3) = 0
RANGE (2, 3) = 8
RANGE (3, 3) = 4
```

```

    ! 进程组子集 3, 包含进程{a, e, i}。
RANGE(1,4)=10
RANGE(2,4)=15
RANGE(3,4)=5
    ! 进程组子集 4, 包含进程{k, p}。
IF(RANK.EQ.0)THEN
    CALL MPI_GROUP_RANGE_INCL(GROUP,4,RANGE,NEWGROUP,IERR)
    ! 创建新进程组 NEWGROUP, 包含 4 个子集的所有进程{b, d, f, g, j, m, a, e, i, k, p}。
ELSE IF(RANK.EQ.1)THEN
    CALL MPI_GROUP_RANGE_EXCL(GROUP,4,RANGE,NEWGROUP,IERR)
    ! 创建新进程组 NEWGROUP, 包含不属于这 4 个子集的
    ! 所有进程{c, h, l, n, o}组成。
ENDIF

```

6.1.2 进程组访问与比较

/* 进程组大小查询函数, 查询进程组成员的个数。

```

MPI_GROUP_SIZE(group, size)
    IN      group      进程组
    OUT     size       进程组成员个数
C      int MPI_Group_size(MPI_Group group, int * size)
Fortran MPI_GROUP_SIZE(GROUP,SIZE,IERROR)
        INTEGER GROUP,SIZE,IERROR

```

函数 MPI_GROUP_SIZE 返回进程组 group 包含的进程组成员的个数, 称之为**进程组大小**。如果 group = MPI_GROUP_EMPTY, 则 size = 0; 若 group = MPI_GROUP_NULL, 则函数调用出错。

/* 进程序号查询函数, 查询进程组某个成员的序号。

```

MPI_GROUP_RANK(group, rank)
    IN      group      进程组
    OUT     rank       进程组成员的序号
C      int MPI_Group_rank(MPI_Group group, int * rank)
Fortran MPI_GROUP_RANK(GROUP,RANK,IERROR)
        INTEGER GROUP,RANK,IERROR

```

函数 MPI_GROUP_RANK 返回进程在进程组 group 中的序号 rank; 如果进程不属于 group, 则返回 MPI 常数 rank = MPI_UNDEFINED。

/* 进程序号对应关系查询函数, 查询一个进程组内某些进程
/* 在另一个进程组内的相应序号。

MPI_GROUP_TRANSLATE_RANKS(group1,n,ranks1,group2,ranks2)

IN	group1	进程组 1
IN	n	数组 ranks1 与数组 ranks2 的长度
IN	ranks1	数组, 包含进程组 group1 内待查询的进程的序号
IN	group2	进程组 2
OUT	ranks2	数组, 包含进程组 group1 内待查询的进程在进程组 group2 中的序号

C int MPI_Group_translate_ranks (MPI_Group group1, int n, int * ranks1,
 MPI_Group group2, int * ranks2)

Fortran MPI_GROUP_TRANSLATE_RANKS (GROUP1, N, RANKS1, GROUP2,
 RANKS2, IERROR)
 INTEGER GROUP1, N, RANKS1 (*), GROUP2, RAKNS2 (*), IERROR

数组 ranks1 列出进程组 group1 中的 n 个进程, 函数 MPI_GROUP_TRANSLATE_RANKS 为这些进程获取在进程组 group2 中的序号, 并存储在数组 ranks2 的对应位置。若数组 ranks1 中列出的进程不属于进程组 group2, 则返回常数 MPI_UNDEFINED。因此, 函数 MPI_GROUP_TRANSLATE_RANKS 通常可用于决定同一进程在不同两个进程组中序号的对应关系。

例 6.4 函数 MPI_GROUP_TRANSLATE_RANKS 应用示例

```

INTEGER GROUP1     ! 进程组 1, 包含进程{a, b, c, d, e, f}。
INTEGER GROUP2     ! 进程组 2, 包含进程{b, c, e, g, h}。
INTEGER RANKS1(4), RANKS2(4), RANK
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
RANKS1(1) = 0
RANKS1(2) = 1
RANKS1(3) = 2
RANKS1(4) = 4
IF(RANK.EQ.0)THEN
  CALL MPI_GROUP_TRANSLATE_RANKS(GROUP1, 4, RANKS1,
                                 GROUP2, RANKS2, IERR)
  ! 数组 RANKS2 = {MPI_UNDEFINED, 0, 2, 1}。
ELSE IF(RANK.EQ.1)THEN
  CALL MPI_GROUP_TRANSLATE_RANKS(GROUP2, 4, RANKS1,
                                 GROUP1, RANKS2, IERR)
  ! 数组 RANKS2 = {1, 4, 2, MPI_UNDEFINED}。
ENDIF
/* 进程组比较函数, 返回两个进程组的比较结果。

```

	MPI_GROUP_COMPARE (group1, group2, result)	
IN	group1	进程组 1
IN	group2	进程组 2
Out	result	比较结果

```

C      int MPI_Group_compare (MPI_Group group1, MPI_Group group2, int * result)
Fortran MPI_GROUP_COMPARE (GROUP1, GROUP2, RESULT, IERROR)
      INTEGER GROUP1, GROUP2, RESULT, IERROR

```

函数 MPI_GROUP_COMPARE 比较两个进程组, 返回 3 种可能的结果: (1) result = MPI_IDENT, 表示两个进程组包含的进程及它们的序号相同; (2) result = MPI_SIMILAR, 表示两个进程组包含的进程相同, 但进程的序号不同; (3) result = MPI_UNEQUAL, 表示两个进程组分别包含不同的进程。

例 6.5 函数 MPI_GROUP_COMPARE 应用示例

```

INTEGER GROUP1 ! 进程组 1, 包含进程{a, b, c, d, e, f}
INTEGER GROUP2 ! 进程组 2, 包含进程{a, b, c, d, f, e}
INTEGER GROUP3 ! 进程组 3, 包含进程{a, b, c, d, f, e}
INTEGER GROUP4 ! 进程组 4, 包含进程{a, b, c, d, f, g}
INTEGER RESULT
CALL MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERR)
      ! RESULT = MPI_SIMILAR
CALL MPI_GROUP_COMPARE(GROUP2, GROUP3, RESULT, IERR)
      ! RESULT = MPI_IDENT
CALL MPI_GROUP_COMPARE(GROUP3, GROUP4, RESULT, IERR)
      ! RESULT = MPI_UNEQUAL

```

6.1.3 进程组释放

```

/* 进程组释放函数, 释放一个已创建的进程组。
   MPI_GROUP_FREE (group)
   INOUT group      进程组
C      int MPI_Group_free (MPI_Group * group)
Fortran MPI_GROUP_FREE (GROUP, IERROR)
      INTEGER GROUP, IERROR

```

当定义的进程组 group 不再需要时, MPI 程序可以调用函数 MPI_GROUP_FREE 释放该进程组, 同时将变量 group 的值自动改变为 MPI 系统常数 MPI_GROUP_NULL。特别是, MPI 系统提供的两个进程组常数 MPI_GROUP_NULL 和 MPI_GROUP_EMPTY 不能作为该函数的参数; 同时, MPI 系统定义的附属于通信器 MPI_COMM_WORLD 和 MPI_COMM_SELF 的进程组, 也不能作为该函数的参数, 敬请用户注意。

6.2 通信器管理

6.2.1 通信器创建

在第三章, 我们曾简单地提到过, MPI 系统规定: 任意进程之间的点对点通信和聚合

通信，必须基于某个通信器发生。实际上，应用程序创建一个进程组后，就隐含地确定了该进程组各成员之间的一个通信域；但是，只有在此基础上创建一个通信器后，各进程之间才能进行 MPI 通信。而且，MPI 系统还规定：任何新通信器的创建必须基于某个或多个已存在的通信器才能进行，而且需要这些通信器包含的进程组所有成员的共同参与。因此，通信器创建属于聚合通信的特例。

```
/* 通信器复制函数，复制一个进程通信器。  
  
MPI_COMM_DUP (comm, newcomm)  
    IN      comm      原通信器  
    OUT     newcomm    复制的新通信器  
C   int MPI_Comm_dup (MPI_Comm comm, MPI_Comm * newcomm)  
Fortran MPI_COMM_DUP (COMM, NEWCOMM, IERROR)  
        INTEGER COMM, NEWCOMM, IERROR
```

基于通信器 comm，函数 MPI_COMM_DUP 创建一个具有同样固定属性（进程组和拓扑结构）的通信器 newcomm，它需要通信器 comm 包含的所有进程的共同参与。

例 6.6 函数 MPI_COMM_DUP 应用示例

```
INTEGER NEWCOMM ! 新通信器。  
INTEGER RANK0, RANK1, SIZE, NPROC  
INTEGER STATUS (MPI_STATUS_SIZE)  
REAL A  
CALL MPI_INIT(IERR)  
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROC, IERR)  
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK0, IERR)  
CALL MPI_COMM_DUP(MPI_COMM_WORLD, NEWCOMM, IERR)  
    ! 复制创建新通信器 NEWCOMM。  
CALL MPI_COMM_SIZE(NEWCOMM, SIZE, IERR)  
CALL MPI_COMM_RANK(NEWCOMM, RANK1, IERR)  
IF(SIZE.NE.NPROC.OR.RANK1.NE.RANK0)THEN  
    PRINT *, "MPI_COMM_DUP FAILED"  
    STOP  
ELSE  
    ! 基于新通信器的点对点通信。  
    IF(RANK1.EQ.0)THEN  
        CALL MPI_SEND(A, 1, MPI_REAL, 1, 555, NEWCOMM, IERR)  
    ELSE IF(RANK1.EQ.1)THEN  
        CALL MPI_RECV(A, 1, MPI_REAL, 0, 555, NEWCOMM, STATUS, IERR)  
    ENDIF  
ENDIF
```

在上例中,如果我们将点对点通信改变为:

```
IF(RANK1.EQ.0)THEN
    CALL MPI_SEND(A,1,MPI_REAL,1,555,MPI_COMM_WORLD,IERR)
ELSE IF(RANK1.EQ.1)THEN
    CALL MPI_RECV(A,1,MPI_REAL,0,555,NECOMM,IERR)
ENDIF
```

则进程 1 将被死锁,因为进程 0 发出的消息包含在通信器 MPI_COMM_WORLD 中,而进程 1 接收的消息来自于通信器 NECOMM,尽管两个通信器包含相同的进程组,但是它们仍然是两个不同的通信器,而任何 MPI 通信操作只能发生在同一个通信器内部。

进一步,如果我们将调用通信器复制函数的语句改写为:

```
IF(RANK0.EQ.0)THEN
    CALL MPI_COMM_DUP(MPI_COMM_WORLD,NECOMM,IERR)
ENDIF
```

则也将导致进程的死锁,因为函数 MPI_COMM_DUP 需要通信器 MPI_COMM_WORLD 包含的所有进程的共同参与,属于全局函数。

/* 通信器创建函数,基于某个通信器包含的进程组子集,创建一个新通信器。

```
MPI_COMM_CREATE(comm,group,newcomm)
    IN      comm      原通信器
    IN      group     组成新通信器的进程组子集
    OUT     newcomm   新通信器
C     int MPI_Comm_create(MPI_Comm comm,MPI_Group group,
                        MPI_Comm *newcomm)
Fortran MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
        INTEGER COMM, GROUP, NEWCOMM, IERROR
```

函数 MPI_COMM_CREATE 创建一个新的通信器 newcomm, 它由原通信器 comm 包含的进程组子集 group 中所有进程组成;但是,新通信器 newcomm 不继承原通信器 comm 的属性(例如拓扑结构)。实际上,新通信器 newcomm 将定义进程组 group 内的一个新通信域。函数 MPI_COMM_CREATE 需要通信器 comm 包含的所有进程的共同参与,但是,对那些不属于进程组子集 group 的进程,该函数调用将返回常数 newcomm = MPI_COMM_NULL。

显然,函数 MPI_COMM_CREATE 要求各进程提供的进程组子集 group 必须相同,且为通信器 comm 的子集;否则,函数调用出错。

例 6.7 函数 MPI_COMM_CREATE 应用示例

```
INTEGER GROUP
INTEGER NEWGROUP      ! 进程组子集,由所有奇数序号的进程组成。
INTEGER RANGE (3,1)
INTEGER NECOMM        ! 新通信器。
INTEGER A,NPROC,SIZE,RANK
```

```

CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NPROC, IERR)
RANGE(1,1) = 1
RANGE(2,1) = NPROC
RANGE(3,1) = 2
CALL MPI_COMM_GROUP(MPI_COMM_WORLD, GROUP, IERR)
CALL MPI_GROUP_RANGE_INCL(GROUP, 1, RANGE, NEWGROUP, IERR)
! 创建由所有奇数序号进程组成的进程组。
CALL MPI_COMM_CREATE(MPI_COMM_WORLD, NEWGROUP,
                      NEWCOMM, IERR)
! 创建包含所有奇数序号进程的新的通信器。
RANK = -1
CALL MPI_COMM_RANK(NEWCOMM, RANK, IERR)
!
! 新通信器的进程 0 与进程 1 之间的点对点通信, 等价于原通信器。
! MPI_COMM_WORLD 的进程 1 与进程 3 之间的点对点通信。
IF(RANK.EQ.0)THEN
    CALL MPI_SEND(A, 1, MPI_REAL, 1, 555, NEWCOMM, IERR)
ELSE IF(RANK.EQ.1)THEN
    CALL MPI_RECV(A, 1, MPI_REAL, 0, 555, NEWCOMM, IERR)
ENDIF

```

/* 通信器创建函数, 分裂某个通信器成多个进程组,
/* 并同时为每个进程组创建一个新通信器。

```

MPI_COMM_SPLIT (comm, color, key, newcomm)
    IN      comm      原通信器
    IN      color     各个进程提供的用于分组的整型参数
    IN      key       各个进程提供的用于同组内进程排序的整型参数
    OUT     newcomm   新通信器
C      int MPI_Comm_split (MPI_Comm comm, int color, int key,
                           MPI_Comm * newcomm)
Fortran MPI_COMM_SPLIT (COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

```

函数 MPI_COMM_SPLIT 规定: 首先, 根据各个进程提供的整型参数 color, 将原通信器 comm 包含的所有进程分割成多个子进程组, 其中, 属于同一个子进程组的进程提交的参数 color 必须相等; 然后, 在子进程组内, 按由小到大的整型参数 key 的值, 依次排列它所包含的各个进程, 如果两个进程提交的参数 key 相等, 则按它们在原通信器 comm 中由小到大的序号排列; 最后, 为每个子进程组创建新通信器 newcomm。

每个进程均可提供相等或不等的整型参数 color 和 key, 但 color 必须不小于 0。一个进程, 若提供参数 color = MPI_UNDEFINED, 则说明该进程不属于任何子进程组, 且返回参数 newcomm = MPI_COMM_NULL。

表 6.1 提供不同 color 和 key 值的 10 个进程

序号	0	1	2	3	4	5	6	7	8	9
进程	a	b	c	d	e	f	g	h	i	j
color	0	1	3	0	3	0	0	5	3	1
key	3	1	2	5	1	1	1	2	1	0

* 1 表示该进程提供 color = MPI_UNDEFINED。

例 6.8 函数 MPI_COMM_SPLIT 应用示例

```

INTEGER    COMM      ! 初始通信器, 包含 10 个进程, 在表 6.1 中列出。
INTEGER    COLOR     ! 各进程提供的分组参数, 在表 6.1 中列出。
INTEGER    KEY       ! 各进程提供的同组内排序参数, 在表 6.1 中列出。
INTEGER    NEWCOMM   ! 新通信器。
CALL MPI_COMM_RANK(COMM, RANK, IERR)
IF(RANK.EQ.0) THEN
    COLOR=0
    KEY = 3
ELSE IF(RANK.EQ.1) THEN
    COLOR=MPI_UNDEFINED
    KEY = 1
ELSE IF(RANK.EQ.2)THEN
    COLOR=3
    KEY = 2
ELSE IF(RANK.EQ.3)THEN
    COLOR=0
    KEY = 5
ELSE IF(RANK.EQ.4)THEN
    COLOR=3
    KEY = 1
ELSE IF(RANK.EQ.5)THEN
    COLOR=0
    KEY = 1
ELSE IF(RANK.EQ.6)THEN
    COLOR=0
    KEY = 1
ELSE IF(RANK.EQ.7)THEN
    COLOR=5
    KEY = 2
ELSE IF(RANK.EQ.8)THEN
    COLOR=3
    KEY = 1
ELSE IF(RANK.EQ.9)THEN
    COLOR=MPI_UNDEFINED

```

```

    KEY = 0
ENDIF
CALL MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERR)
! 创建 3 个新通信器, 分别包含进程组 {f, g, a, d}, {e, i, c} 和 {h}。

```

显然, 函数调用:

```
CALL MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERR)
```

等价于函数调用:

```
CALL MPI_COMM_SPLIT(COMM, color, 1, NEWCOMM, IERR)
```

其中, 属于 GROUP 的进程提供参数 color = 0, 而所有其他进程提供参数 color = MPI_UNDEFINED。

6.2.2 通信器访问与比较

通信器创建后, 可通过下列函数访问其属性。

/* 通信器大小查询函数, 获取某个通信器包含的进程个数。

```

MPI_COMM_SIZE (comm, size)
IN      comm      通信器
OUT     size      通信器包含的进程个数
C      int MPI_Comm_size (MPI_Comm comm, int * size)
Fortran MPI_COMM_SIZE (COMM, SIZE, IERROR)
        INTEGER COMM, SIZE, IERROR

```

函数 MPI_COMM_SIZE 返回通信器 comm 包含的进程个数 size, 称之为通信器大小。

/* 进程序号查询函数, 获取某个进程在通信器中的序号。

```

MPI_COMM_RANK (comm, rank)
IN      comm      通信器
OUT     rank      进程的序号
C      int MPI_Comm_rank (MPI_Comm comm, int * rank)
Fortran MPI_COMM_RANK (COMM, RANK, IERROR)
        INTEGER COMM, RANK, IERROR

```

函数 MPI_COMM_RANK 返回进程在通信器 comm 中的序号 rank。

/* 通信器比较函数, 比较两个通信器的异同。

```

MPI_COMM_COMPARE (comm1, comm2, result)
IN      comm1      通信器 1
IN      comm2      通信器 2
OUT     result     比较结果

```

```

C      int MPI_Comm_compare (MPI_Comm comm1, MPI_Comm comm2, int * result)
Fortran  MPI_COMM_COMPARE (COMM1, COMM2, RESULT, IERROR)
          INTEGER COMM1, COMM2, RESULT, IERROR

```

函数 MPI_COMM_COMPARE 比较通信器 comm1 和通信器 comm2, 返回 4 种可能的结果:(1) result = MPI_IDENT:comm1 与 comm2 代表同一个通信域; (2) result = MPI_CONGRUENT:comm1 与 comm2 拥有相同的进程组, 但代表不同的通信域; (3) result = MPI_SIMILAR:comm1 与 comm2 拥有的进程组成员相同, 但进程序号不同; (4) result = MPI_UNEQUAL:comm1 与 comm2 拥有的进程组成员不同。

例 6.9 函数 MPI_COMM_COMPARE 应用示例

```

INTEGER COMM1, COMM2, COMM3, COMM4, COMM5      ! 通信器。
INTEGER RANGE(3,2), GROUP1, GROUP2, GROUP3, GROUP4, SIZE
INTEGER RESULT
!
CALL MPI_COMM_DUP(MPI_COMM_WORLD, COMM1, IERR)
COMM2 = COMM1
CALL MPI_COMM_DUP(COMM1, COMM3, IERR)           ! 复制通信器。
CALL MPI_COMM_GROUP(COMM1, GROUP1, IERR)         ! 获得进程组。
CALL MPI_COMM_SIZE(COMM1, SIZE, IERR)
RANGE(1,1) = 0
RANGE(2,1) = SIZE
RANGE(3,1) = 2
RANGE(1,2) = 1
RANGE(2,2) = SIZE
RANGE(3,2) = 2
CALL MPI_GROUP_RANGE_INCL(GROUP1, 2, RANGE, GROUP2, IERR)
! 进程组 GROUP2 由进程组 GROUP1 中所有序号为偶数的进程, 再加上
! 所有序号为奇数的进程组成。
CALL MPI_COMM_CREATE(COMM2, GROUP2, COMM4, IERR)
CALL MPI_GROUP_RANGE_INCL(GROUP1, 1, RANGE, GROUP3, IERR)
! 进程组 GROUP3 由进程组 GROUP1 中所有序号为偶数的进程构成。
CALL MPI_COMM_CREATE(COMM2, GROUP3, COMM5, IERR)
CALL MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERR)
! 返回 RESULT = MPI_IDENT。
CALL MPI_COMM_COMPARE(COMM1, COMM3, RESULT, IERR)
! 返回 RESULT = MPI_CONGRUENT。
CALL MPI_COMM_COMPARE(COMM1, COMM4, RESULT, IERR)
! 返回 RESULT = MPI_SIMILAR。
CALL MPI_COMM_COMPARE(COMM1, COMM5, RESULT, IERR)
! 返回 RESULT = MPI_UNEQUAL。
CALL MPI_COMM_FREE(COMM5, IERR)! 释放通信器 COMM5, 见下面介绍。

```

```
CALL MPI_COMM_FREE(COMM4, IERR)! 释放通信器 COMM4。  
CALL MPI_COMM_FREE(COMM3, IERR)! 释放通信器 COMM3。  
CALL MPI_COMM_FREE(COMM2, IERR)! 释放通信器 COMM2。  
CALL MPI_COMM_FREE(COMM, IERR)! 释放通信器 COMM。
```

6.2.3 通信器释放

/* 通信器释放函数, 释放一个已创建的进程通信器。

```
MPI_COMM_FREE(comm)  
INOUT comm 通信器  
C int MPI_Comm_free(MPI_Comm *comm)  
Fortran MPI_COMM_FREE(COMM, IERROR)  
INTEGER COMM, IERROR
```

函数 MPI_COMM_FREE 释放已创建的通信器 comm, 并将 comm 自动改写为 MPI 常数 MPI_COMM_NULL。该函数要求属于该通信器的所有进程的共同参与。函数执行过程中, 如果通信器内部还存在尚未完成的通信操作, 则阻塞等待这些通信操作完成后, 才释放该通信器。若参数 comm = MPI_COMM_NULL 或 MPI_COMM_WORLD 或 MPI_COMM_SELF, 则函数调用出错, 敬请读者注意。

6.2.4 通信器附加属性

前面, 我们详细介绍了通信器的进程组属性, 并在第二章简单举例说明了通信器的拓扑结构属性(详细的介绍请参考第七章)。其实, MPI 系统还提供了为通信器增加其他属性的功能, 以满足不同的应用需求, 并称它们为**附加属性**。例如, 传递函数库的初始化参数、MPI 异常处理、MPI 系统常数等等, 这些均属于通信器的附加属性。类似, 附加属性可以被创建、访问和释放。

MPI 系统中, 通信器的每个属性均与一个关键值相关联, 函数 MPI_KEYVAL_CREATE 和 MPI_KEYVAL_FREE 可用于为通信器分配和释放关键值。关键值一旦分配, 则可通过函数 MPI_ATTR_PUT 来设置与该关键值相关联的属性, 并通过函数 MPI_ATTR_GET 来访问。函数 MPI_ATTR_DELETE 可用于释放属性。

一般情况下, 应用程序均不会使用到这些函数。因此, 本书不准备对以上函数进行详细的讨论, 有兴趣者可参考其他 MPI 文献。

6.3 域间通信器

在实际应用程序中, 根据进程从事任务的不同, 通常可将它们组合成多个不同的进程组, 并为每个进程组建立域内通信器, 以方便各进程组成员间的通信, 提高通信性能; 但是, 某些实际应用题也需要在不同进程组, 或者不同通信域的两个进程之间进行数据交换。目前, 我们有两种办法解决这个问题: 第一, 将两个进程组合为一个进程组, 并建立通信器, 于是这些数据交换就变为该通信器内部的通信, 即**域内通信**, 例如本书前面提到的

所有 MPI 进程通信均属于该类通信；第二，直接组织属于不同通信域的两个进程之间的通信，即域间通信。

给定属于不同域内通信器，但属于同一域间通信器的两个进程，MPI 系统只支持它们之间的点对点通信，而不支持第五章讨论的聚合通信。因此，MPI 系统域间通信模式可简单描述如下：

假设进程组 A 和进程组 B 分别属于不同的通信器，进程组 A 的某个进程执行一个消息发送函数：

```
CALL MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)
```

而进程组 B 的某个进程执行一个相匹配的消息接收函数：

```
CALL MPI_RECV(buf, count, datatype, source, tag, comm, ierr)
```

其中，通信器 comm 比较特殊，它即不是包含进程组 A 的通信器，也不是包含进程组 B 的通信器，而是联接进程组 A 和进程组 B 的域间通信器（详见 6.3.2 节），且基于该通信器，进程组 A 与进程组 B 的任意两个进程之间可以直接组织点对点通信。此外，参数 dest 和参数 source 分别为消息接收进程和消息发送进程在各自进程组中的序号。相对于进程组 A，进程组 B 称为远程进程组，反之亦然。

与域内点对点通信一样，域间点对点通信也可以采用第三章中介绍的四种模式的阻塞或非阻塞通信函数。因此，只要了解域间通信器的创建后，我们便可以自由地组织属于不同进程组的任意两个进程之间的域间通信。

6.3.1 域间通信器的创建与释放

域间通信器可通过函数 MPI_COMM_FREE（见 6.2.3 节）来释放，但必须通过以下函数来创建。

```
/* 域间通信器创建函数，创建一个联接两个不同域内通信器的域间通信器。
```

```
MPI_INTERCOMM_CREATE (local_comm, local_leader, bridge_comm,
                       remote_leader, tag, newintercomm)

IN      local_comm 本地域内通信器
IN      local_leader 本地桥进程在本地域内通信器 local_comm 中的序号
IN      bridge_comm 包含本地桥进程和远程桥进程的域内通信器
IN      remote_leader 远程桥进程在通信器 bridge_comm 中的序号
IN      tag        安全标号
OUT     newintercomm 域间通信器

C      int MPI_Intercomm_create (MPI_Comm local_comm, int local_leader,
                               MPI_Comm bridge_comm, int remote_leader,
                               int tag, MPI_Comm * newintercomm)

Fortran MPI_INTERCOMM_CREATE (LOCAL_COMM, LOCAL_LEADER,
                               BRIDGE_COMM, REMOTE_LEADER,
                               TAG, NEWINTERCOMM, IERROR)
INTEGER LOCAL_COMM, LOCAL_LEADER, BRIDGE_COMM,
         REMOTE_LEADER, TAG, NEWINTERCOMM, IERROR
```

函数 MPI _ INTERCOMM _ CREATE 创建联接包含不同进程的两个域内通信器的域间通信器 newintercomm, 需要两个域内通信器所有进程的共同参与, 且属于同一域内通信器的进程必须提供相同的域内通信器 local _ comm (即该通信器自身)。参数 local _ leader 和 remote _ leader 是两个域内通信器分别提供的进程, 称之为桥进程, 它们必须属于同一个域内通信器 bridge _ comm, 其中参数 local _ leader 表示该进程在本地域内通信器 local _ comm 中的序号, 称之为本地桥进程, 而参数 remote _ leader 表示该进程在通信器 bridge _ comm 的序号, 称之为远端桥进程。所有进程必须提供一致的参数 tag, 且参数 remote _ leader, local _ leader 和 tag 必须给出具体值, 不允许通配。

例 6.10 创建一个联接两个域内通信器的域间通信器

假设通信器 MPI _ COMM _ WORLD 包含 7 个进程。其中, 进程 0, 1 和 2 属于域内通信器 COMM1, 且对应 COMM1 中的序号依次为 0, 1 和 2; 进程 3, 4, 5 和 6 属于另一个域内通信器 COMM2, 且对应 COMM2 中的序号依次为 0, 1, 2 和 3。以 COMM1 和 COMM2 中的 0 号进程为桥进程, MPI _ COMM _ WORLD 为桥通信器, 则可以通过如下函数定义一个 COMM1 和 COMM2 之间的域间通信器。其中, 属于通信器 COMM1 的进程调用函数为:

```
CALL MPI _ INTERCOMM _ CREATE(COMM1, 0, MPI _ COMM _ WORLD, 3, TAG,
                               INTERCOMM, IERR)
```

而属于通信器 COMM2 的进程调用函数为:

```
CALL MPI _ INTERCOMM _ CREATE(COMM2, 0, MPI _ COMM _ WORLD, 0, TAG,
                               INTERCOMM, IERR)
```

之后, 通信器 COMM1 中的进程 2 发送的消息:

```
CALL MPI _ SEND(BUF, COUNT, DATATYPE, 3, 555, INTERCOMM, IERR)
```

通信器 COMM2 中的进程 3 可用如下函数来接收。

```
CALL MPI _ RECV(BUF, COUNT, DATATYPE, 2, 555, INTERCOMM, STATUS, IERR)
```

/* 域间通信器合并函数, 通过合并两个域间通信器, 创建一个域内通信器。

MPI _ INTERCOMM _ MERGE (intercomm, high, newintracomm)

IN intercomm 域间通信器

IN high 见下面的解释

OUT newintracomm 域内通信器

C int MPI _ Intercomm _ merge (MPI _ Comm intercomm, int high, MPI _ Comm * newintracomm)

Fortran MPI _ INTERCOMM _ MERGE (INTERCOMM, HIGH, NEWINTRACOMM, IERROR)

INTEGER INTERCOMM, NEWINTRACOMM, IERROR

LOGICAL HIGH

假设域间通信器 intercomm 联接的两个域内通信器为 A 和 B, 则函数 MPI _ INTERCOMM _ MERGE 要求属于通信器 A 的进程提供参数 high = true, 而属于通信器 B 的进程提供参数 high = false, 并创建一个新的域内通信器 newintracomm, 它包含通信器 A 和通信器 B 的所有进程, 并保持这些进程在原来通信器中的排列次序, 但是, 属于通信

器 A 的进程由于提供参数 high = true, 故先对通信器 A 中的进程进行排序。

6.3.2 域间通信器访问

/* 域间通信器查询函数, 返回一个通信器是否为域间通信器的信息。

```
MPI_COMM_TEST_INTER (comm, flag)
IN      comm      通信器
OUT     flag      若 comm 为域间通信器, 则返回 true
C      int MPI_Comm_test_inter (MPI_Comm comm, int * flag)
Fortran MPI_COMM_TEST_INTER (COMM, FLAG, IERROR)
        INTEGER COMM, IERROR
        LOGICAL FLAG
```

函数 MPI_COMM_TEST_INTER 属于局部函数, 如果通信器 comm 为域间通信器, 则返回 flag = true; 否则, 返回 flag = false。

/* 域间通信器查询函数, 返回域间通信器联接的远程进程组大小。

```
MPI_COMM_REMOTE_SIZE (comm, size)
IN      comm      通信器
OUT     size      域间通信器联接的远程进程组大小
C      int MPI_Comm_remote_size (MPI_Comm comm, int * size)
Fortran MPI_COMM_REMOTE_SIZE (COMM, SIZE, IERROR)
        INTEGER COMM, SIZE, IERROR
```

函数 MPI_COMM_REMOTE_SIZE 返回属于同一域间通信器的远程域内通信器包含的进程个数, 也就是远程进程组的大小。

/* 域间通信器查询函数, 返回域间通信器联接的远程进程组。

```
MPI_COMM_REMOTE_GROUP (comm, group)
IN      comm      通信器
OUT     group     域间通信器联接的远程进程组
C      int MPI_Comm_remote_group (MPI_Comm comm, MPI_Group group)
Fortran MPI_COMM_REMOTE_GROUP (COMM, GROUP, IERROR)
        INTEGER COMM, GROUP, IERROR
```

函数 MPI_COMM_REMOTE_GROUP 返回属于同一域间通信器的远程进程组。

第七章 进程拓扑结构

进程拓扑结构 (process topology) 是进程组各成员之间相互联接的一种通信结构, 是 MPI 系统为了方便进程组成员之间的相互联系, 以及进程组到并行机的具体映射, 并为域内通信器提供的一个固有属性。域间通信器不包含任何拓扑结构, 所以本书有关拓扑结构的讨论均是针对域内通信器而言的, 敬请读者注意。

由前面各章的讨论可知, 假设通信器包含 n 个进程, 则每个进程均分配一个惟一的序号 ($0 \sim n - 1$), 且任意两个进程之间可以自由地进行点对点通信。但在许多实际应用中, 进程间的相互通信关系, 可用一种固定的拓扑结构来表示。该拓扑结构由结点和连线构成, 其中, 结点代表进程, 而连线代表它所联接的两个结点之间存在点对点通信。MPI 系统为任意域内通信器提供这项服务, 并称之为该通信器包含的进程组的**虚拟 (virtual)** 拓扑结构, 简称**拓扑结构**。

在拓扑结构中, 每个进程均有一个坐标, 用于定义该进程在拓扑结构中的位置, 且坐标与进程序号一一对应。同时, 通过这些坐标, 每个进程能方便地调用函数来查询需要与其进行点对点通信的进程。

在进程组中建立拓扑结构, 有利于合理地映射各个进程到具体并行机各处理器中, 并尽量保证在拓扑结构中, 相邻的进程被映射到相邻的处理器中, 以便减少通信延迟, 改进通信性能。

MPI 系统提供一系列函数, 为进程组创建、访问和释放拓扑结构。一个通信器, 只能拥有一种拓扑结构, 但在同一进程组中, 可定义多个通信器, 从而也可定义多个拓扑结构。

任何进程拓扑结构均可以用图 (graph) 来表示, 其中, 图的结点代表进程, 结点间连线代表进程间通信, 我们将在 7.2 节中详细介绍基于图的拓扑结构的创建、访问和释放。

但是, 对许多实际应用问题, 例如二维和三维流体力学数值模拟, 可用基于二维和三维 Cartesian 坐标的规则网格图来充分表示进程间的相互通信关系, 称之为 **Cartesian 拓扑结构**。为此, MPI 系统提供了一系列创建、访问和释放 Cartesian 拓扑结构的函数, 将在 7.1 节中详细讨论。

7.1 Cartesian 拓扑结构

7.1.1 Cartesian 拓扑结构创建

/* Cartesian 拓扑结构创建函数, 创建一个包含 Cartesian 拓扑结构的新通信器。

```
MPI_CART_CREATE (comm_old, ndim, dims, periods, reorder, comm_cart)
    IN      comm_old    原通信器
    IN      ndim        待创建的 Cartesian 拓扑结构的维数
    IN      dims         数组, 含拓扑结构各维包含的进程个数
    IN      periods      数组, 含各维的进程是否周期联接的标志
```

IN	reorder	true 表示重新排列进程的序号
OUT	comm_cart	含 Cartesian 拓扑结构的新通信器
C	MPI_Cart_create (MPI_Comm comm_old, int ndim, int * dims, int * periods, int reorder, MPI_Comm * comm_cart)	
Fortran	MPI_CART_CREATE (COMM_OLD, NDIM, DIMS, PERIODS, REORDER, COMM_CART, IERROR)	
	INTEGER COMM_OLD, NDIM, DIMS(*), COMM_CART, IERROR	
	LOGICAL PERIODS(*), REORDER	

基于通信器 comm_old, 函数 MPI_CART_CREATE 返回包含 Cartesian 拓扑结构的新通信器 comm_cart, 它要求通信器 comm_old 所有进程的共同参与, 且各个进程提供完全一致的参数 (ndim, dims, periods, reorder)。

参数 NDIM 表示待创建的拓扑结构的维数; 参数 dims 是长度为 ndim 的数组, 其元素分别表示拓扑结构各维包含的进程个数; 参数 periods 是长度为 ndim 的逻辑型变量数组, 其元素若为真, 则表示拓扑结构沿着该维的所有结点首尾相连; 参数 reorder 为逻辑型变量, 若 reorder = false (或 0), 则表示新通信器 comm_cart 的进程将继承它们在原通信器 comm_old 中的排列次序, 若 reorder = true (或非 0), 则这些进程将被在新通信器中重新排序。

若 ndim = 1, 且 periods(1) = false, 则称拓扑结构为一维阵列; 若 periods(1) = true, 称之为环 (ring); 若 ndim = 2 或 3, 且 periods 的元素均为 false, 则称拓扑结构为二维或三维阵列, 若 periods 各元素均为 true, 则称之为二维或三维环阵列 (Torus); 特别地, 对 n 维 Torus, 如果每个方向只包含 2 个进程, 则称之为 n 维超立方体 (Hypercube)。

新通信器 comm_cart 包含的进程个数等于参数 dims 各元素的乘积, 若该乘积小于通信器 comm_old 包含的进程个数, 则存在一些进程, 其函数调用将返回参数 comm_cart = MPI_COMM_NULL; 若该乘积大于通信器 comm_old 包含的进程个数, 则各进程的函数调用将出错。

例 7.1 函数 MPI_CART_CREATE 应用示例

如图 7.1 所示, 假设某个通信器包含 12 个进程, 进程序号依次为 0, 1, …, 11, 要求基于该通信器, 建立一个包含二维 Cartesian 3×4 网格拓扑结构的新通信器。具体程序如下:

0/(0,0)	1/(0,1)	2/(0,2)	3/(0,3)
4/(1,0)	5/(1,1)	6/(1,2)	7/(1,3)
8/(2,0)	9/(2,1)	10/(2,2)	11/(2,3)

图 7.1 二维 Cartesian 3×4 网格拓扑结构, 其中,
斜线“/”下方括号内给出的是各进程的坐标

```

INTEGER COMM0      ! 原通信器, 包含 12 个进程, 不包含拓扑结构。
INTEGER COMM1      ! 新通信器, 包含二维 Cartesian  $3 \times 4$  网格拓扑结构。
INTEGER NDIM, DIMS(2), SIZE
LOGICAL PERIODS(2), REORDER
!
NDIM = 2
DIMS(1) = 3
DIMS(2) = 4
CALL MPI_COMM_SIZE(COMM0, SIZE, IERR)
IF (DIMS(1) * DIMS(2).GE.SIZE) THEN
    PRINT *, "+++ ERROR FOR PARAMETER DIMS +++"
    STOP
ENDIF
PERIODS(1) = .FALSE.          ! 沿第 1 个方向, 进程不周期联接。
PERIODS(2) = .FALSE.          ! 沿第 2 个方向, 进程不周期联接。
REORDER = .TRUE.              ! 进程被重新排序。
CALL MPI_CART_CREATE(COMM0, NDIM, DIMS, PERIODS, REORDER,
                     COMM1, IERR)
! 创建包含二维 Cartesian  $3 \times 4$  网格拓扑结构的通信器 COMM1。

```

在图 7.1 中, 每个进程均对应一个坐标, 在进程序号的下方标出。这里, 我们有必要提醒读者, 无论对 C 语言, 还是对 Fortran 语言, MPI 系统均规定 Cartesian 拓扑结构以“行序 (row-major)”排列进程, 且坐标序号从 0 开始。因此, 在应用 Cartesian 拓扑结构时, 请用户根据具体应用问题的需要, 调整好坐标与序号的对应关系。

7.1.2 Cartesian 拓扑结构辅助函数

```

/* Cartesian 拓扑结构辅助函数。给定结点总数和维数, 该函数返回
/* Cartesian 拓扑结构各维包含的最优结点个数。

MPI_DIMS_CREATE (nnode, ndim, dims)
    IN      nnode        结点总数
    IN      ndim         Cartesian 拓扑结构的维数
    INOUT     dims        长度为 NDIM 的数组, 其元素代表各维包含的结点个数
C      int MPI_Dims_create (int nnode, int ndim, int * dims)
Fortran MPI_DIMS_CREATE (NNODE, NDIM, DIMS, IERROR)
    INTEGER NNODE, NDIM, DIMS(*), IERROR

```

给定结点总数 nnode 和维数 ndim, 函数 MPI_DIMS_CREATE 按等分的原则, 确定各维的结点个数, 并存储在数组 dims 的对应元素中。初始输入时, 如果 $\text{dims}(i) = 0$, 则表示沿第 $i - 1$ 维由 MPI 系统确定结点个数; 若 $\text{dims}(i) \neq 0$, 则表示第 $i - 1$ 维分配的结点个数为 $\text{dims}(i)$ 。因此, 如果 $\text{dims}(i) < 0$, 或者 nnode 不被 $\prod_{i=1, \text{ndim}}^{i-1} \text{dims}(i)$ 整除, 则函数调用

出错。函数返回时, $nnode = \prod_{i=1, ndim} \text{dims}(i)$ 。

函数 MPI_DIMS_CREATE 是局部函数, 表 7.1 给出了该函数调用的几种情形。

表 7.1 局部函数调用的几种情形

input dims	函数调用	返回 dims
(0, 0)	(6, 2, dims)	(3, 2)
(0, 0)	(7, 2, dims)	(7, 1)
(0, 3, 0)	(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	(7, 3, dims)	出错

例 7.2 函数 MPI_DIMS_CREATE 应用示例 (参考例 7.1)

```
INTEGER COMM0 ! 原通信器, 包含 12 个进程。
INTEGER COMM1 ! 新通信器, 包含二维 Cartesian 拓扑结构。
INTEGER NDIM, DIMS(2), SIZE
LOGICAL PERIODS(2), REORDER
!
NDIM=2
DIMS(1)=0
DIMS(2)=0
CALL MPI_DIMS_CREATE(12, NDIM, DIMS)
    ! 获得二维 Cartesian 拓扑结构的各维结点个数。
CALL MPI_COMM_SIZE(COMM0, SIZE, IERR)
IF (DIMS(1) * DIMS(2).GE.SIZE) THEN
    PRINT *, "+++ ERROR FOR PARAMETER DIMS +++"
    STOP
ENDIF
PERIODS(1)= .FALSE.
PERIODS(2)= .FALSE.
REORDER = .TRUE.
CALL MPI_CART_CREATE(COMM0, NDIM, DIMS, PERIODS, REORDER,
                     COMM1, IERR)
    ! 创建包含二维 Cartesian 拓扑结构的通信器 COMM1。
```

7.1.3 Cartesian 拓扑结构查询

/* Cartesian 拓扑结构查询函数, 返回 Cartesian 拓扑结构的维数。

```
MPI_CARTDIM_GET (comm, ndim)
    IN      comm      含 Cartesian 拓扑结构的通信器
    OUT     ndim      Cartesian 拓扑结构的维数
C     int MPI_Cartdim_get (MPI_Comm comm, int * ndim)
```

```
Fortran MPI_CARTDIM_GET (COMM, NDIM, IERROR)
      INTEGER COMM, NDIM, IERROR
```

函数 MPI_CARTDIM_GET 返回通信器 comm 包含的 Cartesian 拓扑结构的维数 ndim。

/* Cartesian 拓扑结构查询函数, 返回 Cartesian 拓扑结构的基本信息。

```
MPI_CART_GET (comm, maxdim, dims, periods, coords)
  IN      comm      含 Cartesian 拓扑结构的通信器
  IN      maxdim    Cartesian 拓扑结构的维数
  OUT     dims      长度为 maxdim 的数组, 含各维的结点个数
  OUT     periods   长度为 maxdim 的数组, 含各维是否周期联接的信息
  OUT     coords    长度为 maxdim 的数组, 含调用该函数进程的坐标
C      int MPI_Cart_get (MPI_Comm comm, int maxdim, int * dims, int * periods, int * coords)
Fortran MPI_CART_GET (COMM, MAXDIM, DIMS, PERIODS, COORDS, IERROR)
      INTEGER COMM, MAXDIM, DIMS (*), COORDS (*), IERROR
      LOGICAL PERIODS (*)
```

函数 MPI_CART_GET 返回通信器 comm 包含的 Cartesian 拓扑结构的如下信息:
(1) 第 $i - 1$ 维 ($i = 1, 2, \dots, \text{maxdim}$) 包含的结点个数 $\text{dims}(i)$; (2) 第 $i - 1$ 维的结点是否周期联接的信息 $\text{periods}(i)$, 如果 $\text{periods}(i) = \text{true}$, 则表示沿第 $i - 1$ 维, 结点是周期联接的; (3) 调用函数进程的第 $i - 1$ 维坐标 $\text{coords}(i)$, $\text{coords}(i)$ 必须取值于 $0, 1, \dots, \text{dims}(i) - 1$ 之间。

/* Cartesian 拓扑结构查询函数, 返回某个坐标对应的进程的序号。

```
MPI_CART_RANK (comm, coords, rank)
  IN      comm      含 Cartesian 拓扑结构的通信器
  IN      coords    数组, 含待查询进程的 Cartesian 坐标
  OUT     rank      待查询进程的序号
C      int MPI_Cart_rank (MPI_Comm comm, int * coords, int * rank)
Fortran MPI_CART_RANK (COMM, COORDS, RANK, IERROR)
      INTEGER COMM, COORDS (*), RANK, IERROR
```

基于包含 Cartesian 拓扑结构的通信器 comm, 函数 MPI_CART_RANK 返回坐标为 coords 的进程的序号, 并存储在参数 rank 中。

/* Cartesian 拓扑结构查询函数, 返回某个进程的坐标。

```
MPI_CART_COORDS (comm, rank, maxdim, coords)
  IN      comm      含 Cartesian 拓扑结构的通信器
  IN      rank      待查询进程的序号
  IN      maxdim   Cartesian 拓扑结构维数
```

```

    OUT      coords      数组, 长度为 maxdim, 含待查询进程的 Cartesian 坐标
C      int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdim, int * coords)
Fortran MPI_CART_COORDS (COMM, RANK, MAXDIM, COORDS, IERROR)
      INTEGER COMM, RANK, MAXDIM, COORDS (*), IERROR

```

基于包含 Cartesian 拓扑结构的通信器 comm, 函数 MPI_CART_COORDS 返回序号为 rank 的进程的坐标, 并存储在参数 coords 中。

/* Cartesian 拓扑结构相邻进程查询函数, 查询某个进程沿某个方向的相邻进程。

```

MPI_CART_SHIFT (comm, disp, direction, rank_source, rank_dest)
    IN      comm      含 Cartesian 拓扑结构的通信器
    IN      disp      查询的坐标维
    IN      direction  查询的坐标方向
    OUT     rank_source 源进程的序号
    OUT     rank_dest   目的进程的序号
C      int MPI_Cart_shift (MPI_Comm comm, int disp, int direction,
                           int * rank_source, int * rank_dest)
Fortran MPI_CART_SHIFT (COMM, DISP, DIRECTION, RANK_SOURCE,
                        RANK_DEST, IERROR)
      INTEGER COMM, DISP, DIRECTION, RANK_SOURCE,
              RANK_DEST, IERROR

```

函数 MPI_CART_SHIFT 沿方向 direction (>0 表示沿坐标值大的方向, <0 表示沿坐标值小的方向), 在拓扑结构的第 disp 维 (disp=0, 1, ..., n-1, 其中 n 为拓扑结构的维数) 执行一次移位查询操作。这样, 调用该函数的进程就会获得它在拓扑结构中第 disp 维的两个相邻进程: 源进程和目的进程, 其中, 源进程是指沿移位逆方向, 函数调用进程的相邻进程, 而目的进程是指沿移位方向, 函数调用进程的相邻进程。参数 rank_source 和参数 rank_dest 分别记录源进程和目的进程的序号。

例 7.3 Cartesian 拓扑结构查询函数应用示例

基于图 7.1 中的二维 Cartesian 3×4 网格拓扑结构, 下面的程序演示以上各拓扑结构查询函数的使用。

```

INTEGER RANK, COORDS(2), SOURCE, DEST, DISP, DIRECTION
INTEGER NDIM, DIMS(2), PERIODS(2), COORDS0(2)
INTEGER SIZE, COMM1, RANK0, RANK1, IERR
!
CALL MPI_COMM_RANK(COMM1, RANK0, IERR)
CALL MPI_COMM_SIZE(COMM1, SIZE, IERR)
IF(RANK0.EQ.5)THEN ! 仅 5 号进程执行以下拓扑结构查询。
  CALL MPI_CARTDIM_GET(COMM1, NDIM, IERR)
  ! 通信器 COMM1 包含的 Cartesian 拓扑结构的维数为 NDIM。

```

```

IF(NDIM.NE.2)PRINT *, "+++ ERROR : NDIM ! = 2 + + +"
CALL MPI_CART_GET(COMM1, NDIM, DIMS, PERIODS, COORDS, IERR)
! 获得维数 DIMS (1) 和 DIMS (2), 二维的连接信息 periods 和调用
! 进程第 0 维和第 1 维的坐标 Coords (1) 和 Coords (2)。
IF(DIMS(1) * DIMS(2).NE.SIZE)PRINT *, "ERROR: DIMS + + +"
IF(PERIODS(1).NE..FALSE..OR..PERIODS(2).NE..FALSE..)
    PRINT *, "+++ ERROR: DIMS + + +"
CALL MPI_CART_RANK(COMM1, COORDS, RANK, IERR)
! 获得通信器 COMM1 中坐标为 Coords 的进程的序号。
IF(RANK.NE.RANK0) PRINT *, "+++ ERROR: RANK ! = RANK0 + + +"
CALL MPI_CART_COORDS(COMM1, RANK, NDIM, COORDS0, IERR)
IF(COORDS(1).NE.COORDS0(1).OR.COORDS(2).NE.COORDS0(2))
    PRINT *, "+++ ERROR: COORDS0 ! = COORDS + + +"
COORDS(1)=COORDS(1)-1
COORDS0(1)=COORDS0(1)+1
CALL MPI_CART_RANK(COMM1, COORDS, RANK0, IERR)
CALL MPI_CART_RANK(COMM1, COORDS0, RANK1, IERR)
CALL MPI_CART_SHIFT(COMM1, 0, 1, SOURCE, DEST, IERR)
IF(SOURCE.NE.RANK0.OR.DEST.NE.RANK1)
    PRINT *, "+++ ERROR: SHIFT + + +"
CALL MPI_CART_SHIFT(COMM1, 0, -1, SOURCE, DEST, IERR)
IF(SOURCE.NE.RANK1.OR.DEST.NE.RANK0)
    PRINT *, "+++ ERROR: SHIFT + + +"
ENDIF

```

在给定的 Cartesian 网格拓扑结构中, 通过以上定义的查询函数, 各个进程可以很方便地获得与之相邻的进程的序号, 从而组织消息传递。例如, 在通常的二维流体力学程序中, 基于区域分解的并行策略, 每个进程将分配一个子区域, 且消息传递一般只需沿着相邻子区域的边界进行, 这样, 也只有分配相邻子区域的进程之间才需要消息传递。对应于图 7.1, 假设求解的流体力学问题的离散格式为经典的五点有限差分, 每个网格单元上定义一个进程, 而某个进程的坐标为 (i, j) , 则该进程只需与其上、下、左、右共 4 个进程进行通信, 而这 4 个进程的坐标依次为 $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$, 从而通过这些坐标可以很方便地查询到这 4 个进程的序号, 方便了并行程序的设计。

7.1.4 Cartesian 拓扑结构分解

```

/* Cartesian 拓扑结构分解函数, 按坐标的不同维, 将某个通信器分解成多个独立的新
/* 通信器, 且每个新通信器继承它包含的所有进程在原通信器中的拓扑结构。

```

```

MPI_CART_SUB(comm, remain_dims, newcomm)
IN      comm      含 Cartesian 拓扑结构的通信器
IN      remain_dims 数组, 具体见下面解释

```

```

    OUT      newcomm 新通信器
C      int MPI_Cart_sub (MPI_Comm comm, int * remain_dims,
                      MPI_Comm * newcomm)
Fortran MPI_CART_SUB (COMM, REMAIN_DIMS, NEWCOMM, IERROR)
      INTEGER COMM, NEWCOMM, IERROR
      LOGICAL REMAIN_DIMS (*)

```

函数 MPI_CART_SUB 需要通信器 comm 所有进程的共同参与, 它根据各进程提供的参数 remain_dims, 确定如何分解通信器 comm, 形成多个独立的新通信器 newcomm。其中, 参数 remain_dims 是长度为 ndim 的数组, ndim 是原通信器 comm 包含的拓扑结构的维数。若 remain_dims(i) = true, 表示沿第 i-1 维的所有进程将被保留在新通信器中; 若 remain_dims(i) = false, 则表示第 i-1 维的进程将排除在新通信器之外。另外, 新通信器 newcomm 将继承它所包含的所有进程在原通信器 comm 中的 Cartesian 网格拓扑结构。

例如, 假设 MPI_CART_CREATE 定义了一个 $2 \times 3 \times 4$ 网格拓扑结构, 令 remain_dims = (true, false, true), 则所有进程的函数调用:

```
CALL MPI_CART_SUB (comm, remain_dims, comm_new, ierr)
```

将创建三个独立的通信器, 每个包含 8 个进程, 具有 2×4 Cartesian 拓扑结构。若 remain_dims = (false, false, true), 则将创建 6 个独立的通信器, 每个含 4 个进程, 具有一维 Cartesian 拓扑结构。

7.1.5 Cartesian 拓扑结构映射

为了更好地映射 Cartesian 拓扑结构到具体并行机, MPI 系统提供函数 MPI_CART_MAP, 重新对所有进程进行排序。

/* Cartesian 拓扑结构映射函数, 为每个进程映射一个新的序号。

	MPI_CART_MAP (comm, ndim, dims, periods, newrank)	
IN	comm	被映射的通信器
IN	ndim	被映射 Cartesian 拓扑结构的维数
IN	dims	数组, 含 Cartesian 拓扑结构各维的进程个数
IN	periods	数组, 含各维的进程是否周期连接的信息
OUT	newrank	进程的新序号

```

C      int MPI_Cart_map (MPI_Comm comm, int ndim, int * dims, int * periods,
                      int * newrank)
Fortran MPI_CART_MAP (COMM, NDIM, DIMS, PERIODS, NEWRANK, IERROR)
      INTEGER COMM, NDIM, DIMS, PERIODS, NEWRANK, IERROR

```

函数 MPI_CART_MAP 优化各个进程在并行机中的具体位置, 返回新序号 newrank, 其参数的含义与函数 MPI_CART_CREATE 一致。

7.2 图拓扑结构

在 7.1 节中, 我们讨论了 Cartesian 拓扑结构的创建、查询、分解和映射, 本节进一步介绍更普通的拓扑结构: 图拓扑结构, 它通过图的结点和连线分别表示进程和进程间通信。Cartesian 拓扑结构是图拓扑结构的特例。

7.2.1 图拓扑结构创建

```
/* 图拓扑结构创建函数, 给定一个图, 在原通信器上创建一个以该图为拓扑结构的新  
/* 通信器。
```

```
MPI_GRAPH_CREATE (comm_old, nnode, indexs, edges, reorder, comm_graph)  
    IN      comm_old    原通信器  
    IN      nnode       图拓扑结构包含的结点数  
    IN      indexs      长度为 nnode 的数组 (见下面解释)  
    IN      edges       数组 (见下面解释)  
    IN      reorder     true 表示进程将在新通信器中被重新排序  
    OUT     comm_graph  含图拓扑结构的新通信器  
  
C      int MPI_Graph_create (MPI_Comm comm_old, int nnode, int * indexs,  
                           int * edges, int reorder, MPI_Comm * comm_graph)  
  
Fortran MPI_GRAPH_CREATE (COMM_OLD, NNODE, INDEXS, EDGES,  
                           REORDER, COMM_GRAPH, IERROR)  
INTEGER COMM_OLD, NNODE, INDEXS (*), EDGES (*),  
REORDER, COMM_GRAPH, IERROR
```

函数 MPI_GRAPH_CREATE 需要所有进程的共同参与, 且要求各进程提供完全一致的参数 (nnode, indexs, edges), 函数调用将返回包含该图拓扑结构的新通信器 comm_graph。

参数 (nnode, indexs, edges) 唯一地定义了一个图拓扑结构。其中, nnode 表示图中结点个数, 结点编号依次为 $0, \dots, nnode - 1$ 。给定某个结点, 称与之相连的结点为它的**邻居结点**, 而所有邻居结点的总数为该结点的**邻居结点数**。参数 indexs 是长度为 nnode 的数组, 其元素 indexs(i) 包含前 i 个结点的邻居结点数之和。特别地, 称数组 indexs 为该图拓扑结构的**邻居结点数组**。参数 edges 也是一个整型数组, 是该图拓扑结构所有边的展开表示, 其元素依次记录的是结点 $0, \dots, nnode - 1$ 的邻居结点的序号。显然, 数组 edges 的长度是图拓扑结构边的条数的两倍。特别地, 称之为该图拓扑结构的**边数组**。因此, 结点个数、邻居结点数组和边数组唯一地定义了一个图拓扑结构。

给定一个通信器, 我们可以采用图拓扑结构来描述各个进程之间的通信关系, 并用参数 (nnode, indexs, edges) 来唯一表示, 其中结点代表各个进程, 结点之间的连线代表被连接的两个进程之间存在相互通信关系。

例 7.4 图拓扑结构创建函数应用示例

假设某个通信器 COMM0 包含 4 个进程,且具有如下的通信关系:



则图拓扑结构可表示为 ($nnode = 4$, $indexs = (2, 3, 4, 6)$, $edges = (1, 3, 0, 3, 0, 2)$),各进程通过如下的函数调用:

```
CALL MPI_GRAPH_CREATE(COMM0, nnode, indexs, edges, .FALSE., NEW_COMM,  
                      IERR)
```

即可获得包含该图拓扑结构的新通信器 NEW_COMM。

7.2.2 图拓扑结构查询

一个图拓扑结构建立后,可通过如下的函数来查询它的相关信息。

/* 图拓扑结构查询函数,查询图拓扑结构包含的结点个数和边的条数。

```
MPI_GRAPHDIMS_GET(comm, nnode, nedge)  
  IN      comm      含图拓扑结构的通信器  
  OUT     nnode     图中结点总数  
  OUT     nedge     图中边的条数  
C      int MPI_Graphdims_get(MPI_Comm comm, int *nnode, int *nedge)  
Fortran MPI_GRAPHDIMS_GET(COMM, NNODE, NEDGE, IERROR)  
        INTEGER COMM, NNODE, NEDGE, IERROR
```

函数 MPI_GRAPHDIMS_GET 返回通信器 comm 包含的图拓扑结构的结点总数和边的条数,并分别存储在变量 nnode 和变量 nedge 中。

例如,对例 7.4 中创建的新通信器 NEW_COMM,函数调用:

```
CALL MPI_GRAPHDIMS_GET(NEW_COMM, nnode, nedge, IERR)
```

将返回变量 $nnode = 4$, $nedge = 3$ 。

/* 图拓扑结构查询函数,查询图拓扑结构包含的邻居结点数组和边数组。

```
MPI_GRAPH_GET(comm, maxindexs, maxedges, indexs, edges)  
  IN      comm      含图拓扑结构的通信器  
  IN      maxindexs    图拓扑结构包含的结点总数  
  IN      maxedges     图拓扑结构边的条数的二倍,也就是数组 edges 的长度。  
  OUT     indexs      图拓扑结构的邻居结点数组(见函数 MPI_GRAPH_  
                      CREATE 的参数解释)  
  OUT     edges       图拓扑结构的边数组(见函数 MPI_GRAPH_CREATE 的参  
                      数解释)
```

```

C      int MPI_Graph_get (MPI_Comm comm, int maxindexs, int maxedges, int * indexs, int * edges)
Fortran MPI_GRAPH_GET (COMM, MAXINDEXS, MAXEDGES, INDEXS, EDGES, IERROR)
      INTEGER COMM, MAXINDEXS, MAXEDGES, INDEXS(*), EDGES(*), IERROR

```

函数 MPI_GRAPH_GET 返回通信器 comm 包含的图拓扑结构的邻居结点数组 indexs 和边数组 edges。例如, 对例 7.4 中创建的新通信器 NEW_COMM, 函数调用:

```
CALL MPI_GRAPH_GET(NEW_COMM, 4, 6, indexs, edges, ierr)
```

将返回数组 indexs = (2, 3, 4, 6), edges = (1, 3, 0, 3, 0, 2)。

/* 图拓扑结构查询函数, 返回某个进程的邻居结点数。

```

MPI_GRAPH_NEIGHBORS_COUNT (comm, rank, nneighbor)
      IN      comm      含图拓扑结构的通信器
      IN      rank      进程序号
      OUT     nneighbor  进程 rank 的邻居结点数
C      int MPI_Graph_neighbors_count (MPI_Comm comm, int rank, int * nneighbor)
Fortran MPI_GRAPH_NEIGHBORS_COUNT (COMM, RANK, NNEIGHBOR, IERROR)
      INTEGER COMM, RANK, NNEIGHBOR, IERROR

```

函数 MPI_GRAPH_NEIGHBORS_COUNT 返回进程 rank 的邻居结点数。例如, 对例 7.4 中创建的新通信器 NEW_COMM, 函数调用:

```
CALL MPI_GRAPH_NEIGHBORS_COUNT(NEW_COMM, 3, nneighbor, ierr)
```

将返回变量 nneighbor = 2, 而函数调用:

```
CALL MPI_GRAPH_NEIGHBORS_COUNT(NEW_COMM, 1, nneighbor, ierr)
```

将返回变量 nneighbor = 1。

/* 图拓扑结构查询函数, 返回某个进程的所有邻居结点。

```

MPI_GRAPH_NEIGHBORS (comm, rank, maxneighbors, neighbors)
      IN      comm      含图拓扑结构的通信器
      IN      rank      进程序号
      IN      maxneighbors 数组 neighbors 的长度
      OUT     neighbors  进程 rank 的所有邻居结点序号组成的数组
C      int MPI_Graph_neighbors (MPI_Comm comm, int rank, int maxneighbors,
                           int * neighbors)
Fortran MPI_GRAPH_NEIGHBORS (COMM, RANK, MAXNEIGHBORS,
                           NEIGHBORS, IERROR)
      INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR

```

函数 MPI_GRAPH_NEIGHBORS 返回通信器 comm 中进程 rank 的所有邻居结点, 并按由小到大的序号, 将这些结点依次存储在数组 neighbors 中。例如, 对例 7.4 中创建的新通信器 NEW_COMM, 函数调用:

```
CALL MPI_GRAPH_NEIGHBORS(NEW_COMM, 0, 2, neighbors, ierr)
```

将返回数组 neighbors = (1, 3), 而函数调用:

```
CALL MPI_GRAPH_NEIGHBORS(NEW_COMM, 3, 2, neighbors, ierr)
```

将返回数组 neighbors = (0, 2)。

7.2.3 图拓扑结构映射

/* 图拓扑结构映射函数, 为每个进程映射一个新的序号。

```
MPI_GRAPH_MAP (comm, nnode, indexs, edges, newrank)
    IN      comm      通信器
    IN      nnode     图拓扑结构的结点总数
    IN      indexs    图拓扑结构的邻居结点数组
    IN      edges     图拓扑结构的边数组
    OUT     newrank   新的进程序号
C      int MPI_Graph_map (MPI_Comm comm, int nnode, int * indexs, int * edges,
                        int * newrank)
Fortran MPI_GRAPH_MAP (COMM, NNODE, INDEXS, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODE, INDEXS(*), EDGES(*), NEWRANK, IERROR
```

类似于函数 MPI_CART_MAP, 函数 MPI_GRAPH_MAP 映射一个图拓扑结构到具体并行机, 并为每个进程返回最有利于通信的新序号。

7.3 拓扑结构类型查询

/* 图拓扑结构类型查询函数, 返回某个通信器包含的拓扑结构的类型。

```
MPI_TOPO_TEST (comm, type)
    IN      comm      通信器
    OUT     type      通信器 comm 包含的拓扑结构类型
C      int MPI_Topo_test (MPI_Comm comm, int * type)
Fortran MPI_TOPO_TEST (COMM, TYPE, IERROR)
    INTEGER COMM, TYPE, IERROR
```

函数 MPI_TOPO_TEST 返回通信器 comm 包含的拓扑结构的类型: type = MPI_GRAPH 表示图拓扑结构; type = MPI_CART 表示 Cartesian 拓扑结构; type = MPI_UNDEFINED 该通信器不包含任何拓扑结构。例如, 如下的函数调用:

```
CALL MPI_TOPO_TEST (MPI_COMM_WORLD, TYPE, IERR)
```

将返回参数 TYPE = MPI_UNDEFINED, 因为 MPI 系统提供的通信器 MPI_COMM_WORLD 不包含任何拓扑结构。

第八章 并行 I/O

目前为止,本书还没有讨论 MPI 程序的输入/输出(I/O)。实际上,MPI 1.0 版本并没有对 MPI 程序的 I/O 做任何规定。但是,由于受操作系统和编程语言的限制,一个文件(或标准输入/标准输出),在同一时刻,一般只能被一个进程访问,如果存在多个进程要求同时访问一个文件,且其中一个执行的是写操作,则这些进程只能串行访问,读写的次序和数据的正确性均无法得到保证。因此,基于 MPI 1.0 版本,MPI 程序的 I/O 通常只能按以下两种方法来组织:

第一,进程 0 负责所有进程的 I/O,即进程 0 读入其他各进程需要的数据,并将这些数据传送给其他各进程;另一方面,其他各进程将它们各自输出的数据传送给进程 0,由进程 0 负责具体输出。这种方法是典型的串行方法,我们称之为**串行 I/O**。

第二,各进程执行完全独立的 I/O 操作,即不同进程访问不同的文件。这样,MPI 程序可以执行并行 I/O 操作,但与 MPI 系统无关,故称之为**非 MPI 并行 I/O**。

尽管非 MPI 并行 I/O 达到了并行化 I/O 的目的,但是,它存在二个明显的缺点:(1)为了便于数据的理解和后处理,MPI 程序通常希望得到与串行程序一致的输出文件,但非 MPI 并行 I/O 无法提供这项功能,因为执行 MPI 程序的各个进程将形成各自独立的零碎文件,而整理这些文件是非常繁琐的;(2)如果并行计算机是异构的,不同处理机的数据表示格式不同,则非 MPI 并行 I/O 文件的整理还涉及到数据格式的转换,情况变得更加复杂。

因此,为了简化并行 I/O 程序设计和提高并行 I/O 的性能,MPI 2.0 版本在 1.0 版本的基础上,提供了一类特殊的函数,它们能辅助执行 MPI 程序的各个进程并行地访问不同的文件,或者并行地访问同一个文件,我们称之为**MPI 并行 I/O 函数**,而这些函数执行的并行 I/O 操作,我们称之为**MPI 并行 I/O**。

目前,在 MPICH 1.2 版本和各类并行机中,所有 MPI 并行 I/O 函数已经得到了具体实现。本章将从串行 I/O、非 MPI 并行 I/O、MPI 并行 I/O 等三个方面讨论 MPI 程序的并行 I/O,其中,最后一个是我们讨论的重点。

8.1 串行 I/O

MPI 1.0 版本不提供专门的并行 I/O 函数,但对操作系统和编程语言提供的 I/O 函数也没有任何限制。因此,用户通常根据编程语言提供的串行 I/O 函数来组织 MPI 应用程序的 I/O。其中,一个最直接的方法是由其中一个进程,例如进程 0,来负责所有进程的 I/O。具体地,如图 8.1 所示,当某个进程要求从一个文件读数据时,进程 0 首先读这些数据,然后将它们发送给该进程;当某个进程要求向一个文件写数据时,首先将这些数据发送给进程 0,进程 0 接收这些数据后,将它们按某种顺序写入文件。由于这种方法的所有 I/O 操作在进程 0 均是串行执行的,因此,我们称之为**串行 I/O**。

例 8.1 给出了一个基于串行 I/O 的 MPI 程序, 其中, 每个进程均向进程 0 发送 100 个整型数, 进程 0 接收这些数据, 并按由小到大的进程序号, 将它们写入某个文件中。

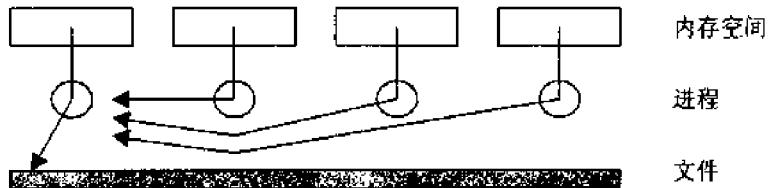


图 8.1 MPI 并行程序的串行 I/O 示意图

例 8.1 并行 MPI 程序的串行 I/O。

```

PROGRAM SEQIO
INCLUDE "mpif.h"
PARAMETER (NSIZE=100)
INTEGER I, MYRANK, NPROC, BUF(NSIZE)
INTEGER STATUS(MPI_STATUS_SIZE)
!
CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROC, IERR)
DO I=1, NSIZE
    BUF(I)=MYRANK * NSIZE + I
ENDDO
IF(MYRANK .NE. 0)THEN
    CALL MPI_SEND(BUF, NSIZE, MPI_INTEGER, 0, 99,
                  MPI_COMM_WORLD, IERR)
ELSE
    OPEN(10, FILE = "testfile", STATUS = "unknown")
        ! 进程 0 打开一个名为"testfile"的文件
    DO I=1, NPROC - 1
        CALL MPI_RECV(BUF, NSIZE, MPI_INTEGER, I, 99,
                      MPI_COMM_WORLD, STATUS, IERR)
        WRITE(10, *) (BUF(I), I=1, NSIZE)           ! 进程 0 将数据写入文件。
    ENDDO
    CLOSE(10)                                     ! 进程 0 关闭文件。
ENDIF
!
CALL MPI_FINALIZE(IERR)
END

```

例 8.1 和图 8.1 讨论的是多个进程如何访问同一个文件, 这种方法的好处主要有:

(1) 可移植性: 某些并行机可能只支持一个进程执行 I/O;

(2) 用户可以直接使用高级的数据管理库函数,而不必局限于编程语言提供的简单 I/O 函数;

(3) 形成的单一文件便于后处理(例如数据可视化和文件拷贝);

(4) 受 I/O 设备的限制,如果在同一时刻,并行机只允许写一个文件,则串行 I/O 还有利于 MPI 程序的性能提高,因为进程 0 可以集中所有进程的数据,执行一次大数据量的读/写操作,这比多次小数据量的读写操作的效率要高。

这里,我们还必须区别对待另一类特殊的 I/O 函数,就是通常所说的标准输入/输出函数。例如,某个进程要求从当前终端屏幕读入数据,或者某个进程要求向当前终端屏幕输出数据。对此,MPI 标准规定:

第一,只有进程 0 具备标准输入功能,其他进程的标准输入只能通过进程 0 执行相应的标准输入之后,将输入数据发送给它;如果其他进程直接执行标准输入,将导致该进程的死锁。

第二,所有进程均具备标准输出功能,即任意进程可以直接向标准输出设备写数据,但是,MPI 系统的策略是哪个进程先写,哪个进程先输出。因此,MPI 系统无法保证数据按进程的序号输出,敬请读者注意。

8.2 非 MPI 并行 I/O

如果并行机配备有多台 I/O 设备,且在同一时刻,允许有多个文件被并行访问,则上节介绍的串行 I/O 无法发挥该并行机的并行 I/O 性能。此时,我们可以采用另外一种方法,就是允许执行 MPI 程序的各个进程同时访问不同的文件,从而达到并行 I/O 的目的。由于这种方法直接利用编程语言的 I/O 函数就可以完成,而不需要 MPI 系统的参与,故我们称之为非 MPI 并行 I/O。

针对图 8.1,图 8.2 给出了非 MPI 并行 I/O 示意图,其中,不同的进程并行访问不同的文件。针对例 8.1,例 8.2 给出了相应问题的非 MPI 并行 I/O 实现,其中,每个进程打开不同的文件,并对各自打开的文件执行 I/O 操作。

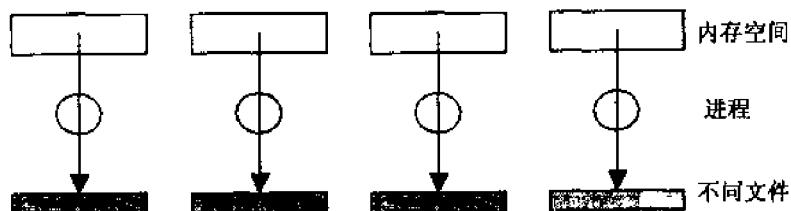


图 8.2 MPI 程序的非 MPI 并行 I/O 示意图

例 8.2 MPI 程序的非 MPI 并行 I/O

```
PROGRAM N_MPIPIO
INCLUDE "mpif.h"
PARAMETER (NSIZE=100)
INTEGER I, MYRANK, NPROC, BUF (NSIZE)
```

```

INTEGER STATUS(MPI_STATUS_SIZE)
CHARACTER*20 testfile
!
CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROC, IERR)
DO I=1, NSIZE
    BUF(I) = MYRANK * NSIZE + I
ENDDO
testfile[1:8] = "testfilc"
testfile[9:12] = string(myrank) ! 不同进程访问的不同文件名,例如 testfile0000,
                                ! testfile0001, testfile0002, ...
OPEN(10, FILE = testfile, STATUS = "unknown") ! 各进程打开不同的文件。
WRITE(10, *) (BUF(I), I=1, NSIZE)           ! 各进程将数据写入各自的文件。
CLOSE(10)                                     ! 各进程关闭各自的文件。
!
CALL MPI_FINALIZE(IERR)
END

```

在 MPI 系统不提供并行 I/O 函数的前提下,非 MPI 并行 I/O 能组织 MPI 程序的并行 I/O,对那些需要大规模数据输出的问题,它是一个值得推广应用的方法。例如,在微机机群中,每台微机均拥有一个独立的硬盘和一个独立的文件读写设备,我们可以利用非 MPI 并行 I/O 来充分发挥微机机群的并行 I/O 性能。

非 MPI 并行 I/O 尽管能发挥并行机的并行 I/O 性能,但是它将形成多个分布在各台处理机中的文件,而只有并行程序设计人员充分理解这些文件后,才可能通过一些方法,形成类似于串行 I/O 输出的单个文件,便于数据后处理。

8.3 MPI 并行 I/O: 并行访问不同的文件

MPI 2.0 版本提供的 MPI 并行 I/O 函数可分为两类:第一类是各个进程并行访问不同的文件,第二类是各个进程并行访问同一个文件。本节将介绍第一类。

执行 MPI 程序的各个进程访问不同文件的并行 I/O,非常类似于 8.2 节中介绍的非 MPI 并行 I/O。实际上,我们只需将非 MPI 并行 I/O 的文件访问函数替换成 MPI 的并行 I/O 函数,便可将基于非 MPI 并行 I/O 的程序改造成基于 MPI 并行 I/O 的程序。例 8.3 给出了对应于例 8.2 的 MPI 并行 I/O 程序,它只改变了例 8.2 的三个文件访问函数,具体如下:

例 8.3 MPI 程序的 MPI 并行 I/O

```

PROGRAM MPI_PIO
INCLUDE "mpif.h"
PARAMETER (NSIZE=100)

```

```

INTEGER I, MYRANK, NPROC, BUF (NSIZE), MYFILE
INTEGER STATUS (MPI_STATUS_SIZE)
CHARACTER * 20 testfile
!
CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROC, IERR)
DO I=1, NSIZE
    BUF(I) = MYRANK * NSIZE + I
ENDDO
testfile[1:8] = "testfile"
testfile[9:12] = string(myrank) ! 不同进程访问的不同文件名。
CALL MPI_FILE_OPEN(MPI_COMM_SELF, testfile, MPI_MODE_WRONLY +
    MPI_MODE_CREATE, MPI_INFO_NULL, MYFILE, IERR)
                                ! 各进程打开不同的文件。
CALL MPI_FILE_WRITE(MYFILE, BUF, NSIZE, MPI_INTEGER,
    MPI_STATUS_IGNORE, IERR) ! 各进程将数据写入各自的文件。
CALL MPI_FILE_CLOSE(MYFILE, IERR) ! 各进程关闭各自的文件
!
CALL MPI_FINALIZE(IERR)
END

```

在上例中，函数 MPI_FILE_OPEN 用于打开一个名为 testfile 的文件，具体定义如下：

/* MPI 并行文件打开函数，属于同一个通信器的各进程打开不同的文件，
/* 或者各进程打开同一个文件

```

MPI_FILE_OPEN (comm, filename, mode, info, myfile)
    IN      comm      各进程所属的通信器
    IN      filename  各进程打开的文件名
    IN      mode       文件的创建或访问模式
    IN      info       各进程提交给 MPI 系统的附加提示信息
    OUT     myfile    各进程获得的文件链接器
C      MPI_File_open (MPI_Comm comm, char * filename, int mode, MPI_Info info,
                    MPI_File * myfile)
Fortran MPI_FILE_OPEN (COMM,FILENAME,MODE,INFO,MYFILE,IERROR)
    INTEGER        COMM, MODE, INFO, MYFILE, IERROR
    CHARACTER( *) FILENAME

```

函数 MPI_FILE_OPEN 属于聚合通信函数，它需要通信器 comm 包含的所有进程的共同参与。第 1 个参数 comm 表示打开文件的进程所属的通信器，若 comm = MPI_COMM_SELF，则表示各个进程独立打开不同的文件，它要求各进程提供的参数

filename 不同；若 comm = MPI_COMM_WORLD，则要求属于该通信器的所有进程必须提供相同的参数 filename，它表示该通信器的所有进程将访问同一个文件。这里，我们只讨论 comm = MPI_COMM_SELF 的情形，comm = MPI_COMM_WORLD 的情形将在下节讨论。

函数 MPI_FILE_OPEN 的第 2 个参数 filename 表示各进程将要打开的文件名，第 3 个参数 mode 表示打开文件时，赋予该进程访问文件的权限，MPI 系统支持四种权限：

- (1) mode = MPI_MODE_CREATE：表示创建一个新文件；
- (2) mode = MPI_MODE_RDONLY：表示打开的文件只能被读；
- (3) mode = MPI_MODE_WRONLY：表示被打开的文件只能被写；
- (4) mode = MPI_MODE_RDWR：表示打开的文件既可被读也可被写。

这四种权限还可以组合使用，多个选项可以用符号“+”联接。例如，mode = (MPI_MODE_CREATE + MPI_MODE_WRONLY) 表示将创建一个新文件，且该文件只能被写入数据。

函数 MPI_FILE_OPEN 的第 4 个参数，是执行 MPI 程序的各个进程为了提高并行 I/O 性能，提交给 MPI 系统的附加提示信息，我们将在 8.7 节介绍。在例 8.3 中，常数 MPI_INFO_NULL 表示没有提示信息。

函数 MPI_FILE_OPEN 的第 5 个参数 myfile 称为文件联接器，表示它将充当进程与文件之间的联接桥梁。对 C 语言，它可以认为是文件描述符；对 Fortran 语言，它可以认为是访问文件的通道。同时，该文件联接器隐含一个文件指针，当函数 MPI_FILE_OPEN 返回时，指向文件的初始位置。函数 MPI_FILE_OPEN 被执行后，后续的所有 MPI 并行 I/O 函数均必须通过参数 myfile 才能访问文件 filename。

在例 8.3 中，函数 MPI_FILE_WRITE 用于向函数 MPI_FILE_OPEN 打开的文件写入数据，具体定义如下：

```
/* MPI 并行写函数，各进程将各自数据写入不同的文件或者同一个文件。
```

	MPI_FILE_WRITE (fh, buf, count, datatype, status)	
IN	fh	文件联接器
IN	buf	待写入文件的数据缓存区初始地址
IN	count	待写入文件的数据单元个数
IN	datatype	待写入文件的数据单元类型
INOUT	status	数据写入状态信息
C	MPI_File_write (MPI_File fh, void * buf, int count, MPI_Datatype datatype, MPI_Status * status)	
Fortran	MPI_FILE_WRITE (FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, BUF, COUNT, DATATYPE, STATUS, IERROR	

函数 MPI_FILE_WRITE 将缓存区 (buf, count, datatype) 包含的连续 count 个类型为 datatype 的数据单元，写入文件联接器 fh 联接的文件中。函数 MPI_FILE_WRITE 返回后，参数 status 可被函数 MPI_GET_COUNT 和函数 MPI_GET_ELEMENT 用于查询该次文件写操作实际写入的数据单元个数。此外，该参数在这里还可以作为输入

参数使用,例如当 `status = MPI_STATUS_IGNORE` 时,表示忽略参数 `status` 的输出。注意,在例 8.3 中,我们取 `status = MPI_STATUS_IGNORE`。

在例 8.3 中,函数 `MPI_FILE_CLOSE` 用于关闭一个被并行打开的文件,该函数定义如下:

```
/* MPI 并行文件关闭函数,属于同一个通信器的各进程关闭已并行打开的文件。
```

```
MPI_FILE_CLOSE (myfile)
    IN          myfile      待关闭的文件联接器
C     MPI_File_close (MPI_File myfile)
Fortran MPI_FILE_CLOSE (MYFILE, IERROR)
        INTEGER      MYFILE, IERROR
```

这里,我们还有必要介绍另一个 MPI 并行 I/O 函数,即 `MPI_FILE_READ`,它是函数 `MPI_FILE_WRITE` 的逆函数,表示各个进程将从不同的文件或同一个文件中读取数据。具体定义如下:

```
/* MPI 并行读函数,各进程从不同的文件或者同一个文件中读取数据。
```

```
MPI_FILE_READ (fh, buf, count, datatype, status)
    IN      fh      文件联接器
    IN      buf      存储数据的缓存区初始地址
    IN      count    读取的数据单元个数
    IN      datatype 读取的数据单元类型
    INOUT   status   数据读取状态信息
C     MPI_File_read (MPI_File fh, void * buf, int count, MPI_Datatype datatype,
                    MPI_Status * status)
Fortran MPI_FILE_READ (FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
        INTEGER      FH, BUF, COUNT, DATATYPE, STATUS, IERROR
```

函数 `MPI_FILE_READ` 从文件联接器 `fh` 联接的文件中,读取连续 `count` 个类型为 `datatype` 的数据单元,并将它们依次存储在数据缓存区 (`buf, count, datatype`) 中。参数 `status` 类似于函数 `MPI_FILE_WRITE` 的参数 `status`,通过函数 `MPI_GET_COUNT` 和函数 `MPI_GET_ELEMENT` 可以查询实际读取的数据单元个数。但是,该参数在这里还可以作为输入参数使用,例如当 `status = MPI_STATUS_IGNORE` 时,表示 MPI 应用程序将忽略该函数返回状态信息。

8.4 MPI 并行 I/O: 并行访问同一个文件

当前,许多并行机能够提供充分的硬件资源和高性能并行文件系统,例如 IBM PIOFS, IBM GPFS, SGI XFS, HP HFS 和 NEC SFS 等等,它们均支持多个进程并行访问同一个文件。因此,MPI 2.0 版本提供并行 I/O 函数,使得 MPI 程序的各个进程可并行

访问同一个文件，获得并行机提供的高性能文件系统服务。

根据不同的特征，MPI 2.0 版本的并行 I/O 函数可以分为简单 I/O 函数、非连续 I/O 函数、聚合 I/O 函数、阵列 I/O 函数、非阻塞 I/O 函数和分步聚合 I/O 函数等几类，并在 MPICH 1.2 版本以及当前流行的各种并行机中已具体实现。

这里有一点需要引起读者的特别注意：MPI 的并行 I/O 函数只支持对无格式数据文件的访问，如果 MPI 程序需要并行输出有格式的正文文件，则本节和上节介绍的并行 I/O 函数都不能使用，但 8.1 节介绍的串行 I/O 方法和 8.2 节介绍的非 MPI 并行 I/O 方法可用于完成此项任务。

8.4.1 简单的并行 I/O

考虑这样一个 MPI 程序例子：一个文件被等分成 n 段， n 个进程按由小到大的序号依次读取文件“/pfa/datafile”的不同段。具体程序在例 8.4 中给出。

例 8.4 多个进程从同一个文件中读不同的数据段

```
PROGRAM MPI_PIO
INCLUDE "mpif.h"
PARAMETER (NSIZE=1024 * 1024)
INTEGER I, MYRANK, NPROC, BUF (NSIZE), MYFILE, NINTS
INTEGER STATUS (MPI_STATUS_SIZE)
!
CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROC, IERR)
NINTS = NSIZE/NPROC
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, "/pfs/datafile",
                  MPI_MODE_RDONLY, MPI_INFO_NULL,
                  MYFILE, IERR)
! 各进程打开同一个文件，获得各自的文件联接器 MYFILE。
CALL MPI_FILE_SEEK(MYFILE, MYRANK * NINTS * SIZEOF(MPI_INTEGER),
                   MPI_SEEK_SET, IERR)
CALL MPI_FILE_READ(MYFILE, BUF, NINTS, MPI_INTEGER,
                  STATUS, IERR) ! 各进程从文件中读数据。
CALL MPI_FILE_CLOSE(MYFILE, IERR) ! 各进程释放各自的文件联接器。
!
CALL MPI_FINALIZE(IERR)
END
```

在上例中，我们使用了文件定位函数 MPI_FILE_SEEK，它非常类似于 C 语言中的文件定位函数 seek，可用于确定各进程文件联接器隐含的文件指针在文件中的位置。

/* MPI 并行 I/O 定位函数，确定各进程在文件中的访问位置。

MPI_FILE_SEEK (fh, offset, whence)

IN	fb	文件联接器
IN	offset	文件指针的偏移量(字节为单位)
IN	whence	文件指针偏移量的起始位置

C MPI_File_seek (MPI_File fb, MPI_Offset offset, int whence)

Fortran MPI_FILE_SEEK (FH, OFFSET, WHENCE, IERROR)

INTEGER	FH, WHENCE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND)	OFFSET

函数 MPI_FILE_SEEK 以字节为单位, 确定文件联接器 *fh* 隐含的文件指针, 在文件中的位置为 *whence* + *offset*。其中, 参数 *offset* 表示指针位置相对于某个初始位置的偏移量(字节为单位), 而该初始位置由参数 *whence* 提供(以字节为单位, 若 *whence* = MPI_SEEK_SET, 则表示初始位置为文件的起始位置)。特别对 Fortran 语言, 参数 *offset* 必须被说明为具有足够字长的整型数据类型, 能支持操作系统所允许的最大文件的指针定位。例如, 对 Fortran 90, 可将其说明为 KIND = MPI_OFFSET_KIND 类型的整型变量; 而对 Fortran 77, 一般可将其说明为 8 个字节或 16 个字节的整型变量。

在上例中, 调用函数 MPI_FILE_OPEN 之后, 各进程将获得各自独立的文件联接器, 且基于该联接器, 它们可以独立地访问文件的任何数据。第*i* 个进程调用函数 MPI_FILE_SEEK 之后, 文件指针将被定位在文件的第 *i* * *nints* * sizeof(MPI_INTEGER) 个字节, 而函数 MPI_FILE_READ 将从这个位置开始, 连续读取 *nints* 个整型数, 并存储在数组 *BUF* 中。

至此, 前面介绍的 5 个 MPI 并行 I/O 函数 MPI_FILE_OPEN, MPI_FILE_SEEK, MPI_FILE_READ, MPI_FILE_WRITE 和 MPI_FILE_CLOSE, 已经具备足够的能力来满足 MPI 并行应用程序的任何 I/O 需求, 而且, 这些函数的功能和使用方式也非常类似于 UNIX 操作系统提供的相应 I/O 函数。尽管如此, MPI 系统还提供一系列其他的并行 I/O 函数, 以简化并行编程、提高 I/O 性能和增强程序的可移植性, 我们将在后面各节中陆续介绍。

8.4.2 显式偏移并行 I/O

前面介绍的两个 MPI I/O 函数 MPI_FILE_READ 和 MPI_FILE_WRITE 具有这样一个特征, 就是各个进程拥有独立的文件指针, 且数据的读/写必须从文件指针给定的初始位置开始。因此, MPI 系统也称它们为**基于独立文件指针的 I/O 函数**。为了简化文件指针的定位, MPI 系统也提供另外两个 I/O 函数 MPI_FILE_READ_AT 和 MPI_FILE_WRITE_AT, 它们支持进程直接从文件的任意给定位置读/写数据, 而不必借助函数 MPI_FILE_SEEK 在读/写操作执行前为文件指针定位, 称之为**显式偏移 I/O 函数**。具体定义如下:

/* MPI 显式偏移 I/O 读函数, 各进程从指定的位置开始, 从文件中读取数据。

IN	fh	文件联接器
IN	offset	读取的数据在文件中的起始位置(字节为单位)

MPI_FILE_READ_AT (fh, offset, buf, count, datatype, status)

IN	buf	存储数据的缓存区初始地址
IN	count	读取的数据单元个数
IN	datatype	读取的数据单元类型
INOUT	status	数据读取状态信息

C MPI_File_read_at (MPI_File fh, MPI_Offset offset, void * buf, int count,
 MPI_Datatype datatype, MPI_Status * status)

Fortran MPI_FILE_READ_AT (FH,OFFSET,BUF,COUNT,DATATYPE,
 STATUS,IERROR)
 INTEGER FH,BUF,COUNT,DATATYPE,STATUS,IERROR
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

/* MPI 显式偏移 I/O 写函数, 各进程从指定的位置开始, 将数据写入文件中。

IN	fh	文件联接器
IN	offset	写入的数据在文件中的起始位置
IN	buf	待写入文件的数据缓存区初始地址
IN	count	待写入文件的数据单元个数
IN	datatype	待写入文件的数据单元类型
INOUT	status	数据写入状态信息

C MPI_File_write_at (MPI_File fh, MPI_Offset offset, void * buf, int count,
 MPI_Datatype datatype, MPI_Status * status)

Fortran MPI_FILE_WRITE_AT (FH,OFFSET,BUF,COUNT,DATATYPE,STATUS,IERROR)
 INTEGER FH,BUF,COUNT,DATATYPE,STATUS,IERROR
 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

函数 MPI_FILE_READ_AT 表示各个进程将从 fh 联接的文件的第 offset 个字节开始, 读取连续 count 个类型为 datatype 的数据单元, 并存储在数组 BUF 中。类似, 函数 MPI_FILE_WRITE_AT 表示各个进程将从 fh 联接的文件的第 offset 个字节开始, 将数组 BUF 包含的连续 count 个类型为 datatype 的数据单元写入文件中。参数 status 与它们在函数 MPI_FILE_READ 和函数 MPI_FILE_WRITE 中的定义一致, 可用于确定实际读取或写入的数据单元个数。

应用函数 MPI_FILE_READ_AT, 例 8.4 可改写如下。

例 8.5 多个进程从同一个文件中读不同的数据段

```

PROGRAM     MPI_PIO
INCLUDE "mpif.h"
PARAMETER (NSIZE = 1024 * 1024)
INTEGER    I, MYRANK, NPROC, BUF (NSIZE), MYFILE, NINTS
INTEGER    STATUS(MPI_STATUS_SIZE)
INTEGER * 8 OFFSET
!
CALL MPI_INIT(IERR)

```

```

CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROC, IERR)
NINTS = NSIZE/NPROC
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, "/pfs/datafile",
                  MPI_MODE_RDONLY, MPI_INFO_NULL,
                  MYFILE, IERR)
! 各进程打开同一个文件, 获得各自的文件联接器 MYFILE。
OFFSET = MYRANK * NINTS * SIZEOF(MPI_INTEGER)
CALL MPI_FILE_READ_AT(MYFILE, OFFSET, BUF, NINTS, MPI_INTEGER,
                      STATUS, IERR) ! 各进程从文件中读数据。
CALL MPI_FILE_CLOSE(MYFILE, IERR) ! 各进程释放各自的文件联接器。
!
CALL MPI_FINALIZE(IERR)
END

```

类似, 如果我们将上例中的函数 MPI_FILE_READ_AT 改写为 MPI_FILE_WRITE_AT, 并将函数 MPI_FILE_OPEN 中的参数 MPI_MODE_RDONLY 改写为 MPI_MODE_RDWR, 则上例表示, 各个进程将写同一个文件的不同段。

8.4.3 非连续访问并行 I/O

给定某个文件, 假设执行 MPI 程序的某个进程需要访问该文件的 n 个数据单元, 但这些数据单元存储在文件的非连续位置, 如果直接应用前面各节介绍的并行 I/O 函数, 则我们只能调用函数 MPI_FILE_READ 或者函数 MPI_FILE_WRITE 来一个一个地读/写这 n 个数据单元, 需要 n 次 I/O 操作。如果我们能够类似于第四章介绍的自定义数据类型一样, 在执行 I/O 操作之前, 调用特殊的 MPI 函数定义一个特殊的数据类型, 说明将要被访问的 n 个数据单元在文件中的位置, 则我们就可以调用函数 MPI_FILE_READ 或者函数 MPI_FILE_WRITE 一次地读/写这 n 个数据单元。显然, 由于减少了文件访问的次数, 后面提出的方法将会较好地改进 MPI 程序的并行 I/O 性能。我们称以上这种情形为**非连续访问并行 I/O**。

首先, 我们引入一个重要概念:**文件窗口**。给定一个文件, 函数 MPI_SET_VIEW 可用于确定该文件中哪些数据可以被某个进程访问, 并称这些数据为该进程在该文件中获得的**文件窗口**。在文件窗口内部, 所有数据单元可以认为是连续存储的。因此, 给定某个文件和进程将要访问的多个非连续存储的数据单元, 我们可以设置一个文件窗口, 包含所有这些将要被访问的数据单元。之后, 该进程对这些数据单元的访问等价于该进程对该文件窗口中连续数据单元的访问。

MPI 系统提供函数 MPI_SET_VIEW 来辅助 MPI 程序定义文件窗口, 具体定义如下:

/* MPI 文件窗口创建函数, 创建一个文件窗口。

MPI_FILE_SET_VIEW (fh, disp, etype, filetype, datarep, info)

IN	fh	拥有文件窗口的文件联接器
IN	disp	文件窗口在文件中的起始位置(字节为单位)
IN	ctype	I/O访问的数据单元类型
IN	filetype	定义文件窗口的数据单元类型
IN	datarep	数据表示格式
IN	info	提示信息

```

C      MPI_File_Set_View(MPI_File fh, MPI_Offset disp, MPI_Datatype ctype,
                      MPI_Datatype filetype, char * datarep, MPI_Info info)
Fortran MPI_FILE_SET_VIEW(FH,DISP,ETYPE,FILETYPE,DATAREP,INFO,IERROR)
        INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
        INTEGER(KIND=MPI_OFFSET_KIND) DISP, DATAREP
    
```

函数 MPI_FILE_SET_VIEW 在 fh 联接的文件中, 为进程定义一个文件窗口, 包含该进程将要访问的所有数据单元。

图 8.3 给出了一个文件窗口示意图。其中, 文件包含多个整数(方格表示), 从文件的第 disp 个字节开始, 每隔 extent 个整数, 挑选起始的 2 个整数组成文件窗口, 图中用灰色标示。基于该文件窗口, 执行 MPI 程序的进程就可以访问这些用灰色标示的非连续整型数据单元了。

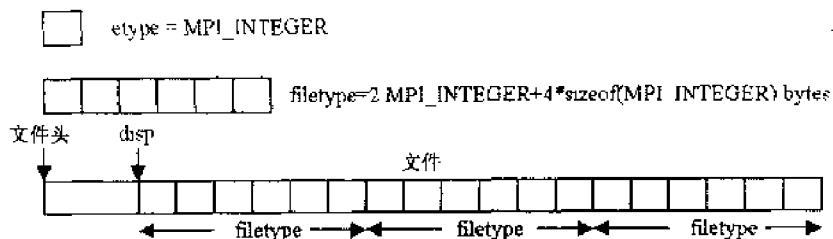


图 8.3 文件窗口示意图, 其中, 文件窗口用灰色标示

定义一个文件窗口需要三个参数: 第一个是该文件窗口的在文件中的起始位置 `disp`(字节为单位); 第二个是 I/O 访问的数据单元类型 `ctype`, 它可以是任意的基本数据类型或第四章介绍的自定义数据类型; 第三个是定义文件窗口的数据单元类型 `filetype`。假设 `filetype` 的域为 `extent`, 则文件从 `disp` 开始, 被分割成连续的大小为 `extent` 的多个数据块, 在每个数据块内部, 对进程可见的部分仅是 `filetype` 说明的那些类型为 `etype` 的数据单元, 而一个文件窗口可由所有这些对进程可见的类型为 `etype` 的数据单元组成。

由此可知, 定义文件窗口的关键在于定义数据类型 `filetype`, 它可由第四章介绍的自定义数据类型函数来定义。但是, 为了简化并行程序设计, MPI 系统提供函数 `MPI_Type_create_resized` 为一个已经存在的数据类型 `oldtype` 扩展下界和域, 并由此定义一个新的数据类型, 但在该数据类型内部, 只有 `oldtype` 说明的那个数据单元才能被进程访问。具体定义如下:

```

/* 数据类型扩展函数, 扩展某个数据类型的下界和域。

MPI_TYPE_CREATE_RESIZED (oldtype, lb, extent, filetype)
    IN      oldtype      进程可访问的数据单元类型
    IN      lb           filetype 的下界(字节为单位)
    IN      extent       数据类型 filetype 的域 (字节为单位)
    OUT     filetype     新的数据类型

C      MPI_Type_create_resized (MPI_Datatype oldtype, MPI_Aint lb,
                               MPI_Aint extent, MPI_Datatype filetype)

Fortran MPI_TYPE_CREATE_RESIZED (OLDTYPE, LB, EXTENT, FILETYPE, IERROR)
        INTEGER OLDTYPE, FILETYPE, IERROR
        INTEGER(KIND=MPI_OFFSET_KIND) LB, EXTENT

```

在图 8.3 中, 数据类型 filetype 由两个整数 (`etype = MPI_INTEGER`) 外加 $4 * \text{sizeof}(\text{MPI_INTEGER})$ 个字节组成, 且其中只有位于前面的两个整数才被包含在文件窗口中。由此, 从进程的视角出发, 文件将从位置 `disp` 开始, 由多个连续的类型为 filetype 的数据单元组成, 而在每个数据单元中, 对进程可见的部分仅是位于初始位置的类型为 oldtype 的单个数据单元, 特别地, 在图 8.3 中, oldtype 由两个整数组成, 而参数 `lb = 0`, `extent = 6 * sizeof(MPI_INTEGER)`。函数 `MPI_TYPE_CREATE_RESIZED` 的具体应用, 请读者参考例 8.6。

如果数据类型 filetype 由第四章介绍的函数来定义, 则函数 `MPI_SET_VIEW` 定义的文件窗口, 就由该数据类型 filetype 包含的所有类型为 `etype` 的数据单元组成, 敬请读者注意, 并参考例 8.7。

在函数 `MPI_FILE_SET_VIEW` 中, 参数 `datarep` 表示文件窗口中数据单元的表示格式。MPI 系统提供三种格式:(1) `datarep = "native"`, 表示文件中数据单元表示格式就是它在内存中的表示格式, 该格式只在相同结构的并行机上被支持;(2) `datarep = "internal"`, 表示文件中数据单元表示格式是 MPI 系统内部定义的格式, 该格式在所有支持同一个 MPI 系统的并行机上, 都是可以被访问的;(3) `datarep = "external32"`, 表示文件中数据单元表示格式是 32 位 IEEE big-endian 标准格式, 该格式在所有支持 MPI 系统的并行机上, 都是可以被访问的。

函数 `MPI_FILE_SET_VIEW` 和函数 `MPI_TYPE_CREATE_RESIZED` 属于局部函数, 各个进程可以创建各自独立的文件窗口。文件窗口创建后, 后续的文件访问均局限在该文件窗口内部。

例 8.6 非连续访问并行 I/O

对应于图 8.3, 进程将 1000 个整型数按如下的方式写入文件中: 跳过文件起始的 5 个整数, 写入 2 个整数, 再跳过 4 个整数, 写入 2 个整数, 再跳过 4 个整数, ……, 依次类推下去, 直到写入第 1000 个整数为止。

```

!
INTEGER * 8 LB, EXTENT, DISP

```

```

INTEGER ETYPE, FILETYPE, CONTIG, FH, BUF(1000)
!
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, "/pfs/datafile",
                   MPI_MODE_CREATE + MPI_MODE_RDWR,
                   MPI_INFO_NULL, FH, IERR)
CALL MPI_TYPE_CONTIGUOUS(2, MPI_INTEGER, CONTIG, IERR)
! 定义由连续存储的两个整数组成的数据类型。
LB = 0           ! 数据类型下界。
EXTENT = 6 * SIZEOF(MPI_INTEGER)      ! 数据类型的域。
!
! 定义数据类型, 它由 6 个整数组成, 其中, 位于初始位置的两个整数将被
! 包含在文件窗口中。
CALL MPI_TYPE_CREATE_RESIZED(CONTIG, LB, EXTENT, FILETYPE, IERR)
CALL MPI_TYPE_COMMIT(FILETYPE, IERR)    ! 提交定义的数据类型。
!
DISP = 5 * SIZEOF(MPI_INTEGER)          ! 文件窗口在文件中的初始位置。
ETYPE = MPI_INTEGER
!
! 定义文件窗口, 后续的文件访问将被局限在该文件窗口中。
CALL MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, "native",
                       MPI_INFO_NULL, IERR)
! 将整数按文件窗口定义的位置, 写入文件中。
CALL MPI_FILE_WRITE(FH, BUF, 1000, MPI_INTEGER,
                     MPI_STATUS_IGNORE, IERR)
CALL MPI_FILE_CLOSE(FH, IERR)

```

8.4.4 聚合并行 I/O

在同一时刻, 执行 MPI 程序的多个进程同时访问同一个资源文件时, 有两种情形将影响并行 I/O 性能: 第一, 进程之间竞争同一个 I/O 设备, 引起资源访问冲突; 第二, 独立访问的多个进程将导致多次小数据量的 I/O 访问。为此, MPI 系统提供了另一类函数, 称之为**聚合 I/O 函数**, 可用于协调执行 I/O 访问的多个进程之间的关系, 并将多个进程要求访问的数据汇集到一起, 一次访问文件, 提高并行 I/O 性能。这些函数类似于第五章介绍的聚合通信函数, 要求属于同一通信器的所有进程的共同参与。

函数 MPI_FILE_READ_ALL 和函数 MPI_FILE_WRITE_ALL 可用于分别完成聚合 I/O 读和聚合 I/O 写操作, 具体定义如下:

/* MPI 聚合 I/O 读函数, 各进程从同一个文件中读取数据。

	MPI_FILE_READ_ALL(fh, buf, count, datatype, status)	
IN	fh	文件联接器
IN	buf	存储数据的缓存区初始地址
IN	count	读取的数据单元个数
IN	datatype	读取的数据单元类型

```

    INOUT      status      数据读取状态信息
C     MPI_File_read_all (MPI_File fh, void * buf, int count, MPI_Datatype datatype,
                       MPI_Status * status)
Fortran MPI_FILE_READ_ALL (FH,BUF,COUNT,DATATYPE,STATUS,IERROR)
      INTEGER      FH,BUF,COUNT,DATATYPE,STATUS,IERROR

/* MPI 聚合 I/O 写函数, 各进程将各自数据写入同一个文件中。

    MPI_FILE_WRITE_ALL (fh, buf, count, datatype, status)
      IN      fh      文件联接器
      IN      buf      待写入文件的数据缓存区初始地址
      IN      count    待写入文件的数据单元个数
      IN      datatype  待写入文件的数据单元类型
      INOUT     status   数据写入状态信息
C     MPI_File_write_all (MPI_File fh, void * buf, int count,
                        MPI_Datatype datatype, MPI_Status * status)
Fortran MPI_FILE_WRITE_ALL (FH,BUF,COUNT,DATATYPE,
                           STATUS,IERROR)
      INTEGER      FH,BUF,COUNT,DATATYPE,STATUS,IERROR

```

聚合 I/O(读/写)函数的各参数含义与前面介绍的并行 I/O(读/写)函数完全一致,这里不再讨论。

例 8.7 聚合 I/O 函数应用示例

```

PARAMETER (FILESIZE = 1048576, INTS_PER_BLK = 16)
INTEGER  BUF(FILESIZE), RANK, NPROCS, NINTS, BUFSIZE
INTEGER  FILETYPE, FH
!
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROCS, IERR)
BUFSIZE = FILESIZE/NPROCS           ! 各进程将读取的数据大小。
NINTS = BUFSIZE/SIZEOF(MPI_INTEGER) ! 各进程将读取的整数个数。
!
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, "/ufs/datafile",
                  MPI_MODE_RDONLY, MPI_INFO_NULL, FH, IERR)
! 定义数据类型, 由 NINTS/INTS_PER_BLK 块组成, 每块含 INTS_PER_BLK 个整
! 数, 块与块之间的间距为 INTS_PER_BLK * NPROCS。该数据类型代表了该进程将
! 要从文件中访问的所有数据单元。
CALL MPI_TYPE_VECTOR(NINTS/INTS_PER_BLK, INTS_PER_BLK,
                      INTS_PER_BLK * NPROCS, MPI_INTEGER, FILETYPE, IERR)
CALL MPI_TYPE_COMMIT(FILETYPE, IERR)
!
CALL MPI_FILE_SET_VIEW(FH, INTS_PER_BLK * SIZEOF(MPI_INTEGER) * RANK,

```

```

MPI_INTEGER, FILETYPE, "native", MPI_INFO_NULL, IERR)
! 各进程定义各自的文件窗口。
CALL MPI_FILE_READ_ALL(FH, BUF, NINTS, MPI_INTEGER,
MPI_STATUS_IGNORE, IERR)
! 各进程从各自文件窗口中读取 NINTS 个整数。
CALL MPI_FILE_CLOSE(FH, IERR)
CALL MPI_TYPE_FREE(FILETYPE, IERR)

```

在上例中,函数 MPI_FILE_SET_VIEW 中的数据类型参数 FILETYPE 由第四章介绍的函数 MPI_TYPE_VECTOR 来定义,表示该文件窗口包含的数据单元就是该数据类型参数 FILETYPE 包含的所有整数,敬请读者注意。

8.4.5 分布存储数组的并行 I/O

在 MPI 程序中,一个数组 (array) 通常被分割成多个独立的子块,并存储在各个进程的内存空间中,我们称之为分布存储数组。在实际应用中,这些数组通常对应于流场数据,例如速度、压力、温度、密度等等,它们是要求执行并行 I/O 的主要对象。在 MPI 并行程序中,用户通常希望分布存储数组的每个元素按某种特殊的序号(例如自然序)依次存储在文件中,得到与串行程序一致的文件格式,以方便数据的后处理。此时,如果采用前面介绍的并行 I/O 函数来完成这项工作,则要求用户收集分布在各个进程的数据,并将它们重新排序,然后写入文件中,使得文件格式与串行程序一致。这种流场数据的收集工作非常繁琐,且容易出错。

为了简化对分布存储数组的访问和并行程序的设计,MPI 系统提供了三个函数 MPI_TYPE_CREATE_DARRAY, MPI_TYPE_CREATE_SUBARRAY 和 MPI_TYPE_INDEXED_BLOCK,可用于分别定义四类特殊的分布存储数组数据类型:分布数组 (darray)、子数组 (subarray)、影像数组 (local subarray with ghost area) 和非规则分布数组 (irregularly distributed array),它们可满足应用程序对数组的 I/O 要求。

1. 分布数组

在图 8.4 中,我们给出了一个 m 行 n 列的二维数组,它被等分成 2×3 的 6 个子块,并分布在具有 2×3 拓扑结构的 6 个进程中。具体地,该数据结构可用如下函数来定义:

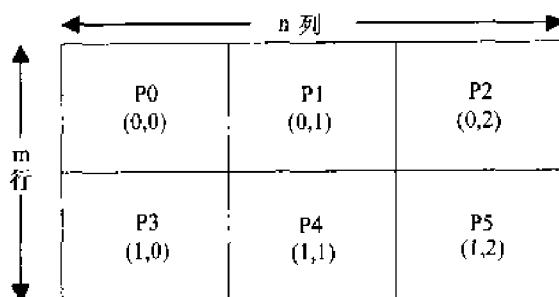


图 8.4 分布在 2×3 进程拓扑结构上的 2 维分布数组

```

/* MPI 分布数组定义函数, 定义一个分布存储数组数据类型。

MPI_TYPE_CREATE_DARRAY (size, rank, ndim, array_of_gsizes,
                        array_of_distrib, array_of_dargs, array_of_psizes, order,
                        oldtype, newtype)

IN      size          数组分布的进程个数
IN      rank          调用该函数的进程序号
IN      ndim          数组的维数
IN      array_of_gsizes 一维数组, 含数组各维数据单元个数
IN      array_of_distrib 一维数组, 含数组各维数据单元的分布模式
IN      array_of_dargs 一维数组, 含数组各维的分布模式的参数
IN      array_of_psizes 一维数组, 含数组各维分布的进程个数
IN      order          数据单元在文件中的存储序
IN      oldtype        数据单元类型
OUT     newtype        分布存储数组数据类型

C      MPI_Type_create_darray (int size, int rank, int ndim, int * array_of_gsizes,
                            int * array_of_distrib, int * array_of_dargs, int * array_of_psizes,
                            int order, MPI_Datatype oldtype, MPI_Datatype * newtype)

Fortran MPI_TYPE_CREATE_DARRAY (SIZE, RANK, NDIM,
                                ARRAY_OF_GSIZES, ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS,
                                ARRAY_OF_PSIZES, ORDER, OLDTYPE, NEWTYPE, IERROR)
      INTEGER SIZE, RANK, NDIM, ARRAY_OF_GSIZES, ARRAY_OF_DISTRIBS,
              ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER, OLDTYPE,
              NEWTYPE, IERROR

```

函数 MPI_TYPE_CREATE_DARRAY 属于全局函数, 需要通信器所有进程的共同参与。其中, 第 1 个参数 size 是数组分布的进程个数, 在上图中, size = 6; 第 2 个参数 rank 是局部数据块所属的进程序号, 也就是调用该函数的进程序号; 第 3 个参数是数组的维数, 上图中 ndim = 2; 第 4 个参数是一个长度为 ndim 的一维数组, 各元素记录各维的全局数据单元个数, 上图中 gsizes(1) = m, gsizes(2) = n; 第 5, 6 个参数均是长度为 ndim 的一维数组, 各元素分别记录数据单元在各维的分布模式和分布模式的参数, MPI 系统提供了三种分布模式: 块分布模式 (MPI_DISTRIBUTE_BLOCK, 上图给出的就是块分布模式, 此时第 6 个参数提供缺省值 MPI_DISTRIBUTE_DFLT_DARG)、循环分布模式 (MPI_DISTRIBUTE_CYCLIC, 各元素按序号依次循环分布存储在各进程中, 第 6 个参数提供缺省值 MPI_DISTRIBUTE_DFLT_DARG) 和块循环分布模式 (MPI_DISTRIBUTE_CYCLIC, 各元素按块循环分布在各进程中, 其中每块数据单元的个数由第 6 个参数提供); 第 7 个参数也为长度为 ndim 的数组, 各元素包含各维进程的个数, 上图中 psizes(1) = 2, psizes(2) = 3; 第 8 个参数提供数据单元在内存中的排列次序, 其中 order = MPI_ORDER_C 表示行序 (对应 C 语言), order = MPI_ORDER_F 表示列序 (对应 Fortran 语言); 第 9 个参数表示待分布的数据单元的类型; 第 10 个参数为该函数定义的分布存储数组数据类型, 可直接应用于 MPI 并行 I/O 函数中。例如, 上图中的数据

分布可用以下的程序来具体实现。

例 8.8 分布数组定义函数应用示例

```
INTEGER m, n
INTEGER gsize(2), distribs(2), dargs(2), psizes(2), rank, filetype, fh
INTEGER local_array_size, num_local_rows, num_local_cols
REAL local_array(10000)
!
m = 200
n = 300
!
gsizes(1) = m
gsizes(2) = n
distribs(1) = MPI_DISTRIBUTE_BLOCK
distribs(2) = MPI_DISTRIBUTE_BLOCK
dargs(1) = MPI_DISTRIBUTE_DFLT_DARG
dargs(2) = MPI_DISTRIBUTE_DFLT_DARG
psizes(1) = 2
psizes(2) = 3
num_local_rows = m / psizes(1)
num_local_cols = n / psizes(2)
!
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
! 创建一个分布存储数组数据类型。
call MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs, psizes, MPI_ORDER_F,
                           MPI_REAL, filetype, ierr)
call MPI_Type_commit(filetype, ierr)
call MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_CREATE +
                  MPI_MODE_WRONLY, MPI_INFO_NULL, fh, ierr)
call MPI_File_set_view(fh, 0, MPI_REAL, filetype, "native", MPI_INFO_NULL, ierr)
! 确定每个进程将写入的数据单元个数。
local_array_size = num_local_rows * num_local_cols
! 各个进程将局部数组 local_array 中 local_array_size 个数据单元写入文件中。
call MPI_File_write_all(fh, local_array, local_array_size, MPI_REAL, status, ierr)
call MPI_File_close(fh, ierr)
```

函数 MPI_TYPE_CREATE_DARRAY 创建分布存储数组时, 存在一个限制, 就是要求沿各维的数据单元个数必须能被该维的进程个数整除, 且分布存储数组的维数必须与进程拓扑结构的维数一致, 敬请读者注意。

2. 子数组

为了克服函数 MPI_TYPE_CREATE_DARRAY 的限制, MPI 系统提供了子数组

定义函数 MPI_TYPE_CREATE_SUBARRAY, 它可用于按指定的方式, 为每个进程定义一个分布存储数组的子块, 称之为子数组 (subarray), 包含该进程将要执行 I/O 操作的所有局部数据单元。所有这些子数组构成 MPI 程序的分布存储数组, 称之为全局数组 (global array)。具体地, 该函数定义如下:

```
/* MPI 子数组定义函数, 定义一个子数组数据类型。

MPI_TYPE_CREATE_SUBARRAY (ndim, array_of_sizes, array_of_subsizes,
                           array_of_starts, order, oldtype, newtype)

IN      ndim          全局数组和子数组的维数
IN      array_of_sizes 一维数组, 含全局数组各维包含的数据单元个数
IN      array_of_subsizes 一维数组, 含子数组各维包含的数据单元个数
IN      array_of_starts 一维数组, 含子数组在全局数据各维的起始位置
IN      order          数据单元在文件中的存储序
IN      oldtype         文件访问的数据单元类型
OUT     newtype         子数组数据类型

C      MPI_Type_create_subarray (int ndim, int * array_of_sizes, int * array_of_subsizes,
                               int * array_of_starts, int order, MPI_Datatype oldtype,
                               MPI_Datatype * newtype)

Fortran MPI_TYPE_CREATE_SUBARRAY (NDIM, ARRAY_OF_SIZES,
                                  ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER,
                                  OLDTYPE, NEWTYPE, IERROR)
INTEGER NDIM, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
        ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR
```

函数 MPI_TYPE_CREATE_SUBARRAY 的第 1 个参数 `ndim` 是数组的维数; 第 2 个参数是长度为 `ndim` 的一维数组, 记录全局数组各维的数据单元个数; 第 3 参数是长度为 `ndim` 的一维数组, 记录子数组各维包含的数据单元个数; 第 4 个参数是长度为 `ndim` 的数组, 记录子数组在全局数组中的起始位置, 特别在计算这些起始位置时, 无论对 C 语言, 还是对 Fortran 语言, 全局数组的下标均从 0 开始, 敬请用户注意; 第 5 个参数提供数据单元在内存中的排列次序, 其中 MPI_ORDER_C 表示行序 (对应 C 语言), MPI_ORDER_F 表示列序 (对应 Fortran 语言); 第 6 个参数表示数组数据单元的类型; 第 7 个参数为该函数定义的子数组数据类型, 可直接应用于 MPI 并行 I/O 函数中。

例 8.9 子数组定义函数应用示例

```
! INTEGER m, n, gsize(2), psizes(2), filetype, fh
INTEGER dims(2), periods(2), lsizes(2), coords(2), start_indices(2)
!
m = 200
n = 300
!
gsize(1) = m
```

```

gsizes(2) = n
psizes(1) = 2
psizes(2) = 3
lsizes(1) = m / psizes(1)
lsizes(2) = n / psizes(2)
dims(1) = 2
dims(2) = 3
periods(1) = .true.
periods(2) = .true.
call MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, comm, ierr)
call MPI_Comm_rank(comm, rank, ierr)
call MPI_Cart_coords(comm, rank, 2, coords, ierr)
start_indices(1) = coords(1) * lsizes(1)
start_indices(2) = coords(2) * lsizes(2)
call MPI_Type_create_subarray(2, gsizes, lsizes, start_indices, MPI_ORDER_F, MPI_REAL,
                           filetype, ierr) ! 定义分布存储的子数组。
call MPI_Type_commit(filetype, ierr)
!
call MPI_File_open(MPI_COMM_WORLD, "/pts/datafile", MPI_MODE_CREATE +
                  MPI_MODE_WRONLY, MPI_INFO_NULL, fh, ierr)
call MPI_File_set_view(fh, 0, MPI_REAL, filetype, "native", MPI_INFO_NULL, ierr)
! 定义文件窗口, 包含分布存储的子数组。
local_array_size = lsizes(1) * lsizes(2)
call MPI_File_write_all(fh, local_array, local_array_size, MPI_REAL, status, ierr)
call MPI_File_close(fh, ierr)

```

3. 影像数组

在实际应用中, 沿子数组的各个方向, 通常配备有几行或几列增加的数据单元, 用于存储相邻进程在该位置的数据单元的值, 我们称之为子数组的影像区 (ghost area), 并称增加的行或列的个数为影像区的宽度, 带影像区的子数组为影像子数组。例如, 在图 8.5

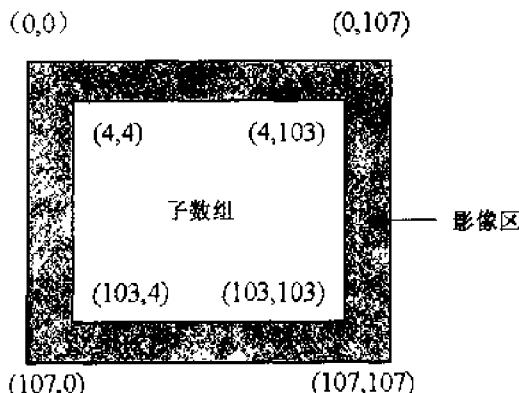


图 8.5 影像子数组示意图

中, 大小为(100, 100)的子数组附带有一个宽度为4的影像区, 形成一个大小为(108, 108)的影像子数组。

在并行I/O的执行过程中, 用户通常不希望将影像子数组影像区中的数据单元写入文件, 因为对应位置上的数据单元将会被相邻进程写入文件。其实, 在子数组的基础上, 该项要求可通过8.4.4节中介绍的函数, 定义新的数据类型来实现。下例给出了图8.4的MPI程序实现。

例8.10 子数组定义函数应用示例

```
! 变量说明请参考例8.9
INTEGER memsizes(2)
!
gsizes(1) = m
gsizes(2) = n
psizes(1) = 2
psizes(2) = 3
lsizes(1) = m/psizes(1)
lsizes(2) = n/psizes(2)
dims(1) = 2
dims(2) = 3
periods(1) = periods(2) = .true.
call MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, comm, ierr)
call MPI_Comm_rank(comm, rank, ierr)
call MPI_Cart_coords(comm, rank, 2, coords, ierr)
start_indices(1) = coords(1) * lsizes(1)
start_indices(2) = coords(2) * lsizes(2)
call MPI_Type_create_subarray(2, gsizes, lsizes, start_indices, MPI_ORDER_F, MPI_REAL,
                           filetype, ierr)      ! 定义分布存储的子数组。
call MPI_Type_commit(filetype, ierr)
!
call MPI_File_open(MPI_COMM_WORLD, "/afs/datafile", MPI_MODE_CREATE +
                  MPI_MODE_WRONLY, MPI_INFO_NULL, fh, ierr)
call MPI_File_set_view(fh, 0, MPI_REAL, filetype, "native", MPI_INFO_NULL, ierr)
! 定义文件窗口, 包含分布存储的子数组。
!
! 定义影像子数组数据类型, 描述影像子数组在内存中的分布。
memsizes(1) = lsizes(1) + 8      ! 影像子数组各维的数据单元个数。
memsizes(2) = lsizes(2) + 8
start_indices(1) = 4            ! 子数组在影像子数组各维的起始位置。
start_indices(2) = 4
! 定义新的数据类型, 它描述局部存储的子数组在影像子数组中的分布。
call MPI_Type_create_subarray(2, memsizes, lsizes, start_indices, MPI_ORDER_F,
                             MPI_REAL, memtype, ierr)
```

```

call MPI_Type_commit(memtype, ierr)
call MPI_File_write_all(fh, local_array, 1, memtype, status, ierr)
call MPI_File_close(fh, ierr)

```

4. 非规则的分布存储数组

在前面介绍的有关分布数组、子数组和影像子数组的函数中,只要给定一个进程的序号和该进程包含的数据单元在子数组中的位置,我们就可以用数学公式计算出该数据单元在全局数组中的位置。也就是说,数组的分布存储是规则的,我们称之为**规则的分布存储数组**。反之,对某类分布存储数组,如果我们无法找到这样严格的数学公式,则称之为**非规则的分布存储数组**,这是本小节将要讨论的重点。由一个数组中各元素在该数组中的排列序号组成的数组称为该数组的映射数组。

非规则的分布存储数组通过各进程提供的**局部映射数组**来实现。具体地,假设以长度固定的数据块为单位,各进程将其局部数据单元分割成多个数据块,则映射数组的各个元素将依次给出进程各局部数据块在全局数组中所处的数据块的位置。于是,按照各进程提供的局部映射数组,我们可以将各进程提供的所有数据单元形成一个全局数组。显然,各进程提供的局部映射数组必须是互不相交的,即不存在两个局部映射数组的元素,它们将数据块映射到全局数组的同一个位置。

在图 8.6 中,对应于 3 个进程,分别定义了 3 个局部映射数组,从这些映射数组出发,我们就可以定义一种新的数据类型,使得并行输出的文件与串行程序一致。

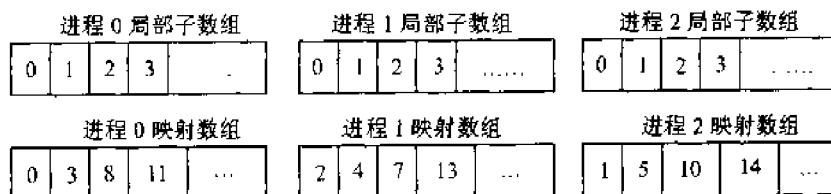


图 8.6 映射数组示意图

函数 `MPI_TYPE_CREATE_INDEXED_BLOCK` 可用于定义一个非规则的分布存储数组数据类型,具体定义如下:

`/* MPI 非规则分布存储数组定义函数, 定义一个非规则的分布存储数组数据类型。`

```

MPI_TYPE_CREATE_INDEXED_BLOCK (count, blocklength,
                               array_of_displacements, oldtype, newtype)

IN      count          各进程包含的数据块的个数
IN      blocklength    每个数据块包含的数据单元个数
IN      array_of_displacements 长度为 count 的局部映射数组
IN      oldtype        数据单元类型
OUT     newtype        非规则的分布存储数组数据类型

C      MPI_Type_create_indexed_block (int count, int blocklength,
                                     int * array_of_displacements, MPI_Datatype oldtype, MPI_Datatype * newtype)

```

```

Fortran MPI_TYPE_CREATE_INDEXED_BLOCK (COUNT,BLOCKLENGTH,
           ARRAY_OF_DISPLACEMENTS,OLDTYPE,NEWTYPE,IERROR)
      INTEGER COUNT,BLOCKLENGTH,ARRAY_OF_DISPLACEMENTS,
             OLDTYPE,NEWTYPE,IERROR

```

在函数 MPI_TYPE_CREATE_INDEXED_BLOCK 中, 每个局部数组被分解为 count 个数据块, 每个数据块包含相同个数 (blocklength) 的数据单元, 这些数据单元在全局数组中是连续存放的, 其类型为 oldtype。但是, 这些数据块在全局数组中是无规则存储的, 数组 array_of_displacements 的各元素, 分别描述各数据块在全局数组中的存储位置, 且各元素的值必须是单调递增的, 互不相同。最后, 函数 MPI_TYPE_CREATE_INDEXED_BLOCK 定义一个非规则的分布存储数组数据类型 newtype。

例 8.11 非规则的分布存储数组定义函数应用示例

```

integer bufsize
double precision buf(bufsize)
integer map(bufsize), fh, filetype, status(MPI_STATUS_SIZE)
integer(kind=MPI_OFFSET_KIND) disp
integer i, ierr
!
call MPI_File_open(MPI_COMM_WORLD,"/afs/datafile",MPI_MODE_CREATE|
                  MPI_MODE_WRONLY,MPI_INFO_NULL,fh,ierr)
! 由映射数组定义非规则的分布存储数组数据类型。
call MPI_Type_create_indexed_block(bufsize,1,map,MPI_DOUBLE_PRECISION,
                                    filetype,ierr)
call MPI_Type_commit(filetype,ierr)
disp = 0
! 定义文件窗口, 包含非规则的分布存储数组。
call MPI_Type_set_view(fh,disp,MPI_DOUBLE_PRECISION,filetype,"native",&
                      MPI_INFO_NULL,ierr)
call MPI_File_write_all(fh,buf,bufsize,MPI_DOUBLE_PRECISION,status,ierr)
call MPI_File_close(fh,ierr)

```

8.5 非阻塞并行 I/O 与分裂聚合并行 I/O

前面各节介绍的并行 I/O(读/写)函数都是阻塞式的, 因为只有当这些函数执行的 I/O 访问完成后, 函数才返回。其实, MPI 系统也支持非阻塞并行 I/O(读/写)操作, 具体由函数 MPI_FILE_IREAD 和函数 MPI_FILE_IWRITE 来实现。

比较阻塞式并行读函数 MPI_FILE_READ_AT 和并行写函数 MPI_FILE_WRITE_AT, 非阻塞并行读函数 MPI_FILE_IREAD 和并行写函数 MPI_FILE_IWRITE 在参数序列的末尾, 追加一个参数, 当函数返回时, 该参数包含联接该次 I/O 操作的通信请求, 用户可以通过调用第三章介绍的各类通信请求完成函数, 来确保该通信请

求所联接的并行 I/O 操作的正确完成。这样，我们就可以重叠 I/O 与计算，进一步提高应用程序的并行性能。

通常情形下，非阻塞并行 I/O 函数可按如下格式具体应用：

```
integer request
call MPI_File_iread(fh, offset, buf, count, datatype, request, ierr)
    ! 提交非阻塞读操作。
do i=1,1000
    ! 执行局部计算。
enddo
call MPI_Wait(request, status, ierr)      ! 等待完成并行读操作。
```

或者：

```
integer request
call MPI_File_iwrite(fh, offset, buf, count, datatype, request, ierr)
    ! 提交非阻塞写操作。
do i=1,1000
    ! 执行局部计算。
enddo
call MPI_Wait(request, status, ierr)      ! 等待完成并行写操作。
```

同样，MPI 系统也支持分裂聚合并行 I/O 功能，它将聚合并行 I/O 函数分裂成两个阶段，其中一个阶段提交聚合并行 I/O 操作，另一个阶段完成已提交的聚合并行 I/O 操作。但是，与非阻塞并行 I/O 不同的是，MPI 系统规定，在同一个时刻，应用程序只能提交一个聚合并行 I/O 操作，如果应用程序要求提交另一个聚合并行 I/O 操作，则必须等待上一个已经提交的聚合并行 I/O 操作完成后，才能进行，敬请读者注意。

MPI 系统提供函数 MPI_FILE_READ_ALL_BEGIN 和函数 MPI_FILE_READ_ALL_END 辅助应用程序提交和完成一个聚合并行读操作，提供函数 MPI_FILE_WRITE_ALL_BEGIN 和函数 MPI_FILE_WRITE_ALL_END 辅助应用程序提交和完成一个聚合并行写操作。由于在同一个时刻，应用程序中只可能存在一个未完成的聚合并行 I/O 操作，因此，我们没必要依靠通信请求来联接聚合并行 I/O 操作的提交与完成。具体地，除了不包含参数 status，聚合并行 I/O 提交函数的参数序列和 8.4.4 节中介紹的聚合并行 I/O 函数的参数序列一致，而聚合并行 I/O 完成函数定义如下：

/* MPI 聚合 I/O 完成函数，完成已经提交的聚合 I/O 操作。

```
MPI_FILE_WRITE_ALL_END(fh, buf, status)
    IN          fh          文件联接器
    IN          buf         读取/写入文件的数据缓存区初始地址
    OUT         status       数据读取/写入返回状态信息
C     MPI_File_write_all_end(MPI_File fh, void * buf, MPI_Status * status)
Fortran MPI_FILE_WRITE_ALL_END(FH,BUF,STATUS,IERROR)
        INTEGER FH,BUF,STATUS,IERROR
```

通常情形下，分裂聚合并行 I/O 函数可以按如下格式应用：

```

call MPI_File_write_all_begin(fh, buf, count, datatype, ierr)
do i=1,1000
    ! 局部计算。
enddo
call MPI_File_write_all_end(fh, buf, status, ierr)

```

8.6 共享文件指针

在前面各节介绍的 MPI 并行 I/O 函数中, 每个进程均拥有各自独立的文件指针, 其中某个进程文件指针的变化将不会改变其他进程的文件指针。其实, MPI 系统也提供共享文件指针的服务, 即所有进程共享一个文件指针, 对同一个文件执行 I/O 操作。具体地, MPI 系统提供函数 MPI_FILE_READ_SHARED 和函数 MPI_FILE_WRITE_SHARED 支持各个进程从共享文件指针的当前位置读/写数据, 而各个进程可以调用函数 MPI_FILE_SEEK_SHARED 来显式移动共享文件指针。三个函数的参数序列和含义与前面各节介绍的对应函数 MPI_FILE_READ, MPI_FILE_WRITE 和 MPI_FILE_SEEK 的参数序列和含义完全一致。

此外, 在调用共享文件指针的并行 I/O 函数之前, 要求所有共享该文件指针的进程提供相同的文件窗口, 敬请读者注意。

例 8.12 共享文件指针应用示例

```

integer buf(1000),fh
call MPI_File_open(MPI_COMM_WORLD,"/pfs/datafile",MPI_MODE_CREATE +
                  MPI_MODE_WRONLY,MPI_INFO_NULL,fh,ierr)
call MPI_File_write_shared(buf,1000,ierr)
    ! 从当前共享文件指针指定的位置, 各进程将 1000 个整数写入文件中。
    ! 例如, 假设总共有 4 个进程, 其中进程 0 首先写入 1000 个整数,
    ! 进程 4 写入 1000 个整数, 然后进程 2 写入 1000 个整数, 最后进程 1
    ! 写入 1000 个整数。这样, 文件将包含 4000 个整数,
    ! 依次为进程 0、进程 4、进程 2 和进程 1 写入的。
call MPI_File_close(fh,ierr)

```

共享文件指针的聚合并行 I/O 函数为 MPI_FILE_READ_ORDERED 和 MPI_FILE_WRITE_ORDERED, 这两个函数的参数序列和含义与前节介绍的 MPI_FILE_READ, MPI_FILE_WRITE 完全一致, 它们能保证各个进程按进程序号依次读/写同一个文件。

共享文件指针的非阻塞并行 I/O 函数为 MPI_FILE_IREAD_SHARED 和 MPI_FILE_IWRITE_SHARED, 而共享文件指针的分阶段聚合并行 I/O 函数为 MPI_FILE_READ_ORDERED_BEGIN 和 MPI_FILE_READ_ORDERED_END。这些函数的具体应用类似于 8.5 节中介绍的基于独立文件指针的对应函数的应用。

8.7 提示信息

MPI 系统提供一种特殊的数据类型: 提示信息 (MPI_Info), 应用程序通过它, 可以向 MPI 系统传递具体并行机的参数, 辅助进一步优化 MPI 系统, 提高 MPI 程序的并行性能。

MPI 程序可以通过函数 MPI_INFO_CREATE 和函数 MPI_INFO_FREE 创建和释放一个提示信息, 在 C 语言中, 其数据类型为 MPI_Info (它们是 MPI 系统定义的一种特殊数据类型), 在 Fortran 语言中, 其数据类型为 INTEGER。给定一个提示信息 MPI_Info, MPI 程序可以通过函数 MPI_SET_INFO 为它设置属性 (key, value), 其中 key 表示该属性的关键词, 而 value 是对应于该关键词的属性值, 特别地, key 和 value 均必须说明为字符串, 且长度至多为 MPI 系统定义的常数 MPI_MAX_INFO_KEY 和 MPI_MAX_INFO_VAL。反之, 给定一个提示信息 MPI_Info, MPI 程序可以通过函数 MPI_INFO_GET, 查询附属于该提示信息关键词的某个具体属性的值, 通过函数 MPI_INFO_GET_NKEYS 查询附属于该提示信息的属性个数, 通过函数 MPI_INFO_GET_NTHKEY 查询附属于该提示信息的某个属性的关键词。具体地, 这些函数定义如下:

/* MPI 提示信息创建函数, 创建一个提示信息。

MPI_INFO_CREATE (info)		
OUT	info	被创建的 MPI 提示信息
C	MPI_Info_create (MPI_Info * info)	
Fortran	MPI_INFO_CREATE (INFO, IERROR)	
	INTEGER	INFO, IERROR

/* MPI 提示信息释放函数, 释放一个已经被创建的提示信息。

MPI_INFO_FREE (info)		
IN	info	将被释放的 MPI 提示信息
C	MPI_Info_free (MPI_Info info)	
Fortran	MPI_INFO_FREE (INFO, IERROR)	
	INTEGER	INFO, IERROR

/* MPI 提示信息属性设置函数, 为给定的 MPI 提示信息设置一个属性。

MPI_INFO_SET (info, key, value)		
IN	info	给定的 MPI 提示信息
IN	key	附属属性的关键词
IN	value	附属属性的值
C	MPI_Info_set (MPI_Info info, char * key, char * value)	
Fortran	MPI_INFO_SET (INFO, KEY, VALUE, IERROR)	
	INTEGER	INFO, IERROR
	CHARACTER (*)	KEY, VALUE

/* MPI 提示信息属性查询函数, 基于关键词, 查询 MPI 提示信息的属性。

MPI_INFO_GET (info, key, valuelen, value, flag)

IN	info	给定的 MPI 提示信息
IN	key	附属属性的关键词
IN	valuelen	附属属性的值的长度
OUT	value	附属属性的值
OUT	flag	若该属性存在, 则 flag = 1; 否则 flag = 0

C MPI_Info_get (MPI_Info info, char * key, int valuelen, char * value, int * flag)

Fortran MPI_INFO_GET (INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)

INTEGER	INFO, VALUELEN, FLAG, IERROR
CHARACTER * (*)	KEY, VALUE

/* MPI 提示信息属性查询函数, 查询 MPI 提示信息的属性个数。

MPI_INFO_GET_NKEYS (info, nkeys)

IN	info	给定的 MPI 提示信息
OUT	nkeys	属性个数

C MPI_Info_get_nkeys (MPI_Info info, int * nkeys)

Fortran MPI_INFO_GET_NKEYS (INFO, NKEYS, IERROR)

INTEGER	INFO, NKEYS, IERROR
---------	---------------------

/* MPI 提示信息属性查询函数, 查询 MPI 提示信息的某个属性的关键词。

MPI_INFO_GET_NTHKEY (info, n, key)

IN	info	给定的 MPI 提示信息
IN	n	待查询属性的序号
OUT	key	附属属性的关键词

C MPI_Info_get_nthkey (MPI_Info info, int n, char * key)

Fortran MPI_INFO_SET_NTHKEY (INFO, N, KEY, IERROR)

INTEGER	INFO, N, IERROR
CHARACTER * (*)	KEY

以上函数创建和定义的提示信息, 可由函数 MPI_FILE_OPEN, MPI_FILE_SET_VIEW 和函数 MPI_FILE_SET_INFO 通过参数 info 传递给 MPI 系统。在前面各节的介绍中, 我们均假设 info = MPI_INFO_NULL, 表示不向 MPI 系统提供任何提示信息。其实, 在许多具体并行机上, 为了提高并行 I/O 性能, 我们有必要通过该参数向 MPI 系统提供一些关键参数, 例如一个文件跨越的硬盘个数、一个文件在每个硬盘连续存储的数据块的大小、文件访问的模式, 以及文件的权限等。

函数 MPI_FILE_SET_INFO 定义如下:

/* MPI 并行 I/O 提示信息设置函数, 为文件联接器设置一个属性。

MPI_FILE_SET_INFO (fh, info)

IN	fh	文件联接器
----	----	-------

```

    IN      info      提示信息
C     MPI_File_Set_Info(MPI_File fh, MPI_Info *info)
Fortran MPI_FILE_SET_INFO(FH, INFO, IERROR)
        INTEGER           FH, INFO, IERROR

```

同时,给定某个文件联接器,函数 MPI_FILE_GET_INFO 可用于查询附属在该联接器的提示信息,具体定义如下:

```
/* MPI 并行 I/O 提示信息查询函数,查询附属在某个文件联接器的提示信息。
```

```

MPI_FILE_GET_INFO(fh, info)
    IN      fh      文件联接器
    OUT     info      提示信息
C     MPI_File_get_info(MPI_File fh, MPI_Info info)
Fortran MPI_FILE_GET_INFO(FH, INFO, IERROR)
        INTEGER           FH, INFO, IERROR

```

MPI 系统为并行 I/O 预定义了四个属性关键词:(1)“striping_factor”表示文件将跨越的硬盘个数;(2)“striping_unit”表示文件在每个硬盘连续存储的数据长度(字节为单位);(3)“cb_buffer_size”表示聚合并行 I/O 函数将使用的数据缓存区大小(字节为单位);(4)“cb_nodes”表示聚合并行 I/O 操作过程中,将有多少个进程访问硬盘。

同时,对某些特殊的并行机平台,MPI 系统还提供预定义好的属性关键词。例如,对 SGI 的 XFS 文件系统,MPI 系统预定义了两个关键词:(1)“direct_read”,当它等于“真”时,表示进程可以直接读该文件系统;(2)“direct_write”,当它等于“真”时,表示进程可以直接写该文件系统。提示信息 info 给出的就是上面叙述的 MPI 系统为并行 I/O 预定义的属性关键词的属性信息。

例 8.13 MPI 提示信息应用示例

```

integer fh, info
call MPI_Info_create(info, ierr)      ! 创建提示信息。
!
! 下面是 MPI 系统为并行 I/O 预定义的 4 个属性
call MPI_Info_set(info, "striping_factor", "4", ierr)      ! 文件将跨越 4 个硬盘。
call MPI_Info_set(info, "striping_unit", "65536", ierr)
                ! 文件在每个硬盘连续存储的数据块大小为 65536 个字节。
call MPI_Info_set(info, "cb_buffer_size", "8388608", ierr)
                ! 聚合并行 I/O 函数将利用的数据缓存区大小为 8388608 个字节。
call MPI_Info_set(info, "cb_nodes", "4", ierr)
                ! 聚合并行 I/O 操作时,将访问硬盘的进程个数为 4。
!
! 下面是 MPI 系统为 SGI XFS 文件系统并行 I/O 预定义的 2 个属性,
call MPI_Info_set(info, "direct_read", "true", ierr)

```

```

call MPI_Info_set(info,"direct_write","true",ierr)
!
! 打开文件，并将以上定义的提示信息传递给 MPI 系统。
call MPI_File_open(MPI_COMM_WORLD,"/pfs/datafile",
                   MPI_MODE_CREATE+MPI_MODE_RDWR,info,fh,ierr)
!
! 释放提示信息。
call MPI_Info_free(info,ierr)

```

8.8 并行 I/O 的数据一致性

为了保证并行 I/O 的正确性，我们必须要求执行 MPI 程序的各个进程在同一个时刻所访问的同一个数据单元的值是相等的，称之为**并行 I/O 的数据一致性**。

如果每个进程访问不同的文件，则显然，并行 I/O 的数据一致性可自然地得到保证，因为每个进程对文件的访问不会影响其他进程。如果所有进程只从同一个文件中读取数据，而不执行任何写操作，每个进程得到的数据单元的值是相同的，也可以保证数据的一致性。但是，如果多个进程访问同一个文件，且其中存在一个进程执行的是写操作，则我们必须充分重视并行 I/O 操作的数据一致性。

假设各个进程访问同一个文件的不同部分，例如：

进程 0	进程 1
call MPI_File_open(COMM, "file", ..., fh1, ...)	call MPI_File_open(COMM, "file", ..., fh2, ...)
call MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ...)	call MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ...)
call MPI_File_read_at(fh1, 0, buf, 100, MPI_BYTE, ...)	call MPI_File_read_at(fh2, 100, buf, 100, MPI_BYTE, ...)

则数据一致性可以得到保证，因为各个进程将不会改变其他进程访问的数据单元。

假设某个进程需要访问被其他进程写入的数据单元，例如：

进程 0	进程 1
call MPI_File_open(COMM, "file", ..., fh1, ...)	call MPI_File_open(COMM, "file", ..., fh2, ...)
call MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ...)	call MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ...)
call MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ...)	call MPI_File_read_at(fh2, 0, buf, 100, MPI_BYTE, ...)

则数据的一致性就变得相当复杂。此时，MPI 系统不提供任何函数来保证文件访问的数据一致性，用户必须采取适当的措施。通常情形下，有 3 种选择。

第一种选择是在各进程将数据写入文件之前，调用函数 MPI_FILE_SET_ATOMICITY 将文件访问模式设置为“atomic”，例如：

进程 0	进程 1
call MPI_File_open(COMM, "file", ..., fh1, ...)	call MPI_File_open(COMM, "file", ..., fh2, ...)
call MPI_File_set_atomicity(fh1, 1, ierr)	call MPI_File_set_atomicity(fh2, 1, ierr)
call MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ...)	call MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ...)
call MPI_Barrier(COMM, ierr)	call MPI_Barrier(COMM, ierr)
call MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ...)	call MPI_File_read_at(fh2, 0, buf, 100, MPI_BYTE, ...)

在“atomic”访问模式中，MPI 系统规定：某个进程写缓存区中数据时，其他进程不能对这些数据进行访问，但缓存区数据写入完毕，这些数据就能被其他进程读取。同时，在上面的程序中，数据写入后，各进程之间的一次同步通信是必须的，它能保证在数据在其他进程读取之前，已经被写入文件中。

第二种选择是写入数据后，关闭文件，然后重新打开，例如：

进程 0	进程 1
call MPI_File_open(COMM, "file", ..., fh1, ...)	call MPI_File_open(COMM, "file", ..., fh2, ...)
call MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ...)	call MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ...)
call MPI_File_close(fh1, ierr)	call MPI_File_close(fh2, ierr)
call MPI_Barrier(COMM, ierr)	call MPI_Barrier(COMM, ierr)
call MPI_File_open(COMM, "file", ..., fh1, ...)	call MPI_File_open(COMM, "file", ..., fh2, ...)
call MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ...)	call MPI_File_read_at(fh2, 0, buf, 100, MPI_BYTE, ...)

第三种选择是各个进程执行并行写操作后，调用函数 MPI_FILE_SYNC 将写操作中的数据强制写入文件中，让其他进程可以看到。函数 MPI_FILE_SYNC，它需要所有进程的共同参与。例如：

进程 0	进程 1
call MPI_File_open(COMM, "file", ..., fh1, ...)	call MPI_File_open(COMM, "file", ..., fh2, ...)
call MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ...)	call MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ...)
call MPI_File_sync(fh1, ierr)	call MPI_File_sync(fh2, ierr)
call MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ...)	call MPI_File_read_at(fh2, 0, buf, 100, MPI_BYTE, ...)
call MPI_File_close(fh1, ierr)	call MPI_File_close(fh2, ierr)

函数 MPI_FILE_SET_ATOMICITY 和函数 MPI_FILE_SYNC 具体定义如下：

```

/* MPI 并行 I/O 同步函数。

MPI_FILE_SYNC(fh)
    IN          fh          文件联接器
C      MPI_File_sync (MPI_File fh)
Fortran MPI_FILE_SYNC (FH, IERROR)
        INTEGER      FH, IERROR

/* MPI 并行 I/O 文件访问模式设置函数。

MPI_FILE_SET_ATOMICITY(fh, flag)
    IN          fh          文件联接器
    IN          flag         若 flag = 1, 表示将文件访问模式设置为"atomic"
C      MPI_File_Set_atomicity (fh, flag)
Fortran MPI_FILE_SET_ATOMICITY (FH, FLAG, IERROR)
        INTEGER      FH, FLAG, IERROR

```

8.9 文件的可移植性

为了利用 MPI 系统提供的并行 I/O 函数进行并行程序设计, 我们必须搞清楚两个问题: 第一, 一个被 MPI 系统创建的文件能否像普通的 UNIX 操作系统创建的文件一样, 被非 MPI 的程序所访问? 第二, 由于 MPI 并行 I/O 只支持无格式文件, 如何将一个在某台计算机上创建的 MPI 无格式文件移植到另一台计算机, 也就是说, MPI 无格式文件能否具有可移植性?

首先, 我们回答第一个问题。实际上, 由于 MPI 系统并行 I/O 函数的具体实现一般均基于普通的 UNIX 操作系统的文件操作, 因此, 任何 MPI 无格式文件均可以像其他普通文件一样, 被非 MPI 的程序访问。

其次, 我们回答第二个问题。由于不同的计算机具有不同的数据表示格式(例如字节顺序、数据类型的大小等), 通常情形下, 在某台计算机上创建的无格式文件不能直接移植到其他计算机上, 除非程序能自动识别不同表示格式的差别。但是, MPI 系统提供文件的可移植功能, 用户通过某些选项, 可以创建可移植的 MPI 无格式文件。具体地, 在函数 MPI_FILE_SET_VIEW 中, 我们可以通过设置参数 datarep 的值来保证 MPI 无格式文件的可移植性。

MPI 系统提供 3 种预定义好的格式: native, internal 和 external32。其中, 格式 native 表示数据将以它在内存中的表示格式, 直接写入文件而不做任何转换, 它也是 MPI 文件数据的缺省表示格式。显然, 以格式 native 写的文件只能在该类并行机上运行, 因此, 该格式将不保证文件的可移植性。格式 internal 表示数据将以 MPI 系统内部格式写入文件, 它可以使文件在所有支持同一类 MPI 系统的并行机上具有可移植性。格式 external32 是 MPI 系统按 32 位 IEEE big-endian 标准格式提供的数据表示格式, 以该格式创建的文件将在所有支持 MPI 并行程序设计平台的并行机上具有可移植性, 同时它也隐含数据格式的自动转换。

由于格式 native 不需要数据的格式转换, 而格式 external32 需要数据格式的转换, 因此, 基于格式 external32 的 MPI 并行 I/O 操作的性能, 将低于基于格式 native 的相应 MPI 并行 I/O 操作。

除了数据格式的可移植性外, 为了保证 MPI 无格式文件移植的正确性, 用户还要注意不同机器上相同数据类型的差异, 也就是数据类型的域。为此, MPI 系统提供函数 MPI_FILE_TYPE_EXTENT 来获取文件中某个数据类型的域, 具体定义如下:

/* MPI 并行 I/O 文件数据类型域查询函数。

```
MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)
  IN      fh          文件联接器
  IN      datatype    数据类型
  OUT     extent      datatype 的域
C      MPI_File_get_type_extent (fh, datatype, extent)
Fortran MPI_FILE_GET_TYPE_EXTENT (FH, DATATYPE, EXTENT, IERROR)
      INTEGER    FH, DATATYPE, EXTENT, IERROR
```

MPI 系统除了提供预定义好的 3 种数据表示格式, 还支持用户自定义数据表示格式。这属于 MPI 2.0 的新特征, 这里不再介绍。

第九章 MPI 系统环境管理

本章讨论 MPI 系统中与具体并行机执行环境相关的函数,其中包括进入和退出 MPI 系统、获取墙上时间、查询 MPI 系统信息及处理 MPI 系统异常等。

9.1 进入和退出 MPI 系统

应用程序通过调用函数 MPI_INIT 和函数 MPI_FINALIZE, 分别进入和退出 MPI 系统。当且仅当在函数 MPI_INIT 被调用之后和函数 MPI_FINALIZE 被调用之前, 应用程序才能调用 MPI 函数。

```
/* MPI 初始化函数, 应用程序进入 MPI 系统

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

/* MPI 退出函数, 应用程序退出 MPI 系统

MPI_FINALIZE ()
C      int MPI_Finalize ()
Fortran MPI_FINALIZE (IERROR)
           INTEGER IERROR
```

函数 MPI_INIT 将为 MPI 程序的各个进程设置并行计算所必需的初始化信息, 其中包括通信器 MPI_COMM_WORLD 和 MPI_COMM_SELF、所有进程的同步与进程的序号等。函数 MPI_FINALIZE 在进程退出 MPI 系统之前, 执行一次同步操作, 并清除所有与 MPI 系统相关的设置。应当注意, 函数 MPI_INIT 和函数 MPI_FINALIZE 只能被调用一次。

在许多并行机的 MPI 具体实现中, 函数 MPI_FINALIZE 不仅使执行 MPI 程序的各个进程退出 MPI 系统外, 而且还隐含中断进程的执行。因此, 为了保证程序的可移植性, 一般情形下, 我们均建议读者在应用程序的最后一条语句调用函数 MPI_FINALIZE。

MPI 系统提供函数 MPI_INITIALIZED, 辅助执行 MPI 程序的各进程查询函数 MPI_INIT 是否已经被调用, 它是唯一一个能在函数 MPI_INIT 之前被调用的 MPI 函数。

```
/* MPI 查询函数, 返回函数 MPI_INIT 是否被调用的信息。
```

```
MPI_INITIALIZED (flag)
```

```

        OUT      flag      true 表示函数 MPI_INIT 已被调用, flase 表示否
C      int MPI_Initialized (int * flag)
Fortran MPI_INITIALIZED (FLAG, IERROR)
        LOGICAL FLAG
        INTEGER IERROR

```

除函数 MPI_FINALIZE 外, MPI 系统还提供函数 MPI_ABORT, 使包含在某个通信器中的所有进程退出 MPI 系统。

/* MPI 系统退出函数, 进程退出 MPI 系统。

```

MPI_ABORT (comm, errorcode)
        IN      comm      通信器
        IN      errorcode    错误处理联接器
C      int MPI_Abort (MPI_Comm comm, int errorcode)
Fortran MPI_ABORT (COMM, ERRORCODE, IERROR)
        INTEGER COMM, ERRORCODE, IERROR

```

函数 MPI_ABORT 使通信器 comm 包含的所有进程退出 MPI 系统, 并中断这些进程的执行。如果参数 comm = MPI_COMM_WORLD, 则隐含所有进程将退出 MPI 系统, 并中断执行; 如果参数 comm ≠ MPI_COMM_WORLD, 则除了 comm 包含的所有进程退出 MPI 系统外, 其他进程是否退出 MPI 系统取决于 MPI 系统在并行机上的具体实现。因此, 为了保证 MPI 程序的可移植性, 用户最好只选择参数 comm = MPI_COMM_WORLD。

参数 errorcode 是应用程序提交给函数 MPI_ABORT 的错误处理联接器, 它将被函数 MPI_ABORT 用于激活 MPI 系统内部定义的或用户自定义的相应错误处理程序。

9.2 获取墙上时间

MPI 系统提供函数 MPI_WTIME, 辅助执行 MPI 程序的各个进程获取当前程序执行指令的格林威治时间, 即从过去某个固定的时间 (例如, 1970 年 1 月 1 日零时零分零秒零...) 开始, 到目前时刻, 墙上时钟已经走过的时间 (单位为秒)。

/* MPI 系统时间函数, 返回当前的格林威治时间。

```

MPI_WTIME ()
C      double MPI_Wtime (void)
Fortran DOUBLE PRECISION MPI_WTIME ()

```

墙上时间 指某段程序从开始执行到完成, 墙上时钟所走过的时间, 即该段程序的运行时间。用户若想知道某段程序执行的墙上时间, 则只需在进入和退出该段程序的前后分别调用函数 MPI_WTIME, 然后用退出时的时间减去进入时的时间, 即可得该段程序

执行的墙上时间：

```
+ double precision starttime, endtime  
starttime = MPI_WTIME ()  
程序段执行  
endtime = MPI_WTIME ()  
print *, "That took", endtime-starttime, "seconds"  
|
```

函数 MPI_WTIME 属于局部函数，只返回调用进程当前的格林威治时间，与其他进程的状态无关。

/* 时间刻度函数，返回格林威治时间统计的最小刻度。

```
C      MPI_WTICK ()  
C      double MPI_Wtick (void)  
Fortran DOUBLE PRECISION MPI_WTICK ()
```

函数 MPI_WTICK 以秒为单位，返回函数 MPI_WTIME 的最短时间刻度，即连续时钟信号 (ticks) 之间的间隔。例如，若时钟信号计数器每隔 1 毫秒累加一次，则该函数调用将返回 10^{-3} 。

9.3 MPI 系统常数的查询

为了方便用户设计可移植并行 MPI 程序，MPI 系统定义了许多常数。但是，部分常数的值依赖于具体并行机，必须在使用之前加以确认。

以下 4 个 MPI 常数是初始通信器 MPI_COMM_WORLD 的附加属性，且依赖于具体并行机：(1) MPI_TAG_UB：消息号的上界值；(2) MPI_HOST：位于主机的进程序号，若主机不存在，则返回 MPI_PROC_NULL；(3) MPI_IO：位于 I/O 结点上进程的序号；(4) MPI_WTIME_IS_GLOBAL：逻辑变量，true 表示所有进程的时钟是同步的。此外，我们可以使用第六章中介绍的函数 MPI_ATTR_GET 来获得这些常数的值。

MPI 系统提供函数 MPI_GET_PROCESSOR_NAME，辅助执行 MPI 程序的各个进程获取它所在处理机上的名字。其中，参数 name 的长度一般可以设置为 MPI_MAX_PROCESSOR_NAME。

/* 处理机名字查询函数，获取当前调用进程所在处理机的名字。

```
MPI_GET_PROCESSOR_NAME (name, resultlen)  
    OUT      name      处理机名字  
    OUT      resultlen  处理机名字的长度  
C      int MPI_Get_processor_name (char * name, int * resultlen)  
Fortran MPI_GET_PROCESSOR_NAME (NAME, RESULTLEN, JERROR)  
        CHARACTER * (*) NAME
```

```
INTEGER RESULTLEN, IERROR
```

9.4 MPI 异常及其处理

应用程序在调用 MPI 系统函数过程中, 可能会出现诸如 3.1.1 节中表 3.1 列出的各类信息码所对应的错误, 或者 MPI 系统内部的错误等, 这些均称为 **MPI 错误** (error)。每个 MPI 错误均将产生一个 **MPI 异常** (exception), 需要相应的 MPI 异常处理程序进行处理。一个好的 MPI 实现应该能处理尽可能多的 MPI 异常。

通常情形下, MPI 异常将激活相应的 MPI 错误处理程序 (error handler), 并返回一个相应的信息码, 同时也可能中断进程的执行。如果某个 MPI 函数导致了多个 MPI 异常, 则需要激活多个 MPI 错误处理程序, 并返回多个错误码。

9.4.1 错误处理程序

MPI 系统的错误处理程序是 MPI 系统内部定义的对象, 可通过相应的错误处理联接器进行访问。用户可以将某个错误处理程序通过联接器附属到某个通信器, 一旦与该通信器相关的某个 MPI 发生异常, 则该错误处理程序将被激活。

错误处理程序可附属于某个通信器, 并具有:(1) 可继承性: 即新创建的通信器将继承父通信器的所有错误处理程序;(2) 局部性: 即各个进程可以独立地将多个不同的错误处理程序附属于该通信器。由此, MPI 程序通常可以在调用函数 MPI_INIT 之后, 将一些错误处理程序附属到通信器 MPI_COMM_WORLD 上, 使之成为将来有任意通信器的属性。

MPI 系统提供两个已经定义好的错误处理程序:

- (1) MPI_ERRORS_FATAL: MPI 函数调用产生的任何异常均是致命的, 并中断所有进程的执行。
- (2) MPI_ERRORS_RETURN: MPI 函数调用产生的任何异常均返回相应错误码, 但对程序的执行不产生任何影响。

MPI_ERRORS_FATAL 是附属于初始通信器 MPI_COMM_WORLD 的缺省错误处理程序。如果用户希望判别 MPI 函数调用返回的错误码, 则必须通过函数 MPI_ERRHANDLER_SET 将 MPI 系统的缺省错误处理程序替换为 MPI_ERRORS_RETURN。此外, 用户还可以根据需要, 创建满足特殊需求的错误处理程序, 并通过联接器附属于某个通信器上。

错误处理程序可由如下函数来创建、设置、查询或释放。

```
/* MPI 错误处理程序创建函数, 创建一个 MPI 异常错误处理程序。
```

```
MPI_ERRHANDLER_CREATE (function, errhandler)
    IN      function      进程定义的 MPI 异常错误处理程序
    OUT     errhandler    MPI 错误处理程序联接器
C      int MPI_Errhandler_create (MPI_Handler_function * function,
                                MPI_Errhandler * errhandler)
```

```

Fortran MPI_ERRHANDLER_CREATE (FUNCTION,ERRHANDLER,IERROR)
      EXTERNAL FUNCTION
      INTEGER ERRHANDLER,IERROR

```

函数 MPI_ERRHANDLER_CREATE 局部地将函数 function 设置为一个 MPI 异常处理程序，并返回相应的错误处理联接器 errhandler。函数 function 由各个进程自己提供，在 C 语言中，函数 function 可定义为：

```
typedef void(MPI_Handler_function)(MPI_Comm * comm, int * error_code, ...)
```

而在 Fortran 语言中，格式可定义为：

```
subroutine function(COMM,Error_code,...)
```

其中，第一个参数 comm 是当前进程使用的通信器，error_code 是 MPI 程序返回的错误信息码。

```
/* MPI 错误处理程序设置函数，将某个 MPI 异常错误处理程序附属于某个通信器。
```

```

MPI_ERRHANDLER_SET (comm,errhandler)
    IN      comm      通信器
    IN      errhandler      MPI 异常错误处理程序联接器
C     int MPI_Errhandler_set (MPI_Comm comm, MPI_Errhandler * errhandler)
Fortran MPI_ERRHANDLER_SET (COMM,ERRHANDLER,IERROR)
      INTEGER COMM,ERRHANDLER,IERROR

```

函数 MPI_ERRHANDLER_SET 局部地将进程定义的 MPI 异常错误处理程序，通过其联接器 errhandler 附属于通信器 comm。

```
/* MPI 错误处理程序查询函数，获取附属于某个通信器的错误处理程序。
```

```

MPI_ERRHANDLER_GET (comm,errhandler)
    IN      comm      通信器
    OUT     errhandler      MPI 异常错误处理程序联接器
C     int MPI_Errhandler_get (MPI_Comm comm, MPI_Errhandler * errhandler)
Fortran MPI_ERRHANDLER_GET (COMM,ERRHANDLER,IERROR)
      INTEGER COMM,ERRHANDLER,IERROR

```

函数 MPI_ERRHANDLER_GET 查询进程附属于通信器 comm 的 MPI 异常错误处理程序。

```
/* MPI 错误处理程序释放函数，释放附属于某个通信器的 MPI 异常错误处理程序。
```

```

MPI_ERRHANDLER_FREE (errhandler)
    IN      errhandler      MPI 错误处理程序联接器
C     int MPI_Errhandler_free (MPI_Errhandler * errhandler)
Fortran MPI_ERRHANDLER_FREE (ERRHANDLER,IERROR)

```

```
INTEGER ERRHANDLER, IERROR
```

函数 MPI_ERRHANDLER_FREE 是进程释放联接器 errhandler 联接的错误处理程序。如果该错误处理程序是所有进程共享的，并存在某个进程正在使用该程序，则必须等到该进程使用完毕，该错误处理程序才被真正释放。

9.4.2 信息码

在第三章 3.1.1 节的表 3.1 中，我们已列出 MPI 函数可能返回的标准信息码及其含义。为了更好地理解这些标准信息码，MPI 系统提供函数 MPI_ERROR_STRING 将每个标准信息码翻译成对应的消息。

```
/* 信息码翻译函数，将信息码翻译成对应的消息。
```

```
MPI_ERROR_STRING (errorcode, string, resultlen)
    IN      errorcode      MPI 函数返回的信息码
    OUT     string         与信息码对应的消息
    OUT     resultlen     参数 string 的长度
C      int  MPI_Error_string (int errorcode, char * string, int * resultlen)
Fortran MPI_ERROR_STRING (ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER  ERRORCODE, RESULTLEN, IERROR
    CHARACTER * ( *) STRING
```

如果函数返回的信息码不是表 3.1 中列出的标准信息码，则 MPI 系统还提供函数 MPI_ERROR_CLASS 将这些信息码矫正为标准信息码。

```
/* 信息码矫正函数，将任意 MPI 函数调用返回的信息码矫正成标准信息码。
```

```
MPI_ERROR_CLASS (errorcode, errorclass)
    IN      errorcode      MPI 函数返回的信息码
    OUT     errorclass     矫正的 MPI 标准信息码
C      int  MPI_Error_class (int errorcode, int * errorclass)
Fortran MPI_ERROR_CLASS (ERRORCODE, ERRORCLASS, IERROR)
    INTEGER  ERRORCODE, ERRORCLASS, IERROR
```

第十章 MPI 与 OpenMP 的混合编程

随着对称多处理 (SMP) 共享存储并行机的出现, 另一种并行编程环境 OpenMP, 由于其使用简单, 已逐渐被用户所接受。OpenMPI 1.0 版于 1997 年形成, 是目前最流行的基于线程的并行程序设计平台, 并被所有现代高性能共享存储并行机所支持。有关它的详细讨论和应用, 请读者参考相关文献, 本书这里只讨论它与 MPI 的混合编程技术。

10.1 MPI 进程的多线程执行

在第一章, 我们介绍了线程的概念, 指出在现代操作系统中, 尤其是对称多处理 (SMP) 并行机中, 一个进程的指令路径可以由多个线程来执行, 而这些线程共享进程的内存地址空间, 又可由不同的 CPU 来执行, 故可以达到并行执行进程的目的。

MPI 1.0 版支持在每个 MPI 进程的内部, 实现多个线程的执行, 而 MPI 2.0 版进一步增强了 MPI 系统在这方面的功能。MPI 进程的多线程执行, 具有如下的益处:

(1) 多线程执行能很方便地实现进程间的非阻塞通信, 有助于重叠通信与计算, 降低 MPI 程序的通信延迟。例如, 当一个进程执行一个 MPI 阻塞式消息接收操作时, 它可以创建一个新的线程, 由这个线程来阻塞完成消息接收操作, 而其自身可以继续后面的计算; 直到需要该消息包含的数据为止, 该进程才与创建的线程同步, 确保该次消息接收操作的完成。同样的情况也适合于阻塞式消息发送操作。

(2) 多线程执行能极大地方便聚合通信操作的具体实现, 并提高它们的通信效率。
(3) 在 SMP-Cluster 并行机上, SMP 结点内部的多线程执行能极大地发挥 SMP 结点的内存访问带宽, 减少网络流量, 有助于提高并行计算的性能。

(4) 多线程执行将有助于降低并行应用程序的开发代价。一般情况下, 多线程并行程序的可读性优于 MPI 并行程序, 其设计 (例如 OpenMP) 的难度也小于 MPI 并行程序设计的难度, 而当处理器个数较少时, 多线程并行程序的并行性能可以与 MPI 并行程序的性能相当。因此, 根据某些应用问题和已经开发好的串行程序的特点, 首先, 我们可以简单地实现它们基于 MPI 的大粒度并行; 然后, 对每个 MPI 进程, 实现它们基于多线程的并行, 从而在付出较小的开发代价的基础上, 尽可能地获得较高的性能。

为了实现进程内部的多线程并行, MPI 系统的消息传递库函数必须是线程安全的, 即多个线程可以同时调用消息传递库函数而不相互干扰。反之, 并行机操作系统的线程库函数也必须知道 MPI 的具体实现, 其中包括:(1) 当某个线程由于各种原因被阻塞时, 它必须将 CPU 控制权转交给它所属的进程的其他线程, 而不是阻塞该进程;(2) 当线程阻塞的条件被满足时, 该线程能立即投入执行;(3) 当线程执行一个系统调用时, 操作系统只阻塞该线程, 而不影响其他线程的状态。

10.2 MPI 与 OpenMP 的混合编程

为了在 MPI 进程的内部实现 OpenMP 多线程并行，并保证 MPI 应用程序的线程安全，我们必须用函数 MPI_INIT_THREAD 替换函数 MPI_INIT，具体定义如下：

```
/* MPI 多线程初始化函数,进入 MPI 系统,并保证线程安全

MPI_INIT_THREAD (required, provided)
    IN      required      应用程序请求的线程安全模式
    OUT     provided      具体并行机 MPI 实现能提供的线程安全模式
C      int MPI_Init_thread (int * argc, char ** * argv, int required, int * provided)
Fortran MPI_INIT_THREAD (REQUIRED, PROVIDED, IERROR)
        INTEGER REQUIRED, PROVIDED, IERROR
```

其中，输入参数 required 是应用程序请求的线程安全模式，而输出参数 provided 是应用程序能满足的线程安全模式。MPI 2.0 标准提供以下四种线程安全模式：

- (1) MPI_THREAD_SINGLE: MPI 进程仅由一个线程执行，表示当前并行机的 MPI 不支持 MPI 进程的多线程执行；
- (2) MPI_THREAD_FUNNELED: MPI 进程可以由多个线程执行，但是，只有主线程才能调用 MPI 函数；
- (3) MPI_THREAD_SERIALIZED: MPI 进程可以由多个线程执行，各个线程也可以调用 MPI 函数，但是，在同一个时刻，至多存在一个线程调用 MPI 函数，MPI 程序自身必须保证这一点，否则，程序的执行将可能出现错误；
- (4) MPI_THREAD_MULTIPLE: MPI 进程可以由多个线程执行，各个线程也可以在任何时刻调用 MPI 函数。

实际上，可以看出，只有第四种模式，才是真正的线程安全。

MPI 系统提供函数 MPI_IS_THREAD_MAIN，各个线程可用它来查询自己是否为主线程。

```
/* 主线程查询函数。

MPI_IS_THREAD_MAIN (flag)
    OUT      flag          若 flag = true，则该线程是主线程，否则该线程是从线程
C      int MPI_Is_thread_main (int * flag)
Fortran MPI_IS_THREAD_MAIN (FLAG, IERROR)
        LOGICAL FLAG
        INTEGER IERROR
```

OpenMP 环境变量 OMP_NUM_THREADS，可用于为每个执行 MPI 程序的进程设置可并行执行的线程个数，而各个进程也可以在执行过程中，直接调用 OpenMP 的函数，动态地设置并行执行的线程个数。例如：

```

call MPI_Comm_rank(MPI_COMM_WORLD, ierr)
nthreads = getenv("OMP_NUM_THREADS")
          ! 获取环境变量设置的线程个数。
if(rank.eq.0)then
    if(nthread.ne.4)call OMP_SET_NUM_THREADS(4)
          ! 设置线程个数等于 4。
else
    if(nthread.ne.2)call OMP_SET_NUM_THREADS(2)
          ! 设置线程个数等于 2。
endif
.....

```

该例中, 进程 0 内部可并行执行的线程个数为 4, 而其他进程内部可并行执行的线程个数为 2。

10.3 并行矩阵乘混合编程示例

对应于第三章例 3.2, 下面给出并行矩阵乘的 MPI 与 OpenMP 混合编程例子。

```

PARAMETER(MP=M/P, LP=L/P)           ! 子矩阵规模参数说明。
REAL * 8     A(MP,N), B(N,LP), C(MP,L)   ! 子矩阵说明。
INTEGER      I, J, K, KK, MYRANK, MYLEFT, MYRIGHT, IC, IERR
INTEGER      NSTATUS(MPI_STATUS_SIZE)
INTEGER      REQUIRED, PROVIDED
!
! 初始化
REQUIRED = MPI_THREAD_FUNNELED
CALL MPI_INIT_THREAD(REQUIRED, PROVIDED, IERR)
IF(PROVIDED.LT.REQUIRED)THEN
    PRINT *, "+++" Required MPI thread-safe model is not supported + + +"
    STOP
ENDIF
C $OMP CALL OMP_SET_NUM_THREADS(2)      ! 每个进程可由两个线程并行执行。
!
! 并行子矩阵赋初值 (忽略)。
!
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
MYLEFT = MYRANK - 1
MYRIGHT = MYRANK + 1
IF(MYLEFT.EQ.1) MYLEFT = P - 1
IF(MYRIGHT.EQ.P) MYRIGHT = 0
IC = (MYRANK - 1) * LP
!
```

```

! MPI 并行矩阵乘。
DO 100 K=1,P
    IC= IC+ LP
    IF (IC.GE. L)  IC= IC- L
    !
    ! OpenMP 多线程并行执行以下循环的子矩阵乘。
C$OMP PARALLEL DO,Default (shared), Local (J,I,KK)
    ! OpenMP 并行指导语句。
    DO 10 J= 1,LP
    DO 10 I= 1,MP
    DO 10 KK= 1,N
        C(I,IC+ J)= C(I,IC+ J)+ A(I,KK) * B(KK,J)
10    CONTINUE
C$OMP END PARALLEL DO          ! OpenMP 并行指导语句。
    ! OpenMP 多线程并行执行子矩阵乘结束。
    !
    IF(K.EQ. P)GOTO 100
    !
    ! 主线程执行消息传递,依次交换矩阵 B 的子块。
    CALL MPI_SEND(B, N * LP, MPI_DOUBLE_PRECISION, MYLEFT, K * 100,
                  MPI_COMM_WORLD, IERR)
    CALL MPI_RECV(B, N * LP, MPI_DOUBLE_PRECISION, MYRIGHT, K * 100,
                  MPI_COMM_WORLD, NSTATUS, IERR)
    !
100  CONTINUE
    !
    ! 正确性验证,程序退出(忽略)。

```

第十一章 MPI 程序示例

基于前面各章介绍的 MPI 函数, 本章将设计具体的 MPI 程序, 求解定义在二维规则区域上的 Laplace 方程:

$$\begin{cases} -\Delta u(x, y) = f(x, y) & (x, y) \in \Omega = (0, a) \times (0, b) \\ u(x, y)|_{\partial\Omega} = g(x, y) \end{cases} \quad (11.1)$$

其中, $f(x, y)$ 和 $g(x, y)$ 为已知函数, 分别定义在区域 Ω 的内部和边界。

11.1 并行算法设计

沿坐标轴 x 和 y 方向, 分别取步长为:

$$h_x = a/IM, h_y = b/JM \quad (11.2)$$

将区域 Ω 离散成规模为 $IM \times JM$ 的网格, 其中 IM 和 JM 分别为沿坐标轴 x 和 y 方向的网格单元个数。

不妨设方程 (11.1) 的近似解 $u(x, y)$ 定义在所有网格结点上, 且用如下未知量表示:

$$\begin{cases} u_{i,j} = u(i \times h_x, j \times h_y) & 1 \leq i \leq IM - 1, 1 \leq j \leq JM - 1 \\ u_{i,j} = g_{i,j} = g(i \times h_x, j \times h_y) & i = 0 \text{ 或 } i = IM \text{ 或 } j = 0 \text{ 或 } j = JM \end{cases} \quad (11.3)$$

令:

$$u_{xx}|_{i,j} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2}, \quad u_{yy}|_{i,j} = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} \quad (11.4)$$

并记:

$$f_{i,j} = f(i \times h_x, j \times h_y) \quad (11.5)$$

将以上各式代入方程 (11.1), 则问题转化为求解稀疏线性代数方程组:

$$\begin{cases} 2(h_x^2 + h_y^2)u_{i,j} - h_y^2(u_{i-1,j} + u_{i+1,j}) - h_x^2(u_{i,j-1} + u_{i,j+1}) = h_x^2h_y^2f_{i,j} \\ u_{i,j} = g_{i,j} \end{cases} \quad \begin{array}{l} 1 \leq i \leq IM - 1, 1 \leq j \leq JM - 1 \\ i = 0 \text{ 或 } i = IM \text{ 或 } j = 0 \text{ 或 } j = JM \end{array} \quad (11.6)$$

具体地, 我们将采用众所周知的 Jacobi 点迭代算法求解方程 (11.6)。

11.2 MPI 并行程序设计

设计求解方程 (11.1) 的 MPI 并行程序必须考虑两个关键问题:

第一, 选择正确的区域分解策略。将区域 Ω 分解为多个子区域, 分配给不同的进程, 并保证进程间的负载平衡和最小的消息传递通信开销。通常有两种方式:(1) 沿 y 方向

的一维条分解策略,如图 11.1(a)所示;(2)沿两个方向的二维块分解策略,如图 11.1(b)所示。显然,如果取 x 方向的进程个数等于 1,则二维块分解策略就退化为一维条分解策略。无论哪种方式,都应该尽量保证每个子区域包含的网格结点个数相等,因为这样才能保证进程间的负载平衡。

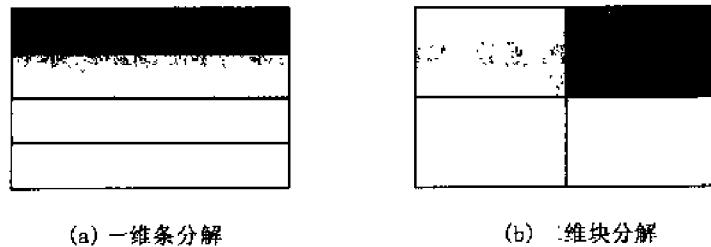


图 11.1 两种区域分解策略

第二,选择合适的通信数据结构。由式(11.6)可知,在任意网格结点上,执行 Jacobi 点迭代需要知道该结点的上、下、左、右四个相邻结点上的近似解。因此,在每次 Jacobi 迭代之前,每个进程必须与其相邻的进程交换边界结点上的近似解。

为了描述通信数据结构,不妨设方程近似解定义在网格单元的中心。图 11.2 给出了一个 3×3 的二维块区域分解,其中,各个子区域被分配给不同的进程,各个进程负责求解该子区域的近似解。具体地,进程 5 将向其相邻的四个进程(进程 2、进程 4、进程 6 和进程 8)输出“○”标示的网格单元的近似解;同时,从这四个进程接收用“○”标示的网格单元上的近似解。因此,如何管理这些沿区域分解边界交换的网格单元上的量,将会直接影响到 MPI 程序的并行性能。图 11.3 给出一个比较有效的办法,就是沿各个子区域的边界,向外增加一个宽度为 1 的辅助网格单元,用于存储相邻子区域在这些网格单元上的近似解。

类似,同样的数据结构也适应于近似解定义在网格结点上的情形,这里不再讨论。

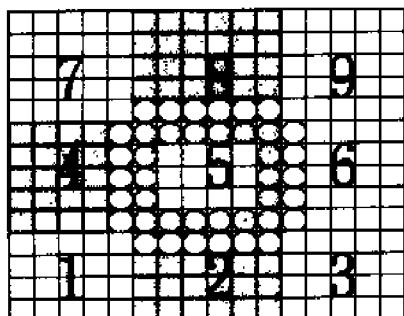


图 11.2 3×3 二维块区域分解

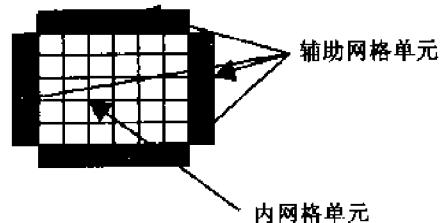


图 11.3 辅助网格单元示意图

下面,基于以上二维块区域分解策略和通信数据结构,给出求解差分方程(11.6)的 MPI 并行程序。其中,近似解定义在网格结点上。为了简单,这里假设网格单元个数 IM 和 JM 均能分别被沿 x 方向和 y 方向的进程个数 NPX 和 NPY 整除,进程的序号按自然序排列(先沿 x 方向,后沿 y 方向)。

例 11.1 并行 Jacobi 点迭代 MPI 程序: 二维块分解策略

```
!
INCLUDE "mpif.h"
PARAMETER (DA = 2.0, DB = 3.0)      ! 问题求解区域沿 x, y 方向的大小。
PARAMETER (IM = 1028, JM = 1028)    ! 沿 x, y 方向的全局网格规模。
PARAMETER (NPX = 4, NPY = 2)        ! 沿 x, y 方向的进程个数。
PARAMETER (IML = IM/NPX, JML = JM/NPY)
                                ! 各进程沿 X, Y 方向的局部网格规模, 仅为全局网格规模的 1/(NPX * NPY)。
REAL   U (0:IML+1, 0:JML+1)       ! 定义在网格结点的近似解。
REAL   US (0:IML+1, 0:JML+1)     ! 定义在网格结点的精确解。
REAL   U0 (IML, JML)             ! Jacobi 迭代辅助变量。
REAL   F (IML, JML)              ! 函数 f(x, y) 在网格结点上的值。
INTEGER NPROC                   ! mpirun 启动的进程个数, 必须等于 NPX * NPY。
INTEGER MYRANK, MYLEFT, MYRIGHT, MYUPPER, MYLOW
                                ! 各进程自身的序号, 4 个相邻进程的序号。
INTEGER MEPX, MEPY              ! 各进程自身的序号沿 x, y 方向的坐标。
REAL   XST, YST                 ! 各进程拥有的子区域沿 x, y 方向的起始坐标。
REAL   HX, HY                   ! 沿 x, y 方向的网格离散步长。
REAL   HX2, HY2, HXY2, RHXY
INTEGER IST, IEND, JST, JEND
                                ! 各进程沿 x, y 方向的内部网格结点的起始和终止坐标。
INTEGER HTYPE, VTYPE
                                ! MPI 用户自定义数据类型, 表示各进程沿 x, y 方向。
                                ! 与相邻进程交换的数据单元。
INTEGER STATUS(MPI_STATUS_SIZE)
!
! 程序可执行语句开始。
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROC, IERR)
IF(NPROC.NE.NPX * NPY.OR. MOD(IM, NPX).NE.0.OR. MOD(JM, NPY).NE.0)THEN
  PRINT *, "+ + + mpirun -np xxx error OR grid scale error, exit out + + +"
  STOP
ENDIF
!
! 按自然序(先沿 x 方向, 后沿 Y 方向)确定各进程自身及其 4 个相邻进程的序号。
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
MYLEFT = MYRANK - 1
IF(MOD(MYLEFT, NPX).EQ.NPX - 1) MYLEFT = MPI_PROC_NULL
MYRIGHT = MYRANK + 1
IF(MOD(MYRIGHT, NPX).EQ.0) MYRIGHT = MPI_PROC_NULL
MYUPPER = MYRANK + NPX
IF(MYUPPER.GT.NPROC - 1) MYUPPER = MPI_PROC_NULL
```

```

MYLOW = MYRANK - NPX
IF(MYLOW.LT.0)  MYLOW = MPI _ PROC _ NULL
MEPY = MOD(MYRANK, NPX)
MEPX = MYRANK - MEPY * NPX
! 对应二维  $NPY \times NPX$  Cartesian 行主序坐标为 (MEPY, MEPX)。
!
! 基本变量赋值, 确定各进程负责的子区域。
HX = DA/IM
HX2 = HX * HX
HY = DB/JM
HY2 = HY * HY
HXY2 = HX2 * HY2
RHXY = 0.5/(HX2 + HY2)
XST = MEPX * DA/NPX
YST = MEPY * DB/NPY
IST = 1
IEND = IML
IF (MEPX.EQ.NPX - 1)  IEND=IEND - 1
JST = 1
JEND = JML
IF (MEPY.EQ.NPY - 1)  JEND=JEND - 1
!
! 数据类型定义。
CALL MPI _ TYPE _ CONTIGUOUS(IEND - IST + 1, MPI _ REAL, HTYPE, IERR)
! 沿  $x$  方向的连续 IEND - IST + 1 个 MPI _ REAL 数据单元。
! 可用于表示该进程与其上、下进程交换的数据单元。
CALL MPI _ TYPE _ VECTOR(JEND - JST + 1, 1, IML + 2, MPI _ REAL, VTYPE, IERR)
! 沿  $y$  方向的连续 JEND - JST + 1 个 MPI _ REAL 数据单元。
! 可用于表示该进程与其左、右进程交换的数据单元。
!
! 近似解赋初值。
DO 10 J = JST - 1, JEND + 1
DO 10 I = IST - 1, IEND + 1
    U(I,J) = 0.0
10  CONTINUE
!
! 边界条件处理
IF(MEPX.EQ.0)THEN          ! 左 Dirichlet 边界条件。
    DO J = JST, JEND
        xx = 0.0
        yy = YST + J * HY
        U(0,J) = g(xx, yy)
    ENDDO

```

```

IF(MEPY.EQ.0)      U(0,0)=g(0.0,0.0)
IF(MEPY.EQ.NPY-1)U(0,JEND+1)=g(0.0,DB)
ENDIF
IF(MEPX.EQ.NPX-1)THEN          !右 Dirichlet 边界条件。
DO J=JST,JEND
  xx=DA
  yy=YST+J*HY
  U(IEND+1,J)=g(xx,yy)
ENDDO
IF(MEPY.EQ.0)      U(IEND+1,0)=g(DA,0.0)
IF(MEPY.EQ.NPY-1)U(IEND+1,JEND+1)=g(DA,DB)
ENDIF
IF(MEPY.EQ.0)THEN          !下 Dirichlet 边界条件。
DO I=JST,JEND
  xx=XST+I*HX
  yy=0.0
  U(I,0)=g(xx,yy)
ENDDO
ENDIF
IF(MEPY.EQ.NPY-1)THEN          !上 Dirichlet 边界条件。
DO I=JST,JEND
  xx=XST+I*HX
  yy=DB
  U(I,JEND+1)=g(xx,yy)
ENDDO
ENDIF
!
! 精确解赋值。
DO 15 J=JST-1,JEND+1
DO 15 I=IST-1,IEND+1
  US(I,J)=U(I,J)
  IF(I.EQ.IST-1.OR.I.EQ.IEND+1.OR.J.EQ.JST-1.OR.J.EQ.JST-1)GOTO 15
  xx=XST+I*HX
  yy=YST+J*HY
  US(I,J)=solution(xx,yy)      ! solution 表示精确解。
15 CONTINUE
!
! 右端函数赋值。
DO 20 J=JST,JEND
DO 20 I=IST,IEND
  xx=XST+I*HX
  yy=YST+J*HY
  F(I,J)=f(xx,yy)

```

```

20  CONTINUE
!
! Jacobi 迭代求解。
NITER = 0
100 CONTINUE
NITER = NITER + 1
DO 30 J=JST,JEND
DO 30 I=IST,IEND
U0(I,J) = RHXY * (HXY2 * F(I,J) + HX2 * (U(I,J-1) + U(I,J+1))
+ HY2 * (U(I-1,J) + U(I+1,J)))
30  CONTINUE
ERR = 0.0
DO 40 J=JST,JEND
DO 40 I=IST,IEND
U(I,J) = U0(I,J)
ERR = ERR + (U(I,J) - US(I,J)) * * 2
40  CONTINUE
ERR0 = ERR
CALL MPI_ALLREDUCE(ERR0,ERR,1,MPI_REAL,MPI_SUM,
MPI_COMM_WORLD,IERR)
ERR = DSRQT(ERR/(IM * JM))
!
IF(ERR.GT.1.E-3)THEN      ! 收敛性判断。
!
! 交换定义在辅助网格结点上的近似解。
CALL MPI_SEND(U(1,1),1,VTYPE,MYLEFT,NITER+100,
MPI_COMM_WORLD,IERR)    ! 发送左边界。
CALL MPI_SEND(U(IEND,1),1,VTYPE,MYRIGHT,NITER+100,
MPI_COMM_WORLD,IERR)    ! 发送右边界。
CALL MPI_SEND(U(1,1),1,HTYPE,MYLOW,NITER+100,
MPI_COMM_WORLD,IERR)    ! 发送下边界。
CALL MPI_SEND(U(1,JEND),1,HTYPE,MYUPPER,NITER+100,
MPI_COMM_WORLD,IERR)    ! 发送上边界。
CALL MPI_RECV(U(IEND+1,1),1,VTYPE,MYRIGHT,NITER+100,
MPI_COMM_WORLD,STATUS,IERR)    ! 接收右边界。
CALL MPI_RECV(U(0,1),1,VTYPE,MYLEFT,NITER+100,
MPI_COMM_WORLD,STATUS,IERR)    ! 接收左边界。
CALL MPI_RECV(U(1,JEND+1),1,HTYPE,MYUPPER,NITER+100,
MPI_COMM_WORLD,STATUS,IERR)    ! 接收上边界。
CALL MPI_RECV(U(1,0),1,HTYPE,MYLOW,NITER+100,
MPI_COMM_WORLD,STATUS,IERR)    ! 接收下边界。
GOTO 100      ! 没有收敛，进入下次迭代
ENDIF

```

```

!
IF(MYRANK.EQ.0) PRINT *, "!!! Successfully converged after", NITER,
    "iterations."
! 输出近似解(忽略)。
!
CALL MPI_FINALIZE(IERR)
END

```

在上例中,我们固定了该 MPI 程序产生的进程个数 $NP = NPY \times NPX$,这样做的一个主要目的,是为了使各进程的内存空间仅为相应串行程序的 $1 / (NPY * NPX)$,从而使原来串行程序在单机上由于内存不够而无法计算的问题,通过多机并行计算而成为可能。当然,这样做也带来一些不便,例如它要求 MPI 运行命令“mpirun-np xxx”中的参数 xxx 等于 $NPY \times NPX$,且如果参数 NPY 和 NPX 被改变后,必须重新编译该程序。

11.3 MPI 并行程序的改进

在例 11.1 中,交换定义在辅助网格结点近似解的消息传递 8 条语句,是该 MPI 程序的关键。但是,由第三章 3.4 节的讨论可知,它们是不安全的,因为在某些并行机上,当消息较长时(例如大于 16KB),可能由于 MPI 系统缓存区大小的限制,而导致执行该 MPI 程序进程的死锁。因此,我们不妨将它们替换成如下的非阻塞通信函数。

例 11.2 改进一: 非阻塞通信

```

.....
INTEGER REQ(8), STATUS(MPI_STATUS_SIZE,8)
.....
!
! 非阻塞地交换定义在辅助网格结点上的近似解
CALL MPI_ISEND(U(1,1),1,VTYPE,MYLEFT,NITER+100,
    MPI_COMM_WORLD,REQ(1),IERR) ! 发送左边界。
CALL MPI_ISEND(U(IEND,1),1,VTYPE,MYRIGHT,NITER+100,
    MPI_COMM_WORLD,REQ(2),IERR) ! 发送右边界。
CALL MPI_ISEND(U(1,1),1,HTYPE,MYLOW,NITER+100,
    MPI_COMM_WORLD,REQ(3),IERR) ! 发送下边界。
CALL MPI_ISEND(U(1,JEND),1,HTYPE,MYUPPER,NITER+100,
    MPI_COMM_WORLD,REQ(4),IERR) ! 发送上边界。
CALL MPI_IRecv(U(IEND+1,1),1,VTYPE,MYRIGHT,NITER+100,
    MPI_COMM_WORLD,REQ(5),IERR) ! 接收右边界。
CALL MPI_IRecv(U(0,1),1,VTYPE,MYLEFT,NITER+100,
    MPI_COMM_WORLD,REQ(6),IERR) ! 接收左边界。
CALL MPI_IRecv(U(1,JEND+1),1,HTYPE,MYUPPER,NITER+100,
    MPI_COMM_WORLD,REQ(7),IERR) ! 接收上边界。

```

```

CALL MPI_IRECV(U(1,0),1,HTYPE,MYLOW,NITER+100,
               MPI_COMM_WORLD,REQ(8),IERR) !接收下边界。
CALL MPI_WAITALL(8,REQ,STATUS) !阻塞式等待消息传递的结束。
!
.....

```

在例 11.1 中,我们可以将标号为 30 的循环分裂成两个部分,其中一个部分需要辅助网格点上的近似解,而另一个部分不需要辅助网格点上的近似解。这样,为了改进该 MPI 程序的并行性能,我们可以将后一个部分的计算与例 11.2 的非阻塞消息传递重叠起来,从而达到屏蔽网络延迟的目的。具体改进如下:

例 11.3 改进二:重叠通信与计算

```

!
NITER = 0
100 CONTINUE
NITER = NITER + 1
!
! 非阻塞地交换定义在辅助网格结点上的近似解。
CALL MPI_ISEND(U(1,1),1,VTYPE,MYLEFT,NITER+100,
                MPI_COMM_WORLD,REQ(1),IERR) !发送左边界。
CALL MPI_ISEND(U(IEND,1),1,VTYPE,MYRIGHT,NITER+100,
                MPI_COMM_WORLD,REQ(2),IERR) !发送右边界。
CALL MPI_ISEND(U(1,1),1,HTYPE,MYLOW,NITER+100,
                MPI_COMM_WORLD,REQ(3),IERR) !发送下边界。
CALL MPI_ISEND(U(1,JEND),1,HTYPE,MYUPPER,NITER+100,
                MPI_COMM_WORLD,REQ(4),IERR) !发送上边界。
CALL MPI_IRECV(U(JEND+1,1),1,VTYPE,MYRIGHT,NITER+100,
                MPI_COMM_WORLD,REQ(5),IERR) !接收右边界。
CALL MPI_IRECV(U(0,1),1,VTYPE,MYLEFT,NITER+100,
                MPI_COMM_WORLD,REQ(6),IERR) !接收左边界。
CALL MPI_IRECV(U(1,JEND+1),1,HTYPE,MYUPPER,NITER+100,
                MPI_COMM_WORLD,REQ(7),IERR) !接收上边界。
CALL MPI_IRECV(U(1,0),1,HTYPE,MYLOW,NITER+100,
                MPI_COMM_WORLD,REQ(8),IERR) !接收下边界。
!
DO 30 J = JST+1,JEND-1
DO 30 I = IST+1,IEND-1
U0(I,J) = RXY * (HXY2 * F(I,J) + HX2 * (U(I,J-1) + U(I,J+1))
                  + HY2 * (U(I-1,J) + U(I+1,J)) )
30 CONTINUE
CALL MPI_WAITALL(8,REQ,STATUS) !阻塞式等待消息传递的结束。
DO 31 J = JST,JEND

```

```

DO 31 I=IST,IEND
  IF (I.NE.IST.OR.I.NE.IEND.OR.J.NE.JST.OR.J.NE.JEND) GOTO 31
    U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1)
                                +HY2*(U(I-1,J)+U(I+1,J)) ) )
31   CONTINUE
!
! 收敛性检查。
.....
IF(ERR.GT.1.E-3)GOTO 100
!

```

在例 11.1 中, 各进程按自然顺序(先沿 x 方向, 后沿 y 方向)确定与它相邻的 4 个进程的序号(MYLEFT, MYRIGHT, MYLOW, MYUPPER), 以及它自己所处的行主序位置(MEPY, MEPX)。实际上, 这些进程按区域分解策略可以很自然地映射到 $NPY \times NPX$ 的二维 Cartesian 拓扑结构中, 而(MEPY, MEPX) 就是各进程在该拓扑结构中的坐标。因此, 我们可以从通信器 MPI_COMM_WORLD 出发, 建立二维 Cartesian 拓扑结构, 从而方便地确定各进程的相邻关系, 并使得之后的所有 MPI 消息传递均基于该拓扑结构而进行。

例 11.4 改进三: 二维 Cartesian 拓扑结构

```

!
.....
INTEGER COMM,DIMS(2),COORD(2)
LOGICAL PERIOD(2),REORDER
!
.....
DIMS(1)=NPY          ! 拓扑结构中  $x$  方向的进程个数。
DIMS(2)=NPX          ! 拓扑结构中  $y$  方向的进程个数。
PERIOD(1)=.FALSE.    ! 沿  $x$  方向, 拓扑结构非周期连接。
PERIOD(2)=.FALSE.    ! 沿  $y$  方向, 拓扑结构非周期连接。
REORDER=.TRUE.        ! 在新通信器中, 进程重新排列序号。
CALL MPI_CART_CREATE(MPI_COMM_WORLD,2,DIMS,PERIOD,REORDER,
                      COMM,IERR)
CALL MPI_COMM_RANK(COMM,MYRANK,IERR)
CALL MPI_CART_COORDS(COMM,MYRANK,2,COORD,IERR)
MEPY=COORD(1)
MEPX=COORD(2)
COORD(2)=MEPX-1
CALL MPI_CART_COORDS(COMM,MYLEFT,2,COORD,IERR)
COORD(2)=MEPX+1
CALL MPI_CART_COORDS(COMM,MYRIGHT,2,COORD,IERR)
COORD(2)=MEPX

```

```

COORD(1) = MEPY - 1
CALL MPI_CART_COORDS(COMM, MYLOW, 2, COORD, IERR)
COORD(1) = MEPY + 1
CALL MPI_CART_COORDS(COMM, MYUPPER, 2, COORD, IERR)
.....
!
```

在例 11.1 中, 我们忽略了近似解的输出, 这里, 我们用第八章介绍 MPI 并行 I/O 函数实现近似解的并行输出。此外, 我们要求输出的近似解按自然顺序排列, 且包含物理边界结点。

例 11.5 改进四: 并行 I/O

```

!
.....
INTEGER FILE
INTEGER FILETYPE, MEMTYPE, GSIZE(2), LSIZE(2), START(2)
.....
!
! 近似解输出。
GSIZE(1) = IM + 1
GSIZE(2) = JM - 1
LSIZE(1) = IML
IF(MEPX.EQ.0) LSIZE(1) = LSIZE(1) + 1
LSIZE(2) = JML
IF(MEPY.EQ.0) LSIZE(2) = LSIZE(2) + 1
START(1) = IML * MEPX
IF(MEPX.NE.0) START(1) = START(1) + 1
START(2) = JML * MEPY
IF(MEPY.NE.0) START(2) = START(2) + 1
!
! 定义局部子数组数据类型。
CALL MPI_TYPE_CREATE_SUBARRAY(2, GSIZ, LSIZE, START,
                               MPI_ORDER_F, MPI_REAL, FILETYPE, IERR)
CALL MPI_COMMIT(FILETYPE, IERR)
!
! 打开二进制文件。
CALL MPI_FILE_OPEN(COMM, "result.dat",
                   MPI_MODE_CREATE + MPI_MODE_WRONLY,
                   MPI_INFO_NULL, FILE, IERR)
CALL MPI_FILE_SET_VIEW(FILE, 0, MPI_REAL, FILETYPE,
                      "NATIVE", MPI_INFO_NULL, IERR)
!
```

```

! 定义数据类型,描述子数组在内存中的分布。
GSIZE(1) = IML + 2
GSIZE(2) = JML + 2
START(1) = 1
IF(MEPX.EQ.0)START(1) = 0
START(2) = 1
IF(MEPY.EQ.0)START(2) = 0
CALL MPI_TYPE_SUBARRAY(2,GSIZE,LSIZE,START,MPI_ORDER_F,MPI_REAL,
    MEMTYPE,IERR)
CALL MPI_TYPE_COMMIT(MEMTYPE,IERR)
!
! 输出近似解(含物理边界结点)。
CALL MPI_FILE_WRITE_ALL(FILE,U,1,Mem_TYPE,STATUS,IERR)
CALL MPI_FILE_CLOSE(FILE,IERR)
!
.....
!
```

至此,例 11.2 例 11.5 分别从非阻塞通信、重叠通信与计算、拓扑结构和并行 I/O 四个方面,利用第三章~第八章介绍的相应 MPI 函数,依次改进了例 11.1 中 MPI 程序的功能和并行性能。通过该应用示例,读者可以较好地将各章介绍的 MPI 函数联系在一起,解决实际问题。

第十二章 MPI 2.0 的新特征

本章在前面各章介绍的 MPI 1.2 版本的基础上,简单介绍了 MPI 2.0 版本的一些新特征,主要包括单边通信和动态进程管理。其中,这些特征的部分函数已经在具体并行机上实现,或者即将被各类并行机所支持。

这里需要说明的是,本章只介绍与这些新特征相关的基本函数及其功能,而不涉及性能更强、使用更方便的其他函数,有兴趣的读者请参考相关文献。

12.1 单边通信

在第一章,我们曾经指出,执行 MPI 程序的各个进程之间是分布式存储的,它们之间的数据交换只能通过消息传递来实现。MPI 1.0 版本正是建立在这一严格的分布式存储消息传递并行程序设计模式的基础上。依据该版本,如果数据从一个进程的内存空间拷贝到另一个进程的内存空间,则前面的进程必须显式地提交一个消息发送操作,而后面的进程必须显式地提交一个相匹配的消息接收操作,也就是说,用户必须显式地规定各个进程之间的通信和同步,这并不是一件轻松的工作。

为了进一步方便并行程序的设计,MPI 2.0 版推广了 MPI 1.0 版,它允许进程通过 `get`, `put`, `accumulate` 等操作,直接读写另一个进程的内存数据单元,称之为远程内存访问 (RMA: Remote Memory Access)。由于远程内存访问只需要一个进程的参与,因此我们称之为单边通信 (One Side Communications)。同时,为便于区别,我们称前面各节介绍的 MPI 进程间通信为双边通信 (Two Sides Communications),因为它们至少需要两个以上进程的直接参与。

这里,我们需要指出的是,相对于显式的消息发送/接收操作,MPI 2.0 版的远程内存访问并不能改进 MPI 程序的性能,而它们的主要益处是简化并行程序的设计。当然,在某些对远程内存访问提供直接硬件支持的并行机上,单边通信的性能可能比双边通信的性能要高。

为了实现单边通信,每个进程首先必须将自己的某段内存空间公布给通信器包含的所有进程,并允许它们直接访问这一段内存空间。MPI 系统将维护和管理各个进程公布的内存空间,并称之为各个进程的远程内存访问窗口 (RMA Window)。如图 12.1 所示,进程 0 将其阴影部分的内存空间公布给进程 1,这样,进程 1 就可以通过 `get` 操作,直接读取该段内存空间包含的所有数据,而不需要进程 0 显式地发送这些数据;反之,进程 1 也将其阴影部分的内存空间公布给进程 0,这样,进程 0 就可以通过 `put` 操作,将数据直接写入该段内存空间,或者通过 `accumulate` 操作,将数据直接累加到该段内存空间的数据单元中,而不需要进程 1 显式地接收这些数据。于是,进程 0 和进程 1 的内存空间实际上分解成了两个部分,其中一个部分是只有自己才能访问的内存空间,称之为局部内存空间 (Local Address Space),另一个部分就是其他进程能通过操作 `get`, `put` 和 `accumulate` 直接

访问的内存空间，即远程内存访问(RMA)窗口。

RMA 窗口由函数 MPI_WIN_CREATE 创建。

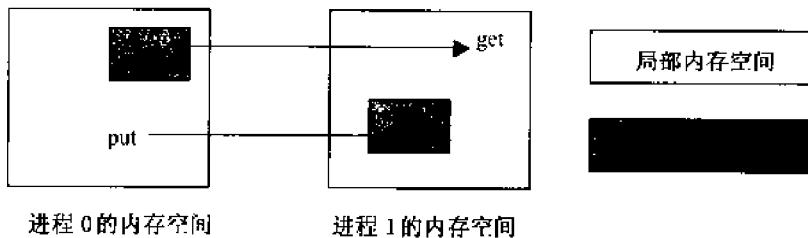


图 12.1 两个进程的 RMA 窗口示意图，
其中，阴影部分为 RMA 窗口

/* MPI RMA 窗口创建函数，创建一个 RMA 窗口。

```
MPI_Win_create (base, size, disp_unit, info, comm, win)
    IN      base      包含在窗口中的内存空间起始地址
    IN      size      包含在窗口中的内存空间大小(字节为单位)
    IN      disp_unit 窗口中连续数据单元之间的偏移量(字节为单位)
    IN      info      提示信息
    IN      comm      通信器
    OUT     win       RMA 窗口

C      int MPI_Win_create (void * base, MPI_Aint size, int disp_unit, MPI_Info info,
                         MPI_Comm comm, MPI_Win * win)

Fortran MPI_WIN_CREATE (BASE,SIZE,DISP_UNIT,INFO,COMM,
                        WIN,IERROR)
<type>   BASE(*)
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
INTEGER DISP_UNIT,INFO,COMM,WIN,IERROR
```

函数 MPI_WIN_CREATE 需要通信器 comm 包含的所有进程的共同参与，并创建一个被所有进程共享的 RMA 窗口 win。任意进程均可以自由地访问该窗口包含的内存地址空间。其中，参数 base 表示包含在窗口中的内存起始地址，参数 size 表示内存空间的大小，参数 disp_unit 表示窗口中连续数据单元之间的偏移量，参数 info 是提示信息。

函数 MPI_WIN_CREATE 属于聚合通信函数，需要通信器中所有的进程的共同参与。

例 12.1 函数 MPI_WIN_CREATE 应用示例

```
integer n,nwin
call MPI_Comm_rank( COMM,myrank,ierr)
if(myrank.eq.0)then
  call MPI_Win_create( n,4,1,MPI_INFO_NULL,COMM,nwin,ierr)
```

```

else
    call MPI_Win_create( MPI_BOTTOM, 0, 1, MPI_INFO_NULL, COMM, nwin, ierr)
endif

```

在上例中, 进程 0 将创建一个 RMA 窗口, 包含一个整数 n , 而其他进程则提供参数 $base = MPI_BOTTOM$, $size = 0$, 表示不创建窗口。进程 0 创建的窗口将被所有进程共享。但是, 各个进程为了正确地访问窗口 $nwin$, 必须在具体访问操作执行的前后, 各调用一次函数 MPI_Win_fence 来保证进程间的一种特殊同步机制。特别地, MPI 系统称两个函数之间的那段程序为该进程的 RMA 访问区间 (RMA access epoch)。

```

/* RMA 访问区间创建函数, 连续的两次函数调用语句之间的程序构成一个 RMA
/* 访问区间。

```

MPI_Win_fence (assert, win)		
IN	assert	进程间 RMA 的同步机制
IN	win	RMA 窗口
C	int MPI_Win_fence (int assert, MPI_Win * win)	
Fortran	MPI_WIN_FENCE (ASSERT, WIN, IERROR)	
	INTEGER ASSERT, WIN, IERROR	

定义好 RMA 访问区间后, 各个进程就可以直接访问 RMA 窗口包含的数据单元了。MPI 系统主要提供 3 个函数 MPI_GET , MPI_PUT 和 MPI_ACCUMULATE , 分别执行 RMA 读/写操作。

函数 MPI_WIN_FENCE 的参数 ASSER 下通常可取为 0 (参考例 12.2)。当然, MPI 系统还提供其他四种同步机制, 这里不一一介绍。

```

/* RMA 读操作

```

MPI_Get (rbuf, rcount, rdtype, target, sbuf, scount, sdtype, win)		
IN	rbuf	数据接收缓存区起始地址
IN	rcount	接收的数据单元个数
IN	rdtype	接收的数据单元类型
IN	target	创建 RMA 窗口 win 的进程序号
IN	sbuf	读取的数据在 RMA 窗口 win 的起始地址(字节为单位)
IN	scount	将被读取的数据单元个数
IN	sdtype	将被读取的数据单元类型
IN	win	RMA 窗口
C	int MPI_Get (void * rbuf, int rcount, MPI_Datatype rdtype, int target,	
	MPI_Aint sbuf, int scount, MPI_Datatype sdtype, MPI_Win win)	
Fortran	MPI_Get (RBUF, RCOUNT, RDTYPE, TARGET, SBUF, SCOUNT, SDTYPE,	
	WIN, IERROR)	
	<type>	RBUF(*)
	INTEGER(KIND=MPI_ADDRESS_KIND)	SBUF

```

INTEGER RCOUNT, RDTYPE, TARGET, SCOUNT, SDTYPE,
WIN, IERROR

```

函数 MPI _ GET 等价于执行一个标准模式非阻塞点对点通信。其中,序号为 target 的进程非阻塞发送消息 MPI _ ISEND (sbuf, scount, sdtype, ……), 而调用该函数的进程非阻塞接收消息 MPI _IRECV (rbuf, rcount, rdtype, ……)。

/* RMA 写函数。

```

MPI _ Put ( sbuf, scount, sdtype, target, rbuf, rcount, rdtype, win)
    IN      sbuf      待写入窗口的数据缓存区起始地址
    IN      scount    待写入的数据单元个数
    IN      sdtype    待写入的数据单元类型
    IN      target    创建 RMA 窗口 win 的进程序号
    IN      rbuf      写入的数据在 RMA 窗口 win 的起始地址(字节为单位)
    IN      rcount    写入的数据单元个数
    IN      rdtype    写入的数据单元类型
    IN      win       RMA 窗口

C     int MPI _ Put ( void * sbuf, int scount, MPI _ Datatype sdtype, int target,
                    MPI _ Aint rbuf, int rcount, MPI _ Datatype rdtype, MPI _ Win win)

Fortran MPI _ PUT ( SBUF, SCOUNT, SDTYPE, TARGET, RBUF, RCOUNT, RDTYPE,
                     WIN, IERROR )
<type>   SBUF( * )
INTEGER(KIND= MPI _ ADDRESS _ KIND) RBUF
INTEGER  SCOUNT, SDTYPE, TARGET, RCOUNT, RDTYPE,
          WIN, IERROR

```

函数 MPI _ PUT 等价于执行一个标准模式非阻塞点对点通信。其中,调用该函数的进程非阻塞发送消息 MPI _ ISEND (sbuf, scount, sdtype, ……), 而序号为 target 的进程非阻塞接收消息 MPI _IRECV (rbuf, rcount, rdtype, ……)。

/* RMA 写函数。

```

MPI _ Accumulate ( sbuf, scount, sdtype, target, rbuf, rcount, rdtype, op, win)
    IN      sbuf      待写入窗口的数据缓存区起始地址
    IN      scount    待写入的数据单元个数
    IN      sdtype    待写入的数据单元类型
    IN      target    创建 RMA 窗口 win 的进程序号
    IN      rbuf      写入的数据在 RMA 窗口 win 的起始地址(字节为单位)
    IN      rcount    写入的数据单元个数
    IN      rdtype    写入的数据单元类型
    IN      op        操作
    IN      win       RMA 窗口

C     int MPI _ Accumulate ( void * sbuf, int scount, MPI _ Datatype sdtype, int target,
                           MPI _ Aint rbuf, int rcount, MPI _ Datatype rdtype, MPI _ Op op, MPI _ Win win)

```

```

        MPI_Aint rbuf, int recount, MPI_Datatype rdtype,
        MPI_Op op, MPI_Win win)

Fortran MPI_ACCUMULATE (SBUF, SCOUNT, SDTYPE, TARGET, RBUF, RCOUNT,
                        RDTYPE, OP, WIN, IERROR)
<type>    SBUF(*)
INTEGER(KIND=MPI_ADDRESS_KIND) RBUF
INTEGER SCOUNT, SDTYPE, TARGET, RCOUNT, RDTYPE,
          OP, WIN, IERROR

```

函数 MPI_ACCUMULATE 在函数 MPI_PUT 的基础上, 规定: 数据写入 RMA 窗口前, 必须先与被覆盖的地址空间数据单元的值执行一次操作 op, 然后将操作结果写入地址空间。其中, 操作 op 可以是 MPI 系统定义的归约操作 (参考表 5.1), 例如 MPI_SUM, MPI_MAX, MPI_MIN 等, 也可以是用户自定义的规约操作。

例 12.2 RMA 应用示例: 并行计算 π 值

```

integer n, nwin, nproc, piwin
call MPI_Comm_size( COMM, nproc, ierr)
call MPI_Comm_rank( COMM, myrank, ierr)
!
! 进程 0 创建两个 RMA 窗口
if(myrank.eq.0)then
    call MPI_Win_create( n, 4, 1, MPI_INFO_NULL, COMM, nwin, ierr)
    call MPI_Win_create( pi, 8, 1, MPI_INFO_NULL, COMM, piwin, ierr)
else
    call MPI_Win_create( MPI_BOTTOM, 0, 1, MPI_INFO_NULL, COMM, nwin, ierr)
    call MPI_Win_create( MPI_BOTTOM, 0, 1, MPI_INFO_NULL, COMM, piwin, ierr)
endif
!
10 continue
if(myrank.eq.0)then
    print *, "+++ please enter the number of intervals: (0 quit)"
    read(*,n)
    pi = 0.0
endif
!
! 定义 RMA 访问区间, 并执行 RMA 操作。
call MPI_Win_fence(0, nwin, ierr)
if(myrank.ne.0)then
    call MPI_Get(n, 1, MPI_INTEGER, 0, 0, 1, MPI_INTEGER, nwin, ierr)
    ! 在 RMA 区间中, 从进程 0 读取一个整数 n,
endif
call MPI_Win_fence(0, nwin, ierr)

```

```

if(n.ne.0)then
    h = 1.0/n
    sum = 0.0
    do k = myrank + 1, n, nproc
        x = h * (k - 0.5)
        sum = sum + 4.0/(1.0 + x * x)
    enddo
    pil = h * sum
    ! Call MPI_Win_fence(0, piwin, ierr)
    call MPI_Accumulate(pil, 1, MPI_DOUBLE_PRECISION, 0, 0, 1,
                        MPI_DOUBLE_PRECISION, MPI_SUM, piwin, ierr)
    ! 在 RMA 区间中, 执行 Accumulate 操作。
    Call MPI_Win_fence(0, piwin, ierr)
    !
    goto 10
endif
!
! 释放 RMA 窗口。
call MPI_Win_free(nwin, ierr)
call MPI_Win_free(piwin, ierr)
!
call MPI_Finalize()

```

12.2 动态进程管理

本书在第二章曾指出, MPI 1.2 版目前只支持严格的静态进程管理, 即执行 MPI 程序的进程个数必须在程序初始启动时确定, 而且程序在执行过程中, 不允许进程个数的动态增加和减少。尽管这种进程静态管理简化了并行程序的设计和 MPI 系统的具体实现, 有利于提高 MPI 程序的性能, 且能满足大多数应用的需求, 但是, 某些特殊的应用问题仍然需要灵活地动态进程管理。例如, 假设应用问题的计算量是在程序执行过程中动态创建的, 而所需的计算量一旦产生, 就必须动态产生合适的进程个数来执行。再如, 假设应用问题是两个 MPI 并行程序组成, 一般情形下, 这两个并行程序可以完全独立地执行, 但在某些特殊时刻, 需要将这两个并行程序联接在一起, 并交换信息。

MPI 2.0 版支持动态进程管理。首先, 应用程序可以调用函数 MPI_COMM_SPAWN 来动态产生通信器, 该通信器包含所有新产生的进程; 然后, 在新通信器和旧通信器之间, 通过调用第六章 6.3 节介绍的 MPI 函数, 创建新的域间通信器, 从而可以组织消息传递。

在大多数并行机上, MPI 2.0 版的动态进程管理还没有具体实现。因此, 本书这里不再讨论有关动态进程管理的各种函数, 有兴趣的读者可以到 MPI 网站下载相关资料。

第十三章 MPI 的现状与发展

本章讨论目前 MPI 系统标准版本在具体并行机上的实现状况,以及它们的进一步发展。

尽管 MPI 2.0 版本的标准早在 1998 年就被提出,但是,由于它远比 MPI 1.0 版本复杂,实现的难度更大,因此,许多并行机只实现了 MPI 2.0 版本的部分函数,例如第八章介绍的 MPI 并行 I/O 函数。

目前,在具体商用并行机中,日本的 Fujitsu 提供了 MPI 2.0 版本的一个较为完备的具体实现;Compaq、HP、IBM、NEC 和 SGI 等各大并行机生产商都从并行 I/O 入手,正在逐步地实现 MPI 2.0 版本;HP 并行机支持远程内存访问和 MPI 函数的多线程调用;Edinburgh 并行计算研究中心在 SGI Cray/T3D 上也实现了远程内存访问。

目前,使用最广泛的免费 MPI 软件主要有 MPICH 和 LAM-MPI 两种,它们比较适合微机机群,也可以安装在许多并行机中。目前 MPICH 和 LAM-MPI 均支持 MPI 2.0 版本的并行 I/O 函数,且二者都提供了基于 C++ 语言的函数调用标准接口和函数的具体实现。

在 MPI 2.0 版本之后,是否还可能发展 MPI 3.0 版本,目前我们还无法确定。但是,消息传递 MPI 系统的设计和实现肯定不会停留在 2.0 版本。具体地, MPI 系统将会在如下几个方面进一步加强功能:

- (1) 提供更丰富的远程内存访问操作;
- (2) 硬件和软件环境将提供更有效的 MPI 进程的多线程执行;
- (3) 基于程序设计语言,例如 C++、COBOL 等,提出和实现 MPI 标准;
- (4) 多个 MPI 并行程序之间的互操作性;
- (5) VIA-MPICH,即基于 VIA 技术实现 MPI 标准,它能大幅度降低点对点通信延迟;
- (6) 在具体 MPI 实现和应用程序之间,建立一个中间层,基于该中间层提供的各项服务,用户可以直接组织应用程序的并行化,从而可以屏蔽显式的消息传递通信,达到简化并行程序设计的目的。例如,美国 Argonne 实验室研制的并行可扩展科学计算工具箱(PETSc)就是一个典型的例子;
- (7) 实时 MPI(Real-Time MPI)标准,使得 MPI 系统能适应某些实时计算问题的需求。

不管怎样,MPI 系统是一个不断完善和发展的消息传递并行程序设计平台,在今后较长的一段时间内,它将是最具有竞争力的并行程序设计平台之一。

附录

附录 A MPI 程序的编译和运行

本附录介绍 MPI 程序的编译和运行。特别地, 我们假设 MPICH 已经在具体并行机中安装, 且 MPI 系统执行命令 (例如 mpif77, mpicc, mpirun 等) 的路径也设置在用户的省缺路径中 (参考附录 B)。

A.1 MPI 程序的编译

在一般的并行计算环境 (如机群系统) 中, MPI 程序可用如下命令编译:

```
mpif77 [选项] Fortran 源程序  
mpicc [选项] C 源程序
```

其中, “选项”类同于一般的 f77 和 cc 编译选项。例如, 假设 foo.f 为某 MPI Fortran 源程序, 则编译命令:

```
mpif77 -o foo -O2 foo.f
```

表示二级优化编译 foo.f, 且与 MPI 函数库连接, 形成可执行目标代码 foo。

在具体并行机 (如 SGI 系列共享存储并行机) 上, MPI 系统函数库的使用与其他函数库相同, 编译命令为:

```
f77 [选项] Fortran 源程序 -lmpi  
cc [选项] C 源程序 -lmpi
```

如果应用程序采用 MPI 和 OpenMP 并行编程, 则编译命令为:

```
f77 [选项]-mp Fortran 源程序 -lmpi  
cc [选项]-mp C 源程序 -lmpi
```

A.2 MPI 程序的运行

在一般的并行计算环境 (如机群系统) 中, MPI 程序可用如下命令来运行:

```
mpirun [mpirun 选项] MPI 可执行程序名 <输入数据文件> 输出数据文件
```

其中, mpirun 选项包括:

-arch	结构名	: 在哪种结构的处理机上启动进程;
-h		: mpirun 使用帮助;
-machine	机器名	: 在指定的处理机上启动进程;
-machinefile	机器文件名	: 在机器文件中列出的处理机上启动进程;
-np	进程个数	: 启动的进程个数;
-nolocal		: 不在本地处理机上启动进程;
-pg		: 使用进程组文件启动进程, 为缺省方式;
-e		: 使用 execer 启动进程;
-leave_pg		: 程序执行完毕, 保存进程组文件于当前目录;

-p4pg 进程组文件 :按进程组文件中列出的方式启动进程;
-dbx :如果可能,在 dbx 调试环境下启动进程;
-gdb :如果可能,在 gdb 调试环境下启动进程。

其中,进程组文件由具有如下格式的行组成:

处理机名 进程个数 MPI 可执行程序名 [<login>]

例如某进程组文件由下面 4 行构成:

```
sun1 0 /users/jones/myprog  
sun2 1 /users/jones/myprog  
sun3 1 /users/jones/myprog .  
hp1 1 /home/mb3/myprog mbj
```

则该文件表示 MPI 程序将由 4 个进程执行,其中,sun1 为本地主机,在当前帐号上启动一个进程;sun1 ~ sun3 在同样的帐号上各启动一个进程 myprog;hp1 在帐号 mbj 上启动进程 myprog。特别地,如果在本地主机上列出的进程数为 k,则表示将在本地主机上启动 k + 1 个进程;而对其他非本地主机,如果列出的进程数为 k,则表示启动 k 个进程。

命令:

```
mpirun -arch sun4 -np 2 -arch rs6000 -np 3 program
```

表示在 sun4 上启动 2 个进程,在 rs6000 上启动 3 个进程。命令:

```
mpirun -p4pg pgfile myprog
```

表示以进程组文件 pgfile 来启动进程。命令:

```
mpirun -machinefile host -np 4 myprog
```

表示在文件 host 包含的机器名中选择前面 4 台,启动 4 个 myprog 进程。

在具体并行机(如 SGI 系列并行机),MPI 程序的运行命令可得到简化,一般为:

```
mpirun -np 进程数 MPI 可执行程序 <输入数据文件> 输出数据文件
```

附录 B MPICH 的安装

目前,MPICH 是最流行的基于网络并行计算环境的 MPI 标准的具体实现,许多商用 MPI 软件均是在它的基础上,结合具体并行机的高性能特征而研制的。MPICH 的软件及其相关文件可从 MPI 网站 <http://www.mcs.anl.gov/mpi>, 或者匿名 ftp 到地址 info.mcs.anl.gov 的目录 pub/mpi 中自由下载。目前,该网站提供 MPICH 1.2 版,它支持本书第三章~第八章介绍的所有 MPI 函数,支持 Fortran, C 和 C++ 语言,并可安装在当前流行的几乎所有类型的并行机上。但是,在各公司研制的商用并行机上,MPICH 的性能要低于并行机厂商提供的 MPI 系统的性能。

本附录侧重于微机机群 Linux 操作系统,讨论 MPI 1.2 版的安装。

B.1 快速安装与使用

在微机机群 Linux 操作系统 Bash Shell 环境中,可按如下步骤快速安装 MPI 1.2 版。

- (1) 从网站 <http://www.mcs.anl.gov/mpi> 中下载 MPICH 1.2 自由软件;
- (2) 解开 MPICH 1.2 软件包到当前目录,如果软件由 compress 压缩,则用命令“zcat mpich.tar.Z | tar xvf -”;如果软件由 gzip 压缩,则用命令“gzip -dc mpich.tar.

- ```
gz | tar xvf ~";
(3) cd mpich;进入 MPICH 主目录;
(4) ./configure --with-arch=LINUX-comm=ch_p4 --with-device=ch_p4 -prefix
= 安装目录 (例如/usr/local/mpi);结合具体并行机特征,配置 MPI 系统;
(5) make>make.log:编译形成 MPI 系统函数库,需要数分钟的时间;
(6) 编辑文件“mpich/util/machines/machines.LINUX”,在该文件中一行一个地列出
微机机群的所有计算结点的名字,具体格式请参考文件“mpich/util/machines/
machines.sample”;同时,编辑文件“/etc/hosts.equiv”或者“$HOME/.rhosts”,在该文件中一行一个地列出微机机群的所有计算结点的名字,表示允许各个计算结点之间的远程 shell 命令的执行;
(7) cd examples/basic
(8) make cpi
(9) mpirun -np 4 cpi;该程序的成功执行将标志此次安装是成功的;
(10) make install prefix=安装目录 (例如/usr/local/mpi);形成最终的 MPI 系统,并
将其安装在 prefix 指定的目录中,供所有用户共享。
```

至此,为了使用 MPI 软件,用户还需要将 MPI 的执行命令 (如 mpif77, mpicc, mpirun 等) 和联机帮助的路径分别加入到用户帐号环境变量 PATH 和 MANPATH 指定的缺省路径中。具体是,假设 MPI 被安装在目录/usr/local/mpi 中,在 C Shell 环境中,可用如下命令设置环境变量:

```
setenv PATH $PATH:/usr/local/mpi/bin
setenv MANPATH $MANPATH:/usr/local/mpi/man
```

而在 Bash Shell 环境中,可用如下命令设置环境变量:

```
PATH=$PATH:/usr/local/mpi/bin
MANPATH=$MANPATH:/usr/local/mpi/man
export PATH MANPATH
```

其实,用户可以采用两种办法将以上环境变量加入到自己帐号的缺省路径中,第一种是用户自己修改环境变量:对 C Shell 环境,可以修改文件 \$HOME/.cshrc,而对 Bash Shell 环境,可以修改文件 \$HOME/.profile;第二种是系统管理员为所有用户修改环境变量,对 C Shell 环境,修改文件/etc/csh.csh.cshrc,对 Bash Shell,修改文件“/etc/profile”。

路径设置完毕,用户就可以自由地使用 MPICH 了。在其他并行机上,也可以类似地安装 MPI 软件,只是第 4 个步骤和第 6 个步骤不同。其中,第 6 个步骤只需将该并行机的名字列在相应文件“mpich/util/machines/machines.XXX”中,而后缀 XXX 代表配置选项 --with-arch 中指定的并行机结构;第 4 个步骤必须结合具体并行机的特征来选择参数,我们将在下一节讨论。

## B.2 配置 MPICH

安装 MPI 软件的关键,是如何结合具体并行机的高性能特征来配置 MPI 系统,具体反映在如何为命令 configure 配置各个选项参数。

在 MPICH 主目录下,执行命令:

```
./configure -usage
```

可以列出 MPI 系统配置命令 configure 的所有选项：

```
Configuring with args -usage
Configuring MPICH Version 1.2.0 of : 1999/12/02 13:22:11
Usage: ./configure [--with-arch=ARCH_TYPE][-comm=COMM_TYPE]
 [--with-device=DEVICE]
 [--with-mpe][--without-mpe]
 [--disable-f77][--disable-f90][--with-f90nag][--with-f95nag]
 [--disable-f90modules]
 [--enable-c++][--disable-c++]
 [--enable-mpedbg][--disable-mpedbg]
 [--enable-devdebug][--disable-devdebug]
 [--enable-debug][--disable-debug]
 [--enable-short-longs][--disable-short-longs]
 [--with-jumpshot[-JUMPSHOT_OPTS]]
 [-prefix=INSTALL_DIR]
 [-c++[=C++_COMPILER]][noc++]
 [-opt=OPTFLAGS]
 [-cc=C_COMPILER][-fc=FORTRAN_COMPILER]
 [-clinker=C_LINKER][-flinker=FORTRAN_LINKER]
 [-c++linker=CC_LINKER]
 [-cflags=CFLAGS][-fflags=FFLAGS][-c++flags=CCFLAGS]
 [-optcc=C_OPTFLAGS][-optf77=F77_OPTFLAGS]
 [-f90=F90_COMPILER][-f90flags=F90_FLAGS]
 [-f90inc=INCLUDE_DIRECTORY_SPEC_FORMAT_FOR_F90]
 [-f90linker=F90_LINKER]
 [-f90libpath=LIBRARY_PATH_SPEC_FORMAT_FOR_F90]
 [-lib-LIBRARY][-mpilibname=MPINAME]
 [-mpe_opts=MPE_OPTS][-jumpshot_opts=JUMPSHOT_OPTS]
 [-make=MAKEPGM]
 [-memdebug][-ptrdebug][-tracing][-dlast]
 [-listener_sig=SIGNAL_NAME]
 [-usesysv][-cross]
 [-pkt_size=LENGTH]
 [-adi_collective]
 [-fortnames=FORTRAN NAMES]
 [-automountfix=AUTOMOUNTFIX]
 [-noranlib][-ar_nolocal]
 [-rsh=RSHCOMMAND][-rshno]
 [-globusdir=GLOBUSDIR][-noromio][-file_system=FILE_SYSTEM]
 [-p4_opts=P4_OPTS]
```

其中，所有选项的配置都是任选的。但是，选项--with-arch, -comm 和-prefix 只能出现一次，且选项--with-arch 必须出现在选项-comm 之前。根据各个选项及其参数名字，我

们不难理解各个选项的具体含义。这里，我们列出一些关键的选项及其含义。

|               |                                                                                                 |
|---------------|-------------------------------------------------------------------------------------------------|
| ARCH_TYPE     | = 安装 MPICH 的并行机构结构                                                                              |
| COMM_TYPE     | = MPICH 使用的通信层                                                                                  |
| DEVICE        | = MPICH 使用的通信设备                                                                                 |
| INSTALL_DIR   | = MPICH 的安装目录                                                                                   |
| MPE_OPTS      | = 配置 MPE 的选项                                                                                    |
| JUMPSHOT_OPTS | = 配置 MPI 工具 jumpshot                                                                            |
| P4_OPTS       | = 配置通信设备 ch_p4, 其前提条件选择选项--with-device = ch_p4                                                  |
| C++_COMPILER  | = C++ 编译选项                                                                                      |
| OPTFLAGS      | = 传递给所有编译器的公共选项                                                                                 |
| CFLAGS        | = C 编译器选项                                                                                       |
| FFLAGS        | = Fortran 编译器选项                                                                                 |
| MAKEPGM       | = 编译工具 Make 的版本                                                                                 |
| LENGTH        | = 决定消息是短消息，还是长消息的消息长度（字节为单位），为了提高通信性能，MPICH 对短消息和长消息分别采用不同的协议                                   |
| FORTRAN NAMES | = MPI 函数 Fortran 版的名字格式（见下面介绍）                                                                  |
| AUTOMOUNTFIX  | = 自动装载设备的命令名字                                                                                   |
| RSHCOMMAND    | = 在其他处理机上启动进程的远程 Shell 名字（例如 rsh）                                                               |
| MPILIBNAME    | = MPICH 库函数的名字（缺省为 libmpich）                                                                    |
| GLOBUSDIR     | = 软件 Globus 的安装目录                                                                               |
| FILE_SYSTEM   | = 支持并行 I/O 文件系统 ROMIO 的名字，其中包括 nfs, ufs, , pfs(Intel), piofs(IBM), hfs(HP), sfs(NEC) 和 xfs(SGI) |
| SIGNAL_NAME   | = 通信设备 ch_p4 中用于建立通信联接的信号（缺省 SIGUSR1），其前提条件是选择选项--with-device = ch_p4                           |

MPICH 包含如下可供选择的软件包：

|                          |                                                                                                                                                           |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| --with-device = name     | 通信使用的设备名，其中包括 ch_p4, ch_mpl, ch_shmem, 和 globus 可供选择。缺省值为 ch_p4.                                                                                          |
| --with-romio[ = OPTIONS] | 使用文件系统 ROMIO 支持并行 I/O（缺省值），选项 -file_system = FSTYPE 可为 nfs, ufs, pfs(intel), piofs(IBM), hfs(HP), sfs(NEC) 和 xfs(SGI) 的任意组合，如果不选择该选项，则不能建立 Fortran 90 模块。 |
| --with-mpe               | 建立 MPE 环境（缺省值）                                                                                                                                            |
| --with-f90nag            | 选择 NAG f90 编译器为 Fortran 编译器                                                                                                                               |
| --with-f95nag            | 选择 NAG f95 编译器为 Fortran 编译器                                                                                                                               |

这些选项可以用--without-〈选项〉来关闭。

MPICH 支持的通信设备--with-device 包括：

|          |                                                 |
|----------|-------------------------------------------------|
| ch_nx    | (native Intel NX calls)                         |
| ch_mpl   | (native IBM EUI or MPL calls)                   |
| ch_p4    | (p4)                                            |
| globus   | (Globus)                                        |
| ch_meiko | (for Meiko CS2, using NX compatibility library) |
| ch_shmem | (for shared memory systems, such as SMPs)       |

|            |                                                                           |
|------------|---------------------------------------------------------------------------|
| ch_lfshmem | (for shared memory systems, such as SMPs; uses lock-free message buffers) |
| ch_cenju3  | (native NEC Cenju-3 calls)                                                |

其中：

- with-device = ch\_lfshmem[ :-usesysv ] ,适合共享存储并行机，并使用 lock-free 的消息缓存区，参数-usesysv 表明通信设备将使用 UNIX 系统 V 的共享存储和信号量子程序，而不是缺省的 mmap 或其他方法。
- with-device = ch\_meiko, 并行机底层通信库为 Meiko CS2。
- with-device = ch\_mpl, 并行机底层通信库为 IBM EUI 或 MPL。
- with-device = ch\_p4[ :-listener\_sig = SIGNALNAME ][ -dlast ][ -socksize = BYTES ] ,缺省值，参数-listener\_sig 给出通信设备建立新联接的信号名字，缺省为 SIGUSR1；参数-dlast 使得通信设备记录最近一次的调试信息，当进程中断时，打印该信息；参数-socksize 改变 Socket 的缓存区大小，合理地配置该参数，可以改进 MPICH 的通信性能。
- with-device = ch\_p4mpd[ :-listener\_sig = SIGNALNAME ][ -dlast ][ -socksize = BYTES ]
- with-device = ch\_shmem[ :-usesysv ] ,适合共享存储并行机。
- with-device = globus[ :-globusdir = GLOBUSDIR ] , 并行机底层通信库为 Globus，参数-globusdir 说明软件 Globus 的安装目录。

对分布式存储并行机和微机机群，用户可选择--with-device = ch\_p4；对共享存储并行机，用户可选择--with-device = ch\_shmem，或者--with-device = ch\_p4。

MPICH 支持如下参数选项：

|                           |                                                      |
|---------------------------|------------------------------------------------------|
| --enable-c++              | 建立 MPI 函数的 C++ 接口(缺省值)                               |
| --enable-f77              | 建立 MPI 函数的 Fortran 77 接口(缺省值)                        |
| --enable-weak-symbols     | MPI/PMPI 函数使用弱符号(缺省值)                                |
| --enable-debug            | 支持调试器能访问消息队列                                         |
| --enable-mpcdbg           | 支持-mpedbg 命令行参数                                      |
| --enable-sharedlib        | 建立共享函数库(缺省为静态函数库)                                    |
| --enablec-sharedlib = dir | 建立共享函数库并将其安装在指定的目录                                   |
| --enable-f90modules       | 建立 Fortran90 模块(如果系统安装了 Fortran90 或 95 编译器，则该选项为缺省值) |

以上选项可用选项--disable-<选项>来关闭。

选项-opt 可用于改变编译器的优化选项，通常可选择-opt = -O；选项-optcc 和 -optf77 可分别用于改变 C 和 Fortran 语言的编译选项；选项-cflags 和 -fflags 是 C 和 Fortran 的编译优化选项。注意，这些优化选项只被用于建立 MPICH，而不会被传递给编译命令 mpicc, mpif77, mpiCC 和 mpif90。

选项-lib 可用于说明某种特殊的通信设备所在的目录，选项-make 可用于选择工具 Make 的版本，缺省值为 -make = gnumake，选项--disable-short-longs 可用于压制对数据类型 long long 和 long double(ANSI/ISO C)的支持，他们将与数据类型 long 和 double 的长度一致。

选项-fortnames 可用于说明 MPI 函数名字的书写格式，例如：

-fortnames = DOUBLEUNDERSCORE 双下划线 (mpi\_send\_\_)

|                         |                  |
|-------------------------|------------------|
| -fortnames= underscore  | 单下划线 (mpi_send_) |
| -fortnames= caps        | 大写格式 (MPI_SEND)  |
| -fortnames= nunderscore | 无下划线 (mpi_send)  |

一般情况下, 用户不要使用该选项, MPICH 会根据使用的编译器类型, 确定函数名字的书写格式。

选项-finttype=<type>说明 C 的整型数据类型 int 对应于 Fortran 的整型数据类型 INTEGER, 选项-ar\_nolocal 可用于阻止 ar 命令使用本地目录作为临时硬盘空间, 选项-noranlib 可用于越过 ranlib 的步骤。

选项-memdebug 使调试器可以使用内存代码, 一般情形下, 这个选项很少使用, 除非用户发现了一个内存错误; 选项-tracing 使调试器可以追踪 MPICH 的内部函数。

选项-adi\_collective 允许 MPI 抽象设备接口提供一些聚合通信操作, 而不只是点对点通信操作。

选项--with-arch 支持的并行机结构包括:

|             |                                                                         |
|-------------|-------------------------------------------------------------------------|
| sun4        | (SUN OS 4.x)                                                            |
| solaris     | (Solaris)                                                               |
| solaris86   | (Solaris on Intel platforms)                                            |
| hpux        | (HP UX)                                                                 |
| sppux       | (SPP UX)                                                                |
| rs6000      | (AIX for IBM RS6000)                                                    |
| sgi         | (Silicon Graphics IRIX 4.x, 5.x or 6.x)                                 |
| sgi5        | (Silicon Graphics IRIX 5.x on R4400's, for the MESHINE)                 |
| IRIX        | (synonym for sgi)                                                       |
| IRIX32      | (IRIX with 32bit objects -32)                                           |
| IRIXN32     | (IRIX with -n32)                                                        |
| IRIX64      | (IRIX with 64bit objects)                                               |
| alpha       | (DEC alpha)                                                             |
| intelnx     | (Intel i860 or Intel Delta)                                             |
| paragon     | (Intel Paragon)                                                         |
| meiko       | (Meiko CS2)                                                             |
| CRAY        | (CRAY XMP, YMP, C90, J90, T90)                                          |
| cray_t3d    | (CRAY T3D)                                                              |
| freebsd     | (PC clones running FreeBSD)                                             |
| netbsd      | (PC clones running NetBSD)                                              |
| LINUX       | (PC clones running LINUX)                                               |
| ksr         | (Kendall Square KSR1 and KSR2)                                          |
| EWS_UX_V    | (NEC EWS4800/360AD Series workstation. Untested.)                       |
| UXPM        | (UXP/M. Untested.)                                                      |
| uxpv        | (uxp/v. Untested.)                                                      |
| SX_4_float0 | (NEC SX-4; Floating point format float0<br>Conforms IEEE 754 standard.) |
| C:          | sizeof(int) = 4; sizeof(float) = 4                                      |

```

FORTRAN: sizeof(INTEGER) = 4; sizeof(REAL) = 4
SX_4_float1
(NEC SX-4; Floating point format float1
IBM floating point format.
C: sizeof(int) = 4; sizeof(float) = 4
FORTRAN: sizeof(INTEGER) = 4; sizeof(REAL) = 4

SX_4_float2
(NEC SX-4; Floating point format float2
CRAY floating point format.
C: sizeof(int) = 4; sizeof(float) = 8
FORTRAN: sizeof(INTEGER) = 8; sizeof(REAL) = 8
!!! WARNING !!! This version will not run
together with FORTRAN routines.
sizeof(INTEGER)! = sizeof(int)

SX_4_float2_int64
(NEC SX-4; Floating point format float2 and
64-bit int's)
C: sizeof(int) = 8; sizeof(float) = 8
FORTRAN: sizeof(INTEGER) = 8; sizeof(REAL) = 8

```

注意,对 SGI(--with-arch = IRIX)并行机,如果选择--with-device = ch\_p4,则必须使用-comm = ch\_p4 关闭共享存储通信设备的使用,选项-comm = shared 可用于恢复共享存储通信设备的使用。

## 附录 C MPI 网站

目前,国际互联网上建立了许多 MPI 网站,下面列出几个比较著名的网站,读者通过这些网站可以访问 MPI 系统各个方面的内容。

| 网 站                                                                                                   | 说 明                                                                                                                |
|-------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| <a href="http://www.mpi-forum.org">http://www.mpi-forum.org</a>                                       | MPI 系统的最重要网站,它包含了 MPI 标准的制定和实现的方方面面。                                                                               |
| <a href="http://www.mcs.anl.gov/mpi">http://www.mcs.anl.gov/mpi</a>                                   | 美国 Argonne 国家重点实验室的 MPI 网站,它提供 MPICH 各个版本的源代码及其相关文档, MPI 并行程序设计辅助工具箱, MPI 例子程序, MPI 系统性能测试程序,以及与其他重要 MPI 网站的超级链接等。 |
| <a href="http://www.osc.edu/lam">http://www.osc.edu/lam</a>                                           | 提供 MPICH 标准的 LAMMPI 实现。                                                                                            |
| <a href="http://www.mpi1.nd.edu/MPI">http://www.mpi1.nd.edu/MPI</a>                                   | 包含 MPI 系统的免费实现软件和商业实现软件。                                                                                           |
| <a href="http://comp.parallel mpi">http://comp.parallel mpi</a>                                       | MPI 系统各个方面 的自由讨论网站。                                                                                                |
| <a href="http://www.erc.msstate.edu/mpi/mpi-faq.html">http://www.erc.msstate.edu/mpi/mpi-faq.html</a> | 解答 MPI 系统应用过程中经常遇到的各类问题的网站。                                                                                        |
| <a href="http://parallel.nas.nasa.gov/MPI IO">http://parallel.nas.nasa.gov/MPI IO</a>                 | MPI 并行 I/O 标准论坛。                                                                                                   |

## 附录 D MPI 函数索引

| MPI 函数                | 名称与含义                  | 页码  |
|-----------------------|------------------------|-----|
| MPI_ABORT             | MPI 系统退出函数             | 156 |
| MPI_ACCUMULATE        | MPI 远程内存写函数            | 179 |
| MPI_ADDRESS           | 绝对内存地址获取函数             | 62  |
| MPI_ALLGATHER         | 全收集聚合通信函数              | 77  |
| MPI_ALLGATHERV        | 全收集向量聚合通信函数            | 78  |
| MPI_ALLREDUCE         | 全归约聚合通信函数              | 86  |
| MPI_ALLTOALL          | 全交换聚合通信函数              | 79  |
| MPI_ALLTOALLV         | 全交换向量聚合通信函数            | 80  |
| MPI_ATTR_DELETE       | 通信器附加属性释放函数            | 108 |
| MPI_ATTR_GET          | 通信器附加属性查询函数            | 108 |
| MPI_ATTR_PUT          | 通信器附加属性设置函数            | 108 |
| MPI_BARRIER           | 同步通信函数                 | 70  |
| MPI_BCAST             | 标准模式消息广播函数             | 71  |
| MPI_BSEND             | 缓存区模式阻塞式消息发送函数         | 45  |
| MPI_BSEND_INIT        | 缓存区模式阻塞式消息发送持久通信请求创建函数 | 50  |
| MPI_BUFFER_ATTACH     | MPI 系统显式消息缓存区提交函数      | 46  |
| MPI_BUFFER_DETACH     | MPI 系统显式消息缓存区释放函数      | 46  |
| MPI_CANCEL            | 通信请求取消函数               | 44  |
| MPI_CART_COORDS       | Cartesian 拓扑结构进程坐标查询函数 | 115 |
| MPI_CART_CREATE       | Cartesian 拓扑结构创建函数     | 111 |
| MPI_CART_GET          | Cartesian 拓扑结构基本信息查询函数 | 115 |
| MPI_CART_MAP          | Cartesian 拓扑结构映射函数     | 118 |
| MPI_CART_RANK         | Cartesian 拓扑结构进程序号查询函数 | 115 |
| MPI_CART_SHIFT        | Cartesian 拓扑结构进程移位函数   | 116 |
| MPI_CART_SUB          | Cartesian 拓扑结构子集创建函数   | 117 |
| MPI_CARTDIM_GET       | Cartesian 拓扑结构维数查询函数   | 114 |
| MPI_COMM_COMPARE      | 通信器比较函数                | 105 |
| MPI_COMM_CREATE       | 通信器创建函数                | 102 |
| MPI_COMM_DUP          | 通信器复制函数                | 101 |
| MPI_COMM_FREE         | 通信器释放函数                | 107 |
| MPI_COMM_GROUP        | 通信器内进程序名查询函数           | 93  |
| MPI_COMM_RANK         | 通信器内进程序号查询函数           | 105 |
| MPI_COMM_REMOTE_GROUP | 域间通信器远程进程组名查询函数        | 110 |
| MPI_COMM_REMOTE_SIZE  | 域间通信器远程进程组大小查询函数       | 110 |
| MPI_COMM_SIZE         | 通信器大小查询函数              | 105 |
| MPI_COMM_SPLIT        | 通信器分裂函数                | 103 |
| MPI_COMM_TEST_INTER   | 通信器类型查询函数              | 110 |
| MPI_DIMS_CREATE       | Cartesian 拓扑结构最优维数计算函数 | 113 |
| MPI_ERRHANDLER_CREATE | MPI 异常错误处理程序创建函数       | 158 |
| MPI_ERRHANDLER_FREE   | MPI 异常错误处理程序释放函数       | 159 |
| MPI_ERRHANDLER_GET    | MPI 异常错误处理程序查询函数       | 159 |
| MPI_ERRHANDLER_SET    | MPI 异常错误处理程序设置函数       | 159 |

| MPI 函数                    | 名称与含义                | 页码  |
|---------------------------|----------------------|-----|
| MPI_ERROR_CLASS           | MPI 异常错误分类函数         | 160 |
| MPI_ERROR_STRING          | MPI 异常错误翻译函数         | 160 |
| MPI_FILE_CLOSE            | MPI 并行文件关闭函数         | 129 |
| MPI_FILE_GET_INFO         | MPI 并行 I/O 提示信息查询函数  | 150 |
| MPI_FILE_GET_TYPE_EXTENT  | MPI 并行 I/O 数据类型域查询函数 | 154 |
| MPI_FILE_IREAD            | MPI 非阻塞并行读函数         | 145 |
| MPI_FILE_IWRITE           | MPI 非阻塞并行写函数         | 145 |
| MPI_FILE_OPEN             | MPI 并行文件打开函数         | 127 |
| MPI_FILE_READ             | MPI 并行读函数            | 129 |
| MPI_FILE_READ_ALL_BEGIN   | MPI 分裂聚合并行读提交函数      | 146 |
| MPI_FILE_READ_ALL_END     | MPI 分裂聚合并行读完成函数      | 146 |
| MPI_FILE_READ_ALL         | MPI 聚合并行读函数          | 136 |
| MPI_FILE_READ_AT          | MPI 显式偏移并行读函数        | 131 |
| MPI_FILE_READ_SHARED      | MPI 共享文件指针并行读函数      | 147 |
| MPI_FILE_SEEK             | MPI 并行 I/O 文件指针定位函数  | 130 |
| MPI_FILE_ATOMICITY        | MPI 并行 I/O 访问模式设置函数  | 153 |
| MPI_FILE_SET_INFO         | MPI 并行 I/O 提示信息设置函数  | 149 |
| MPI_FILE_SET_VIEW         | MPI 文件窗口创建函数         | 133 |
| MPI_FILE_SYNC             | MPI 并行 I/O 同步函数      | 153 |
| MPI_FILE_WRITE            | MPI 并行写函数            | 128 |
| MPI_FILE_WRITE_ALL_BEGIN  | MPI 分裂聚合并行写提交函数      | 146 |
| MPI_FILE_WRITE_ALL_END    | MPI 分裂聚合并行写完成函数      | 146 |
| MPI_FILE_WRITE_ALL        | MPI 聚合并行写函数          | 136 |
| MPI_FILE_WRITE_AT         | MPI 显式偏移并行写函数        | 132 |
| MPI_FILE_WRITE_SHARED     | MPI 共享文件指针写函数        | 147 |
| MPI_FINALIZE              | 应用程序退出 MPI 系统函数      | 155 |
| MPI_GATHER                | 收集聚合通信函数             | 71  |
| MPI_GATHERV               | 收集向量聚合通信函数           | 73  |
| MPI_GET                   | MPI 远程内存读函数          | 178 |
| MPI_GET_COUNT             | 消息数据单元个数查询函数         | 27  |
| MPI_GET_ELEMENTS          | 消息基本数据单元个数查询函数       | 64  |
| MPI_GET_PROCESSOR_NAME    | 进程所在处理机名字查询函数        | 157 |
| MPI_GRAPH_CREATE          | 图拓扑结构创建函数            | 119 |
| MPI_GRAPH_GET             | 图拓扑结构基本信息查询函数        | 120 |
| MPI_GRAPH_MAP             | 图拓扑结构映射函数            | 121 |
| MPI_GRAPH_NEIGHBORS       | 图拓扑结构进程邻居结点查询函数      | 121 |
| MPI_GRAPH_NEIGHBORS_COUNT | 图拓扑结构进程邻居结点个数查询函数    | 121 |
| MPI_GRAPHDIMS_GET         | 图拓扑结构维数查询函数          | 120 |
| MPI_GROUP_COMPARE         | 进程组比较函数              | 99  |
| MPI_GROUP_DIFFERENCE      | 进程组差集创建函数            | 94  |
| MPI_GROUP_EXCL            | 进程组补集创建函数            | 95  |
| MPI_GROUP_FREE            | 进程组释放函数              | 100 |
| MPI_GROUP_INCL            | 进程组子集创建函数            | 95  |
| MPI_GROUP_INTERSECTION    | 进程组交集创建函数            | 94  |
| MPI_GROUP_RANGE_EXCL      | 基于三元数组的进程组补集创建函数     | 97  |

| MPI 函数                    | 名称与含义                  | 页码  |
|---------------------------|------------------------|-----|
| MPI_GROUP_RANGE_INCL      | 基于三元数组的进程组子集创建函数       | 96  |
| MPI_GROUP_RANK            | 进程组内进程序号查询函数           | 98  |
| MPI_GROUP_SIZE            | 进程组大小查询函数              | 98  |
| MPI_GROUP_TRANSLATE_RANKS | 两个进程组内进程对应序号查询函数       | 98  |
| MPI_GROUP_UNION           | 进程组并集创建函数              | 93  |
| MPI_IBSEND                | 缓存区模式非阻塞消息发送函数         | 49  |
| MPI_INFO_CREATE           | MPI 提示信息创建函数           | 148 |
| MPI_INFO_FREE             | MPI 提示信息释放函数           | 148 |
| MPI_INFO_GET              | MPI 提示信息属性查询函数         | 148 |
| MPI_INFO_GET_NKEYS        | MPI 提示信息属性查询函数         | 148 |
| MPI_INFO_GET_NTHKEY       | MPI 提示信息属性查询函数         | 148 |
| MPI_INFO_SET              | MPI 提示信息属性设置函数         | 148 |
| MPI_INIT                  | MPI 系统初始化函数            | 155 |
| MPI_INIT_THREAD           | MPI 系统多线程初始化函数         | 162 |
| MPI_INITIALIZED           | 应用程序是否已被 MPI 系统初始化查询函数 | 155 |
| MPI_INTERCOMM_CREATE      | 域间通信器创建函数              | 108 |
| MPI_INTERCOMM_MERGE       | 域间通信器合并函数              | 109 |
| MPI_IProbe                | 非阻塞消息查询函数              | 38  |
| MPI_IRecv                 | 标准模式非阻塞消息接收提交通信函数      | 32  |
| MPI_ISend                 | 就绪模式非阻塞消息发送函数          | 49  |
| MPI_IS_THREAD_MAIN        | 主线程查询函数                | 162 |
| MPI_ISEND                 | 标准模式非阻塞消息发送函数          | 31  |
| MPI_ISSEND                | 同步模式非阻塞消息发送函数          | 49  |
| MPI_KEYVAL_CREATE         | 通信器附加属性关键字创建函数         | 116 |
| MPI_KEYVAL_FREE           | 通信器附加属性关键字释放函数         | 116 |
| MPI_OP_CREATE             | 用户定义归约操作创建函数           | 90  |
| MPI_OP_FREE               | 用户定义归约操作释放函数           | 90  |
| MPI_PACK                  | 数据封装函数                 | 65  |
| MPI_PACK_SIZE             | 封装某个消息缓存区所需的内存大小查询函数   | 66  |
| MPI_PROBE                 | 消息是否到达阻塞式测试函数          | 37  |
| MPI_PUT                   | MPI 远程内存写函数            | 179 |
| MPI_RECV                  | 标准模式阻塞式消息接收通信函数        | 20  |
| MPI_RECV_INIT             | 标准模式阻塞式消息接收持久通信请求创建函数  | 41  |
| MPI_REDUCE                | 归约聚合通信函数               | 83  |
| MPI_REDUCE_SCATTER        | 归约分散聚合通信函数             | 87  |
| MPI_REQUEST_FREE          | 通信请求释放函数               | 43  |
| MPI_RSEND                 | 就绪模式阻塞式消息发送函数          | 48  |
| MPI_RSEND_INIT            | 就绪模式阻塞式消息发送持久通信请求创建函数  | 51  |
| MPI_SCAN                  | 并行前缀计算聚合通信函数           | 89  |
| MPI_SCATTER               | 消息分散聚合通信函数             | 74  |
| MPI_SCATTERV              | 消息向量分散聚合通信函数           | 76  |
| MPI_SEND                  | 标准模式阻塞式消息发送通信函数        | 20  |
| MPI_SEND_INIT             | 标准模式阻塞式消息发送持久通信请求创建函数  | 41  |
| MPI_SENDORecv             | 标准模式阻塞式消息发收通信函数        | 25  |
| MPI_SENDRECV_REPLACE      | 标准模式阻塞式消息发收替换通信函数      | 27  |
| MPI_SSEND                 | 同步模式阻塞式消息发送通信函数        | 48  |

| MPI 函数                                   | 名称与含义                   | 页码  |
|------------------------------------------|-------------------------|-----|
| <code>MPI_SSEND_INIT</code>              | 同步模式阻塞式消息发送持久通信请求函数     | 50  |
| <code>MPI_START</code>                   | 某个持久通信请求启动函数            | 42  |
| <code>MPI_STARTALL</code>                | 某类持久通信请求全启动函数           | 42  |
| <code>MPI_TEST</code>                    | 非阻塞式查询某个通信请求是否完成函数      | 35  |
| <code>MPI_TEST_CANCELLED</code>          | 非阻塞式查询某个通信请求是否已取消函数     | 44  |
| <code>MPI_TESTALL</code>                 | 非阻塞式查询某类通信请求是否全部完成函数    | 36  |
| <code>MPI_TESTANY</code>                 | 非阻塞式查询某类通信请求是否存在某个已完成函数 | 36  |
| <code>MPI_TESTSOME</code>                | 非阻塞式查询某类通信请求是否存在多个已完成函数 | 37  |
| <code>MPI_TOPO_TEST</code>               | 拓扑结构查询函数                | 122 |
| <code>MPI_TYPE_COMMIT</code>             | 自定义数据类型提交函数             | 63  |
| <code>MPI_TYPE_CONTIGUOUS</code>         | 连续空间拷贝的自定义数据类型创建函数      | 55  |
| <code>MPI_TYPE_CREATE_DARRAY</code>      | MPI 分布存储数组数据类型创建函数      | 138 |
| <code>MPI_TYPE_CREATE_INDED_BLOCK</code> | MPI 非规则分布存储数组数据类型创建函数   | 144 |
| <code>MPI_TYPE_CREATE_SUBARRAY</code>    | MPI 分布存储子数组数据类型创建函数     | 141 |
| <code>MPI_TYPE_CREATE_RESIZED</code>     | 数据类型域扩展函数               | 134 |
| <code>MPI_TYPE_EXTENT</code>             | 数据类型域查询函数               | 54  |
| <code>MPI_TYPE_FREE</code>               | 自定义数据类型释放函数             | 63  |
| <code>MPI_TYPE_HINDEXED</code>           | 字节索引的自定义数据类型创建函数        | 59  |
| <code>MPI_TYPE_HVECTOR</code>            | 字节索引等数据块的自定义数据类型创建函数    | 57  |
| <code>MPI_TYPE_INDEXED</code>            | 数据单元素引的自定义数据类型创建函数      | 58  |
| <code>MPI_TYPE_LB</code>                 | 数据类型域下界函数               | 54  |
| <code>MPI_TYPE_SIZE</code>               | 数据类型大小函数                | 55  |
| <code>MPI_TYPE_STRUCT</code>             | 结构索引的自定义数据类型创建函数        | 60  |
| <code>MPI_TYPE_UB</code>                 | 数据类型域上界函数               | 54  |
| <code>MPI_TYPE_VECTOR</code>             | 数据单元索引等数据块的自定义数据类型创建函数  | 56  |
| <code>MPI_UNPACK</code>                  | 消息数据拆卸函数                | 65  |
| <code>MPI_WAIT</code>                    | 某个通信请求完成函数              | 33  |
| <code>MPI_WAITALL</code>                 | 某类通信请求全部完成函数            | 34  |
| <code>MPI_WAITANY</code>                 | 某类通信请求至少存在一个已完成函数       | 33  |
| <code>MPI_WAITSOME</code>                | 某类通信请求部分完成函数            | 35  |
| <code>MPI_WIN_CREATE</code>              | MPI 系统远程内存访问函数          | 177 |
| <code>MPI_WIN_FENCE</code>               | MPI 远程内存访问区间创建函数        | 178 |
| <code>MPI_WTIME</code>                   | 应用程序墙上时间累计最小刻度          | 157 |
| <code>MPI_WTIME</code>                   | 应用程序墙上时间                | 156 |

## 附录 E 术语中英文对照及索引

| 中文术语           | 英文术语                            | 页码  |
|----------------|---------------------------------|-----|
| 边数组            | Edge Array                      | 119 |
| 标准模式通信         | Standard Communication          | 19  |
| 并行 I/O 数据一致性   | Data Coherence for Parallel I/O | 151 |
| 并行前缀计算         | Parallel Prefix Computations    | 82  |
| 不安全消息传递        | Unsafe Message Passing          | 32  |
| Cartesian 拓扑结构 | Cartesian Topology              | 111 |

| 中文术语         | 英文术语                                    | 页码  |
|--------------|-----------------------------------------|-----|
| 持久通信请求       | Persist Communication Request           | 41  |
| 串行 I/O       | Sequential I/O                          | 123 |
| 错误处理程序       | Error Handler                           | 158 |
| 大规模并行处理      | Massively Parallel Processing           | 4   |
| 单边通信         | One-side Communication                  | 176 |
| 单程序多数据流      | Single Program Multiple Dataflow        | 16  |
| 导出数据类型       | Derived Data Type                       | 52  |
| 动态进程管理       | Dynamic Process Management              | 181 |
| 点对点通信        | Point-to-Point Communication            | 15  |
| 多程序多数据流      | Multiple Program Multiple Dataflow      | 16  |
| 对称多处理        | Symmetric MultiProcessing               | 2   |
| 非连续访问并行 I/O  | Non-Continuous Parallel I/O             | 133 |
| 非 MPI 并行 I/O | Non-MPI Parallel I/O                    | 125 |
| 非阻塞并行 I/O    | Non-Blocking Parallel I/O               | 145 |
| 非阻塞通信        | Non-Blocking Communication              | 18  |
| 非阻塞消息发送函数    | Non-Blocking Message Sending Function   | 30  |
| 非阻塞消息完成函数    | Non-Blocking Message Receiving Function | 31  |
| 分布共享存储       | Distributed Shared Memory               | 3   |
| 分布存储数组       | Distributed Array                       | 138 |
| 分发           | Scatter                                 | 68  |
| 分裂聚合并行 I/O   | Splitting Collective Parallel I/O       | 146 |
| 共享文件指针并行 I/O | Shared File Pointer Parallel I/O        | 147 |
| 广播           | Broadcast                               | 68  |
| 根进程          | Root                                    | 68  |
| 归约操作         | Reduce Operator                         | 82  |
| 缓存模式通信       | Buffered Communication                  | 19  |
| 基本数据类型       | Basic Data Type                         | 22  |
| 进程           | Process                                 | 7   |
| 进程间通信        | Interprocess Communication              | 7   |
| 进程序号         | Process Rank                            | 14  |
| 进程组          | Process Group                           | 92  |
| 聚合并行 I/O     | Collective Parallel I/O                 | 136 |
| 就绪模式通信       | Ready Model Communication               | 19  |
| 聚合通信         | Collective Communication                | 16  |
| 局部通信器        | Local Communicator                      | 92  |
| 空进程          | Null Process                            | 28  |
| 空进程组         | Null Process Group                      | 93  |
| 联接器          | Handler                                 | 16  |
| 邻居结点         | Neighbor Nodes                          | 119 |
| 邻居结点数        | Number of Neighbor Nodes                | 119 |
| 邻居结点数组       | Neighbor Nodes Array                    | 119 |
| 零地址          | Bottom Address                          | 62  |
| MPI 并行 I/O   | MPI Parallel I/O                        | 123 |
| MPI 错误       | MPI Error                               | 158 |
| MPI 对象       | MPI Object                              | 16  |
| MPI 异常       | MPI Exception                           | 158 |

| 中文术语       | 英文术语                            | 页码  |
|------------|---------------------------------|-----|
| 收集         | Gather                          | 68  |
| 死锁         | Deadlock                        | 39  |
| 数据拆卸       | Unpack                          | 64  |
| 数据单元对界大小   | Alignment Size for Data Type    | 53  |
| 数据封装       | Pack                            | 64  |
| 数据类型大小     | Size of Data Type               | 53  |
| 双边通信       | Two-side Communication          | 176 |
| 墙上时间       | Wall Time                       | 156 |
| 桥进程        | Bridge Process                  | 109 |
| 全局通信器      | Global Communicator             | 92  |
| 全归约        | All Reduce                      | 82  |
| 全交换        | All-to-All Exchange             | 68  |
| 全收集        | All Gather                      | 68  |
| 提示信息       | Hint                            | 148 |
| 同步         | Barrier                         | 68  |
| 同步模式通信     | Synchronous Model Communication | 19  |
| 通信请求       | Communication Request           | 31  |
| 通信域        | Communication Domain            | 92  |
| 通信器        | Communicator                    | 93  |
| 拓扑结构       | Topology                        | 111 |
| 图拓扑结构      | Graph Topology                  | 111 |
| 无效的进程组     | Undefined Process Group         | 93  |
| 微机机群       | Beowulf PC-Cluster              | 5   |
| 文件窗口       | File Window                     | 133 |
| 线程         | Thread                          | 8   |
| 线程安全       | Thread Safe                     | 161 |
| 显式偏移并行 I/O | Obvious Offset Parallel I/O     | 131 |
| 消息         | Message                         | 8   |
| 消息长度       | Message Length                  | 23  |
| 消息传递       | Message Passing                 | 8   |
| 消息传递界面 MPI | Message Passing Interface MPI   | 11  |
| 消息大小       | Message Size                    | 23  |
| 用户自定义数据类型  | User Defined Data Type          | 52  |
| 域          | Extent                          | 53  |
| 域间通信器      | Intercommunicator               | 92  |
| 域内通信器      | Intracomunicator                | 92  |
| 远程内存访问     | Remote Memory Address           | 176 |
| 远程内存访问窗口   | Remote Memory Addressing Window | 176 |
| 阻塞通信       | Blocking Communication          | 18  |
| 坐标         | Coordinate                      | 111 |

