高性能计算之并行编程技术
—— MPI并行程序设计

都志辉　　编著
李三立　　审
陈渝　刘鹏　　校

内容提要

本书介绍目前最常见的并行程序—MPI并行程序的设计方法，它适合高校三四年级本科生、非计算机专业研究生作为教材和教学自学参考书，也适合于广大的并行计算（高性能计算）用户作为自学参考书使用，对于有FORTRAN和C编程经验的人员，都可以阅读并掌握本书的内容。

首先介绍了并行程序设计的基础，提供给读者进行并行程序设计所需要的基本知识；然后介绍了MPI的基本功能，从简单的例子入手，告诉读者MPI程序设计的基本过程和框架，这一部分是具有C或/FORTRAN串行程序设计经验的人员很容易理解和接受的；接下来介绍MPI程序设计的高级特征，是已经掌握了MPI基本程序设计的人员进一步编写简洁、高效的MPI程序，使用各种高级和复杂的MPI功能所需要的；最后一部分介绍了MPI的最新发展和扩充MPI-2，主要包括三个部分，动态进程管理、远程存储访问和并行文件读写。

本书包括了MPI-1的全部调用和MPI-2的关键扩充部分的调用，并附以大量的图表和示例性程序，对程序的关键部分给出了讲解或注释。读者若能将例子和对MPI调用的讲解结合起来学习，会取得更好的效果。

本书的目的，不仅是教给读者如何去编写从简单到复杂的MPI并行程序，更重要的是，希望在学习本书之后，在读者以后解决问题的过程中，能够树立并行求解的概念，使并行方法真正成为广大应用人员和程序开发员手中的重要工具。

目录

序

中国工程院院士　李三立

前言

程序列表

图列表

表格列表

第一部分　并行程序设计基础

本部分包括如下内容：并行计算机、并行编程模型与并行语言、并行算法。

通过本部分的介绍，使读者对并行计算和并行程序设计有一个基本的概念，为后续几章具体讲解MPI并行程序设计方法提供基础知识。

第1章 并行计算机

本章给出了并行计算机的基本划分方法和与之相关的体系结构，是宏观的总体的论述而不是具体的细节，这主要是考虑到读者是并行程序设计人员而不是并行机的设计与开发人员。对并行机有一个总体上的了解可以帮助编程者设计出更高效的并行程序。

本章还给出了一个物理问题如何一步步在并行机上得到解决的，以及并行程序设计在这一过程中所起的作用。

1.1 并行计算机的分类

为什么要采用并行计算？这是因为：1 它可以加快速度，即在更短的时间内解决相同的问题或在相同的时间内解决更多更复杂的问题，特别是对一些新出现的巨大挑战问题，不使用并行计算是根本无法解决的；2 节省投入，并行计算可以以较低的投入完成串行计算才能够完成的任务；3 物理极限的约束，光速是不可逾越的速度极限，设备和材料也不可能做得无限小，只有通过并行才能够不断提高速度。

并行计算机即能在同一时间内执行多条指令（或处理多个数据）的计算机，并行计算机是并行计算的物理载体。通过下面对并行计算机的不同分类方式，可以对它有一个总体上的了解，为并行程序设计奠定基础。

1.1.1 指令与数据

根据一个并行计算机能够同时执行的指令与处理数据的多少，可以把并行计算机分为SIMD（Single-Instruction Multiple-Data）单指令多数据并行计算机和MIMD（Multiple-Instruction Multiple-Data）多指令多数据并行计算机（图 1）。

SIMD计算机同时用相同的指令对不同的数据进行操作，比如对于数组赋值运算

A=A+1

在SIMD并行机上可以用加法指令同时对数组A的所有元素实现加1。即数组（或向量）运算特别适合在SIMD并行计算机上执行，SIMD并行机可以对这种运算形式进行直接地支持，高效地实现。

MIMD计算机同时有多条指令对不同的数据进行操作，比如对于算术表达式

A=B+C+D-E+F*G

可以转换为

A=(B+C)+(D-E)+(F*G)

加法（B+C），减法（D-E），乘法（F*G）如果有相应的直接执行部件，则这三个不同的计算可以同时进行。

SIMD和MIMD这种表达方法虽然至今还在广泛使用，但是，随着新的并行计算机组织方式的产生，比照上面的划分方法，人们按同时执行的程序和数据的不同，又提出了SPMD（Single-Program Multuple-Data）单程序多数据并行计算机和MPMD（Multiple-Program Multiple-Data）多程序多数据并行计算机（图 1）的概念。这种划分方式依据的执行单位不是指令而是程序，显然其划分粒度要大得多。

如果一个程序的功能就是为一个矩形网格内的不同面片涂上相同的颜色，则对于一个划

分得很细的特大矩形面片，可以将它划分为互不交叉的几个部分，每一部分都用相同的程序进行着色。SPMD并行计算机可以很自然地实现类似的计算。一般地，SPMD并行计算机是由多个地位相同的计算机或处理器组成的，而MPMD并行计算机内计算机或处理器的地位是不同的，根据分工的不同，它们擅长完成的工作也不同，因此，可以根据需要将不同的程序（任务）放到MPMD并行计算机上执行，使得这些程序协调一致地完成给定的工作。

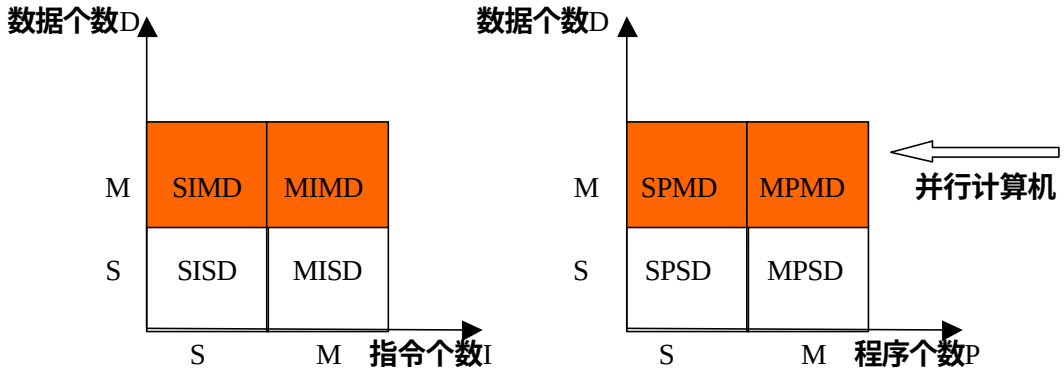


图 1 按指令（程序）数据的个数对并行计算机进行分类

1.1.2 存储方式

从物理划分上，共享内存和分布式内存是两种基本的并行计算机存储方式，除此之外，分布式共享内存也是一种越来越重要的并行计算机存储方式（图 2）。

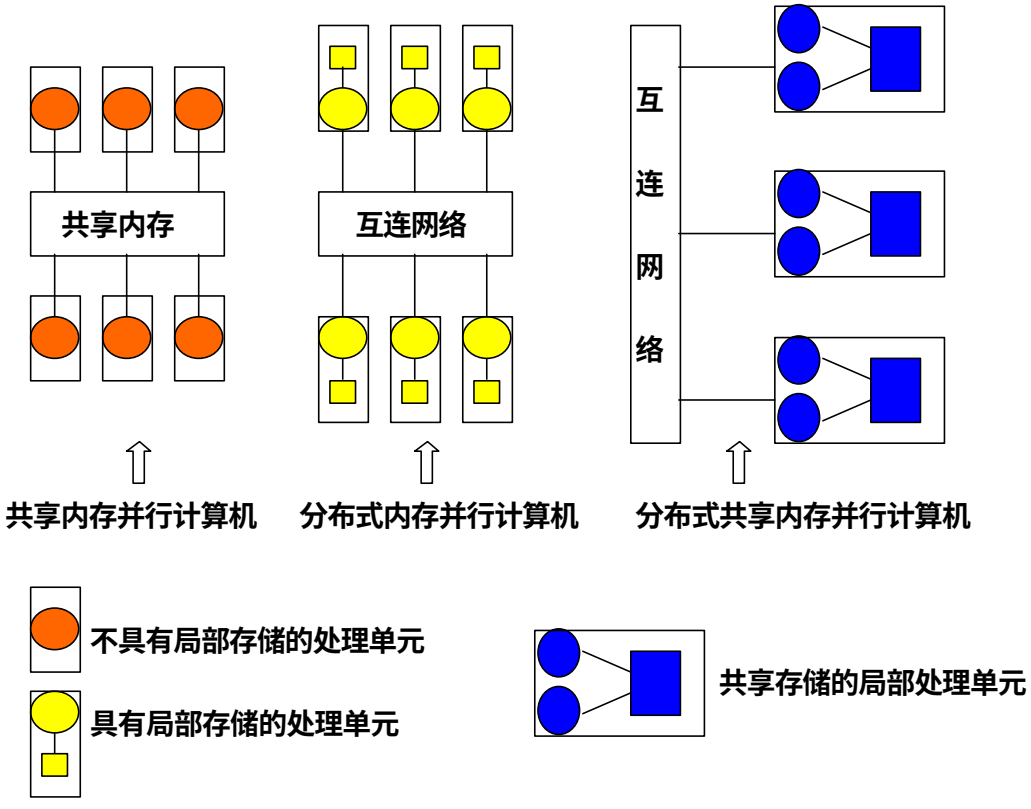


图 2 按存储方式对并行计算机进行分类

对于共享内存的并行计算机，各个处理单元通过对共享内存的访问来交换信息、协调各处理器对并行任务的处理。对这种共享内存的编程，实现起来相对简单，但共享内存往往成为性能特别是扩展性的重要瓶颈。

对于分布式内存的并行计算机，各个处理单元都拥有自己独立的局部存储器，由于不存在公共可用的存储单元，因此各个处理器之间通过消息传递来交换信息，协调和控制各个处理器的执行。这是本书介绍的消息传递并行编程模型所面对的并行计算机的存储方式。不难看出，通信对分布式内存并行计算机的性能有重要的影响，复杂的消息传递语句的编写成为在这种并行计算机上进行并行程序设计的难点所在，但是，对于这种类型的并行计算机，由于它有很好的扩展性和很高的性能，因此，它的应用非常广泛。

分布式共享内存的并行计算机结合了前两者的特点，是当今新一代并行计算机的一种重要发展方向。对于目前越来越流行的机群计算（Cluster Computing），大多采用这种形式的结构。通过提高一个局部结点内的计算能力，使它成为所谓的“超结点”，不仅提高了整个系统的计算能力，而且可以提高系统的模块性和扩展性，有利于快速构造超大型的计算系统。

1.2 物理问题在并行机上的求解

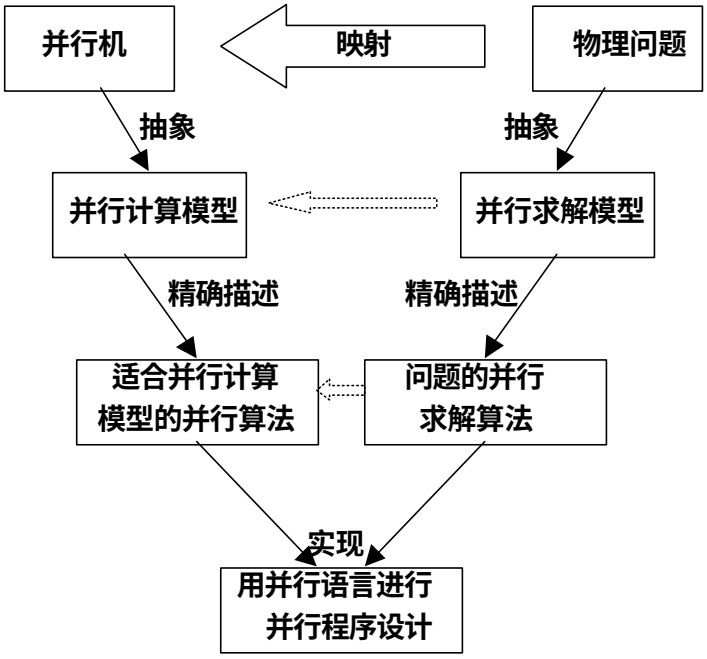


图 3 问题的并行求解过程

一个物理问题并行求解的最终目的是将该问题映射到并行机上，这一物理上的映射是通过不同层次上的抽象映射来实现的（图 3）。

忽略并行机的非本质的细节特征，可以得到该并行机的并行计算模型，在这一模型上，可以设计各种适合该模型的并行算法，这些算法精确描述了该并行模型能够实现的功能，而这些算法是通过用特定的并行语言设计并行程序后得以实现的。对于现实世界的物理问题，为了能够高效地并行求解，必须建立它的并行求解模型，一个串性的求解模型是很难在并行机上取得满意的并行效果的。有了并行求解模型，就可以针对该模型设计高效的并行算法，这样就可以对该问题的求解进行精确描述和定量分析，就可以对各种不同的算法进行性能上

的比较，最后通过并行程序设计，实现问题和并行机的结合。

并行程序设计，需要将问题的并行求解算法转化为特定的适合并行计算模型的并行算法，为了达到这一目的，首先是问题的并行求解算法必须能够将问题内在的并行特征充分体现出来，否则并行求解算法将无法利用这些并行特征，从而使问题的高效并行求解成为不可能；其次是并行求解模型要和并行计算模型尽量吻合，这样，就为问题向并行机上的高效解决提供了前提。

1.3 小结

本章对并行机和物理问题在并行机上的求解进行了简单地介绍，按照指令和数据的不同对并行计算机进行划分是一种经典的方法，至今仍在使用，而不同的编程模式和并行计算机的存储方式有很大的关系，其实互连网络也是并行计算机的重要组成部分，但是对于并行程序员来说看不到这种不同，因此本章没有对它进行介绍。本书的目的是力求以最简单的方式将最重要和最基本的内容介绍给读者。

第2章 并行编程模型与并行语言

本章介绍最重要的两种并行编程模型–数据并行和消息传递及它们之间的相互关系，讲述了它们各自的优缺点和适用范围，给出了几种并行语言的产生方法和各自的特点。

2.1 并行编程模型

目前两种最重要的并行编程模型是数据并行和消息传递，数据并行编程模型的编程级别比较高，编程相对简单，但它仅适用于数据并行问题；消息传递编程模型的编程级别相对较低，但消息传递编程模型可以有更广泛的应用范围。

数据并行即将相同的操作同时作用于不同的数据，因此适合在SIMD及SPMD并行计算机上运行，在向量机上通过数据并行求解问题的实践也说明数据并行是可以高效地解决一大类科学与工程计算问题的。

数据并行编程模型是一种较高层次上的模型，它提供给编程者一个全局的地址空间，一般这种形式的语言本身就提供并行执行的语义，因此对于编程者来说，只需要简单地指明执行什么样的并行操作和并行操作的对象，就实现了数据并行的编程，比如对于数组运算，使得数组B和C的对应元素相加后送给A，则通过语句

A=B+C（或其它的表达方式）

就能够实现上述功能，使并行机对B、C的对应元素并行相加，并将结果并行赋给A。因此数据并行的表达是相对简单和简洁的，它不需要编程者关心并行机是如何对该操作进行并行执行的。

数据并行编程模型虽然可以解决一大类科学与工程计算问题，但是对于非数据并行类的问题，如果通过数据并行的方式来解决，一般难以取得较高的效率，数据并行不容易表达甚至无法表达其它形式的并行特征。

数据并行发展到现在，高效的编译实现成为它面临的一个主要问题，有了高效的编译器，数据并行程序就可以在共享内存和分布式内存的并行机上都取得高效率，这样可以提高并行程序的开发效率，提高并行程序的可移植性，进一步推广并行程序设计。

消息传递即各个并行执行的部分之间通过传递消息来交换信息、协调步伐、控制执行。消息传递一般是面向分布式内存的，但是它也可适用于共享内存的并行机。消息传递为编程者提供了更灵活的控制手段和表达并行的方法，一些用数据并行方法很难表达的并行算法，都可以用消息传递模型来实现，灵活性和控制手段的多样化，是消息传递并行程序能提供高的执行效率的重要原因。

消息传递模型一方面为编程者提供了灵活性，另一方面，它也将各个并行执行部分之间复杂的信息交换和协调、控制的任务交给了编程者，这在一定程度上增加了编程者的负担，这也是消息传递编程模型编程级别低的主要原因。虽然如此，消息传递的基本通信模式是简单和清楚的，学习和掌握这些部分并不困难，因此目前大量的并行程序设计仍然是消息传递并行编程模式。

表格 1是对数据并行和消息传递两种并行编程模式的简单对比。

表格 1 数据并行与消息传递并行编程模型

对比内容	数据并行	消息传递
编程级别	高	低
适用的并行机类型	SIMD/SPMD	SIMD/MIMD/SPMD/MPMD
执行效率	依赖于编译器	高
地址空间	单一	多个
存储类型	共享内存	分布式或共享内存
通信的实现	编译器负责	程序员负责
问题类	数据并行类问题	数据并行、任务并行
目前状况	缺乏高效的编译器支持	使用广泛

2.2 并行语言

并行程序是通过并行语言来表达的，并行语言的产生主要有三种方式：1 设计全新的并行语言；2 扩展原来的串行语言的语法成分，使它支持并行特征；3 不改变串行语言，仅为串行语言提供可调用的并行库。

设计一种全新的并行语言的优点是可以完全摆脱串行语言的束缚，从语言成分上直接支持并行，这样就可以使并行程序的书写更方便，更自然，相应的并行程序也更容易在并行机上实现。但是，由于并行计算至今还没有象串行计算那样统一的冯•诺伊曼模型可供遵循，因此并行机、并行模型、并行算法和并行语言的设计和开发千差万别，没有一个统一的标准，虽然有多种多样全新的并行语言出现，但至今还没有任何一种新出现的并行语言，成为普遍接受的标准，设计全新的并行语言，实现起来难度和工作量都很大，但各种各样并行语言的出现、实践和研究，无疑都为并行语言和并行计算的发展作出了贡献。

一种重要的对串行语言的扩充方式就是标注，即将对串行语言的并行扩充作为原来串行语言的注释，对于这样的并行程序，若用原来的串行编译器来编译，标注的并行扩充部分将不起作用，仍将该程序作为一般的串行程序处理，若使用扩充后的并行编译器来编译，则该并行编译器就会根据标注的要求，将原来串行执行的部分转化为并行执行。对串行语言的并行扩充，相对于设计全新的并行语言，显然难度有所降低，但需要重新开发编译器，使它能够支持扩充的并行部分，一般地，这种新的编译器往往和运行时支持的并行库相结合。

仅仅提供并行库，是一种对原来的串行程序设计改动最小的并行化方法。这样，原来的串行编译器也能够使用，不需要任何修改，编程者只需要在原来的串行程序中加入对并行库的调用，就可以实现并行程序设计，本书所介绍的MPI并行程序设计，就属于这种方式。

对于这三种并行语言的实现方法，目前最常使用的是第二种和第三种方法，特别是第三种方法。

图 4 给出了并行语言的实现方式和实现难度之间的关系。

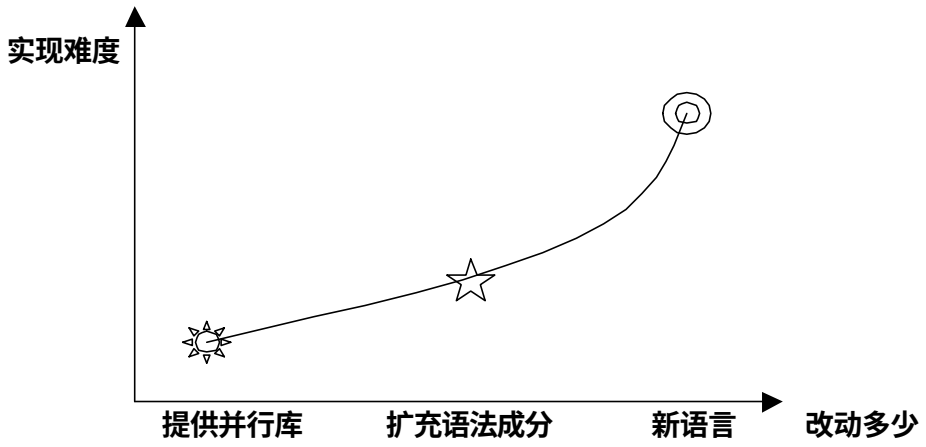


图 4 并行语言的实现方法

2.3 小结

并行编程模型除了数据并行和消息传递之外，还有共享变量模型、函数式模型等等，但它们的应用都没有数据并行和消息传递那样普遍，因此本书不再介绍，况且只要知道了一两种并行编程模型，再自学其它的模型也会比较容易些。

并行语言的发展其实十分迅速，并行语言的种类也非常多，但真正使用起来并被广为接受的却寥寥无几，因此这里并没有介绍某一具体的并行语言，而只是给出了并行语言产生的基本方法，对FORTRAN和C的扩充是最常见的并行语言产生方法，MPI并行程序设计就是和FORTRAN或C结合起来实现的。

第3章 并行算法

本章首先介绍了并行算法的各种分类方法，然后讲述了如何在目前特别流行的一种并行系统–机群系统上设计高效的并行算法，说明了在机群系统上适合什么样的并行算法，给出了一些基本的原则和方法，为以后的并行程序设计打下基础。

3.1 并行算法分类

并行算法是给定并行模型的一种具体、明确的解决方法和步骤。按照不同的划分方法，并行算法有多种不同的分类。

根据运算的基本对象的不同，可以将并行算法分为数值并行算法（数值计算）和非数值并行算法（符号计算）。当然，这两种算法也不是截然分开的，比如在数值计算的过程中会用到查找、匹配等非数值计算的成分，当然非数值计算中也一般会用到数值计算的方法，划分为什么类型的算法主要取决于主要的计算量和宏观的计算方法。

根据进程之间的依赖关系可以分为同步并行算法（步调一致）、异步并行算法（步调、进展互不相同）和纯并行算法（各部分之间没有关系）。对于同步并行算法，任务的各个部分是同步向前推进的，有一个全局的时钟（不一定是物理的）来控制各部分的步伐；而对于异步并行算法，各部分的步伐是互不相同的，它们根据计算过程的不同阶段决定等待、继续或终止；纯并行算法是最理想的情况，各部分之间可以尽可能快地向前推进，不需要任何同步或等待，但是一般这样的问题是少见的。

根据并行计算任务的大小，可以分为粗粒度并行算法（一个并行任务包含较长的程序段和较大的计算量）、细粒度并行算法（一个并行任务包含较短的程序段和较小的计算量）以及介于二者之间的中粒度并行算法。一般而言，并行的粒度越小，就有可能开发更多的并行性，提高并行度，这是有利的方面，但是另一个不利的方面就是并行的粒度越小，通信次数和通信量就相对增多，这样就增加了额外的开销，因此合适的并行粒度需要根据计算量、通信量、计算速度、通信速度进行综合平衡，这样才能够取得高效率。

3.2 并行算法的设计

对于相同的并行计算模型，可以有多种不同的并行算法来描述和刻画，由于并行算法设计的不同，可能对程序的执行效率有很大的影响，不同的算法可以有几倍、几十倍甚至上百倍的性能差异是完全正常的。

并行算法基本上是随着并行机的发展而发展的，从本质上说，不同的并行算法是根据问题类别的不同和并行机体系结构的特点产生出来的，一个好的并行算法要既能很好地匹配并行计算机硬件体系结构的特点，又能反映问题内在并行性。

对于SIMD并行计算机一般适合同步并行算法，而MIMD并行计算机则适合异步并行算法。对于SPMD和MPMD这些新流行起来的并行计算机，设计并行算法的思路和以前并行算法的思路有很大的不同。下面针对机群系统，重点讲一下SPMD和MPMD并行算法的设计。

对于机群计算，有一个很重要的原则就是设法加大计算时间相对于通信时间的比重，减少通信次数甚至以计算换通信。这是因为，对于机群系统，一次通信的开销要远远大于一次计算的开销，因此要尽可能降低通信的次数，或将两次通信合并为一次通信。基于同样的原

因，机群计算的并行粒度不可能太小，因为这样会大大增加通信的开销。如果能够实现计算和通信的重叠，那将会更大地提高整个程序的执行效率。

因此，对于机群计算，可以是数值或非数值的计算，这些都不是影响性能的关键，也可以是同步、松同步或异步的，但以同步和松同步为主，并行的粒度一般是大粒度或中粒度的。一个好的算法一般应该呈现如下的计算模式：

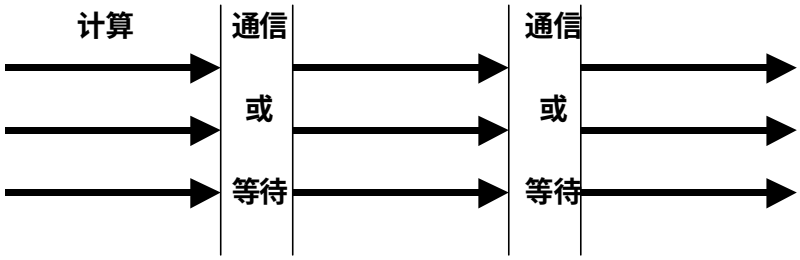


图 5 适合机群系统的SPMD并行算法的计算模式

图 5 没有考虑计算与通信的重叠，若能够实现计算与通信的重叠，那将是更理想的计算模式。

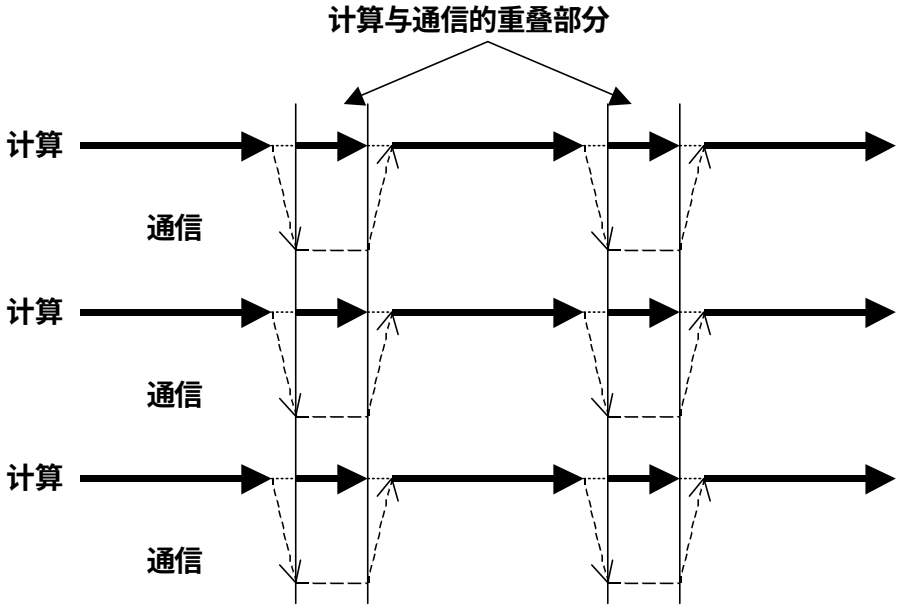


图 6 计算与通信重叠的SPMD并行算法的计算模式

图 6 是加入了计算和通信重叠技术后的SPMD并行算法的计算模式。

对于MPMD并行算法，各并行部分一般是异步执行的，而不是象SPMD那样的同步或松同步方式，因此只要能够大大降低通信次数，增大计算相对于通信的比重，则该MPMD算法就可以取得较高的效率。图 7 给出了MPMD算法的一种比较合适的计算模式。

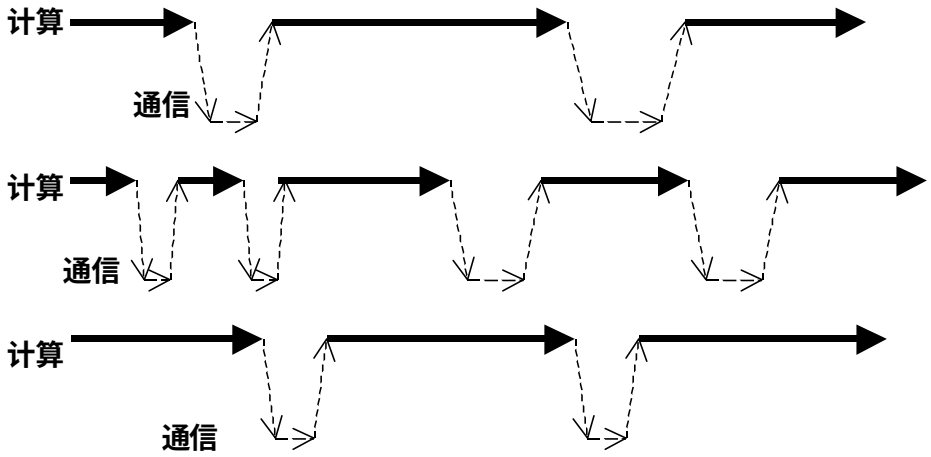


图 7 适合机群系统的MPMD并行算法

3.3 小结

在并行计算中，由于并行算法可以对性能产生重大的影响，因此受到广泛的重视，并行算法也成为一个专门的十分活跃的研究领域。并行算法设计也是并行程序设计的前提，没有好的并行算法，就没有好的并行程序，因此在并行程序设计之前，必须首先考虑好并行算法，该算法要能够将并行机和实际的问题很好地结合起来，既能够充分利用并行机体系结构的特点，又能够揭示问题内在的并行性。

第二部分　基本的MPI并行程序设计

本部分讲述基本的MPI并行程序设计所需要的知识，包括MPI的简单介绍，一个相对完备的MPI子集，对等模式和主从模式MPI程序的编写，MPI的一个具体实现MPICH在Linux和NT操作系统下的安装和MPI程序的执行。

通过本部分的学习，可以掌握MPI最基本和最常见的程序设计方法，编写出能满足一般需求的MPI并行程序。这一部分主要基于MPI-1来进行讲解，不涉及MPI-2的内容。

第4章 MPI简介

本章对MPI进行简要介绍，使读者对MPI有一个宏观的总体了解，为下面深入理解MPI，掌握MPI并行编程技术提供必要的准备。在本章不涉及编程细节。

4.1 什么是MPI

对MPI的定义是多种多样的，但不外乎下面三个方面，它们限定了MPI的内涵和外延。

① MPI是一个库，而不是一门语言。许多人认为MPI就是一种并行语言，这是不准确的。但是按照并行语言的分类，可以把FORTRAN+MPI或C+MPI，看作是一种在原来串行语言基础之上扩展后得到的并行语言。MPI库可以被FORTRAN77/C/Fortran90/C++调用，从语法上说，它遵守所有对库函数/过程的调用规则，和一般的函数/过程没有什么区别。

② MPI是一种标准或规范的代表，而不特指某一个对它的具体实现。迄今为止，所有的并行计算机制造商都提供对MPI的支持，可以在网上免费得到MPI在不同并行计算机上的实现，一个正确的MPI程序，可以不加修改地在所有的并行机上运行。

③ MPI是一种消息传递编程模型，并成为这种编程模型的代表和事实上的标准。MPI虽然很庞大，但是它的最终目的是服务于进程间通信这一目标的。

在MPI上很容易移植其它的并行代码，而且编程者不需要去努力掌握许多其它的全新概念，就可以学习编写MPI程序。当然，这并不意味着MPI已经十分完美，必须承认MPI自身还存在着一些缺点，在本书的后续章节将对它进行讨论。

消息传递方式是广泛应用于多类并行机的一种模式,特别是那些分布存储并行机，尽管在具体的实现上有许多不同，但通过消息完成进程通信的基本概念是容易理解的。十多年来，这种模式在重要的计算应用中已取得了实质进步。有效和可移植地实现一个消息传递系统是可行的，因此，通过定义核心库程序的语法、语义，这将在大范围计算机上可有效实现，将有益于广大用户，这是MPI产生的重要原因。

4.2 MPI的目的

MPI为自己制定了一个雄心勃勃的目标，总结概括起来，它包括几个在实际使用中都十分重要但有时又是相互矛盾的三个方面：1 较高的通信性能；2 较好的程序可移植性；3 强大的功能。具体地说，包括以下几个方面：

- 提供应用程序编程接口。
- 提高通信效率。措施包括避免存储器到存储器的多次重复拷贝，允许计算和通信的重叠等。
- 可在异构环境下提供实现。
- 提供的接口可以方便 C 语言和 Fortran 77的调用。
- 提供可靠的通信接口。即用户不必处理通信失败。
- 定义的接口和现在已有接口（如PVM，NX，Express，p4等）差别不能太大，但是允许扩展以提供更大的灵活性。
- 定义的接口能在基本的通信和系统软件无重大改变时，在许多并行计算机生产商的平台上实现。接口的语义是独立于语言的。

- 接口设计应是线程安全的。

MPI提供了一种与语言和平台无关，可以被广泛使用的编写消息传递程序的标准，用它来编写消息传递程序，不仅实用、可移植、高效和灵活，而且和当前已有的实现没有太大的变化。

4.3 MPI的产生

许多组织对MPI标准付出了努力，它们主要来自美国和欧洲，大约有六十几个人，分属四十几个不同的单位。这包括了并行计算机的多数主要生产商，还有来自大学、政府实验室和工厂的研究人员。

有关的工作及其组织包括 Venus (IBM) 、NX/2 (Intel) 、Express (Parasoft) 、Vertex (nCUBE) 、P4 (ANL) 、PARMACS (ANL) ，还包括Zipcode (MSU) 、Chimp (Edinburgh University) 、PVM (ORNL, UTK, Emory U.) 、Chameleon (ANL) 、PICL (ANL)等。

MPI的标准化开始于一九九二年四月二十九日至三十日在威吉尼亚的威廉姆斯堡召开的分布存储环境中消息传递标准的讨论会，由Dongarra,Hempel,Hey和Walker建议的初始草案于一九九二年十一月推出，并在一九九三年二月完成了修订版，这就是MPI 1.0。为了促进MPI的发展，一个称为MPI论坛的非官方组织应运而生，该论坛对MPI的发展起了重要的作用。

一九九五年六月推出了MPI的新版本MPI1.1，对原来的MPI作了进一步的修改、完善和扩充。但是当初推出MPI标准时，为了能够使它尽快实现并迅速被接受，许多其它方面的重要但实现起来比较复杂的功能都没有定义，比如并行I/O，当MPI被广为接受之后，扩充并提高MPI功能的要求就越来越迫切了。

于是在一九九七年的七月，在对原来的MPI作了重大扩充的基础上，又推出了MPI的扩充部分MPI-2，而把原来的MPI各种版本称为MPI-1。MPI-2的扩充很多，但主要是三个方面：并行I/O，远程存储访问和动态进程管理。本书的这一部分主要讲述MPI-1的内容，对于MPI-2，将在最后做专门介绍。

4.4 MPI的语言绑定

由于MPI是一个库而不是一门语言，因此对MPI的使用必须和特定的语言结合起来进行，FORTRAN是科学与工程计算的领域语言，而C又是目前使用最广泛的系统和应用程序开发的语言之一，因此对FORTRAN和C的支持是必须的。

因此在MPI-1中，明确提出了MPI和FORTRAN 77与C语言的绑定，并且给出了通用接口和针对FORTRAN 77与C的专用接口说明，MPI-1的成功说明MPI选择的语言绑定策略是正确和可行的。

Fortran90是FORTRAN的扩充，它在表达数组运算方面有独特的优势，还增加了模块等现代语言的方便开发与使用的各种特征，它目前面临的一个问题是Fortran90编译器远不如FORTRAN 77编译器那样随处可见，但提供Fortran90编译器的厂商正在逐步增多。C++作为面向对象的高级语言，随着编译器效率和处理器速度的提高，它可以取得接近于C的代码效率，面向对象的编程思想已经被广为接受，因此在MPI-2中，除了和原来的FORTRAN 77和C语言实现绑定之外，进一步与Fortran90和C++结合起来，提供了四种不同的接口，为编程者提供了更多选择的余地。但是MPI-2目前还没有完整的实现版本。

4.5 目前主要的MPI实现

MPICH是一种最重要的MPI实现，它可以免费从http://www-unix.mcs.anl.gov/mpi/mpich取得。更为重要的是，MPICH是一个与MPI-1规范同步发展的版本，每当MPI推出新的版本，就会有相应的MPICH的实现版本，目前MPICH的最新版本是MPICH-1.2.1，它支持部分的MPI-2的特征。Argonne国家试验室和MSU对MPICH作出了重要的贡献。

CHIMP是Edinburgh开发的另一个免费MPI实现，是在EPCC（Edinburgh Parallel Computing Centre）的支持下进行的，从ftp://ftp.epcc.ed.ac.uk/pub/packages/chimp/release/可以免费下载该软件，CHIMP的开发从1991年到1994年，主要开发人员有Alasdair Bruce, James (Hamish) Mills,和Gordon Smith。

LAM (Local Area Multicomputer)也是免费的MPI实现，由Ohio State University 开发，它目前的最新版本是LAM/MPI 6.3.2，可以从http://www.mpi.nd.edu/lam/download/ 下载。它主要用于异构的计算机网络计算系统。

表格 2列出了一些主要的MPI免费实现。

表格 2 MPI的一些实现

实现名称	研制单位	网址
Mpich	Argonne and MSU	http://www-unix.mcs.anl.gov/mpi/mpich
Chimp	Edinburgh	ftp://ftp.epcc.ed.ac.uk/pub/packages/chimp/
Lam	Ohio State University	http://www.mpi.nd.edu/lam/

4.6 小结

本章的目的是给读者一个关于MPI的概貌，MPI的一个最重要的特点就是免费和源代码开放，MPI可以被迅速接受和它为自己定下的高效率、方便移植和功能强大三个主要目标密不可分。它采用广为使用的语言FORTRAN和C进行绑定，也是它成功的一个重要因素，当然，MPI的成功，还因为它总结和吸收了前期大量消息传递系统的经验，一个成功的标准是需要大量的实践和艰苦的努力的，MPI就是这种实践和努力的结果。

第5章 第一个MPI程序

本章以一个简单具体的例子“Hello World”为切入点，给出了MPI程序的一个基本框架，使读者对MPI并行程序有一个基本的感性认识。这里首先给出的例子十分简单，目的是使读者消除对编写并行程序的疑虑，任何有FORTRAN或C编程经验的读者，是可以按照本书介绍的次序，一步步从简单到复杂，逐步掌握MPI并行程序设计的各种方法和技巧的。

5.1 MPI实现的“Hello World!”

C语言的程序设计者对“Hello World”这一例子也许还记忆犹新，因为这一例子非常简单，又具有一定的代表性，因此在介绍MPI并行程序设计时，本书首先向读者介绍这一例子。

5.1.1 用FORTRAN77+MPI实现

如程序 1所示的第一个FORTRAN77+MPI并行程序。下面分几个部分对它的结构进行介绍。

第一部分，首先要有MPI相对于FORTRAN实现的头文件mpif.h，对于MPI相对于C语言的实现，其头文件是不同的。即用FORTRAN语言编写的MPI并行程序，必须有MPI的FORTRAN头文件mpif.h。现在已经有些实现支持Fortran90+MPI，在MPI-2中明确提出了对Fortran90和C++的支持。如果是Fortran90程序，则需要将“include mpif.h”改为“use mpi”，即MPI被定义为一个Fortran90调用的模块（程序 4）。

第二部分，定义程序中所需要的与MPI有关的变量。MPI_MAX_PROCESSOR_NAME是MPI预定义的宏，即某一MPI的具体实现中允许机器名字的最大长度，机器名放在变量processor_name中；整型变量myid和numprocs分别用来记录某一个并行执行的进程的标识和所有参加计算的进程的个数；namelen是实际得到的机器名字的长度；rc和ierr分别用来得到MPI过程调用结束后的返回结果和可能的出错信息。

第三部分，MPI程序的开始和结束必须是MPI_INIT和MPI_FINALIZE，分别完成MPI程序的初始化和结束工作。

第四部分，MPI程序的程序体，包括各种MPI过程调用语句和FORTRAN语句。MPI_COMM_RANK得到当前正在运行的进程的标识号，放在myid中；MPI_COMM_SIZE得到所有参加运算的进程的个数，放在numprocs中；MPI_GET_PROCESSOR_NAME得到运行本进程的机器的名称，结果放在processor_name中，它是一个字符串，而该字符串的长度放在namelen中；write语句是普通的FORTRAN语句，它将本进程的标识号，并行执行的进程的个数，运行当前进程的机器的名字打印出来，和一般的FORTRAN程序不同的是这些程序体中的执行语句是并行执行的，每一个进程都要执行。不妨指定本程序启动时共产生4个进程同时运行，而运行本程序的机器的机器名为“tp5”，4个进程都在tp5上运行，其标识分别为0，1，2，3，执行结果如图 8 所示，虽然这一MPI程序本身只有一条打印语句，但是由于它启动了四个进程同时执行，每个进程都执行打印操作，故而最终的执行结果有四条打印语句。本程序的执行流程可以用图 10表示。

```
      program main
      include 'mpif.h'
      character * (MPI_MAX_PROCESSOR_NAME) processor_name
      integer myid, numprocs, namelen, rc,ierr

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      call MPI_GET_PROCESSOR_NAME(processor_name, namelen, ierr)
      write(*,10) myid,numprocs,processor_name
10    FORMAT('Hello World! Process ',I2,' of ',I1,' on ', 20A)
      call MPI_FINALIZE(rc)
      end
```

程序 1 第一个FORTRAN77+MPI程序

```
Hello World! Process  1 of 4 on tp5
Hello World! Process  0 of 4 on tp5
Hello World! Process  2 of 4 on tp5
Hello World! Process  3 of 4 on tp5
```

图 8 第一个FORTRAN77+MPI程序在1台机器上的执行结果

如果该程序在4台不同的机器tp1，tp3，tp4，tp5上执行，则其最终的执行结果将如图 9 所示，即4个进程所运行的机器是不同的。由于4个进程同时执行，在本程序中没有限定打印语句的顺序，因此不管哪个进程的打印语句在前，哪个在后，都没有关系，只要有4条正确的输出语句即可。

```
Hello World! Process  0 of 4 on tp5
Hello World! Process  1 of 4 on tp1
Hello World! Process  2 of 4 on tp3
Hello World! Process  3 of 4 on tp4
```

图 9 第一个FORTRAN77+MPI程序在4台机器上的执行结果

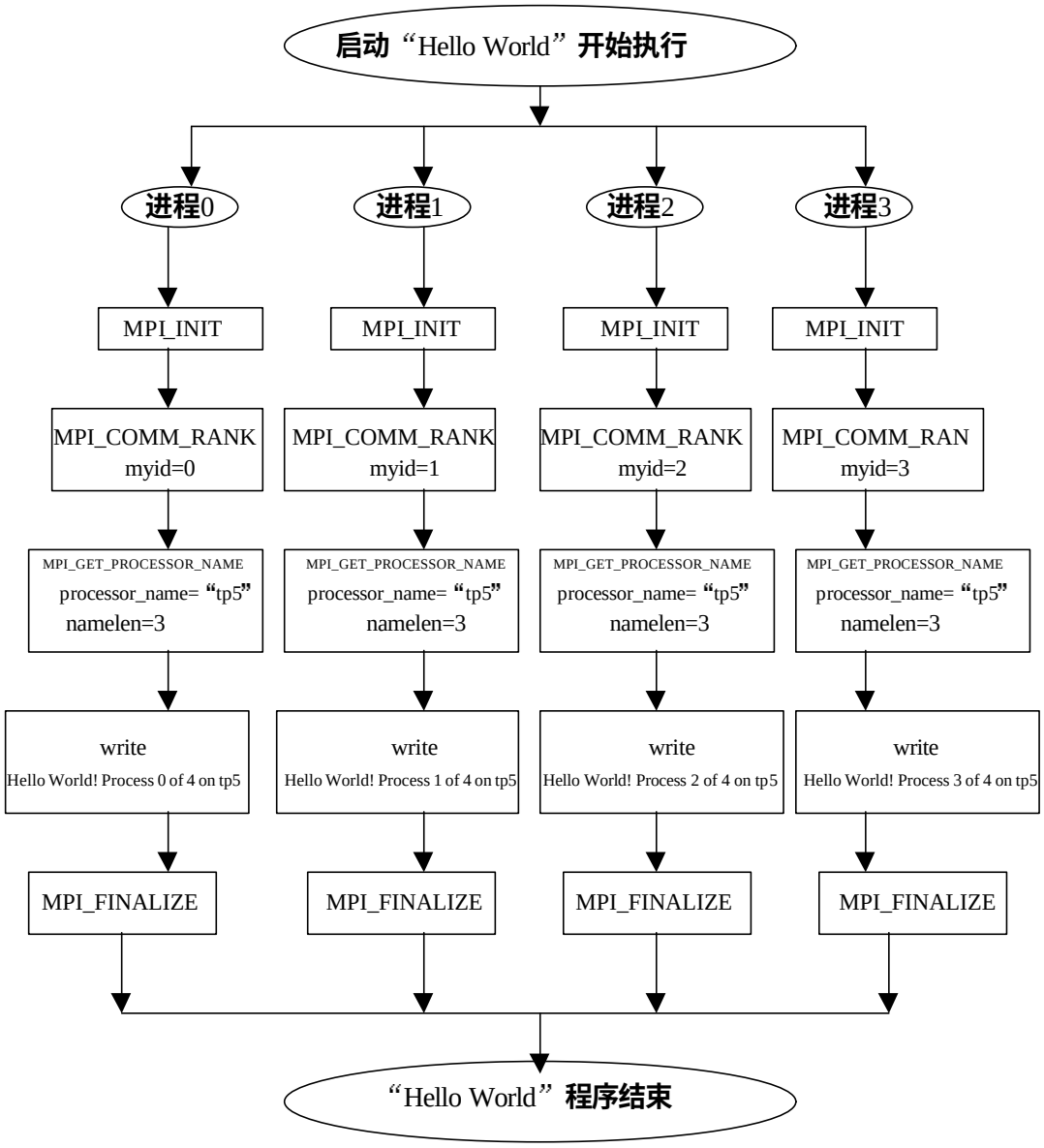


图 10 第一个FORTRAN77+MPI程序的执行流程

5.1.2 用C+MPI实现

如程序 3所示的第一个C+MPI并行程序。它的程序结构和FORTRAN77+MPI的完全相同，但对于不同的调用，在形式和语法上有所不同。下面分几个部分对它的结构进行介绍。

第一部分，首先要有MPI相对于C实现的头文件mpi.h，而不是mpif.h，这和FORTRAN77是不同的。

第二部分，定义程序中所需要的与MPI有关的变量。MPI_MAX_PROCESSOR_NAME是MPI预定义的宏，即某一MPI的具体实现中允许机器名字的最大长度，机器名放在变量

processor_name中，这和FORTRAN77中的没有区别；整型变量myid和numprocs分别用来记录某一个并行执行进程的标识和所有参加计算的进程的个数；namelen是实际得到的机器名字的长度。

第三部分，MPI程序的开始和结束必须是MPI_Init和MPI_Finalize，分别完成MPI程序的初始化和结束工作。对比FORTRAN77+MPI程序，这两个调用在FORTRAN77和C中所需要的参数是不同的，而且习惯上，在FORTRAN77中，所有的MPI调用均为大写（由于FORTRAN77源程序对大小写无关，因此使用小写的程序仍然是正确的，这里全用大写主要是遵守MPI书写FORTRAN77程序的惯例），而在C中则是以“MPI_”开头，后面的部分第一个字母大写，而其它的后续部分小写。

第四部分，MPI程序的程序体，包括各种MPI过程调用语句和C语句。MPI_Comm_rank得到当前正在运行的进程的标识号，放在myid中；MPI_Comm_size得到所有参加运算的进程的个数，放在numprocs中；MPI_Get_processor_name得到本进程运行的机器的名称，结果放在processor_name中，它是一个字符串，而该字符串的长度放在namelen中；fprintf语句将本进程的标识号，并行执行的进程的个数，本进程所运行的机器的名字打印出来，和一般的C程序不同的是这些程序体中的执行语句是并行执行的，每一个进程都要执行。不妨指定本程序启动时共产生4个进程同时运行，而运行本程序的机器的机器名为“tp5”，4个进程都在tp5上运行，其标识分别为0，1，2，3，执行结果如图 11所示，虽然这一MPI程序本身只有一条打印语句，但是由于它启动了四个进程同时执行，每个进程都执行打印操作，故而最终的执行结果有四条打印语句。本程序的执行流程和上面的FORTRAN77+MPI的实现版本是一样的。

```
Hello World! Process 0 of 4 on tp5
Hello World! Process 1 of 4 on tp5
Hello World! Process 3 of 4 on tp5
Hello World! Process 2 of 4 on tp5
```

图 11 第一个C+MPI程序在1台机器上的执行结果

如果该程序在4台不同的机器tp1，tp3，tp4，tp5上执行，则其最终的执行结果将如图 12所示，即4个进程所运行的机器是不同的。由于4个进程同时执行，在本程序中没有限定打印语句的顺序，因此不管哪个进程的打印语句在前，哪个在后，都没有关系，只要有4条正确的输出语句即可。对比FORTRAN77+MPI和C+MPI程序的输出结果，不难发现不管是在一台机器上运行，还是在多台机器上运行，其最终执行输出结果是完全一样的。

```
Hello World! Process 0 of 4 on tp5
Hello World! Process 1 of 4 on tp1
Hello World! Process 2 of 4 on tp3
Hello World! Process 3 of 4 on tp4
```

图 12 第一个C+MPI程序在4台机器上的执行结果

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
void main(argc,argv)
int argc;
char *argv[];
    {
        int myid, numprocs;
        int  namelen;
        char  processor_name[MPI_MAX_PROCESSOR_NAME];

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Get_processor_name(processor_name,&namelen);

        fprintf(stderr,"Hello World! Process %d of %d on %s\n",
        myid,  numprocs, processor_name);

        MPI_Finalize();
    }
```

程序 3 第一个C+MPI并行程序

```
      program  main
       use mpi
      character  *  (MPI_MAX_PROCESSOR_NAME)  processor_name
      integer  myid,  numprocs,  namelen,  rc,  ierr

      call  MPI_INIT(  ierr  )
       call  MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call  MPI_COMM_SIZE(  MPI_COMM_WORLD,  numprocs,  ierr  )
      call  MPI_GET_PROCESSOR_NAME(processor_name,  namelen,  ierr)
       print  *,"Hello World! Process ",myid," of ", numprocs, "  on", processor_name
        call MPI_FINALIZE(rc)
        end
```

程序 4 第一个Fortran90+MPI 程序

从上面的简单例子可以看出，一个MPI程序的框架结构可以用图 13表示。把握了其结构之后，下面的主要任务就是掌握MPI提供的各种通信方法与手段。

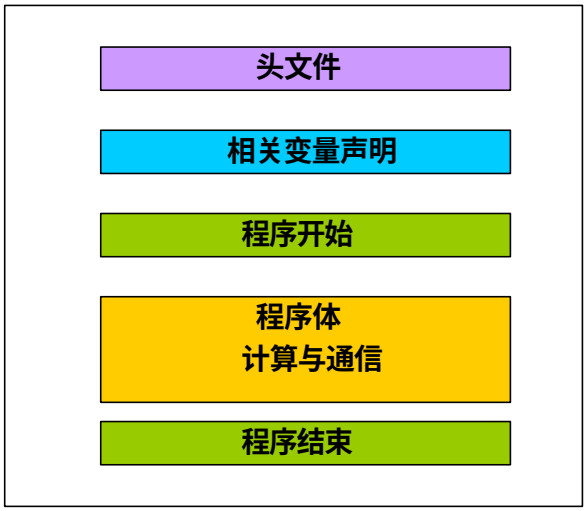


图 13　MPI程序的框架结构

5.2 MPI程序的一些惯例

下面在上述例子的基础上，介绍一下MPI的命名规则等通常的惯例。

所有MPI的名字都有前缀“MPI_”，不管是常量、变量还是过程或函数调用的名字都是这样。在自己编写的程序中不准说明以前缀“MPI_”开始的任何变量和函数，这样做的主要目的是为了避免与MPI可能的名字混淆。

FORTRAN形式的MPI调用，一般全为大写（虽然FORTRAN不区分大小写），而C形式的MPI调用，则为MPI_Aaaa_aaa的形式。

所有MPI的FORTRAN子程序在最后参数中都有一个返回代码，对于成功的返回代码值是MPI_SUCCESS，其他的错误代码是依赖于实现的。一些MPI操作是函数，它没有返回代码参数。

FORTRAN中的句柄以整型表示，二值变量是逻辑类型。FORTRAN的数组下标是以1开始，但在C中是以0开始。

除非明显说明，FORTRAN 77的MPI程序与ANSI FORTRAN 77标准标准相一致。但有些地方不同于 ANSI FORTRAN 77 标准，比如：

- MPI标识符限于三十个有效符号，而不是六个。
- MPI标识符可在第一个字符后包含下划线。
- 具有一个选择参数的MPI子程序可以用不同的参数类型来调用。
- 在一个包含文件mpif.h中提供所命名的常量。
- 在支持用户定义类型的系统中，鼓励生产商在mpif.h文件中提供类型说明。

5.3 小结

从上面的例子，我们对MPI程序已经有了一定的感性认识，由于MPI并行程序是在原来串行程序基础上的扩展，在许多地方和串行程序是相同的，串行程序设计的许多经验是可以应用到并行程序设计中的，但是同时我们必须注意，在设计MPI程序的时候，头脑中必须有程序并行执行的概念，而不是原来有序的串行执行，这才是串行和并行最主要的区别。

本章的主要目的是通过简单的例子，给出MPI程序的框架结构，使读者对MPI程序有一个简单的总体上的认识，为此，采用了最常见也是最简单的“Hello World”程序，它虽然还不涉及任何通信的成分，但是，它给出了MPI程序与原来的串性程序的主要区别，它可以同时打印出多个“Hello World”语句，而不是一个，它已经包含了SPMD（Single Program Multiple Data）程序的精髓。

第6章 六个接口构成的MPI子集

在MPI-1中，共有128个调用接口，在MPI-2中有287个，应该说MPI是比较庞大的，完全掌握这么多的调用对于初学者来说是比较困难的。但是，从理论上说，MPI所有的通信功能可以用它的6个基本的调用来实现，掌握了这6个调用，就可以实现所有的消息传递并行程序的功能，因此，首先在本章介绍这6个最基本的调用。

本章首先对这6个调用组成的完备子集进行介绍，然后用例子说明如何用这6个调用来完成基本的消息传递并行编程，最后介绍了MPI数据类型和MPI消息。

6.1 子集介绍

这一部分对6个调用构成的MPI完备子集进行介绍，在对具体的调用介绍之前，首先介绍MPI是如何对其FORTRAN 77和C的调用进行说明的。

6.1.1 MPI调用的参数说明

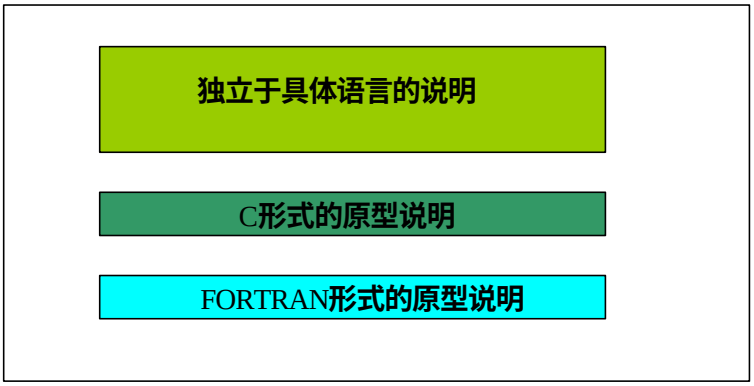


图 14 MPI调用的说明格式

如图 14所示，对于有参数的MPI调用，MPI首先给出一种独立于具体语言的说明，对各个参数的性质进行介绍，然后在给出它相对于FORTRAN 77和C的原型说明，在MPI-2中还给出了C++形式的说明。MPI对参数说明的方式有三种，分别是IN、OUT和INOUT。它们的含义分别是：

- IN（输入）：调用部分传递给MPI的参数，MPI除了使用该参数外不允许对这一参数做任何修改
- OUT（输出）：MPI返回给调用部分的结果参数，该参数的初始值对MPI没有任何意义
- INOUT（输入输出）：调用部分首先将该参数传递给MPI，MPI对这一参数引用、修改后，将结果返回给外部调用，该参数的初始值和返回结果都有意义

如果某一个参数在调用前后没有改变，比如某个隐含对象的句柄，但是该句柄指向的对象被修改了，这一参数仍然被说明为OUT或INOUT。MPI的定义在最大范围内避免INOUT参数的使用,因为这些使用易于出错，特别是对标量参数。

还有一种情况是MPI **函数的一个参数被一些并行执行的进程用作** IN，**而被另一些同时执行的进程用作**OUT，**虽然在语义上它不是同一个调用的输入和输出，这样的参数语法上也记为**INOUT。

当一个MPI**参数仅对一些并行执行的进程有意义而对其它的进程没有意义时，不关心该参数取值的进程可以将任意的值传递给该参数。**

在MPI**中**OUT**或**INOUT**类型的参数不能被其它的参数作为别名使用，如下定义一个 C 过程**

```
void copyIntBuffer( int *pin, int *pout, int len )
{    int i;
     for (i=0; i<len; ++i)   *pout++ =   *pin++;
}
```

那么在下面代码段中，对这个函数的调用使用了参数别名。

```
int a[10];
copyIntBuffer( a, a+3, 7);
```

虽然 C **语言中允许这样,但除非特别说明，**MPI**调用禁止这样使用，**FORTRAN77**是禁止使用别名参数的。**

在本书后面所有的MPI**调用说明中，首先给出的是**MPI**调用不依赖任何语言的说明，然后给出这个调用的标准**C**版本，最后给出这个函数的** FORTRAN 77 **版本。以**MPI_INIT**为例，**

MPI_INIT()

独立于语言的说明，对于这一个调用没有参数说明

```
int MPI_Init(int *argc, char ***argv)
```

C**语言的说明，对于**C**语言调用，需要给出参数**argc**和**argv**，注意这里给出的是**argc**和**argv

```
MPI_INIT(IERROR)
INTEGER IERROR
```

FORTRAN77**说明部分，对于所有的**FORTRAN77**调用，都有返回的错误代码**IERROR

在C**和**FORTRAN77**的说明中，对**void*,<type>**需要进行特殊说明。**MPI**的库和一般的**C**和**FORTRAN77**库在语法上基本上是相同的，但是对于**MPI**的调用，允许不同的数据类型使用相同的调用，比如对于数据的发送操作，整型、实型、字符型等都用一个相同的调用** MPI_SEND，**对于这样的数据类型在**C**和**FORTRAN77**的原型说明中，分别用**void *和 <type>**来表示，即用户可根据通信的要求，对不同的数据类型，可以用相同的调用。**

6.1.2 MPI初始化

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

MPI调用接口 1 MPI_INIT

MPI_INIT是MPI程序的第一个调用，它完成MPI程序所有的初始化工作，所有MPI程序的第一条可执行语句都是这条语句。

6.1.3 MPI结束

```
MPI_FINALIZE()
int MPI_Finalize(void)
MPI_FINALIZE(IERROR)
INTEGER IERROR
```

MPI调用接口 2 MPI_FINALIZE

MPI_FINALIZE是MPI程序的最后一个调用，它结束MPI程序的运行，它是MPI程序的最后一条可执行语句，否则程序的运行结果是不可预知的。

6.1.4 当前进程标识

```
MPI_COMM_RANK(comm,rank)
IN      comm    该进程所在的通信域（句柄）
OUT     rank    调用进程在comm中的标识号
int MPI_Comm_rank(MPI_Comm comm, int *rank)
MPI_COMM_RANK(COMM,RANK,IERROR)
INTEGER COMM,RANK,IERROR
```

MPI调用接口 3 MPI_COMM_RANK

这一调用返回调用进程在给定的通信域中的进程标识号，有了这一标识号，不同的进程就可以将自身和其它的进程区别开来，实现各进程的并行和协作。

6.1.5 通信域包含的进程数

```
MPI_COMM_SIZE(comm,size)
IN      comm    通信域（句柄）
OUT     size    通信域comm内包括的进程数（整数）
int MPI_Comm_size(MPI_Comm comm, int *size)
MPI_COMM_SIZE(COMM,SIZE,IERROR)
    INTEGER  COMM,SIZE,IERROR
```

MPI调用接口 4 MPI_COMM_SIZE

这一调用返回给定的通信域中所包括的进程的个数，不同的进程通过这一调用得知在给定的通信域中一共有多少个进程在并行执行。

6.1.6 消息发送

```
MPI_SEND(buf,count,datatype,dest,tag,comm)
IN      buf     发送缓冲区的起始地址(可选类型)
IN      count   将发送的数据的个数(非负整数)
IN      datatype 发送数据的数据类型(句柄)
IN      dest    目的进程标识号(整型)
IN      tag     消息标志(整型)
IN      comm    通信域(句柄)
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
comm)
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER  COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

MPI调用接口 5 MPI_SEND

MPI_SEND将发送缓冲区中的count个datatype数据类型的数据发送到目的进程，目的进程在通信域中的标识号是dest，本次发送的消息标志是tag，使用这一标志，就可以把本次发送的消息和本进程向同一目的进程发送的其它消息区别开来。

MPI_SEND操作指定的发送缓冲区是由count个类型为datatype的连续数据空间组成，起始地址为buf。注意这里不是以字节计数，而是以数据类型为单位指定消息的长度，这样就独立于具体的实现，并且更接近于用户的观点。

其中datatype数据类型可以是MPI的预定义类型，也可以是用户自定义的类型（将在后面的部分介绍）。通过使用不同的数据类型调用MPI_SEND，可以发送不同类型的数据。

6.1.7 消息接收

MPI_RECV从指定的进程source接收消息，并且该消息的数据类型和消息标识和本接收进程指定的datatype和tag相一致，接收到的消息所包含的数据元素的个数最多不能超过count。

接收缓冲区是由count个类型为datatype的连续元素空间组成，由datatype指定其类型，起始地址为buf。接收到消息的长度必须小于或等于接收缓冲区的长度，这是因为如果接收到的数据过大，MPI没有截断，接收缓冲区会发生溢出错误，因此编程者要保证接收缓冲区的长度不小于发送数据的长度。如果一个短于接收缓冲区的消息到达，那么只有相应于这个消息的那些地址被修改。count可以是零，这种情况下消息的数据部分是空的。

其中datatype数据类型可以是MPI的预定义类型，也可以是用户自定义的类型。通过指定不同的数据类型调用MPI_RECV，可以接收不同类型的数据。

MPI_RECV(buf,count,datatype,source,tag,comm,status)

OUT	buf	接收缓冲区的起始地址(可选数据类型)
IN	count	最多可接收的数据的个数(整型)
IN	datatype	接收数据的数据类型(句柄)
IN	source	接收数据的来源即发送数据的进程的进程标识号(整型)
IN	tag	消息标识，与相应的发送操作的表示相匹配相同(整型)
IN	comm	本进程和发送进程所在的通信域(句柄)
OUT	status	返回状态 (状态类型)

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

MPI调用接口 6　MPI_RECV

6.1.8 返回状态status

返回状态变量status用途很广，它是MPI定义的一个数据类型，使用之前需要用户为它分配空间。

在C实现中，状态变量是由至少三个域组成的结构类型，这三个域分别是：MPI_SOURCE，MPI_TAG和MPI_ERROR。它还可以包括其它的附加域。这样通过对status.MPI_SOURCE，status.MPI_TAG和status.MPI_ERROR的引用，就可以得到返回状态中所包含的发送数据进程的标识，发送数据使用的tag标识和本接收操作返回的错误代码。

在FORTRAN实现中，status是包含MPI_STATUS_SIZE个整型的数组，status(MPI_SOURCE)，status(MPI_TAT)和status(MPI_ERROR)分别表示发送数据的进程标识，发送数据使用tag标识和该接收操作返回的错误代码。

除了以上三个信息之外，对status变量执行MPI_GET_COUNT调用可以得到接收到的消息的长度信息。这在后面的部分会对这一调用进行介绍。

6.1.9 一个简单的发送和接收的例子

下面介绍一个简单的同时包含发送和接收调用的例子，其中一个进程（进程0）向另一个进程（进程1）发送一条消息，该消息是一个字符串“Hello, process 1”，进程1在接收到该消息后，将这一消息打印到屏幕上。

```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
    char  message[20];
     int myrank;
    MPI_Init( &argc, &argv );
    /* MPI程序的初始化*/
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    /*  得到当前进程的标识*/
    if (myrank == 0) /*  若是 0  进程*/
     {
    /*  先将字符串拷贝到发送缓冲区message中，然后调用MPI_Send语句将它发出，用strlen(message)指定消息的长度，用MPI_CHAR指定消息的数据类型，1指明发往进程1，使用的消息标识是99，MPI_COMM_WORLD是包含本进程（进程0）和接收消息的进程（进程1）的通信域。发送方和接收方必须在同一个通信域中。由通信域来统一协调和控制消息的发送和接收*/
          strcpy(message,"Hello,  process 1");
          MPI_Send(message,  strlen(message), MPI_CHAR, 1,
           99,MPI_COMM_WORLD);
    }
    else if(myrank==1) /*  若是进程  1 */
    {
/*进程1直接执行接收消息的操作，这里它使用message作为接收缓冲区，由此可见，对于同一个变量，在发送进程和接收进程中的作用是不同的。它指定接收消息的最大长度为20，消息的数据类型为MPI_CHAR字符型，接收的消息来自进程0，而接收消息携带的标识必须为99，使用的通信域也是MPI_COMM_WORLD，接收完成后的各种状态信息存放在status中。接收完成后，它直接将接收到的字符串打印在屏幕上。*/
          MPI_Recv(message,  20,  MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
          printf("received  :%s:",  message);
    }
    MPI_Finalize();
```

```
    /* MPI程序结束*/
}
```

程序 5 简单的发送接收程序

6.2 MPI预定义数据类型

在FORTRAN77中，MPI预定义了如下数据类型可以直接使用，它们和FORTRAN77数据类型的对应关系如表格 3所示。

表格 3 MPI预定义数据类型与FORTRAN77数据类型的对应关系

MPI预定义数据类型	相应的FORTRAN77数据类型
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	无对应类型
MPI_PACKED	无对应类型

同样地，MPI也有和C对应的数据类型，如表格 4所示：

表格 4 MPI预定义数据类型与C数据类型的对应关系

MPI预定义数据类型	相应的C数据类型
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	无对应类型
MPI_PACKED	无对应类型

MPI_BYTE和MPI_PACKED数据类型没有相应于一个FORTRAN77 或C的数据类型。类型MPI_BYTE的一个值由一个字节组成(8个二进制位)。一个字节不同于一个字符，因为对于字符表示不同的机器，可以用一个以上的字节表示字符；另一方面，在所有的机器上，一个字节有相同的二进制值。

MPI要求支持上述数据类型，以匹配Fortran 77 和ANSI C的基本数据类型。如果宿主语言有附加的数据类型，那么MPI应提供附加的相应数据类型，如表格 5所示。

表格 5 附加的MPI数据类型

附加的MPI数据类型	相应的C数据类型
MPI_LONG_LONG_INT	long long int

附加的MPI数据类型	相应的FORTRAN77数据类型
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_REAL2	REAL*2
MPI_REAL4	REAL*4
MPI_REAL8	REAL*8
MPI_INTEGER1	INTEGER*1
MPI_INTEGER2	INTEGER*2
MPI_INTEGER4	INTEGER*4

6.3 MPI数据类型匹配和数据转换

6.3.1 MPI类型匹配规则

在MPI消息传递的整个过程中，什么时候要涉及类型匹配？首先看MPI消息传递的过程，如图 15所示。

图 15 MPI消息传递过程

MPI的消息传递过程可以分为三个阶段：

① 消息装配，将发送数据从发送缓冲区中取出，加上消息信封等形成一个完整的消息。

② 消息传递，将装配好的消息从发送端传递到接收端。

③ 消息拆卸，从接收到的消息中取出数据送入接收缓冲区。

在这三个阶段，都需要类型匹配：①在消息装配时，发送缓冲区中变量的类型必须和相应的发送操作指定的类型相匹配；②在消息传递时，发送操作指定的类型必须和相应的接收操作指定的类型相互匹配；③在消息拆卸时，接收缓冲区中变量的类型必须和接收操作指定的类型相匹配。

以上指出了在什么时候需要类型匹配，进一步，类型匹配到底包括哪些方面呢，在MPI中，类型匹配有两个方面的意思：① 宿主语言的类型和通信操作所指定的类型相匹配；② 发送方和接收方的类型相匹配。

对于类型匹配的第一个方面，比如在FORTRAN77中，声明为INTEGER类型的变量在发送和接收时要使用MPI_INTEGER与之相对应，声明为REAL类型的变量在发送和接收时要使用MPI_REAL与之相对应，前面部分给出的FORTRAN77 与MPI类型的对应关系就是为了满足类型匹配的要求；同样在C语言中，int类型要和MPI_INT相对应，float要和MPI_FlOAT相对应，这些对应关系在前面已经介绍了。

对于类型匹配的第二个方面，要求在发送方和接收方对数据类型的指定必须是一样的，即发送方用MPI_INTEGER，则接收方也必须使用MPI_INTEGER，发送方用MPI_REAL，则接收方也必须用MPI_REAL，对于C语言，虽然有些系统中int和long有相同的表示，但是MPI认为MPI_INT和MPI_LONG是不同的类型，即MPI_INT和MPI_LONG是不匹配的，发送方和接收方必须同时使用MPI_INT或MPI_LONG。

上述类型匹配规则的例外是对于MPI提供的MPI_BYTE和MPI_PACKED，它们可以和任何以字节为单位的存储相匹配，包含这些字节的类型是任意的，MPI_TYPE用于不加修改地传送内存中的二进制值，MPI_PACK用于数据的打包和解包（MPI_UNPACK）。

```
…
REAL a(20),b(20)
…
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)
END IF
```

程序 6 正确的类型匹配（MPI_REAL**对应于**MPI_REAL）

```
…
REAL a(20),b(20)
…
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), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

程序 7 不正确的类型匹配（MPI_REAL**和**MPI_BYTE**不匹配**）

```
…
REAL a(20),b(20)
…
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

程序 8 正确的无类型数据匹配（MPI_BYTE和MPI_BYTE对应）

归纳起来，类型匹配规则可以概括为：

- 有类型数据的通信，发送方和接收方均使用相同的数据类型。
- 无类型数据的通信，发送方和接收方均以MPI_BYTE作为数据类型。
- 打包数据的通信，发送方和接收方均使用MPI_PACKED。

程序 6给出了有类型数据的正确匹配，程序 7的类型匹配是不正确的，因为MPI_REAL不能和MPI_BYTE进行匹配，程序 8是正确的无类型数据匹配，发送方和接收方均使用MPI_BYTE。

类型MPI_CHARACTER和FORTRAN 77 的CHARACTER变量的一个字符相对应，而不是存储在这个变量中的全部字符。类型为CHARACTER的FORTRAN 77变量是作为字符数组被传送的。

```
…
CHARACTER*10 a
CHARACTER*10 b
…
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
    CALL MPI_RECV(b(6), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
END IF
```

程序 9 MPI_CHARACTER数据类型

如程序 9所示，进程1中的字符串b的最后五个字符被进程0中的字符串a的前五个字符替代。

Fortran的一个字符变量是定长的字符串，没有特别的终结符号。关于怎样表示字符、怎样存储它们的长度没有固定的约定。有些编译器把一个字符参数作为一对参数传送给一个程序，其中之一是保存这个串的地址，另一个是保存串的长度。如果这个通信缓冲区包含CHARACTER类型的变量，那么关于他们的长度将不被传送给MPI程序。这个问题迫使我们给MPI调用提供显式的字符长度的信息。

有些编译器把Fortran中CHARACTER类型的参数作为一个结构传给实际串一个长度和一个指针。在这样的环境中，MPI调用为得到这个串须间接引用这个指针。

6.3.2 数据转换

所谓的数据转换包括两个方面的意思：

- 数据类型的转换
- 数据表示的转换

数据类型的转换是指改变一个值的数据类型，比如将实型转换为整型（通过舍入操作）或将整型转换为实型等。

而数据表示的转换是指改变一个值的二进制表示，比如高字节和低字节顺序的改变，将

浮点数从32为表示改变为64为表示等。

由于MPI严格要求类型匹配，所以在MPI中不存在数据类型转换的问题。但是，MPI必须实现数据表示的转换，这是因为MPI的目的之一是对异构环境的支持，在异构系统中，不同的系统其数据的内部表示往往是不同的，因此MPI必须负责实现这些不同表示之间的相互转换。

在MPI中，没有限定数据表示转换的细节，但总的目标是希望这样的转换保留整型、逻辑以及字符值不变，而把浮点值转为在目标系统上能表示的最接近的值。

在浮点转换过程，上溢和下溢可能发生。当一个值能在一个系统中表示而不能在另一个系统中表示时，整型或字符的转换也可导致异常。在表示转换过程中，一个异外发生会导致一个通信失败。在发送操作、接收操作或两者也都发生错误。

如果在一个消息中发送的一个值是无类型的（例如,MPI_BYTE类型），那么在接收者存储的字节的二进制表示与接收者接收的字节的二进制表示一样。无论发送者和接收者运行在同一环境或不同环境，这都是正确的。不要求表示转换。

当一个MPI程序在同构系统中运行，其所有进程运行在同一环境时，没有转换发生。考虑前面的三个例子。因为a和b是实型的跨度大于10的数组，第一个程序是正确的。如果发送者和接收者在不同的环境中执行，那么从发送缓冲区取出的10个实型值，在存到接收缓冲区以前被转换为接收者的实型表示。当从发送缓冲区取出的实型元素的个数等于接收缓冲区所存的实型元素的个数时，存储的字节数不必等于接收的字节数。例如，发送者可以使用四个字节表示实型数，接收者使用八个子字节表示实型数。

第二个程序是错误的，它的动作无定义。

第三个程序是正确的。既使发送者和接收者在不同的环境运行，从发送缓冲区装入的四十个字节将被存在接收缓冲区。发送的消息与接收的消息有完全相同的长度(按字节)和相同的二进制表示。如果a和b是不同的类型，或他们类型相同但使用不同的数据表示，那么接收缓冲区所存的各位可以译码出不同于发送缓冲区的译码值。

数据表示的转换也应用于一个消息的信封：源，目的和标识都需要被转换为整型。

6.4 MPI消息

6.4.1 MPI消息的组成

MPI消息包括信封和数据两个部分，信封指出了发送或接收消息的对象及相关信息，而数据是本消息将要传递的内容。信封和数据又分别包括三个部分。可以用一个三元组来表示。

信封：<源/目，标识，通信域>

数据：<起始地址，数据个数，数据类型>

以MPI_SEND和MPI_RECV为例，图 17和图 17分别给出了它们的信封和数据部分。

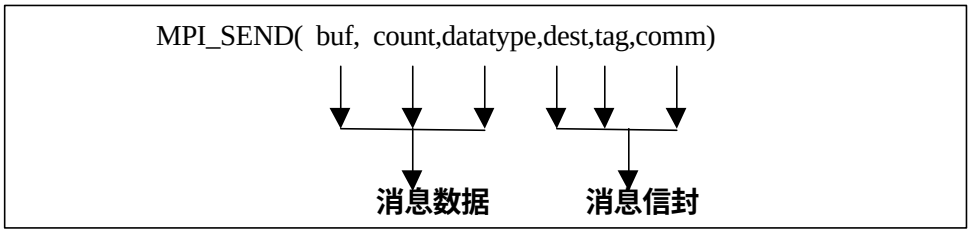


图 16　MPI_SEND 语句的消息信封和消息数据

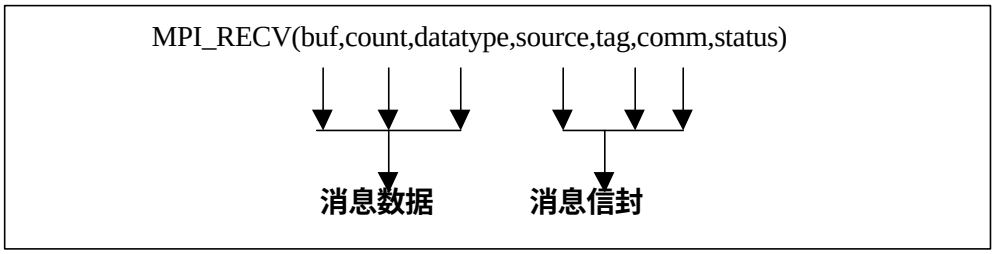


图 17　MPI_RECV语句的消息信封和消息数据

在消息信封中除了源/目外，为什么还有tag标识呢？这是因为，当发送者发送两个相同类型的数据给同一个接收者时，如果没有消息标识，接收者将无法区别这两个消息。如图 18 所示。

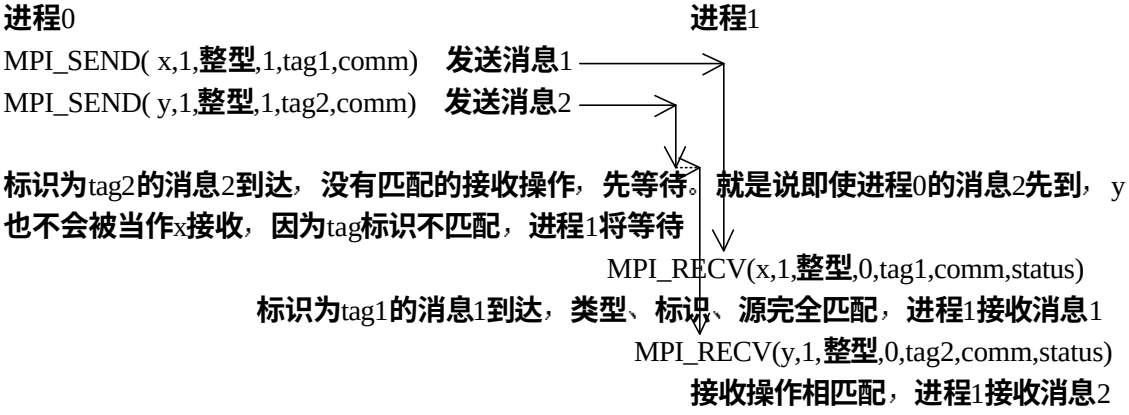


图 18　tag在MPI消息发送和接收中的作用

6.4.2 任意源和任意标识

一个接收操作对消息的选择是由消息的信封管理的。如果消息的信封与接收操作所指定的值source,tag和comm相匹配,那么这个接收操作能接收这个消息。接收者可以给source指定一个任意值MPI_ANY_SOURCE，标识任何进程发送的消息都可以接收，即本接收操作可以匹配任何进程发送的消息，但其它的要求还必须满足，比如tag的匹配；如果给tag一个任意值MPI_ANY_TAG，则任何tag都是可接收的。在某种程度上，类似于统配符的概念。MPI_ANY_SOURCE和MPI_ANY_TAG可以同时使用或分别单独使用。但是不能给 comm指定任意值。如果一个消息被发送到接收进程，接收进程有匹配的通信域，有匹配的 source (或其source = MPI_ANY_SOURCE)，有匹配的tag(或其tag = MPI_ANY_TAG)，那么这个消息能被这个接收操作接收。

由于MPI_ANY_SOURCE和MPI_ANY_TAG的存在，导致了发送操作和接收操作间的不对称性，即一个接收操作可以接收任何发送者的消息，但是对于一个发送操作，则必须指明一个单独的接收者。

MPI允许发送者=接收者（Source = destination），即一个进程可以给自己发送一个消息。但是这种操作要注意死锁的产生。

6.4.3 MPI通信域

MPI通信域包括两部分：进程组和通信上下文。进程组即所有参加通信的进程的集合，如果一共有N个进程参加通信，则进程的编号从0到N-1；通信上下文提供一个相对独立的通信区域，不同的消息在不同的上下文中进行传递，不同上下文的消息互不干涉，通信上下文可以将不同的通信区别开来。

一个预定义的通信域MPI_COMM_WORLD由MPI提供。MPI初始化后，便会产生这一描述子，它包括了初始化时可得的全部进程，进程是由它们在MPI_COMM_WORLD组中的进程号所标识。

用户可以在原有的通信域的基础上，定义新的通信域。通信域为库和通信模式提供一种重要的封装机制。他们允许各模式有其自己的独立的通信域，和它们自己的进程计数方案。

6.5 小结

MPI调用接口的总数虽然庞大，但是根据我们实际编写MPI程序的经验，最常使用的MPI调用的个数却是十分有限的。因此本章专门就最常使用的MPI调用拿出来进行讲解，目的就是希望读者能够较容易的进入MPI，并快速掌握MPI的基本程序设计方法。

虽然本章介绍的调用很少，但是它们涉及到的MPI程序设计的基本因素：数据类型、消息、通信域等，这些都是必须掌握的内容。

掌握了MPI程序设计的基本规则和内容，对于其它的众多的MPI调用，学习起来就会相对简单。

第7章 简单的MPI程序示例

本章在前面一章介绍的关于MPI基本的通信语句和相关知识的基础上，给出了几个简单的例子。这些例子只是实现一些简单的功能，代码量很小，很容易阅读和理解，目的是让读者从中学会使用MPI来实现一定的功能；同时这些例子包含了一些新的调用，读者可以在掌握例子的过程中学习更多的MPI调用。

7.1 用MPI实现计时功能

在MPI程序中，经常会用到时间函数，比如用来统计程序运行的时间，或根据时间的不同选取不同的随机数种子，或根据时间的不同对程序的执行进行控制等。因此这里首先介绍MPI提供的时间函数调用，然后给出简单的应用例子。

```
MPI_WTIME()
double MPI_Wtime(void)
DOUBLE PRECISION MPI_WTIME()
```

MPI调用接口 7 MPI_WTIME

MPI_WTIME返回一个用浮点数表示的秒数，它表示从过去某一时刻到调用时刻所经历的时间。这样如果需要对特定的部分进行计时，一般采取的方式是：

```
double  starttime, endtime;
...
starttime  = MPI_Wtime()
    需计时部分
endtime  = MPI_Wtime()
printf("That tooks %f secodes\n", endtime-starttime);
```

程序 10 对特定的部分进行计时

```
MPI_WTICK（）
double MPI_Wtick（）
DOUBLE PRECISION MPI_WTICK（）
```

MPI调用接口 8 MPI_WTICK

MPI_WTICK返回MPI_WTIME的精度，单位是秒，可以认为是一个时钟滴答所占用的

时间。

下面的程序测试MPI**的时间函数是否正确。**

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "test.h"

int main( int argc, char **argv )
{
    int      err = 0;
    double  t1, t2;
    double  tick;
    int      i;

    MPI_Init( &argc, &argv );
    t1 = MPI_Wtime();/*得到当前时间t1*/
    t2 = MPI_Wtime();/*得到当前时间t2*/
    if (t2 - t1 > 0.1 || t2 - t1 < 0.0) {
    /* 若连续的两次时间调用得到的时间间隔过大，这里是超过0.1秒，或者后调用的函数得到的时间比先调用的时间小，则时间调用有错*/
    err++;
    fprintf( stderr,
            "Two successive calls to MPI_Wtime gave strange results: (%f) (%f)\n",
            t1, t2 );
    }
/* 循环测试10次，每次循环调用两次时间函数，两次时间调用的时间间隔是1秒 */
    for (i = 0; i<10; i++) {
            t1 = MPI_Wtime();/*计时开始*/
            sleep(1);/*睡眠1秒钟*/
            t2 = MPI_Wtime();/*计时结束*/
            if (t2 - t1 >= (1.0 - 0.01) && t2 - t1 <= 5.0) break;
            /* 两次计时得到的时间间隔合理，则退出*/
            if (t2 - t1 > 5.0) i = 9;
            /* 若两次计时得到的时间间隔过大，则改变循环计数变量的值，迫使程序从循环退出*/
    }
    /* 从上知，若计时函数正确，则不需循环10次程序即从循环退出，否则会重复执行到10次*/
    if (i == 10) {
    /*   计时函数不正确 */
            fprintf( stderr,  "Timer around sleep(1) did not give 1 second; gave %f\n",t2 - t1 );
            err++;
```

```
    }
    tick = MPI_Wtick();
    /* 得到一个时钟滴答的时间*/
    if (tick > 1.0 || tick < 0.0) {
    /* 该时间太长或者为负数，则该时间不正确*/
        err++;
        fprintf( stderr, "MPI_Wtick gave a strange result: (%f)\n", tick );
    }
    MPI_Finalize( );
}
```

程序 11 MPI**时间函数的测试**

7.2 获取机器的名字和MPI版本号

在实际使用MPI**编写并行程序的过程中，经常要将一些中间结果或最终的结果输出到程序自己创建的文件中，对于在不同机器上的进程，常希望输出的文件名包含该机器名，或者是需要根据不同的机器执行不同的操作，这样仅仅靠进程标识**rank**是不够的，**MPI**为此提供了一个专门的调用，使各个进程在运行时可以动态得到该进程所运行机器的名字。**

```
MPI_GET_PROCESSOR_NAME（name, resultlen）
    OUT  name      当前进程所运行机器的名字
    OUT  resultlen   返回名字的的长度（以可打印字符的形式）
int MPI_Get_processor_name ( char *name, int *resultlen)
MPI_GET_PROCESSOR_NAME（NAME, RESULTLEN, IERROR）
    CHARACTER  *(*) NAME
    INTEGER  RESULTLEN,  IERROR
```

MPI**调用接口** 9　MPI_GET_PROCESSOR_NAME

MPI_GET_PROCESSOR_NAME**调用返回调用进程所在机器的名字。**

```
MPI_GET_VERSION(version, subversion)
OUT      version
OUT      subversion
int MPI_Get_version(int * version, int * subversion)
MPI_GET_VERSION(VERSION, SUBVERSION,IERROR)
   INTEGER  VERSION,  SUBVERSION,  IERROR
```

MPI**调用接口** 10　MPI_GET_VERSION

MPI_GET_VERSION**返回**MPI**的主版本号**version**和次版本号**subversion。

下面的小例子测试得到机器的名字和MPI**的版本号。**

```
      program  main
      include  'mpif.h'
      character*(MPI_MAX_PROCESSOR_NAME)  name
      integer    resultlen,  version, subversion, ierr

     call  MPI_Init(  ierr  )
     name  =  "  "
C    首先将名字赋为空

     call  MPI_Get_processor_name(  name,  resultlen,  ierr  )
C     得到机器的名字name和该名字的字符长度resultlen
      call  MPI_GET_VERSION(version,  subversion,ierr)
C     得到MPI的版本号
     errs  =  0
      do  i=resultlen+1, MPI_MAX_PROCESSOR_NAME
        if  (name(i:i)  .ne.  "  ")  then
C     若返回的名字name的resultlen后还有非空字符，则认为该名字有错误
            errs = errs + 1
         endif
      enddo
      if (errs .gt. 0) then
         print  *, 'Non-blanks after name'
      else
         print  *, name, " MPI version",version, ".", subversion
      endif
     call  MPI_Finalize(  ierr  )
      end
```

程序 12 获取当前机器名和MPI版本号

7.3 是否初始化及错误退出

在MPI程序中唯一一个可以用在MPI_INIT之前的MPI调用是MPI_INITALIZED，它的功能就是判断MPI_INIT是否已经执行。

```
MPI_INITALIZED(flag)
OUT  flag     MPI_INIT是否已执行标志。
int MPI_Initialized(int *flag)
MPI_INITALIZED(FLAG, IERROR)
        LOGICAL  FLAG
        INTEGER  IERROR
```

MPI调用接口 11 MPI_INITALIZED

MPI_INITALIZED**判断当前进程是否已经调用了**MPI_INIT，**若已调用，则**flag=true，**否则**flag=false。

在编写MPI**程序的过程中，若发现已出现无法恢复的严重错误，因而只好退出**MPI**程序的执行，**MPI**提供了这样的调用，并且在退出时可以返回给调用环境一个错误码。**

```
MPI_ABORT(comm, errorcode)
        IN comm  退出进程所在的通信域
        IN errorcode  返回到所嵌环境的错误码
int MPI_Abort(MPI_Comm comm, int errorcode)
MPI_ABORT(COMM, ERRORCODE, IERROR)
      INTEGER  COMM,  ERRORCODE,  IERROR
```

MPI**调用接口** 12　MPI_ABORT

MPI_ABORT **使通信域**comm**中的所有进程退出。本调用并不要求外部环境对错误码采取任何动作。**

下面的例子将指定的master**结点杀掉。**

```
#include "mpi.h"
#include <stdio.h>
/*  本例子分两种情况，一种是masternode == 0，是缺省的情况，另一种是 masternode != 0，是指定最后一个进程为主进程的情况。
  */
int main( int argc, char **argv )
{
    int node, size, i;
    int masternode = 0;
/*  设置缺省初始值*/
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &node);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /*检查参数  */
   for (i=1; i<argc; i++) {
          fprintf(stderr,"myid=%d,procs=%d,argv[%d]=%s\n",node,size,i,argv[i]);
          if (argv[i] && strcmp( "lastmaster", argv[i] ) == 0) {
              masternode  =  size-1;
       /*  将最后一个进程设置为master*/
          }
    }
    if(node == masternode) {
       /*  由master进程执行退出操作*/
       fprintf(stderr,"myid=%d  is masternode Abort!\n",node);
       MPI_Abort(MPI_COMM_WORLD,  99);
    }
```

```
    else {
      /* 非master进程等待*/
      fprintf(stderr,"myid=%d  is not masternode Barrier!\n",node);
       MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

程序 13　MPI主动退出执行的例子

7.4 数据接力传送

下面给出一个数据接力传送的例子，数据的传送过程如图 19所示。

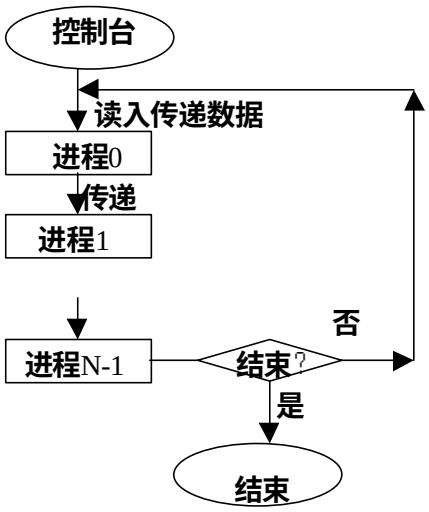


图 19 数据在进程间的接力传送

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{
      int  rank, value, size;
      MPI_Status  status;
     MPI_Init(  &argc,  &argv  );
     MPI_Comm_rank(  MPI_COMM_WORLD,  &rank  );
      MPI_Comm_size( MPI_COMM_WORLD, &size );
      /* 得到当前进程标识和总的进程个数*/
      do {
      /* 循环执行，直到输入的数据为负时才退出*/
```

```
    if (rank == 0) {
   fprintf(stderr, "\nPlease give new value=");
    /*进程0读入要传递的数据*/
   scanf( "%d", &value );
              fprintf(stderr,"%d    read    <-<-   (%d)\n",rank,value);
              if (size>1) {
             MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
                    fprintf(stderr,"%d    send   (%d)->->    %d\n",
                                rank,value,rank+1);
               /*  若不少于一个进程，则向下一个进程传递该数据*/
              }
}
else {
   MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
           &status );
    /*  其它进程从前一个进程接收传递过来的数据*/
              fprintf(stderr,"%d    receive (%d)<-<-    %d\n",rank,value,rank-1);
   if (rank < size - 1) {
    MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
                    fprintf(stderr,"%d    send   (%d)->->    %d\n",
                                rank,value,rank+1);
    /*若当前进程不是最后一个进程，则将该数据继续向后传递*/
              }
}
    MPI_Barrier(MPI_COMM_WORLD);
    /*  执行一下同步，加入它主要是为了将前后两次数据传递分开*/
    } while ( value>=0);
    MPI_Finalize( );
}
```

程序 14 数据在进程间的接力传送

用7个进程运行该程序，分别输入76和-3，执行结果如图 20所示。

```
Please give new value=76
0       read          <-<-     (76)
0       send      (76)->->      1
1     receive  (76)<-<-         0
1       send      (76)->->      2
2     receive  (76)<-<-         1
2       send      (76)->->      3
3     receive  (76)<-<-         2
3       send      (76)->->      4
4     receive  (76)<-<-         3
4       send      (76)->->      5
5     receive  (76)<-<-         4
5       send      (76)->->      6
6     receive  (76)<-<-         5
```

第一轮接力

```
Please give new value=-3
0       read          <-<-     (-3)
0       send      (-3)->->       1
1     receive  (-3)<-<-          0
2     receive  (-3)<-<-          1
3     receive  (-3)<-<-          2
4     receive  (-3)<-<-          3
4       send      (-3)->->       5
5     receive  (-3)<-<-         4
6     receive  (-3)<-<-         5
1       send      (-3)->->       2
2       send      (-3)->->       3
3       send      (-3)->->       4
5       send      (-3)->->       6
```

第二轮接力

图 20 接力程序的输出结果

7.5 任意进程间相互问候

在许多情况下需要任意两个进程之间都进行数据的交换，下面给出一个例子，任意进程都向其它的进程问好（图 21）。

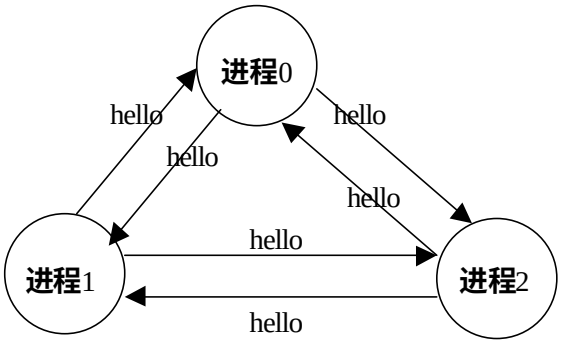


图 21 任意进程间相互问候

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
void Hello( void );
int main(int argc, char *argv[])
{
      int me, option, namelen, size;
     char  processor_name[MPI_MAX_PROCESSOR_NAME];
     MPI_Init(&argc,  &argv);
      MPI_Comm_rank(MPI_COMM_WORLD,&me);
      MPI_Comm_size(MPI_COMM_WORLD,&size);
      /*得到当前进程标识和总的进程数*/
    if (size < 2) {
      /*若总进程数小于2，则出错退出*/
            fprintf(stderr, "systest requires at least 2 processes" );
            MPI_Abort(MPI_COMM_WORLD,1);
      }
      MPI_Get_processor_name(processor_name,&namelen);
      /*得到当前机器名字*/
    fprintf(stderr,"Process %d is alive on %s\n", me, processor_name);
      MPI_Barrier(MPI_COMM_WORLD);
      /*同步*/
      Hello();
      /*调用问候过程*/
      MPI_Finalize();
}
```

```
void Hello( void )
/*任意两个进程间交换问候信息，问候信息由发送进程标识和接收进程标识组成*/
{
     int nproc, me;
    int type = 1;
     int buffer[2], node;
     MPI_Status status;
     MPI_Comm_rank(MPI_COMM_WORLD, &me);
     MPI_Comm_size(MPI_COMM_WORLD, &nproc);
      /*得到当前进程标识和总的进程数*/
     if (me == 0) {
      /* 进程0负责打印提示信息*/
      printf("\nHello test from all to all\n");
      fflush(stdout);
      }
    for (node = 0; node<nproc; node++) {
      /*循环对每一个进程进行问候*/
           if (node != me) {
           /* 得到一个和自身不同的进程标识*/
          buffer[0] = me; /*将自身标识放入消息中*/
          buffer[1] = node; /*将被问候的进程标识也放入消息中*/
           MPI_Send(buffer, 2, MPI_INT, node, type, MPI_COMM_WORLD);
           /*首先将问候消息发出*/
           MPI_Recv(buffer, 2, MPI_INT, node, type, MPI_COMM_WORLD, &status);
           /*然后接收被问候进程对自己发送的问候消息*/
          if ( (buffer[0] != node) || (buffer[1] != me) ) {
                /*若接收到的消息的内容不是问候自己的或不是以被问候方的身份问候自
己，则出错*/
                (void) fprintf(stderr, "Hello: %d!=%d or %d!=%d\n",
                    buffer[0], node, buffer[1], me);
                printf("Mismatch on hello process ids; node = %d\n", node);
           }
          printf("Hello from %d to %d\n", me, node);
           /*打印出问候对方成功的信息*/
           fflush(stdout);
           }
     }
}
```

程序 15 任意进程间相互问候

7.6 任意源和任意标识的使用

在接收操作中，通过使用任意源和任意tag**标识，使得该接收操作可以接收任何进程以任何标识发送给本进程的数据，但是该消息的数据类型必须和接收操作的数据类型相一致。**

下面给出一个使用任意源和任意标识的例子（图 22**）。其中**ROOT**进程（进程**0**）接收来自其它所有进程的消息，然后将各消息的内容，消息来源和消息标识打印出来。**

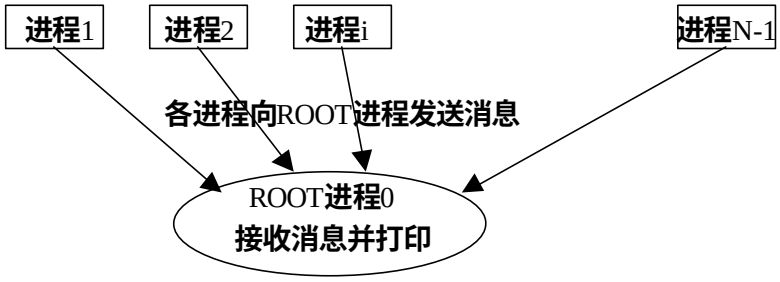


图 22 **接收任意源和任意标识的消息**

```
#include "mpi.h"
#include <stdio.h>
int main(argc, argv)
int argc;
char **argv;
{
    int  rank, size, i, buf[1];
    MPI_Status  status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    if (rank == 0) {
        for (i=0; i<100*(size-1); i++) {
        MPI_Recv( buf, 1, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status );
        printf( "Msg=%d  from %d with tag %d\n",
        buf[0], status.MPI_SOURCE, status.MPI_TAG );
        }
    }
    else {
    for (i=0; i<100; i++)
        buf[0]=rank+i;
       MPI_Send( buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
    }
    MPI_Finalize();
}
```

程序 16 **接收任意源和任意标识的消息**

7.7 编写安全的MPI程序

编写MPI程序，如果通信调用的顺序使用的不当，很容易造成死锁，比如对于程序 17 所示的例子总会死锁。

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF( rank .EQ. 1)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

程序 17 总会死锁的发送接收序列

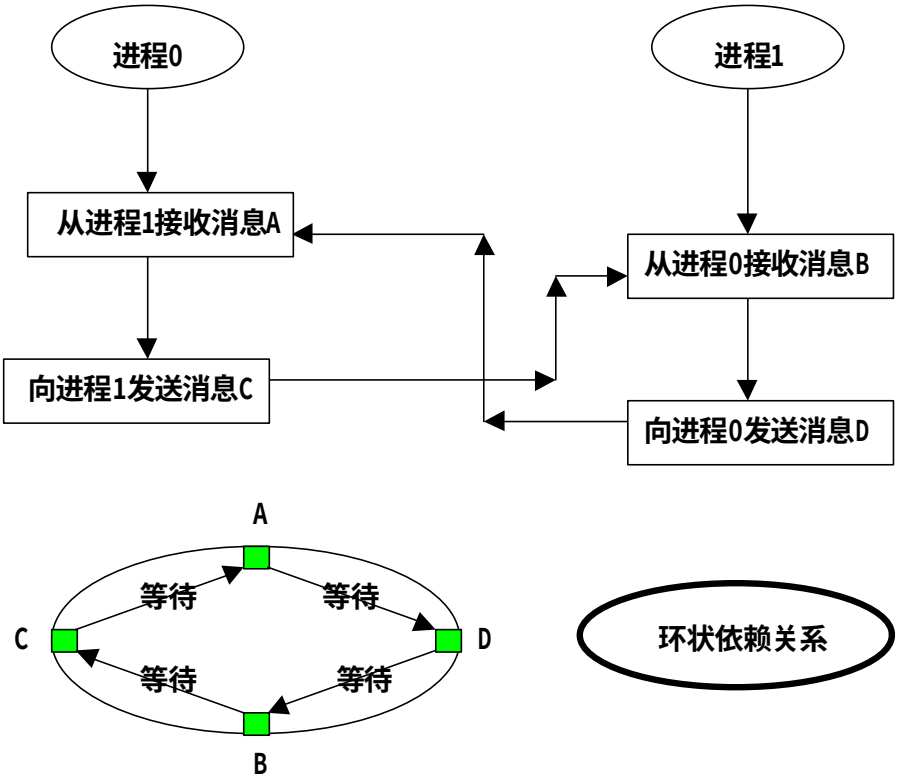


图 23 总会死锁的通信调用次序

进程0的第一条接收语句A能否完成取决于进程1的第二条发送语句D，即A依赖于D，从执行次序上可以明显地看出，进程0向进程1发送消息的语句C的执行又依赖于它前面的接收语句A的完成，即C依赖于A；同时，进程1的第一条接收语句B能否完成取决于进程0的第二条发送语句C的执行，即B依赖于C，从执行次序上可以明显地看出，向进程0发送消息的语句D的

执行又依赖于B的完成，故有A依赖于D，而D又依赖于B，B依赖于C，C依赖于A，形成了一个环，进程0和进程1相互等待，彼此都无法执行下去，必然导致死锁。

若两个进程需要相互交换数据，在两个进程中首先都进行接收调用显然是不合适的，那么，同时先进行发送调用（图 24）的结果又是怎样的呢?

```
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 （rank .EQ.1）
   CALL MPI_SEND(sendbuf, count, MPI_REAK, 0, tag, comm, status, ierr)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

程序 18 不安全的发送接收序列

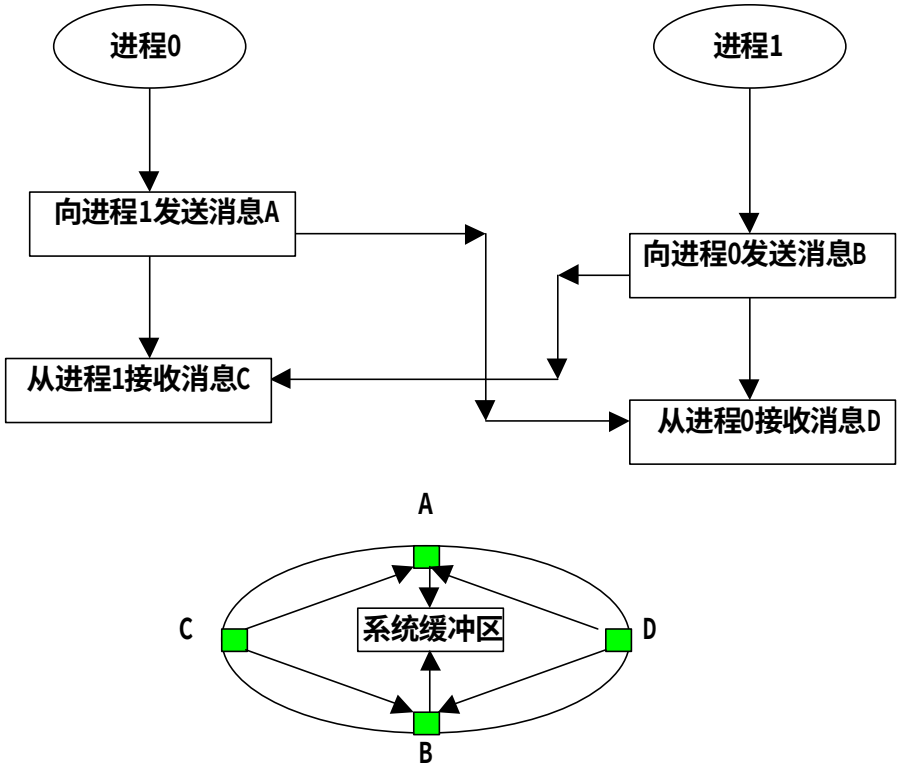


图 24 不安全的通信调用次序

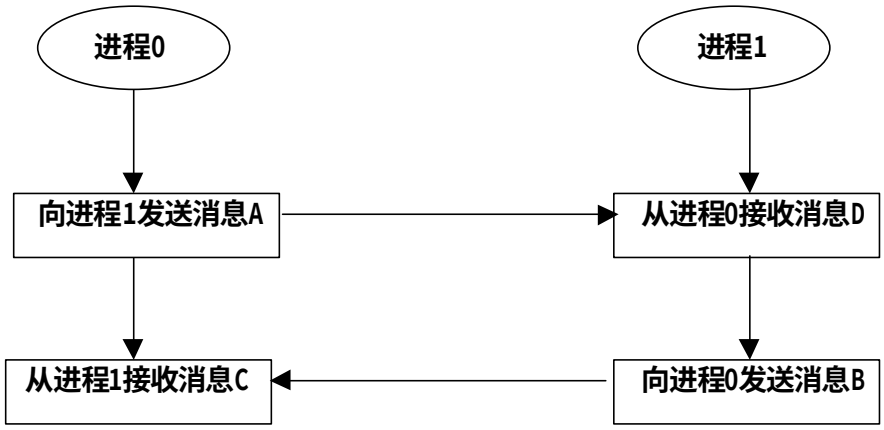
由于进程0或进程1的发送需要系统提供缓冲区（在MPI的四种通信模式中有详细的解释），如果系统缓冲区不足，则进程0或进程1的发送将无法完成，相应的，进程1和进程0的接收也无法正确完成。显然对于需要相互交换数据的进程，直接将两个发送语句写在前面也是不安全的。

下面介绍一种可以保证消息安全传递的通信调用次序（图 25），即当两个进程需要相互交换数据时，一定要将它们的发送和接收操作按照次序进行匹配，即一个进程的发送操作在

前，接收操作在后；而另一个进程的接收操作在前，发送操作在后，前后两个发送和接收操作要相互匹配。

```
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 （rank .EQ. 1）
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

程序 19 安全的发送接收序列



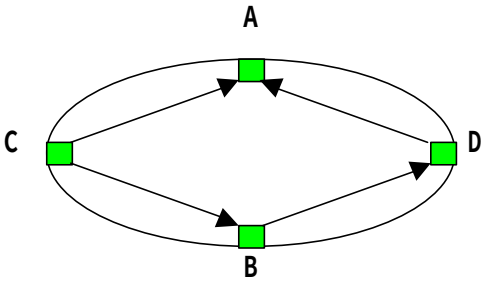


图 25 安全的通信调用次序

C的完成只需要A完成，而A的完成只要有对应的D存在，则不需要系统提供缓冲区也可以进行，这里恰恰满足这样的条件，因此A总能够完成，因此D也一定能完成。当A和D完成后，B的完成只需要相应的C，不需要缓冲区也能完成，因此B和C也一定能完成，所以说这样的通信形式是安全的。显然A和C，D和B同时互换，从原理上说和这种情况是一样的，因此也是安全的。

7.8 小结

本章的内容还没有涉及到并行程序具体的算法设计，还主要是关于MPI最基本的功能方面的内容，即最基本的MPI使用方法和注意的问题。在实际的使用中遇到的问题会复杂地多，需要把MPI多方面不同的功能综合在一起，这是后面章节将要介绍的内容。

第8章 MPI并行程序的两种基本模式

本章介绍MPI的两种最基本的并行程序设计模式，即对等模式和主从模式。可以说绝大部分MPI的程序都是这两种模式之一或二者的组合，掌握了这两种模式，就掌握了MPI并行程序设计的主线。

对于对等模式的MPI程序，本章是通过一个典型的例子—Jacobi迭代来逐步讲解的，并且将每一种具体的实现都和特定的MPI增强功能结合起来，达到既介绍并行程序设计方法，又讲解MPI特定功能调用的目的。

对于主从模式的MPI程序，是通过几个简单的例子来讲解主从进程功能的划分和主从进程之间的交互作用。

MPI程序一般是SPMD程序，当然也可以用MPI来编写MPMD程序，但是，所有的MPMD程序，都可以用SPMD程序来表达，二者的表达能力是相同的。

对于MPI的SPMD程序，实现对等模式的问题是比较容易理解和接受的，因为各个部分地位相同，功能和代码基本一致，只不过是处理的数据或对象不同，也容易用同样的程序来实现；但是对于主从模式的问题，用SPMD程序来实现是否合适你呢？不管是从理论的角度，还是从下面具体例子的实践，都说明了主从模式的问题是完全可以用SPMD程序来高效解决的，即SPMD程序有很强的表达能力，SPMD只是形式上的表现，其内容可以是十分丰富的，本书中介绍的所有例子都是SPMD形式的程序。

8.1 对等模式的MPI程序设计

8.1.1 问题描述—Jacobi迭代

Jacobi迭代是一种比较常见的迭代方法，其核心部分可以用程序 20来表示，简单地说，Jacobi迭代得到的新值是原来旧值点相邻数值点的平均。

Jacobi迭代的局部性很好，可以取得很高的并行性，是并行计算中常见的一个例子。将参加迭代的数据按块分割后，各块之间除了相邻的元素需要通信外，在各块的内部可以完全独立地并行计算，随着计算规模的扩大，通信的开销相对于计算来说比例会降低，这将更有利于提高并行效果。

```
…
REAL A(N+1,N+1), B(N+1,N+1)
…
DO K=1,STEP
      DO J=1,N
            DO I=1,N
                  B(I,J)=0.25*(A(I-1,J)+A(I+1,J)+A(I,J+1)+A(I,J-1))
            END DO
      END DO
      DO J=1,N
            DO I=1,N
                  A(I,J)=B(I,J)
            END DO
      END DO
```

程序 20　串行表示的Jacobi迭代

8.1.2 用MPI程序实现Jacobi迭代

为了并行求解，这里将参加迭代的数据按列进行分割，并假设一共有4个进程同时并行计算，数据的分割结果如图 26所示。

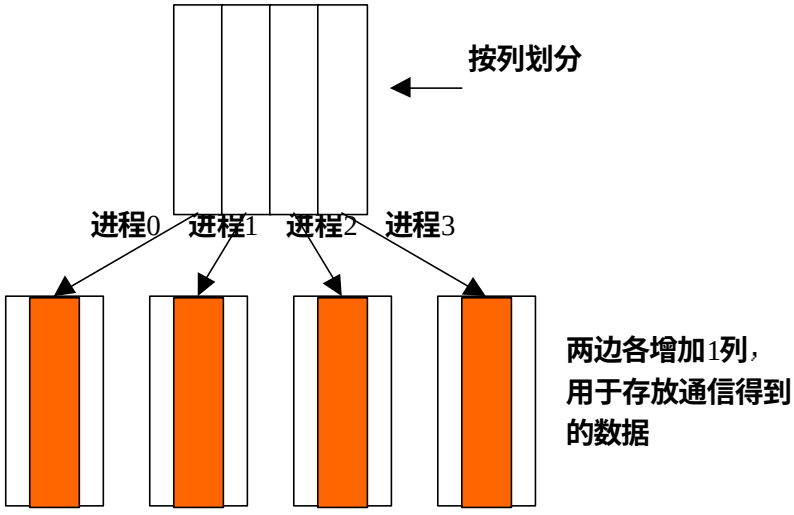


图 26　Jacobi迭代的数据划分及其与相应进程的对应

假设需要迭代的数据是M×M**的二维数组**A(M,M)，**令**M=4*N，**按图示进行数据划分，则分布在四个不同进程上的数据分别是：进程**0，A(M,1:N)，**进程**1，A(M,N+1:2*N),**进程**3，A(M,2*N+1:3*N)，**进程**3，A(M,3*N+1:M)。

由于在迭代过程中，边界点新值的计算需要相邻边界其它块的数据，因此在每一个数据块的两侧又各增加1**列的数据空间，用于存放从相邻数据块通信得到的数据。这样原来每个**

数据块的大小从M*N扩大到M*（N+2），进程0和进程1的数据块只需扩大一块即可满足通信的要求，但这里为了编程的方便和形式的一致，在两边都增加了数据块。

计算和通信过程是这样的，首先对数组赋初值，边界赋为8，内部赋为0，注意对不同的进程，赋值方式是不同的（两个内部块相同，但内部块和两个外部块两两互不相同）。然后便开始进行Jacobi迭代，在迭代之前，每个进程都需要从相邻的进程得到数据块，同时每一个进程也都需要向相邻的进程提供数据块（图 27，注意FORTRAN数组在内存中是按列优先排列的）。由于每一个新迭代点的值是由相邻点的旧值得到，所以这里引入一个中间数组，用来记录临时得到的新值，一次迭代完成后，再统一进行更新操作。程序 21是这一例子的完整程序。

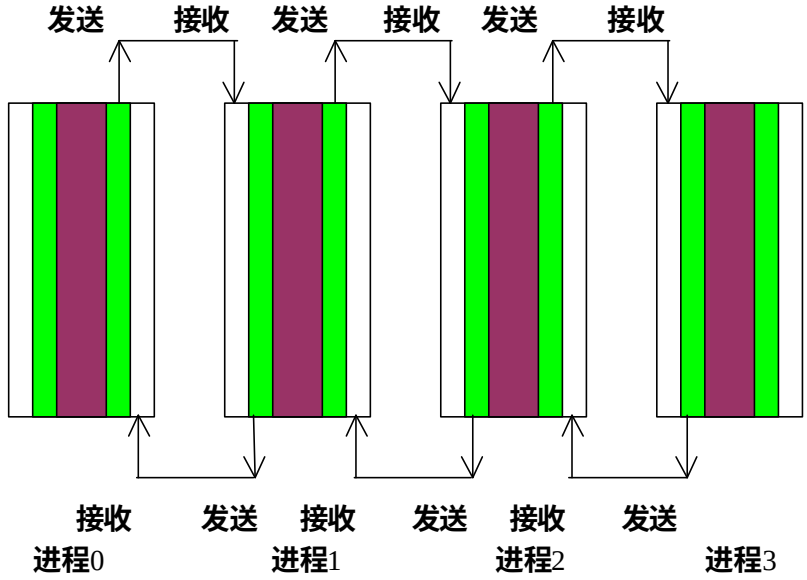


图 27　Jacobi迭代的数据通信图示

```
      program  main
      implicit  none
      include  'mpif.h'
      integer  totalsize,mysize,steps
      parameter    (totalsize=16)
C     定义全局数组的规模
      parameter    (mysize=totalsize/4,steps=10)

     integer n, myid, numprocs, i, j,rc
      real  a(totalsize,mysize+2),b(totalsize,mysize+2)
C     定义局部数组
      integer  begin_col,end_col,ierr
      integer  status(MPI_STATUS_SIZE)

     call  MPI_INIT( ierr )
      call  MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
     call  MPI_COMM_SIZE(  MPI_COMM_WORLD,  numprocs,  ierr )
```

```
      print *, "Process ", myid, " of ", numprocs, " is alive"

C     数组初始化
      do  j=1,mysize+2
          do  i=1,totalsize
              a(i,j)=0.0
          end  do
      end  do
       if (myid .eq. 0) then
          do  i=1,totalsize
              a(i,2)=8.0
          end  do
      end  if
       if (myid .eq. 3) then
          do  i=1,totalsize
              a(i,mysize+1)=8.0
          end  do
      end  if
      do  i=1,mysize+2
           a(1,i)=8.0
           a(totalsize,i)=8.0
      end  do

C     Jacobi 迭代部分
      do  n=1,steps

C     从右侧的邻居得到数据
       if (myid .lt. 3)    then
          call  MPI_RECV(a(1,mysize+2),totalsize,MPI_REAL,myid+1,10,
     *              MPI_COMM_WORLD,status,ierr)
       end  if
C     向左侧的邻居发送数据
       if  ((myid  .gt.  0)  )  then
          call  MPI_SEND(a(1,2),totalsize,MPI_REAL,myid-1,10,
     *          MPI_COMM_WORLD,ierr)
       end  if
C     向右侧的邻居发送数据
       if (myid .lt. 3) then
          call  MPI_SEND(a(1,mysize+1),totalsize,MPI_REAL,myid+1,10,
     *              MPI_COMM_WORLD,ierr)
       end  if
C     从左侧的邻居接收数据
       if (myid .gt. 0) then
          call  MPI_RECV(a(1,1),totalsize,MPI_REAL,myid-1,10,
```

```
*              MPI_COMM_WORLD,status,ierr)
 end  if

 begin_col=2
 end_col=mysize+1
 if (myid .eq. 0) then
     begin_col=3
 endif
 if (myid .eq. 3) then
     end_col=mysize
 endif

do  j=begin_col,end_col
    do  i=2,totalsize-1
        b(i,j)=(a(i,j+1)+a(i,j-1)+a(i+1,j)+a(i-1,j))*0.25
    end  do
end  do
do  j=begin_col,end_col
    do  i=2,totalsize-1
        a(i,j)=b(i,j)
    end  do
end  do
end  do
do  i=2,totalsize-1
     print  *,  myid,(a(i,j),j=begin_col,end_col)
end  do

call  MPI_Finalize(rc)
 end
```

程序 21 用MPI_SEND和MPI_RECV实现的Jacobi迭代

8.1.3 用捆绑发送接收实现Jacobi迭代

在上面的Jacobi迭代这一例子中，每一个进程都要向相邻的进程发送数据，同时从相邻的进程接收数据，MPI提供了捆绑发送和接收操作，可以在一条MPI语句中同时实现向其它进程的数据发送和从其它进程接收数据操作，下面对它进行介绍。

捆绑发送和接收操作把发送一个消息到一个目的地和从另一个进程接收一个消息合并到一个调用中，源和目的可以是相同的。捆绑发送接收操作虽然在语义上等同于一个发送操作和一个接收操作的结合，但是它可以有效地避免由于单独书写发送或接收操作时，由于次序的错误而造成的死锁。这是因为该操作由通信系统来实现，系统会优化通信次序，从而有效地避免不合理的通信次序，最大限度避免死锁的产生。

MPI_SENDRECV(sendbuf,sendcount,sendtype,dest,sendtag,recvbuf,recvcount,
recvtype, source,recvtag,comm,status)
IN sendbuf **发送缓冲区起始地址(可选数据类型)**
IN sendcount **发送数据的个数(整型)**
IN sendtype **发送数据的数据类型(句柄)**
IN dest **目标进程标识(整型)**
IN sendtag **发送消息标识(整型)**
OUT recvbuf **接收缓冲区初始地址(可选数据类型)**
IN recvcount **最大接收数据个数(整型)**
IN recvtype **接收数据的数据类型(句柄)**
IN source **源进程标识(整型)**
IN recvtag **接收消息标识(整型)**
IN comm **通信域(句柄)**
OUT status **返回的状态**(status)

```
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)
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
```

MPI**调用接口** 13 MPI_SENDRECV

捆绑发送接收操作是不对称的，即一个由捆绑发送接收调用发出的消息可以被一个普通接收操作接收，一个捆绑发送接收调用可以接收一个普通发送操作发送的消息。

该操作执行一个阻塞的发送和接收，接收和发送使用同一个通信域，但是可能使用不同的标识。发送缓冲区和接收缓冲区必须分开，他们可以是不同的数据长度和不同的数据类型。

一个与 MPI_SENDRECV **类似的操作是** MPI_SENDRECV_REPLACE **，它与** MPI_SENDRECV**的不同就在于它只有一个缓冲区，该缓冲区同时作为发送缓冲区和接收缓冲区，这一调用的执行结果是发送前缓冲区中的数据被传递给指定的目的进程，该缓冲区被从指定进程接收到的相应类型的数据所取代。因此从功能上说，这两者没有什么区别，只是** MPI_SENDRECV_REPLACE**相对于** MPI_SENDRECV**节省了一个接收缓冲区（和发送缓冲区公用）**

MPI_SENDRECV_REPLACE(buf,count,datatype,dest,sendtag,source,recvtag,comm, status)

INOUT	buf	**发送和接收缓冲区初始地址(可选数据类型)**
IN	count	**发送和接收缓冲区中的数据的个数(整型)**
IN	datatype	**发送和接收缓冲区中数据的数据类型(句柄)**
IN	dest	**目标进程标识(整型)**
IN	sendtag	**发送消息标识(整型)**
IN	source	**源进程标识(整型)**
IN	recvtag	**接收消息标识(整型)**
IN	comm	**发送进程和接收进程所在的通信域(句柄)**
OUT	status	**状态目标**(status)

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)

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR) BUF(*)

INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI**调用接口** 14 MPI_SENDRECV_REPLACE

对于Jacobi**迭代，发送和接收操作是成对出现的，因此特别适合使用捆绑发送接收操作调用。对于中间块来说，每一块都要向两侧的相邻块发送数据，同时也要从两侧的相邻块接收数据，可以非常方便地用**MPI_SENDRECV**调用来实现。如图** 28**所示。**

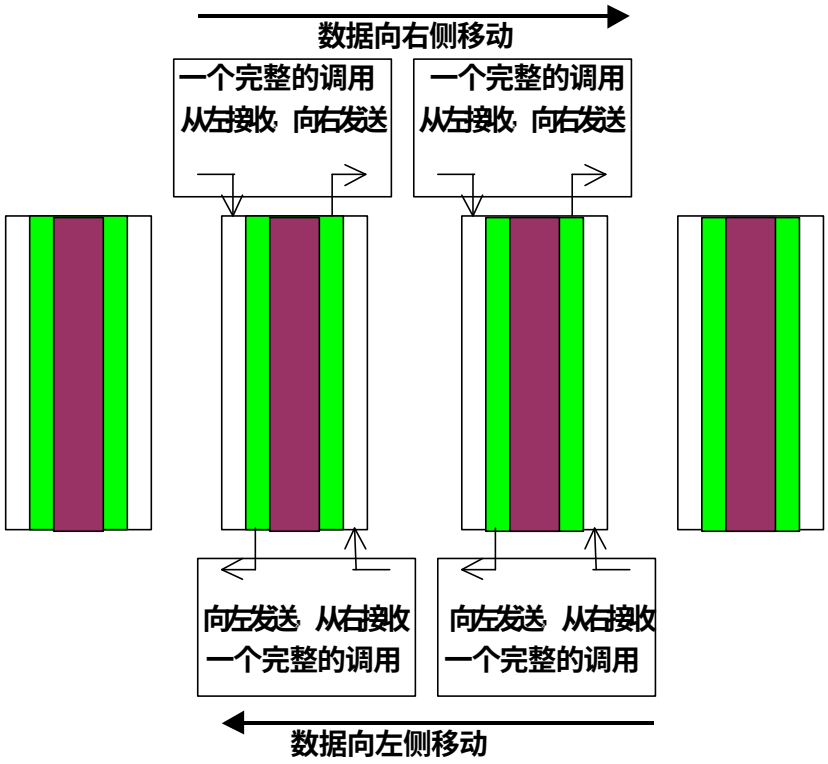


图 28 **用**MPI_SENDRECV**实现**Jacobi**迭代示意图**

但是对于左侧和右侧的边界块，却不容易将各自的发送和接收操作合并到一个调用之中，因此在程序 22**中，对边界块仍编写单独的发送和接收语句。在后面的章节，当引入进程拓扑和虚拟进程后，就可以可以把边界块和内部块统一看待，全部通信都用** MPI_SENDRECV**语句实现。**

```
 program  main
 implicit  none
 include  'mpif.h'
 integer  totalsize,mysize,steps
  parameter    (totalsize=16)
  parameter    (mysize=totalsize/4,steps=10)

integer n,  myid,  numprocs,  i,  j,rc
 real  a(totalsize,mysize+2),b(totalsize,mysize+2)
 integer  begin_col,end_col,ierr
 integer  status(MPI_STATUS_SIZE)

call  MPI_INIT(  ierr  )
 call  MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call  MPI_COMM_SIZE(  MPI_COMM_WORLD,  numprocs,  ierr  )
print *, "Process ", myid, " of ", numprocs, " is alive"
```

C **数组初始化**

```
 do  j=1,mysize+2
     do  i=1,totalsize
         a(i,j)=0.0
     end  do
 end  do
  if (myid .eq. 0) then
     do  i=1,totalsize
         a(i,2)=8.0
     end  do
 end  if
  if (myid .eq. 3) then
     do  i=1,totalsize
         a(i,mysize+1)=8.0
     end  do
 end  if
 do  i=1,mysize+2
      a(1,i)=8.0
      a(totalsize,i)=8.0
 end  do
```

C **开始迭代**

```
 do  n=1,steps

C      从左向右平移数据
       if (myid .eq. 0) then
          call  MPI_SEND(a(1,mysize+1),totalsize,MPI_REAL,myid+1,10,
     *             MPI_COMM_WORLD,ierr)
     else if (myid .eq. 3) then
          call  MPI_RECV(a(1,1),totalsize,MPI_REAL,myid-1,10,
     *             MPI_COMM_WORLD,status,ierr)
       else
          call  MPI_SENDRECV(a(1,mysize+1),totalsize,MPI_REAL,myid+1,10,
     *                       a(1,1),totalsize,MPI_REAL,myid-1,10,
     *                       MPI_COMM_WORLD,status,ierr)
       end  if

C      从右向左平移数据
       if (myid .eq. 0)    then
          call  MPI_RECV(a(1,mysize+2),totalsize,MPI_REAL,myid+1,10,
     *             MPI_COMM_WORLD,status,ierr)
     else if (myid .eq. 3) then
          call  MPI_SEND(a(1,2),totalsize,MPI_REAL,myid-1,10,
     *              MPI_COMM_WORLD,ierr)
       else
          call  MPI_SENDRECV(a(1,2),totalsize,MPI_REAL,myid-1,10,
     *                       a(1,mysize+2),totalsize,MPI_REAL,myid+1,10,
     *                       MPI_COMM_WORLD,status,ierr)
       end  if
       begin_col=2
       end_col=mysize+1
       if (myid .eq. 0) then
          begin_col=3
       endif
       if (myid .eq. 3) then
          end_col=mysize
       endif

       do  j=begin_col,end_col
          do  i=2,totalsize-1
             b(i,j)=(a(i,j+1)+a(i,j-1)+a(i+1,j)+a(i-1,j))*0.25
          end  do
       end  do
       do  j=begin_col,end_col
          do  i=2,totalsize-1
             a(i,j)=b(i,j)
```

```
      end do
    end do
    end do
    do i=2,totalsize-1
        print *, myid,(a(i,j),j=begin_col,end_col)
    end do

    call MPI_Finalize(rc)
    end
```

程序 22 用MPI_SENDRECV实现的Jacobi迭代

8.1.4 引入虚拟进程后Jacobi迭代的实现

虚拟进程（MPI_PROC_NULL）是不存在的假想进程，在MPI中的主要作用是充当真实进程通信的目或源，引入虚拟进程的目的是为了在某些情况下编写通信语句的方便。当一个真实进程向一个虚拟进程发送数据或从一个虚拟进程接收数据时，该真实进程会立即正确返回，如同执行了一个空操作。

在很多情况下为通信指定一个虚拟的源或目标是非常方便的，这不仅可以大大简化处理边界的代码，而且使程序显得简洁易懂。在捆绑发送接收操作中经常用到这种通信手段。一个真实进程向虚拟进程MPI_PRC_NULL发送消息时会立即成功返回；一个真实进程从虚拟进程MPI_PROC_NULL的接收消息时也会立即成功返回，并且对接收缓冲区没有任何改变。下面用虚拟进程实现的Jacobi迭代的通信部分显然比不用虚拟进程的程序要简单得多。

```
    program main
    implicit none
    include 'mpif.h'
    integer totalsize,mysize,steps
    parameter   (totalsize=16)
    parameter   (mysize=totalsize/4,steps=10)

   integer n, myid, numprocs, i, j,rc
    real a(totalsize,mysize+2),b(totalsize,mysize+2)
    integer begin_col,end_col,ierr
    integer left,right,tag1,tag2
    integer status(MPI_STATUS_SIZE)

   call MPI_INIT( ierr )
    call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
   call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
  print *, "Process ", myid, " of ", numprocs, " is alive"
```

```
C       数组初始化
      do  j=1,mysize+2
          do  i=1,totalsize
              a(i,j)=0.0
          end  do
      end  do
       if (myid .eq. 0) then
          do  i=1,totalsize
              a(i,2)=8.0
          end  do
      end  if
       if (myid .eq. 3) then
          do  i=1,totalsize
              a(i,mysize+1)=8.0
          end  do
      end  if
      do  i=1,mysize+2
           a(1,i)=8.0
           a(totalsize,i)=8.0
      end  do

       tag1=3
       tag2=4
C       设置当前进程左右两侧的进程标识
       if (myid .gt. 0) then
          left=myid-1
       else
          left=MPI_PROC_NULL
      end  if
       if (myid .lt. 3) then
          right=myid+1
       else
          right=MPI_PROC_NULL
      end  if
C      Jacobi  迭代
      do  n=1,steps
C       从左向右平移数据
          call  MPI_SENDRECV(a(1,mysize+1),totalsize,MPI_REAL,right,tag1,
     *                       a(1,1),totalsize,MPI_REAL,left,tag1,
     *                       MPI_COMM_WORLD,status,ierr)
C       从右向左平移数据
          call  MPI_SENDRECV(a(1,2),totalsize,MPI_REAL,left,tag2,
     *                       a(1,mysize+2),totalsize,MPI_REAL,right,tag2,
     *                       MPI_COMM_WORLD,status,ierr)
```

```
begin_col=2
end_col=mysize+1
if (myid .eq. 0) then
    begin_col=3
endif
if (myid .eq. 3) then
    end_col=mysize
endif
do  j=begin_col,end_col
    do  i=2,totalsize-1
        b(i,j)=(a(i,j+1)+a(i,j-1)+a(i+1,j)+a(i-1,j))*0.25
    end  do
end  do
do  j=begin_col,end_col
    do  i=2,totalsize-1
        a(i,j)=b(i,j)
    end  do
end  do
end  do
do  i=2,totalsize-1
     print  *,  myid,(a(i,j),j=begin_col,end_col)
end  do
call  MPI_Finalize(rc)
end
```

程序 23 用虚拟进程实现的Jacobi迭代

8.2 主从模式的MPI程序设计

8.2.1 矩阵向量乘

下面的例子实现C=A×B。**具体的实现方法是（图** 29）：**主进程将向量**B**广播给所有的从进程，然后将矩阵**A**的各行依次发送给从进程，从进程计算一行和**B**相乘的结果，然后将结果发送给主进程。主进程循环向各个从进程发送一行的数据，直到将**A**各行的数据发送完毕，一旦主进程将**A**的各行发送完毕，则每收到一个结果，就向相应的从进程发送结束标志，从进程接收到结束标志后退出执行。主进程收集完所有的结果后也结束。**

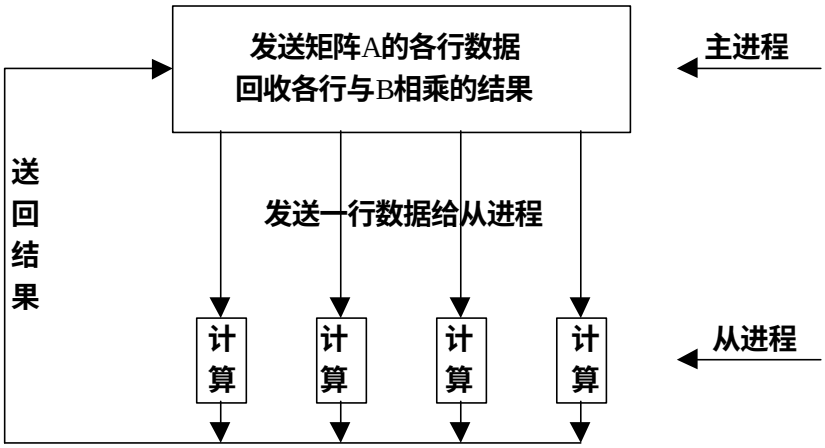


图 29 矩阵向量乘

```
      program    main
      include "mpif.h"
      integer MAX_ROWS,MAX_COLS, rows, cols
      parameter (MAX_ROWS=1000, MAX_COLS=1000)
      double precision a(MAX_ROWS, MAX_COLS),b(MAX_COLS),c(MAX_COLS)
      double precision buffer (MAX_COLS), ans

      integer    myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
      integer    i,j,numsent, numrcvd, sender
      integer    anstype,  row

      call    MPI_INIT(ierr)
      call    MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      call    MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
      master=0
      rows=100
      cols=100

      if (myid .eq. master) then
C     主进程对矩阵A和B赋初值
      do i=1,cols
        b(i)=1
        do  j=1,rows
            a(i,j)=i
        end  do
      end do
      numsent=0
      numrcvd=0
```

```
C       将矩阵B发送给所有其它的从进程，通过下面的广播语句实现
        call MPI_BCAST(b,cols,MPI_DOUBLE_PRECISION,master,
     $  MPI_COMM_WORLD, ierr)
C       依次将矩阵A的各行发送给其它的numprocs-1个从进程
        do i=1,min(numprocs-1,rows)
          do j=1,cols
C           将一行的数据取出来依次放到缓冲区中
            buffer(j)=a(i,j)
          end do
C         将准备好的一行数据发送出去
          call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION,i,
     $    i,MPI_COMM_WORLD, ierr)
          numsent=numsent+1
        end do
        do i=1,row
C         对所有的行，依次接收从进程对一行数据的计算结果
          call MPI_RECV(ans, 1,MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
     $    MPI_ANY_TAG,MPI_COMM_WORLD, status, ierr)
          sender=status(MPI_SOURCE)
          anstype=status(MPI_TAG)
C         将该行数据赋给结果数组C的相应单元
          c(anstype)=ans
          if (numsent .lt. rows) then
C           如果还有其它的行没有被计算，则继续发送
            do j=1,cols
C             准备好新一行的数据
              buffer(j)=a(numsent+1,j)
            end do
C           将该行数据发送出去
            call MPI_SEND(buffer,cols, MPI_DOUBLE_PRECISION, sender,
     $      numsent+1,MPI_COMM_WORLD, ierr)
            numsent=numsent+1
          else
C           若所有行都已发送出去，则每接收一个消息则向相应的从进程
C           发送一个标识为0的空消息，终止该从进程的执行
            call    MPI_SEND(1.0,0,MPI_DOUBLE_PRECISION,sender,
     $                  0, MPI_COMM_WORLD, ierr)
          end if
        else
C       下面为从进程的执行步骤，首先是接收数组B
        call MPI_BCAST(b,cols,MPI_DOUBLE_PRECISION,master,
     $                       MPI_COMM_WORLD, ierr)
```

```
C       接收主进程发送过来的矩阵A一行的数据
90      call MPI_RECV(buffer,cols, MPI_DOUBLE_PRECISION, master,
     $                    MPI_ANY_TAG, MPI_COMM_WORLD, status,ierr)
C       若接收到标识为0的消息，则退出执行
        if (status(MPI_TAG) .ne. 0) then
          row=status(MPI_TAG)
          ans=0.0
          do  i=1,cols
            ans=ans+buffer(i)*b(i)
          end  do
C         计算一行的结果，并将结果发送给主进程
          call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, row,
     $          MPI_COMM_WORLD, ierr)
          goto 90
        end if
        endif
        call MPI_FINALIZE(ierr)
        end
```

程序 24 矩阵向量乘

8.2.2 主进程打印各从进程的消息

下面主从模式的例子实现如下功能（图 30）：主进程（进程0）接收从进程（其它所有进程）的消息，根据消息的不同，分两种方式将消息打印，1是按从进程结点编号的大小依次打印；2是以任意的顺序打印。而从进程则实现向主进程发送消息的任务，消息也分两种：即以任意顺序打印的消息和按进程编号的顺序打印的消息。

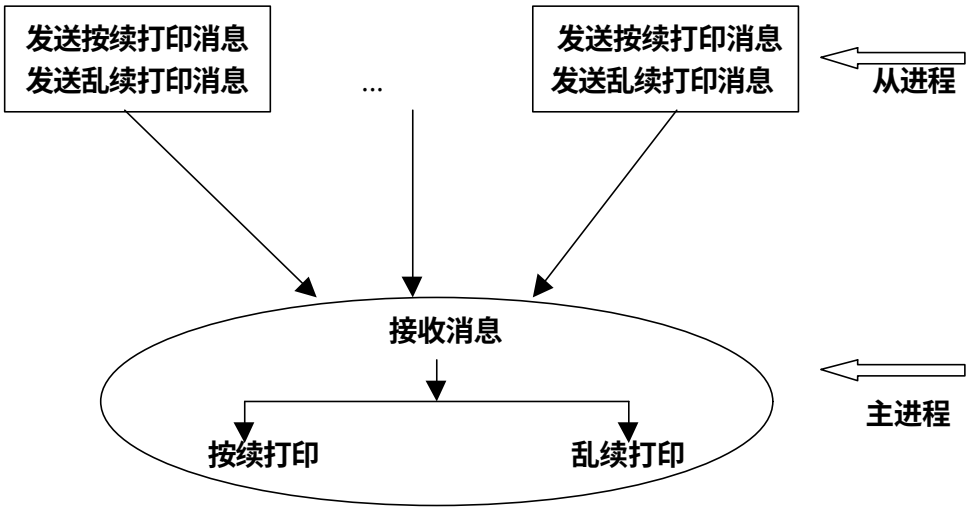


图 30 主进程的按续与乱续打印

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{
    int  rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if (rank == 0)
        master_io();    /*进程0作为主进程*/
     else
        slave_io();    /*其它进程作为从进程*/
    MPI_Finalize( );
}

#define MSG_EXIT 1
#define MSG_PRINT_ORDERED 2    /*定义按续打印标识*/
#define MSG_PRINT_UNORDERED 3 /*定义乱续打印标识*/

/*  下面的函数为主进程执行的部分  */
int master_io( void )
{
    int             i,j,  size,  nslave, firstmsg;
    char            buf[256],  buf2[256];
    MPI_Status  status;
     MPI_Comm_size( MPI_COMM_WORLD, &size );/*得到总的进程数*/
    nslave = size - 1;/*得到从进程的进程数*/
    while (nslave > 0) {/*  只要还有从进程，则执行下面的接收与打印*/
        MPI_Recv( buf, 256, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
       MPI_COMM_WORLD, &status );/*从任意从进程接收任意标识的消息*/
  switch (status.MPI_TAG) {
  case MSG_EXIT: nslave--; break;/*若该从进程要求退出，则将总的从进程个数减1*/
  case MSG_PRINT_UNORDERED:/*若该从进程要求乱续打印，则直接将该消息打印*/
      fputs( buf, stdout );
       break;
  case  MSG_PRINT_ORDERED:/*从进程要求按续打印，这种打印方式处理起来复杂一些，首先需要对收到的消息进行排序，若有些消息还没有收到，则需要调用接收语句接收相应的有序消息*/
       firstmsg  =  status.MPI_SOURCE;
      for (i=1; i<size; i++) {/*标识号从小到大开始打印*/
       if (i == firstmsg)
           fputs( buf, stdout );/*若接收到的消息恰巧是需要打印的消息，则直接打印*/
```

```
        else {/* 否则，先接收需要打印的消息，然后再打印*/
            MPI_Recv( buf2, 256, MPI_CHAR, i, MSG_PRINT_ORDERED,
                MPI_COMM_WORLD, &status );/*注意这一接收语句指定了源进程和消息标识，而不是象一开始的那样用任意源和任意标识*/
            fputs( buf2, stdout );
        }
        }
        break;
    }
        }
    }

    /*下面的函数为从进程执行的部分*/
    int slave_io( void )
    {
        char buf[256];
        int  rank;
        MPI_Comm_rank( MPI_COMM_WORLD, &rank );/*得到自己的标识*/
        sprintf( buf, "Hello from slave %d，ordered print\n", rank );
        MPI_Send( buf, strlen(buf) + 1, MPI_CHAR, 0, MSG_PRINT_ORDERED,
            MPI_COMM_WORLD );/*先向主进程发送一个有序打印消息*/

        sprintf( buf, "Goodbye from slave %d, ordered print\n", rank );
        MPI_Send( buf, strlen(buf) + 1, MPI_CHAR, 0, MSG_PRINT_ORDERED,
            MPI_COMM_WORLD );/*再向主进程发送一个有序打印消息*/
        sprintf( buf, "I'm exiting (%d),unordered print\n", rank );
        MPI_Send( buf, strlen(buf) + 1, MPI_CHAR, 0, MSG_PRINT_UNORDERED,
            MPI_COMM_WORLD );/*最后，向主进程发送一个乱续打印消息*/
        MPI_Send( buf, 0, MPI_CHAR, 0, MSG_EXIT, MPI_COMM_WORLD );/*向主进程发送退出执行的消息*/
    }
```

程序 25 主进程按续与乱续打印从进程的消息

图 31是启动10个进程执行该程序的结果（1**个主进程**9**个从进程**）。

```
Hello from slave 1,ordered print
Hello from slave 2,ordered print
Hello from slave 3,ordered print
Hello from slave 4,ordered print
Hello from slave 5,ordered print
Hello from slave 6,ordered print
Hello from slave 7,ordered print
Hello from slave 8,ordered print
Hello from slave 9,ordered print
```

```
Goodbye from slave 1,ordered print
Goodbye from slave 2,ordered print
Goodbye from slave 3,ordered print
Goodbye from slave 4,ordered print
Goodbye from slave 5,ordered print
Goodbye from slave 6,ordered print
Goodbye from slave 7,ordered print
Goodbye from slave 8,ordered print
Goodbye from slave 9,ordered print
I'm exiting (1),unordered print
I'm exiting (3),unordered print
I'm exiting (4),unordered print
I'm exiting (7),unordered print
I'm exiting (8),unordered print
I'm exiting (9),unordered print
I'm exiting (2),unordered print
I'm exiting (5),unordered print
I'm exiting (6),unordered print
```

图 31 按续与乱续打印的结果

8.3 小结

本章的例子虽然不复杂，但是它给出了MPI程序的两种最基本的模式，只要掌握了这两种基本的模式及其在MPI中的表达方式，则其它更复杂的问题都可以转换为其中任何一种模式或者这两种模式的任意排列组合或多层次的嵌套组合。因此本章对于MPI并行程序设计来说是十分重要的一章，以后对MPI的介绍大多是从功能方面的扩充，但最基本的功能划分模式和本章介绍的方法是没有太大区别的。

第9章 不同通信模式MPI并行程序的设计

MPI共有四种通信模式，首先是标准通信模式（standard mode），前面介绍的通信方式都是标准通信方式，此外还有其它另外三种通信方式，它们分别是缓存通信模式（buffered-mode），同步通信模式（synchronous-mode）和就绪通信模式（ready-mode）。这几种通信模式对应于不同的通信需求，MPI为用户提供功能相近的不同通信方式，为用户编写高效的并行程序提供了可能。

这几种通信模式主要是根据以下不同的情况来区分的：1 是否需要对发送的数据进行缓存？2 是否只有当接收调用执行后才可以执行发送操作？3 什么时候发送调用可以正确返回？ 4 发送调用正确返回是否意味着发送已完成？即发送缓冲区是否可以被覆盖？发送数据是否已到达接收缓冲区？针对这些不同的情况，MPI给出了不同的通信模式。

表格 6 MPI的通信模式

通信模式	发送	接收
标准通信模式	MPI_SEND	MPI_RECV
缓存通信模式	MPI_BSEND	
同步通信模式	MPI_SSEND	
就绪通信模式	MPI_RSEND	

从表中可以看出，对于非标准的通信模式，只有发送操作，而没有相应的接收操作。MPI对不同的通信模式，在调用的命名上就区别开来，标准通信模式不加特殊的字母，而对其它三个通信模式提供三个附加的发送函数，用一个字母前缀表示通信模式：B用于缓存通信模式，S 用于同步通信模式，R 用于就绪通信模式。下面分别对它们进行介绍。

9.1 标准通信模式

在MPI采用标准通信模式（图 32）时，是否对发送的数据进行缓存是由MPI自身决定的，而不是由并行程序员来控制。

如果MPI决定缓存将要发出的数据，发送操作不管接收操作是否执行，都可以进行，而且发送操作可以正确返回而不要求接收操作收到发送的数据。

由于缓存数据是需要付出代价的，它会延长数据通信的时间，而且缓冲区也并不是总可以得到的，这样MPI也可以不缓存将要发出的数据，这样只有当相应的接收调用被执行后，并且发送数据完全到达接收缓冲区后，发送操作才算完成。对于非阻塞通信，发送操作虽然没有完成，但是发送调用可以正确返回，程序可以接下来执行其它的操作。

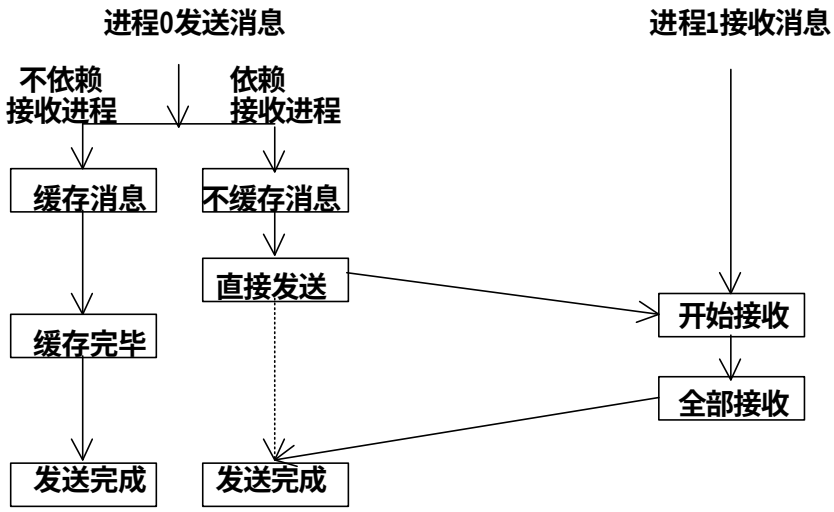


图 32 标准通信模式

前面介绍的所有例子都是用标准模式实现的。

9.2 缓存通信模式

当用户对标准通信模式不满意，希望直接对通信缓冲区进行控制时，可采用缓存通信模式（图 33）。在这种模式下，由用户直接对通信缓冲区进行申请、使用和释放，因此，缓存模式下对通信缓冲区的合理与正确使用是由程序设计人员自己保证的。

MPI_BSEND(buf, count, datatype, dest, tag, comm)
IN buf 发送缓冲区的起始地址(可选数据类型)
IN count 发送数据的个数(整型)
IN datatype 发送数据的数据类型(句柄)
IN dest 目标进程标识号(整型)
IN tag 消息标志(整型)
IN comm 通信域(句柄)

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm)
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type>BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

MPI调用接口 15 MPI_BSEND

MPI_BSEND的各个参数的含义和MPI_SEND的完全相同，不同之处仅表现在通信时是使用标准的系统提供的缓冲区还是用户自己提供的缓冲区。缓存通信模式不管接收操作是否启动，发送操作都可以执行，但是在发送消息之前必须有缓冲区可用，这由用户保证，否则该发送将失败返回。对于非阻塞发送，正确退出并不意味者缓冲区可以被其它的操作任意使用，但阻塞发送返回后其缓冲区是可以重用的。

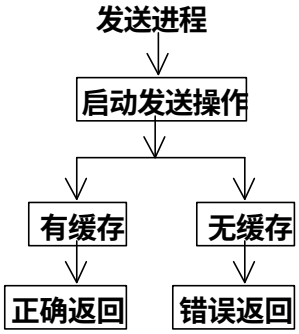


图 33 缓存通信模式

采用缓存通信模式是，消息发送能否进行及能否正确返回不依赖于接收进程，完全依赖于是否有足够的通信缓冲区可用，当缓存发送返回后，并不意味者该缓冲区可以自由使用，只有当缓冲区中的消息发送出去后，才可以释放该缓冲区。

用户可以首先申请缓冲区，然后把它提交给MPI**作为发送缓存，用于支持发送进程的缓存通信模式。这样，当缓存通信方式发生时，**MPI**就可以使用这些缓冲区对消息进行缓存。当不使用这些缓冲区时，可以将这些缓冲区释放。**

```
MPI_BUFFER_ATTACH( buffer, size)
   IN buffer  初始缓存地址(可选数据类型)
   IN size  按字节计数的缓存跨度(整型)
int MPI_Buffer_attach( void* buffer, int size)
MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR)
   <type>BUFFERR(*)
   INTEGER SIZE, IERROR
```

MPI调用接口 16 MPI_BUFFER_ATTACH

MPI_BUFFER_ATTACH**将大小为**size**的缓冲区递交给**MPI**，这样该缓冲区就可以作为缓存发送时的缓存来使用。**

```
MPI_BUFFER_DETACH( buffer, size)
OUT buffer  缓冲区初始地址(可选数据类型)
OUT size  以字节为单位的缓冲区大小(整型)
int MPI_Buffer_detach( void** buffer, int* size)
MPI_BUFFER_DETACH( BUFFER, SIZE, IERROR)
<type>BUFFER(*) INTEGER SIZE, IERROR
```

MPI调用接口 17 MPI_BUFFER_DETACH

MPI_BUFFER_DETACH**将提交的大小为**size**的缓冲区**buffer**收回。该调用是阻塞调用，它一直等到使用该缓存的消息发送完成后才返回。这一调用返回后用户可以重新使用该缓冲区或者将这一缓冲区释放。**

下面的例子首先用缓存模式发送包含5**个双精度类型的数据，再发送一个双精度类型的**

数据。而接收操作仍然使用标准的接收。

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define SIZE 6
/*总的数据大小*/
static int src   = 0;
static int dest = 1;

void Generate_Data ( double *, int );
/*  产生发送的数据*/
void Normal_Test_Recv ( double *, int );
/*标准接收*/
void Buffered_Test_Send ( double *, int );
/*缓存发送*/
void Generate_Data(buffer, buff_size)
double *buffer;
int buff_size;
{
      int  i;
     for (i = 0; i < buff_size; i++)
       buffer[i] = (double)i+1;
}

void Normal_Test_Recv(buffer, buff_size)
double *buffer;
int buff_size;
{
     int  i,  j;
     MPI_Status  Stat;
     double         *b;

     b  =  buffer;
      /*  先接收buff_size - 1个连续的双精度数  */
      MPI_Recv(b, (buff_size - 1), MPI_DOUBLE, src,
       2000, MPI_COMM_WORLD, &Stat);
           fprintf(stderr,"standard  receive a message of %d data\n",buff_size-1);
           for  (j=0;j<buff_size-1;j++)
              fprintf(stderr,"  buf[%d]=%f\n",j,b[j]);
      b += buff_size - 1;
      /*  再接收一个双精度数  */
      MPI_Recv(b,  1, MPI_DOUBLE, src, 2000, MPI_COMM_WORLD, &Stat);
           fprintf(stderr,"standard  receive a message of one data\n");
            fprintf(stderr,"buf[0]=%f\n",*b);
```

```
}

void Buffered_Test_Send(buffer, buff_size)
double *buffer;
int buff_size;
{
      int i, j;
      void  *bbuffer;
       int    size;
            fprintf(stderr,"buffered  send message of %d data\n",buff_size-1);
            for  (j=0;j<buff_size-1;j++)
               fprintf(stderr,"buf[%d]=%f\n",j,buffer[j]);
                  /*先用缓存方式方式发送buff_size - 1个双精度的数*/
                  MPI_Bsend(buffer, (buff_size - 1), MPI_DOUBLE, dest, 2000,
                              MPI_COMM_WORLD);
                  buffer += buff_size - 1;
                  fprintf(stderr,"buffered  send message of one data\n");
                     fprintf(stderr,"buf[0]=%f\n",*buffer);
                  /*再用缓存方式方式发送1个双精度数*/
                 MPI_Bsend(buffer,  1,  MPI_DOUBLE,
                          dest,  2000,  MPI_COMM_WORLD);
           /*  强制收回发送缓冲，这样也保证了该操作返回后消息已经送出*/
            MPI_Buffer_detach( &bbuffer, &size );
            /*再重新递交发送缓存*/
           MPI_Buffer_attach(  bbuffer,  size );
}

int main(int argc, char **argv)
{
    int rank; /* My Rank (0 or 1) */
      double  buffer[SIZE], *tmpbuffer, *tmpbuf;
      int  tsize, bsize;
     char  *Current_Test  =  NULL;

      MPI_Init(&argc,  &argv);
      MPI_Comm_rank(MPI_COMM_WORLD,  &rank);

     if  (rank  ==  src)
       /*  若当前进程为发送进程*/
            Generate_Data(buffer, SIZE);/*产生发送数据*/
            MPI_Pack_size( SIZE, MPI_DOUBLE, MPI_COMM_WORLD, &bsize );
            /*计算为发送SIZE个MPI_DOUBLE  类型的数据所需要的空间*/
            tmpbuffer = (double *) malloc( bsize + 2*MPI_BSEND_OVERHEAD );
            /*申请缓存发送所需要的空间*/
```

```
            if (!tmpbuffer) {
                fprintf( stderr, "Could not allocate bsend buffer of size %d\n",
                   bsize  );
                MPI_Abort( MPI_COMM_WORLD, 1 );
            }
           MPI_Buffer_attach(  tmpbuffer, bsize + MPI_BSEND_OVERHEAD );
            /*将申请到的空间递交给MPI，从而MPI可以利用该空间进行消息缓存*/
            Buffered_Test_Send(buffer, SIZE);/*执行缓存消息发送*/
            MPI_Buffer_detach( &tmpbuf, &tsize );/*发送完成后收回递交的缓冲区*/
        } else if (rank == dest) {
          /*  若当前进程为接收进程  */
            Normal_Test_Recv(buffer, SIZE);/*执行标准的接收操作*/

        } else {
            fprintf(stderr, "*** This program uses exactly 2 processes! ***\n");
            /*本程序只能使用两个进程*/
            MPI_Abort( MPI_COMM_WORLD, 1 );
        }
        MPI_Finalize();
    }
```

程序 26 使用缓存通信模式发送消息

9.3 同步通信模式

```
MPI_SSEND(buf, count, datatype, dest, tag, comm)
    IN      buf         发送缓冲区的初始地址(可选数据类型)
    IN      count       发送数据的个数(整型)
    IN      datatype    发送数据的数据类型(句柄)
    IN      dest        目标进程号(整型)
    IN      tag         消息标识(整型)
    IN      comm        通信域(句柄)
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

MPI调用接口 18 MPI_SSEND

同步通信模式（图 34）的开始不依赖于接收进程相应的接收操作是否已经启动，但是同步发送却必须等到相应的接收进程开始后才可以正确返回。因此，同步发送返回后，意味着发送缓冲区中的数据已经全部被系统缓冲区缓存，并且已经开始发送。这样当同步发送返

回后，发送缓冲区可以被释放或重新使用。

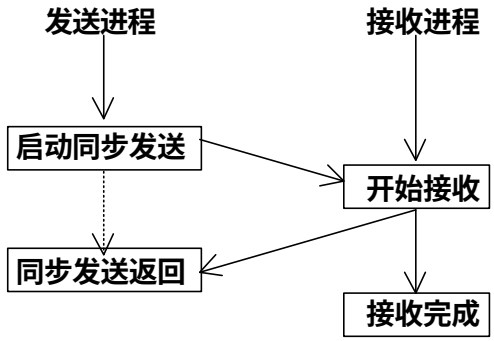


图 34 同步通信模式

下面的例子使用了同步消息发送，发送进程分别以同步方式发送1个和4个数据，消息标识tag分别为1和2。接收进程用标准接收操作接收它，并检查接收到数据的个数。

```
#include <stdio.h>
#include "mpi.h"
#define SIZE 10
/* Amount of time in seconds to wait for the receipt of the second Ssend
    message */
static int src  = 0;
static int dest = 1;
int main( int argc, char **argv)
{
    int rank; /* My Rank (0 or 1) */
     int act_size = 0;
     int flag, np, rval, i;
     int buffer[SIZE];
     MPI_Status status, status1, status2;
     int count1, count2;
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size( MPI_COMM_WORLD, &np );
     if (np != 2) {
          fprintf(stderr, "*** This program uses exactly 2 processes! ***\n");
           MPI_Abort( MPI_COMM_WORLD, 1 );
           }

            act_size = 5;/*最大消息长度*/
          if (rank == src) { /*当前进程为发送进程*/
               act_size = 1;
             MPI_Ssend( buffer, act_size, MPI_INT, dest, 1, MPI_COMM_WORLD );
               /*同步消息发送，发送一个整型数，tag标识为1*/
              fprintf(stderr,"MPI_Ssend %d data,tag=1\n", act_size);
```

```
            act_size = 4;
            MPI_Ssend( buffer, act_size, MPI_INT, dest, 2, MPI_COMM_WORLD );
              /*同步发送4个整型数，tag标识为2*/
            fprintf(stderr,"MPI_Ssend %d data,tag=2\n", act_size);
        } else if (rank == dest) {/*当前进程为接收进程*/
            MPI_Recv( buffer, act_size, MPI_INT, src, 1, MPI_COMM_WORLD,
                    &status1 );
             /*标准消息接收，这里指定的消息长度act_size是最大消息长度，tag为1*/
            MPI_Recv( buffer, act_size, MPI_INT, src, 2, MPI_COMM_WORLD,
                    &status2 );
             /*标准消息接收，这里指定的消息长度act_size是最大消息长度，tag为2*/
             MPI_Get_count( &status1, MPI_INT, &count1 );/*消息1包含的数据的个数*/
            fprintf(stderr,"receive %d data,tag=%d\n",count1,status1.MPI_TAG);
             MPI_Get_count( &status2, MPI_INT, &count2 );/*消息2包含的数据的个数*/
            fprintf(stderr,"receive %d data,tag=%d\n",count2,status2.MPI_TAG);
        }
    MPI_Finalize();
}
```

程序 27 同步模式的消息发送

9.4 就绪通信模式

```
MPI_RSEND(buf, count, datatype, dest, tag, comm)
    IN    buf       发送缓冲区的初始地址(可选数据类型)
    IN    count     将发送数据的个数(整型)
    IN    datatype  发送数据的数据类型(句柄)
    IN    dest      目标进程标识(整型)
    IN    tag       消息标识(整型)
    IN    comm      通信域(句柄)
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type>BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

MPI调用接口 19 MPI_RSEND

在就绪通信模式（图 35）中，只有当接收进程的接收操作已经启动时，才可以在发送进程启动发送操作，否则，当发送操作启动而相应的接收还没有启动时，发送操作将出错。对于非阻塞发送操作的正确返回，并不意味着发送已完成，但对于阻塞发送的正确返回，则发送缓冲区可以重复使用。

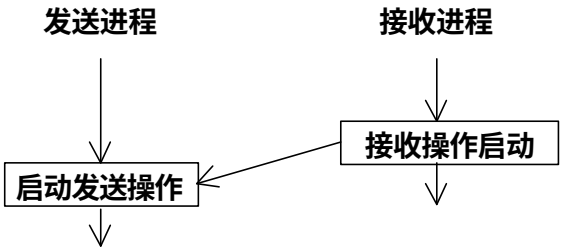


图 35 就绪通信模式

就绪通信模式的特殊之处就在于它要求接收操作先于发送操作而被启动，因此，在一个正确的程序中，一个就绪发送能被一个标准发送替代，它对程序的语义没有影响，而对程序的性能有影响。

下面给出一个使用就绪通信模式的例子（图 36）。这一例子的特点是它为了保证消息发送时消息接收已经执行（即确保①早于④）所采取的措施。

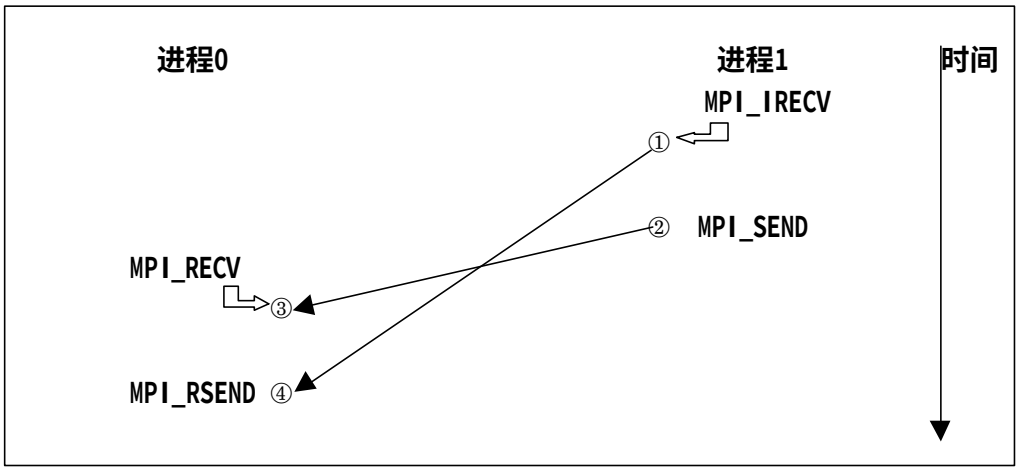


图 36 就绪发送例子各调用的时间关系图

为了叙述和表达的方便，作标记如下：

①：MPI_IRECV 调用返回的时刻

②：开始执行MPI_SEND的时刻

③：MPI_RECV完成的时刻

④：开始执行MPI_RSEND的时刻

为了保证MPI_RSEND在相应的接收操作执行后再调用，首先在它的前面放置了一个阻塞调用MPI_RECV,从时间上看， ③<④，（这里用“<”表示③在时间上先于④）。显然对于阻塞接收操作，若相应的阻塞发送操作MPI_SEND不执行，MPI_RECV就不可能结束，显然又有关系②<③，而MPI_SEND的执行必须等到MPI_IRECV返回后才可以开始，因此又有①<②。综上有①<④即就绪发送调用一定在相应的接收调用启动后才执行。

```
 program rsendtest
include  'mpif.h'
integer  ierr
call  MPI_Init(ierr)
call  test_rsend
```

C 该例程给出了一个简单的就绪通信的方法

```
      call MPI_Finalize(ierr)
      end

      subroutine test_rsend
      include 'mpif.h'
      integer TEST_SIZE
      parameter (TEST_SIZE=2000)
      integer ierr, prev, next, count, tag, index, i, outcount,
     $        requests(2), indices(2), rank, size,
     $        status(MPI_STATUS_SIZE), statuses(MPI_STATUS_SIZE,2)
      logical flag
      real send_buf( TEST_SIZE ), recv_buf ( TEST_SIZE )

      call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
      call MPI_Comm_size( MPI_COMM_WORLD, size, ierr )
      if (size .ne. 2) then
         print *, 'This test requires exactly 2 processes'
         call MPI_Abort( 1, MPI_COMM_WORLD, ierr )
      endif

      next = rank + 1
      if (next .ge. size) next = 0
      prev = rank - 1
      if (prev .lt. 0) prev = size - 1
C     设置进程标识间的关系
      if (rank .eq. 0) then
         print *, "      Rsend   Test   "
      end if
      tag = 1456
      count = TEST_SIZE / 3
      if (rank .eq. 0) then
         call MPI_Recv( MPI_BOTTOM, 0, MPI_INTEGER, next, tag,
     $                  MPI_COMM_WORLD, status, ierr )
C        进程0在执行就绪发送之前首先执行一个阻塞接收，这样就保证了
C        就绪发送操作一定在该阻塞接收完成后才能够进行
C        其实该阻塞接收操作没有接收任何数据，因为接收数据个数为0
C        MPI_BOTTOM是MPI预定义的一个内存地址
         print *,"Process ",rank," post Ready send"
         call MPI_Rsend(send_buf, count, MPI_REAL, next, tag,
     $                  MPI_COMM_WORLD, ierr)
C        执行就绪发送操作
      else
         print *, "process ",rank," post a receive call"
         call MPI_Irecv(recv_buf, TEST_SIZE, MPI_REAL,
```

```
      $              MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
      $               requests(1), ierr)
C          进程1先执行一个非阻塞的接收调用，该调用可以立即返回
        call MPI_Send( MPI_BOTTOM, 0, MPI_INTEGER, next, tag,
      $                        MPI_COMM_WORLD, ierr )
C          然后在执行一个发送空消息的阻塞调用，将这一阻塞发送放在
C          和就绪发送对应的MPI_Irecv之后，可以保证就绪发送一定在相应
C          的接收调用之后才调用
        call MPI_Wait( requests(1), status, ierr )
C          完成非阻塞接收调用。
        print *,"Process ", rank," Receive Rsend message from ",
      $                 status(MPI_SOURCE)
       end if
       end
```

程序 28 就绪通信模式的例子

9.5 小结

标准通信模式之外的其它通信模式，是MPI提供给程序员附加的并行程序通信手段，它是当程序员认为标准通信模式不能满足要求或者不能很好地满足给定的要求时所采取的措施，它要求程序员对程序和通信过程有更准确更深刻地理解，在此基础上，根据不同的需要，使用不同的通信模式，往往可以达到优化和提高效率的目的。

第10章 MPICH的安装与MPI程序的运行

本章对MPI的一个最成熟和最广泛使用的版本MPICH进行介绍，包括MPICH在两种典型的操作系统Linux和NT上的安装、MPI程序的编译和运行。读者可以按照本章介绍的方法，安装MPICH，并且在该实现下编写和运行各种MPI程序。

MPICH是MPI的一种具体实现，该实现可以免费从网上下载，MPICH的开发与MPI规范的制订是同步进行的，因此MPICH最能反映MPI的变化和发展。

MPICH的开发主要是由Argonne National Laboratory和Mississippi State University共同完成的，在这一过程中，IBM也做出了自己的贡献。但是MPI规范的标准化工作是由MPI论坛完成的。

10.1 Linux环境下的MPICH

10.1.1 安装

① MPICH软件包的下载

根据自己及其配置的不同，可以下载不同的软件包，名字分别是mpich.tar.gz 和mpich.tar.Z。mpich.tar.gz需要用gunzip来解压。

可以通过浏览器下载，网址是http://www.mcs.anl.org/mpi/mpich/

也可以通过匿名ftp下载ftp://ftp.mcs.anl.org/pub/mpi和ftp://ftp.mcs.anl.org/pub/mpisplit，其中ftp://ftp.mcs.anl.org/pub/mpisplit是将文件拆开，使用者可以分别下载小的片段，然后通过cat命令将它们拼接在一起。

② 将软件包解压

通过如下命令：

```
tar zxvf mpich.tar.gz
```

或

```
gunzip –c mpich.tar.gz | tar xovf –
```

或

```
zcat mpich.tar.Z | tar xovf –
```

或

```
uncompress mpich.tar.Z
tar xvf mpich.tar
```

③ 进入解开的mpich子目录

```
cd mpich
```

有些包带有版本信息，如1.1.1，1.1.2等

④ 创建Makefile和编译

```
./configure
```

也可以加prefix指出安装的位置：./configure –prefix=/usr/local/mpich-1.2.1

```
make
```

其中的configure命令完成MPI的自动配置，而make对MPI进行编译。

⑤测试安装是否正确
cd examples/basic
make cpi
../../bin/mpirun –np 4 cpi
或者直接在$(HOME)/mpich下运行
make testing
⑥将mpich安装到指定的目录
make install
其中安装位置由配置时prefix指定。

10.1.2 主要目录介绍

$（HOME）/mpich-1.2.1/MPI-2-C++
mpich对C++的支持部分
$（HOME）/mpich-1.2.1/bin
mpich的执行脚本
$（HOME）/mpich-1.2.1/doc
mpich的相关文档
$（HOME）/mpich-1.2.1/examples
mpich自带的例子程序
$（HOME）/mpich-1.2.1/f90modules
mpich对Fortran90的支持
$（HOME）/mpich-1.2.1/include
mpich的头文件
$（HOME）/mpich-1.2.1/lib
mpich的可联接库
$（HOME）/mpich-1.2.1/man
mpich的参考手册
$（HOME）/mpich-1.2.1/mpe
mpich的扩展部分
$（HOME）/mpich-1.2.1/mpid
mpich对不同设备的支持
$（HOME）/mpich-1.2.1/romio
mpich对并行I/O的支持部分
$（HOME）/mpich-1.2.1/share
通过upshot或jumpshot查看的例子
$（HOME）/mpich-1.2.1/src
mpich的可移植源程序
$（HOME）/mpich-1.2.1/util
mpich应用程序
$（HOME）/mpich-1.2.1/www
通过浏览器访问的mpich参考手册
下面介绍对于一个已经设计好的MPI程序，如何对它进行编译、运行、查看结果等。

10.1.3 编译命令

mpiCC/mpicc/mpif77/mpif90

mpiCC编译并联接用C++编写的MPI程序，而mpicc是编译并联接用C编写的MPI程序，mpif77和mpif90分别编译并联接用FORTRAN77和Fortran90编写的MPI程序，这些命令在联接时可以自动提供MPI需要的库，并提供特定的开关选项。注意mpiCC不能不能用于编译C程序。常用的编译选项是

-mpilog　产生MPE的log文件

-mpitrace　产生跟踪文件，这样在该MPI程序执行时会打印出其运行踪迹信息。但是它和-mpilog 在编译时不能同时存在，只能二者选一。

-mpianim　产生实时动画

-show　显示编译时产生的命令，但并不执行它

-help　给出帮助信息

-echo　显示出当前正在编译联接的命令信息

此外它们还可以使用一般的C++/C/FORTRAN77/Fortran90通用的选项，含义和原来的编译器相同。

10.1.4 执行步骤

MPI程序一般被称为SPMD（Single Program Multiple Data）程序，即相同的程序对不同的数据进行处理。当然用MPI也可以编写出MASTER/SLAVER类的具有明显主从关系的程序。

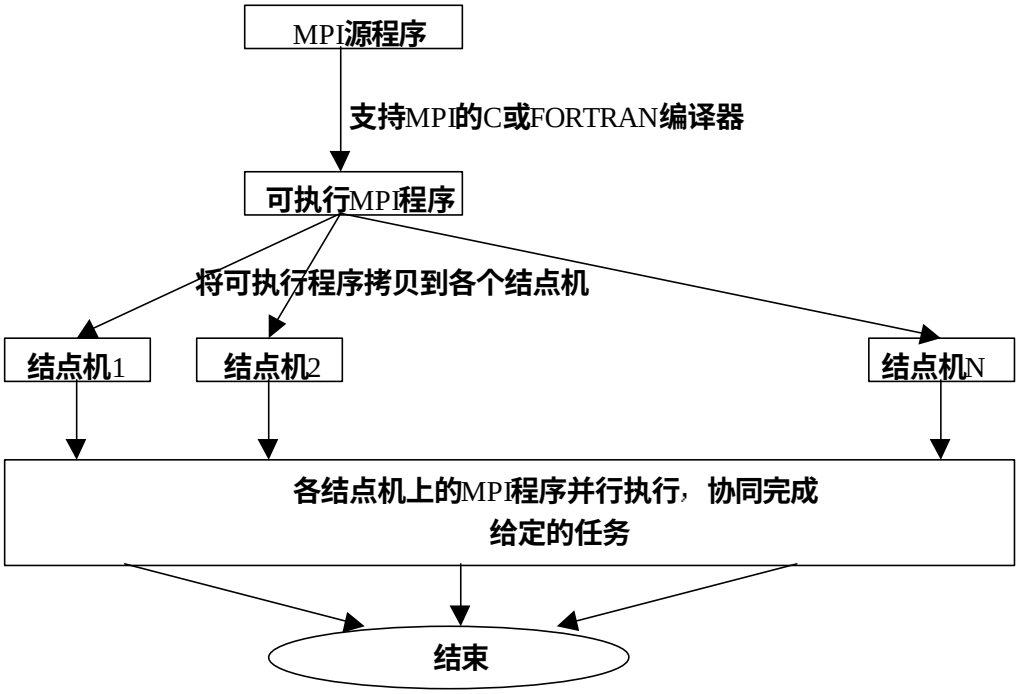


图 37　MPI程序的执行过程

MPI程序的执行步骤（图 37）一般为:

① 编译得到MPI可执行程序

若在同构的系统上执行，则只需编译一次，若系统是异构的，则在每一个异构的系统上都需要对MPI源程序进行编译。

② 将可执行程序拷贝到各个结点机上

对于编译得到的可执行程序，将它拷贝到将要运行的各个结点机上。

③ 通过mpirun命令并行执行该MPI程序

10.1.5 放权

为了能够在多个不同的机器上运行MPI程序，首先需要其它机器对启动MPI程序的机器放权，即允许启动MPI程序的机器能够访问其它的机器。主要有两种方式，一种是在其它所有机器的/etc/hosts.equiv文件中加入启动MPI程序的机器名。比如将要在tp5这台机器上启动16个MPI进程，用到的机器分别是tp1,tp2,...,tp16，则需要在tp1,...,tp16机器上的/etc/hosts.equiv文件中都加入一行

```
tp5
```

这样就表示其它的机器都允许tp5进行访问，为了在一台机器上同时运行多个进程，在启动进程所在的机器的/etc/hosts.equiv文件中，也要加入自身的机器名。如果/etc/hosts.equiv文件不存在，则需要自己创建它。

另一种方式是通过.rhosts文件来放权。即在MPI程序所要用到的各个机器上，在运行该程序的帐户的home路径下创建一个.rhosts文件，在该文件中写上允许那些机器的那些帐户对自己的帐户进行访问。比如在tp1机器的pact帐户下，它允许tp5的pact帐户对它进行访问，则需要在tp1机器上的pact帐户的home下创建一个.rhosts文件，该文件加入一行

```
tp5    pact
```

为了简单起见，最好在各个机器上都建立相同的帐户名，使得MPI程序在相同的帐户下运行。

10.1.6 运行命令和配置文件

最简单的MPI运行命令是

mpirun –np N program

其中N是同时运行的进程的个数，program是可执行MPI程序名。以这种方式进行执行，需要首先对可用的机器进行配置，配置文件是$(HOME)/mpich/util/machines/machines.LINUX在这个文件中，每一行写上可用的机器名，比如

```
tp5.cs.tsinghua.edu.cn
tp1.cs.tsinghua.edu.cn
tp2.cs.tsinghua.edu.cn
tp3.cs.tsinghua.edu.cn
tp4.cs.tsinghua.edu.cn
tp8.cs.tsinghua.edu.cn
```

这样就有6台机器可供MPI使用。使用这种方式启动时，可执行程序必须放在不同机器的相同帐户的相同路径下。比如在tp5.cs.tsinghua.edu.cn上$(HOME)/mpich/examples/basic/下运行

mpirun –np 6 cpi

则需要在{tp1,tp2,tp3,tp4,tp8}上的$(HOME)/mpich/examples/basic/下都有该cpi程序。

如果不使用缺省的配置文件，则需要在命令行给出配置文件，该配置文件的格式和mashines.LINUX相同。比如

mpirun –machinefile hosts –np 6 cpi

只需在hosts中给出可使用的机器名字即可。

还有一种更为灵活的配置方式，它允许可执行程序有不同的名字，有不同的路径，它的启动方式是

mpirun –p4pg pgfile cpi

它的配置文件pgfile的格式如图 38所示。

<机器名>	<进程数>	<程序名>
<机器名>	<进程数>	<程序名>
<机器名>	<进程数>	<程序名>

图 38 配置文件的通用格式

需要多少机器，就写几行。注意在这种启动格式中，不需要指出启动多少个进程，进程数由配置文件指定。一种可能的格式如图 39所示。

tp5	0	/home/pact/mpich/examples/basic/cpi
tp1	1	/home/pact/mpich/examples/basic/cpi
tp2	1	/home/pact/mpich/examples/basic/cpi
tp3	1	/home/pact/mpich/examples/basic/cpi
tp4	1	/home/pact/mpich/examples/basic/cpi
tp8	1	/home/pact/mpich/examples/basic/cpi

图 39 配置文件示例

注意第一行的0并不表示在tp5上没有进程，这里0特指在tp5上启动MPI程序的执行。

mpirun是MPI程序的启动脚本，它可以简化作业的启动程序，并且尽可能把不同的设备特征屏蔽掉，提供给用户一个通用的MPI并行机的概念。

MPI程序的一般启动方式是

mpirun -np <number of processes> <program name and arguments>

一般MPI**会自动决定使用什么样的设备和什么样的结构，若**MPI**无法决定，则可以通过选择开关指定，可用的设备选项有**：

chameleon (**包括**chameleon/pvm, chameleon/p4, ...)
meiko (**使用**meiko**设备**)
paragon (paragon**上的**ch_nx**设备**)
p4 (**工作站机群上的**ch_p4**设备**)
ibmspx (IBM SP2**上的**ch_eui)
anlspx (ANLs SPx**上的**ch_eui)
ksr (KSR 1**和**2**上的**ch_p4)
sgi_mp (SGI**多处理器上的**ch_shmem)
cray_t3d (Cray T3D**上的**t3d)
smp (SMPs**上的**ch_shmem)
execer (**一个定制脚本，目前还不稳定**)

对于MPI**无法识别的选项，它将抛弃，完整的**MPI**运行方式为**:

mpirun [mpirun_options...] <progname> [options...]

-arch <architecture> **指明结构信息，在**${MPIR_HOME}/util/machines**下有对应的** machines.<arch> **文件**

-h **帮助信息**

-machine <machine name> use startup procedure for <machine name>

-machinefile <machine-file name> **列出可选的机器**

-np <np> **指出运行需要的处理器个数**

-nolocal **不在本地机运行**

-stdin filename **用给定的文件名作为标准输入**

-t **用于测试，只显示执行的命令，而不实际运行它**

-v **尽可能显示详细的信息**

-dbx **在**dbx**下启动第一个进程**

-gdb **在**gdb**下启动第一个进程**

-xxgdb **在**xxgdb**下启动第一个进程**

-tv **在**totalview**上启动**

针对NEC - CENJU-3**的特殊选项有**

-batch**作为批处理作业执行**

-stdout filename **用指定的文件名作为输出**

-stderr filename **用指定的文件名作为标准输出**

针对Nexus**设备的特殊选项有**

-nexuspg filename**用给定的文件作为配置文件，并且使**-np -nolocal**无效，自动选择** -leave_pg

-nexusdb filename **使用**Nexus**给定的资源数据库**

针对工作站机群的特殊选项有

-e **用**execer**来启动程序**

-pg **用配置文件来启动一个**p4**程序，而不是**execer

-leave_pg **运行结束后不删除**P4**配置文件**

-p4pg filename**用指定的进程组配置文件而不是临时创建一个，使得**-np**和**-nolocal

无效，自动选择 -leave_pg

-tcppg filename使用指定的tcp进程组配置文件而不是临时创建一个使得-np和-nolocal无效，自动选择 -leave_pg

-p4ssport num 使用p4安全服务程序来启动该程序，该服务器使用的端口号为num，如果num=0，则使用环境变量MPI_P4SSPORT的值，该服务器可以加速进程的启动。如果设置了MPI_USEP4SSPORT和MPI_P4SSPORT的值，其效果就如同-p4ssport 0。

针对批处理环境的特殊选项

-mvhome 将可执行程序移到home路径下

-mvback files 将指定的文件移到当前路径下

-maxtime min 以分钟为单位的最大运行时间

-nopoll 不使用查询模式进行通信

-mem value 每个结点需要的内存

-cpu time 硬件CPU约束时间

针对IBM SP2的特殊选项

-cac name 指定ANL 调度期。

针对Intel Paragon的特殊选项

-paragontype name 选择递交作业的方式

-paragonname name 指定运行作业的远程shells的名字

-paragonpn name 在Paragon上运行部分的名字。

异构系统上的运行

通过指定多个-arch -np 参数对，可以在不同的结构上协同运行一个MPI程序。比如利用本地机sun4和另一个机器rs6000同时执行一个程序，在sun4上启动2个进程，在rs6000上启动3个进程，则启动命令为

```
mpirun -arch sun4 -np 2 -arch rs6000 -np 3 program
```

如果不同机器上的程序名字不同，比如sun4上的程序名字是program.sun4，rs6000上的机器名字是program.rs6000，则可以用%a代替机器名。

```
mpirun -arch sun4 -np 2 -arch rs6000 -np 3 program.%a
```

如果执行程序的存放路径也不相同，比如分别存放在 /tmp/me/sun4 和 /tmp/me/rs6000下，则启动命令为

```
mpirun -arch sun4 -np 2 -arch rs6000 -np 3 /tmp/me/%a/program
```

10.1.7 其它可执行命令

mpiman

启动MPI的手册帮助程序，它提供两种显示方式，一种是UNIX的man方式，一种是通过Web的HTML格式。缺省情况下， mpiman使用xman, 即X窗口系统的手册帮助浏览器来阅读各个帮助页面。其它的开关选项是

-xmosaic 指定使用xmosaic Web浏览器

-mosaic 指定使用mosaic Web浏览器

-netscape 指定使用netscape Web浏览器

-xman 指定使用X窗口系统的xman手册浏览器

-man 指定使用man program (比如mpiman -man MPI_Send)

mpireconfig

根据模板文件产生make文件，它可以根据特定MPICH的配置，将模板中的变量替换为合适的参数形成所需要的make文件。

命令格式是

mpireconfig filename

filename 是将要产生的文件名，但是相应的filename.in文件必须是已经存在了的。

10.2 Windows NT环境下的MPICH

这里介绍的NT上的MPICH版本是MPICH.NT.1.2.0.4。该版本支持tcp/ip，共享内存和VIA连接。同一个机器上的进程间通信是通过共享内存队列实现的，而不同机器上的进程间通信是通过sockets或VI实现的。

本包可以用MS Visual C++ 6.0 和 Digital Fortran 6.0编译，使用其它的FORTRAN编译器需要修改编译选项。动态连接库包括MPI、PMPI调用的C和FORTRAN实现。

10.2.1 安装

从ftp://ftp.mcs.anl.gov/pub/mpi/nt/mpich.nt.1.2.0.4.all.zip下载该压缩包解压后执行setup命令，便会将MPICH自动安装到NT上。缺省安装路径是c:\Program Files\Argonne National Lab\MPICH.NT.1.2.0.4

安装的内容包括：

- 运行时动态连接库
- MPI程序的启动程序（launcher）
- 若需编译MPI应用，则需安装sdk
- 源程序树是可选的

10.2.2 编译

首先是C/C++程序的编译：要编译一个用C/C++编写的MPI程序，用MS Visual C++来编译的步骤是：

- 创建一个新的makefile或项目（project）文件，然后进行必要的设置。
- 在include路径增加 [MPICH Home]\include
- 设置开关选项

  Debug - /MTd

  Release - /MT
- 需要连接的库

  Debug - ws2_32.lib mpichd.lib pmpichd.lib romiod.lib

  Release - ws2_32.lib mpich.lib pmpich.lib romio.lib

  pmpich*.lib 库是在MPI程序中使用了PMPI_ *调用是才需要的
- 增加库lib路径 [MPICH Home]\lib

在项目中加入MPI源程序，然后执行build

FORTRAN 程序的编译：对于FORTRAN程序，建议使用Visual Fortran 6+编译器，

- 增加 mpif.h
- Visual Fortran 6+的编译开关

  /iface:cref

  /iface:nomixed_str_len_arg
- 连接的库和C/C++的相同

NT下的MPICH包可以被重新编译，还可以支持VIA。由于其配置比较复杂，这里不再介绍。

10.2.3 配置和运行

NT下的MPICH有三种启动运行的方式，它们分别是:

- 使用Remote Shell Server提供的MPIRun.exe来启动，它是推荐使用的方式
- 使用Simple Launcher提供的MPIRun.exe来启动，这种方式在能力上受到限制
- 使用MPICH的数据库服务来自己手工启动程序的执行

Remote Shell Server驻留在各个将运行MPI程序的主机上，它是一个DCOM server，该服务属于SYSTEM帐户，当MPIRun与Remote Shell Server连接时，该服务以启动该MPIRun程序的用户的身份，在该用户的环境下启动相应的MPI进程。

一种用使用Remote Shell Server提供的MPIRun.exe来启动MPI程序的方式是使用如下格式:

MPIRun -np 进程数 程序名

MPIRun.exe放在c:\Program Files\Argonne National Lab\MPICH.NT.1.2.0.4\RemoteShell\Bin下面，为了使用方便，可以把该路径加入系统的环境变量中。

使用这一格式执行MPI程序时，相应的可执行MPI程序必须放在所有使用的机器上的相同路径下，而这些进程到底在哪些机器上执行，需要先用MPIConfig命令来设置。

MPIConfig放在c:\Program Files\Argonne National Lab\MPICH.NT.1.2.0.4\RemoteShell\Bin下面，为了在多个不同的机器上运行一个MPI程序而又不需要指定一个配置文件，需要通过运行MPIConfig命令来配置。MPIConfig 查找可用的机器，并且让用户进行选择后将这些机器的名字写入注册表中，这样当启动MPI程序时，就可以直接在注册表中选取机器，然后在这些机器上运行程序。它有几个选项:

- Refresh: 重新搜索可用的机器
- Find: 检查注册表中的机器是否可以成功连接并协同运行，检查结束后选种的机器是启动进程成功安装的机器。
- Verify:本版本还没有实现该功能，检验DCOM server是否可用。
- Set:设置如下参数
  - "set HOSTS" 将选中的机器名写入每一个机器的注册表。以后用MPIRun在任何一个选中的机器上执行时都使进程在这里列出的机器上运行。
  - "set TEMP" 设置临时目录，该目录必须对remote shell service和用户的MPI程序都是可读写的。却省是C:\

- **最后是设定**timeout**时间**

另一种用使用Remote Shell Server**提供的**MPIRun.exe**来启动**MPI**程序的方式是使用如图**40**所示的格式**:

```
MPIRun configfile [-logon] [args ...]    或者
MPIRun -np #processes [-logon] [-env "var1=val1|var2=val2..."]
 executable [args ...]    或者
MPIRun -localonly #processes [-env "var1=val1|var2=val2..."]
 executable [args ...]
```

图 40 NT**下启动**MPI**程序的几种方式**

配置文件的格式如图 41**所示**。

```
exe c:\somepath\myapp.exe
  或 \\host\share\somepath\myapp.exe
[args arg1 arg2 arg3 ...]
[env VAR1=VAL1|VAR2=VAL2|...|VARn=VALn]
hosts
hostA #procs [path\myapp.exe]
hostB #procs [\\host\share\somepath\myapp2.exe]
hostC #procs
...
```

图 41 NT**下运行**MPI**程序配置文件的格式**

中括号内的部分是可选的，下面给出几个具体的配置文件的例子。

假设有8**台机器可以使用，分别是**NT01，NT02，...，NT08，**可执行的**MPI**程序是**testmpint，**该可执行程序都放在各个机器的**c:\mpint **目录下面，则对于图** 42**所示的配置文件**mpiconf1

```
exe c:\mpint\testmpint.exe
hosts
NT01    1
NT02    1
NT03    1
NT04    1
NT05    1
NT06    1
NT07    1
NT08    1
```

图 42 NT**上**MPI**配置文件示例**1**（相同路径和名字）**

可以通过命令

mpirun mpiconf1

启动这一程序，它使得testmpint**在**8**台机器同时执行。若各个机器上可执行程序存放的路径**

不同，则需要明确指出，而不能将路径省略，同时它也允许不同机器上的可执行程序的名字互不相同。如图 43配置文件mpiconf2所示

```
exe c:\mpint\testmpint.exe
hosts
NT01    1    c:\mpint\testmpint.exe
NT02    1    d:\mpint\testmpint2.exe
NT03    1    e:\mpint\testmpint1.exe
NT04    1    c:\testmpint.exe
NT05    1    c:\test\testmpint9.exe
NT06    1    d:\abc\abc.exe
NT07    1    c:\temp\testmpint7.exe
NT08    1    c:\mpint\testmpint.exe
```

图 43 NT上MPI配置文件示例2（不同路径和名字）

通过命令

mpirun mpiconf2

启动这一程序，它使得testmpint在8台机器同时执行，各个机器上可执行程序的路径是不同的，各个机器为可执行程序起的名字也可以是不同的，但最终的执行效果是相同的。

在一台机器上，还可以同时启动多个进程，如图 44配置文件mpiconf3所示

```
exe c:\mpint\testmpint.exe
hosts
NT01    2    c:\mpint\testmpint.exe
NT02    3    d:\mpint\testmpint2.exe
NT03    1    e:\mpint\testmpint1.exe
NT04    4    c:\testmpint.exe
NT05    1    c:\test\testmpint9.exe
NT06    1    d:\abc\abc.exe
NT07    2    c:\temp\testmpint7.exe
NT08    1    c:\mpint\testmpint.exe
```

图 44 NT上MPI配置文件示例3（一个机器上多个进程）

通过命令

mpirun mpiconf2

启动该MPI程序时，将有15个进程同时执行，其中NT01上2个，NT02上3个，...，NT08上1个。

通过命令

mpirun -localonly 8 testmpint

只在本地机器上单机模拟8个处理器，同时在一台机器上运行8个进程。

对于MPIRun.exe的其它选择开关含义如下：

-localonly #procs -tcp

选择开关 -tcp 强制使用sockets而不是共享内存。

-env "var1=val1|var2=val2|var3=val3|...varn=valn"

设置环境变量

-logon

该选项导致mpirun启动时要求给出帐户和密码。使用它可以将可执行程序放在共享目录下面，不使用它则可执行程序必须放在不同的机器上。执行mpiregister.exe将帐户和密码加入注册表可以避免提示。

MPIRegister.exe 是注册程序，它放在c:\Program Files\Argonne National Lab\MPICH.NT.1.2.0.4\RemoteShell\Bin\MPIRegister.exe，使用它可以将帐户和密码加密后加入注册表。MPIRun.exe在启动程序时使用该信息，没有注册信息将导致mpirun每次启动时都提示给出帐户和密码。其运行方式是

MPIRegister 或

MPIRegister -remove

首先它提示给出帐户，然后要求输入两遍密码，最后提示将它永久地存下来，若保存，则以后mpirun仍然可以使用它，否则当启动重新启动时该注册信息会丢失。

而

MPIRegister -remove

将删除注册信息。

10.2.4 小结

并行程序设计如同其它的程序设计方法一样，如果不亲自动手编写程序，是不可能真正把握住它的，本章的目的就在于不仅使读者能够理解并行程序的设计方法，而且能够在并行环境下实际运行和调试并行程序。因此本章用了大量的篇幅来讲解和MPI程序的运行有关的细节，希望能够使初学者少走弯路，节省时间，其中所有的内容都是经过实际运行过的，可以说是作者花了大量时间摸索和总结出来的，在这里贡献给广大的读者，希望能使更多的人迅速掌握和使用并行计算这一先进工具来解决各自遇到的不同问题。

第11章 常见错误

本章指出了在MPI程序设计中和程序运行中经常出现的错误类型及其改正办法，以利于初学者少走弯路，供读者参考。

11.1 程序设计中的错误

- 缺少ierr参数

在Fortran源程序中，MPI子程序的最后一个参数ierr用于返回错误代码，而C形式的调用中却没有这一参数，因此该参数经常被漏掉；同时，由于有些Fortran编译器检查得严，有些Fortran编译器并不检查参数是否严格匹配，因此在编译阶段不被发现，导致在运行时出现一些莫名其妙的错误。

- 对status的错误声明

status是一个整数数组，而不是一个整数，在MPI_Recv调用中，一些返回信息保存在status中，由于一些编译器对它的检查不严格，这样在程序运行时，便经常出现写变量出界而产生不可预见的错误。

- 字符串的错误声明

在Fortran 中，字符串和字符数组是不同的，这和C不一样，对于一个具有10个字符的字符串string10，应该声明为

```
character*10 string10
```

如果声明为

```
character string10（10）
```

则为包含10个字符的字符数组。Fortran程序的MPI调用，字符串是不能用字符数组代替的。

- 以MPI_开始声明变量

这样很容易和MPI自身的调用名字或常量名字相同，因而造成混淆，故应避免声明以MPI_开始的变量或常量。

- argc，argv参数的使用

为了程序的通用性和移植性，最好不要在MPI程序中对argc和argv进行引用。在C程序的MPI调用中，可以将argc和argv传递给MPI_Init，这样，就可以把argc和argv的值传递给所有的进程，但是MPI标准却并不要求一定要实现这一要求，因此对于一些MPI实现可以不将argc和argv传递给所有的进程，这样，如果一个MPI程序依赖argc或argv这一参数特征，则在一定的条件下就会出错。

- 不要在MPI_Init前和MPI_Finalize后写可执行程序代码

在MPI程序中，在MPI_Init前和MPI_Finalize后的可执行程序如何执行，MPI标准是没有定义的，因此这些位置的程序代码的执行会出现不可预料的结果。

- 不要用MPI_Recv和MPI_Bcast匹配

在语义上，可以认为MPI_Bcast是一个“多次发送”的过程，这是没有错误的，但在语法上，不可以将MPI_Bcast等同为多个MPI_Send语句，因此，在MPI程序中，就不能用MPI_Recv去和MPI_Bcast相匹配。MPI_Bcast是集合操作，在每一个进程中都必须有相应的MPI_Bcast语句，而不是MPI_Recv语句。

- 不能假定MPI支持多线程

在MPI标准中，并不要求MPI必须支持多线程MPI调用，因此在编写MPI程序时就不能有这样的假设。实际上，我们的确尝试过将pthread和MPICH-1.1.1结合起来，但是目前为止MPICH还不支持多线程。

- MPI_Send和MPI_Recv的不合理次序

对于进程1和进程2相互发送和接收数据，MPI语句的不合理次序为：

进程1	进程2
MPI_Send 发送数据到进程2	MPI_Send发送数据到进程1
MPI_Recv 从进程2接收数据	MPI_Recv从进程1接收数据

之所以不合理，是因为当系统内存空间缺乏时，进程1和进程2都无法将数据发送完成，因而也无法从对方接收数据，造成程序的死锁。避免这种情况常采取的措施是：

- 将发送和接收重叠起来，即当一方在发送时另一方处于接收状态，这样可以避免因相互等待而造成死锁。
- 对于成对的交互发送和接收，鼓励使用MPI_Sendrecv语句，因为该语句本身提供了优化的可能，可以既提高效率，又避免编写单独的MPI_Send和MPI_Recv语句可能造成的死锁问题。
- 用MPI_Buffer_attach来显式分配用户自己的存储空间。
- 鼓励使用非阻塞操作MPI_Isend和MPI_Irecv来代替相应的阻塞操作

- 数据类型不匹配

在发送者缓冲区中的每个变量类型必须匹配该发送操作为输如指定的类型；由发送操作指定的类型必须匹配由接收操作指定的类型；在接收缓冲区中的每个变量的类型必须匹配由接收操作为指定的类型。不能遵从这三个规则的程序是错误的。

- 接收缓冲区溢出

接收缓冲区太小，但允许接收的数据容量却超过了接收缓冲区的大小。

- 行优先与列优先

C语言数组在内存中是以行优先存放，因此一行的数据是连续的，但同一列的数据是不连续的；FORTRAN语言数组在内存中以列优先存放，因此一列的数据是连续的，但同一行的数据是不连续的。

- 正确使用地址

在C或FORTRAN中相继定义的变量不一定存储在相连的地方，因此需要注意对一个变量的偏移不一定对另一个变量适用。另外，对于使用段地址空间的机器，地址是不统一的并且地址计算有各自的特点。所以address(相对于MPI_BOTTOM开始地址的偏移量)的使用是有限制的，如果几个变量属于同一个矩阵则属于同一个顺序存储区，在FORTRAN中属于同一个COMMON块，在C 中属于同一个结构。

11.2 运行时的错误

- 访问拒绝

初次运行MPI程序，经常遇到的一个错误的访问拒绝，这是因为没有配置好各个机器（包括启动机器自身）的权限管理，关于放权的问题请参见MPICH的安装与运行一章的相关部分。

- 没有将可执行程序拷贝到各个计算结点

MPI程序是SPMD程序，其执行需要可执行程序和该程序所需要的数据在各个计算结点上都有一份拷贝。

- 重新编译后的程序没有在其它的结点上更新

对程序修改后进行了重编译，但是该编译结果仅仅在本地机被修改，而远程的其它机器还是原来的版本。这种错误一般很难识别，因为该程序在许多情况下都是可以运行的，只是执行结果和预期的不同。

- 缺少可执行程序名

注意在NT的某些运行方式在命令行可以省略可执行程序名，但是在 Lilux环境下MPI程序的启动必须有可执行程序名。

- 配置文件路径错

缺省配置不符合程序运行的要求，有时不同机器上相同帐户的绝对路径是不同的，因此需要特别注意，不能想当然的写配置文件。

11.3 小结

在程序书写和运行中的错误是多种多样的，远远超出了本章介绍的内容，这里只是给出一些基本的错误类型及其解决办法，避免在一些简单的次要问题上花太多的时间，将主要精力用于设计高质量的MPI程序和实际问题的解决上。

第三部分　高级MPI并行程序设计

这一部分介绍为了提高MPI程序的性能、通用性、移植性等而引入的MPI高级特性，包括非阻塞通信、组通信、不连续数据的传送以及虚拟进程拓扑等。

第12章 非阻塞通信MPI程序设计

前面介绍的通信方式都是阻塞通信，在本章介绍非阻塞通信。非阻塞通信主要用于实现计算与通信的重叠。本章在简单回顾一下阻塞通信之后，重点介绍非阻塞通信。

12.1 阻塞通信

从程序员的角度看，当一个阻塞通信正确返回后，其后果是：1 该调用要求的通信操作已正确完成，即消息已成功发出或成功接收；2 该调用的缓冲区可用，若是发送操作，则该缓冲区可以被其它的操作更新，若是接收操作，该缓冲区中的数据已经完整，可以被正确引用。

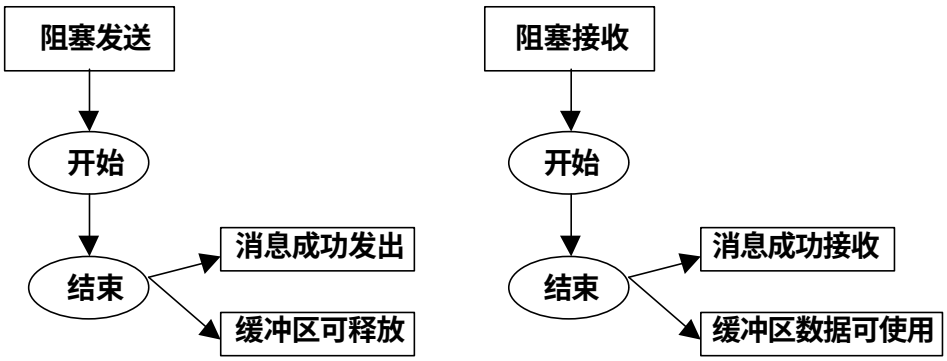


图 45 阻塞消息发送和接收

在阻塞通信中，对于接收进程，在接收消息时，除了要求接收到的消息的消息信封和接收操作自身的消息信封相一致外，还要求它接收到的消息是最早发送给自己的消息。若两个消息的消息信封都和自己的消息信封吻合，则必须先接收首先发送的消息，哪怕后发送的消息首先到达，该接收操作也必须等待第一个消息。

如图 46所示，进程0先后发送两条消息给进程1，进程1的第一个接收语句可以与任何一个发送语句的消息相匹配，但是，根据有序接收的语义约束，由进程0发送的第一个消息必须被进程0的第一个接收语句接收,而由进程0发送的第二个消息必须被进程1的第二个接收语句接收。如果进程0的第一个消息先到达，进程1的第一条接收语句也不能接收它。

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
   CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE   IF（rank .EQ.1） THEN
   CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
   CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

程序 29 消息接收次序示例

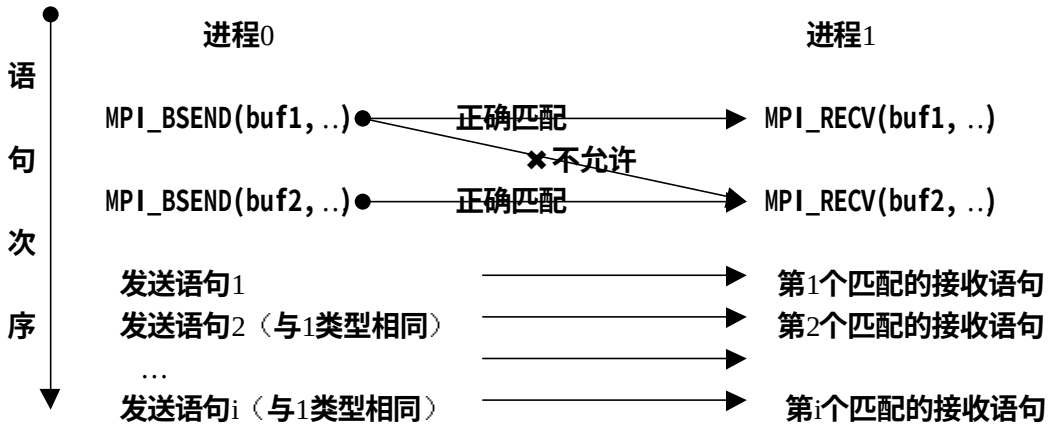


图 46 消息的接收次序

12.2 非阻塞通信简介

MPI提供的非阻塞通信调用的函数十分丰富，所有阻塞通信的形式都有相应的非阻塞通信的形式。

由于通信经常需要较长的时间，在阻塞通信还没有结束的时候，处理机只能等待，这样就浪费了处理机的计算资源。一种常见的技术就是设法使计算与通信重叠，非阻塞通信可以实现这一目的。非阻塞通信主要用于计算和通信的重叠，从而提高整个程序执行的效率，此外，非阻塞通信还可以实现一些特殊的控制功能。

对于非阻塞通信，不必等到通信操作完全完成便可以返回，该通信操作可以交给特定的通信硬件去完成，在该通信硬件完成该通信操作的同时，处理机可以同时进行计算操作，这样便实现了计算与通信的重叠。通过计算与通信的重叠，可以大大提高程序执行的效率。这一方法和通过异步I/O实现I/O与计算的重叠思路是完全一样的。

当然，由于当非阻塞通信调用返回时一般该通信操作还没有完成，因此对于非阻塞的发送操作，发送缓冲区必须等到发送完成后才能释放，这样便需要引入新的手段（非阻塞通信

完成对象）让程序员知道什么时候该消息已成功发送；同样，对于非阻塞的接收操作，该调用返回后并不意味着接收消息已全部到达，必须等到消息到达后才可以引用接收到的消息数据。

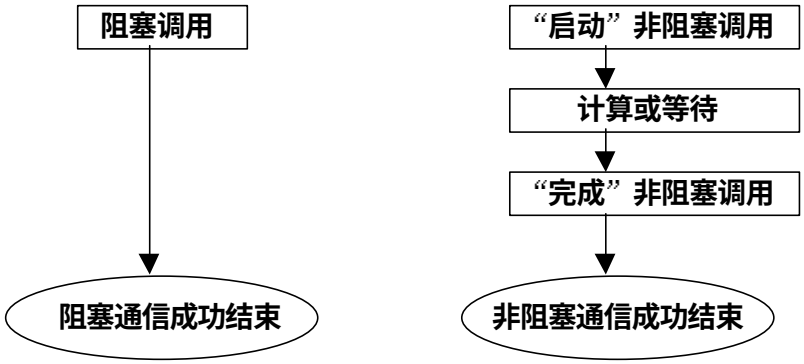


图 47 阻塞与非阻塞调用的对比

从上可以看出，对于阻塞通信，只需要一个调用函数即可以完成，但是对于非阻塞通信，一般需要两个调用函数，首先是非阻塞通信的“启动”，但启动并不意味着该通信过程的完成，因此，为了保证通信的完成，还必须调用与该通信相联系的通信“完成”调用接口，通信完成调用才真正将非阻塞通信完成。

非阻塞通信和四种通信模式相结合，可以有四类不同的形式；针对某些通信是在一个循环中重复执行的情况，为了进一步提供优化的可能和提高效率，MPI又引入了重复非阻塞通信方式，对于重复非阻塞通信，和四种通信模式相结合，又有四种不同的具体形式。

各种非阻塞通信形式，其效果都是将非阻塞通信的基本特征和具体的通信模式相结合后的综合体现。比如非阻塞缓存发送就是非阻塞+缓存的结果。

由于非阻塞通信返回后并不意味着通信的完成，MPI还提供了各种非阻塞通信的完成方法和完成检测方法。MPI可以一次完成一个非阻塞通信，也可以一次完成所有的非阻塞通信，还可以一次完成任意一个或任意多个非阻塞调用；对于非阻塞通信是否完成的检测也有以上各种形式。

表格 7 非阻塞MPI通信模式

通信模式		发送	接收
标准通信模式		MPI_ISEND	MPI_IRECV
缓存通信模式		MPI_IBSEND	
同步通信模式		MPI_ISSEND	
就绪通信模式		MPI_IRSEND	
重复 非阻塞通信	标准通信模式	MPI_SEND_INIT	MPI_RECV_INIT
	缓存通信模式	MPI_BSEND_INIT	
	同步通信模式	MPI_SSEND_INIT	
	就绪通信模式	MPI_RSEND_INIT	

表格 8 非阻塞通信的完成与检测

非阻塞通信的数量	检测	完成
一个非阻塞通信	MPI_TEST	MPI_WAIT
任意一个非阻塞通信	MPI_TESTANY	MPI_WAITANY
一到多个非阻塞通信	MPI_TESTSOME	MPI_WAITSOME
所有非阻塞通信	MPI_TESTALL	MPI_WAITALL

发送和接收操作的类型虽然很多，但只要只要消息信封相吻合，并且符合有序接收的语义约束，任何形式的发送和任何形式的接收都可以匹配。

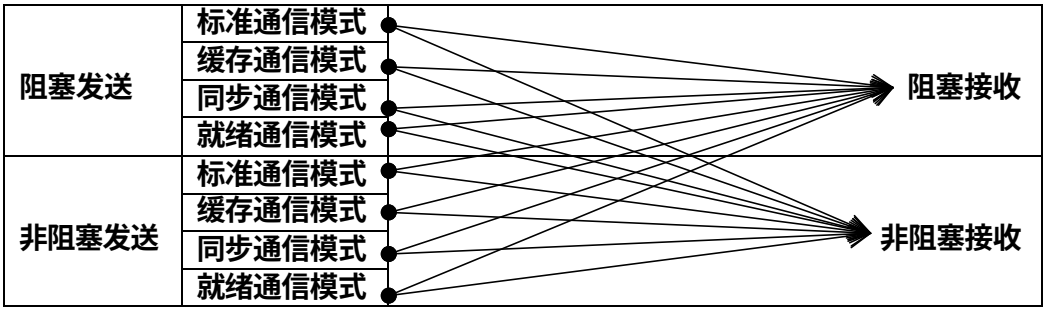


图 48 不同类型的发送与接收的匹配

12.3 非阻塞标准发送和接收

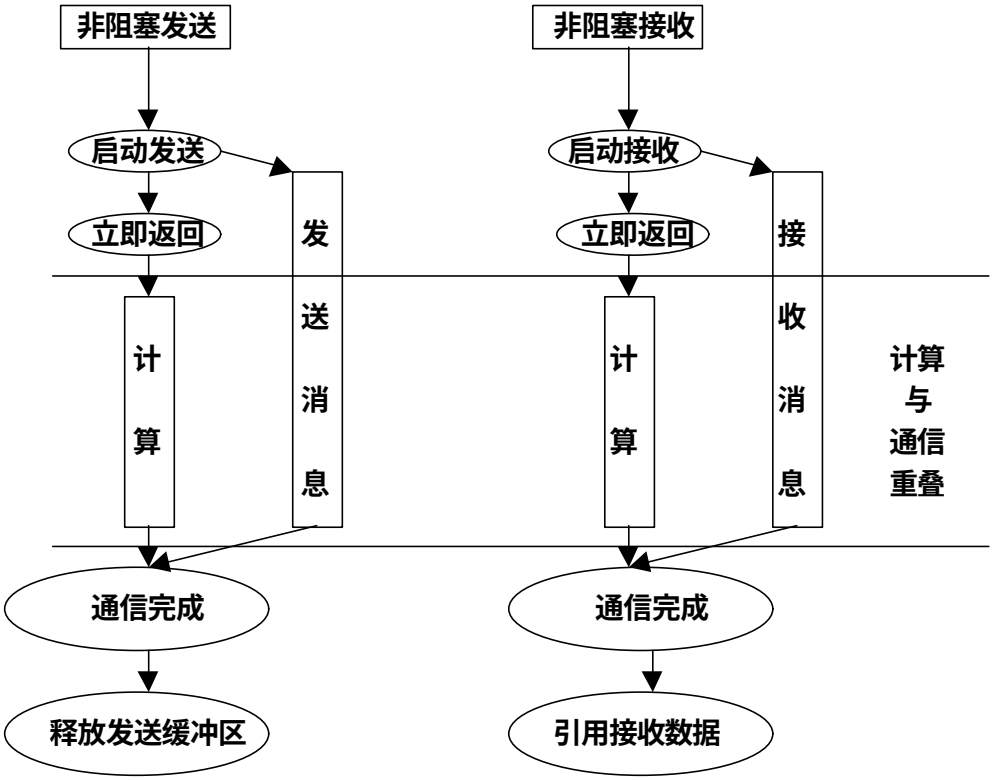


图 49 标准非阻塞消息发送和接收

MPI_ISEND**的功能是启动一个标准的非阻塞发送操作，它调用后立即返回**。MPI_ISEND**的调用返回并不意味着消息已经成功发送，它只表示该消息可以被发送。和阻塞发送调用相比，它多了一个参数**request**，这一参数是一个用来描述非阻塞通信状况的对象，这里不妨称之为“非阻塞通信对象”，通过对这一对象的查询，就可以知道与之相应的非阻塞发送是否完成。在后面将介绍如何对这一对象进行查询。**

MPI_IRECV**的功能是启动一个标准的非阻塞接收操作，它调用后立即返回**。MPI_IRECV**调用的返回并不意味着已经接收到了相应的消息，它只表示符合要求的消息可以被接收。和阻塞接收调用相比，它多了一个参数**request**，这一参数的功能和非阻塞发送一样，只不过在这里是用来描述非阻塞接收的完成状况，通过对这一对象的查询，就可以知道与之相应的非阻塞接收是否完成。**

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
IN buf **发送缓冲区的起始地址(可选数据类型)**
IN count **发送数据的个数(整型)**
IN datatype **发送数据的数据类型(句柄)**
IN dest **目的进程号(整型)**
IN tag **消息标志(整型)**
IN comm **通信域(句柄)**
OUT request **返回的非阻塞通信对象(句柄)**
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request)
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type>BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI调用接口 20 MPI_ISEND

MPI_IRECV(buf, count, datatype, source, tag, comm, request)
OUT buf **接收缓冲区的起始地址(可选数据类型)**
IN count **接收数据的最大个数(整型)**
IN datatype **每个数据的数据类型(句柄)**
IN source **源进程标识(整型)**
IN tag **消息标志(整型)**
IN comm **通信域(句柄)**
OUT request **非阻塞通信对象(句柄)**
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Request *request)
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,
IERROR)
<type>BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI调用接口 21 MPI_IRECV

12.4 非阻塞通信与其它三种通信模式的组合

对于阻塞通信的四种消息通信模式，标准通信模式、缓存通信模式、同步通信模式和接收就绪通信模式，非阻塞通信也具有相应的四种不同模式。MPI**使用与阻塞通信一样的命名约定，前缀**B,S,R**分别表示缓存通信模式，同步通信模式和就绪通信模式。前缀**I(immediate)**表示这个调用是非阻塞的。**

MPI_ISSEND**开始一个同步模式的非阻塞发送，它的返回只是意味着相应的接收操作已经启动，并不表示消息发送的完成。**

MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
IN buf **发送缓冲区的起始地址(可选数据类型)**
IN count **发送数据的个数(整型)**
IN datatype **发送数据的数据类型(句柄)**
IN dest **目的进程标识(整型)**
IN tag **消息标志(整型)**
IN comm **通信域(句柄)**
OUT request **非阻塞通信完成对象(句柄)**
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request)
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type>BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI调用接口 22 MPI_ISSEND

MPI_IBSEND**开始一个缓存模式的非阻塞发送。与阻塞发送一样，它也需要程序员主动为该发送操作提供发送缓冲区。**

MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)
IN buf **发送缓冲区的起始地址(可选数据类型)**
IN count **发送数据的个数(整型)**
IN datatype **每个数据的数据类型(句柄)**
IN dest **目的进程标识(整型)**
IN tag **消息标志(整型)**
IN comm **通信域(句柄)**
OUT request **非阻塞通信完成对象(句柄)**
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request)
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type>BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)

MPI调用接口 23 MPI_IBSEND

MPI_IRSEND开始一个接收就绪通信模式非阻塞发送。与阻塞通信一样，它也要求当这一调用启动之前，相应的接收操作必须已经启动，否则回出错。

```
MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)
    IN buf          发送缓冲区的起始地址(可选数据类型)
    IN count        发送数据的个数(整型)
    IN datatype     发送数据的数据类型(句柄)
    IN dest         目的进程标识(整型)
    IN tag          消息标志(整型)
    IN comm         通信域(句柄)
    OUT request     非阻塞通信对象(句柄)
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request)
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type>BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

MPI调用接口 24 MPI_IRSEND

12.5 非阻塞通信的完成

对于非阻塞通信，通信调用的返回并不意味着通信的完成，因此，需要专门的通信语句来完成或检查该非阻塞通信，不管非阻塞通信是什么样的形式，对于完成调用是不加区分的。当非阻塞完成调用结束后，就可以保证该非阻塞通信已经正确完成了。

12.5.1 单个非阻塞通信的完成

由于非阻塞通信的返回并不意味着该通信过程的完成，那么如何才能明确得知该非阻塞通信已经完成了呢？MPI提供两个调用MPI_WAIT和MPI_TEST用于这一目的。

MPI_WAIT以非阻塞通信对象为参数，一直等到与该非阻塞通信对象相应的非阻塞通信完成后才返回，同时释放该阻塞通信对象，因此程序员就不需要再显式释放该对象。与该非阻塞通信完成有关的信息放在返回的状态参数status中。

与MPI_WAIT类似，MPI_TEST也以非阻塞通信对象为参数，但是它的返回不一定等到与非阻塞通信对象相联系的非阻塞通信的结束。若在调用MPI_TEST时，该非阻塞通信已经结束，则它和MPI_WAIT的效果完全相同，完成标志flag=true；若在调用MPI_TEST时，该非阻塞通信还没有完成，则它和MPI_WAIT不同，它不必等待该非阻塞通信的完成，可以直接返回，但是完成标志flag=false，同时也不释放相应的非阻塞通信对象。

```
MPI_WAIT(request, status)
      INOUT request     非阻塞通信对象 (句柄)
      OUT status        返回的状态 (状态类型)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
MPI_WAIT(REQUEST, STATUS, IERROR)
      INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI调用接口 25 MPI_WAIT

```
MPI_TEST(request, flag, status)
      INOUT request     非阻塞通信对象(句柄)
      OUT flag          操作是否完成标志(逻辑型)
      OUT status        返回的状态 (状态类型)
int MPI_Test(MPI_Request*request, int *flag, MPI_Status *status)
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
      LOGICAL FLAG
      INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI调用接口 26 MPI_TEST

```
      CALL MPI_COMM_RANK(comm, rank, ierr)
      IF(rank.EQ.0) THEN
        CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
C       非阻塞标准发送
        CALL MPI_WAIT(request, status, ierr)
C       等待该发送的完成
      ELSE (rank .EQ. 1) THEN
        CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
C       标准非阻塞接收
        CALL MPI_WAIT(request, status, ierr)
C       等待该接收操作的完成
      END IF
```

程序 30 非阻塞操作和MPI_WAIT简单的使用方法

12.5.2 多个非阻塞通信的完成

除了一次完成一个非阻塞通信的调用外，MPI**还提供其它的调用，可以一次完成多个已经启动的非阻塞通信调用。**

与一次完成一个非阻塞通信的MPI_WAIT**相对应**，MPI_WAITANY**用于等待非阻塞通信对象表中任何一个非阻塞通信对象的完成，释放已完成的非阻塞通信对象，然后返回，**MPI_WAITANY**返回后**index=I，**即**MPI_WAITANY**完成的是非阻塞通信对象表中的第**I**个对象对应的非阻塞通信，则其效果等价于调用了**MPI_WAIT(array_of_requests[I],status)。

MPI_WAITALL**必须等到非阻塞通信对象表中所有的非阻塞通信对象相应的非阻塞操作都完成后才返回，它在效果上等价于**

```
DO I=1,COUNT
    MPI_WAIT(array_of_requests[I],status)
END DO
```

这里的调用是有序的，而实际上是可以以任意次序调用的，不管以什么样的次序，其最终效果应该是一样的。

MPI_WAITSOME**介于**MPI_WAITANY**和**MPI_WAITALL**之间，只要有一个或多个非阻塞通信完成，则该调用就返回。完成非阻塞通信的对象的个数记录在**outcount**中，相应的对象在**array_of_requests**中的下标记录在下表数组**array_of_indices**中，完成对象的状态记录在状态数组**array_of_statuses**中。**

```
MPI_WAITANY(count, array_of_requests, index, status)
    IN count                 非阻塞通信对象的个数(整型)
    INOUT array_of_requests  非阻塞通信完成对象数组(句柄数组)
    OUT index                完成对象对应的句柄索引(整型)
    OUT status               返回的状态(状态类型)
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
    MPI_Status *status)
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
    INTEGER    COUNT, ARRAY_OF_REQUESTS(*), INDEX,
               STATUS(MPI_STATUS_SIZE)，IERROR
```

MPI**调用接口** 27　MPI_WAITANY

MPI_WAITALL(count, array_of_requests, array_of_statuses)
IN count **非阻塞通信对象的个数(整型)**
INOUT array_of_requests **非阻塞通信完成对象数组(句柄数组)**
OUT array_of_statuses **状态数组(状态数组类型)**
int MPI_Waitall(int count, MPI_Request *array_of_requests,
MPI_Status *array_of_statuses)
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES,
IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*)
INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI**调用接口** 28 MPI_WAITALL

MPI_WAITSOME(incount,array_of_requests,outcount,array_of_indices,array_of_statuses)
IN incount **非阻塞通信对象的个数(整型)**
INOUT array_of_requests **非阻塞通信对象数组(句柄数组)**
OUT outcount **已完成对象的数目(整型)**
OUT array_of_indices **已完成对象的下标数组(整型数组)**
OUT array_of_statuses **已完成对象的状态数组(状态数组)**
int MPI_Waitsome(int incount,MPI_Request *array_of_request, int *outcount,
int *array_of_indices, MPI_Status *array_of_statuses)
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
ARRAY_OF_INDICES,ARRAY_OF_STATUSES, IERROR)
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT,
ARRAY_OF_INDICES(*),
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI**调用接口** 29 MPI_WAITSOME

MPI_TESTANY**用于测试非阻塞通信对象表中是否有任何一个对象已经完成，若有对象完成（若有多个非阻塞通信对象完成则从中任取一个）则令**flag=true,**释放该对象后返回；若没有任何一个非阻塞通信对象完成,则令**flag=false**返回。**

MPI_TESTALL**只有当所有的非阻塞通信对象都完成时，才使得**flag=true**返回，并且释放所有的查询对象，只要有一个非阻塞通信对象没有完成，则令**flag=false**立即返回。**

MPI_TESTSOME**和**MPI_WAITSOME**类似，只不过它可以立即返回。有几个非阻塞通信已经完成，则**outcount**就等于几，而且完成对象在**array_of_requests**中的下标依次记录在完成对象下标数组**array_of_indices**中，完成状态记录在相应的状态数组**array_of_statuses**中。若没有非阻塞通信完成，则返回值**outcount=0。

MPI_TESTANY(count, array_of_requests, index, flag, status)

IN count **非阻塞通信对象的个数(整型)**

INOUT array_of_requests **非阻塞通信对象数组(句柄数组)**

OUT index **非阻塞通信对象的索引或**MPI_UNDEFINED **(整型)**

OUT flag **是否有对象完成(逻辑型)**

OUT status **状态(状态类型)**

int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
int *flag, MPI_Status *status)

MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS,
IERROR)

LOGICAL FLAG

INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX,
STATUS(MPI_STATUS_SIZE)，IERROR

MPI**调用接口** 30 MPI_TESTANY

MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)

IN count **非阻塞通信对象的个数(整型)**

INOUT array_of_requests **非阻塞通信对象数组(句柄数组)**

OUT flag **所有非阻塞通信对象是否都完成(逻辑型)**

OUT array_of_statuses **状态数组(状态数组)**

int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
MPI_Status *array_of_statuses)

MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES,
IERROR)

LOGICAL FLAG

INTEGER COUNT, ARRAY_OF_REQUESTS(*),
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI**调用接口** 31 MPI_TESTALL

MPI_TESTSOME(incount,array_of_requests,outcount,array_of_indices,array_of_statuses)
IN incount 非阻塞通信对象的个数(整型)
INOUT array_of_requests 非阻塞通信对象数组(句柄数组)
OUT outcount 已完成对象的数目(整型)
OUT array_of_indices 已完成对象的下标数组(整型数组)
OUT array_of_statuses 已完成对象的状态数组(状态数组)
int MPI_Testsome(int incount,MPI_Request *array_of_request, int *outcount,
int *array_of_indices, MPI_Status *array_of_statuses)
MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
ARRAY_OF_INDICES,ARRAY_OF_STATUSES, IERROR)
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT,
ARRAY_OF_INDICES(*),
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI调用接口 32 MPI_TESTSOME

12.6 非阻塞通信对象

由于非阻塞通信在该调用返回后并不保证通信的完成，因此需要它提供给程序员一些手段来查询通信的状态，MPI调用通过提供给程序员一个“非阻塞通信对象”，程序员可以通过对这一对象的查询，得到非阻塞通信的相关信息。所有的非阻塞发送或接收通信都会返回一个“非阻塞通信对象”。

使用非阻塞通信对象，可以识别各种通信操作，并判断相应的非阻塞操作是否完成。非阻塞通信对象是MPI内部的对象，通过一个句柄存取，使用非阻塞通信对象可以识别非阻塞通信操作的各种特性，例如发送模式，和它联结的通信缓冲区，通信上下文，用于发送的标识和目的参数，或用于接收的标识和源参数。此外，非阻塞通信对象还存储关于这个挂起通信操作状态的信息。

12.6.1 非阻塞通信的取消

MPI_CANCEL操作允许取消已调用的非阻塞通信，用取消命令来释放发送或接收操作所占用的资源，该调用立即返回。取消调用并不意味着相应的通信一定会被取消。若取消操作调用时相应的非阻塞通信已经开始，则它会正常完成，不受取消操作的影响；若取消操作调用时相应的非阻塞通信还没有开始，则可以释放通信占用的资源，取消该非阻塞通信。对于非阻塞通信，即使调用了取消操作，也必须调用非阻塞通信的完成操作或查询对象的释放操作来释放查询对象。

```
MPI_CANCEL(request)
IN  request          非阻塞通信对象(句柄)
int MPI_Cancel(MPI_Request *request)
MPI_CANCEL(REQUEST,IERROR)
      INTEGER  REQUEST,IERROR
```

MPI调用接口 33　MPI_CANCEL

如果一个非阻塞通信已经被执行了取消操作，则该通信的MPI_WAIT**或**MPI_TEST**将释放取消通信的非阻塞通信对象，并且在返回结果**status**中指明该通信已经被取消。**

```
MPI_TEST_CANCELLED(status,flag)
      IN status          状态 (状态类型)
      OUT  flag          是否取消标志(逻辑类型)
int MPI_Test_cancelled(MPI_Status status, int *flag)
MPI_TEST_CANCELLED(STATUS,FLAG,IERROR)
      LOGICAL  FLAG
      INTEGER  STATUS(MPI_STATUS_SIZE),IERROR
```

MPI调用接口 34　MPI_TEST_CANCELLED

一个通信操作是否被取消，可以通过调用测试函数MPI_TEST_CANCELLED**来检查，如果，**MPI_TEST_CANCELLED**返回结果**flag=true，**则表明该通信已经被成功取消，否则说明该通信还没有被取消。**

下面的程序给出了取消操作以及测试取消操作的使用方法及注意事项。

```
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if (rank == 0) {
        MPI_Send(sbuf, 1, MPI_INT, 1, 99, MPI_COMM_WORLD );
/*      进程0执行标准数据发送*/
    } else if (rank ==1)
    {
        MPI_Irecv( rbuf, 1, MPI_INT, 0, 99,
                    MPI_COMM_WORLD, request);
/*      进程1执行非阻塞接收      */
        MPI_Cancel( request); /* 然后立即释放该接收操作*/
        MPI_Wait(&request,&status);/* 即使该通信被取消，也必须执行完成操作*/
        MPI_Test_cancelled(&status,&flag);/*测试取消操作是否成功*/
        if (flag)   MPI_Irecv( rbuf, 1, MPI_INT, 0, 99
                    MPI_COMM_WORLD, request);/* 若取消成功，则需要再执行一次接收操作*/
    }
```

程序 31　非阻塞通信的取消

12.6.2 非阻塞通信对象的释放

当程序员能够确认一个非阻塞通信操作完成时，可以直接调用非阻塞通信对象释放语句MPI_REQUEST_FREE**将该对象所占用的资源释放，而不是通过调用非阻塞通信完成操作来间接地释放。原来的非阻塞通信对象**request**变为**MPI_REQUEST_NULL**。一旦执行了释放操作，非阻塞通信对象就无法再通过其它任何的调用访问。但是，如果与该非阻塞通信对象相联系的通信还没有完成，则该对象的资源并不会立即释放，它将等到该非阻塞通信结束后再释放，因此，非阻塞通信对象的释放并不影响该非阻塞通信的完成。**

```
MPI_REQUEST_FREE(request)
INOUT  request          非阻塞通信对象
int MPI_Request_free(MPI_Request * request)
MPI_REQUEST_FREE(REQUEST, IERROR)
   INTEGER REQUEST, IERROR
```

MPI**调用接口** 35　MPI_REQUEST_FREE

下面的例子给出了非阻塞通信对象的释放方法。

```
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
      IF(rank.EQ.0) THEN
         DO i=1, n
            CALL MPI_ISEND(outval, 1, MPI_real, 1, 0, req, ierr)
            CALL MPI_REQUEST_FREE(req, ierr)
C     释放掉req，仍能保证MPI_ISEND正常完成
            CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, req, ierr)
C     重新使用req作为另一个非阻塞通信的句柄
            CALL MPI_WAIT(req, status, ierr)
C     正常完成MPI_IRECV
         END DO
      ELSE IF(rank.EQ.1) THEN
            CALL MPI_IRECV(inva, 1, MPI_REAL, 0, 0, req, ierr)
            CALL MPI_WAIT(req, status)
C     为了通信的安全，先执行一个接收操作，和进程0的发送操作相匹配
            DO I=1, n-1
               CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, req, ierr)
               CALL MPI_REQUEST_FREE(req, ierr)
C     释放掉req，仍可以保证MPI_ISEND正常完成
               CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, req, ierr)
C     利用释放后的句柄req进行新的非阻塞通信
               CALL MPI_WAIT(req, status, ierr)
```

```
C       正常完成MPI_IRECV
            END DO
          CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, req, ierr)
          CALL MPI_WAIT(req, status)
END IF
```

程序 32 使用MPI_REQUEST_FREE的一个例子

12.7 消息到达的检查

MPI提供MPI_PROBE和MPI_IPROBE调用，允许程序员在不实际执行接收操作的情况下，检查给定的消息是否到达。程序员可以根据返回的信息决定如何接收该消息，另外，程序员可以根据被检查消息的长度分配缓冲区大小。

当非阻塞的消息到达检查函数MPI_IPROBE(source,tag,comm,flag,status)被调用时，如果存在一个消息可被接收，并且该消息的消息信封和MPI_IPROBE的消息信封<source, tag, comm>相匹配,则该调用返回flag=true，返回状态status如同在调用了MPI_RECV(..., source, tag,cmm,status)后得到的status一样，即MPI_IPROBE的status和MPI_RECV的status的返回值安全相同。若没有消息到达或到达的消息的消息信封和MPI_IPROBE的消息信封不匹配，则MPI_IPROBE调用立即返回，返回结果flag=false并且不对status定义。

如果MPI_IPROBE返回结果flag=true，则可以从返回的状态status中获取source,tag和检查消息的长度。然后就可以使用相匹配的接收语句接收该消息。

MPI_IPROBE的source参数可以是MPI_ANY_SOURCE，tag参数可以是MPI_ANY_TAG，以便用户可以检查来自不确定的源source以及不确定的标识tag。不过，必须指定一个通信域comm来指明通信的上下文。一个消息被检查后并不一定会被立即接收，同样，一个消息在被接收以前可能会被检查多次。

MPI_PROBE和MPI_IPROBE相似，只不过它是一个阻塞调用，只有找到一个匹配的消息到达之后它才会返回.

```
MPI_PROBE (source,tag,cmm,status)
      IN  source        源进程标识或任意进程标识MPI_ANY_SOURCE(整型)
      IN tag            特定tag值或任意tag值MPI_ANY_TAG（整型）
      IN  comm          通信域（句柄）
      OUT  status       返回的状态（状态类型）
int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status)
MPI_PROBE(SOURCE,TAG,COMM,STATUS,IERROR)
    INTEGER SOURCE,TAG,COMM,STATUS(MPI_STATUS_SIZE),IERROR
```

MPI调用接口 36 MPI_PROBE

```
MPI_IPROBE(source,tag,comm,flag,status)
        IN  source      源进程标识或任意进程标识MPI_ANY_SOURCE（整型）
        IN tag          特定tag值或任意tag值MPI_ANY_TAG（整型）
        IN  comm        通信域（句柄）
        OUT  flag       是否有消息到达标志（逻辑）
        OUT  status     返回的状态（状态类型）
int MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag, MPI_Status *status)
MPI_IPROBE(SOURCE,TAG,COMM,FLAG,STATUS,TERROR)
       LOGICAL  FLAG
       INTEGER  SOURCE,TAG,COMM,STATUS(MPI_STATUS_SIZE),IERROR
```

MPI调用接口 37　MPI_IPROBE

```
      CALL MPI_COMM_RANK(comm,rank,ierr)
      IF (rank .EQ. 0) THEN
          CALL MPI_SEND(i,1,MPI_INTEGER,2,0,comm,ierr)
C     进程0向进程2发送一个整型数
      ELSE IF (rank.EQ.1) THEN
          CALL MPI_SEND(x,1,MPI_REAL,2,0,comm,ierr)
C     进程1向进程2发送一个实型数
      ELSE IF (rank .EQ.2 ) THEN
          DO  i=1,2
              CALL MPI_PROBE(MPI_ANY_SOURCE,0, comm,status,ierr)
C     进程2检查消息的到达
              IF  (status(MPI_SOURCE)  =  0)  THEN
C     若到达的消息是进程0发送的，则按整型接收
                  CALL MPI_RECV(i,1,MPI_INTEGER,0,0,status,ierr)
              ELSE
C     否则（是进程1发送的）按实型接收
                  CALL  MPI_RECV(x,1,MPI_REAL,1,0,status,ierr)
              END  IF
          END  DO
      END IF
```

程序 33 使用阻塞检查等待接收消息

```
      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)=0) THEN
                        CALL  MPI_RECV(i,1,MPI_INTEGER,MPI_ANY_SOURCE
     $                                              0,status,ierr)
C                   若按任意源来接收，则可能接收到其它进程发来的消息，而不是
C                   MPI_PROBE检查到的消息
                    ELSE
                        CALL  MPI_RECV(x,1,MPI_REAL,MPI_ANY_SOURCE,
                                              0,status,ierr)
C                   若按任意源来接收，则可能接收到其它进程发来的消息，而不是
C                   MPI_PROBE检查到的消息
                    END IF
                END DO
            END  IF
```

程序 34 错误的消息接收方式

这里用MPI_ANY_SOURCE**作为两个接收调用语句的**source**参数，程序这是就出现了错误：接收操作收到的消息可能和前面调用的**MPI_PROBE**检查的消息不一致。**

12.8 非阻塞通信有序接收的语义约束

对于非阻塞通信，和阻塞通信一样，也有有序接收的语义约束，两者的含义是类似的。

进程A**向进程**B**发送的消息只能被进程**B**第一个匹配的接收语句接收，下面的接收语句即使匹配也不能超前接收消息。**

```
        CALL MPI_COMM_RANK(comm, rank, ierr)
        IF (RANK.EQ.0) THEN
            CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
C           进程0先向进程1发送数据a
            CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
C           进程0再向进程1发送数据b
        ELSE IF ( rank.EQ.1)
            CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
C           进程1一定先接收到数据a，哪怕b先到，这一接收语句也不会接收b
            CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
C           进程1然后再接收b
        END IF
        CALL MPI_WAIT(r1,status)
        CALL MPI_WAIT(r2,status)
```

C　　等待非阻塞通信的完成

程序 35 非阻塞通信的语义约束

从这一例子可以看出，不管是以前介绍的阻塞通信，还是非阻塞通信，都保持“顺序接收”的语义约束，即根据程序的书写顺序，先发送的消息一定被先匹配的接收调用接收，若在实际运行过程中后发送的消息先到达，它也只能等待。

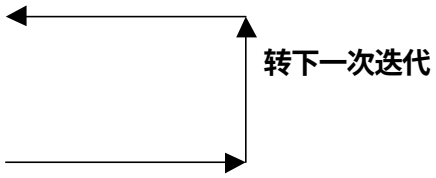
12.9 用非阻塞通信来实现Jacobi迭代

关于Jacobi迭代，前面已经介绍了许多不同的实现方法，一次比一次有所提高，这里再从提高性能的角度，用非阻塞通信，来实现Jacobi迭代中的通信与计算的重叠。

为了实现计算与通信的最大重叠，一个通用的原则就是“尽早开始通信，尽晚完成通信”，在开始通信和完成通信之间进行计算，这样通信启动得越早，完成得越晚，就有可能有更多的计算任务可以和通信重叠，也使通信可以在计算任务执行期间完成，而不需要专门的等待时间。

为此，修改Jacobi迭代过程如下：

1 计算迭代任务中下次需要通信的数据
2 启动非阻塞通信，传递这些数据
3 计算剩余的迭代部分
4 完成非阻塞通信



本程序仍然按列分成四块来进行计算。

```
      program  main
      implicit  none
      include  'mpif.h'
      integer  totalsize,mysize,steps
       parameter    (totalsize=16)
       parameter    (mysize=totalsize/4,steps=10)

     integer n, myid, numprocs, i, j,rc
      real  a(totalsize,mysize+2),b(totalsize,mysize+2)
      integer  begin_col,end_col,ierr
      integer  left,right,tag1,tag2
      integer  status(MPI_STATUS_SIZE,4)
      integer  req(4)



     call  MPI_INIT( ierr )
      call  MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
     call  MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
    print *, "Process ", myid, " of ", numprocs, " is alive"
```

C　　数组赋初值

```
do  j=1,mysize+2
    do  i=1,totalsize
        a(i,j)=0.0
    end  do
end  do
    do  i=1,totalsize
        a(i,1)=8.0
        a(i,mysize+2)=8.0
    end  do

 if (myid .eq. 0) then
    do  i=1,totalsize
        a(i,2)=8.0
    end  do
end  if
 if (myid .eq. 3) then
    do  i=1,totalsize
        a(i,mysize+1)=8.0
    end  do
end  if
do  i=1,mysize+2
     a(1,i)=8.0
     a(totalsize,i)=8.0
end  do

 tag1=3
 tag2=4

C       计算当前进程的左右相邻进程
 if (myid .gt. 0) then
    left=myid-1
 else
    left=MPI_PROC_NULL
end  if
 if (myid .lt. 3) then
    right=myid+1
 else
    right=MPI_PROC_NULL
end  if
C       设置每一块迭代的开始列和终止列
 begin_col=2
 end_col=mysize+1
 if (myid .eq. 0) then
     begin_col=3
```

```
      endif
      if (myid .eq. 3) then
         end_col=mysize
      endif
C     执行迭代
      do  n=1,steps

C     先计算需要通信的边界数据
         do  i=2,totalsize-1
            b(i,begin_col)=(a(i,begin_col+1)+a(i,begin_col-1)+
     *                      a(i+1,begin_col)+a(i-1,begin_col))*0.25
            b(i,end_col)=(a(i,end_col+1)+a(i,end_col-1)+
     *                      a(i+1,end_col)+a(i-1,end_col))*0.25
         end  do

C     执行非阻塞通信，将下一次计算需要的数据首先进行通信
         call  MPI_ISEND(b(1,end_col),totalsize,MPI_REAL,right,tag1,
     *                   MPI_COMM_WORLD,req(1),ierr)
         call  MPI_ISEND(b(1,begin_col),totalsize,MPI_REAL,left,tag2,
     *                   MPI_COMM_WORLD,req(2),ierr)

         call  MPI_IRECV(a(1,1),totalsize,MPI_REAL,left,tag1,
     *                   MPI_COMM_WORLD,req(3),ierr)
         call  MPI_IRECV(a(1,mysize+2),totalsize,MPI_REAL,right,tag2,
     *                   MPI_COMM_WORLD,req(4),ierr)
C     计算剩余的部分
      do  j=begin_col+1,end_col-1
         do  i=2,totalsize-1
            b(i,j)=(a(i,j+1)+a(i,j-1)+a(i+1,j)+a(i-1,j))*0.25
         end  do
      end  do
C     更新数组
      do  j=begin_col,end_col
         do  i=2,totalsize-1
            a(i,j)=b(i,j)
         end  do
      end  do
C     完成非阻塞通信
      do  i=1,4
         CALL MPI_WAIT(req(i),status(1,i),ierr)
      end  do
      end  do
      do  i=2,totalsize-1
         print  *,  myid,(a(i,j),j=begin_col,end_col)
```

```
C       打印迭代后的结果
        end do
        call MPI_Finalize(rc)
        end
```

程序 36 非阻塞通信实现的Jacobi迭代

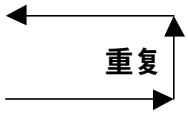
12.10 重复非阻塞通信

如果一个通信会被重复执行，比如循环结构内的通信调用，MPI提供了特殊的实现方式，对这样的通信进行优化，以降低不必要的通信开销，它将通信参数和MPI的内部对象建立固定的联系，然后通过该对象完成重复通信的任务。这样的通信方式在MPI中都是非阻塞通信。

重复非阻塞通信需要如下步骤：

1 **通信的初始化，比如**MPI_SEND_INIT

2 **启动通信**，MPI_START



3 **完成通信**，MPI_WAIT

4 **释放查询对象**，MPI_REQUEST_FREE

注意重复通信时，通信的初始化操作并没有启动消息通信，消息真正开始通信是由MPI_START**触发的，消息的完成操作并不释放相应的非阻塞通信对象，只是将其状态置为非活动状态，若下面进行重复通信，则再由**MPI_START**将该对象置为活动状态，并启动通信。当不需要再进行通信时，必须通过显式的语句**MPI_REQUEST_FREE**将非阻塞通信对象释放掉，这是重复通信和一般的非阻塞通信不同的地方。**

根据通信模式的不同，重复非阻塞通信也有四种不同的形式，即标准模式、同步模式、缓存模式和就绪模式。

```
MPI_SEND_INIT(buf,count,datatype,dest,tag,comm,request)
      IN  buf          发送缓冲区起始地址(可选数据类型)
      IN  count        发送数据个数(整型)
      IN  datatype     发送数据的数据类型(句柄)
      IN  dest         目标进程标识(整型)
      IN  tag          消息标识(整型)
      IN  comm         通信域(句柄)
      OUT request      非阻塞通信对象(句柄)
int MPI_Send_init(void* buf, int count, MPI_Data type,int dest, int tag,
      MPI_Comm comm, MPI_Request *request)
MPI_SEND_INIT(BUF,COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,
      IERRR)
      <type> BUF (*)
      INTEGER   COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,IERRROR
```

MPI调用接口 38 MPI_SEND_INIT

MPI_SEND_INIT**创建一个标准模式重复非阻塞发送对象，该对象和相应的发送操作的所有参数捆绑到一起。**

```
MPI_BSEND_INIT(buf,count,datatype,dest,tag,comm,request)
    IN  buf        发送缓冲区初始地址(可选数据类型)
    IN  count      发送数据个数(整型)
    IN  datatype   发送数据的数据类型(句柄)
    IN  dest       目标进程标识(整型)
    IN  tag        消息标识(整型)
    IN  comm       通信域(句柄)
    OUT request    非阻塞通信完成(句柄)
int MPI_Bsend_init(void* buf,int count,MPI_Datatype datatype,int dest, int tag,
    MPI_Comm comm,MPI_Request *request)
MPI_BSEND_INIT(BUF,COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,IERROR)
    <type> BUF (*)
    INTEGER,COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,IERROR
```

MPI**调用接口** 39　MPI_BSEND_INIT

MPI_BSEND_INIT**创建一个缓冲模式重复非阻塞发送对象，该对象和相应的发送操作的所有参数捆绑到一起。**

```
MPI_SSEND_INIT(buf,count,datatype,dest,tag,comm,request)
    IN  buf        发送缓冲区初始地址(可选数据类型)
    IN  count      发送数据的个数(整型)
    IN  datatype   发送数据的数据类型(句柄)
    IN  dest       目标进程标识(整型)
    IN  tag        消息标识(整型)
    IN  comm       通信域(句柄)
    OUT request    非阻塞通信对象(句柄)
int MPI_Ssend_init(void* buf,int count,MPI_Datatype datatype,int dest, int tag,
    MPI_Comm comm,MPI_Request *request)
MPI_SSEND_INIT(BUF,COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,IERROR)
    <type> BUF (*)
    INTEGER COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,IERROR
```

MPI**调用接口** 40　MPI_SSEND_INIT

MPI_SSEND_INIT**创建一个同步模式非阻塞重复发送对象，该对象和相应的发送操作的所有参数捆绑到一起。**

```
MPI_RSEND_INIT(buf,count,datatype,dest,tag,comm,request)
        IN buf          发送缓冲区初始地址(可选数据类型)
        IN count        发送数据的个数(整型)
        IN datatype     发送数据的数据类型(句柄)
        IN dest         目标进程标识(整型)
        IN tag          消息标识(整型)
        IN  comm        通信域(句柄)
        OUT  request    非阻塞通信对象(句柄)
int MPI_Rsend_init(void* buf,int count,MPI_Datatype datatype,int dest, int tag,
      MPI_Comm comm,MPI_Request *request)
MPI_RSEND_INIT(BUF,COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,
      IERROR)
      <type> BUF (*)
       INTEGER  COUNT,DATATYPE,DEST,TAG,COMM,REQUEST,IERROR
```

MPI调用接口 41 MPI_RSEND_INIT

MPI_RSEND_INIT创建一个接收就绪模式非阻塞重复发送对象，该对象和相应的发送操作的所有参数捆绑到一起。

```
MPI_RECV_INIT(buf,count,datatype,source,tag,comm,request)
        OUT  buf        接收缓冲区初始地址(可选数据类型)
        IN count        接收数据的最大个数(整型)
        IN datatype     接收数据的数据类型(句柄)
        IN  source      发送进程的标识或任意进程MPI_ANY_SOURCE(整型)
        IN tag          消息标识或任意标识MPI_ANY_TAG(整型)
        IN  comm        通信域(句柄)
        OUT  request    非阻塞通信对象(句柄)
int MPI_Recv_init(void* buf,int count,MPI_Datatype datatype,int source, int tag,
      MPI_Comm comm,MPI_Request *request)
MPI_RECV_INIT(BUF,COUNT,DATATYPE,SOURCE,TAG,COMM,REQUEST,
      IERROR)
      <type> BUF (*)
      INTEGER COUNT,DATATYPE,SOURCE,TAG,COMM,REQUEST,IERROR
```

MPI调用接口 42 MPI_RECV_INIT

MPI_RECV_INIT创建一个标准模式非阻塞重复接收对象，该对象和相应的接收操作的所有参数捆绑到一起。参数buf标为OUT是因为用户通过传递该参数给MPI_RECV_INIT给予接收缓冲区写权限。

一个重复非阻塞通信在创建后处于非活动状态，没有活动的通信附在该对象中。一个使用重复非阻塞通信的对象(发送或接收)使用MPI_START将其激活。

```
MPI_START(request)
        INOUT request          非阻塞通信对象(句柄)
int MPI_Start(MPI_Request *request)
MPI_START(REQUEST,IERROR)
        INTEGER  REQUEST ,IERROR
```

MPI调用接口 43　MPI_START

request参数是一个由前面介绍的初始化非阻塞重复调用返回的句柄。MPI_START调用之前，该对象处于非激活状态，一旦使用该调用，该调用成为激活状态。如果这是一个接收就绪模式发送请求，则在该调用之前应该先有一个匹配的接收操作被启动。通信缓冲区在该调用后应该被禁止访问，直到操作完成。

一个用MPI_SEND_INIT创建的非阻塞重复通信对象来调用MPI_START产生的通信和用MPI_ISEND调用产生的通信效果一样；一个用MPI_BSEND_INIT创建的非阻塞重复通信对象来调用MPI_START产生的通信和直接调用MPI_IBSEND产生的通信效果相同，其它的也一样。

```
MPI_STARTALL(count,array_of_requests)
        IN count                开始非阻塞通信对象的个数 (整型)
        IN array_of_requests    非阻塞通信对象数组(句柄队列)
int MPI_Startall(int count, MPI_Request *array_of_requests)
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS,IERROR)
      INTEGER  COUNT,  ARRAY_OF_REQUESTS(*),IERROR
```

MPI调用接口 44　MPI_STARTALL

一个MPI_STARTALL调用等价于对array_of_request表中的每一个非阻塞重复通信对象用MPI_START调用。一个由MPI_START或MPI_STARTALL调用开始的通信如同前面介绍的各种非阻塞通信一样，由MPI_WAIT或MPI_TEST调用来完成。这些调用的成功完成将使非阻塞通信对象处于非激活状态，但是该对象并未被释放，它可以被MPI_START 或MPI_STARTALL重新激活。一个重复非阻塞通信对象可以用MPI_REQUEST_FREE来释放，MPI_REQUEST_FREE可以在重复非阻塞通信被创建以后的任何地方被调用，可是,只有当该对象成为非活动状态时才可以被取消。

一个用MPI_START初始化的发送操作可以被任何接收操作匹配,类似地,一个用MPI_START初始化的接收操作可以接收任何发送操作产生的消息.

12.11 用重复非阻塞通信来实现Jacobi迭代

由于在Jacobi迭代中，通信放在循环之内，要重复进行，因此可以用重复非阻塞通信来实现它。

```
program  main
implicit  none
include  'mpif.h'
```

```
      integer  totalsize,mysize,steps
      parameter    (totalsize=16)
      parameter    (mysize=totalsize/4,steps=10)

     integer n, myid, numprocs, i, j,rc
      real  a(totalsize,mysize+2),b(totalsize,mysize+2)
      integer  begin_col,end_col,ierr
      integer  left,right,tag1,tag2
      integer  status(MPI_STATUS_SIZE,4)
      integer  req(4)


      call  MPI_INIT( ierr )
      call  MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
     call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
    print *, "Process ", myid, " of ", numprocs, " is alive"
C       数组初始化
      do  j=1,mysize+2
          do  i=1,totalsize
              a(i,j)=0.0
          end  do
      end  do
          do  i=1,totalsize
              a(i,1)=8.0
              a(i,mysize+2)=8.0
          end  do
C       边界元素赋初值
       if (myid .eq. 0) then
          do  i=1,totalsize
              a(i,2)=8.0
          end  do
      end  if
       if (myid .eq. 3) then
          do  i=1,totalsize
              a(i,mysize+1)=8.0
          end  do
      end  if
      do  i=1,mysize+2
           a(1,i)=8.0
           a(totalsize,i)=8.0
      end  do

       tag1=3
       tag2=4
```

```
C       设置当前进程左右两侧的进程
        if (myid .gt. 0) then
           left=myid-1
        else
           left=MPI_PROC_NULL
        end  if
        if (myid .lt. 3) then
           right=myid+1
        else
           right=MPI_PROC_NULL
        end  if

C       设置迭代的开始和终止列
        begin_col=2
        end_col=mysize+1
        if (myid .eq. 0) then
            begin_col=3
        endif
        if (myid .eq. 3) then
            end_col=mysize
        endif

C       初始化重复非阻塞通信
        call  MPI_SEND_INIT(b(1,end_col),totalsize,MPI_REAL,right,tag1,
       *                    MPI_COMM_WORLD,req(1),ierr)
        call  MPI_SEND_INIT(b(1,begin_col),totalsize,MPI_REAL,left,tag2,
       *                    MPI_COMM_WORLD,req(2),ierr)

        call  MPI_RECV_INIT(a(1,1),totalsize,MPI_REAL,left,tag1,
       *                    MPI_COMM_WORLD,req(3),ierr)
        call  MPI_RECV_INIT(a(1,mysize+2),totalsize,MPI_REAL,right,tag2,
       *                    MPI_COMM_WORLD,req(4),ierr)
C       执行迭代
        do  n=1,steps
C       先计算需要通信的部分
           do  i=2,totalsize-1
              b(i,begin_col)=(a(i,begin_col+1)+a(i,begin_col-1)+
       *                      a(i+1,begin_col)+a(i-1,begin_col))*0.25
              b(i,end_col)=(a(i,end_col+1)+a(i,end_col-1)+
       *                      a(i+1,end_col)+a(i-1,end_col))*0.25
           end  do

C          激活非阻塞通信对象，启动4个非阻塞通信
```

```
      call  MPI_STARTALL(4,req,ierr)
C     计算剩余的迭代部分
      do  j=begin_col+1,end_col-1
        do  i=2,totalsize-1
          b(i,j)=(a(i,j+1)+a(i,j-1)+a(i+1,j)+a(i-1,j))*0.25
        end  do
      end  do
      do  j=begin_col,end_col
        do  i=2,totalsize-1
          a(i,j)=b(i,j)
        end  do
      end  do

C     完成非阻塞通信，非阻塞通信对象变为非活动态
      call  MPI_WAITALL(4,req,status,ierr)

      end  do
      do  i=2,totalsize-1
        print  *,  myid,(a(i,j),j=begin_col,end_col)
      end  do

C     释放非阻塞通信对象
      do  i=1,4
        CALL MPI_REQUEST_FREE(req(i),ierr)
      end  do
      call  MPI_FINALIZE(rc)
      end
```

程序 37 用重复非阻塞通信实现Jacobi迭代

12.12 小结

非阻塞通信是一种重要的提高MPI程序整体性能的手段，特别是当特定的硬件能够对非阻塞通信直接支持时更是这样。

第13章 组通信MPI程序设计

MPI组通信和点到点通信的一个重要区别，就在于它需要一个特定组内的所有进程同时参加通信，而不是象点到点通信那样只涉及到发送方和接收方两个进程。组通信在各个不同进程的调用形式完全相同，而不象点到点通信那样在形式上就有发送和接收的区别。本章主要介绍如何使用MPI提供的各种组通信功能，方便编程，提高程序的可读性和移植性，提高通信效率。

13.1 组通信概述

组通信由哪些进程参加，以及组通信的上下文，都是由该组通信调用的通信域限定的。组通信调用可以和点对点通信共用一个通信域，MPI保证由组通信调用产生的消息不会和点对点调用产生的消息相混淆。在组通信中不需要通信消息标志参数，如果将来的MPI新版本定义了非阻塞的组通信函数，也许那时就需要引入消息标志来防止组通信彼此之间造成的混淆。

组通信一般实现三个功能：通信、同步和计算。通信功能主要完成组内数据的传输，而同步功能实现组内所有进程在特定的地点在执行进度上取得一致，计算功能稍微复杂一点，要对给定的数据完成一定的操作。

13.1.1 组通信的消息通信功能

对于组通信，按通信的方向的不同，又可以分为以下三种：一对多通信，多对一通信和多对多通信。

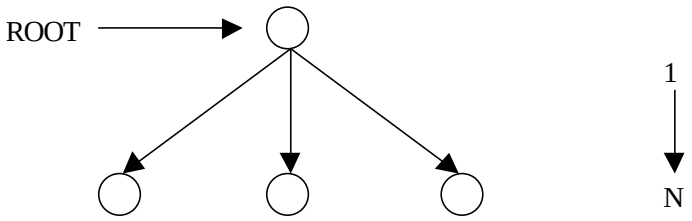


图 50 一对多通信

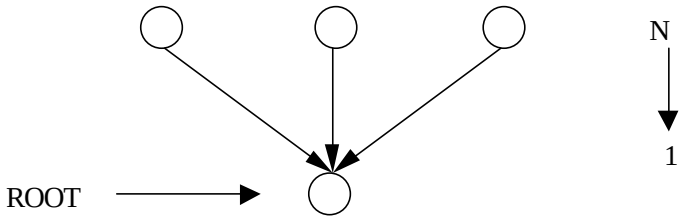


图 51 多对一通信

如图 50所示的一对多通信，其中一个进程向其它所有的进程发送消息，一般地，把这样的进程称为ROOT，在一对多的组通信中，调用的某些参数只对ROOT有意义，对其它的进程只是满足语法的要求。广播是最常见的一对多通信的例子。同样对于多对一通信，一个进程从其它所有的进程接收消息，这样的进程也称为ROOT，收集是最常见的多对一通信的例子。

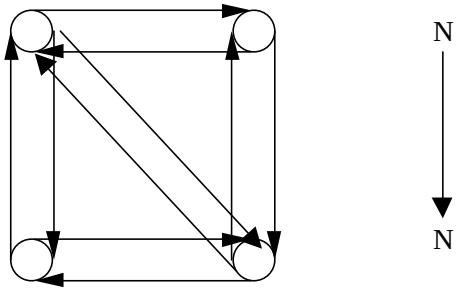


图 52 多对多通信

图 52所示的是多对多通信，其中每一个进程都向其它所有的进程发送消息，或者每个进程都从其它所有的进程接收消息，或者每个进程都同时向所有其它的进程发送和从其它所有的进程接收消息。

一个进程完成了它自身的组通信调用返回后，就可以释放数据缓冲区或使用缓冲区中的数据，但是一个进程组通信的完成并不表示其它所有进程的组通信都已完成，即组通信并不一定意味着同步的发生（当然同步组通信调用除外）。

13.1.2 组通信的同步功能

同步是许多应用中必须提供的功能，组通信的还提供专门的调用以完成各个进程之间的同步，从而协调各个进程的进度和步伐。

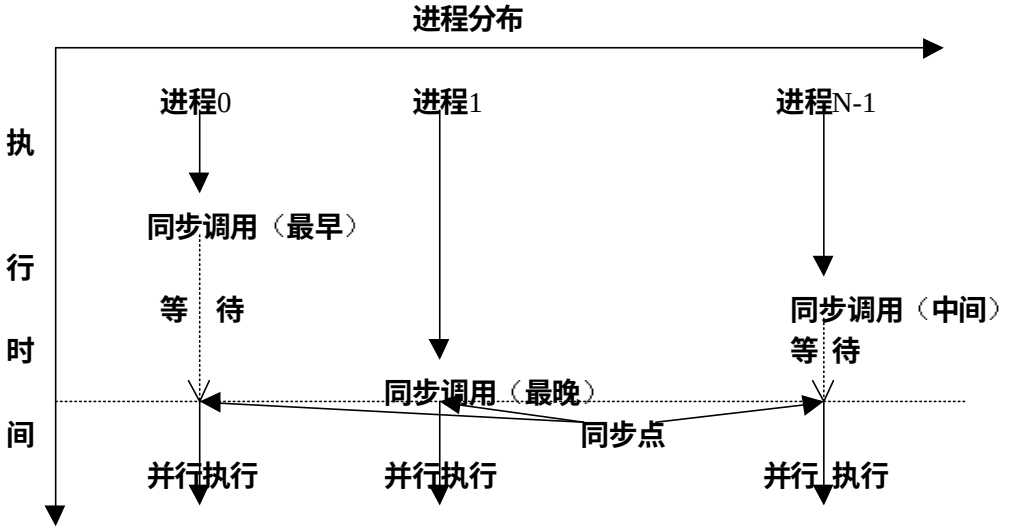


图 53 MPI同步调用

如图 53所示，所有的进程并行执行，但是，不同的进程执行的进度是不同的，在本例

中，进程0首先执行到同步调用，执行同步操作，但是，由于其它的进程还没有到达同步调用点，因此进程0只好等待；接下来是其它的进程（如进程N-1）陆续到达同步调用点，但是，只要有一个进程未到达同步调用点，则所有其它已到达同步调用点的进程都必须得等待；当最后到达同步调用点的进程--进程1到达同步调用点后，它也执行了同步调用操作，这时，由于所有的进程都执行了这一操作，因此，它们此时才可以从同步调用返回，继续并行执行下面的操作。

同步的作用是当进程完成同步调用后，可以保证所有的进程都已执行了同步点前面的操作。

13.1.3 组通信的计算功能

组通信除了能够完成通信和同步的功能外，还可以进行计算，完成计算的功能。从效果上，可以认为，MPI组通信的计算功能是分三步实现的，首先是通信的功能，即消息根据要求发送到目的进程，目的进程也已经接收到了各自所需要的消息，然后是对消息的处理，即计算部分，MPI组通信有计算功能的调用都指定了计算操作，用给定的计算操作对接收到的数据进行处理；最后一步是将处理结果放入指定的接收缓冲区。

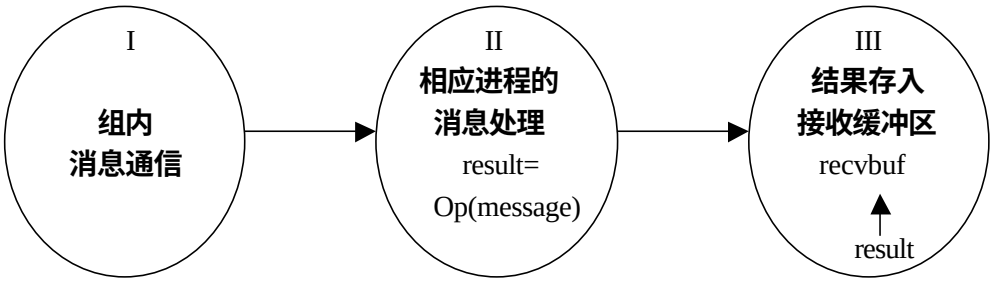


图 54　MPI组通信的计算功能

13.2 广播

MPI_BCAST(buffer,count,datatype,root,comm)

IN/OUT buffer 通信消息缓冲区的起始地址(可选数据类型)

IN count 将广播出去/或接收的数据个数(整型)

IN datatype 广播/接收数据的数据类型(句柄)

IN root 广播数据的根进程的标识号(整型)

IN comm 通信域(句柄)

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

MPI_BCAST(BUFFER,COUNT,DATATYPE,ROOT,COMM,IERROR)

<type> BUFFER(*)

INTEGER COUNT,DATATYPE,ROOT,COMM,IERROR

MPI调用接口 45 MPI_BCAST

MPI_BCAST是一对多组通信的典型例子，它完成从一个标识为root的进程将一条消息广播发送到组内的所有其它的进程，同时也包括它本身在内。在执行该调用时组内所有进程（不管是root进程本身还是其它的进程）都使用同一个通信域comm和根标识root，其执行结果是将根进程通信消息缓冲区中的消息拷贝到其他所有进程中去。

一般说来，数据类型datatype可以是预定义数据类型或派生数据类型，其它进程中指定的通信元素个数count、数据类型datatype必须和根进程中的指定的通信元素个数count、数据类型datatype保持一致。即对于广播操作调用，不管是广播消息的根进程，还是从根接收消息的其它进程，在调用形式上完全一致，即指明相同的根，相同的元素个数以及相同的数据类型。除MPI_BCAST之外，其它完成通信功能的组通信调用都有此限制。

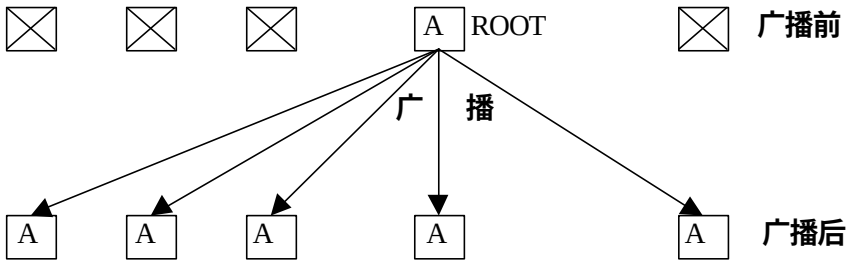


图 55 广播前后各进程缓冲区中数据的变化

下面的程序首先由ROOT进程从键盘读如一个整数，然后广播到其它所有的进程，各进程打印出收到的数据。若该数据非负，则循环执行上面的步骤。

```
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
```

```
int argc;
char **argv;
{
    int  rank, value;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
     do {
     if (rank == 0) /*进程0读入需要广播的数据*/
    scanf( "%d", &value );

     MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );/*将该数据广播出去*/

     printf( "Process %d got %d\n", rank, value );/*各进程打印收到的数据*/
     } while (value >= 0);

    MPI_Finalize( );
    return  0;
}
```

程序 38　广播程序示例

13.3 收集

收集MPI_GATHER**是典型的多对一通信的例子。在收集调用中，每个进程（包括根进程本身）将其发送缓冲区中的消息发送到根进程，根进程根据发送进程的进程标识的序号即进程的**rank**值，将它们各自的消息依次存放到自己的消息缓冲区中。和广播调用不同的是，广播出去的数据都是相同的，但对于收集操作，虽然从各个进程收集到的数据的个数必须相同，但从各个进程收集到的数据一般是互不相同的。其结果就象一个进程组中的**N**个进程（包括根进程在内）都执行了一个发送调用，同时根进程执行了**N**次接收调用。

对于所有非根进程，接收消息缓冲区被忽略，但是各个进程必须提供这一参数。

收集调用每个进程的发送数据个数**sendcount**和发送数据类型**sendtype**都是相同的，都和根进程中接收数据个数**recvcount**和接收数据类型**recvtype**相同。注意根进程中指定的接收数据个数是指从每一个进程接收到的数据的个数，而不是总的接收个数。

此调用中的所有参数对根进程来说都是有意义的,而对于其它进程只有sendbuf、sendcount、sendtype、root和comm是有意义的。其它的参数虽然没有意义，但是却不能省略。root和comm在所有进程中都必须是一致的。

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

IN sendbuf 发送消息缓冲区的起始地址(可选数据类型)

IN sendcount 发送消息缓冲区中的数据个数(整型)

IN sendtype 发送消息缓冲区中的数据类型(句柄)

OUT recvbuf 接收消息缓冲区的起始地址(可选数据类型)

IN recvcount 待接收的元素个数(整型,仅对于根进程有意义)

IN recvtype 接收元素的数据类型(句柄,仅对于根进程有意义)

IN root 接收进程的序列号(整型)

IN comm 通信域(句柄)

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

MPI调用接口 46 MPI_GATHER

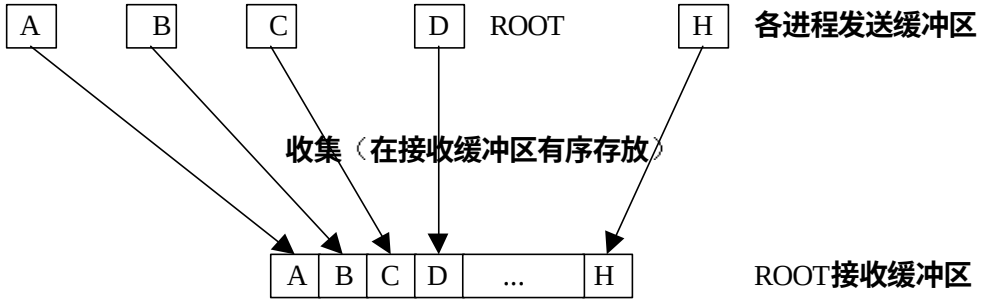


图 56 数据收集

MPI_GATHERV和MPI_GATHER的功能类似，也完成数据收集的功能，但是它可以从不同的进程接收不同数量的数据。为此，接收数据元素的个数recvcounts是一个数组，用于指明从不同的进程接收的数据元素的个数。根从每一个进程接收的数据元素的个数可以不同，但是发送和接收的个数必须一致。除此之外，它还为每一个接收消息在接收缓冲区的位置提供了一个位置偏移displs数组，用户可以将接收的数据存放到根进程消息缓冲区的任意位置。也就是说，MPI_GATHERV明确指出了从不同的进程接收数据元素的个数以及这些数据在ROOT的接收缓冲区存放的起始位置，这是它相对于MPI_GATHER灵活的地方，也是比MPI_GATHER复杂的地方。

此调用中的所有参数对根进程来说都是有意义的，而对于其它的进程只有sendbuf、sendcount、sendtype、root和comm是有意义的。参数root和comm在所有进程中都必须是一致的。

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,recvtype,
root, comm)

IN	sendbuf	发送消息缓冲区的起始地址(可选数据类型)
IN	sendcount	发送消息缓冲区中的数据个数(整型)
IN	sendtype	发送消息缓冲区中的数据类型(句柄)
OUT	recvbuf	接收消息缓冲区的起始地址(可选数据类型,仅对于根进程有意义)
IN	recvcounts	整型数组(长度为组的大小), 其值为从每个进程接收的数据个数
IN	displs	整数数组,每个入口表示相对于recvbuf的位移
IN	recvtype	接收消息缓冲区中数据类型 (句柄)
IN	root	接收进程的标识号(句柄)
IN	comm	通信域(句柄)

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf,
        int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
        DISPLS,  RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE,
            ROOT, COMM, IERROR
```

MPI调用接口 47 MPI_GATHERV

下面的程序片段实现从进程组中的每个进程收集100个整型数送给根进程。

```
MPI_Comm  comm;
int  gsize,sendarray[100];
int  root,*rbuf;
.....
MPI_Comm_size(comm,&gsize);
rbuf=(int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray,100,MPI_INT,rbuf,100,MPI_INT,root,comm);
```

程序 39 MPI_Gather使用示例

下面的程序实现每个进程向根进程发送100个整型数,但在接收端设置(100个数据)步长,用MPI_GATHERV调用和displs参数来实现，假设步长≥100。

```
MPI_Comm  comm;
int  gsize, sendarray[100];
int root, *rbuf, stride;
int *displs, i, *rcounts;
.....
MPI_Comm_size(comm, &gsize);
```

```
 rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
      displs[i] = i*stride;
      rcounts[i] = 100;
 }
 MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                  root, comm);
```

程序 40 MPI_Gatherv使用示例

13.4 散发

MPI_SCATTER**是一对多的组通信调用，但是和广播不同，**ROOT**向各个进程发送的数据可以是不同的。**MPI_SCATTER**和**MPI_GATHER**的效果正好相反，两者互为逆操作。**

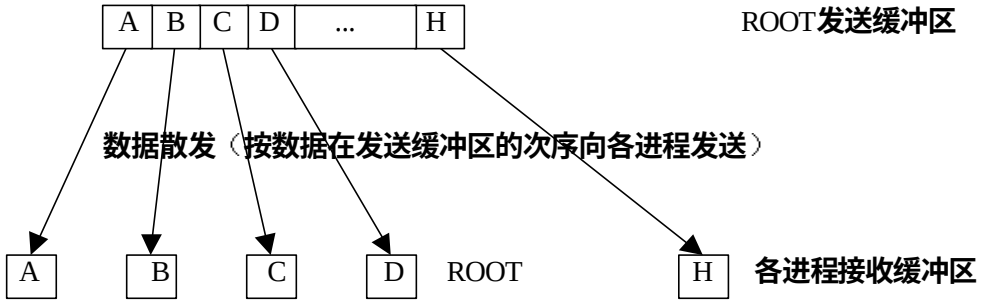


图 57 数据散发

对于所有非根进程，发送消息缓冲区被忽略。根进程中的发送数据元素个数sendcount**和发送数据类型**sendtype**必须和所有进程的接收数据元素个数**recvcount**和接收数据类型**recvtype**相同。根进程发送元素个数指的是发送给每一个进程的数据元素的个数，而不是总的数据个数。这就意味着在每个进程和根进程之间，发送的数据个数必须和接收的数据个数相等。**

此调用中的所有参数对根进程来说都是有意义的，而对于其他进程来说，只有recvbuf、recvcount、recvtype、root**和**comm**是有意义的。参数**root**和**comm**在所有进程中都必须是一致的。**

MPI_GATHER**有一个更灵活的形式**MPI_GATHERV，MPI_SCATTER**也有一个更灵活的形式**MPI_SCATTERV。

正如MPI_SCATTER**是**MPI_GATHER**的逆操作一样，**MPI_SCATTERV**是**MPI_GATHERV**的逆操作。**

MPI_SCATTERV**对**MPI_SCATTER**的功能进行了扩展，它允许**ROOT**向各个进程发送个数不等的数据，因此要求**sendcounts**是一个数组。同时还提供一个新的参数**displs**，指明根进程发往其它不同进程数据在根发送缓冲区中的偏移位置。对于所有非根进程,发送消息缓冲区被忽略。根进程中**sendcount[i]**和**sendtype**的类型必须和进程**i**的**recvcount**和**recvtype**的类型相同，这就意谓着在每个进程和根进程之间，发送的数据量必须和接收的数据量相等。**

此调用中的所有参数对根进程来说都是很重要的，而对于其他进程来说只有recvbuf、

recvcount、recvtype、root**和**comm**是必不可少的。参数**root**和**comm**在所有进程中都必须是一致的。**

MPI_SCATTER(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,
root,comm)

IN sendbuf **发送消息缓冲区的起始地址(可选数据类型)**
IN sendcount **发送到各个进程的数据个数(整型)**
IN sendtype **发送消息缓冲区中的数据类型(句柄)**
OUT recvbuf **接收消息缓冲区的起始地址(可选数据类型)**
IN recvcount **待接收的元素个数(整型)**
IN recvtype **接收元素的数据类型(句柄)**
IN root **发送进程的序列号(整型)**
IN comm **通信域(句柄)**

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

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

MPI**调用接口** 48 MPI_SCATTER

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN sendbuf **发送消息缓冲区的起始地址(可选数据类型)**
IN sendcounts **发送数据的个数，整数数组 (整型)**
IN displs **发送数据偏移，整数数组(整型)**
IN sendtype **发送消息缓冲区中元素类型(句柄)**
OUT recvbuf **接收消息缓冲区的起始地址(可变)**
IN recvcount **接收消息缓冲区中数据的个数(整型)**
IN recvtype **接收消息缓冲区中元素的类型(句柄)**
IN root **发送进程的标识号(句柄)**
IN comm **通信域(句柄)**

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

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

MPI**调用接口** 49 MPI_SCATTERV

下面的程序片段实现根进程将向组内的每个进程分散100个整型数据。

```
MPI_Comm comm;
int gsize,*sendbuf;
int root,rbuf[100];
......
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
......
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

程序 41　MPI_Scatter应用示例

下面的程序片段实现根进程将向组内的每个进程分散100个整型数据，但这每100个数据的集合在根进程的发送消息缓冲区中相隔一定的步长。

```
MPI_Comm comm;
int gsize,*sendbuf;
int root,rbuf[100],i,*displs,*scounts;
......
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
......
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i)  {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100,
                MPI_INT, root, comm);
```

程序 42　MPI_Scatterv应用示例

13.5 组收集

MPI_GATHER是将数据收集到ROOT进程，而MPI_ALLGATHER相当于每一个进程都作为ROOT执行了一次MPI_GATHER调用，即每一个进程都收集到了其它所有进程的数据。从参数上看，MPI_ALLGATHER和MPI_GATHER完全相同，只不过在执行效果上，对于MPI_GATHER执行结束后，只有ROOT进程的接收缓冲区有意义，MPI_ALLGATHER调用结束后所有进程的接收缓冲区都有意义，它们接收缓冲区的内容是相同的。

由MPI_ALLGATHER和MPI_GATHER的关系，不难得知MPI_ALLGATHERV和MPI_GATHERV的关系。MPI_ALLGATHERV也是所有的进程都将接收结果，而不是只有根进程接收结果。从每个进程发送的第j块数据将被每个进程接收，然后存放在各个进程接收消息缓冲区recvbuf的第j块。进程j的sendcount和sendtype的类型必须和其他所有进程的

recvcounts[j]**和**recvtype**相同**。

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,comm)

IN sendbuf **发送消息缓冲区的起始地址(可选数据类型)**

IN sendcount **发送消息缓冲区中的数据个数(整型)**

IN sendtype **发送消息缓冲区中的数据类型(句柄)**

OUT recvbuf **接收消息缓冲区的起始地址(可选数据类型)**

IN recvcount **从其它进程中接收的数据个数(整型)**

IN recvtype **接收消息缓冲区的数据类型(句柄)**

IN comm **通信域(句柄)**

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                  void* recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm  comm)
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
              RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM,
            IERROR
```

MPI**调用接口** 50 MPI_ALLGATHER

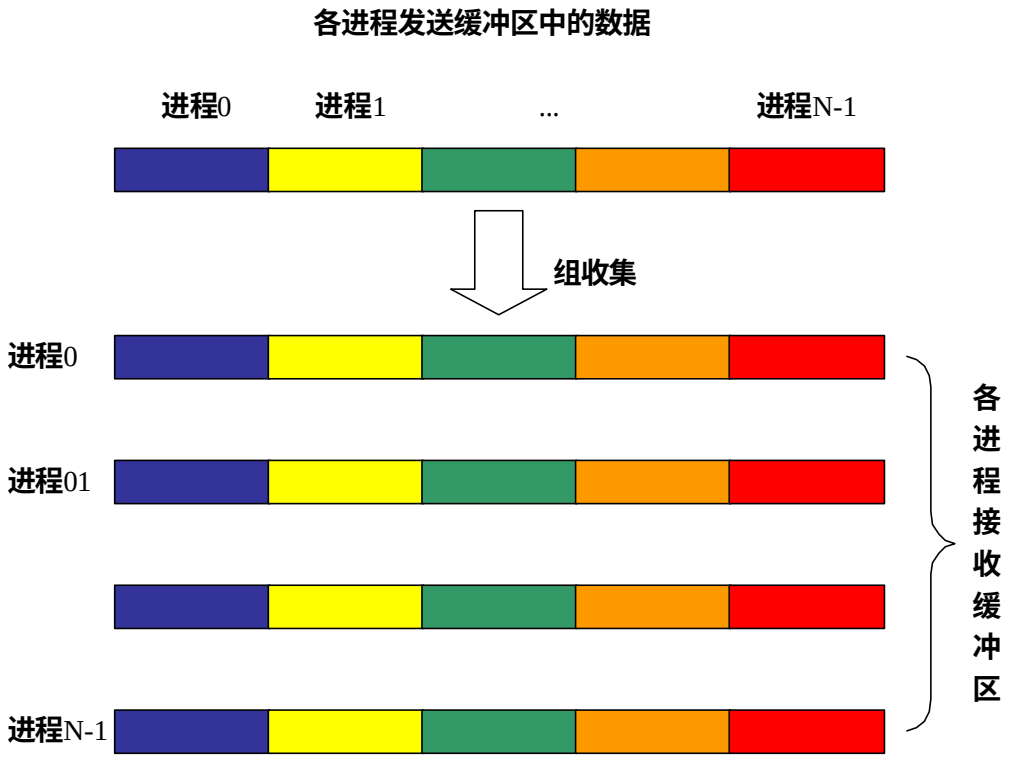


图 58 组收集

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN sendbuf 发送消息缓冲区的起始地址(可选数据类型)
IN sendcount 发送消息缓冲区中的数据个数(整型)
IN sendtype 发送消息缓冲区中的数据类型(句柄)
OUT recvbuf 接收消息缓冲区的起始地址(可选数据类型)
IN recvcounts 接收数据的个数，整型数组(整型)
IN displs 接收数据的偏移，整数数组(整型)
IN recvtype 接收消息缓冲区的数据类型(句柄)
IN comm 通信域(句柄)

```
int MPI_Allgatherv(void* sendbuf, int sendcount,MPI_Datatype sendtype,
                   void* recvbuf, int *recvcounts, int *displs,
                    MPI_Datatype recvtype, MPI_Comm comm)
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
               RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)
    <type>  SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*),
            RECVTYPE,  COMM, IERROR
```

MPI调用接口 51 MPI_ALLGATHERV

下面的程序片段实现组内每个进程都从其它进程收集100个数据，存入各自的接收缓冲区。

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
.....
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

程序 43 MPI_Allgather应用示例

用MPI_Allgatherv来实现

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf, stride;
int *displs, i, *rcounts;
.....
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
```

```
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
      displs[i] = i*stride;
      rcounts[i] = 100;
 }
 MPI_Allgatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                root, comm);
```

程序 44　MPI_Allgatherv应用示例

13.6 全互换

MPI_ALLTOALL**是组内进程之间完全的消息交换，其中每一个进程都相其它所有的进程发送消息，同时，每一个进程都从其它所有的进程接收消息。**

MPI_ALLGATHER**每个进程散发一个相同的消息给所有的进程，**MPI_ALLTOALL**散发给不同进程的消息是不同的，因此它的发送缓冲区也是一个数组。**MPI_ALLTOALL**的每个进程可以向每个接收者发送数目不同的数据。第**i**个进程发送的第**j**块数据将被第**j**个进程接收并存放在其接收消息缓冲区**recvbuf**的第**i**块。每个进程的**sendcount**和**sendtype**的类型必须和所有其他进程的**recvcount**和**recvtype**相同，这就意谓着在每个进程和根进程之间，发送的数据量必须和接收的数据量相等。**

调用MPI_ALLTOALL**相当于每个进程依次将它的发送缓冲区的第**i**块数据发送给第**i**个进程。同时每个进程又都依次从第**j**个进程接收数据放到各自接收缓冲区的第**j**块数据区的位置。**

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                recvtype, comm)
 IN   sendbuf      发送消息缓冲区的起始地址(可选数据类型)
 IN   sendcount    发送到每个进程的数据个数(整型)
 IN   sendtype     发送消息缓冲区中的数据类型(句柄)
 OUT  recvbuf      接收消息缓冲区的起始地址(可选数据类型)
 IN   recvcount    从每个进程中接收的元素个数(整型)
 IN   recvtype     接收消息缓冲区的数据类型(句柄)
 IN   comm         通信域(句柄)
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM,
            IERROR
```

MPI调用接口 52　MPI_ALLTOALL

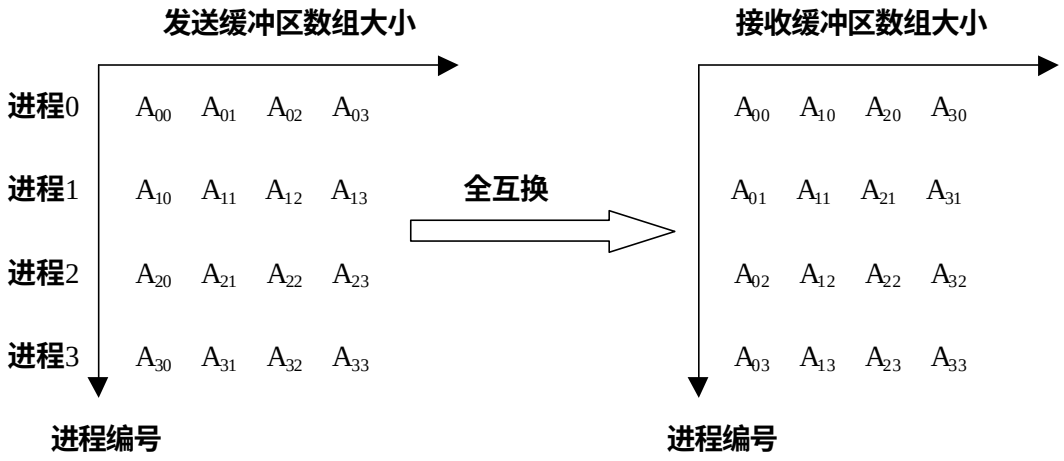


图 59 MPI_ALLTOALL全互换

从图中可以看出，在互换之前依次将各进程的发送缓冲区组织在一起，互换之后依次将各进程的接收缓冲区组织在一起，则接收缓冲区组成的矩阵是发送缓冲区组成的矩阵的转置（若每次向一个进程发送的数据是多个，则将这多个数据看作是一个数据单元）。

下面的程序使用MPI_ALLTOALL**调用。它在调用之间，每一个进程将所有发往其它不同进程的数据打印，调用结束后，再将所有从其它进程接收的数据打印，从中可以看出发送和接收的对应关系。**

```
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
int main( argc, argv )
int argc;
char *argv[];
{
    int  rank, size;
   int  chunk  =  2;
    /*发送到一个进程的数据块的大小*/
    int  i,j;
    int  *sb;
    int  *rb;
   int  status,  gstatus;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

   sb  =  (int  *)malloc(size*chunk*sizeof(int));/*申请发送缓冲区*/
   if  (  !sb  )  {
     perror( "can't allocate send buffer" );
```

```
    MPI_Abort(MPI_COMM_WORLD,EXIT_FAILURE);
    }
   rb = (int *)malloc(size*chunk*sizeof(int));/*申请接收缓冲区*/
   if ( !rb ) {
        perror( "can't allocate recv buffer");
        free(sb);
        MPI_Abort(MPI_COMM_WORLD,EXIT_FAILURE);
    }
  for ( i=0 ; i < size ; i++ ) {
  for ( j=0 ; j < chunk ; j++ ) {
        sb[i*chunk+j] = rank + i*chunk+j;/*设置发送缓冲区的数据*/
        printf("myid=%d,send to id=%d, data[%d]=%d\n",rank,i,j,sb[i*chunk+j]);
        rb[i*chunk+j] = 0;/*将接收缓冲区清0*/
    }
    }
    /* 执行MPI_Alltoall 调用*/
    MPI_Alltoall(sb,chunk,MPI_INT,rb,chunk,MPI_INT,
          MPI_COMM_WORLD);
  for ( i=0 ; i < size ; i++ ) {
  for ( j=0 ; j < chunk ; j++ ) {
        printf("myid=%d,recv from id=%d, data[%d]=%d\n",rank,i,j,rb[i*chunk+j]);
        /*打印接收缓冲区从其它进程接收的数据*/
    }
    }
    free(sb);
    free(rb);
    MPI_Finalize();
}
```

程序 45　MPI_Alltoall使用示例

正如 MPI_ALLGATHERV **和** MPI_ALLGATHER **的关系一样，** MPI_ALLTOALLV **在** MPI_ALLTOALL**的基础上进一步增加了灵活性，它可以由**sdispls**指定待发送数据的位置，在接收方则由**rdispls**指定接收的数据存放在缓冲区的偏移量。**

所有参数对每个进程都是有意义的，并且所有进程中的comm**值必须一致。**

MPI_ALLTOALL**和**MPI_ALLTOALLV**可以实现**n**次独立的点对点通信，但也有限制：**1) **所有数据必须是同一类型；**2)**所有的消息必须按顺序进行散发和收集。**

```
MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                    recvcounts, rdispls, recvtype, comm)
 IN   sendbuf      发送消息缓冲区的起始地址(可选数据类型)
 IN   sendcounts   向每个进程发送的数据个数(整型数组)
 IN   sdispls      向每个进程发送数据的位移（整型数组）
 IN   sendtype     发送数据的数据类型(句柄)
 OUT  recvbuf      接收消息缓冲区的起始地址(可选数据类型)
 IN   recvcounts   从每个进程中接收的数据个数(整型数组)
 IN   rdispls      从每个进程接收的数据在接收缓冲区的位移（整型数组）
 IN   recvtype     接收数据的数据类型(句柄)
 IN   comm         通信域(句柄)
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
                  MPI_Datatype  sendtype, void* recvbuf,
                  int  *recvcounts,  int  *rdispls,
                  MPI_Datatype  recvtype, MPI_Comm comm)
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
              RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
    <type>  SENDBUF(*),  RECVBUF(*)
   INTEGER  SENDCOUNTS(*),  SDISPLS(*),  SENDTYPE,  RECVCOUNTS(*),
            RDISPLS(*),  RECVTYPE,  COMM,  IERROR
```

MPI调用接口 53 MPI_ALLTOALLV

13.7 同步

```
MPI_BARRIER(comm)
IN   comm   通信域(句柄)
int MPI_Barrier(MPI_Comm comm)
MPI_BARRIER(COMM, IERROR)
INTEGER  COMM,  IERROR
```

MPI调用接口 54 MPI_BARRIER

MPI_BARRIER阻塞所有的调用者直到所有的组成员都调用了它，各个进程中这个调用才可以返回。

```
#include "mpi.h"
#include "test.h"
#include <stdlib.h>
```

```
#include <stdio.h>
int main( int argc, char **argv )
{
      int                    rank,  size,  i;
      int                    *table;
      int                     errors=0;
      MPI_Aint                address;
      MPI_Datatype            type, newtype;
      int                     lens;
     MPI_Init(  &argc,  &argv  );
     MPI_Comm_rank(  MPI_COMM_WORLD,  &rank  );
      MPI_Comm_size( MPI_COMM_WORLD, &size );
     /*  Make  data  table  */
      table  = (int *) calloc (size, sizeof(int));
     table[rank]  =  rank  +  1;   /*准备要广播的数据*/
      MPI_Barrier  ( MPI_COMM_WORLD );
      /*  将数据广播出去*/
     for  (  i=0;  i<size;  i++  )
       MPI_Bcast(  &table[i], 1, MPI_INT, i, MPI_COMM_WORLD  );
      /*  检查接收到的数据的正确性  */
     for  (  i=0;  i<size;  i++  )
         if (table[i] != i+1) errors++;
      MPI_Barrier  ( MPI_COMM_WORLD );/*检查完毕后执行一次同步*/
        ...
      /*其它的计算*/
      MPI_Finalize();
}
```

程序 46 同步示例

13.8 归约

MPI_REDUCE**将组内每个进程输入缓冲区中的数据按给定的操作**op**进行运算，并将其结果返回到序列号为**root**的进程的输出缓冲区中。输入缓冲区由参数**sendbuf、count**和**datatype**定义，输出缓冲区由参数**recvbuf、count**和**datatype**定义，要求两者的元素数目和类型都必须相同。因为所有组成员都用同样的参数**count、datatype、op、root**和**comm**来调用此例程，故而所有进程都提供长度相同、元素类型相同的输入和输出缓冲区。每个进程可能提供一个元素或一系列元素，组合操作依次针对每个元素进行。**

操作op**始终被认为是可结合的，并且所有**MPI**定义的操作被认为是可交换的。用户自定义的操作被认为是可结合的，但可以不是可交换的。**

MPI_REDUCE(sendbuf,recvbuf,count,datatype,op,root,comm)

IN sendbuf **发送消息缓冲区的起始地址(可选数据类型)**

OUT recvbuf **接收消息缓冲区中的地址(可选数据类型)**

IN count **发送消息缓冲区中的数据个数(整型)**

IN datatype **发送消息缓冲区的元素类型(句柄)**

IN op **归约操作符(句柄)**

IN root **根进程序列号(整型)**

IN comm **通信域(句柄)**

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

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)

<type> SENDBUF(*), RECVBUF(*)

INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

MPI**调用接口** 55 MPI_REDUCE

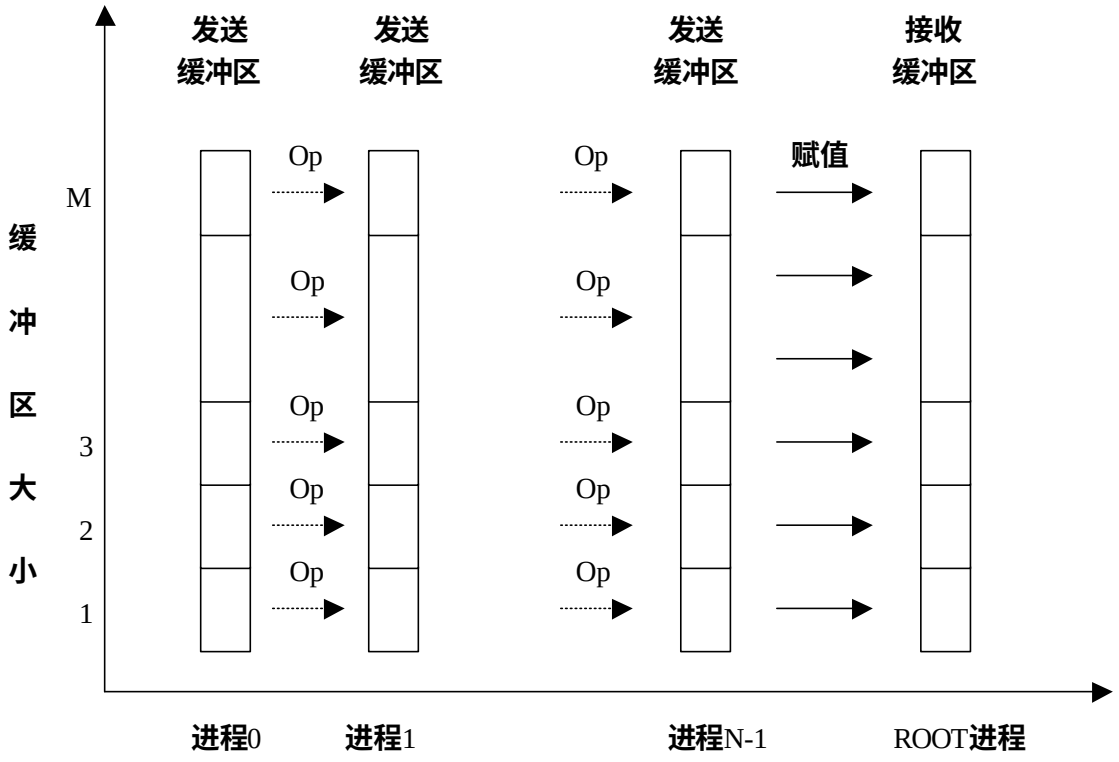


图 60 MPI**归约操作图示**

13.9 MPI预定义的归约操作

MPI中已经定义好的一些操作,它们是为函数MPI_REDUCE和一些其他的相关函数,如MPI_ALLREDUCE、MPI_REDUCE_SCATTER和MPI_SCAN而定义的。这些操作用来设定相应的op。

表格 9 MPI预定义的归约操作

名字	含义
MPI_MAX	最大值
MPI_MIN	最小值
MPI_SUM	求和
MPI_PROD	求积
MPI_LAND	逻辑与
MPI_BAND	按位与
MPI_LOR	逻辑或
MPI_BOR	按位或
MPI_LXOR	逻辑异或
MPI_BXOR	按位异或
MPI_MAXLOC	最大值且相应位置
MPI_MINLOC	最小值且相应位置

MPI_MINLOC和MPI_MAXLOC这两个操作将在4.9.3节中分别讨论.下面列出MPI中定义的其他有关于op和datatype参数的操作以及它们之间允许的组合.首先列出基本数据类型组:

表格 10 C或FORTRAN类型与MPI类型的对应

C或FORTRAN类型	相应的MPI定义类型
C语言中的整型	MPI_INT MPI_LONG MPI_SHORT MPI_UNSIGNED_SHORT MPI_UNSIGNED MPI_UNSIGNED_LONG
Fortran语言中的整型	MPI_INTEGER
浮点数	MPI_FLOAT MPI_DOUBLE MPI_REAL MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE
逻辑型	MPI_LOGICAL
复数型	MPI_COMPLEX
字节型	MPI_BYTE

对每种操作允许的数据类型如下:

表格 11 归约操作与相应类型的对应关系

操作	允许的数据类型
MPI_MAX, MPI_MIN	C整数,Fortran整数,浮点数
MPI_SUM, MPI_PROD	C整数,Fortran整数,浮点数,复数
MPI_LAND, MPI_LOR, MPI_LXOR	C整数,逻辑型
MPI_BAND, MPI_BOR, MPI_BXOR	C整数,Fortran整数,字节型

13.10 求p值

由于求π值的近似计算方法有很好的并行性，同时由于它用到了组通信的广播和归约操作，因此这里较为详细地对它进行介绍。

根据下面的积分公式

$$\int_0^1 \frac{1}{1+x^2}dx = \arctan(x)\,|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{p}{4}$$

令函数f(x)=4/(1+x²)，则有

$$\int_0^1 f(x)\,dx = p$$

而f(x)的图象为

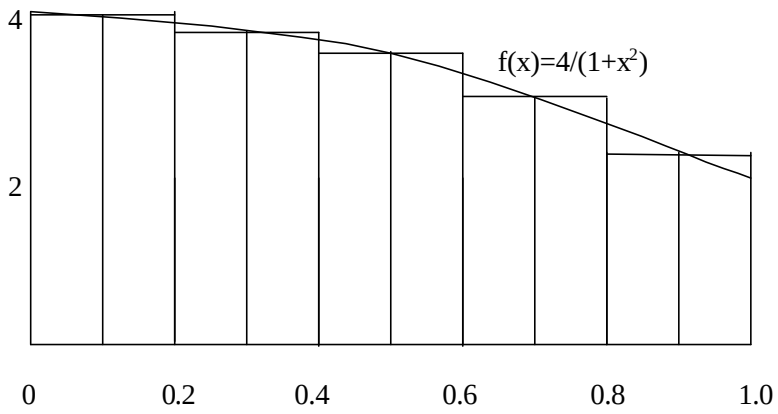


图 61 求π近似值方法的示意图

计算f(x)图象下面从0到1之间的面积即为π的值。而该面积可以用图示的5个小矩形面积的和来近似，矩形的高度取函数在矩形中间点的取值，当用更多的矩形来划分时，该近似值就越接近于真实的π值。设将0到1的区间划分为N个矩形，则近似公式为：

$$p \approx \sum_{i=1}^{N} f\left(\frac{2\times i-1}{2\times N}\right)\times\frac{1}{N} = \frac{1}{N}\times\sum_{i=1}^{N} f\left(\frac{i-0.5}{N}\right)$$

图 62 求π值的近似公式

由上，可得求π值的程序如下：

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

double f(double);
double f(double x)
/*  定义函数f(x) */
{
     return (4.0 / (1.0 + x*x));
}

int main(int argc,char *argv[])
{
    int done = 0, n, myid, numprocs, i;
      double PI25DT = 3.14159265358979323846264 3;
/*  先给出已知的较为准确的π值*/
      double  mypi, pi, h, sum, x;
     double startwtime = 0.0, endwtime;
      int    namelen;
     char  processor_name[MPI_MAX_PROCESSOR_NAME];

      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD,&myid);
      MPI_Get_processor_name(processor_name,&namelen);

      fprintf(stdout,"Process %d of %d on %s\n",
      myid, numprocs, processor_name);

     n = 0;
          if (myid == 0)
           {      printf("Please give N=");
                   scanf(&n);
                 startwtime = MPI_Wtime();
           }
           MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);/*将n值广播出去*/
          h    = 1.0 / (double) n;/*得到矩形的宽度，所有矩形的宽度都相同*/
           sum = 0.0; /*给矩形面积赋初值*/
         for (i = myid + 1; i <= n; i += numprocs)
```

/*每一个进程计算一部分矩形的面积，若进程总数numprocs为4，将0-1区间划分为100个矩形，则各个进程分别计算矩形块

```
0进程    1，5，9，  13，...，97
1进程    2，6，10，14，...，98
2进程    3，7，11，15，...，99
3进程    4，8，12，16，...，100
*/
            {
                    x = h * ((double)i - 0.5);
                     sum += f(x);
              }
            mypi = h * sum;/*各个进程并行计算得到的部分和*/

            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                                MPI_COMM_WORLD);
                /*将部分和累加得到所有矩形的面积，该面积和即为近似π值*/
              if (myid == 0)
                /*执行累加的0号进程将近似值打印出来*/
                {
                  printf("pi is approximately %.16f, Error is %.16f\n",
                              pi, fabs(pi - PI25DT));
                    endwtime = MPI_Wtime();
                    printf("wall clock time = %f\n", endwtime-startwtime);
                    fflush( stdout );
                }
      MPI_Finalize();
}
```

程序 47 求π值

13.11 组归约

只要理解了归约操作，就可以很容易地掌握组归约操作。组归约MPI_ALLREDUCE**就相当于组中每一个进程都作为**ROOT**分别进行了一次归约操作。即归约的结果不只是某一个进程拥有，而是所有的进程都拥有。它在某种程度上和组收集与收集的关系很相似。**

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

IN sendbuf 发送消息缓冲区的起始地址(可选数据类型)

OUT recvbuf 接收消息缓冲区的起始地址(可选数据类型)

IN count 发送消息缓冲区中的数据个数(整型)

IN datatype 发送消息缓冲区中的数据类型(句柄)

IN op 操作(句柄)

IN comm 通信域(句柄)

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM,
              IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

MPI调用接口 56 MPI_ALLREDUCE

13.12 归约并散发

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

IN sendbuf 发送消息缓冲区的起始地址(可选数据类型)

OUT recvbuf 接收消息缓冲区的起始地址(可选数据类型)

IN recvcounts 接收数据个数（整型数组）

IN datatype 发送缓冲区中的数据类型(句柄)

IN op 操作(句柄)

IN comm 通信域(句柄)

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

MPI调用接口 57 MPI_REDUCE_SCATTER

MPI_REDUCE_SCATTER操作可以认为是MPI对每个归约操作的变形，它将归约结果分散到组内的所有进程中去，而不是仅仅归约到ROOT进程。

MPI_REDUCE_SCATTER对由sendbuf、count和datatype定义的发送缓冲区数组的元素逐

个进行归约操作，发送缓冲区数组的长度count= $\sum$ irecvcount[i]。然后，将结果数组的前recvcounts[0]个元素送给进程0的接收缓冲区，再将接下来的recvcounts[1]个元素送给进程1的接收缓冲区，依次类推，最后将最后的recvcounts[N-1]个元素送给进程N-1的接收缓冲区。

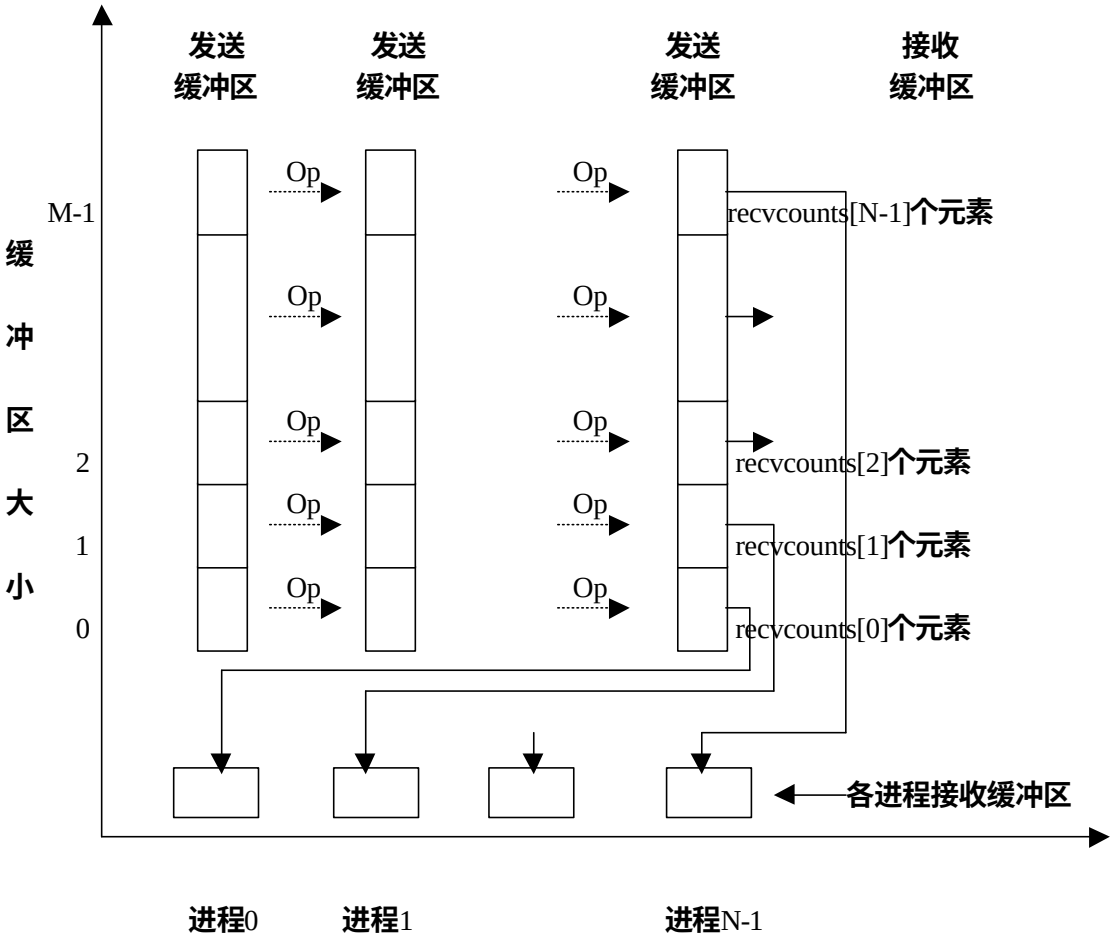


图 63 归约并散发操作

13.13 扫描

可以将扫描看作是一种特殊的归约，即每一个进程都对排在它前面的进程进行归约操作。MPI_SCAN调用的结果是，对于每一个进程i，它对进程0,...,i的发送缓冲区的数据进行指定的归约操作，结果存入进程i的接收缓冲区。

也可以换一个角度，将扫描操作看作是每一个进程i发送缓冲区中的数据与它前面的进程i-1接收缓冲区中的数据进行指定的归约操作后，将结果存入进程i的接收缓冲区；而进程i接收缓冲区中的数据用来和进程i+1发送缓冲区中的数据进行归约。进程0接收缓冲区中的数据就是发送缓冲区的数据。

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	**发送消息缓冲区的起始地址(可选数据类型)**
OUT	recvbuf	**接收消息缓冲区的起始地址(可选数据类型)**
IN	count	**输入缓冲区中元素的个数(整型)**
IN	datatype	**输入缓冲区中元素的类型(句柄)**
IN	op	**操作(句柄)**
IN	comm	**通信域(句柄)**

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

MPI调用接口 58 MPI_SCAN

13.14 不同类型归约操作的简单对比

本节对归约操作、组归约操作和归约并散发操作进行简单地对比，希望通过对比中能加深对各种类型归约操作的理解，并正确使用各种类型归约操作。

归约操作的效果就是接收缓冲区中数据的改变，而接收缓冲区中数据的改变是直接与各个进程发送缓冲区中的数据密切相关的。

下面就不同类型归约前后各个进程发送缓冲区与接收缓冲区中数据的关系，来说明归约操作的最终效果。

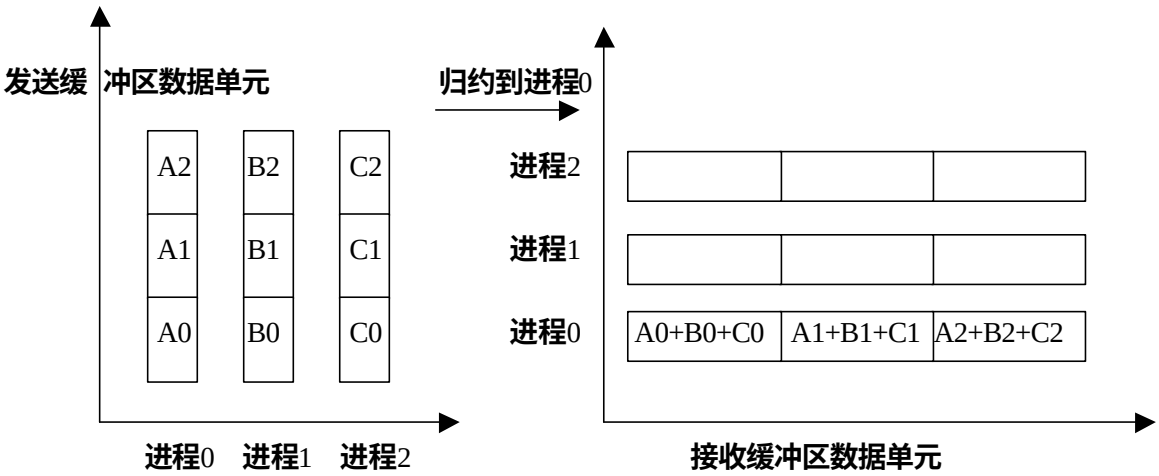


图 64 归约前后发送与接收缓冲区的对比

简单的归约操作只有ROOT**进程的接收缓冲区在归约后其内容有意义，而其它进程接收缓冲区中的内容没有意义。**ROOT**进程接收缓冲区中的数据需要所有进程发送缓冲区中的数**

据进行指定的运算后才能够得到。ROOT进程的接收缓冲区和各个进程的发送缓冲区大小是一样的。

而组归约完成后所有进程中接收缓冲区的内容都有意义，都是对其它所有进程发送缓冲区中的数据进行指定的运算之后的结果。各个进程接收缓冲区中的内容是相同的。对于组归约，各进程发送缓冲区和接收缓冲区的大小相同。

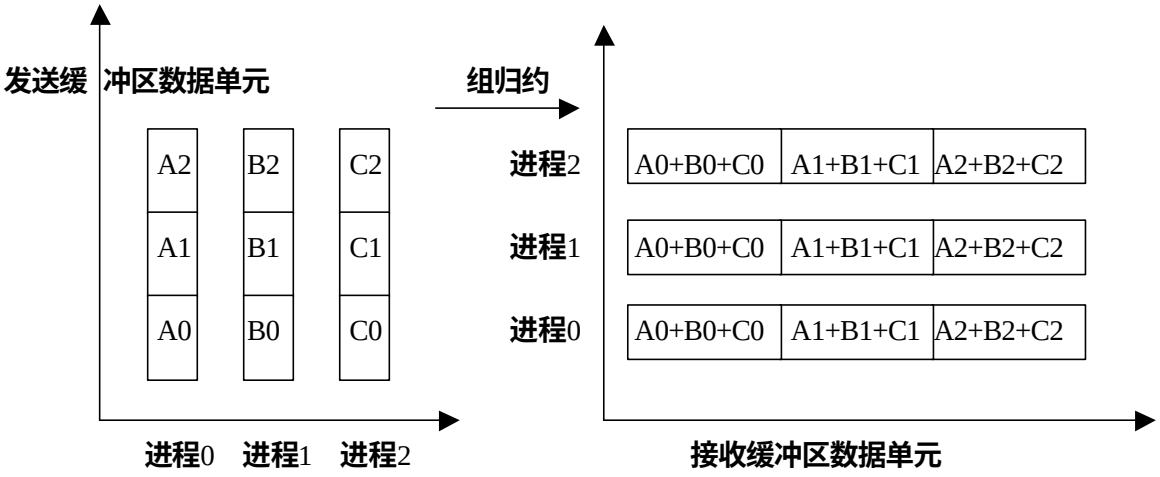


图 65 组归约操作前后发送与接收缓冲区的对比

归约并散发操作较前面两种操作更复杂一些。它在执行归约操作的同时将归约的结果散发到不同进程的接收缓冲区中。在效果上，就如同先使某一个进程作为ROOT进程执行归约操作，然后ROOT进程再将归约的结果放到发送缓冲区中，执行散发操作。对于归约并散发操作，各进程的接收缓冲区的大小是发送缓冲区大小的1/N（N为总的进程个数）。

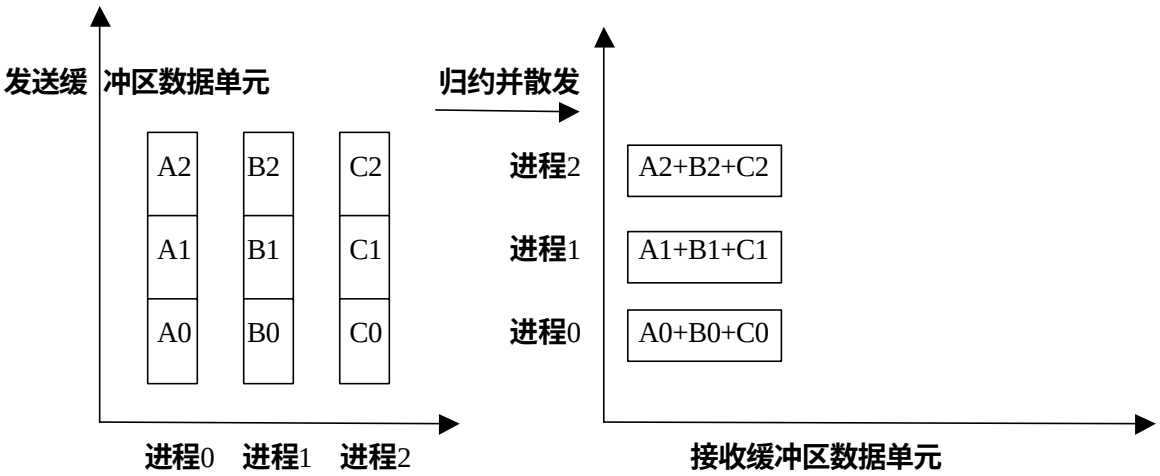


图 66 归约并散发操作前后发送与接收缓冲区的对比

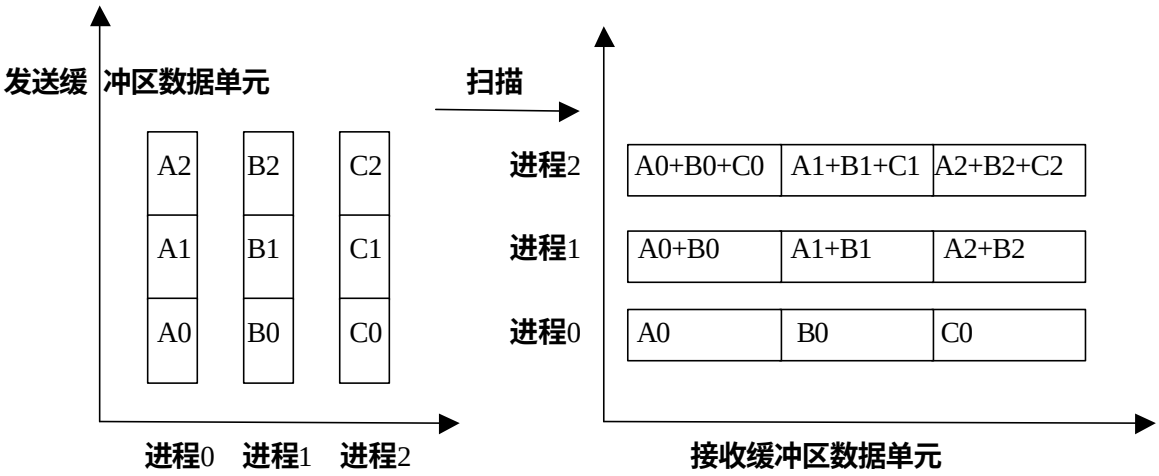


图 67 扫描操作前后发送与接收缓冲区的对比

扫描操作可以认为是每一个进程都执行了一次归约操作，只要理解了归约操作，就很容易理解扫描操作。扫描操作各个进程的发送缓冲区和接收缓冲区的大小相同。

13.15 不正确的组通信方式

无论组通信是同步的还是异步的,一个正确的、可移植调用组通信的程序都不应引起死锁。下面是一些组调用例程的危险例子:

下面的例子是错误的。

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
          break;
    case 1:
        MPI_Bcast(buf2, count, type, 1, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
          break;
}
...
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
          break;
    case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
```

```
            MPI_Bcast(buf2, count, type, 0, comm0);
              break;
        case 2:
            MPI_Bcast(buf1, count, type, 2, comm2);
            MPI_Bcast(buf2, count, type, 1, comm1);
              break;
}
```

程序 48 错误的广播调用次序

假设通信域comm**对应的进程组是**{0,1}，**两个进程按相反的顺序执行两个广播操作,如果操作是同步的话就会引起死锁。在同一个进程组所有成员上的组操作必须按同一个顺序执行。假设通信域**comm0**对应的进程组是**{0,1},**通信域**comm1**对应的进程组是**{1,2},**通信域**comm2**对应的进程组是**{2,0}。**如果广播是同步操作,那么存在一个循环依赖**：comm2**中的广播通信中仅当**comm0**中的广播通信结束后才能结束**；comm0**中的广播通信中仅当**comm1**中的广播通信结束后才能结束**，comm1**中的广播通信仅当**comm2**中的广播通信结束后才能结束**，**这样就会产生死锁**.

下面的例子也是错误的

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
          break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
          break;
}
```

程序 49 错误的广播和点到点阻塞通信调用次序

进程0**执行一个广播操作，后面跟一个阻塞发送操作；进程**1**执行一个与进程**0**的发送操作相匹配的阻塞接收操作，后面跟一个与进程**0**广播相匹配的广播操作。这个程序会产生死锁，因为进程**0**上的广播通信调用可能被阻塞直到进程**1**执行了相应广播通信调用,所以发送不能执行,而进程**1**正是在接收处阻塞,所以永远不可能执行广播操作。无论是组通信还是点对点通信,即使是同步的也不应引起死锁**.

结果不确定的程序。

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
          break;
```

```
        case 1:
            MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
              /*有可能是来自进程1的消息，也可能是来自进程2的消息*/
            MPI_Bcast(buf1, count, type, 0, comm);
            MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
              break;
        case 2:
            MPI_Send(buf2, count, type, 1, tag, comm);
            MPI_Bcast(buf1, count, type, 0, comm);
              break;
}
```

程序 50 结果不确定的调用次序

上述的三个进程都参与了广播通信,进程0在广播通信后向进程1发送了一条消息,进程2在广播通信前向进程1发送了一条消息，进程1在广播通信前后采用通配符进行接收操作。这个程序有两种可能的执行方式,它们随发送和接收匹配的不同而不同。

13.16 MINLOC和MAXLOC

MPI_MINLOC**操作符用于计算全局最小值和这个最小值的索引号**，MPI_MAXLOC**操作符用于计算全局最大值和这个最大值的索引号，这两个函数的一个用途是计算一个全局最小值(最大值)和这个值所在的进程序列号。**

两个操作都是可结合、可交换的。如将MPI_MAXLOC**应用于**$(u_0,0),(u_1,1),..,(u_{n-1},n-1)$**这个序列上进行归约，那么返回结果**(u,r)**满足**

$$u = u_r = \max_{i=0}^{n-1}\{u_i\}, \text{并且} \quad u_i < u_r, i = 0,.., r-1$$

即u**是全局最大值并且**r**是第一个全局最大值所在的位置。同样**MPI_MINLOC**可以被用于返回全局最小值和第一个全局最小值所在的位置。**

如将MPI_MINLOC**应用于**$(u_0,0),(u_1,1),..,(u_{n-1},n-1)$**这个序列上进行归约，那么返回结果**(u,r)**满足**

$$u = u_r = \min_{i=0}^{n-1}\{u_i\}, \text{并且} \quad u_i > u_r, i = 0,.., r-1$$

为了在归约操作中使用MPI_MINLOC**和**MPI_MAXLOC,**必须提供表示这个值对(值及其索引号)参数的类型。**MPI**定义了七个这样的类型,**MPI_MAXLOC**和**MPI_MINLOC**可以采用下列的数据类型**:

表格 12　MPI定义的Fortran语言的值对类型

名字	描述
MPI_2REAL	实型值对
MPI_2DOUBLE_PRECISION	双精度变量值对
MPI_2INTEGER	整型值对

表格 13　MPI定义的C语言的值对类型

名字	描述
MPI_FLOAT_INT	浮点型和整型
MPI_DOUBLE_INT	双精度和整型
MPI_LONG_INT	长整型和整型
MPI_2INT	整型值对
MPI_SHORT_INT	短整型和整型
MPI_LONG_DOUBLE_INT	长双精度浮点型和整型

类型MPI_2REAL**可以理解成按下列方式定义的类型。**

MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)

MPI_2INTEGER、MPI_2DOUBLE_PRECISION**和**MPI_2INT**的定义方式和**MPI_2REAL**相仿**，MPI_FLOAT_INT**类型和下列类型定义等价**:

```
type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
disp[1] = sizeof(float)
block[0] = 1
block[1] = 1
MPI_TYPE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)
```

MPI_LONG_INT**和**MPI_DOUBLE_INT**的定义方式和**MPI_FLOAT_INT**相仿**.

下面的例子每个进程都有一3**个双精度数的数组(以**C**语言为例),计算**3**个位置上的值并返回包含最大值的进程序列号**.

```
/* 每个进程都有一3个双精度数组ain[3] */
double ain[3],aout[3];
int ind[3];
struct {
    double val;
    int rank;
} in[3], out[3];/* 分别定义归约操作的发送缓冲区和接收缓冲区*/
int i, myrank, root;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
for (i=0; i<3; ++i) {
```

```
        in[i].val = ain[i];
        in[i].rank = myrank;
    }/*给发送缓冲区赋初值*/
    MPI_Reduce(in, out, 3, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
    /* 将结果归约到根进程 */
    if (myrank == root) {
        /* 读出其值 */
        for (i=0; i<3; ++i) {
            aout[i] = out[i].val;
            ind[i] = out[i].rank;
        }
    }
```

程序 51 归约操作MPI_MAXLOC示例

假设初始值为如下，共有三个进程，则结果可用下面的表格表示：

表格 14 归约操作MPI_MAXLOC的结果

进程0	(30.5,0)	(41.7,0)	(35.9,0)
进程1	(12.1,1)	(11.3,1)	(13.5,1)
进程2	(100.7,2)	(23.2,2)	(98.4,2)
MPI_MAXLOC**归约结果**	(100.7,2)	(41.7,0)	(98.4,2)

13.17 用户自定义归约操作

```
MPI_OP_CREATE(function, commute, op)
 IN     function      用户自定义的函数(函数)
 IN     commute       可交换则为true,否则为false
 OUT    op            操作(句柄)
int MPI_Op_create(MPI_User_function *function,int commute,MPI_Op *op)
MPI_OP_CREATE(FUNCTION, COMMUTE, OP, IERROR)
     EXTERNAL FUNCTION
     LOGICAL  COMMUTE
     INTEGER  OP, IERROR
```

MPI调用接口 59 MPI_OP_CREATE

MPI**的归约调用不仅可以使用**MPI**预定义的归约操作，而且也允许使用用户自己定义的归约操作。**

MPI_OP_CREATE**将用户自定义的函数**function**和操作**op**联系起来，这样的操作**op**可以象**MPI**预定义的归约操作一样用于各种**MPI**的归约函数中。**

用户自定义的操作必须是可以结合的。如果commute=true**，则此操作同时也是可交换的。**

如果commute=false，则此操作不满足交换律。

function是用户自定义的函数，必须具备四个参数: invec, inoutvec,len和datatype。在C中这个函数的原型是:

```
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
                               MPI_Datatype *datatype);
```

在Fortran中对用户自定义的函数描述如下:

```
FUNCTION USER_FUNCTION(INVEC(*), INOUTVEC(*), LEN, TYPE)
  <type> INVEC(LEN), INOUTVEC(LEN)
  INTEGER LEN, TYPE
```

参数datatype用于控制传送给MPI_REDUCE的数据类型。用户定义的归约函数按如下方式进行工作:

invec和inoutvec分别指出将要被归约的数据所在的缓冲区的首址,len指出将要归约的元素的个数,datatype指出归约对象的数据类型。在用户自定义归约函数中，用数组u[0],...,u[len-1]和参数invec、len和datatype描述的归约元素相对应，用数组v[0],...,v[len-1]和参数inoutvec、len和datatype描述的归约元素相对应，w[0],...,w[len-1]记录归约结果，也由参数inoutvec、len和datatype描述，这样，归约函数的任务就是使得

w[i]= u[i] · v[i] ,i从0到len-1,

其中·是该函数将实现的归约操作。从非正式的角度来看，可以认为invec和inoutvec是函数中长度为len的数组,归约的结果重写了inoutvec的值.每次调用此函数都导致了对这len个元素逐个进行相应的操作。

通常的数据类型可以传给用户自定义的参数,然而互不相邻的数据类型可能会导致低效率。在用户自定义的函数中不能调用MPI中的通信函数。当函数出错时可能会调用MPI_ABORT。

```
MPI_OP_FREE(op)
 IN   op          操作(句柄)
int MPI_Op_free(MPI_Op *op)
MPI_OP_FREE(OP, IERROR)
    INTEGER OP, IERROR
```

MPI调用接口 60 MPI_OP_FREE

MPI_OP_FREE将用户自定义的归约操作撤消，将op设置成MPI_OP_NULL。

下面使一个用户自定义归约操作的例子，它计算一个复数数组的积。

```
typedef struct {
    double real,imag;
} Complex;
/* 用户自定义的函数 */
void myProd(Complex *in, Complex *inout, int *len, MPI_Datatype *dptr)
{
    int i;
```

```
        Complex  c;

      for (i=0; i < *len; ++i) {
            c.real = inout->real*in->real - inout->imag*in->imag;
            c.imag = inout->real*in->imag + inout->imag*in->real;
             *inout = c;
              in++; inout++;
        }
  }
  /* 然后调用它 */
  /* 每个进程都有一个100个元素的复数数组 */
 Complex a[100], answer[100];
  MPI_Op myOp;
 MPI_Datatype ctype;

  /* 告知MPI复数结构是如何定义的 */
 MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
  MPI_Type_commit(&ctype);
  /* 生成用户定义的复数乘积操作 */
  MPI_Op_create(myProd, True, &myOp);
  MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
  /* 归约完成后，计算结果(为100个复数)就已经存放在根进程 */
```

程序 52　用户自定义的归约操作

13.18 小结

组通信是较为复杂的一种通信方式，它需要程序员在编写组通信语句时头脑中同时有两个执行模型，一是当程序运行起来后当前正在运行的进程的行为方式，二是将组通信作为一个整体来考虑，所有进程的行为方式。只有将这两者结合起来，才能够准确把握组通信的内涵，才能够编写出正确的组通信语句。

第14章 具有不连续数据发送的MPI程序设计

MPI除了可以发送或接收连续的数据之外，还可以处理不连续的数据，其基本方法有两种，一是允许用户自定义新的数据类型（又称派生数据类型）；二是数据的打包与解包，即在发送方将不连续的数据打包到连续的区域，然后发送出去，在接收方将打包后的连续数据解包到不连续的存储空间。本章分别对这两种方法进行介绍。

14.1 派生数据类型

前面所介绍的通信，只牵涉含有相同数据类型的相邻缓冲区。这对两种用户限制太大：一种是经常想传送含有不同数据类型值的消息的用户(例如,一个整数计数值跟着一些实数)；另一种是经常发送非连续数据的用户(例如发送矩阵的一个子块)。一种解决的办法是在发送端把非连续的数据打包到一个连续的缓冲区,在接收端再解包。这样做的缺点在于在两端都需要额外的内存到内存拷贝操作,甚至当通信域系统具有收集分散数据功能的时候也是如此。

为了介绍派生数据类型，首先介绍一种通用的数据类型描述方法--类型图。使用类型图可以用比较精确和形式化的方法来描述各种各样的类型。类型图是一系列二元组的集合，两个数据类型是否相同取决于它们的类型图是否相同。类型图的二元组为如下形式：

<基类型，偏移>

则类型图为

类型图={<基类型0，偏移0>,<基类型1，偏移1>,...,<基类型n-1，偏移n-1>}

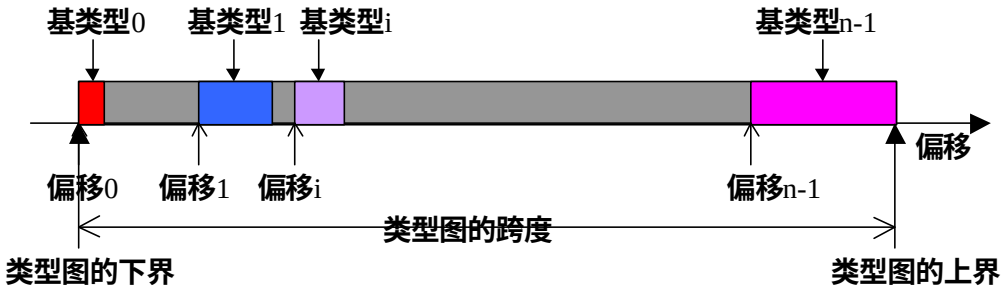


图 68 类型图的图示

基类型指出了该类型图中包括哪些基本的数据类型，而偏移则指出该基类型在整个类型图中的起始位置。基类型可以是预定义类型或派生类型，偏移可正可负，没有递增或递减的顺序要求。而一个类型图中包括的所有基类型的集合称为某类型的类型表，表示为：

类型表={基类型0，...，基类型n-1}

将类型图和一个数据缓冲区的基地址结合起来，可以说明一个通信缓冲区内的数据分布情况。

预定义数据类型是通用数据类型的特例，比如MPI_INT是一个预先定义好了的数据类型句柄，其类型图为{(int, 0)},有一个基类型入口项int和偏移0。其它的基本数据类型与此相似。数据类型的跨度被定义为该数据类型的类型图中从第一个基类型到最后一个基类型间的距离。

即如果某一个类型的类型图为

typemap={($type_0$,$disp_0$),...,($type_{n-1}$,$disp_{n-1}$)},

则该类型图的下界定义为

lb(typemap)=min {$disp_j$}, 0=<j<=n-1

则该类型图的上界定义为

ub(typemap)=max($disp_j$+sizeof($type_j$)), 0=<j<=n-1

该类型图的跨度定义为:

extent(typemap)=ub(typemap)-lb(typemap)+ε

由于不同的类型有不同的对齐位置的要求，ε就是能够使类型图的跨度满足该类型的类型表中的所有的类型都能达到下一个对齐要求所需要的最小非负整数值。

假设type={(double,0),(char,8)}(**一个**double**型的值在偏移**0,**后面在偏移**8**处跟一个字符值**)。**进一步假设**double**型的值必须严格分配到地址为**8**的倍数的存储空间。则该数据类型的**extent**是**16(**从**9**循环到下一个**8**的倍数**)。**一个由一个字符或面紧跟一个双精度值的数据类型**,**其**extent**也是**16。

14.2 新数据类型的定义

14.2.1 连续复制的类型生成

最简单的数据类型生成器是MPI_TYPE_CONTIGUOUS,**它得到的新类型是将一个已有的数据类型按顺序依次连续进行复制后的结果。**

MPI_TYPE_CONTIGUOUS(count,oldtype,newtype)
IN count **复制个数(非负整数)**
IN oldtype **旧数据类型(句柄)**
OUT newtype **新数据类型(句柄)**
int MPI_Type_contiguous(int count,MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_CONTIGUOUS(COUNT,OLDTYPE,NEWTYPE,IERROR)
INTEGER COUNT,OLDTYPE,NEWTYPE,IERROR

MPI**调用接口** 61 MPI_TYPE_CONTIGUOUS

用MPI_TYPE_CONTIGUOUS**构造新的数据类型，设原来的数据类型**oldtype**的类型图为**

{(doubel,0),(char,8)},**其中类型的跨度为**extent=16,**对旧类型重复的次数**count=3,

则newtype**返回的新类型的类型图为**

{(double,0),(char,8),(double,16),(char,24),(double,32),(char,40)}

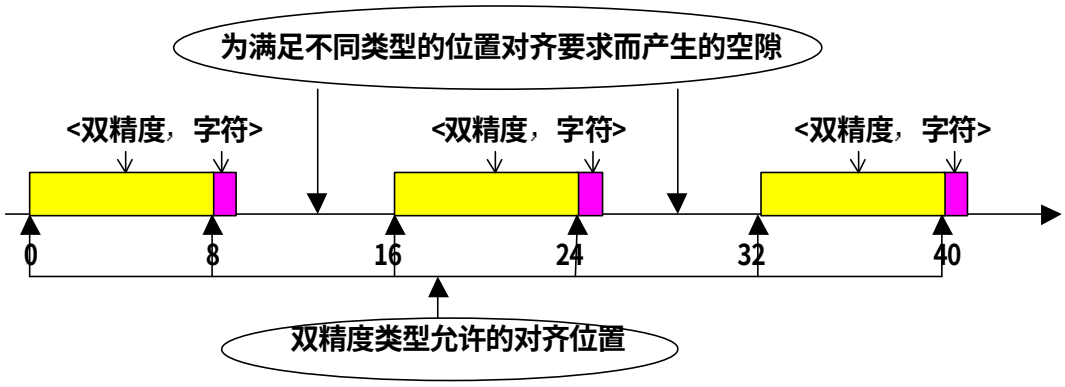


图 69 用MPI_TYPE_CONTIGUOUS产生的新类型

更通用的表达为，设旧类型oldtype**的类型图为**

$\{(type_0, disp_0), (type_{n-1}, disp_{n-1})\}$,**其类型的跨度**extent = ex.

新类型将旧类型连续复制count**次，则新类型**newtype**的类型图为**

$\{(type_0, disp_0),..., (type_{n-1}, disp_{n-1}),$
$(type_0, disp_0+ex),..., (type_{n-1}, disp_{n-1}+ex),$
$...,$
$(type_0, disp_0+ex(count-1)),..., (type_{n-1}, disp_{n-1}+ex(count-1))\}$.

14.2.2 向量数据类型的生成

MPI_TYPE_VECTOR**是一个更通用的生成器，允许复制一个数据类型到含有相等大小块的空间。每个块通过连接相同数量的旧数据类型的拷贝来获得。块与块之间的空间是旧数据类型的**extent**的倍数。**

```
MPI_TYPE_VECTOR(count,blocklength,stride,oldtype,newtype)
        IN count          块的数量(非负整数)
        IN blocklength    每个块中所含元素个数(非负整数)
        IN stride         各块第一个元素之间相隔的元素个数(整数)
        IN oldtype        旧数据类型(句柄)
       OUT  newtypr       新数据类型(句柄)
int MPI_Type_vector(int count,int blocklength,int stride,
            MPI_Datatype oldtype,MPI_Datatype *newtype)
MPI_TYPE_VECTOR(COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,
                            NEWTYPE,IERROR)
      INTEGER  COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR
```

MPI调用接口 62 MPI_TYPE_VECTOR

假设旧数据类型oldtype的类型图为{(double,0),(char,8)},extent=16. 则MPI_TYPE_VECTOR(2,3,4,oldtype,newtype)调用生成的数据类型的类型图为:

{(double,0),(char,8), (double,16),(char,24), (double,32),(char,40),
(double,64),(char,72), (double,80),(char,88),(double,96),(char,104)}.

即两个块,每个旧类型有三个拷贝,相邻块之间的步长stride为4个元素。

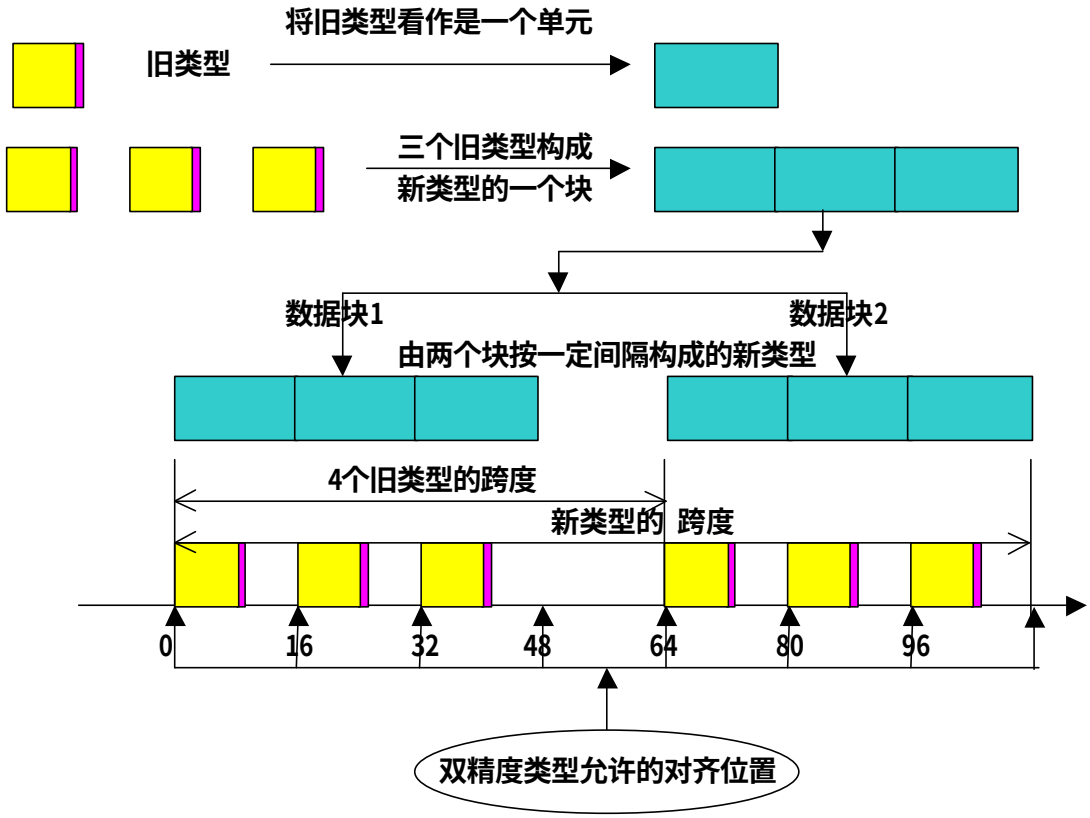


图 70 用MPI_TYPE_VECTOR产生的新数据类型

调用MPI_TYPE_VECTOR(3,1,-2,oldtype,newtype)将生成以下的数据类型:

{(double,0),(char,8),(double,-32),(char,-24),(double,-64),(char,-56)}.

一般地,假设oldtype的类型映像为

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

extent = ex.

设bl为blocklength.新创建的数据类型有count*bl个入口项,类型映像为

{($type_0$, $disp_0$),..., ($type_{n-1}$,$disp_{n-1}$ $type_{n-1}$,$disp_{n-1}$+ex.(stride+bl-1)),...,
($type_0$, $disp_0$+ex.(count-1)),..., ($type_{n-1}$, $disp_{n-1}$+ex.(count-1)),...,
($type_0$, $disp_0$+ex.(count-1).stride),...,
($type_{n-1}$, $disp_{n-1}$+ex.(stride.(count-1)+bl-1)},...,
($type_{n-1}$, $disp_{n-1}$+ex.(stride.(count-1)+bl-1)}.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) 调用等价于调用MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype),或调用MPI_TYPE_VECTOR(1, count, n,

oldtype, newtype), n**为升序**.

MPI_TYPE_HVECTOR(count,blocklength,stride,oldtype,newtype)
IN count **块的数量(非负整数)**
IN blocklength **每个块中所含元素个数(非负整数)**
IN stride **各块起始位置之间相隔的字节数(整数)**
IN oldtype **旧数据类型(句柄)**
OUT newtype **新数据类型(句柄)**
int MPI_Type_hvector(int count,int blocklength,MPI_Aint stride,MPI_Datatype oldtype,
MPI_Datatype *newtype)
MPI_TYPE_HVECTOR(COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR)
INTEGER COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR

MPI**调用接口** 63 MPI_TYPE_HVECTOR

函数MPI_TYPE_HVECTOR**和**MPI_TYPE_VECTOR**基本相同,只是**stride**不再是元素个数,而是字节数。**

假设oldtype**的类型映像为**

{(type$_0$, disp$_0$), (type$_{n-1}$,disp$_{n-1}$)},extent = ex.

设bl**为**blocklength.**新创建的数据类型有**count.bl.n**个入口项,类型映像为**

{(type$_0$, disp$_0$),..., (type$_{n-1}$,disp$_{n-1}$),
(type$_0$, disp$_0$+ex),..., (type$_{n-1}$, disp$_{n-1}$+ex),...,
(type$_0$, disp$_0$+ex.(bl-1)),..., (type$_{n-1}$, disp$_{n-1}$+ex.(bl-1)),
(type$_0$, disp$_0$+stride),...,
(type$_{n-1}$, disp$_{n-1}$+stride),...,
(type$_0$, disp$_0$+stride+ex.(bl-1)),..., (type$_{n-1}$,disp$_{n-1}$+stride+ex.(bl-1)),...,
(type$_0$, disp$_0$+ stride.(count-1)),..., (type$_{n-1}$,disp$_{n-1}$+stride.(count-1)),...,
(type$_0$, disp$_0$+stride.(count-1)+(bl-1).ex),...,
(type$_{n-1}$, dispn-1+stride.(count-1)+(bl-1).ex〉}.

14.2.3 索引数据类型的生成

MPI_TYPE_INDEXED**允许复制一个旧数据类型到一个块序列中(每个块是旧数据类型的一个连接),每个块可以包含不同的拷贝数目和具有不同的偏移.所有的块偏移都是旧数据类型**extent**的倍数**.

MPI_TYPE_INDEXED(count,array_of_blocklengths,array_of_displacemets,oldtype,newtype)

IN count 块的数量（整型）

IN array_of_blocklengths 每个块中所含元素个数(非负整数数组)

IN array_of_displacements 各块偏移值 (整数数组)

IN oldtype 旧数据类型(句柄)

OUT newtypr 新数据类型(句柄)

int MPI_Type_indexed(int count,int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_INDEXED(COUNT,ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS,OLDTYPE,NEWTYPE,IERROR)

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

MPI调用接口 64 MPI_TYPE_INDEXED

设 oldtype 的类型映像为 {(double,0),(char,8)},extent=16. 令 B=(3,1),D=(4,0), 则 MPI_TYPE_INDEXED(2, B, D, oldtype, newtype) 调用生成的数据类型映像是:

{(double,64),(char,72), (double,80),(char,88),(double,96),(char,104), (double,0),(char,8)}.

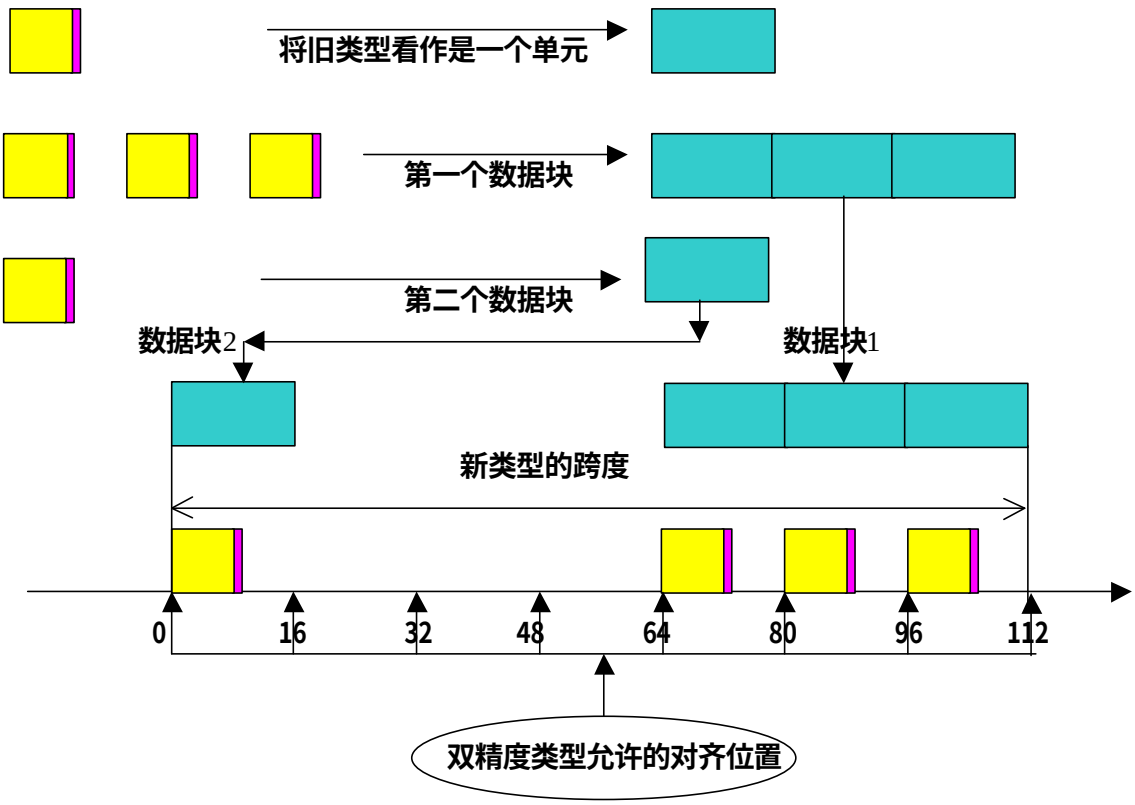


图 71 用MPI_TYPE_INDEXED产生的新数据类型

即,旧类型的三个拷贝在偏移64处开始,一个拷贝从0偏移开始.一般地,假设oldtype的类型映像为

{($type_0$, $disp_0$), ($type_{n-1}$,$disp_{n-1}$)}, extent = ex.

设B为array_of_blocklengths参数,D为array_of_displacements参数.新创建的数据类型有n.SUM(B[i],i=0,...,count-1)项,类型映像为

{($type_0$, $disp_0$+D[0].ex),..., ($type_{n-1}$,$disp_{n-1}$ +D[0].ex),...,

($type_0$, $disp_0$+(D[0]+B[0]-1).ex),...,

($type_{n-1}$,$disp_{n-1}$+(D[0]+B[0]-1).ex),...,

($type_0$, $disp_0$+D[count-1].ex),..., ($type_{n-1}$,$disp_{n-1}$+D[count-1].ex),...,

($type_0$, $disp_0$+(D[count-1]+B[count-1]-1).ex),...,

($type_{n-1}$, $disp_{n-1}$+(D[count-1]+B[count-1]-1).ex.)}.

一个 MPI_TYPE_VECTOR(count,blocklength,stride,oldtype,newtype) 等价于调用MPI_TYPE_INDEXED(count,B,D,oldtype,newtype),其中

D[j]=j.stride, j=0,...,count-1,

B[j]=blocklength, j=0,...,count-1.

函数MPI_TYPE_HINDEXED和MPI_TYPE_INDEXED基本相同,只是array_of_displacements中的块偏移不再是旧数据类型extent的倍数,而是字节数.

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

IN count	块的数量 (整数)
IN array_of_blocklengths	每个块中所含元素个数(非负整数数组)
IN array_of_displacements	各块偏移字节数(整数数组)
IN oldtype	旧数据类型(句柄)
OUT newtypr	新数据类型(句柄)

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

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

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

MPI调用接口 65 MPI_TYPE_HINDEXED

oldtype的类型映像为

{($type_0$, $disp_0$), ($type_{n-1}$,$disp_{n-1}$)},extent = ex.

设B为array_of_blocklengths参数,D为array_of_displacements参数.新创建的数据类型有n.SUM(B[i],i=0,...,count-1)项,类型映像为

{($type_0$, $disp_0$+D[0]),..., ($type_{n-1}$,$disp_{n-1}$ +D[0]),...,

($type_0$, $disp_0$+(D[0]+B[0]-1).ex),...,

($type_{n-1}$, $disp_{n-1}$+D[0]+〈B[0]-1).ex),...,

($type_0$, $disp_0$+D[count-1]),..., (typen-1, $disp_{n-1}$+D[count-1]),...,

($type_0$, $disp_0$+D[count-1]+(B[count-1]-1).ex),...,

(type$_{n-1}$, disp$_{n-1}$+D[count-1]+（B[count-1]-1).ex.)}.

14.2.4 结构数据类型的生成

MPI_TYPE_STRUCT**是最通用的类型生成器，它能够在上面介绍的基础上进一步允许每个块包含不同数据类型的拷贝。**

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

IN count **块的数量（整数)**

IN array_of_blocklengths **每个块中所含元素个数(整数数组)**

IN array_of_displacements **各块偏移字节数(整数数组)**

IN array_of_types **每个块中元素的类型(句柄数组)**

OUT newtypr **新数据类型(句柄)**

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

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

MPI**调用接口** 66 MPI_TYPE_STRUCT

设type1**的类型映像为**

{（double,0），(char,8)}， extent=16.

令B=(2,1,3),D=(0,16,26),T=(MPI_FLOAT, type1, MPI_CHAR).**则**MPI_TYPE_STRUCT(3,B,D,T,newtype)**返回**{(float,0),(float,4),(double,16),(char,24),(char,26),(char,27),(char,28)}.

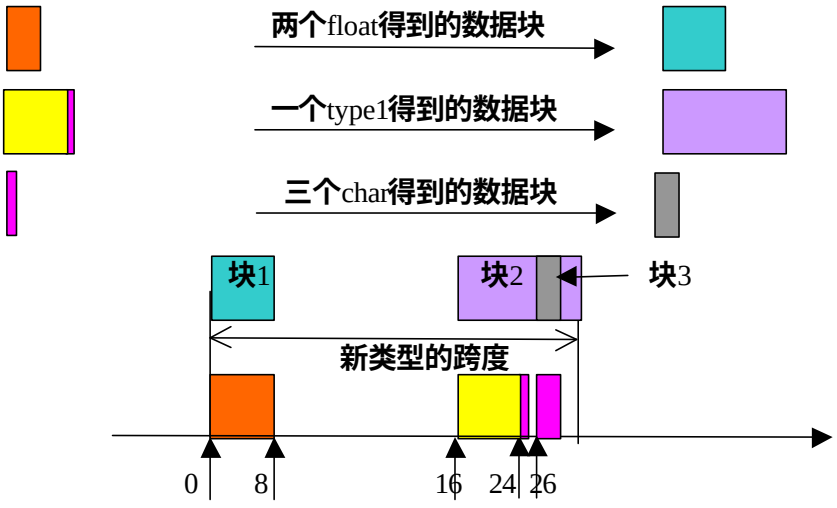


图 72 用MPI_TYPE_STRUCT**生成的新类型**

即两个起始于0的MPI_FLOAT 拷贝后面跟一个起始于16的type1,再跟三个起始于26的MPI_CHAR拷贝。(我们假设一个浮点数占4个字节)。

一般地,假设T是array_of_types参数, T[i]是一个句柄,指向

typemapi ={(type0, disp0), (typen-1,dispn-1)},

extent = ex.

设B为array_of_blocklengths参数,D为array_of_displacements参数.新创建的数据类型有n.SUM(B[i],i=0,...,count-1)项,类型映像为

{($type_0$, $disp_0$+D[0].ex),..., ($type_{n-1}$,$disp_{n-1}$ +D[0].ex),...,

($type_0$, $disp_0$+(D[0]+B[0]-1).ex),..., ($type_{n-1}$,

$disp_{n-1}$+(D[0]+B[0]-1).ex),...,

($type_0$, $disp_0$+D[count-1].ex),..., ($type_{n-1}$,

$disp_{n-1}$+D[count-1].ex),...,

($type_0$, $disp_0$+(D[count-1]+B[count-1]-1).ex),...,

($type_{n-1}$, $disp_{n-1}$+(D[count-1]+B[count-1]-1).ex.)}.

MPI_TYPE_HINDEXED(count,B,D,oldtype,newtype) 等价于调用MPI_TYPE_STRUCT(count, B, D, T, newtype),其中T的每一项都等于oldtype。

14.2.5 新类型递交和释放

新定义的数据类型在使用之前，必须先递交给MPI系统。一个递交后的数据类型，可以作为一个基本类型，用在数据类型生成器中产生新的数据类型。预定义数据类型不需要递交，就可以直接使用。

```
MPI_TYPE_COMMIT(datatype)
        INOUT datatype  递交的数据类型(句柄)
int MPI_Type_commit(MPI_Datatype *datatype)
MPI_TYPE_COMMIT(DATATYPE,IERROR)
        INTEGER  DATATYPE,IERROR
```

MPI调用接口 67 MPI_TYPE_COMMIT

递交操作用于递交新定义的数据类型。注意这里的参数是指向该类型的指针（或句柄），而不是,定义该类型的缓冲区的内容。

```
MPI_TYPE_FREE(datatype)
        INOUT datatype  释放的数据类型(句柄)
int MPI_Type_free(MPI_Datatype *datatype)
MPI_TYPE_FREE(DATATYPE,IERROR)
        INTEGER  DATATYPE,IERROR
```

MPI调用接口 68 MPI_TYPE_FREE

MPI_TYPE_FREE调用将以前已递交的数据类型释放，并且设该数据类型指针或句柄为

空MPI_DATATYPE_NULL。由该派生类型定义的新派生类型不受当前派生类型释放的影响。
释放一个数据类型并不影响另一个根据这个被释放的数据类型定义的其它数据类型。
下面的例子给出了新类型的递交与释放的简单应用。

```
      INTEGER type1, type2
      CALL MPI_TYPE_CNTIGUOUS(5, MPI_REAL, type1, ierr)
C     创建新的类型目标
      CALL MPI_TYPE_COMMIT(type1, ierr)
C     此时type1可以用于通信
      type2 = type1
C     type2可以用于通信，它是一个和type1指向同一个目标的句柄
      CALL MPI_TYPE_VECTOR(3,5,4,MPI_REAL,type1,ierr)
C     创建新的数据类型type1
      CALL MPI_TYPE_COMMIT(type1,ierr)
C     此时type1可作为新类型用于通信之中
```

程序 53 新数据类型的递交与释放

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define NUMBER_OF_TESTS 10

int main( argc, argv )
int argc;
char **argv;
{
    MPI_Datatype  vec1,  vec_n;
     int                blocklens[2];
     MPI_Aint          indices[2];
     MPI_Datatype  old_types[2];
     double             *buf, *lbuf;
    register double *in_p,  *out_p;
     int                rank;
     int                n,  stride;
     double             t1,  t2, tmin;
     int                i, j, k, nloop;
     MPI_Status      status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
     n          =  1000;
```

```
 stride = 24;
  nloop   = 100000/n;
buf = (double *) malloc( n * stride * sizeof(double) );
 if (!buf) {
  fprintf( stderr, "Could not allocate send/recv buffer of size %d\n",
          n * stride );
  MPI_Abort( MPI_COMM_WORLD, 1 );
  }
  lbuf = (double *) malloc( n * sizeof(double) );
 if (!lbuf) {
  fprintf( stderr, "Could not allocated send/recv lbuffer of size %d\n",
          n );
  MPI_Abort( MPI_COMM_WORLD, 1 );
  }

 if (rank == 0)
  printf( "Kind\tn\tstride\ttime (sec)\tRate (MB/sec)\n" );
  /* 使用固定大小向量类型 */
  MPI_Type_vector( n, 1, stride, MPI_DOUBLE, &vec1 );
 MPI_Type_commit( &vec1 );
 tmin = 1000;
for (k=0; k<NUMBER_OF_TESTS; k++) {
 if (rank == 0) {
        /* 保证双方都已准备好 */
       MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
                MPI_BOTTOM, 0, MPI_INT, 1, 14, MPI_COMM_WORLD,
                &status );
        t1 = MPI_Wtime();
       for (j=0; j<nloop; j++) {
         MPI_Send( buf, 1, vec1, 1, k, MPI_COMM_WORLD );
         MPI_Recv( buf, 1, vec1, 1, k, MPI_COMM_WORLD, &status );
         }
       t2 = (MPI_Wtime() - t1) / nloop;
       if (t2 < tmin) tmin = t2;
  }
  else if (rank == 1) {
        /* 保证双方都已准备好 */
       MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
                MPI_BOTTOM, 0, MPI_INT, 0, 14, MPI_COMM_WORLD,
                &status );
       for (j=0; j<nloop; j++) {
         MPI_Recv( buf, 1, vec1, 0, k, MPI_COMM_WORLD, &status );
         MPI_Send( buf, 1, vec1, 0, k, MPI_COMM_WORLD );
         }
```

```
 }
 }
 /* 取往返时间的一半 */
tmin = tmin / 2.0;
if (rank == 0) {
 printf( "Vector\t%d\t%d\t%f\t%f\n",
        n, stride, tmin, n * sizeof(double) * 1.0e-6 / tmin );
 }
 MPI_Type_free( &vec1 );
 /* 使用可变向量类型 */
blocklens[0] = 1;
blocklens[1] = 1;
 indices[0]    = 0;
 indices[1]    = stride * sizeof(double);
old_types[0] = MPI_DOUBLE;
old_types[1] = MPI_UB;
MPI_Type_struct( 2, blocklens, indices, old_types, &vec_n );
 MPI_Type_commit( &vec_n );
 tmin = 1000;
for (k=0; k<NUMBER_OF_TESTS; k++) {
 if (rank == 0) {
       /* 保证双方都已准备好 */
      MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
                MPI_BOTTOM, 0, MPI_INT, 1, 14, MPI_COMM_WORLD,
                &status );
       t1 = MPI_Wtime();
      for (j=0; j<nloop; j++) {
        MPI_Send( buf, n, vec_n, 1, k, MPI_COMM_WORLD );
        MPI_Recv( buf, n, vec_n, 1, k, MPI_COMM_WORLD, &status );
        }
      t2 = (MPI_Wtime() - t1) / nloop;
      if (t2 < tmin) tmin = t2;
 }
 else if (rank == 1) {
       /* 保证双方都已准备好 */
      MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
                MPI_BOTTOM, 0, MPI_INT, 0, 14, MPI_COMM_WORLD,
                &status );
      for (j=0; j<nloop; j++) {
        MPI_Recv( buf, n, vec_n, 0, k, MPI_COMM_WORLD, &status );
        MPI_Send( buf, n, vec_n, 0, k, MPI_COMM_WORLD );
        }
 }
 }
```

```
 /* 取往返时间的一半 */
tmin = tmin / 2.0;
if (rank == 0) {
 printf( "Struct\t%d\t%d\t%f\t%f\n",
        n, stride, tmin, n * sizeof(double) * 1.0e-6 / tmin );
 }
 MPI_Type_free( &vec_n );
 /* Use user-packing with known stride */
 tmin = 1000;
for (k=0; k<NUMBER_OF_TESTS; k++) {
 if (rank == 0) {
       /* Make sure both processes are ready */
      MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
                MPI_BOTTOM, 0, MPI_INT, 1, 14, MPI_COMM_WORLD,
                &status );
       t1 = MPI_Wtime();
      for (j=0; j<nloop; j++) {
        /* If the compiler isn't good at unrolling and changing
           multiplication to indexing, this won't be as good as
           it could be */
        for (i=0; i<n; i++)
             lbuf[i] = buf[i*stride];
        MPI_Send( lbuf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD );
        MPI_Recv( lbuf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD, &status );
        for (i=0; i<n; i++)
             buf[i*stride] = lbuf[i];
        }
      t2 = (MPI_Wtime() - t1) / nloop;
      if (t2 < tmin) tmin = t2;
 }
 else if (rank == 1) {
      /* Make sure both processes are ready */
      MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
                MPI_BOTTOM, 0, MPI_INT, 0, 14, MPI_COMM_WORLD,
                &status );
       for (j=0; j<nloop; j++) {
        MPI_Recv( lbuf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD, &status );
        for (i=0; i<n; i++)
             buf[i*stride] = lbuf[i];
        for (i=0; i<n; i++)
              lbuf[i] = buf[i*stride];
        MPI_Send( lbuf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD );
        }
 }
```

```
 }
/* Convert to half the round-trip time */
 tmin = tmin / 2.0;
 if (rank == 0) {
  printf( "User\t%d\t%d\t%f\t%f\n",
        n, stride, tmin, n * sizeof(double) * 1.0e-6 / tmin );
  }

  /*  Use user-packing with known stride, using addition in the user
       copy code */
  tmin  = 1000;
 for (k=0; k<NUMBER_OF_TESTS; k++) {
  if (rank == 0) {
      /* Make sure both processes are ready */
        MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
                  MPI_BOTTOM, 0, MPI_INT, 1, 14, MPI_COMM_WORLD,
                  &status );
       t1 = MPI_Wtime();
       for (j=0; j<nloop; j++) {
        /* If the compiler isn't good at unrolling and changing
           multiplication to indexing, this won't be as good as
           it could be */
        in_p = buf; out_p = lbuf;
        for (i=0; i<n; i++) {
             out_p[i] = *in_p; in_p += stride;
        }
        MPI_Send( lbuf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD );
        MPI_Recv( lbuf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD, &status );
        out_p = buf; in_p = lbuf;
        for (i=0; i<n; i++) {
             *out_p = in_p[i]; out_p += stride;
        }
        }
       t2 = (MPI_Wtime() - t1) / nloop;
       if (t2 < tmin) tmin = t2;
  }
  else if (rank == 1) {
       /* Make sure both processes are ready */
        MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
                  MPI_BOTTOM, 0, MPI_INT, 0, 14, MPI_COMM_WORLD,
                  &status );
       for (j=0; j<nloop; j++) {
        MPI_Recv( lbuf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD, &status );
        in_p = lbuf; out_p = buf;
```

```
            for (i=0; i<n; i++) {
                *out_p = in_p[i]; out_p += stride;
            }
            out_p = lbuf; in_p = buf;
            for (i=0; i<n; i++) {
                out_p[i] = *in_p; in_p += stride;
            }
            MPI_Send( lbuf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD );
            }
        }
        }
    /* Convert to half the round-trip time */
    tmin = tmin / 2.0;
    if (rank == 0) {
     printf( "User(add)\t%d\t%d\t%f\t%f\n",
            n, stride, tmin, n * sizeof(double) * 1.0e-6 / tmin );
     }

     MPI_Finalize( );
}
```

```
/*************公平性问题*****************************************/
#include "mpi.h"
#include <stdio.h>
int main(argc, argv)
int argc;
char **argv;
{
      int  rank, size, i, buf[1];
      MPI_Status  status;

     MPI_Init( &argc, &argv );
     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
      MPI_Comm_size( MPI_COMM_WORLD, &size );
     if (rank == 0) {
      for (i=0; i<100*(size-1); i++) {
           MPI_Recv( buf, 1, MPI_INT, MPI_ANY_SOURCE,
                    MPI_ANY_TAG, MPI_COMM_WORLD, &status );
            printf( "Msg from %d with tag %d\n",
                 status.MPI_SOURCE, status.MPI_TAG );
      }
      }
```

```
    else {
    for (i=0; i<100; i++)
        MPI_Send( buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
    }
    MPI_Finalize();
    return 0;
}
```

14.3 地址函数

一个MPI**提供的地址调用**MPI_ADDRESS**可以返回某一个变量在内存中相对于预定义的地址**MPI_BOTTOM**的偏移地址。**

```
MPI_ADDRESS(location,address)
      IN location        内存地址(可选数据类型)
      OUT address        相对于位置MPI_BOTTOM的偏移(整型)
int MPI_ADdress(void* location, MPI_Aint *address)
MPI_ADDRESS(LOCATION,ADDRESS,IERROR) <type> LOCATION(*)
      INTEGER ADDRESS,IERROR
```

MPI**调用接口** 69 MPI_ADDRESS

下面的例子根据地址函数得到数组中两个不同元素在内存中的距离。

```
    REAL A(100,100)
    INTEGER I1, I2, DIFF
    CALL MPI_ADDRESS(A(1,1), I1, IERROR)
    CALL MPI_ADDRESS(A(10,10), I2, IERROR)
    DIFF = I2 - I1
```

程序 54 **简单的**MPI_ADDRESS**调用示例**

DIFF**的值是**[(10-1)*10-(10-1)]*sizeof(real);I1**和**I2**的值依赖于具体执行。**

下面的例子为了定义一个新的MPI**数据类型，该类型顺序包括一个整型和一个双精度型，使用了类型生成函数**MPI_Type_struct，**但是提供给该函数的相对地址偏移是通过对一个包含整型和双精度类型的结构的两个成员分别进行**MPI_ADDRESS**调用实现的。**

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{
    int             rank;
```

```
    struct { int a; double b } value; /*定义一个包含整型和双精度型的结构*/
     MPI_Datatype  mystruct;
     int                blocklens[2];
     MPI_Aint         indices[2];
     MPI_Datatype  old_types[2];

     MPI_Init( &argc, &argv );
     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    /* One value of each type */
     blocklens[0] = 1; /*新数据类型中包含一个整型*/
     blocklens[1] = 1; /*新数据类型中包含一个双精度型*/
      /* The base types */
     old_types[0] = MPI_INT;/*新类型的第一个组成部分是整型*/
     old_types[1] = MPI_DOUBLE;/*  新类型的第二个组成部分是双精度型*/
      /*  得到整型和双精度型的相对位置*/
     MPI_Address(  &value.a, &indices[0] );
     MPI_Address(  &value.b, &indices[1] );
      /*  设置在新类型中的相对偏移*/
    indices[1] = indices[1] - indices[0];
     indices[0] = 0;
     MPI_Type_struct( 2, blocklens, indices, old_types, &mystruct );/*生成新的MPI数据类
型*/
     MPI_Type_commit( &mystruct );/*递交*/

      do {
           if (rank == 0)      scanf( "%d %lf", &value.a, &value.b );
  /*只有进程0读需要广播的整型和双精度型数据*/
           MPI_Bcast( &value, 1, mystruct, 0, MPI_COMM_WORLD );/*对新数据类型表示
的数据进行广播*/

           printf( "Process %d got %d and %lf\n", rank, value.a, value.b );
      } while (value.a >= 0);

     /* Clean up the type */
      MPI_Type_free( &mystruct );/*新类型释放*/
     MPI_Finalize( );
}
```

程序 55 包含多种不同类型的新MPI数据类型的定义

14.4 与数据类型有关的调用

```
MPI_TYPE_EXTENT(datatype,extent)
        IN datatype       数据类型(句柄)
        OUT  extent       数据类型extent(整型)
int MPI_Type_extent(MPI_Datatype datatype, int *extent)
MPI_TYPE_EXTENT(DATATYPE,SIZE,IERROR)
        INTEGER  DATATYPE,EXTENT,IERROR
```

MPI调用接口 70　MPI_TYPE_EXTENT

MPI_TYPE_EXTENT**以字节为单位返回一个数据类型的跨度**extent。

```
MPI_TYPE_SIZE(datatype,size)
        IN datatype      数据类型(句柄)
        OUT  size        数据类型大小(整型)
int MPI_Type_size(MPI_Datatype datatype, int *size)
MPI_TYPE_SIZE(DATATYPE,SIZE,IERROR)
        INTEGER  DATATYPE,SIZE,IERROR
```

MPI调用接口 71　MPI_TYPE_SIZE

MPI_TYPE_SIZE**以字节为单位，返回给定数据类型有用部分所占空间的大小，即跨度减去类型中的空隙后的空间大小。和**MPI_TYPE_EXTENT**相比，**MPI_TYPE_SIZE**不包括由于对齐等原因导致数据类型中的空隙所占的空间。**

假设MPI_RECV(buf, count, datatype, dest, tag, comm, status) **被执行,其中**datatype**的数据类型图为**:

{(type0, disp0),..., (typen-1,dispn-1)},

则以接收操作完成后返回的状态status**为参数，可以通过调用**MPI_GET_ELEMENTS、MPI_GET_COUNT**得到不同的信息。**

```
MPI_GET_ELEMENTS（status, datatype, count )
        IN status       接收操作返回的状态(状态类型)
        IN datatype     接收操作使用的数据类型(句柄)
        OUT  count      接收到的基本元素个数(整型)
int MPI_Get_elements( MPI_Status status, MPI_Datatype datatype, int *count)
MPI_GET_ELEMENTS(STATUS,DATATYPE,COUNT,IERROR)
        INTEGER  STATUS(MPI_STATUS_SIZE),DATATYPE,COUNT,IERROR
```

MPI调用接口 72　MPI_GET_ELEMENTS

MPI_CET_COUNT**返回的是以指定的数据类型为单位，接收操作接收到的数据的个数；**

而MPI_GET_ELEMENTS返回的则是以基本的类型为单位的数据的个数。MPI_GET_COUNT和MPI_GET_ELEMENT对基本数据类型使用时返回值相同。

MPI_GET_COUNT(status, datatype,count)
IN status　　接收操作返回的状态(状态类型)
IN datatype　接收操作使用的数据类型(句柄)
OUT count　接收到的以指定的数据类型为单位的数据个数(整型)
int MPI_Get_count(MPI_Status * status, MPI_Datatype datatype, int * count)
MPI_GET_COUNT(STATUS, DATATYPE,COUNT,IERROR)
　INTEGER　STATUS(MPI_STATUS_SIZE), DATATYPE,COUNT
　　IERROR

MPI调用接口 73　MPI_GET_COUNT

MPI_GET_COUNT从状态变量status中得到接收到的数据的个数，它是以指定的数据类型datatype为单位来计算的。

下面的例子展示了MPI_GET_COUNT和MPI_GET_ELEMENT的使用

```
      ....
      CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
C     定义新类型
      CALL MPI_TYPE_COMMIT(Type2,ierr)
      ...
      CALL MPI_COMM_RANK(comm, rank, ierr)
      IF(rank.EQ.0) THEN
          CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
C         向进程1发送2个实型数据
          CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
C         再向进程1发送3个实型数据
      ELSE
          CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
C         从进程0接收Type2类型的数据
          CALL MPI_GET_COUNT(stat, Type2, i, ierr)
C         计算收到的Type2类型数据的个数，这里i=1
          CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
C         计算按基本类型REAL计算收到的数据的个数，这里i=2
          CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
C         再从进程0接收数据
          CALL MPI_GET_COUNT(stat, Type2, i, ierr)
C         计算按Type2类型计算收到的数据的个数，这里i=MPI_UNDEFINED，
C         因为3个REAL类型无法用Type2类型来度量
          CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=3
C         计算按基本类型REAL计算收到的数据的个数，这里i=3
      END IF
```

程序 56 接收数据个数的获取

函数MPI_GET_ELEMENTS也可以在执行检查操作来检查元素个数之后再调用。

14.5 下界标记类型和上界标记类型

MPI提供两个特殊的数据类型，称为伪数据类型，上界标记类型MPI_UB和下界标记类型MPI_LB。这两个数据类型不占空间即extent(MPI_LB)= extent(MPI_UB) =0)。他们主要是用来影响数据类型的跨度，从而对派生数据类型产生影响。

一般地,如果 typemap ={($type_0$, $disp_0$),..., ($type_{n-1}$,$disp_{n-1}$)}，
当typemap的下界被定义为

$$lb(Typemap)=\begin{cases} \min_j disp_j & \text{不包含lb类型} \\ \min_j \{disp_j \text{ such that } type_j=lb\} & \text{若含有lb类型} \end{cases}$$

类似地,typemap的上界定义为

$$ub(Typemap)=\begin{cases} \max_j disp_j+sizeof(type_j)+\varepsilon & \text{不包含ub类型} \\ \max_j \{disp_j \text{ such that } type_j=ub\} & \text{若含有ub类型} \end{cases}$$

则数据类型typemap的跨度为

extent(typemap) =ub(typemap)-lb(typemap)+e

```
MPI_TYPE_LB(datatype,displacement)
        IN datatype          数据类型(句柄)
       OUT  displacement     下界的偏移(整数)
int MPI_Type_lb (MPI_Datatype datatype, int *displacement)
MPI_TYPE_LB (DATATYPE,DISPLACEMENT,IERROR)
        INTEGER  DATATYPE,DISPLACEMENT,IERROR
```

MPI调用接口 74 MPI_TYPE_LB

```
MPI_TYPE_UB(datatype,displacement)
        IN datatype          数据类型(句柄)
       OUT  displacement     上界的偏移(整数)
int MPI_Type_ub (MPI_Datatype datatype, int *displacement)
MPI_TYPE_UB (DATATYPE,DISPLACEMENT,IERROR)
        INTEGER  DATATYPE,DISPLACEMENT,IERROR
```

MPI调用接口 75 MPI_TYPE_UB

比如D=(-3,0,6); T=(MPI_LB,MPI_INT,MPI_UB),B=(1,1,1).则

MPI_TYPE_STRUCT(3,B,D,T,type1)

产生一个extent为9的新的数据类型,并且在偏移0处含有一个整数。该类型是由序列{(lb,-3),(int,0),(ub,6)}定义的数据类型。如果该数据类型被

MPI_TYPE_CONTIGUOUS(2,type1,type2)

调用复制两次,则新产生的数据类型可以用序列{(lb,-3),(int,0),(int,9),(ub,15)}来描述。(如果lb或ub出现在数据类型两端以外的位置，则它们可以被忽略掉)

下面例子定义的数据类型可以用于处理三角阵和矩阵转置。

```
      REAL  a(100,100),b(100,100)
      INTEGER  disp(100),blocklen(100),ltype,myrank,ierr
      INTEGER  status(MPI_STATUS_SIZE)
C 拷贝数组a的下三角部分到数组b的下三角部分
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank)
C 计算每列开始和大小，下三角阵第i列的第一个元素为a(i-1,i)，该列共有100-i个元素
C 转换为一维坐标后为100*(i-1)+i，注意FORTRAN程序是列优先排列
      DO  i=1, 100
          disp(i)  = 100*(i-1) + i
          block(i)  =  100-i
      END  DO
C 创建一个可以代表下三角阵的数据类型
      CALL MPI_TYPE_INDEXED(100,block,disp,MPI_REAL,1type,ierr
      CALL MPI_TYPE_COMMTT(1type,ierr)

      CALL MPE_SENDRECV(a,1,1type,myrank,0,b,1 1type,myrank,0,
      MPI_COMM_WORLD,statusierr)
C 进程0矩阵a和矩阵b的下三角部分互换
```

程序 57 下三角矩阵数据类型的定义和使用

下面的例子通过定义新的数据类型实现转置矩阵的功能。

```
      REAL  a(100,100),b(100,100)
      INTEGER  row,xpose,sizeofreal,myrank,ierr
      INTEGER  status(MPI_STATUS_SIZE)
C 转置矩阵a到b
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank)
      CALL MPI_TYPE_EXTENT(MPI_REAL,sizeofreal,ierr)
C     得到实型的大小
C 定义一行row为100个实型数，相邻两个数之间的间隔为100个实型数的距离。
C 它其实就是内存中一行的表示
      CALL MPI_TYPE_VECTOR(100,1,100,MPI_REAL,row,ierr)
C 在row的基础上定义一个转置数据类型xpose，它表示的数据的排列是原来矩阵的转置
      CALL MPI_TYPE_HVECTOR(100,1,sizeofreal,row,xpose,lerr)
      CALL MPI_TYPE_COMMIT(xpose,ierr)
```

C 以行主元顺序发送矩阵并以列主元顺序接收，实现a和b互为转置

```
CALL MPI_SENDRECV(a,1,xpose,myrank,0,b,100*100,MPI_REAL,MYRANK,0,
_COMM_WORLD,status,ierr)
```

程序 58 矩阵转置数据类型的定义

14.6 打包与解包

打包(Pack)**和解包**(Unpack)**操作是为了发送不连续的数据，在发送前显式地把数据包装到一个连续的缓冲区，在接收之后从连续缓冲区中解包。**

MPI_PACK**把由**inbuf,incount,datatype**指定的发送缓冲区中的**inbount**个**datatype**类型的消息放到起始为**outbuf**的连续空间，该空间共有**outcount**个字节。输入缓冲区可以是**MPI_SEND**允许的任何通信缓冲区。入口参数**position**的值是输出缓冲区中用于打包的起始地址，打包后它的值根据打包消息的大小来增加，出口参数**position**的值是被打包的消息占用的输出缓冲区后面的第一个地址。通过连续几次对不同位置的消息调用打包操作，就将不连续的消息放到了一个连续的空间。**comm**参数是将在后面用于发送打包的消息时用的通信域。**

```
MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm )
      IN  inbuf          输入缓冲区起始地址(可选数据类型)
      IN  incount        输入数据项个数(整型)
      IN datatype        每个输入数据项的类型(句柄)
      OUT  outbuf        输出缓冲区开始地址(可选数据类型)
      IN outcount        输出缓冲区大小(整型)
      INOUT position     缓冲区当前位置(整型)
      IN comm            通信域(句柄)
int MPI_Pack(void* inbuf, int incount, MPI_datatype, void *outbuf, int outcount, int
                  *position, MPI_Comm comm)
MPI_PACK(INBUF,INCOUNT,DATATYPE,OUTBUF,OUTCOUNT,POSITION,COMM,
                  IERROR)
INBUF(*),OUTBUF(*)
 INTEGER  INCOUNT,DATATYPE,OUTCOUNT,POSITION,COMM,IERROR
```

MPI调用接口 76 MPI_PACK

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

IN inbuf **输入缓冲区起始(选择)**

IN insize **输入数据项数目(整型)**

INOUT position **缓冲区当前位置，字节(整型)**

OUT outbuf **输出缓冲区开始(选择)**

IN outcount **输出缓冲区大小，字节(整型)**

IN datatype **每个输入数据项的类型(句柄)**

IN comm **打包的消息的通信域(句柄)**

int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

MPI_UNPACK(INBUF,INSIZE, POSITION,OUTBUF,OUTCOUNT, DATATYPE, COMM, IERROR)

INBUF(*),OUTBUF(*)

INTEGER INSIZE, POSITION,OUTCOUNT, , DATATYPE COMM,IERROR

MPI**调用接口** 77 MPI_UNPACK

MPI_UNPACK**和**MPI_PACK**对应，它从**inbuf**和**insize**指定的缓冲区空间将不连续的消息解开，放到**outbuf,outcount,datatype**指定的缓冲区中。输出缓冲区可以是**MPI_RECV**允许的任何通信缓冲区。输入缓冲区是一个连续的存储空间，大小为**insize**字节，开始地址为**inbuf。**入口参数**position**的初始值是输出缓冲区中被打包消息占用的起始地址，解包后它的值根据打包消息的大小来增加，因此出口参数**position**的值是输出缓冲区中被解包的消息占用空间后面的第一个地址。通过连续几次对已打包的消息调用与打包时相应的解包操作，就可以将连续的消息解开放到一个不连续的空间。**comm**参数是用于接收消息的通信域。

注意MPI_RECV**和**MPI_UNPACK**的区别：在**MPI_RECV**中**,count**参数指明的是可以接收的最大项数。实际接收的项数是由接收的消息的长度来决定的。在**MPI_UNPACK**中**,count**参数指明实际打包的项数，相应消息的大小是参数**position**的增加值。

一个打包单元可以用MPI_PACKED**作为类型发送。发送类型可以是任何点到点通信或者组通信调用。用**MPI_PACKED**发送的数据可以用任意数据类型来接收，只要它和实际接收到的消息的数据类型相匹配。以任何类型发送的消息(包括**MPI_PACKED**类型)都可以用**MPI_PACKED**类型接收。这样的消息于是就可以被调用**MPI_UNPACK**来解包。

接收到的打包消息可以被解包成几个不连续的消息，这是通过几个连续的对MPI_UNPACK**调用来实现的，第一个调用提供**position=0，对于后续的调用，是以前一个调用输出的**position**的值作为输入，而使用和前一个调用相同的**inbuf、insize**和**comm**值。**

MPI_PACK_SIZE(incount, datatype, comm, size)

IN incount **指定数据类型的个数(整型)**

IN datatype **数据类型(句柄)**

IN comm **通信域(句柄)**

OUT size **以字节为单位**，incount**个**datatype**数据类型需要的空间(整型)**

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

MPI_PACK_SIZE(INCOUNT,DATATYPE,COMM,SIZE,IERROR)

INTEGER INCOUNT,DATATYPE,COMM,SIZE,IERROR

MPI**调用接口** 78 MPI_PACK_SIZE

MPI_PACK_SIZE**调用的返回值是**size，**表示**incount**个**datatype**数据类型需要的空间的大小。该调用返回的是上界，而不是一个精确界，这是因为包装一个消息所需要的精确空间可能依赖于上下文(例如，第一个打包单元中包装的消息可能占用相对更多的空间)。**

下面的例子将两个不同的整型数打包后一次以MPI_PACKED**类型发送，而接收进程以整数类型**MPI_INT**接收到一个包含两个整数的数组中。**

```
int position , i,j,a[2];
char buff[1000];
...
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank ==0) { /* 进程0发送消息*/
    position =0;/*打包的起始位置*/
    MPI_Pack(&i,1,MPI_INT,buff,1000,&position, MPI_COMM_WORLD);
   /*将整数i打包*/
    MPI_Pack(&j,1,MPI_INT,buff,1000,&position, MPI_COMM_WORLD);
   /*将整数j打包*/
    MPI_Send(buff,position, MPI_PACKED,1,0,MPI_COMM_WORLD); }
   /*将打包后的数据发送出去*/
else if(myrank==1) {
/* 进程1接收消息 */
    MPI_Recv(a,2, MPI_INT,0,0,MPI_COMM_WORLD)/*以整型从进程0接收消息*/
}
```

程序 59 相同类型数据的打包和解包

下面的例子将一个整型和100**个浮点型打包后再解包。**

```
int position , i;
float a[1000];
char buff[1000];
MPI_Status status;
....
```

```
 MPI_Comm_rank(MPI_Comm_world,&myrank);
if (myrank ==0) {
        /* 进程0发送数据 */
        int len[2];
        MPI_Aint disp[2];
        MPI_Datatype type[2], newtype;
        i=100;
        /* 设置新类型中包含数据的个数 */
        len[0]=1;
        len[1]=i;
        MPI_Address( &i,disp);/*i相对于MPI_BOTTOM的偏移*/
        MPI_Address( a,disp+1); /*a相对于MPI_BOTTOM的偏移*/
        type[0]=MPI_INT;/*设置第一个类型是整型*/
        type[1]=MPI_FLOAT; /*设置第二个类型是浮点型*/
        MPI_Type_struct(2,len,disp,type,&newtype);/*定义新的数据类型，它包括一个整型与1000个浮点型*/
        MPI_Type_commit(&newtype);/*新类型递交*/
        /*数据打包*/
        position =0;/*打包的开始位置*/
        MPI_Pack(MPI_BOTTOM, 1,newtype, buff,
        1000,&position,MPI_COMM_WORLD);/*将i和数组a打包到buff*/
        /* 将打包数据发送出去*/
        MPI_Send(buff,postion, MPI_PACKED,1,0, MPI_COMM_WORLD)
 }
 else if(myrank ==1) {
        MPI_Recv(buff, 1000,MPI_PACKED,0,0,&status); /* 接收打包数据 */
        position =0;
        MPI_Unpack(buff,1000,&position,&i,1,MPI_INT,MPI_COMM_WORLD);
        /* 先将打包的浮点数个数解包*/
        MPI_Unpack(buff,1000,&position,a,i,MPI_FLOAT,MPI_COMM_WORLD);
        /*再将浮点数解包*/
 }
```

程序 60 不同类型数据的打包与解包

下面的程序给出了一个完整的打包与解包的例子，ROOT**进程将读入的一个整数和一个双精度数打包，然后广播给所有的进程，各进程再将数据解包后打印。**

```
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
int argc;
char **argv;
{
    int             rank;
```

```
    int             packsize, position;
    int             a;
    double          b;
    char            packbuf[100];

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    do {
    if (rank == 0) {/*进程0读入数据*/
    scanf( "%d %lf", &a, &b );
    packsize = 0;/*打包开始位置*/
    MPI_Pack( &a, 1, MPI_INT, packbuf, 100, &packsize,
              MPI_COMM_WORLD );/*将整数a打包*/
    MPI_Pack( &b, 1, MPI_DOUBLE, packbuf, 100, &packsize,
              MPI_COMM_WORLD );/*将双精度数b打包*/
    }
    MPI_Bcast( &packsize, 1, MPI_INT, 0, MPI_COMM_WORLD );/*广播打包数据的大小*/
    MPI_Bcast( packbuf, packsize, MPI_PACKED, 0, MPI_COMM_WORLD );/*广播打包的数
据*/
    if (rank != 0) {
        position = 0;
        MPI_Unpack( packbuf, packsize, &position, &a, 1, MPI_INT,
              MPI_COMM_WORLD );/*其他进程先将a解包*/
        MPI_Unpack( packbuf, packsize, &position, &b, 1, MPI_DOUBLE,
              MPI_COMM_WORLD );/*再将b解包*/
    }

    printf( "Process %d got %d and %lf\n", rank, a, b );
        } while (a >= 0);/*若a为负数则结束，否则继续上述过程*/
        MPI_Finalize( );
        return 0;
    }
```

程序 61　一个完成的打包解包例子

14.7 小结

数据类型在MPI中是一个非常重要的概念，它可以大大减轻程序设计的负担，提高程序的可读性和可维护性，而且，在MPI-2中的并行文件I/O也是直接以数据类型的概念为基础建立起来的。因此，数据类型的意义不仅仅在于不连续数据的传送，更重要的是提提供给程序员一个明确而清晰的概念，可以用这一概念来进行程序设计。相对而言，打包和解包的功能就比较单一，就是将不连续的数据打包后再解开。

第15章 MPI的进程组和通信域

进程组和通信域是MPI中的重要概念，本章重点介绍关于它们的调用和管理方法。

15.1 简介

通信域包括通信上下文、进程组、虚拟处理器拓扑、属性等内容，用于综合描述了通信进程间的通信关系。通信域分为组内通信域和组间通信域。组内通信域用于描述属于同一组内进程间的通信；组间通信域用于描述属于不同进程组的进程间的通信。

进程组是通信域的一个重要组成部分，它定义了不同进程的有序集合，每一进程具有一个编号，如0，1，...，N-1等，进程组限定了参加通信的进程的范围。进程是与实现相关的对象。MPI将进程组内的每个进程与一个整数rank相联系。序列号是连续的并从0开始。

MPI_GROUP_EMPTY是一个特殊的预定义组，它没有成员。预定义的常数MPI_GROUP_NULL是为无效组句柄使用的值。因此不应将MPI_GROUP_EMPTY与MPI_GROUP_NULL混淆，前者是一个空组的有效句柄，而后者则是一个无效句柄。前者可以在组操作中作为一个参数使用；后者在组释放时被返回。

通信上下文是通信域所具有的一个特性，它允许对通信空间进行划分。一个上下文所发送的消息不能被另一个上下文所接收。进一步说，允许集合操作独立于挂起的点对点操作。上下文不是显式的MPI对象；它们仅作为通信域实现的一部分而出现。

一旦MPI_INIT被调用，则会产生一个预定义组内通信域MPI_COMM_WORLD，它包括所有的进程。另外还提供了通信域MPI_COMM_SELF，该通信域仅包括自身进程。预定义的常数MPI_COMM_NULL是为无效通信域使用的值。

所有的MPI实现都要求提供MPI_COMM_WORLD通信域。在进程的生命期中不允许将其释放。与该通信域对应的组不是以预定义常数的形式出现的，但是可以使用MPI_COMM_GROUP访问它。

15.2 进程组的管理

本节描述MPI中对进程组的维护。这些操作的执行不要求进程间通信。

```
MPI_GROUP_SIZE(group,size)
IN        group     进程组（句柄）
OUT       size      组内进程数（整数）
int MPI_Group_size(MPI_Group group,int *size)
MPI_GROUP_SIZE(GROUP,SIZE,IERROR)
      INTEGER  GROUP,SIZE,IERROR
```

MPI调用接口 79 MPI_GROUP_SIZE

MPI_GROUP_SIZE返回指定进程组中所包含的进程的个数。

```
MPI_GROUP_RANK(group,rank)
IN      group       进程组（句柄）
OUT     rank        调用进程的序列号/MPI_UNDEFINED（整数）
int MPI_Group_rank(MPI_Group group,int *rank)
MPI_GROUP_RANK(GROUP,RANK,IERROR)
    INTEGER  GROUP,RANK,IERROR
```

MPI调用接口 80　MPI_GROUP_RANK

MPI_GROUP_RANK返回调用进程在给定进程组中的编号rank，有些类似于MPI_COMM_RANK，当然，若调用进程不在给定的进程组内，则返回MPI_UNDEFINED。

```
MPI_GROUP_TRANSLATE_RANKS(group1,n,ranks1,group2,ranks2)
IN      group1      进程组1（句柄）
IN      n           数组rank1和rank2的大小（整数）
IN      ranks1      进程标识数组（整数数组），在进程组group1中的标识
IN      group2      进程组2（句柄）
OUT     ranks2      ranks1在进程组group2中对应的标识数组（整型数组）
int MPI_Group_translate_ranks(MPI_Group group1,int n,int *ranks1,
                              MPI_Group group2,int *ranks2)
MPI_GROUP_TRANSLATE_RANKS(GROUP1,N,RANKS1,GROUP2,RANKS2,
                          IERROR)
    INTEGER  GROUP1,N,RANDS1(*),GROUP2,RANKS2,IERROR
```

MPI调用接口 81　MPI_GROUP_TRANSLATE_RANKS

MPI_GROUP_TRANSLATE_RANKS返回进程组group1中的n个进程（由rank1指定）在进程组group2中对应的编号（相应的编号放在rank2中）。若进程组group2中不包含进程组group1中指定的进程，则相应的返回值为MPI_UNDEFINED。此函数可以检测两个不同进程组中相同进程的相对编号。例如，如果知道了在组MPI_COMM_WORLD中某些进程的序列号，可能也想知道在该组的子集中它们的序列号。

```
MPI_GROUP_COMPARE(group1,group2,result)
IN      group1      进程组（句柄）
IN      group2      进程组（句柄）
OUT     result      比较结果（整数）
int MPI_Group_compare(MPI_Group group1,MPI_Group group2,int *result)
MPI_GROUP_COMPARE(GROUP1,GROUP2,RESULT,IERROR)
    INTEGER  GROUP1,GROUP2,RESULT,IERROR
```

MPI调用接口 82　MPI_GROUP_COMPARE

MPI_GROUP_COMPARE**对两个进程组**group1**和**group2**进行比较，如果两个进程组**group1**和**group2**所包含的进程以及相同进程的编号都完全相同，则，返回**MPI_IDENT；**如果两个进程组**group1**和**group2**所包含的进程完全相同但是相同进程的编号在两个组中并不相同，则返回**MPI_SIMILAR；**否则返回**MPI_UNEQUAL。

从已存在进程组，可以构造该组的子集或超集。不同进程上可定义不同的组；一个进程也可以定义不包括其自身的组。

```
MPI_COMM_GROUP(comm,group)
IN      comm      通信域（句柄）
OUT     group     和comm对应的进程组（句柄）
int MPI_Comm_group(MPI_Comm comm, MPI_Group * group)
MPI_COMM_GROUP(COMM,GROUP,IERROR)
    INTEGER  COMM,GROUP,IERROR
             MPI_COMM_GROUP
```

MPI**调用接口** 83　MPI_COMM_GROUP

MPI_COMM_GROUP**返回指定的通信域所包含的进程组。**

```
MPI_GROUP_UNION(group1,group2,newgroup)
IN      group1      进程组（句柄）
IN      group2      进程组（句柄）
OUT     newgroup    求并后得到的进程组（句柄）
int MPI_Group_union(MPI_Group group1,MPI_Group group2,MPI_Group *newgroup)
MPI_GROUP_UNION(GROUP1,GROUP2, NEWGROUP, IERROR)
    INTEGER  GROUP1,GROUP2,NEWGROUP,IERROR
```

MPI**调用接口** 84　MPI_GROUP_UNION

MPI_GROUP_UNION**返回的新进程组**newgroup**是第一个进程组**group1**中的所有进程加上进程组**group2**中不在进程组**group1**中出现的进程。该并集中的元素次序是第一组中的元素次序后跟第二组中出现的元素。**

```
MPI_GROUP_INTERSECTION(group1,group2,newgroup)
IN      group1      进程组（句柄）
IN      group2      进程组（句柄）
OUT     newgroup    求交后得到的进程组（句柄）
int MPI_Group_intersection(MPI_Group group1,MPI_Group group2,MPI_Group
*newgroup)
MPI_GROUP_INTERSECTION(GROUP1,GROUP2,NEWGROUP,IERROR)
    INTGETER GROUP1,GROUP2,NEWGROUP,IERROR
```

MPI**调用接口** 85　MPI_GROUP_INTERSECTION

MPI_GROUP_INTERSECTION**返回的新进程组**newgroup**是同时在进程组**group1**和进程组**group2**中出现的进程。该交集中的元素次序同第一组。**

```
MPI_GROUP_DIFFERENCE(group1,group2,newgroup)
IN      group1      进程组（句柄）
IN      group2      进程组（句柄）
OUT     newgroup    求差后得到的进程组（句柄）
int MPI_Group_difference(MPI_Group group1,MPI_Group group2,MPI_Group *newgroup)
MPI_GROUP_DIFFERENCE(GROUP1,GROUP2,NEWGROUP,IERROR)
    INTEGER  GROUP1,GROUP2,NEWGROUP,IERROR
```

MPI**调用接口** 86　MPI_GROUP_DIFFERENCE

MPI_GROUP_DIFFERENCE**返回的新进程组**newgroup**是在第一个进程组**group1**中出现但是又不在第二个进程组**group2**中出现的进程。该差集中的元素次序同第一组。**

以上所有的新组可以是空的，也就是说，可以是MPI_GROUP_EMPTY。

```
MPI_GROUP_INCL(group,n,ranks,newgroup)
IN      group       进程组（句柄）
IN      n           ranks数组的大小〔整型〕
IN      ranks       进程标识数组〔整数数组〕
OUT     newgroup    新的进程组〔句柄〕
int MPI_Group_incl(MPI_Group group,int n,int *ranks,MPI_Group *newgroup)
MPI_GROUP_INCL(GROUP,N,RANKS,NEWGROUP,IERROR)
    INTEGER  GROUP,NRANKS(*),NEWGROUP,IERROR
```

MPI**调用接口** 87　MPI_GROUP_INCL

MPI_GROUP_INCL**将已有进程组中的**n**个进程**rank[0]，...，rank[n-1]**形成一个新的进程组**newgroup。**如果**n=0，**则**newgroup**是**MPI_GROUP_EMPTY，**此函数可用于对一个组中的元素进行重排序。**

```
MPI_GROUP_EXCL(group,n,ranks,newgroup)
IN      group       进程组(句柄)
IN      n           数组ranks的大小(整型)
IN      ranks       不出现在newgroup中的进程标识数组（整型数组）
OUT     newgroup    新进程组（句柄）
int MPI_Group_excl(MPI_Group group, int n , int *ranks,MPI_Group *newgroup)
MPI_GROUP_EXCL(GROUP,N,RANKS,NEWGROUP,IERROR)
    INTEGER  GROUP,N,RANKS(*),NEWGROUP,IERROR
```

MPI**调用接口** 88　MPI_GROUP_EXCL

MPI_GROUP_EXCL**将已有进程组**group**中的**n**个进程**ranks[0],...,ranks[n-1]**删除后形成新的进程组**newgroup。ranks**中**n**个元素中的每一个必须是**group**中的有效序列号且所有的元素都必须是不同的，如果**n=0, newgroup**与**group**相同。**

MPI_GROUP_RANGE_INCL(group,n,ranges,newgroup)

IN group **进程组(句柄)**

IN n **数组**ranges**的大小(整型)**

IN ranges **三元组整数数组（整型）**

OUT newgroup **新的进程组（句柄）**

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],MPI_Group *newgroup)

MPI_GROUP_RANGE_INCL(GROUP,N,RANGES,NEWGROUP,IERROR)

INTEGER GROUP,N,RANGES(3,*),NEWGROUP,IERROR

MPI**调用接口** 89 MPI_GROUP_RANGE_INCL

MPI_GROUP_RANGE_INCL**将已有进程组**group**中的**n**组由**ranges**指定的进程形成一个新的进程组**newgroup。**如果**ranges**包含三元组**

$$(first_1,last_1,stride_1),...,(first_n,last_n,stride_n),$$

则newgroup**中包含**group**中具有序列号**

$first_1$, $first_1+stride_1$, ..., $first_1+(last_1-first_1)/stride_1*stride_1$,...

$first_n$, $first_n+stride_n$, ..., $first_n+(last_n-first_n)/stride_n*stride_n$

的一系列进程。

每一个被计算的序列号必须是group**中的有效序列号而且所有被计算的序列号都必须是不同的。比如**(1,9,2) (15,20,3),(21,30,2)**将包含如下进程**

1, 3, 5, 7, 9,15,18,21,23,25,27,29

MPI_GROUP_RANGE_EXCL(group,n,ranges,newgroup)

IN group **进程组(句柄)**

IN n **数组**ranges**的大小(整型)**

IN ranges **三元组整数数组（整型）**

OUT newgroup **新的进程组（句柄）**

int MPI_Group_range_excl(MPI_Group group,int n, int ranges[][3], MPI_Group *newgroup)

MPI_GROUP_RANGE_EXCL(GROUP,N,RANGES,NEWGROUP,IERROR)

INTEGER GROUP,N,RANGES(3,*),NEWGROUP,IERROR

MPI**调用接口** 90 MPI_GROUP_RANGE_EXCL

MPI_GROUP_RANGE_EXCL**从已有进程组**group**中除去**n**个三元组**rangs**所指定的进程后形成新的进程组**newgroup。**它和**MPI_GROUP_INCL**很相近。**

```
MPI_GROUP_FREE(group)
IN/OUT        group          进程组(句柄)
int MPI_Group_free(MPI_Group *group)
MPI_GROUP_FREE(GROUP,IERROR)
      INTEGER  GROUP,IERROR
```

MPI调用接口 91 MPI_GROUP_FREE

MPI_GROUP_FREE释放一个已有的进程组，然后置句柄group为MPI_GROUP_NULL，任何正在使用此组的操作将正常完成。

15.3 通信域的管理

本节描述MPI中对通信域的维护。对通信域的访问不要求进程间通信。创建通信域的操作有时求进程间通信。

```
MPI_COMM_SIZE(comm,size)
IN        comm          通信域(句柄)
OUT       size          comm组内的进程数(整数)
int MPI_Comm_size(MPI_Comm comm, int *size)
MPI_COMM_SIZE(COMM,SIZE,IERROR)
      INTEGER  COMM,SIZE,IERROR
```

这一调用在前面已经介绍过，它返回给定的通信域中包含的进程的个数。

```
MPI_COMM_RANK(comm,rank)
IN        comm          通信域(句柄)
OUT       rank          调用进程的标识号（整型）
int MPI_Comm_rank(MPI_Comm comm, int *rank)
MPI_COMM_RANK(COMM,RANK,IERROR)
      INTEGER  COMM,RANK,IERROR
```

此调用前面也介绍过，它返回调用进程在给定的通信域中的编号rank。

MPI_COMM_COMPARE(comm1,comm2,result)
IN comm1 **第一个通信域(句柄)**
IN comm2 **第二个通信域(句柄)**
OUT result **比较结果(整数)**
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2,int *result)
MPI_COMM_COMPARE(COMM1,COMM2,RESULT,IERROR)
INTEGER COMM1,COMM2,RESULT,IERROR

MPI**调用接口** 92 MPI_COMM_COMPARE

MPI_COMM_COMPARE**该调用对两个给定的通信域进行比较，当**comm1**和**comm2**是同一对象的句柄时，结果为**MPI_IDENT；**如果仅仅是各进程组的成员和序列编号都相同，则结果为**MPI_CONGRUENT；**如果两个通信域的组成员相同但序列编号不同，则结果是**MPI_SIMILAR，**否则结果是**MPI_UNEQUAL。

MPI**通信域的预定义通信域**MPI_COMM_WORLD**它是在**MPI**的外部被定义的。**

MPI_COMM_DUP(comm,newcomm)
IN comm **通信域(句柄)**
OUT newcomm comm**的拷贝(句柄)**
int MPI_Comm_dup(MPI_Comm comm,MPI_Comm *newcomm)
MPI_COMM_DUP(COMM,NEWCOMM,IERROR)
INTEGER COMM, NEWCOMM,IERROR

MPI**调用接口** 93 MPI_COMM_DUP

MPI_COMM_DUP**对已有的通信域**comm**进行复制，得到一个新的通信域**newcomm。**在**newcomm**中除一个新的上下文外，要返回一个具有同样组的新通信域，及任何复制的缓冲信息。**

MPI_COMM_CREATE(comm,group,newcomm)
IN comm **通信域(句柄)**
IN group **进程组 (句柄)**
OUT newcomm **返回的新通信域(句柄)**
int MPI_Comm_create(MPI_Comm comm,MPI_Group group,MPI_Comm *newcomm)
MPI_COMM_CREATE(COMM,GROUP,NEWCOMM,IERROR)
INTEGER COMM,GROUP,NEWCOMM,IERROR

MPI**调用接口** 94 MPI_COMM_CREATE

MPI_COMM_CREATE根据group所定义的进程组，创建一个新的通信域，该通信域具有、新的上下文。对于不在group中的进程，本调用返回MPI_COMM_NULL。所有的group参数都必须具有同样的值,而且group必须是与comm对应进程组的一个子集。

```
MPI_COMM_SPLIT(comm,color,key,newcomm)
IN      comm      通信域(句柄)
IN      color     标识所在的子集（整数)
IN      key       对进程标识号的控制(整数)
OUT     newcomm   新的通信域(句柄)
int MPI_Comm_split(MPI_Comm comm,int color, int key,MPI_Comm *newcomm)
MPI_COMM_SPLIT(COMM,COLOR,KEY,NEWCOMM,IERROR)
INTEGER COMM,COLOR,KEY,NEWCOMM,IERROR
```

MPI调用接口 95 MPI_COMM_SPLIT

MPI_COMM_SPLIT对于通信域comm中的进程都要执行，每一个进程都要指定一个color值，根据color值的不同，此调用首先将具有相同color值的进程形成一个新的进程组，新产生的通信域与这些进程组一一对应，而新通信域中各个进程的顺序编号是根据key的大小决定的，即key越小，则该进程在新通信域中的进程编号也越小，若一个进程中的key相同，则根据这两个进程在原来通信域中的顺序编号决定新的编号。一个进程可能提供color值MPI_UNDEFINED，在这种情况下，newcomm返回MPI_COMM_NULL。实质上，将相同color内的所有进程中的关键字的值置为同一个值导致的结果是在新通信域中进程的相对先后次序和原来的相同。

```
MPI_COMM_FREE(comm)
IN/OUT      comm      将被释放的通信域（句柄）
int MPI_Comm_free(MPI_Comm *comm)
MPI_COMM_FREE(COMM,IERROR)
      INTEGER  COMM,IERROR
```

MPI调用接口 96 MPI_COMM_FREE

MPI_COMM_FREE该调用释放给定的通信域，调用结束后该句柄被置为MPI_COMM_NULL。任何使用此通信域的挂起操作都会正常完成，仅当没有对此对象的活动引用时，它才会被实际撤消。

下面的例子给出了新的进程组和通信域的定义和使用方法。它给出了如何创建一个组，该组包含了第0个进程之外的所有进程，同时阐述了如何为该新组(commslave)形成一个通信域。此新通信域在一个集合调用中被使用，而且所有进程在上下文MPI_COMM_WORLD中都执行一个集合调用。本例还说明了MPI_COMM_WORLD中的通信与commslave中的通信可以互不干扰。

```
main(int argc,char **argv)
{
    int me,count,count2;
```

```
void *send_buf,*recv_buf,*send_buf2,*recv_buf2;
MPI_Group MPI_GROUP_WORLD,grprem;
MPI_Comm commslave;
static int rank[]={0};
...
MPI_Init(&argc,&argv);
MPI_Comm_group(MPI_COMM_WORLD,&MPI_GROUP_WORLD);
/*得到MPI_COMM_WORLD对应的进程组*/
MPI_Comm_rank(MPI_COMM_WORLD,&me);
MPI_Group_excl(MPI_GROUP_WORLD,1,ranks,&grprem);/*创建一个不包括进程0的新的进程组*/
MPI_Comm_create(MPI_COMM_WORLD,grprem,&commslave);/*根据新创建的进程组，创建一个不包括进程0的新的通信域*/

if((me!=0)
{/*进程0之外的进程执行如下操作*/
    ...
    MPI_Reduce(send_buf,recv_buff,count,MPI_INT,MPI_SUM,1,commslave);/*使用新的通信域进行通信，不包括原来的进程0*/
    ...
 }
/*使用MPI_COMM_WORLD使得所有的进程都参加通信 */
MPI_Reduce(send_buf2,recv_buff2,count2,MPI_INT,MPI_SUM,0,
                    MPI_COMM_WORLD);

MPI_Comm_free(&commslave);
MPI_Group_free(&MPI_GROUP_WORLD);
MPI_Group_free(&grprem);
/*释放进程组和通信域*/
MPI_Finalize();
}
```

程序 62 创建进程组和通信域的简单示例

15.4 组间通信域

组间通信域是一种特殊的通信域，该通信域包括两个进程组，通过组间通信域实现这两个不同进程组内进程之间的通信。一般把调用进程所在的进程组叫做本地组，而把另一个组叫做远程组。

```
MPI_COMM_TEST_INTER(comm,flag)
IN      comm        通信域(句柄)
OUT     flag        测试结果(逻辑值)
int MPI_Comm_test_inter(MPI_Comm comm,int *flag)
MPI_COMM_TEST_INTER(COMM,FLAG,IERROR)
      INTEGER  COMM,IERROR  LOGICAL  FLAG
```

MPI调用接口 97　MPI_COMM_TEST_INTER

MPI_COMM_TEST_INTER判断给定的通信域是组内通信域还是组间通信域，如果是组间通信域则返回true，否则返回false。

```
MPI_COMM_REMOTE_SIZE(comm,size)
IN      comm        组间通信域(句柄)
OUT     size        comm的远程组中进程的个数(整数)
int MPI_COMM_Comm_remote_size(MPI_Comm comm,int *size)
MPI_COMM_REMOTE_SIZE(COMM,SIZE,IERROR)
      INTEGER  COMM,SIZE,IERROR
```

MPI调用接口 98　MPI_COMM_REMOTE_SIZE

MPI_COMM_REMOTE_SIZE返回组间通信域内远程进程组的进程个数。

```
MPI_COMM_REMOTE_GROUP(comm,group)
IN      comm        组间通信域(句柄)
OUT     group       comm的远程组(句柄)
int MPI_Comm_remote_group(MPI_Comm comm,MPI_Group *group)
MPI_COMM_REMOTE_GROUP(COMM,GROUP,IERROR)
      INTEGER  COMM,GROUP,IERROR
```

MPI调用接口 99　MPI_COMM_REMOTE_GROUP

MPI_COMM_REMOTE_GROUP返回组间通信域中的远程进程组。

MPI_INTERCOMM_CREATE(local_comm,local_leader,peer_comm,
remote_leader ,tag,newintercomm)

IN	local_comm	**本地组内通信域(句柄)**
IN	local_leader	**本地组内特定进程的标识号(整型)**
IN	peer_comm	**“对等”通信域，仅在**local_leader**中有意义(句柄)**
IN	remote_leader	**远程组特定进程在**peer_comm**中对应的标识号（整型）**
IN	tag	**“安全”标志(整型)**
OUT	newintercomm	**返回的新组间通信域(句柄)**

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

MPI_INTERCOMM_CREATE(LOCAL_COMM,LOCAL_LEADER,PEER_COMM,REMOTE_LEADER ,TAG,NEWINTERCOMM,IERROR)

INTEGER LOCAL_COMM,LOCAL_LEADER,PEER_COMM,REMOTE_LEADER, TAG,NEWINTERCOMM,IERROR

MPI**调用接口** 100 MPI_INTERCOMM_CREATE

MPI_INTERCOMM_CREATE**调用创建了一个组间通信域，它包括两个通信域。其形成方式是，每个进程提供自身所在的一个通信域**local_comm**中特定进程的标识**local_leader**（同一个本地进程组中的进程给出的**local_leader**必须相同），同时给出另一个通信域中特定进程在**peer_comm**中的标识**remote_leader**（同一个本地进程组中的进程给出的**remote_leader**也必须相同），形成相同组间通信域的进程必须提供相同的**tag**，在这里**tag**不允许是**MPI_WILD_TAG。**一般地，用**MPI_COMM_WORLD**的复制品来作为**peer_comm。

MPI_INTERCOMM_MERGE(intercomm,high,newintracomm)

IN	intercomm	**组间通信域(句柄)**
IN	high	**标识(逻辑值)**
OUT	newintracomm	**新的组内通信域(句柄)**

int MPI_Intercomm_merge(MPI_Comm intercomm,int high,MPI_Comm *newintracomm)

MPI_INTERCOMM_MERGE(INTERCOMM,HIGH,INTRACOMM,IERROR)
INTEGER INTERCOMM,INTRACOMM,IERROR LOGICAL HIGH

MPI**调用接口** 101 MPI_INTERCOMM_MERGE

MPI_INTERCOMM_MERGE**将一个组间通信域包含的两个通信域合并，形成一个组内通信域。**high**值用于决定新形成的组内通信域中进程的编号，若对于一个组中的进程都提供**high=true**，另一个组中的进程都提供**high=false**，则提供**true**值的组的进程的编号在前，另一个组的编号在后。如两个组的进程都提供相同的**high**值，则新通信域中进程的编号是任意的。**

下面的例子首先将一个通信域进行分裂，得到三个通信域，也就有三个进程组。然后使得组0与组1通信，组1与组2通信。因此，组0需要一个组间通信域，组1需要两个组间通信域，而组2需要一个组间通信域。

```
main(int argc,char**argv)
{
    MPI_Comm myComm;/*用于标识本地子组的组内通信域*/
    MPI_Comm myFirstComm;/*组间通信域*/
    MPI_Comm mySecondComm;/*组间通信域*/
    int membershipKey;
    int rank;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    membershipKey=rank%3;
    /*将MPI_COMM_WORLD 分裂为三个子通信域，membershipKey 相同的进程在一个子通信域中*/
    MPI_Comm_split(MPI_COMM_WORLD,membershipKey,rank,&myComm);
/*
```

表格 15 通信域分裂

子通信域0	子通信域1	子通信域2
membershipKey=0	membershipKey=1	membershipKey=2
rank=0，3，6	rank=1，4，7	rank=2，5，8

```
*/
    if(membershipKey==0)
    {/*子通信域0对应的进程组*/
/*得到组0与组1的组间通信域myFirstComm */
        MPI_Intercomm_create(myComm,0,MPI_COMM_WORLD,1, 1,&myFirstComm);
    }
    else if (membershipKey==1)
    {/*子通信域1对应的进程组*/
/*得到组1与组0的组间通信域myFirstComm */
        MPI_Intercomm_create(myComm,0,MPI_COMM_WORLD,0, 1,&myFirstComm);
        /*得到组1与组2的组间通信域mySecondComm */
        MPI_Intercomm_ereate(myComm,0,MPI_COMM_WORLD,2,
                12,&mySecondComm);
    }
    else if (membershipKey==2)
    {/*子通信域2对应的进程组*/
    /*得到组2与组1的组间通信域myFirstComm */
        MPI_Intercomm_create(myComm,0,MPI_COMM_WORLD,1,
                12,&myFirstComm);
    }
```

```
    ...

    switch(membershipKey)/*释放生成的新通信域*/
    {
        case 1:
            MPI_COMM_free(&mySecondComm);/*只有1时才有第二个通信域*/
        case 0:
        case 2:
        MPI_COMM_free(&myFirstComm);
        break;
    }
    MPI_Finalize();
}
```

程序 63 通信域的分裂与组间通信域的生成

15.5 属性信息

MPI提供类似于Cache的手段，允许一个应用将任意的信息，称为属性，附加到通信域上。属性对于进程是本地的,它只专属于它所依附的通信域，MPI不能将属性从一个通信域传给另一个通信域，除非使用MPI_COMM_DUP对通信域进行复制。

这里所定义的关于属性的调用包括下列内容：

* 生成一个关键字值，用户指定"回调"函数，当通信域被破坏或被复制时，MPI可通过此函数通知应用程序。

* 存储或查询属性值

MPI_KEYVAL_CREATE(copy_fn,delte_fn,keyval,extra_state)

IN	copy_fn	用于keyval的复制回调函数
IN	delete_fn	用于keyval的删除回调函数
OUT	keyval	用于将来访问的关键字的值（整型）
OUT	extra_state	回调函数的外部状态

```
int MPI_Keyval_create(MPI_Copyfunction *copy_fn,MPI_Delete_function
*delete_fn ,int *keyval,void* extra_state)
MPI_KEYVAL_CREATE(COPY_FN,DELETE_FN,KEYVAL,EXTRA_STATE,IERROR)
      EXTERNAL  COPY_FN,DELETE_FN
INTEGER KEYVAL,EXTRA_STATE,IERROR
```

MPI调用接口 102 MPI_KEYVAL_CREATE

MPI_KEYVAL_CREATE创建一个新的属性关键字，根据返回的关键字，可以对特定的属性进行管理和操作。关键字在进程内是本地唯一的,而且对用户不透明,虽然它们显式地以整数方式存储，一旦被分配，关键字的值可以在任何本地定义的通信域上与属性建立联系并访问它们。

当用MPI_COMM_DUP复制一个通信域时,函数copy_fn被唤醒， copy_fn应是MPI_Copy_function类型,如下述定义:

typedef int MPI_Copy_function(MPI_Comm *oldcomm,int *keyval, void *extra_state,void *attribute_val_in, void **attribute_val_out,int *flag)

此函数的Fortran说明如下所示:

```
FUNCTION
COPY_FUNCTION(OLDCOMM,KEYVAL,EXTRA_STATE,ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT,FLAG)
        INTEGER  OLDCOMM,KEYVAL,EXTRA_STATE,ATTRIBUTE_VAL_IN,
         ATTRIBUTE_VAL_OUT
        LOGICAL  FLAG
```

oldcomm中对应于每个关键字值的复制回调函数可以任意顺序被唤醒。用一个关键字值和其对应的属性可调用每个复制回调函数。如果它返回flag=0，则将删除复制通信域中的属性；若flag=1，则设置为attribute_val_out返回的属性值。若成功，该函数返回MPI_SUCCESS，若失败则返回一个错误码(在此情况下，MPI_COMM_DUP将失败)。

copy_fn可以在C或FORTRAN中指定为MPI_NULL_COPY_FN或MPI_DUP_FN，MPI_NULL_COPY_FN只是返回flag=0和MPI_SUCCESS，不作其它任何事。MPI_DUP_FN设置flag=1，返回attribute_val_out中attribute_val_in的值和MPI_SUCCESS。

下面定义了一个同copy_fn类似的回调删除函数，当通过MPI_COMM_FREE释放一个通信域或显式调用MPI_ATTR_DELETE删除属性时，就会唤醒delete_fn函数。delete_fn具有MPI_Delete_function类型，其定义为：

首先是C的定义:

typedef int MPI_Delete_function(MPI_Comm *comm,int *keyval, void*attribute_val,void *extra_state)

Fortran说明如下所示:

```
FUNCTION DELETE_FUNCTION(COMM,KEYVAL,ATTRIBUTE_VAL,EXTRA_STATE)
        INTEGER  COMM,KEYVAL,ATTRIBUTE_VAL,EXTRA_STATE
```

MPI_COMM_FREE、MPI_ATTR_DELETE和MPI_ATTR_PUT调用此函数进行所需的删除属性的操作。在C或FORTRAN中可以将delete_fn 指定为MPI_NULL_DELETE_FN，它的作用就是返回一个MPI_SUCCESS。

```
MPI_KEYVAL_FREE(keyval)
IN          keyval          释放整数关键字值(整数)
int MPI_Keyval_free(int *keyval)
MPI_KEYVAL_FREE(KEYVAL,IERROR)
        INTEGER  KEYVAL,IERROR
```

MPI调用接口 103 MPI_KEYVAL_FREE

MPI_KEYVAL_FREE释放一个现存的属性关键字，此函数将keyval的值置为MPI_KEYVAL_INVALID。释放一个正在使用的属性关键字并不出错，因为实际释放发生在直到当对此关键字的所有引用都释放时才进行。

MPI_ATTR_PUT(comm,keyval,attribute_val)
IN comm 属性所依附的通信域(句柄)
IN keyval 关键字值，同MPI_KEY_CREATE的返回值(整数)
IN attribute_val 属性值
int MPI_Attr_put(MPI_Comm comm,int keyval,void* attribute_val)
MPI_ATTR_PUT(COMM,KEYVAL,ATTRIBUTE_VAL,IERROR)
INTEGER COMM,KEYVAL,ATTRIBUTE_VAL,IERROR

MPI调用接口 104 MPI_ATTR_PUT

MPI_ATTR_PUT调用设置指定关键字的属性值，该值可以被MPI_ATTR_GET取到。如果没有关键字具有值keyval，则调用出错。

MPI_ATTR_GET(comm,keyval,attribute_val,flag)
IN comm 属性所依附的通信域(句柄)
IN keyval 关键字值(整数)
OUT attribute_val 返回的属性值
OUT flag 是否有属性值的标志
int MPI_Attr_get(MPI_Comm comm,int keyval,void **attribute_val,int *flag)
MPI_ATTR_GET(COMM,KEYVAL,ATTRIBUTE_VAL,FLAG,IERROR)
INTEGER COMM,KEYVAL,ATTRIBUTE_VAL,IERROR LOGICAL FLAG

MPI调用接口 105 MPI_ATTR_GET

MPI_ATTR_GET通过关键字得到属性值，如果没有具有值keyval的关键字，则调用出错。如果关键字值存在,但没有对应于该关键字的属性附到comm上,调用也是正确的，在这种情况下,调用返回flag=false。如果既有关键字存在并且相应的属性值也存在，则flag=true并且找到的属性值放在attribute_val。

MPI_ATTR_DELETE(comm,keyval)
IN comm 属性所依附的通信域(句柄)
IN keyval 被删除属性的关键字值
int MPI_Attr_delete(MPI_Comm comm,int keyval)
MPI_ATTR_DELETE(COMM,KEYVAL,IERROR)
INTEGER COMM,KEYVAL,IERROR

MPI调用接口 106 MPI_ATTR_DELETE

MPI_ATTR_DELETE将给定关键字对应的属性值删除。它唤醒当keyval被创建时指定的删除函数delete_fn。如果指定的关键字不存在或者delete_fn调用出错，则本调用返回一个错

误代码，否则返回MPI_SUCCESS。

无论何时使用函数MPI_COMM_DUP复制一个通信域，都要唤醒所有的置属性值的回调复制函数，无论何时使用函数MPI_COMM_FREE删除一个通信域，都要唤醒所有的置属性值的回调删除函数。

下面的例子给出了关于属性的简单和基本的定义和使用方法。

```
      PROGRAM  MAIN
      include  'mpif.h'
       integer PM_MAX_TESTS
      parameter  (PM_MAX_TESTS=3)
       integer PM_TEST_INTEGER, fuzzy, Error, FazAttr
       integer PM_RANK_SELF
      integer  Faz_World
      parameter  (PM_TEST_INTEGER=12345)
      logical  FazFlag
      external  FazCreate,  FazDelete
       call  MPI_INIT(PM_GLOBAL_ERROR)

      PM_GLOBAL_ERROR  =  MPI_SUCCESS
       call  MPI_COMM_SIZE  (MPI_COMM_WORLD,PM_NUM_NODES,
     $                              PM_GLOBAL_ERROR)
       call  MPI_COMM_RANK  (MPI_COMM_WORLD,PM_RANK_SELF,
     $                              PM_GLOBAL_ERROR)
      call MPI_keyval_create  (  FazCreate,  FazDelete,  FazTag,
     &                             fuzzy,  Error  )
C       产生一个关键字值
      call  MPI_attr_get  (MPI_COMM_WORLD,  FazTag,  FazAttr,
     &                            FazFlag, Error)
C      获取该值的属性
       if (FazFlag ) then
C       若有属性则打印属性值
            print  *,  "True,get attr=",FazAttr
       else
C       若没有属性值则告诉用户
            print  *,  "False，   no attr"
       end  if
       FazAttr  = 120
      call  MPI_attr_put  (MPI_COMM_WORLD,  FazTag,  FazAttr,  Error)
C       设置新的属性值
      call  MPI_Comm_Dup  (MPI_COMM_WORLD,  Faz_WORLD,  Error)
C      对通信域进行复制
       call  MPI_Attr_Get  ( Faz_WORLD, FazTag, FazAttr,
     &                             FazFlag,  Error)
C       对新的通信域取属性值
       if (FazFlag) then
```

```
            print *, "True,dup comm get attr=",FazAttr
         else
            print *,"error"
         end if
         call MPI_Comm_free( Faz_WORLD, Error )
C        释放复制的通信域
         call MPI_FINALIZE (PM_GLOBAL_ERROR)
         end
C
C 定义创建属性关键字时的拷贝函数
C
         SUBROUTINE FazCreate (comm, keyval, fuzzy,
        &                      attr_in, attr_out, flag, ierr )
         INTEGER comm, keyval, fuzzy, attr_in, attr_out
         LOGICAL flag
         include 'mpif.h'
         attr_out = attr_in + 1
         flag = .true.
         ierr = MPI_SUCCESS
         END
C
C 定义创建属性关键字时的删除函数
C

         SUBROUTINE FazDelete (comm, keyval, attr, extra, ierr )
         INTEGER comm, keyval, attr, extra, ierr
         include 'mpif.h'
         ierr = MPI_SUCCESS
         if (keyval .ne. MPI_KEYVAL_INVALID)then
            attr = attr - 1
         end if
         END
```

程序 64 属性的简单使用方法

15.6 小结

进程组是通信域的组成部分，MPI的通信是在通信域的控制和维护下进行的，因此所有的MPI通信都直接或间接的用到通信域这一参数，通过对通信域的重组，可以以非常简单的方式方便地实现任务的划分，表达起来也很容易，因此对于高级的MPI程序设计一定要掌握进程组和通信域的管理。

第16章 具有虚拟进程拓扑的MPI程序设计

简单的MPI通信，不要求参加通信的进程具有特殊的拓扑结构，但是在一些应用中，对进程具有一定的拓扑具有很强的要求，定义不同的进程拓扑结构，可以使程序设计更自然，更易于理解，同时这样的逻辑拓扑也为在相近的物理拓扑上的高效实现提供支持。

本章介绍如何定义和使用不同的进程拓扑。主要包括两种拓扑：具有规则的网格形状的笛卡儿拓扑和具有任意形状的图拓扑。目前应用较多的是笛卡儿拓扑，因此本章的介绍以笛卡儿拓扑为主。

16.1 虚拟拓扑简介

在许多并行应用程序中，进程的线性排列不能充分地反映进程间在逻辑上的通信模型（通常由问题几何和所用的算法决定），进程经常被排列成二维或三维网格形式的拓扑模型，而且，通常用一个图来描述逻辑进程排列，在本章我们指这种逻辑进程排列为“虚拟拓扑”。拓扑是组内通信域上的额外、可选的属性，它不能附加在组间通信域(inter-communicator)上。拓扑能够提供一种方便的命名机制，对于有特定拓扑要求的算法使用起来直接、自然而方便，拓扑还可以辅助运行时系统，将进程映射到实际的硬件结构之上。

一个进程集合的通讯模型可以由一个图表示，结点代表进程，边用来连接彼此之间通信的进程。用图来说明虚拟拓扑，对于所有的应用是足够的。然而，在许多应用中图结构是规则的，而且详细的图的建立，对于用户是不方便的，在运行时可能缺乏有效性。并行应用程序中的大部分使用象环、二维或更高维的网格、圆环那样的进程拓扑。这些结构完全由在每一相应坐标方向的维数和进程数来定义，可以用简单方便的笛卡尔坐标来表示。

MPI提供两种拓扑，即笛卡儿拓扑和图拓扑，分别用来表示简单规则的拓扑和更通用的拓扑。

表格 16 笛卡儿拓扑和图拓扑调用的简单对比

操作	笛卡儿拓扑	图拓扑
创建	MPI_CART_CREATE	MPI_GRAPH_CREATE
得到维数	MPI_CARTDIM_GET	MPI_GRAPHDIMS_GET
得到拓扑信息	MPI_CART_GET	MPI_GRAPH_GET
物理映射	MPI_CART_MAP	MPI_GRAPH_MAP

16.2 笛卡儿拓扑

MPI_CART_CREATE用于描述任意维的笛卡尔结构。对于每一维，说明进程结构是否是周期性的。MPI_CART_CREATE返回一个指向新的通信域的句柄，这个句柄与笛卡尔拓扑信息相联系。如果reorder = false，那么在新的进程组中每一进程的标识数就与在旧进程组中的标识数相一致。否则，该调用会重新对进程编号。该调用得到一个ndims维的处理器阵列，每一维分别包含dims[0]，dims[1]，...，dims[ndims-1]个处理器。如果虚拟处理器阵列包含的总的处理器个数dims[1]*dims[1]*...*dims[ndims-1]小于旧的通信域comm_old包含的进程的个数，则有些进程返回的通信域为MPI_COMM_NULL，类似与MPI_COMM_SPLIT 的情

况，如果虚拟处理器阵列说明的处理器个数大于旧的通信域comm_old**包含的进程的个数，则该调用出错。**

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
IN comm_old **输入通信域（句柄）**
IN ndims **笛卡尔网格的维数（整数）**
IN dims **大小为**ndims**的整数数组，定义每一维的进程数（整数数组）**
IN periods **大小为**ndims**的逻辑数组，定义在一维上网格的周期性（逻辑数组）**
IN reorder **标识数是否可以重排序（逻辑型）**
OUT comm_cart **带有新的笛卡尔拓扑的通信域（句柄）**

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
                    int reorder, MPI_Comm *comm_cart)
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER,
                COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

MPI**调用接口** 107 MPI_CART_CREATE

MPI_DIMS_CREATE(nnodes, ndims,dims)
IN nnodes **网格中的结点数（整数）**
IN ndims **笛卡尔维数（整数）**
INOUT dims **大小为**ndims**的整数数组，定义每一维的结点数**

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
MPI_DIMS_CREATE(NNODE, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

MPI**调用接口** 108 MPI_DIMS_CREATE

MPI_DIMS_CREATE**根据用户指定的总维数**ndims**和总的进程数**nnodes**，帮助用户在每一维上选择进程的个数。返回结果放在**dims**中，它可以作为**MPI_CART_CREATE**的输入参数。但是用户也可以根据需要指定特定某一维i的进程数，比如置**dims[i]=k>0**，则本调用不会修改**dims[i]**的值，只有对于**dims[i]=0**的维，本调用才根据合适的划分算法给它重新赋值。本调用不允许**dims[i]**的初始值为负。**

MPI_TOPO_TEST**调用返回给定通信域进程的拓扑类型，输出值**STATUS**是下面之一**MPI_GRAPH**即图拓扑，**MPI_CART**即笛卡尔拓扑和**MPI_UNDEFINED**即没有定义拓扑。**

```
MPI_TOPO_TEST(comm, status)
        IN  comm      通信域（句柄）
        OUT status    通信域comm的拓扑类型(选择)
int MPI_Topo_test(MPI_Comm comm, int *status)
MPI_TOPO_TEST(COMM, STATUS, IERROR)
      INTEGER  COMM,  STATUS,  IERROR
```

MPI调用接口 109　MPI_TOPO_TEST

```
MPI_CART_GET(comm, maxdims, dims, periods, coords)
        IN  comm       带有笛卡尔结构的通信域（句柄）
        IN maxdims     最大维数（整数）
        OUT  dims      返回各维的进程数（整数数组）
        OUT  periods   返回各维的周期特性（逻辑数组）
        OUT  coords    调用进程的笛卡尔坐标（整数数组）
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords)
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
      INTEGER  COMM,  MAXDIMS,  DIMS(*),  COORDS(*),  IERROR
        LOGICAL PERIODS(*)
```

MPI调用接口 110　MPI_CART_GET

MPI_CART_GET返回给定通信域的拓扑信息，包括每一维的进程数dims，每一维的周期性periods和当前调用进程的笛卡儿坐标coords。

```
MPI_CART_RANK(comm, coords, rank)
        IN  comm      带有笛卡尔结构的通信域（句柄）
        IN  coords    卡氏坐标（整数数组）
        OUT  rank     卡氏坐标对应的一维线性坐标（整数）
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
      INTEGER  COMM,  COORDS(*),  RANK,  IERROR
```

MPI调用接口 111　MPI_CART_RANK

MPI_CART_RANK将给定拓扑的笛卡尔坐标转换成同一进程的用MPI_COMM_RANK调用得到的顺序编号。

```
MPI_CARTDIM_GET(comm, ndims)
        IN  comm      带有笛卡尔结构的通信域（句柄）
        OUT  ndims    笛卡尔网格的维数（整数）
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
      INTEGER  COMM,  NDIMS,  IERROR
```

MPI调用接口 112 MPI_CARTDIM_GET

MPI_CARTDIM_GET返回comm对应的笛卡尔结构的维数ndims。

```
MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)
        IN  comm              带有笛卡尔结构的通信域（句柄）
        IN direction          需要平移的坐标维（整数）
        IN disp               偏移量（整数）
        OUT  rank_source      源进程的卡氏坐标
        OUT  rank_dest        目标进程的卡氏坐标
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int
*rank_source, int *rank_dest)
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST,
                    IERROR)
    INTEGER  COMM,  DIRECTION,  DISP,  RANK_SOURCE,  RANK_DEST,  IERROR
```

MPI调用接口 113　MPI_CART_SHIFT

MPI_CART_SHIFT将有拓扑结构的通信域comm中的一个笛卡儿坐标rank_source，沿着指定的维direction，以偏移量disp进行平移，得到的是调用进程的笛卡儿坐标值，而调用进程的笛卡儿坐标经过同样的平移后，得到的是rank_dest。对于非周期性的拓扑，当超出范围后rank_source与rank_dest可以是MPI_PROC_NULL可以在中返回。

```
MPI_CART_COORDS(comm, rank, maxdims, coords)
        IN  comm          带有笛卡尔结构的通信域（句柄）
        IN rank           一维线性坐标（整数）
        IN maxdims        最大维数（整数）
        OUT  coords       返回该一维线性坐标对应的卡氏坐标
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER  COMM,  RANK,  MAXDIMS,  COORDS(*),  IERROR
```

MPI调用接口 114　MPI_CART_COORDS

MPI_CART_COORDS**将进程**rank**的顺序编号转换为笛卡儿坐标**coords，**其中**maxdims**是维数。**

MPI_CART_SUB(comm, remain_dims, newcomm)
IN comm **带有笛卡尔结构的通信域（句柄）**
IN remain_dims **定义保留的维（逻辑向量）**
OUT newcomm **包含子网格的通信域，这个子网格包含了调用进程（句柄）**
int MPI_Cart_sub(MPI_Comm com, int *remain_dims, MPI_Comm *newcomm)
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR
LOGICAL REMAIN_DIMS(*)

MPI**调用接口** 115 MPI_CART_SUB

MPI_CART_SUB**用于将通信域进行划分成不同的子通信域，**remain_dims**指出保留的维，若**remain_dims[i]**是**true**，则保留该维，若**remain_dims[i]**是**false**，则该维将划分为不同的通信域。若**comm**对应的拓扑网格为**2×3×4，**而**remain_dims=（false,true,true），**则本调用得到两个子通信域，它们在新旧通信域中笛卡儿坐标的对应关系为**

<1,1,1><1,1,2><1,1,3><1,1,4>
<1,2,1><1,2,2><1,2,3><1,2,4>
<1,3,1><1,3,2><1,3,3><1,3,4>

⟹

<1,1><1,2><1,3><1,4>
<2,1><2,2><2,3><2,4>
<3,1><3,2><3,3><3,4>

旧通信域 **新的子通信域**1

<2,1,1><2,1,2><2,1,3><2,1,4>
<2,2,1><2,2,2><2,2,3><2,2,4>
<2,3,1><2,3,2><2,3,3><2,3,4>

⟹

<1,1><1,2><1,3><1,4>
<2,1><2,2><2,3><2,4>
<3,1><3,2><3,3><3,4>

旧通信域 **新的子通信域**2

图 73 卡氏通信域的划分

MPI_CART_MAP（comm, ndims, dims, periods, newrank）

IN comm **输入通信域（句柄）**

IN ndims **笛卡尔结构的维数（整数）**

IN dims **大小为**ndims**的整数数组，定义每一维的进程数**

IN periods **大小为**ndims**的逻辑数组，定义每一维的周期性**

OUT newrank **调用进程优化后的坐标**

```
int MPI_Cart_map(MPI_comm comm, int ndims, int * dims, int * periods, int *newrank)
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)
```

MPI**调用接口** 116 MPI_CART_MAP

在由ndims**和**dims**构成的拓扑上**，MPI_CART_MAP**尽可能为当前进程计算一个优化的映射位置，并进行进程重排序，返回当前进程排序后的坐标**newrank。**若当前进程不在网格上，则返回**MPI_UNDEFINED。

下面的例子通过定义虚拟拓扑，实现数据的接力传送。

```
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size, false=0;
    int right_nbr, left_nbr;
    MPI_Comm    ring_comm;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Cart_create( MPI_COMM_WORLD, 1, &size, &false, 1, &ring_comm );/*创建一维网格，false意味着没有周期性（两端外的进程标识为MPI_PROC_NULL），1表示可以重排序，得到的拓扑坐标相邻关系为:

                MPI_PROC_NULL, 0, 1,  ...  , size-1 ,  MPI_PROC_NULL

*/
    MPI_Cart_shift( ring_comm, 0, 1, &left_nbr, &right_nbr );/*通过在定义的网格上的平移得到左右侧进程的标识*/
    MPI_Comm_rank( ring_comm, &rank );/*得到在新拓扑中的标识或坐标*/
```

```
        MPI_Comm_size( ring_comm, &size );/*得到新通信域中进程的个数*/
        do {
        if (rank == 0) {/*进程0负责读入数据并向下一个进程传递数据*/
       scanf( "%d", &value );
        MPI_Send( &value, 1, MPI_INT, right_nbr, 0, ring_comm );/*将数据传送到右面的进
程*/
    }
    else {
       MPI_Recv( &value, 1, MPI_INT, left_nbr, 0, ring_comm,
                &status );/*后面的进程从左边的进程接收数据*/
        MPI_Send( &value, 1, MPI_INT, right_nbr, 0, ring_comm );/*将接收到的数据在传递
给右面的进程*/
    }
    printf( "Process %d got %d\n", rank, value );/*各进程打印各自得到的数据*/
        } while (value >= 0);/*若读入的数据非负，则继续读入并传递*/

        MPI_Finalize( );
    }
```

程序 65 在具有虚拟拓扑的进程组上进行数据传递

16.3 图拓扑

MPI_GRAPH_CREATE返回一个指向新的通信域的句柄，这个通信域包含的进程的拓扑结构是一个由参数nnodes、index和edges定义的图。如果reorder = false, 那么在新进程组中每一进程的标识数就与在旧进程组中的标识数相一致。否则，会对进程赋以新的编号。如果图包含的结点数nnodes小于comm内进程的个数，那么有些进程返回MPI_COMM_NULL（类似于MPI_COMM_SPLIT）。若图的结点数大于comm内的进程总数，则该调用会出错。

图中总的结点数为nnodes-1，结点编号从0到nnodes-1。若用C来表示，则index[i]是结点0到结点i所有结点的度数之和，因此，0号结点的度数即为index[0],1号结点的度数为index[1]-index[0],i号结点的度数为index[i]-index[i-1]，i从1到nnodes-1。所有结点的边都按照结点编号的次序，从小到大，存放在边数组edges中。

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

IN comm_old **没有定义拓扑的通信域（句柄）**

IN nnodes **图中包含的结点数（整数）**

IN index **结点的度数（整数数组）**

IN edges **图的边数（整数数组）**

IN reorder **标识数是否可以重排序（逻辑型）**

OUT comm_graph **定义了图拓扑的通信域（句柄）**

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
                     int reorder, MPI_Comm *comm_graph)
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER,
                 COMM_GRAPH,IERROR)
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH,
            IERROR
    LOGICAL REORDER
```

MPI**调用接口** 117 MPI_GRAPH_CREATE

比如对于有四个结点的图，图中各结点的连接关系为：

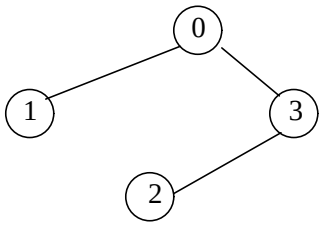


图 74 **简单的图拓扑**

表格 17 **结点、度数、边的对应关系**

结点编号	结点度数	该结点连接的其它结点
0	2	1，3
1	1	0
2	1	3
3	2	0，2

则参数nnodes, index**和**edges**的定义应该为：**

表格 18 **图拓扑的定义参数**

结点总数	结点的顺序累计度数	按结点顺序的边列表
nnodes = 4	index = 2, 3, 4, 6	edges = 1, 3, 0, 3, 0, 2

因此，在C语言中，index[0]**是结点0的度数**，index[i] - index[i-1] **是结点i的度数**，i=1，...，nnodes-1；**结点0的邻居列表存储在**edges[j]**中**，**其中**0≤j≤index[0]-1，**结点i的邻居列表**，i > 0，**存储在**edges[j]**中**，**其中**index[i-1]≤j≤index[i]-1。

```
MPI_GRAPHDIMS_GET(comm, nnodes, nedges)
IN comm          带有图结构的组通信域（句柄）
OUT nnodes       图中结点数（整数）
OUT nedges       图中边数（整数）
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
MPI_GRAPHDIMS_GET(COMM NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR
```

MPI**调用接口** 118　MPI_GRAPHDIMS_GET

MPI_GRAPHDIMS_GET**返回**comm**上定义的图的结点数**nnodes**和边数**nedges。

```
MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)
IN comm          带有图结构的通信域（句柄）
IN maxindex      index数组的大小（整数）
IN maxedges      edges数组的大小（整数）
OUT index        度数累计和数组（整数数组）
OUT edges        边列表数组（整数数组）
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges,
                  int *index,int *edges)
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*),EDGES(*), IERROR
```

MPI**调用接口** 119　MPI_GRAPH_GET

MPI_GRAPH_GET**返回给定通信域对应图的定义参数**index、edges，**其含义同创建图拓扑时一样。**

```
MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)
IN comm          带有图结构的通信域（句柄）
IN rank comm     组中一个进程的标识数（整数）
OUT nneighbors   指定进程的相邻结点数（整数）
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS,
                          IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
```

MPI**调用接口** 120　MPI_GRAPH_NEIGHBORS_COUNT

MPI_GRAPH_NEIGHBORS_COUNT**返回给定进程**rank**所连接边的个数**nneighbors。

```
MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)
IN  comm          带有图结构的通信域（句柄）
IN  rank  comm    特定进程的标识数（整数）
IN maxneighbors   矩阵neighbors的大小（整数）
OUT  neighbors    与指定进程相邻的进程的进程的标识数（整数数组）
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int maxneighbors,
                              int *neighbors)
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS,
                    NEIGHBORS, IERROR)
    INTEGER  COMM,  RANK,  MAXNEIGHBORS,  NEIGHBORS（*）, IERROR
```

MPI**调用接口** 121　MPI_GRAPH_NEIGHBORS

MPI_GRAPH_NEIGHBORS**返回给定进程**rank**所连接的边构成的数组**neighbors。

```
MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)
IN  comm        定义了图拓扑的通信域（句柄）
IN  nnodes      图中结点数（整数）
IN  index       结点度数数组（整型数组）
IN  edges       边数组（整型数组）
OUT  newrank    重排序后的进程标识
int MPI_Graph_map(MPI_Com comm, int nnodes, int *index, int *edges, int*newrank)
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER  COMM,  NNODES,  INDEX(*),  EDGES(*),  NEWRANK,  IERROR
```

MPI**调用接口** 122　MPI_GRAPH_MAP

MPI_GRAPH_MAP**调用如同**MPI_CART_MAP**一样，在由**nnodes、index**和**edges**构成的图上，由**MPI**尽可能为当前进程计算一个优化的映射位置，返回当前进程映射后的坐标**newrank。

16.4 再看Jacobi迭代的例子

在前面的部分，已经对 Jacobi**迭代进行的多次讨论，并给出了越来越简洁和易于表达的方式，这里通过使用**MPI**提供的向量数据类型和虚拟进程拓扑来重新实现**Jacobi**迭代。**

下面的例子用C**语言来实现，而不是**FORTRAN**。希望将待处理的数据（二维数组）同时按行和列来均匀分块，比较理想的情况是计算这一问题的并行计算机的处理器也是排列成二维的网格，这样就可以将不同的数据块交给不同的处理器去处理，这是一种自然和直观的并行化思路。**

通过虚拟进程拓扑，不管具体并行机的处理器物理上是如何排列的，都可以实现前面希望的并行计算模型。

为了讲解的简单与方便，设定只有四个处理器可以使用，划分成2×2的网格，每个处理器计算一块数据。假设整个数组的大小为256×256，则均匀划分后每一块的大小为128×128。数据块和处理器阵列的对应关系如图 75所示。对于更大的数据块或更多的处理器，这里介绍的方法同样适用。由此也可以看出采用虚拟进程拓扑后编写出来程序的通用性与可移植性。

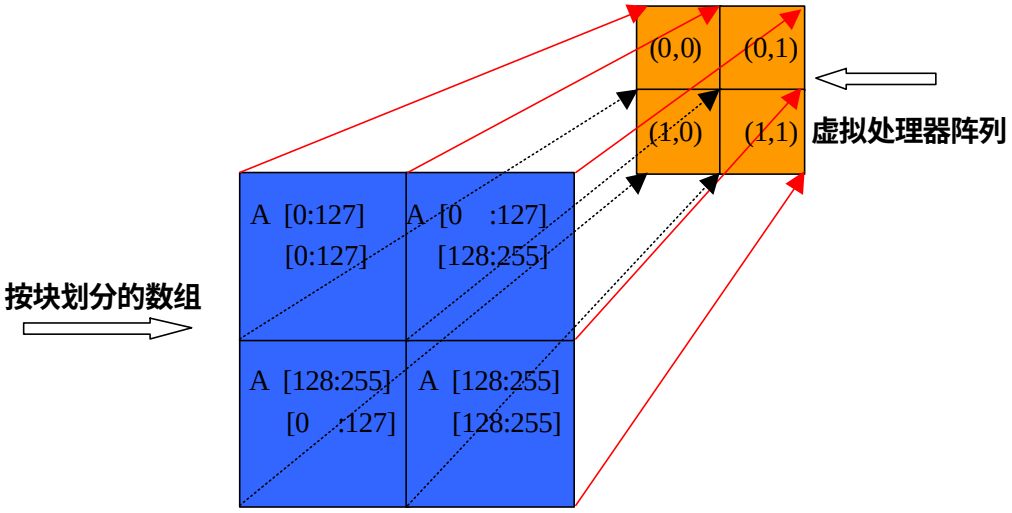


图 75 分块数组向虚拟处理器阵列的映射

注意按行列同时划分时通信形式的变化，按行（或列）划分数组时，分块数组只需和上下（或左右）的邻居通信，按行列同时划分时，分块数组需要和上下、左右的邻居同时通信。为此，分块数组的上下左右都预留出需要通信的部分，用来存放同各个方向邻居通信得到的数据（图 76）。

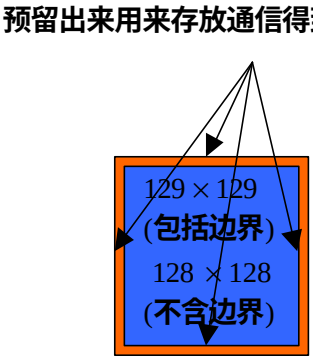


图 76 各处理器上声明的包含通信边界的局部数组

从整体上看，处理器阵列上的通信有四个不同的方向，1 从左向右；2 从右向左；3 从上到下；4 从下到上。具体到每一个处理器，与处理器阵列外的通信方向可以排除。如果处理器个数更多，比如32×32，则处于处理器阵列中间的处理器，其四个方向的通信都需要。

具体到不同的通信，当进行垂直方向的通信时（对一行的数据进行通信），由于C语言定义的数组在内存中是行优先排列，因此一行的数据是连续存放的，可以直接进行；当进行水平方向的通信时（对一列的数据进行通信），这时一列数据是不连续的，因此需要定义新的数据类型来表示一列的数据。在下面介绍的例子中，同时使用了派生数据类型和虚拟进程拓扑。

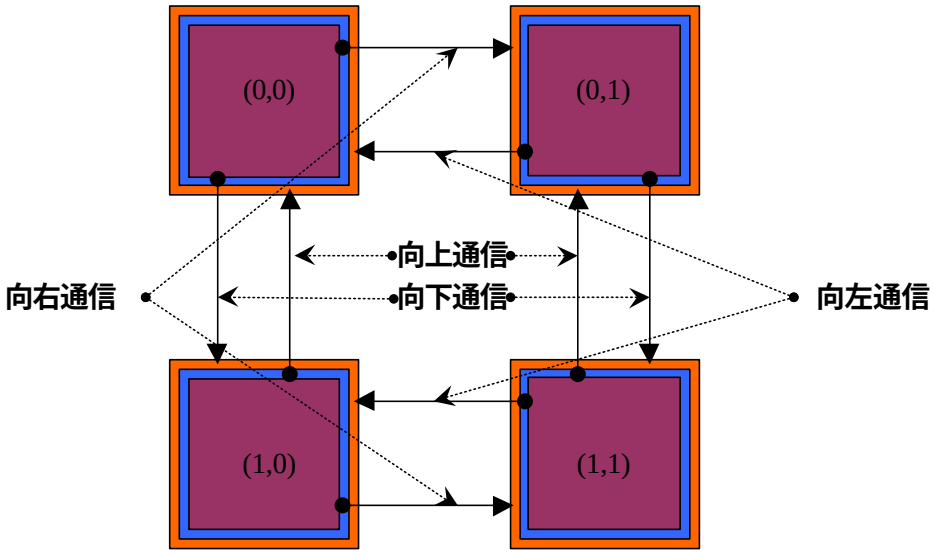


图 77 二维网格上各处理器之间的通信关系

```
#include "mpi.h"
#define arysize 256
#define arysize2 (arysize/2)

int main(int argc, char *argv[])
{
   int n, myid, numprocs, i, j, nsteps=10;
    float a[arysize2+2][arysize2+2],b[arysize2+2][arysize2+2];/*定义局部数组的大小，包含边界空间*/
    double starttime,endtime;
    int col_tag,row_tag,send_col,send_row,recv_col,recv_row;
    int col_neighbor,row_neighbor;
    MPI_Comm   comm2d;
    MPI_Datatype   newtype;
    int   right,left,down,top,top_bound,left_bound,down_bound,right_bound;
    int periods[2];
    int dims[2],begin_row,end_row;
    MPI_Status status;
    MPI_Init(&argc,&argv);
   dims[0] = 2;
   dims[1] = 2;
    periods[0]=0;
    periods[1]=0;
    MPI_Cart_create( MPI_COMM_WORLD, 2, dims, periods, 0,&comm2d);/*定义虚拟进程拓扑，它是一个2×2的网格，得到的包含进程拓扑信息的新的通信域是comm2d*/
```

```
 MPI_Comm_rank(comm2d,&myid);
 MPI_Type_vector( arysize2, 1, arysize2+2,MPI_FLOAT,&newtype);/*定义向量数据类型来表示一列*/
 MPI_Type_commit( &newtype );/*新类型的递交*/
 MPI_Cart_shift( comm2d, 0,  1, &left, &right);/*得到当前进程左右两侧的进程标识*/
 MPI_Cart_shift( comm2d, 1,  1, &down, &top);/*  得到当前进程上下方的进程标识*/

  /*下面的程序为数组赋初值*/
 for(i=0;i<arysize2+2;i++) for(j=0;j<arysize2+2;j++) a[i][j]=0.0;
 if (top == MPI_PROC_NULL)
  {
    for ( i=0;i<arysize2+2;i++) a[1][i]=8.0;
  }
 if (down == MPI_PROC_NULL)
  {
    for ( i=0;i<arysize2+2;i++) a[arysize2][i]=8.0;
  }
 if (left == MPI_PROC_NULL)
  {
    for ( i=0;i<arysize2+2;i++) a[i][1]=8.0;
  }
 if (right == MPI_PROC_NULL)
  {
    for ( i=0;i<arysize2+2;i++) a[i][arysize2]=8.0;
  }
 col_tag = 5; row_tag = 6;
  printf("Laplace  Jacobi#C(BLOCK,BLOCK)#myid=%d#step=%d#total
arysize=%d*%d\n",myid,nsteps,arysize,arysize);

  top_bound=1;
  left_bound=1;
  down_bound=arysize2;
  right_bound=arysize2;
 if (top == MPI_PROC_NULL) top_bound=2;
 if (left == MPI_PROC_NULL) left_bound=2;
 if (down == MPI_PROC_NULL) down_bound=arysize2-1;
 if (right == MPI_PROC_NULL) left_bound= arysize2-1;

  starttime=MPI_Wtime();
 for (n=0; n<nsteps; n++) {

     MPI_Sendrecv( &a[1][1], arysize2, MPI_FLOAT, top, row_tag,& a[arysize2+1][1],
arysize2, MPI_FLOAT, down, row_tag, comm2d, &status );/*向上数据传送*/
```

```
        MPI_Sendrecv( &a[arysize2][1], arysize2, MPI_FLOAT, down, row_tag,& a[0][1],
arysize2, MPI_FLOAT, top, row_tag, comm2d, &status );/*向下数据传送*/

        MPI_Sendrecv( &a[1][1], 1,newtype, left, col_tag,& a[1][arysize2+1], 1, newtype,
right, col_tag, comm2d, &status );/*向左数据传送*/

        MPI_Sendrecv( &a[1][arysize2], 1, newtype, right, col_tag, &a[1][0], 1, newtype, left,
col_tag, comm2d, &status );/*向右数据传送*/

       for ( i=left_bound;i<right_bound;i++) for (j=top_bound;j<down_bound;j++)
            b[i][j] = (a[i][j+1]+a[i][j-1]+ a[i+1][j]+a[i-1][j])*0.25;

       for ( i=left_bound;i<right_bound;i++) for (j=top_bound;j<down_bound;j++)
            a[i][j] = b[i][j];
    }
    endtime=MPI_Wtime();
   printf("elapse  time=%f\n",endtime-starttime);
   MPI_Type_free( &newtype );
    MPI_Comm_free( &comm2d );
    MPI_Finalize();
}
```

程序 66 用虚拟进程拓扑和向量数据类型来实现Jacobi迭代

16.5 小结

当具体的应用或算法对进程的排列有特定的要求时，使用虚拟进程拓扑是一种方便的手段。同时，对于特定的并行计算机，如果其处理器之间的网络连接方式和MPI程序规定的拓扑排列方式相吻合的话，就很有可能充分利用硬件的特定来达到高效率。因此，虚拟进程拓扑是一种重要的并行程序设计概念，使用它可以简化设计，提高效率。

第17章 MPI对错误的处理

MPI调用会返回一个结果，从它可以直到该调用的执行是否正确，当MPI程序在发生错误的时候，会调用相应的例程进行处理。本章MPI程序与错误处理有关的调用。

17.1 与错误处理有关的调用

MPI_ERRHANDLER_CREATE (function, errhandler)
IN function 用户定义的错误处理函数
OUT errhandler 返回的MPI错误句柄（句柄）
int MPI_Errhandler_create (MPI_Handler_function *function, MPI_Errhandler *errhandler)
MPI_ERRHANDLER_CREATE (FUNCTION, ERRHANDLER, IERROR)
EXTERNAL FUNCTION
INTEGER ERRHANDLER, IERROR

MPI调用接口 123 MPI_ERRHANDLER_CREATE

MPI_ERRHANDLER_CREATE将用户例程function向MPI注册，作为一个MPI异常句柄，而返回的errhandler是指向该注册例程的句柄。

用户例程应该是类型为MPI_Handler_function的函数，在C语言中定义如下：

typedef void (MPI_Handler_function) (MPI_Comm *, int *, …)

第一个参数是所用的通信域。第二个是由MPI例程所返回的错误代码。剩下的参数依赖于具体的实现。

MPI_ERRHANDLER_SET (comm, errhandler)
IN comm 设置错误句柄的通信域（句柄）
IN errhandler 新的MPI错误句柄（句柄）
int MPI_Errhandler_set (MPI_Comm comm, MPI_Errhandler errhandler)
MPI_ERRHANDLER_SET (COMM, ERRHANDLER, IERROR)
INTEGER COMM, ERRHANDLER, IERROR

MPI调用接口 124 MPI_ERRHANDLER_SET

MPI_ERRHANDLER_SET将一个指定的错误句柄errhandler与给定的通信域comm相联系。

MPI_ERRHANDLER_GET(comm, errhandler)
IN comm **获取错误句柄的通信域（句柄）**
OUT errhandler **返回与通信域相连的**MPI**错误句柄（句柄）**
int MPI_Errhandler_get (MPI_Comm comm, MPI_Errhandler *errhandler)
MPI_ERRHANDLER_GET (COMM, ERRHANDLER, IERROR)
INTEGER COMM, ERRHANDLER, IERROR

MPI**调用接口** 125 MPI_ERRHANDLER_GET

MPI_ERRHANDLER_GET**返回与通信域**comm**相联系的错误句柄**errhandler。

MPI_ERRHANDLER_FREE (errhandler)
IN errhandler MPI**错误句柄（句柄）**
int MPI_Errhandler_free (MPI_Errhandler *errhandler)
MPI_ERRHANDLER_FREE (ERRHANDLER, IERROR)
INTEGER ERRHANDLER, IERROR

MPI**调用接口** 126 MPI_ERRHANDLER_FREE

MPI_ERRHANDLER_FREE**不仅释放掉给定的错误句柄**errhandler，**并且置**errhandler**为**MPI_ERRHANDLERNULL。**但真正的释放操作是在所有与其相连的通信域都释放后才进行的。**

MPI_ERROR_STRING (errorcode, string, resultlen)
IN errorcode **由**MPI**例程返回的错误码**
OUT string **相应于**errorcode**的字符串**
OUT resultlen **在**string**中所返回字符串的长度**
int MPI_Error_string (int errorcode, char *string, int *resultlen)
MPI_ERROR_STRING (ERRORCODE, STRING, RESULTLEN, IERROR)
INTEGER ERRORCODE, RESULTLEN, IERROR
CHARACTER *(*) STRING

MPI**调用接口** 127 MPI_ERROR_STRING

MPI_ERROR_STRING**返回与一错误代码码相联系的错误字符串**string。**调用前必须首先为参数**string**申请至少**MPI_MAX_ERROR_STRING**字符长的存储空间。**

MPI_ERROR_CLASS (errorcode, errorclass)
IN errorcode **由**MPI**例程返回的错误码**
OUT errorclass **相应于**errorcode**的错误类**
int MPI_Error_class (int errorcode, int *errorclass)
MPI_ERROR_CLASS (ERRORCODE, ERRORCLASS, IERROR)
INTEGER ERRORCODE, ERRORCLASS, IERROR

MPI**调用接口** 128 MPI_ERROR_CLASS

MPI_ERROR_CLASS**将错误码进行解释，转换为相应的错误类，有效的错误类包括如下表格所示。**

表格 19 **错误类列表**

错误类编码	**错误类含义**
MPI_SUCCESS	**无错误**
MPI_ERR_BUFFER	**无效缓冲区指针**
MPI_ERR_COUNT	**无效计数参数**
MPI_ERR_TYPE	**无效数据类型参数**
MPI_ERR_ TAG	**无效标识参数**
MPI_ERR_COMM	**无效通信域**
MPI_ERR_RANK	**无效标识数**
MPI_ERR_REQUEST	**无效请求（句柄）**
MPI_ERR_ROOT	**无效根**
MPI_ERR_GROUP	**无效组**
MPI_ERR_OP	**无效操作**
MPI_ERR_TOPOLOGY	**无效拓扑**
MPI_ERR_DIMS	**无效维参数**
MPI_ERR_ARG	**其它无效种类参数**
MPI_ERR_UNKNOWN	**不知道原因的错误**
MPI_ERR_TRUNCATE	**接受被截断的消息**
MPI_ERR_OTHER	**其它的错误**
MPI_ERR_INTERN	**内部**MPI**错误**
MPI_ERR_LASTCODE	**最后标准错误码**

具体的实现可以自由的定义更多的错误类；然而，应在合适的地方使用标准错误类。错误类满足如下关系：

0 = MPI_SUCCESS＜MPI_ERR_…≤MPI_ERR_LASTCODE

17.2 小结

当我们编写的程序出现错误时，利用MPI**提供的错误处理设施和错误代码，有助于迅速将错误的性质进行定位，不至于无从下手。**

第18章 MPI函数调用原型列表与简单解释

本章按字母顺序列出所有的MPI函数调用，并给出简单/扼要的介绍，以便读者查找和使用。

18.1 MPI-1与C语言的接口

int MPI_Abort(MPI_Comm comm, int errorcode)

终止MPI环境及MPI程序的执行

int MPI_Address(void * location, MPI_Aint * address)

得到给定位置在内存中的地址，将被废弃的函数，建议用MPI_Get_address取代

int MPI_Allgather(void * sendbuff, int sendcount, MPI_Datatype sendtype, void * recvbuf, int * recvcounts, int * displs, MPI_Datatype recvtype, MPI_Comm comm)

每一进程都从所有其它进程收集数据，相当于所有进程都执行了一个MPI_Gather调用。

int MPI_Allgatherv(void * sendbuff, int sendcount, MPI_Datatype sendtype, void * recvbuf, int recvcounts, int * displs, MPI_Datatype recvtype, MPI_Comm comm)

所有进程都收集数据到指定的位置，就如同每一个进程都执行了一个MPI_Gatherv调用

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

归约所有进程的计算结果，并将最终的结果传递给所有其它的进程，相当于每一个进程都执行了一次MPI_Reduce调用。

int MPI_Alltoall(void * sendbuf, void * recvbuf, int count, MPI_Datatype datatype, void * recvbuf, int * recvcounts, int * rdispls, MPI_Datatype recvtype, MPI_Comm comm)

所有进程相互交换数据

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

所有进程相互交换数据，但数据有一个偏移量。

Int MPI_Attr_delete(MPI_Comm comm, int keyval)

删除与指定关键词联系的属性值。即将废弃的特性，建议用MPI_Comm_delete_attr替代

int MPI_Attr_get(MPI_Comm comm, int keyval, void * attribute_val, int * flag)

按关键词查找属性值，即将废弃的特性，建议用MPI_Comm_get_attr替代

int MPI_Attr_put(MPI_Comm comm, int keyval, void * attribute_val)

按关键词设置属性值。即将废弃的特性，建议用MPI_Comm_set_attr替代

int MPI_Barrier(MPI_Comm comm)

等待直到所有的进程都执行到这一例程才继续执行下一条语句。

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

将root进程的消息广播到所有其它的进程

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

使用用户声明的缓冲区进行发送

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

建立发送缓冲句柄

int MPI_Buffer_attach(void * buffer, int size)

将一个用户指定的缓冲区用于消息发送的目的

int MPI_Buffer_detach(void * buffer, int * size)

移走一个指定的发送缓冲区

int MPI_Cancel(MPI_Request * request)

取消一个通信请求

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int * coords)

给出一个进程所在组的标识号，得到其卡氏坐标值

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int * dims, int * periods, int reorder, MPI_Comm * comm_cart)

按给定的拓扑创建一个新的通信域

int MPI_Cart_get(MPI_Comm comm, int maxdims, int * dims, int *periods, int * coords)

得到给定通信域的卡氏拓扑信息

int MPI_Cart_map(MPI_Comm comm, int * ndims, int * periods, int * newrank)

将进程标识号映射为卡氏拓扑坐标

int MPI_Cart_rank(MPI_Comm comm, int * coords, int * rank)

由进程标识号得到卡氏坐标

int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int * rank_source, int * rank_dest)

给定进程标识号、平移方向与大小，得到相对于当前进程的源和目的进程的标识号

int MPI_Cart_sub(MPI_Comm comm, int * remain_dims, MPI_Comm * newcomm)

将一个通信域，保留给定的维，得到子通信域

int MPI_Cartdim_get(MPI_Comm comm, int* ndims)

得到给定通信域的卡氏拓扑

int MPI_Comm_compare(MPI_comm comm1, MPI_Comm comm2, int * result)

两个通信域的比较

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm * newcomm)

根据进程组创建新的通信域

Int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *new_comm)

通信域复制

int MPI_Comm_free(MPI_Comm* comm)

释放一个通信域对象

int MPI_Comm_group(MPI_Comm comm, MPI_Group * group)

由给定的通信域得到组信息

int MPI_Comm_rank(MPI_Comm comm, int * rank)

得到调用进程在给定通信域中的进程标识号

int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group * group)

得到组间通信域的远程组

int MPI_Comm_remote_size(MPI_Comm comm, int * size)

得到远程组的进程数

int MPI_Comm_set_attr(MPI_Comm comm, int keyval, void * attribute_val)

根据关键词保存属性值

int MPI_Comm_size(MPI_Comm comm, int * size)

得到通信域组的大小

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm * newcomm)

按照给定的颜色和关键词创建新的通信域

int MPI_Comm_test_inter(MPI_Comm comm, int * flag)

测试给定通信域是否是域间域

int MPI_Dims_create(int nnodes, int ndims, int * dims)

在卡氏网格中建立进程维的划分

int MPI_Errhandler_create(MPI_handler_function * function, MPI_Errhandler * errhandler)

创建MPI**错误句柄。过时特性，建议用**MPI_Comm_create_errhandler**替代**

int MPI_Errhandler_free(MPI_Errhandler * errhandler)

释放MPI**错误句柄**

int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler * errhandler)

得到给定通信域的错误句柄，即将废弃的特性，建议用MPI_Comm_get_errhandler**代替**

int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)

设置MPI**错误句柄，即将废弃的特性，建议用**MPI_Comm_set_errhandler**代替**

int MPI_Error_class(int errorcode, int * errorclass)

将错误代码转换为错误类

int MPI_Error_string(int errorcode, char * string, int * resultlen)

由给定的错误代码，返回它所对应的字符串

int MPI_Finalize(void)

结束MPI**运行环境**

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

从进程组中收集消息

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

从进程组中收集消息到指定的位置

int MPI_Get_count(MPI_Status * status, MPI_Datatype datatype, int * count)

得到以给定数据类型为单位的数据的个数

int MPI_Get_elements(MPI_Statue * status, MPI_Datatype datatype, int * elements)

返回给定数据类型中基本元素的个数

int MPI_Get_processor_name(char * name, int * resultlen)

得到处理器名称

int MPI_Get_version(int * version, int * subversion)

返回MPI**的版本号**

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int * index, int * edges, int reorder, MPI_Comm * comm_graph)

按照给定的拓扑创建新的通信域

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int * index, int * edges)

得到给定通信域的处理器拓扑结构

int MPI_Graph_map(MPI_Comm comm, int nnodes, int * index, int * edges, int * newrank)

将进程映射到给定的拓扑

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int * nneighbors)

给定拓扑，返回给定结点的相邻结点数

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int * maxneighbors, int * neighbors)

给定拓扑，返回给定结点的相邻结点

int MPI_Graphdims_Get(MPI_Comm comm, int * nnodes, int * nedges)

得到给定通信域的图拓扑

int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int * result)

比较两个组

int MPI_Group_diffenence(MPI_Group group1, MPI_Group group2, MPI_Group * newgroup)

根据两个组的差异创建一个新组

int MPI_Group_excl(MPI_Group group, int n, int * ranks, MPI_Group * newgroup)

通过重新对一个已经存在的组进行排序，根据未列出的成员创建一个新组

int MPI_Group_free(MPI_Group * group)

释放一个组

int MPI_Group_incl(MPI_Group group, int n, int * ranks, MPI_Group * newgroup)

通过重新对一个已经存在的组进行排序，根据列出的成员创建一个新组

int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group * newgroup)

根据两个已存在组的交创建一个新组

int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_group * newgroup)

根据已存在的组，去掉指定的部分，创建一个新组

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group * newgroup)

根据已存在的组，按照指定的部分，创建一个新组

int MPI_Group_rank(MPI_Group group, int * rank)

返回调用进程在给定组中的进程标识号

int MPI_Group_size(MPI_Group group, int * size)

返回给定组的大小

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

将一个组中的进程标识号转换成另一个组的进程标识号

int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group * newgroup)

将两个组合并为一个新组

int MPI_Ibsend(void * buf, int count, MPI_Datatype datatype, int dest, int tga, MPI_Comm comm, MPI_Request * request)

非阻塞缓冲区发送

int MPI_Init(int * argc, char *** argv)

MPI**执行环境初始化**

Int MPI_Initialized(int * flag)

查询MPI_Init**是否已经调用**

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

根据两个组内通信域创建一个组间通信域

int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm * newintracomm)

根据组间通信域创建一个组内通信域

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int * flag, MPI_Status * status)

非阻塞消息到达与否的测试

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

非阻塞接收

int MPI_Irsend(viud * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request * request)

非阻塞就绪发送

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

非阻塞发送

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

非阻塞同步发送

int MPI_Keyval_create(MPI_Copy_function * copy_fn, MPI_Delete_function * delete_fn, int * keyval, void * extra_state)

创建一个新的属性关键词，即将废弃的特性，建议用MPI_Comm_create_keyval**代替**

int MPI_Keyval_free(int * keyval)

释放一个属性关键词

int MPI_Op_create(MPI_Uop function, int commute, MPI_Op * op)

创建一个用户定义的通信函数句柄

int MPI_Op_free(MPI_Op * op)

释放一个用户定义的通信函数句柄

int MPI_Pack(void * inbuf, int incount, MPI_Datatype datetype, void * outbuf, int outcount, int * position, MPI_Comm comm)

将数据打包，放到一个连续的缓冲区中

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int * size)

返回需要打包的数据类型的大小

int MPI_Pcontrol(const int level)

控制剖视

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status * status)

阻塞消息测试

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

标准接收

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

创建接收句柄

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

将所进程的值归约到root**进程，得到一个结果**

int MPI_Reduce_scatter(void * sendbuf, void * recvbuf, int * recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

将结果归约后再发送出去

int MPI_Request_free(MPI_Request * request)

释放通信申请对象

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

就绪发送

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

创建就绪发送句柄

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

在给定的进程集合上进行扫描操作

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

将数据从一个进程发送到组中其它进程

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

将缓冲区中指定部分的数据从一个进程发送到组中其它进程

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

标准的数据发送

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

创建一个标准发送的句柄

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)

同时完成发送和接收操作

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)

用同一个发送和接收缓冲区进行发送和接收操作

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

同步发送

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

创建一个同步发送句柄

int MPI_Start(MPI_Request * request)

启动给定对象上的重复通信请求

int MPI_Startall(int count, MPI_Request * array_of_requests)

启动指定的所有重复通信请求

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

测试发送或接收是否完成

int MPI_Testall(int count, MPI_Request * array_of_requests, int * flag, MPI_Status * array_of_statuses)

测试前面所有的通信是否完成

int MPI_Testany(int count, MPI_Request * array_of_requests, int * index, int * flag, MPI_Status * status)

测试前面任何一个通信是否完成

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

测试是否有一些通信已经完成

int MPI_Test_cancelled(MPI_Status * status, int * flag)

测试一个请求对象是否已经删除

int MPI_Topo_test(MPI_Comm comm, int * top_type)

测试指定通信域的拓扑类型

int MPI_Type_commit(MPI_Datatype * datatype)

提交一个类型

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype * newtype)

创建一个连续的数据类型

int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint * extent)

返回一个数据类型的范围，即将废弃的特性，建议使用MPI_type_get_extent**来代替**

int MPI_Type_free(MPI_Datatype * datatype)

释放一个数据类型

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

按照字节偏移，创建一个数据类型索引，即将废弃的特性，建议使用MPI_type_create_hindexed**来代替**

int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype * newtype)

根据以字节为单位的偏移量，创建一个向量数据类型，即将废弃的特性，建议使用MPI_type_create_hvector**来代替**

int MPI_Type_indexed(int cont, int * array_of_blocklengths, int * array_of_displacements, MPI_Datatype oldtype, MPI_Datatype * newtype)

创建一个索引数据类型

int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint * displacement)

返回指定数据类型的下边界，即将废弃的特性，建议使用MPI_type_get_extent**来代替**

int MPI_Type_size(MPI_Datatype datatype, int * size)

以字节为单位，返回给定数据类型的大小

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

创建一个结构数据类型，即将废弃的特性，建议使用MPI_type_create_struct**来代替**

int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint * displacement)

返回指定数据类型的上边界，即将废弃的特性，建议使用MPI_type_get_extent**来代替**

int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype * newtype)

创建一个向量数据类型

int MPI_Unpack(void * inbuf, int insize, int * position, void * outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

从连续的缓冲区中将数据解开

int MPI_Wait(MPI_Request * request, MPI_Status * status)

等待MPI**的发送或接收语句结束**

int MPI_Waitall(int count, MPI_Request * array_of_requests, MPI_Status * array_of_status)

等待所有给定的通信结束

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,MPI_Status *status)

等待某些指定的发送或接收完成

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

等待一些给定的通信结束

double MPI_Wtick(void)

返回MPI_Wtime**的分辨率**

double MPI_Wtime(void)

返回调用进程的流逝时间

18.2 MPI-1与Fortran语言的接口

MPI_Abort(comm, errorcode, ierror)

integer comm, errorcode, ierror

终止MPI**环境及**MPI**程序的执行**

MPI_Address(location, address, eerror)

<type>location

integer address, ierror

得到给定位置在内存中的地址，将被废弃的函数，建议用MPI_Get_address**取代**

MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer sendcount, sendtype, recvcount, recvtype, comm, ierror

相当于每一个进程都执行了一次MPI_Gather**操作，即每一个进程都从其它的进程收集消息。**

MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recbcounts, displs, recvtype, comm, ierror)

<type>sendbuf(*), recvbuf(*)

integer sendcount, sendtype, recvcounts(*), displs(*), recvtype, comm, ierror

相当于每一个进程都执行了一次MPI_Gatherv**操作，即每一个进程都从其它的进程收集消息到指定的位置，接收数据存放的位置用一个数组来指定。**

MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer count, datatype, op, comm, ierror

所有进程都执行归约操作，相当于每一个进程都执行了一次MPI_Reduce

MPI_Alltoall(sendbuf, sendcount, sendtyupe, recvbuf, recvcount, recvtype, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer sendcount, sendtype, recvcount, recvtype, comm, ierror

所有进程相互交换数据

MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer sendcounts(*), sdispls(*), sendtype, recvcounts(*), rdispls(*), recvtype, comm, ierror

所有进程相互交换数据，但数据有一个偏移量。

MPI_Attr_delete(comm, keyval, ierror)

integer comm, keyval, ierror

删除与指定关键词联系的属性值。即将废弃的特性，建议用MPI_Comm_delete_attr**替代**

MPI_Attr_get(comm, keyval, attribute_val, flag, ierror)

integer comm, keyval, attribute_val, ierror

Logical flag

按关键词查找属性值，即将废弃的特性，建议用MPI_Comm_get_attr**替代**

MPI_Attr_put(comm, keyval, attribute_val, ierror)

integer comm, keyval, attribute_val, ierror

按关键词设置属性值。即将废弃的特性，建议用MPI_Comm_set_attr**替代**

MPI_Barrier(comm, ierror)

integer comm, ierror

等待直到所有的进程都执行到这一例程才继续执行下一条语句。

MPI_Bcast(buffer, count, datatype, root, comm, ierror)

<type> buffer(*)

integer count, datatype, root, comm, ierror

将root**进程的消息广播到所有其它的进程**

MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)

<type> buf(*)

integer cont, datatype, dest, tag, comm, ierror

使用用户声明的缓冲进行发送

MPI_Bsend_init(buf, cont, datatype, dest, tag, comm, request, ierror)

<type> buf(*)

integer count, datatype, dest, tag, comm, request, ierror

建立缓存发送句柄

MPI_Biffer_attch(buffer, size, ierror)

<type> buffer(*)

integer size, ierror

将一个用户指定的缓冲区用于消息发送的目的

MPI_Biffer_detach(buffer, size, ierror)

<type> buffer(*)

integer size, ierror

移走一个指定的发送缓冲区

MPI_Cancel(request, ierror)

integer request, ierror

取消一个通信请求

MPI_Cart_coords(comm, rank, maxdims, coords, ierror)

integer comm, rank , maxdims, coords(*), ierror

给出一个进程所在组的标识号，得到其卡氏坐标值

MPI_Cart_creat(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)

integer comm_old, ndims, dims(*), comm_cart, ierror

Logical periods(*), reorder

按给定的拓扑创建一个新的通信域

MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)

integer comm, maxdims, dims(*), coords(*), ierror
Logical periods(*)

得到给定通信域的卡氏拓扑信息

MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
integer comm, ndims, dims(*), newrank, ierror
Logical periods(*)

将进程向指定的拓扑进行映射

MPI_Cart_rank(comm, coords, rank, ierror)
integer comm, coords(*), rank, ierror

由卡氏坐标得到进程标识号

MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror

给定进程标识号、平移方向与大小，得到相对于当前进程的源和目的进程的标识号

MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
integer comm, newcomm, ierror
Logical remain_dims(*)

将一个通信域，保留给定的维，得到子通信域

MPI_Cartdim_get(comm, ndism, ierror)
integer comm, ndims, ierror

得到给定通信域的卡氏拓扑

MPI_Comm_compare(comm1, comm2, result, ierror)
integer comm, group, newcomm, ierror

两个通信域的比较

MPI_Comm_creat(comm, group, newcomm, ierror)
integer comm, group, newcomm, ierror

创建新的通信域

MPI_Comm_dup(comm, newcomm, ierror)
integer comm, newcomm, ierror

通信域复制

MPI_Comm_free(comm, ierror)
integer comm, ierror

撤消一个通信域对象

MPI_Comm_group(comm, group, ierror)
integer comm, group, ierror

由给定的通信域得到组信息

MPI_Comm_rank(comm, rank, ierror)
integer comm, rank, ierror

得到调用进程在给定通信域中的进程标识号

MPI_comm_remote_group(comm, group, ierror)
integer comm, group, ierror

得到组间通信域的远程组

MPI_comm_remote_size(comm, size, ierror)
integer comm, size, ierror

得到远程组的进程数

MPI_Comm_set_attr(comm, keyval, attribute_val, ierror)

integer comm, keyval, ierror
integer (kind=MPI_ADDRESS_KIND) attribute_val

根据关键词保存属性值

MPI_Comm_size(comm, size, ierror)
integer comm, size, ierror

得到通信域组的大小

MPI_Comm_split(comm, color, key, newcomm, ierror)
integer comm, color, key, newcomm, ierror

按照给定的颜色和关键词创建新的通信域

MPI_Comm_test_inter(comm, flag, ierror)
integer comm, ierror
Logical flag

测试给定通信域是否是域间域

MPI_Dims_create(nnodes, ndims, dims, ierror)
integer nnodes, ndims, dims(*), ierror

在卡氏网格中建立进程维的划分

MPI_Errhandler_create(function, errhandler, ierror)
External function
integer errhandler, ierror

创建MPI**错误句柄。过时特性，建议用**MPI_Comm_create_errhandler**替代**

MPI_Errhandler_free(comm, errhandler, ierror)
integer comm, errhandler, ierror

释放MPI**错误句柄**

MPI_Errhandler_get(comm, errhandler, ierror)
integer errhandler, ierror

得到给定通信域的错误句柄，即将废弃的特性，建议用MPI_Comm_get_errhandler**代替**

MPI_Errhandler_set(comm, errhandler, ierror)
integer comm, errhandler, ierror

设置MPI**错误句柄，即将废弃的特性，建议用**MPI_Comm_set_errhandler**代替**

MPI_Error_class(errorcode, errorclass, ierror)
integer errorcode, errorclass, ierror

将错误代码转换为错误类

MPI_Error_string(errorcode, string, resultlen, ierror)
integer errorcode, resultlem, ierror
character *(MPI_MAX_ERROR_STRING) string

由给定的错误代码，返回它所对应的字符串

MPI_Finalize(ierror)
integer ierror

结束MPI**运行环境**

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, ierror)
<type> sendbuf(*), recvbuf(*)
integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror

从进程组中收集消息

MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer sendcount, sendtype, recvcounts(*), displs(*), recvtype, root, comm, ierror

从各进程组中收集消息到指定的位置

MPI_Get_count(status, datatype, count, ierror)

integer status(*), datatype, count, ierror

得到以给定数据类型为单位的数据的个数

MPI_Get_elements(status, datatype, elements, ierror)

integer status(*), datatype, elements, ierror

返回给定数据类型中基本元素的个数

MPI_Get_processor_name(name, resultlen, ierror)

character * (MPI_MAX_PROCESSOR_NAME) name

integer resultlen, ierror

得到处理器名称

MPI_Get_version(version, subversion, ierror)

integer version, subversion, ierror

返回MPI**的版本号**

MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph, ierror)

integer comm_old, nnodes, index(*), edges(*), comm_graph, ierror

Logical reorder

按照给定的拓扑创建新的通信域

MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)

integer comm, maxindex, maxedges, index(*), edges(*), error

得到给定通信域的处理器拓扑结构

MPI_Graph_map(comm, nnodes, index, edges, newrank, error)

integer comm, nnodes, index(*), edges(*), newrank, error

将进程映射到给定的拓扑

MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)

integer comm, rank, nneighbors, ierror

给定拓扑，返回给定结点的相邻结点数

MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)

integer comm, rank, maxneighbors, neighbors(*), ierror

给定拓扑，返回给定结点的相邻结点

MPI_Graphdims_Get(comm, nnodes, nedges, ierror)

integer comm, nnodes, nedges, ierror

得到给定通信域的图拓扑

MPI_Group_compare(group1, group2, result, ierror)

integer group1, group2, result, ierror

比较两个组

MPI_Group_difference(group1, group2, newgroup, ierror)

integer group1, group2, newgroup, ierror

根据两个组的差异创建一个新组

MPI_Gropu_excl(gropu, n, ranks, newgroup, ierror)

integer group, n, ranks(*), newgroup, ierror

通过重新对一个已经存在的组进行排序，根据未列出的成员创建一个新组

MPI_Group_free(group, ierror)

integer group, ierror

释放一个组

MPI_Group_incl(group, n, ranks, newgroup, ierror)

integer group, n, ranks(*), newgroup, ierror

通过重新对一个已经存在的组进行排序，根据列出的成员创建一个新组

MPI_Group_intersection(group1, group2, newgroup, ierror)

integer group1, group2, newgroup, ierror

根据两个已存在组的交创建一个新组

MPI_Group_range_excl(group, n, ranges, newgroup, ierror)

integer group, n, ranges(3, *), newgroup, ierror

根据已存在的组，去掉指定的部分，创建一个新组

MPI_Group_range_incl(group, n, ranges, newgroup, ierror)

integer group, n, ranges(3, *), newgroup, ierror

根据已存在的组，按照指定的部分，创建一个新组

MPI_Group_rank(group, rank, ierror)

integer group, rank, ierror

返回调用进程在给定组中的进程标识号

MPI_Group_size(group, size, ierror)

integer group, size, ierror

返回给定组的大小

MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)

integer group1, n, ranks1(*), group2, ranks2(*), ierror

将一个组中的进程标识号转换成另一个组的进程标识号

MPI_Group_union(group1, group2, newgroup, ierror)

integer group1, group2, newgroup, ierror

将两个组合并为一个新组

MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)

<type> buf(*)

integer count, datatype, dest, tag, comm, request, ierror

非阻塞缓冲区发送

MPI_Init(ierror)

integer ierror

MPI**执行环境初始化**

MPI_Initialized(flag, ierror)

logical flag

integer ierror

查询MPI_Init**是否已经调用**

MPI_Intercomm_create(local_comm, local_leader, peer_comm,

remote_leader, tag, newintercomm, ierror)

integer local_comm, local_leader, peer_comm, remote_leader, tag,

newintercomm, ierror

根据两个组内通信域创建一个组间通信域

MPI_Intercomm_merge(intercomm, high, intracomm, ierror)

integer intercomm, intracomm, ierror

logical high

根据组间通信域创建一个组内通信域

MPI_Iprobe(source, tag, comm, flag, status, ierror)

integer source, tag, comm, status(*), ierror

非阻塞消息测试

MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)

<type> buf(*)

integer count, datatype, source, tag, comm, request, ierror

非阻塞接收

MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)

<type> buf(*)

integer count, datatype, dest, tag, comm, request, ierror

非阻塞就绪发送

MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)

<type> buf(*)

integer count, datatype, dest, tag, comm, request, ierror

非阻塞发送

MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)

<type> buf(*)

integer count, datatype, dest, tag, comm, request, ierror

非阻塞同步发送

MPI_Keyval_create(copy_fn, delete_fn, keyval, extra_state, ierror)

external copy_fn, delete_fn

integer keyval, extra_state, ierror

创建一个新的属性关键词，即将废弃的特性，建议用MPI_Comm_create_keyval**代替**

MPI_Keyval_free(keyval, ierror)

integer keyval, ierror

释放一个属性关键词

MPI_Op_create(function, commute, op, ierror)

exterval function

logical commute

integer op, ierror

创建一个用户定义的通信函数句柄

MPI_Op_free(op, ierror)

integer op, ierror

释放一个用户定义的通信函数句柄

MPI_Pack(inbuf, incount, datatype, outbuf, outcount, position, comm, ierror)

<type>inbuf(*), outbuf(*)

integer incount, datatype, outcount, position, comm, ierror

将数据打包，放到一个连续的缓冲区中

MPI_Pack_size(incount, datatype, size, ierror)

integer incount, datatype, size, ierror

返回需要打包的数据类型的大小

MPI_Pcontrol(level)

integer level

控制剖视

MPI_Probe(cource, tag, comm, status, ierror)

integer source, tag, comm, status(*), ierror

阻塞消息测试

MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)

<type> buf(*)

integer count, datatype, source, tag, comm, status(*), ierror

标准接收

MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)

<type> buf(*)

integer count, datatype, source, tag, comm, request, ierror

创建接收句柄

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer count, datatype, op, root, comm, ierror

将所进程的值归约到root**进程，得到一个结果**

MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer recvcounts(*), datatype, op, comm, ierror

将结果归约后再发送出去

MPI_Request_free(request, ierror)

integer request, ierror

释放通信申请对象

MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)

<type> buf(*)

integer count, datatype, dest, tag, comm, ierror

就绪发送

MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)

<type> buf(*)

integer count, datatype, dest, tag, comm, request, ierror

创建就绪发送句柄

MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer count, datatype, ip, comm, ierror

在给定的进程组上进行扫描操作

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror

将数据从一个进程发送到组中其它进程

MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm,

ierror)
<type> sendbuf(*), recvbuf(*)
integer sendcounts(*), displs(*), sendtype, recvcount, recvtype,
root, comm, ierror

将缓冲区中的部分数据从一个进程发送到组中其它进程

MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
<type> buf(*)
integer count, datatype, dest, tag, comm, ierror

标准的数据发送

MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
<type> buf(*)
integer count, datatype, dest, tag, comm, request, ierror

创建一个标准发送的句柄

MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtyep, source,
recvtag, comm, status, ierror)
<type> sendbuf(*), recvbuf(*)
integer sendcount, sendtype, dest, sendtag, recvcount, recvtype,
source, recvtag, comm, status(*), ierror

同时完成发送和接收操作

MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, comm, status, ierror)
<type> buf(*)
integer count, datatype, dest, sendtag, source, recvtag, comm,
status(*), ierror

用同一个发送和接收缓冲区进行发送和接收操作

MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
<type> buf(*)
integer count, datatype, dest, tag, comm, ierror

同步发送

MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
<type> buf(*)
integer count, datatype, dest, tag, comm, request, ierror

创建一个同步发送句柄

MPI_Start(request, ierror)
integer request, ierror

启动给定对象上的重复通信请求

MPI_Startall(count, array_of_requests, ierror)
integer count, array_of_requests(*), ierror

启动指定的所有重复通信请求

MPI_Test(request, flag, status, ierror)
integer request, status(*), ierror
logical flag

测试发送或接收是否完成

MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
integer count, array_of_request(*),

array_of_statuses(MPI_STATUS_SIZE, *), ierror
logical flag

测试所有指定的通信是否完成

MPI_Testany(count, array_of_request, index, flag, status, ierror)
integer count, array_of_requests(*), index, status(*), ierror
logical flag

测试任何一个指定的通信是否完成

MPI_Testsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses, ierror)
integer incount, array_of_requests(*), outcount,
array_of_indices(*), array_of_statuses(MPI_STATUS_SIZE, *), ierror

测试是否有一些指定的通信已经完成

MPI_Test_cancelled(status, flag, ierror)
integer status(*), ierror

测试一个请求对象是否已经删除

MPI_Topo_test(comm, top_type, ierror)
integer comm, top_type, ierror

测试指定通信域的拓扑类型

MPI_Type_commit(datatype, ierror)
integer datatype, ierror

提交一个类型

MPI_Type_contiguous(count, oldtype, newtype, ierror)
integer count, oldtype, newtype, ierror

创建一个连续的数据类型

MPI_Type_extent(datatype, extent, ierror)
integer datatype, extent, ierror

返回一个数据类型的范围，即将废弃的特性，建议使用MPI_type_get_extent**来代替**

MPI_Type_free(datatype, ierror)
integer datatype, ierror

释放一个数据类型

MPI_Type_hindexed(count, array_of_blocklenghths, array_of_displacements, oldtype, newtype, ierror)
integer count, array_of_blocklengths(*), array_of_displacements(*),
oldtype, newtype, ierror

按照字节偏移，创建一个数据类型索引，即将废弃的特性，建议使用MPI_type_create_hindexed**来代替**

MPI_Type_hvector(count, blocklength, stride, oldtype, newtype, ierror)
integer count, blocklength, stride, oldtype, newtype, ierror

根据以字节为单位的偏移量，创建一个向量数据类型，即将废弃的特性，建议使用MPI_type_create_hvector**来代替**

MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements, oldtype, newtype, ierror)
integer count, array_of_blocklengths(*), array_of_displacements(*),
oldtype, newtype, ierror

创建一个索引数据类型

MPI_Type_lb(datatype, displacement, ierror)

integer datatype, displacement, ierror

返回指定数据类型的下边界，即将废弃的特性，建议使用MPI_type_get_extent**来代替**

MPI_Type_size(datatype, size, ierror)

integer datatype, size, ierror

以字节为单位，返回给定数据类型的大小

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_type(*), newtype, ierror

创建一个结构数据类型，即将废弃的特性，建议使用MPI_type_create_struct**来代替**

int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint * displacement)

返回指定数据类型的上边界，即将废弃的特性，建议使用MPI_type_get_extent**来代替**

MPI_Type_ub(datatype, displacement, ierror)

integer datatype, displacement, ierror

创建一个向量数据类型

MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)

integer count, blocklength, stride, oldtype, newtype, ierror

创建一个矢量数据类型

MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, ierror)

<type> inbuf(*), outbuf(*)

integer insize, position, outcount, datatype, comm, ierror

从连续的缓冲区中将数据解开

MPI_Wait(request, status, ierror)

integer request, status(*), ierror

等待MPI**的发送或接收语句结束**

MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)

integer count, array_of_requests(*),

array_of_statuses(MPI_STATUS_SIZE, *), ierror

等待所有给定的通信结束

MPI_Waitany(count, array_of_request, index, status, ierror)

integer count, array_of_requests(*), index, status(*), ierror

等待某些指定的发送或接收完成

MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses, ierror)

integer incount, array_of_requests(*), outcount,

array_of_indices(*), array_of_statuses(MPI_STATUS_SIZE, *), ierror

等待一些给定的通信结束

MPI_Wtick()

返回MPI_Wtime**的分辨率**

MPI_Wtime()

返回调用进程的流逝时间

18.3 MPI-2与C语言的接口

int MPI_Accumulate(void * origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

用指定的操作累计目标进程窗口中的数据

int MPI_Add_error_class(int * errorclass)

创建一个新的出错处理类并返回它的值

int MPI_Add_error_code(int errorclass, int * error)

创建一个与错误处理类相联系的错误处理代码，并返回它的值

int MPI_Add_error_string(int errorcode, char * string)

将一个出错提示串与错误处理类或错误代码建立联系

int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void * baseptr)

分配一块内存用于远程存储访问和消息传递操作

int MPI_Alltoallw(void * sendbuf, int sendcounts[], int sdispls[], MPI_Datatype sendtypes[], void * recvbuf, int recvcounts[], int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)

所有进程之间的数据交换，其数量、偏移和数据类型可以互不相同

int MPI_Close_port(char * port_name)

关闭指定的端口

int MPI_Comm_accept(char * port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm * newcomm)

接受请求，和客户端建立联系

MPI_Fint MPI_Comm_c2f(MPI_Comm comm)

C**通信域句柄转换为**Fortran**通信域句柄**

int MPI_Comm_call_errhandler(MPI_Comm comm, int error)

激活与指定通信域相联系的错误处理程序

int MPI_Comm_connect(char * portname, MPI_Info info, int root, MPI_Comm comm, MPI_Comm * newcomm)

请求和服务端建立联系

int MPI_Comm_create_errhandler(MPI_Comm_errhandler_fn *function, MPI_Errhandler *errhandler)

创建一个能附加到通信域的错误处理程序

int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn, MPI_Comm_delete_attr_function *comm_delete_attr_fn, int *comm_keyval, void *extra_state)

建一个能在通信域之上缓存的新属性

int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)

删除建立在通信域之上的缓存属性

int MPI_Comm_disconnect(MPI_Comm *comm)

等待所有在通信域之中排队的通信完成，断开与服务端的联系并释放通信域

MPI_Comm MPI_Comm_f2c(MPI_Fint comm)

把一个Fortran**通信域句柄转换成**C**通信域句柄**

int MPI_Comm_free_keyval(int *comm_keyval)

释放用MPI_Comm_create_keyval**创建的属性**

int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void attribute_val, int *flag)

返回与一个通信域缓存的属性对应的值

int MPI_Comm_get_errhandler(MPI_comm comm, MPI_Errhandler *errhandler)

返回当前与一个通信域对应的错误处理程序

int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)

返回与一个通信域对应的名字

int MPI_Comm_get_parent(MPI_Comm *parent)

返回一个既包括子进程又包括父进程的组间通信域。

int MPI_Comm_join(int fd, MPI_Comm *intercom)

将通过套接字连接的MPI**进程形成一个组间通信域返回**

int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)

设置通信域缓存的属性的值

int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)

把出错处理程序附加到通信域

int MPI_Comm_set_name(MPI_Comm comm, char *comm_name)

把名字对应到通信域

int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercom, int array_of_errcodes[])

产生子进程运行MPI**程序**

int MPI_Comm_spawn_multiple(int count, char *array_of_commands[], Char **array_of_argv[], int array_of_maxprocs[], MPI_Info array_of_info[], int root, MPI_Comm comm, MPI_Comm *intercom, int array_of_errcodes[])

产生子进程运行不同MPI**程序**

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

同MPI_Scan**，只不过归约操作的对象不包括自身**

MPI_Fint MPI_File_c2f(MPI_File file)

把C**文件句柄转换成**Fortran**文件句柄**

int MPI_File_call_errhandler(MPI_File fh, int error)

激活与一个文件对应的出错处理程序

int MPI_File_close(MPI_File *fh)

关闭一个文件

int MPI_File_create_errhanlder(MPI_File_errhandler_fn *function, MPI_Errhandler *errhandler)

创建能附加到文件的出错处理程序

int MPI_File_delete(char *filename, MPI_Info info)

删除一个文件

MPI_File MPI_File_f2c(MPI_Fint file)

把Fortran**文件句柄转换成**C**文件句柄**

int MPI_File_get_amode(MPI_File fh, int *amode)

返回文件的访问模式

int MPI_File_get_atomicity(MPI_File fh, int *flag)

得到文件句柄fh**对应文件的访问模式信息，结果放在**flag**中**

int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset, MPI_Offset *disp)

把一个相对视口的文件偏移转换成绝对字节偏移

int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)

返回与文对应的出错处理程序

int MPI_File_get_group(MPI_file fh, MPI_Group *group)

返回用来打开文件的通信域组的副本

int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)

返回与文件相联系的INFO**对象的值**

int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)

返回非共享文件指针的当前位置

int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)

返回共享文件指针的当前位置

int MPI_File_get_size(MPI_File fh, MPI_Offset *size)

返回共享文件的大小

int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype, MPI_Aint *extent)

返回文件中数据类型的跨度

int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep)

返回当前文件视口

int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)

在非共享文件指针的当前位置开始非阻塞读文件

int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)

在指定的偏移开始非阻塞读文件

int MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)

在共享文件指针的当前位置开始非阻塞读文件

int MPI_File_iwrite(MPI_File fh, void *buf, int count, MPI_Datatype, MPI_Request *request)

在非共享文件指针的当前位置开始非阻塞写文件

int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)

在指定的偏移开始非阻塞写文件

int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)

在共享文件指针的当前位置开始非阻塞写文件

int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)

打开一个文件

int MPI_File_preallocate(MPI_File fh, MPI_Offset size)

为一个文件预分配磁盘空间

int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

在非共享文件指针的当前位置处读数据

int MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

非共享文件的组读取，如同进程组内的所有进程都执行了一个MPI_FILE_READ**调用**

一样

int MPI_File_read_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)

用二步法开始一个非共享文件的组读取

int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)

用二步法完成一个非共享文件的组读取

int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

从指定文件偏移处读数据

int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

从指定位置开始读取的组调用，其效果就如同每个进程都执行了一个相应的 MPI_FILE_READ_AT **操作**

int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, Void *buf, int count, MPI_Datatype datatype)

用二步法开始一个从指定位置读取的组调用

int MPI_file_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)

用二步法完成一个从指定位置读取的组调用

int MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

用共享文件指针进行组读取，就如同每一个进程组内的进程依次执行了一个 MPI_FILE_READ_SHARED**调用**

int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_datatype datatype)

用二步法开始一个共享文件的组读取

int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_status *status)

用二步法完成一个共享文件的组读取

int MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

从共享文件指针的当前位置开始读数据

int MPI_file_seek(MPI_File fh, MPI_Offset offset, int whence)

移动非共享文件指针

int MPI_file_seek_shared(MPI_File fh, MPI_Offset offset, int whence)

移动共享文件指针

int MPI_File_set_atomicity(MPI_File fh, int flag)

组调用设置文件的访问模式

int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)

把一个新的出错处理程序附加到文件上

int MPI_File_set_info(MPI_File fh, MPI_Info info)

设置与一个文件对应的INFO**对象的值**

int MPI_File_set_size(MPI_File fh, MPI_Offset size)

设置文件长度

int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPOI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_info info)

设置文件视口

int MPI_File_sync(MPI_File fh)

将缓冲文件数据与存储设备的数据进行同步

int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

在非共享文件指针的当前位置处写入数据

int MPI_File_write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

对非共享文件执行组写入，其效果就如同每一个进程都执行了一个MPI_File_write**调用**

int MPI_File_write_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)

用二步法开始一个非共享文件的组写入

int MPI_File_write_all_end(MPI_file fh, void *buf, MPI_Status *status)

用二步法完成一个非共享文件的组写入

int MPI_File_write_at(MPI_file fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

在指定的文件偏移处写数据

int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

在各自的指定偏移处开始，执行文件的组写入

int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, Void *buf, int count, MPI_Datatype datatype)

用二步法开始一个指定位置的组写入

int MPI_File_write_at_all_end(MPI_File fh, void *buf, MPI_Status *status)

用二步法完成一个指定位置的组写入

int MPI_File_write_ordered(MPI_file fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

对共享文件进行组写入，其效果就如同每一个进程都按序依次执行了一个MPI_File_write_shared**调用**

int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)

用二步法开始一个共享文件的组写入

int MPI_File_write_ordered_end(MPI_File fh, viod *buf, MPI_Status *status)

用二步法完成一个共享文件的组写入

int MPI_File_write_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

在共享文件指针的当前位置处写入数据

int MPI_Finalized(int *flag)

检查MPI_Finalize**是否完成**

int MPI_Free_mem(void *base)

释放以MPI_Alloc_mem**申请的内存**

int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)

从指定进程的窗口取数据

int MPI_Get_address(void *location, MPI_Aint *address)

返回内存位置的地址

int MPI_Grequest_complete(MPI_Request request)

通知MPI**给定对象上的操作已经结束**

int MPI_Grequest_start(MPI_Grequest_query_function *query_fn, MPI_Grequest_free_function

*free_fn, MPI_Grequest_cancel_function *cancel_fn, void *extra_state, MPI_Request *request)

开始一个通用的非阻塞操作，并返回与该操作相联系的对象

MPI_Fint MPI_Group_c2f(MPI_Group group)

把C**进程组句柄转换为**Fortran**进程组句柄**

MPI_Group MPI_Group_f2c(MPI_Fint group)

把Fortran**进程组组句柄转换为**C**进程组句柄**

MPI_Fint MPI_Info_c2f(MPI_Info info)

把C**信息句柄转换为**Fortran**信息句柄**

int MPI_Info_create(MPI_Info *info)

创建一个新的INFO**对象**

int MPI_Info_delete(MPI_Info info, char *key)

从INFO**对象中删除**<**关键字，值**>**元组**

int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)

返回INFO**对象的副本**

MPI_Info MPI_Info_f2c(MPI_Fint info)

把Fortran INFO**对象句柄转换为**C INFO**对象句柄**

int MPI_Info_free(MPI_Info *info)

释放INFO**对象**

int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value, int *flag)

返回与信息关键字对应的值

int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)

返回INFO**对象当前定义的关键字的数量**

int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)

返回INFO**对象定义的第**n**个关键字**

int MPI_Info_get_valuelen(MPI_Info info, char *key, int *valuelen, int *flag)

返回与一个信息关键字对应的值的长度

int MPI_Info_set(MPI_Info info, char *key, char *value)

往INFO**对象中加入**<**关键字**,**值**>**元组**

int MPI_Init_thread(int *argc, char *((*argv)[]), int required, int *provided)

初始化MPI**和**MPI**线程环境**

int MPI_Is_thread_main(int *flag)

表明调用本函数的线程是否是主线程

int MPI_Lookup_name(char *service_name, MPI_Info info, Char *prot_name)

返回与服务名对应的端口名

MPI_Fint MPI_Op_c2f(MPI_Op op)

把C**操作句柄转换为**Fortran**句柄**

MPI_Op MPI_Op_f2c(MPI_Fint op)

把Fortran**句柄转换为**C**操作句柄**

int MPI_Open_port(MPI_Info info, char *port_name)

建立一个网络地址，以使服务器能够接收客户的连接请求

int MPI_Pack_external(char *datarep, void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, MPI_Aint outsize, MPI_Aint *position)

以指定的数据格式进行打包

int MPI_Pack_external_size(char *datarep, int incount, MPI_Datatype datatype, MPI_Aint *size)

返回以指定的数据格式，数据打包需要的空间的大小

int MPI_Publish_name(char *service_name, MPI_Info info, Char *port_name)

将一个服务名和端口名建立联系，并将该服务名公之与众

int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)

向指定进程的窗口写入数据

int MPI_Query_thread(int *provided)

返回支持线程的级别

int MPI_Register_datarep(char *datarep, MPI_Datarep_conversion_function *read_conversion_fn, MPI_Datarep_conversion_function *write_conversion_fn, MPI_Datarep_extent_function *dtype_file_extent_fn, Void *extra_state)

加入新的文件数据表示到MPI

MPI_Fint MPI_Request_c2f(MPI_Request request)

把C**请求句柄转换成**Fortran**请求句柄**

MPI_Request MPI_Request_f2c(MPI_Fint request)

把Fortran**请求句柄转换成**C**请求句柄**

int MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status)

测试非阻塞操作的完成情况，如完成不释放请求对象

int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status)

把C**状态对象转换成**Fortran**状态对象**

int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)

把Fortran**状态对象转换成**C**状态对象**

int MPI_Status_set_cancelled(MPI_Status *status, int flag)

设置MPI_Test_cancelled**将要返回的值**

int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype, int count)

设置MPI_Get_elements**将要返回的值**

MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)

将C**数据类型句柄转换成**Fortran**数据类型句柄**

int MPI_Type_create_darray(int size, int rank, int ndims, int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

创建一个分布数组数据类型

int MPI_type_create_f90_complex(int p, int r, MPI_Datatype *newtype)

返回一个预定义的MPI**数据类型，它与**Fortran 90**复数变量的指定精度和十进制指数范围一致**

int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)

返回一个预定义的MPI**数据类型，它与**Fortran 90**整数变量的十进制数字的指定个数一致**

int MPI_type_create_f90_real(int p, int r, MPI_Datatype *newtype)

返回一个预定义的MPI**数据类型，它与**Fortran 90**实数变量的指定精度和十进制指数范围一致**

int MPI_Type_create_hindexed(int count, int array_of_blocklengths[], MPI_Ainyt array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)

创建带字节偏移的索引数据类型

int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

通过以字节为单位的间距，创建向量数据类型

int MPI_Type_create_indexed_block(int count, int blocklength, int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)

创建固定块长的索引数据类型

int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn, MPI_Type_delete_attr_function *type_delete_attr_fn, int *type_keyval, void *extra_state)

创建一个能在数据类型上缓冲的新属性关键字

int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype)

返回一个指定下限和范围的新数据类型

int MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[], MPI_Datatype *newtype)

创建结构数据类型

int MPI_type_create_subarray(int ndims, int array_of_sizes[], int array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

创建子数组数据类型

int MPI_type_delete_attr(MPI_Datatype type, int type_keyval)

删除数据类型上缓冲的属性关键字

int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)

返回数据类型的副本

MPI_datatype MPI_Type_f2c(MPI_Fint datatype)

把Fortran**数据类型句柄转换成**C**数据类型句柄**

int MPI_Type_free_keyval(int *type_keyval)

释放用MPI_Type_create_keyval**创建的属性关键字**

int MPI_Type_get_attr(MPI_Datatype type, int type_keyval, Void *attribute_val, int *flag)

返回与一个数据类型上缓冲的属性关键字对应的值

int MPI_type_get_contents(MPI_Datatype datatype, int max_integers, int max_addresses, int max_datatypes, int array_of_integers[], MPI_Aint array_of_addresses[], MPI_Datatype array_of_datatypes[])

返回用来创建派生数据类型的参数的值

int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers, int *num_addresses, int *num_datatypes, int *combiner)

返回数据类型的类型和用来创建数据类型的参数的数量和类型

int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)

返回数据类型的下限和范围

int MPI_Type_get_name(MPI(_Datatype type, char *type_name, int *resultlen)

返回与数据类型对应的名称

int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent)

返回数据类型的真实范围

int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type)

返回与指定类型和大小的局部变量匹配的MPI**数据类型**

int MPI_Type_set_attr(MPI_Datatype type, int type_keyval, void *attribute_val)

设置在一个数据类型上缓冲的属性关键字的值

int MPI_Type_set_name(MPI_Datatype type, char *type_name)

把名字和数据类型相关联

int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize, MPI_Aint *position, void *outbuf, int outocunt, MPI_Datatype datatype)

以指定的格式将数据解包

int MPI_Unpublish_name(char *service_name, MPI_Info info, Char *port_name)

取消一个以前发布的服务名

MPI_Fint MPI_Win_c2f(MPI_Win win)

把C**窗口对象句柄转换为**Fortran**窗口对象句柄**

int MPI_Win_call_errhandler(MPI_Win win, int error)

激活与窗口对象对应的错误处理程序

int MPI_Win_complete(MPI_Win win)

完成从MPI_Win_start**开始的**RMA**访问**

int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)

创建新窗口对象

int MPI_Win_create_errhandler(MPI_Win_errhandler_fn *function, MPI_Errhandler *errhandler)

创建对附加到窗口对象上的错误处理程序

int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn, MPI_Win_delete_attr_function *win_delete_attr_fn, int *win_keyval, void *extra_state)

创建能在窗口对象上缓冲的新属性关键字

int MPI_Win_delete_attr(MPI_Win win, int win_keyval)

删除在窗口对象上缓冲的属性关键字

MPI_Win MPI_Win_f2c(MPI_Fint win)

把Fortran**窗口对象句柄转换成**C**窗口对象句柄**

int MPI_Win_fence(int assert, MPI_Win win)

同步窗口对象上的RMA**操作**

int MPI_Win_free(MPI_Win *win)

释放窗口对象

int MPI_Win_free_keyval(int *win_keyval)

释放用MPI_Win_create_keyval**创建的属性关键字**

int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val, int *flag)

返回与窗口对象上缓冲的属性关键字对应的值

int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)

返回与窗口对象对应的出错处理程序

int MPI_Win_get_group(MPI_Win win, MPI_Group *group)

返回用来创建窗口对象的通信域组的副本

int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)

返回与窗口对象对应的名称

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

对指定进程的窗口加锁

int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)

开始允许其它窗口的远程访问

int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)

设置窗口对象缓冲的属性关键字的值

int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)

把一个新的错误处理程序附加到窗口对象上

int MPI_Win_set_name(MPI_Win win, char *win_name)

将名称与窗口对象关联

int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)

准备访问其它的窗口，实现与MPI_Win_post**的握手**

int MPI_Win_test(MPI_Win win, int *flag)

测试窗口对象之上的RMA**操作是否完成**

int MPI_Win_unlock(int rank, MPI_Win win)

对给定的窗口开锁

int MPI_Win_wait(MPI_Win win)

完成用MPI_Win_post**启动的**RMA**访问**

18.4 MPI-2与Fortran语言的接口

MPI_Accumulate(origin_addr, origin_count, origin_datatype, Target_rank, target_disp, target_count, target_datatype, op, Win, ierror)

<type>origin_addr(*)

integer(kind=MPI_ADDRESS_KIND) target_disp

integer origin_count, origin_datatype, target_rank, target_count,

target_datatype, op, win, ierror

用指定的操作累计目标进程窗口中的数据

MPI_Add_error_class(errorclass, ierror)

integer errorclass, ierror

创建一个新的出错处理类并返回它的值

MPI_Add_error_code(errorclass, errorcode, ierror)

integer errorclass, errorcode, ierror

创建一个与错误处理类相联系的错误处理代码，并返回它的值

MPI_Add_error_string(errorcode, string, ierror)

integer errorcode, ierror

character*(*) string

将一个出错提示串与错误处理类或错误代码建立联系

MPI_Alloc_mem(size, info, baseptr, ierror)

integer info, ierror

integer(kind=MPI_ADDRESS_KIND) size, baseptr

分配一块内存用于远程存储访问和消息传递操作

MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, Recvcounts, rdispls, recvtypes, comm, ierror)

<type> sendbuf(*), recvbuf(*)

integer sendcounts(*), sdispls(*), sendtypes(*), recvcounts(*),

rdispls(*), recvtypes(*), comm, ierror

所有进程之间的数据交换，其数量、偏移和数据类型可以互不相同

MPI_Close_port(port_name, ierror)

character*(*) port_name

integer ierror

关闭指定的端口

MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)

character*(*) port_name

integer info, root, comm, newcomm, ierror

和客户端建立联系

MPI_Comm_call_errhandler(comm, errorcode, ierror)

integer comm, errorcode, ierror

激活与指定通信域相联系的错误处理程序

MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)

character*(*) port_name

integer info, root, comm, newcomm, ierror

和服务端建立联系

MPI_Comm_create_errhandler(function, errhandler, ierror)

External function

integer errhandler, ierror

创建一个能附加到通信域的错误处理程序

MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, Comm_keyval, extra_state, ierror)

External comm_copy_attr_fn, comm_delete_attr_fn

integer comm_keyval, ierror

integer(kind=MPI_ADDRESS_KIND) extra_state

建一个能在通信域之上缓存的新属性

MPI_Comm_delete_attr(comm, comm_keyval, ierror)

integer comm, comm_keyval_ierror

删除建立在通信域之上的缓存属性

MPI_Comm_disconnect(comm, ierror)

integer comm, ierror

等待所有在通信域之中排队的通信完成，断开连接并释放通信域

MPI_Comm_free_keyval(comm_keyval, ierror)

integer comm_keyval, ierror

释放用MPI_Comm_create_keyval**创建的属性**

MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)

integer comm, comm_keyval, ierror

integer(kind=MPI_ADDRESS_KIND) attribute_val

Logical flag

返回与一个通信域缓存的属性对应的值

MPI_Comm_get_errhandler(comm, errhandler, ierror)

integer comm, errhandler, ierror

返回当前与一个通信域对应的错误处理程序

MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
integer comm, resultlen, ierror
character*(*) comm_name

返回与一个通信域对应的名字

MPI_Comm_get_parent(parent, ierror)
integer parent, ierror

返回一个既包括子进程又包括父进程的组间通信域。

MPI_Comm_join(fd, intercom, ierror)
integer fd, intercom, ierror

将通过套接字连接的MPI**进程形成一个组间通信域返回**

MPI_Comm_set_attr(com, comm_keyval, attribute_val, ierror)
integer comm, comm_keyval, ierror
integer(kind=MPI_ADDRESS_KIND) attribute_val

设置通信域缓存的属性的值

MPI_Comm_set_errhandler(comm, errhandler, ierror)
integer comm, errhandler, ierror

把出错处理程序附加到通信域

MPI_Comm_set_name(comm, comm_name, ierror)
integer comm, ierror
character*(*) comm_name

把名字对应到通信域

MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercom, array_of_errcodes, ierror)
character*(*) command, argv(*)
integer info, maxprocs, root, comm, intercomm, array_of_errcodes(*), ierror

产生子进程运行MPI**程序**

MPI_Comm_spwan_multiple(count, array_of_commands, array_of_argv, aray_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes, ierror)
integer count, array_of_info(*), array_of_maxprocs(*), root, comm,
intercomm, array_of_errcodes(*), ierror
character*(*) array_of_commands(*), array_of_argv(count, *)

产生子进程运行不同MPI**程序**

MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
<type> sendbuf(*), recvbuf(*)
integer count, datatype, op, comm, ierror

同MPI_Scan，**只不过归约操作的对象不包括自身**

MPI_File_call_errhandler(fh, errorcode, ierror)
integer fh, errorcode, ierror

激活与一个文件对应的出错处理程序

MPI_File_close(fh, ierror)
integer fh, ierror

关闭一个文件

MPI_File_create_errhandler(function, errhandler, ierror)
External function

integer errhandler, ierror

创建能附加到文件的出错处理程序

MPI_File_delete(filename, info, ierror)

Character*(*) filename

integer info, ierror

删除一个文件

MPI_File_get_amode(fh, amode, ierror)

integer fh, amode, ierror

返回文件的访问模式

MPI_File_get_atomicity(fh, flag, ierror)

integer fh, ferror

Logical flag

得到文件句柄fh**对应文件的访问模式信息，结果放在**flag**中**

MPI_File_get_byte_offset(fh, offset, disp, ierror)

integer fh, ierror

integer(kind=MPI_OFFSET_KIND) offset, disp

把一个相对视口的文件偏移转换成绝对字节偏移

MPI_File_get_errhandler(file, errhandler, ierror)

integer file, errhandler, ierror

返回与文对应的出错处理程序

MPI_File_get_group(fh, group, ierror)

integer fh, group, ierror

返回用来打开文件的通信域组的副本

MPI_File_get_info(fh, info_used, ierror)

integer fh, info_used, ierror

返回与文件相联系的INFO**对象的值**

MPI_File_get_position(fh, offset, ierror)

integer fh, ierror

integer(kind=MPI_OFFSET_KIND) offset

返回非共享文件指针的当前位置

MPI_File_get_position_shared(fh, offset, ierror)

integer fh, ierror

integer(kind=MPI_OFFSET_KIND) offset

返回共享文件指针的当前位置

MPI_File_get_size(fh, size, ierror)

integer fh, ierror

integer(kind=MPI_OFFSET_KIND) size

返回文件的大小

MPI_File_get_type_extent(fh, datatype, extent, ierror)

integer fh, datatype, ierror

integer(kind=MPI_ADDRESS_KIND) extent

返回文件数据类型的跨度

MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)

integer fh, etype, filetype, ierror

Character*(*) datarep, integer(kind=MPI_OFFSET_KIND) disp

返回当前文件视口

MPI_File_iread(fh, buf, count, datatype, request, ierror)

<type> buf(*)

integer fh, count, datatype, request, ierror

在非共享文件指针的当前位置开始非阻塞读文件

MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)

<type> buf(*)

integer fh, count, datatype, request, ierror

integer(kind=MPI_OFFSET_KIND) offset

在指定的偏移开始非阻塞读文件

MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)

<type> buf(*)

integer fh, count, datatype, request, ierror

在共享文件指针的当前位置开始非阻塞读文件

MPI_File_iwrite(fh, buf, count, datatype, request, ierror)

<type> buf(*)

integer fh, count, datatype, request, ierror

在非共享文件指针的当前位置开始非阻塞写文件

MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)

<type> buf(*)

integer fh, count, datatype, request, ierror

integer(kind=MPI_OFFSET_KIND) offset

在指定的偏移开始非阻塞写文件

MPI_File_iwrite_share(fh, buf, count, datatype, request, ierror)

<type> buf(*)

integer fh, count, datatype, request, ierror

在共享文件指针的当前位置开始非阻塞写文件

MPI_File_open(comm., filename, amode, info, fh, ierror)

Character*(*) filename

integer comm., amode, info, fh, ierror

打开一个文件

MPI_File_preallocate(fh, size, ierror)

integer fh, ierror

integer(kind=MPI_OFFSET_KIND) size

为一个文件预分配磁盘空间

MPI_File_read(fh, buf, count, datatype, status, ierror)

<type> buf(*)

integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

在非共享文件指针的当前位置处读数据

MPI_File_read_all(fh, buf, count, datatype, status, ierror)

<type> buf(*)

integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

非共享文件的组读取，如同进程组内的所有进程都执行了一个MPI_FILE_READ**调用**

一样

MPI_File_read_all_begin(fh, buf, count, datatype, ierror)

<type> buf(*)

integer fh, count, datatype, ierror

用二步法开始一个非共享文件的组读取

MPI_File_read_all_end(fh, buf, status, ierror)

<type> buf(*)

integer fh, status(MPI_STATUS_SIZE), ierror

用二步法完成一个非共享文件的组读取

MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)

<type> buf(*)

integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

integer(kind=MPI_OFFSET_KIND) offset

从指定文件偏移处读数据

MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)

<type> buf(*)

integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

integer(kind=MPI_OFFSET_KIND) offset

从指定位置开始读取的组调用，其效果就如同每个进程都执行了一个相应的 MPI_FILE_READ_AT **操作**

MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)

<type> buf(*)

integer fh, count, datatype, ierror

integer(kind=MPI_OFFSET_KIND) offset

用二步法开始一个从指定位置读取的组调用

MPI_File_read_at_all_end(fh, buf, status, ierror)

<type> buf(*)

integer fh, status(MPI_STATUS_SIZE), ierror

用二步法完成一个从指定位置读取的组调用

MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)

<type> buf(*)

integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

用共享文件指针进行组读取，就如同每一个进程组内的进程依次执行了一个 MPI_FILE_READ_SHARED**调用**

MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)

<type> buf(*)

integer fh, count, datatype, ierror

用二步法开始一个共享文件的组读取

MPI_File_read_ordered_end(fh, buf, status, ierror)

<type> buf(*)

integer fh, status(MPI_STATUS_SIZE), ierror

用二步法完成一个共享文件的组读取

MPI_File_read_shared (fh, buf, count, datatype, status, ierror)

<type> buf(*)

integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

从共享文件指针的当前位置开始读数据

MPI_File_seek(fh, offset, whence, ierror)

integer fh, whence, ierror

integer(kind=MPI_OFFSET_KIND) offset

移动文件指针

MPI_File_seek_shared(fh, offset, whence, ierror)

integer fh, whence, ierror

integer(kind=MPI_OFFSET_KIND) offset

移动共享文件指针

MPI_File_set_atomicity(fh, flag, ierror)

integer fh, ierror

logical flag

组调用设置文件的访问模式

MPI_File_set_errhandler(file, errhandler, ierror)

integer file, errhandler, ierror

把一个新的出错处理程序附加到文件上

MPI_File_set_info(fh, info, ierror)

integer fh, info, ierror

设置与一个文件对应的INFO**对象的值**

MPI_File_set_size(fh, size, ierror)

integer fh, ierror

integer(kind=MPI_OFFSET_KIND) size

设置文件长度

MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)

integer fh, etype, filetype, info, ierror

character*(*) datarep

integer(kind=MPI_OFFSET_KIND) disp

设置文件视口

MPI_File_sync(fh, ierror)

integer fh, ierror

将缓冲文件数据与存储设备的数据进行同步

MPI_File_write(fh, buf, count, datatype, status, ierror)

<type> buf(*)

integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

在非共享文件指针的当前位置处写入数据

MPI_File_write_all(fh, buf, count, datatype, status, ierror)

<type> buf(*)

integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

对非共享文件执行组写入，其效果就如同每一个进程都执行了一个MPI_File_write**调用**

MPI_File_write_all_begin(fh, buf, count, datatype, ierror)

<type> buf(*)

integer fh, count, datatype, ierror

用二步法开始一个非共享文件的组写入

MPI_File_write_all_end(fh, buf, status, ierror)
<type> buf(*)
integer fh, status(MPI_STATUS_SIZE), ierror

用二步法完成一个非共享文件的组写入

MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
<type> buf(*)
integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror
integer(kind=MPI_OFFSET_KIND) offset

在指定的文件偏移处写数据

MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
<type> buf(*)
integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror
integer(kind=MPI_OFFSET_KIND) offset

在各自的指定偏移处开始，执行文件的组写入

MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
<type> buf(*)
integer fh, count, datatype, ierror
integer(kind=MPI_OFFSET_KIND) offset

用二步法开始一个指定位置的组写入

MPI_File_write_at_all_end(fh, buf, status, ierror)
<type> buf(*)
integer fh, status(MPI_STATUS_SIZE), ierror

用二步法完成一个指定位置的组写入

MPI_File_write_ordered (fh, buf, count, datatype, status, ierror)
<type> buf(*)
integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

对共享文件进行组写入，其效果就如同每一个进程都按序依次执行了一个MPI_File_write_shared**调用**

MPI_File_write_ordered _begin(fh, buf, count, datatype, ierror)
<type> buf(*)
integer fh, count, datatype, ierror

用二步法开始一个共享文件的组写入

MPI_File_write_ordered _end(fh, buf, status, ierror)
<type> buf(*)
integer fh, status(MPI_STATUS_SIZE), ierror

用二步法完成一个共享文件的组写入

MPI_File_write_shared (fh, buf, count, datatype, status, ierror)
<type> buf(*)
integer fh, count, datatype, status(MPI_STATUS_SIZE), ierror

在共享文件指针的当前位置处写入数据

MPI_Finalized(flag, ierror)
logical flag
integer ierror

表示MPI_Finalize**是否完成**

MPI_Free_mem(base, ierror)
<type> base(*)
integer ierror

释放以MPI_Alloc_mem**申请的内存**

MPI_Get(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierror)
<type> origin_addr(*)
integer(kind=MPI_ADDRESS_KIND) target_disp
integer origin_count, origin_datatype, target_rank, target_count,
target_datatype, win, ierror

从指定进程的窗口取数据

MPI_Get_address(location, address, ierror)
<type> location(*)
integer ierror
integer(kind=MPI_ADDRESS_KIND) address

返回内存位置的地址

MPI_Grequest_complete(request, ierror)
integer request, ierror

通知MPI**给定对象上的操作已经结束**

MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request, ierror)
integer request, ierror
external query_fn, free_fn, cancel_fn
integer (kind=MPI_ADDRESS_KIND) extra_state

开始一个通用的非阻塞操作，并返回与该操作相联系的对象

MPI_Info_create(info, ierror)
integer info, ierror

创建一个新的INFO**对象**

MPI_Info_delete(info, key, ierror)
integer info, ierror
integer info, ierror
character*(*) key

从INFO**对象中删除(关键字，值)元组**

MPI_Info_dup(info, newinfo, ierror)
integer info, newinfo, ierror

返回INFO**对象的副本**

MPI_Info_free(info, ierror)
integer info, ierror

释放INFO**对象**

MPI_Info_get(info, key, valuelen, value, falg, ierror)
integer info, valuelen, ierror
character*(*) key, value
logical flag

返回与信息关键字对应的值

MPI_Info_get_nkeys(info, nkeys, ierror)

integer info, nkeys, ierror

返回INFO**对象当前定义的关键字的数量**

MPI_Info_get_nthkey(info, n, key, ierror)

integer info, n, ierror

character*(*) key

返回INFO**对象定义的第**n**个关键字**

MPI_Info_get_valuelen(info, key, valuelen, falg, ierror)

integer info, valuelen, ierror

logical flag

character*(*) key

返回与一个信息关键字对应的值的长度

MPI_Info_set(info, key, value, ierror)

integer info, ierror

character*(*) key, value

往INFO**对象中加入(关键字,值)元组**

MPI_Init_thread(required, provided, ierror)

integer reuqired, provided, ierror

初始化MPI**和**MPI**线程环境**

MPI_Is_thread_main(flag, ierror)

logical flag

integer ierror

表明调用本函数的线程是否是主线程

MPI_Lookup_name(service_name, info, port_name, ierror)

character*(*) service_name, port_name

integer info, ierror

返回与服务名对应的端口名

MPI_Open_port(info, port_name, ierror)

character*(*) port_name

integer info, ierror

建立一个网络地址，以使服务器能够接收客户的连接请求

MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize, Position, ierror)

integer incount, datatype, ierror

integer(kind=MPI_ADDRESS_KIND) outsize, position

character*(*) datarep

<type> inbuf(*), outbuf(*)

以指定的数据格式进行打包

MPI_Pack_external_size(datarep, incount, datatype, size, ierror)

integer incount, datatype, ierror

integer(kind=MPI_ADDRESS_KIND) size

character*(*) datarep

返回以指定的数据格式，数据打包需要的空间的大小

MPI_Publish_name(service_name, info, port_name, ierror)

integer info, ierror

character*(*) service_name, port_name

将一个服务名和端口名建立联系，并将该服务名公之与众

MPI_Put(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierror)

<type> origin_addr(*)

integer(kind=MPI_ADDRESS_KIND) target_disp

integer origin_count, origin_datatype, target_rank, target_count,

target_datatype, win, ierror

向指定进程的窗口写入数据

MPI_Query_thread(provide, ierror)

integer provided, ierror

返回支持线程的级别

MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn, dtype_file_extent_fn, extra_state, ierror)

character*(*) datarep

external read_conversion_fn, write_conversion_fn, dtype_file_extent_fn

integer(kind=MPI_ADDRESS_KIND) extra_state

integer ierror

加入新的文件数据表示到MPI

MPI_Request_get_status(request, falg, status, ierror)

integer request, status(MPI_STATUS_SIZE), ierror

logical flag

测试非阻塞操作的完成情况，如完成不释放请求对象

MPI_Sizeof(x, size, ierror)

<type> x

integer size, ierror

返回变量的机器表示字节数

MPI_Status_set_cancelled(status, flag, ierror)

integer status(MPI_STATUS_SIZE), ierror

logical falg

设置MPI_Test_cancelled**将要返回的值**

MPI_Status_set_elements(status, datatype, count, ierror)

integer status(MPI_STATUS_SIZE), datatype, count, ierror

设置MPI_Get_elements**将要返回的值**

MPI_Type_create_darray(size, rank, ndims, array_of_gsizes, array_of_distribs, array_of_dargs, array_of_psizes, order, oldtype, newtype, ierror)

integer size, rank, ndims, array_of_gsizes(*), array_of_distribs(*),

array_of_dargs(*), array_of_psizes(*), order, oldtype, newtype, ierror

创建一个分布数组数据类型

MPI_type_create_f90_complex(p, r, newtype, ierror)

integer p, r, newtype, ierror

返回一个预定义的MPI**数据类型，它与**Fortran 90**复数变量的指定精度和十进制指数范围一致**

MPI_Type_create_f90_integer(r, newtype, ierror)

integer r, newtype, ierror

返回一个预定义的MPI数据类型，它与Fortran 90整数变量的十进制数字的指定个数一致

MPI_Type_create_f90_real(p, r, newtype, ierror)
integer p, r, newtype, ierror

返回一个预定义的MPI数据类型，它与Fortran 90实数变量的指定精度和十进制指数范围一致

MPI_Type_create_hindexed(count, array_of_blocklengths, array_of_dispalcements, oldtype, newtype, ierror)
integer count, array_of_blocklengths(*), oldtype, newtype, ierror
integer(kind=MPI_ADDRESS_KIND) array_of_displacements(*)

创建带字节偏移的索引数据类型

MPI_Type_create_hvector(count, blocklength, stide, oldtype, newtype, ierror)
integer count, blocklength, oldtype, newtype, ierror
integer(kind=MPI_ADDRESS_KIND) stride

通过以字节为单位的间距，创建向量数据类型

MPI_Type_create_indexed_block(count, blocklength, array_of_displacements, oldtype, newtype, ierror)
integer count, blocklength, array_of_displacements(*), oldtype,
newtype, ierror

创建固定块长的索引数据类型

MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval, extra_state, ierror)
external type_copy_attr_fn, type_delete_attr_fn
integer type_keyval, ierror
integer(kind=MPI_ADDRESS_KIND) extra_state

创建一个能在数据类型上缓冲的新属性关键字

MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
integer oldtype, newtype, ierror
integer(kind=MPI_ADDRESS_KIND) lb, extent

返回一个指定下限和范围的新数据类型

MPI_Type_create_struct(count, array_of_blocklengths, array_of_displacements, array_of_types, newtyep, ierror)
integer count, array_of_blocklengths(*), array_of_types(*), newtype, ierror
integer(kind=MPI_ADDRESS_KIND) array_of_displacements(*)

创建结构数据类型

MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype, ierror)
integer ndims, array_of_sizes(*), array_of_subsizes(*),
array_of_starts(*), order, oldtype, newtype, ierror

创建子数组数据类型

MPI_type_delete_attr(type, type_keyval, ierror)
integer type, tyep_keyval, ierror

删除数据类型上缓冲的属性关键字

MPI_Type_dup(type, newtype, ierror)
integer type, newtype, ierror

返回数据类型的副本

MPI_Type_free_keyval(type_keyval, ierror)
integer type_keyval, ierror

释放用MPI_Type_create_keyval**创建的属性关键字**

MPI_Type_get_attr(type, type_keyval, attribute_val, flag, ierror)
integer type, type_keyval, ierror
integer(kind=MPI_ADDRESS_KIND) attribute_val
logical flag

返回与一个数据类型上缓冲的属性关键字对应的值

MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes, array_of_integers, array_of_addresses, array_of_datatypes, ierror)
integer datatype, max_integers, max_addresses, max_datatypes,
array_of_integers(*), array_of_datatypes(*), ierror
integer(kind=MPI_ADDRESS_KIND) array_of_addresses(*)

返回用来创建派生数据类型的参数的值

MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes, combiner, ierror)
integer datatype, num_integers, num_addresses, num_datatypes,
combiner, ierror

返回数据类型的类型和用来创建数据类型的参数的数量和类型

MPI_Type_get_extent(datatype, lb, extent, ierror)
integer datatype, ierror
integer(kind=MPI_ADDRESS_KIND) lb, extent

返回数据类型的下限和范围

MPI_Type_get_name(type, type_name, resultlen, ierror)
integer tyep, resultlen, ierror
character*(*) type_name

返回与数据类型对应的名称

MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
integer datatype, ierror
integer(kind=MPI_ADDRESS_KIND) true_lb, true_extent

返回数据类型的真实范围

MPI_Type_match_size(typeclass, size, type, ierror)
integer typeclass, size, type, ierror

返回与指定类型和大小的局部变量匹配的MPI**数据类型**

MPI_type_set_attr(type, type_keyval, attribute_val, ierror)
integer type, type_keyval, ierror
integer(kind=MPI_ADDRESS_KIND) attribute_val

设置在一个数据类型上缓冲的属性关键字的值

MPI_Type_set_name(type, type_name, ierror)
integer type, ierror
character*(*) type_name

把名字和数据类型相关联

MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount, datatype, ierror)

integer outcount, datatype, ierror
integer(kind=MPI_ADDRESS_KIND) insize, position
character*(*) datarep
<type> inbuf(*), outbuf(*)

以指定的格式将数据解包

MPI_Unpublish_name(service_name, info, port_name, ierror)
integer info, ierror
character*(*) service_name, port_name

取消一个以前发布的服务名

MPI_Win_call_errhandler(win, errorcode, ierror)
integer win, errorcode, ierror

激活与窗口对象对应的错误处理程序

MPI_Win_complete(win, ierror)
integer win, ierror

完成从MPI_Win_start**开始的**RMA**访问**

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

创建新窗口对象

MPI_Win_create_errhanlder(function, errhanlder, ierror)
external function
integer errhandler, ierror

创建对附加到窗口对象上的错误处理程序

MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
    extra_state, ierror)
external win_copy_attr_fn, win_delete_attr_fn
integer win_keyval, ierror
integer(kind=MPI_ADDRESS_KIND) extra_state

创建能在窗口对象上缓冲的新属性关键字

MPI_Win_delete_attr(win, win_keyval, ierror)
integer win, win_keyval, ierror

删除在窗口对象上缓冲的属性关键字

MPI_Win_fence(assert, win, ierror)
integer assert, win, ierror

同步窗口对象上的RMA**操作**

MPI_Win_free(win, ierror)
integer win, ierror

释放窗口对象

MPI_Win_free_keyval(win_keyval, ierror)
integer win_keyval, ierror

释放用MPI_Win_create_keyval**创建的属性关键字**

MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
integer win, win_keyval, ierror

integer(kind=MPI_ADDRESS_KIND) attribute_val
logical flag

返回与窗口对象上缓冲的属性关键字对应的值

MPI_Win_get_errhandler(win, errhandler, ierror)
integer iwn, errhandler, ierror

返回与窗口对象对应的出错处理程序

MPI_Win_get_group(win, group, ierror)
integer win, group, irror

返回用来创建窗口对象的通信域组的副本

MPI_Win_get_name(win, win_name, resultlen, ierror)
integer win, resultlen, irror
character*(*) win_name

返回与窗口对象对应的名称

MPI_Win_lock(lock_type, rank, assert, win, ierror)
integer lock_type, rank, assert, win, ierror

对指定进程的窗口加锁

MPI_Win_post(group, assert, win, ierror)
integer group, assert, win, ierror

开始允许其它窗口的远程访问

MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
integer win, win_keyval, ierror
integer(kind=MPI_ADDRESS_KIND) attribute_val

设置窗口对象缓冲的属性关键字的值

MPI_Win_set_errhandler(win, errhandler, ierror)
integer win, errhandler, ierror

把一个新的错误处理程序附加到窗口对象上

MPI_Win_set_name(win, win_name, ierror)
integer win, ierror
character*(*) win_name

将名称与窗口对象关联

MPI_Win_start(group, assert, win, ierror)
integer group, assert, win, ierror

准备访问其它的窗口，实现与MPI_Win_post**的握手**

MPI_Win_test(win, falg, ierror)
integer win, ierror
logical flag

测试窗口对象之上的RMA**操作是否完成**

MPI_Win_unlock(rank, win, ierror)
integer rank, win, ierror

对给定的窗口开锁

MPI_Win_wait(win, ierror)
integer win, ierror

完成用MPI_Win_post**启动的**RMA**访问**

18.5 小结

本章的目的主要是为了查找和使用的方便，故仅仅给出了简单的解释和说明，需要详细了解具体的调用时再查看相关的部分。

第四部分　MPI的最新发展MPI-2

MPI论坛MPIF在1994年推出的MPI的基础上，根据MPI的发展和要求，又推出了MPI的最新版本MPI-2。本章介绍MPI-2相对于MPI-1的新特征，主要包括动态进程管理、远程存储访问和并行文件I/O，以便读者了解MPI的发展和动向。

第19章 动态进程管理

在MPI-1中，一个MPI程序一旦启动，一直到该MPI程序结束，进程的个数是固定的，在程序运行过程中是不可能动态改变的。在MPI-2中，允许在程序运行过程中动态改变进程的数目，并提供了动态进程创建和管理的各种调用，本章介绍各种动态改变进程数目的方法和对动态进程的使用。

组间通信域在动态进程管理中处于核心的地位，只有掌握了它的基本概念，才会准确把握和使用进程的动态特性和动态进程之间的通信。

19.1 组间通信域

在MPI-1中进程如何启动是在MPI程序之外定义的，当MPI_Init返回后进程个数是固定的。PVM的进程管理，一类重要的消息传递应用，如客户/服务系统和任务农场任务，需要对进程进行动态控制。这样的扩展可以用MPI自身来写并行计算环境。

在MPI-1中，虽然提出了组间通信域的概念，但真正使用组间通信域的地方，却是MPI-2提出的动态进程管理。

在MPI-2中，对点到点通信和组通信，都给出了使用组间通信域时的确切含义。在语法上，不管是使用组内还是组间通信域，二者没有任何区别，但其语义是不同的。

对于构成组间通信域的两个进程组，调用进程把自己所在的组看作是本地组，而把另一个组称为远地组。使用组间通信域的一个特点是本地组进程发送的数据被远地组进程接收，而本地组接收的数据必然来自远地组。

在使用组间通信域的点到点通信中，发送语句指定的目的进程是远地组中的进程编号，接收进程指出的源进程编号也是远地组的进程编号。

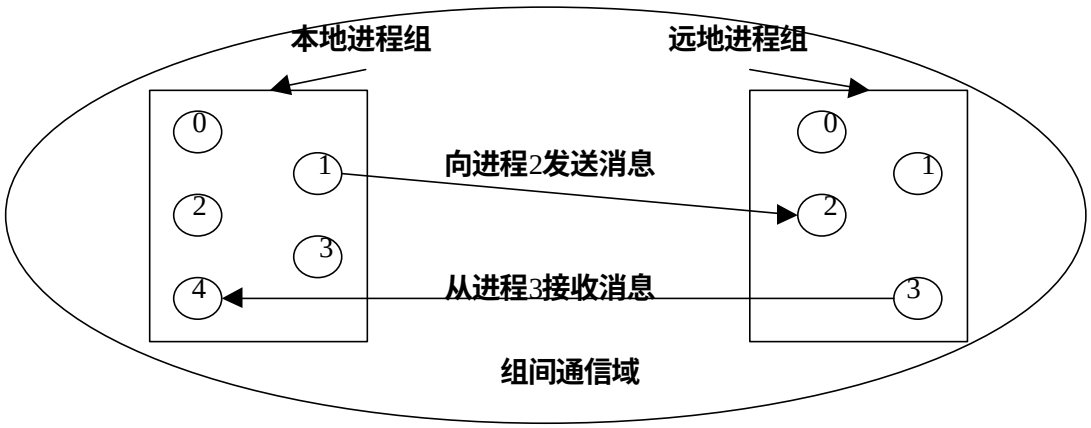


图 78 组间通信域上的点到点通信

对于组通信，如果使用组间通信域，则其含义分不同的形式而有所不同。对于多对多通信，本地进程组的所有进程向远地进程组的所有进程发送数据，同时本地进程组的所有进程从远地进程组的所有进程接收数据。

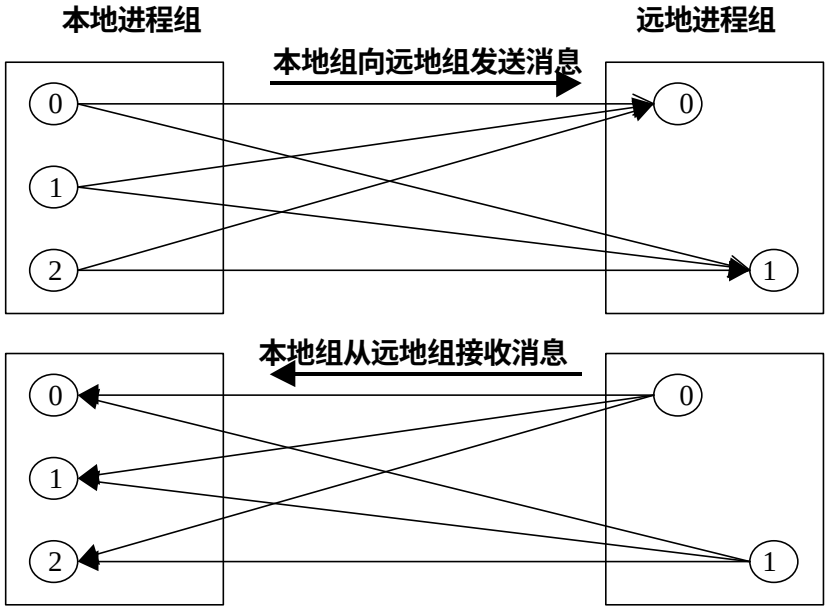


图 79 组间通信域上的多对多组通信

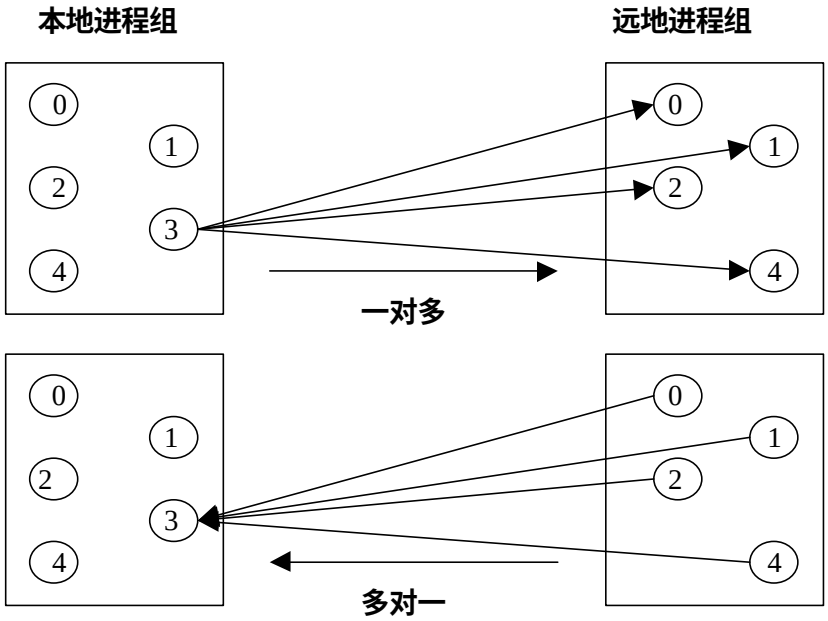


图 80 组间通信域上一对多或多对一通信

对于组间通信域上的一对多操作，本地组的ROOT**进程发送消息，远地组的所有进程都接收；反之，对于组间通信域上的多对一操作，本地组的**ROOT**进程接收消息，而远地组的所有进程都向本地组的**ROOT**进程发送消息。**

对于本地组的非ROOT**进程，它们也要执行此组调用，不过不执行通信操作，本地组的**ROOT**进程和非**ROOT**进程是通过在指定**ROOT**进程是区别开来的，对于**ROOT**进程，在参数**

为ROOT进程标识号的位置写上MPI_ROOT，对于非ROOT进程，在参数为ROOT进程标识号的位置写上MPI_PROC_NULL，对于远地组的进程，则写上本地组ROOT进程的标识号。这是和组内通信域的组通信不同的地方。

利用组间通信域进行通信主要有三种方式：通过动态创建新的进程，父进程和子进程形成的组间通信域上的通信；两个没有父子关系的进程间建立组间通信域进行通信；将socket通信转换为组间通信域上的通信。下面分别进行介绍。

19.2 动态创建新的MPI进程

MPI-2的动态进程的创建是指从已经存在的一个组间通信域进程组的进程，派生出若干个进程形成一个新的进程组，原来的进程组相对于新派生的进程组称为父进程组，而新派生的进程组相对于原来的进程组称为子进程组。父子进程组属于不同的通信域，它们之间的通信是通过父子进程组的通信域形成的组间通信域来进行的。

MPI_COMM_SPAWN(command, argv,maxprocs,info,root,comm,intercomm,array_of_errcodes)
IN command 将派生进程对应的可执行程序名
IN argv 传递给command的参数
IN maxprocs 请求MPI派生的进程的最大个数
IN info 传递给运行时的信息
IN root 负责检查上述参数的进程标识号
IN comm 派生新进程的组内通信域
OUT intercomm 返回的组间通信域，包括原来的进程和新创建的进程
OUT array_of_errcodes 返回的错误代码数组
Int MPI_Comm_spawn(char * command, char ** argv, int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm * intercomm, int * array_of_errcodes)
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM,
INTERCOMM, ARRAY_OF_ERRCODES, IERROR)
INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM ARRAY_OF_ERRCODES(*),
IERROR

MPI调用接口 129 MPI_COMM_SPAWN

MPI派生新的进程是通过调用MPI_COMM_SPAWN实现的。在MPI_COMM_SPAWN中需要指出将派生的进程对应的可执行程序的名字command，传递给可执行程序的参数argv，将要派生的进程的最大个数maxprocs，以及可能的运行时信息info。对于上述这些参数，不需要MPI每个进程都进行解释，只需提供一个解释进程的标识号ROOT即可，由ROOT进程完成对上述参数的解释并传递给其它的进程。此外，还需要指出需要和将派生出来的子进程进行通信的进程形成的组内通信域，即父进程组形成的通信域，此调用返回一个即包括原来的父进程组（组内通信域域comm），又包括新创建的子进程组（组内通信域）的组间通信域intercomm，子进程组的组间通信域没有显式出现。此调用还给出了调用的返回代码信息array_of_errcodes。

此调用存在三种类型的同步，而且必须同时满足才能够进行下面的操作。首先是父进程

之间的同步，必须等到所有的父进程都执行了此调用之后才能够向下执行；然后是子进程之间的同步，即所有派生出来的子进程都执行了各自的初始化调用MPI_INIT之后，子进程才可以执行下面的操作；最后是父子进程之间的同步，必须当所有的父进程都执行了MPI_COMM_SPAWN调用，所有的派生出来的子进程都执行了MPI_INIT调用之后，父子进程才可以执行下面的操作。

父进程组可以通过调用派生进程调用返回的组间通信域和子进程进行通信，子进程是如何得到包括父进程在内的组间通信域的信息呢？它是通过调用MPI_COMM_GET_PARENT得到的。

MPI_COMM_GET_PARENT(parent)
OUT parent 包括子进程和父进程的组间通信域
int MPI_Comm_get_parent(MPI_Comm * parent)
MPI_COMM_GET_PARENT(PARENT, IERROR)
INTEGER PARENT, IERROR

MPI调用接口 130 MPI_COMM_GET_PARENT

MPI_COMM_GET_PARENT的使用非常简单，只需在子进程中执行此调用。就可以得到包括父进程和子进程在内的组间通信域。

当父子进程都拥有了包括所有父子进程的组间通信域，它们就可以通过使用该组间通信域进行通信。

MPI_COMM_SPAWN的另一种更为通用的形式是MPI_COMM_SPAWN_MULTIPLE，可以把MPI_COMM_SPAWN看作是MPI_COMM_SPAWN_MULTIPLE的一个特例。

MPI_COMM_SPAWN_MULTIPLE可以同时创建多组不同的子进程，而不是一组。

MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv,
array_of_max_maxprocs,array_of_info, root, vcomm,intercomm)

IN	count	**进程组的个数**
IN	array_of_commands	**不同的进程组对应的可执行程序**
IN	array_of_maxprocs	**每个组的最大进程数**
IN	array_of_info	**每个组传递给运行时的信息**
IN	root	**解释上述参数的进程的标识号**
IN	comm	**派生新进程的组内通信域**
OUT	intercomm	**由不同子进程组和父进程组形成的组间通信域**
OUT	array_of_errcodes	**不同进程组返回的错误代码**

```
int MPI_Comm_spawn_multiple(int count, char ** array_of_commands, char ***
array_of_argv, int * array_of_maxprocs, MPI_Info * array_of_info, int root, MPI_Comm
  comm, MPI_Comm * intercomm, int * array_of_errcodes)
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS,
ARRAY_OF_ARGV, ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM,
INTERCOMM, ARRAY_OF_ERRCODES, IERROR)
INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*),ROOT,COMM
            INTERCOMM,  ARRAY_OF_ERRCODES,  IERR
CHARACTER *(*)   ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

MPI**调用接口** 131　MPI_COMM_SPAWN_MULTIPLE

MPI_Comm_spawn**创建的子进程是同一个程序的副本，而**MPI_Comm_spawn_multiple**可以创建不同的子进程组，各子进程组使用不同的程序副本，所有这些子进程形成的子进程组和原来的父进程组共同形成一个组间通信域。**

19.3 独立进程间的通信

除了派生新的进程之外，MPI**还允许没有父子关系的独立的进程之间进行通信。它们之间的通信采用客户/服务的方式，对于这两组对立的进程，分别叫做服务端进程组和客户端进程组。**

对于服务端的进程执行如下调用，首先是打开一个端口，

```
MPI_OPEN_PORT(info, port_name)
IN     info              传递给运行时的信息
OUT    port_name         返回的端口名
int MPI_Open_prot(MPI_Info info, char * port_name)
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
CHARACTER  *(*)   PORT_NAME
INTEGER    INFOR, IERROR
```

MPI调用接口 132 MPI_OPEN_PORT

接着，服务端进程在打开的端口上等待客户端进程的连接，

```
MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
IN      port_name    前面打开的端口名
IN      info         传递给运行时的信息
IN      root         服务端进程组的根进程标识号
IN      comm         服务端进程通信域
OUT     newcomm      返回的包括客户端进程和服务端进程的组间通信域
int MPI_Comm_accept(char * port_name, MPI_Info info, int root, MPI_Comm
        comm, MPI_Comm * newcomm, )
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM,
        IERROR)
    CHARACTER  *(*)   PORT_NAME
     INTEGER    INFO, ROOT, COMM,NEWCOMM,IERROR
```

MPI调用接口 133 MPI_COMM_ACCEPT

这一调用的入口参数port_name**和**info**对根结点有用，返回的组间通信域**newcomm**包括服务端进程和客户端进程。通过该组间通信域，服务端进程就可以和客户端进程通信。**

当通信结束后，服务端进程要关闭打开的端口。

```
MPI_CLOSE_PORT(port_name)
IN      port_name      端口号
int MPI_Close_port(char * port_name)
MPI_CLOSE_PORT(PORT_NAME, IERROR)
     CHARACTER  *(*)   PORT_NAME
     INTEGER    IERROR
```

MPI调用接口 134 MPI_CLOSE_PORT

MPI_CLOSE_PORT**将以前打开的端口**port_name**关闭。**

在客户端，要执行如下操作，首先是建立和服务端的连接。

```
MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)
IN      port_name      将连接的端口号
IN      info           传递给运行时的信息
IN      root           执行连接操作的根进程标识号
IN      comm           客户端进程的组内通信域
OUT     newcomm        返回的包括客户端和服务端进程的组间通信域
int MPI_Comm_connect(char * port_name, MPI_Info info, int root, MPI_Comm
    comm, MPI_Comm * newcomm)
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM,
     IERROR)
    CHARACTER  *(*)   PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
```

MPI调用接口 135　MPI_COMM_CONNECT

通过MPI_COMM_CONNECT调用，客户端的进程就可以和打开端口为port_name的服务端进程建立连接。显然端口名port_name和传递给运行时的信息info只对根进程root有意义，通过返回的组间通信域newcomm，客户端进程就可以和服务端进程进行通信。

对于客户端和服务端，当通信结束后，都需要通过调用MPI_COMM_DISCONNECT断开连接。

```
MPI_COMM_DISCONNECT(comm)
INOUT   comm   通信域
int MPI_Comm_disconnect(MPI_Comm * comm)
MPI_COMM_DISCONNECT(COMM, IERROR)
INTEGER   COMM,  IERROR
```

MPI调用接口 136　MPI_COMM_DISCONNECT

将建立在通信域comm上的通信连接断开。

前面对客户/服务方式的连接的建立，端口标识的使用很不方便，它需要客户端进程每次运行时根据服务端得到的端口标识的不同来进行连接，一种改进方法是使用公开名字机制。其主要含义是，在服务端，程序每次启动后，它打开一个端口得到的端口标识有可能是不同的，但是，可以把一个公开的名字和这一新打开的端口标识建立联系，并且公之与众，这样，当客户端的程序每次启动时，先根据公之与众的名字，查找与该名字对应的端口标识，然后再和该端口连接。这样做的好处是对于一组客户/服务程序，给定一个公之与众的名字就可以了，而不必每次都使用具体的端口名字。

MPI_PUBLISH_NAME(service_name, info, port_name)
IN service_name **与端口对应的服务的名字**
IN info **传递给运行时的信息**
IN port_name **端口名**
int MPI_Publish_name(char * service_name, MPI_Info info, char * prot_name)
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
INTEGER INFO, IERROR
CHARACTER *(*) SERVICE_NAME, PORT_NAME

MPI**调用接口** 137 MPI_PUBLISH_NAME

MPI_PUBLISH_NAME**将一个服务名和端口名建立联系，并将该服务名公之与众，以便客户进程的查找。这是服务进程的调用。**

MPI_LOOKUP_NAME(service_name, info, port_name)
IN service_name **公之与众的服务名**
IN info **传递给运行时的信息**
OUT port_name **与服务相联系的端口名**
int MPI_Lookup_name(char * service_name, MPI_Info info, char * port_name)
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
CHARACTER *(*) SERVICE_NAME, PORT_NAME
INTEGER INFO, IERROR

MPI**调用接口** 138 MPI_LOOKUP_NAME

客户端的进程，可以通过公之与众的服务名，得到对应的端口名，从而可以通过该端口与相应的服务建立连接。

MPI_UNPUBLISH_NAME(service_name, info, port_name)
IN service_name
IN info
IN port_name
int MPI_Unpublish_name(char * service_name, MPI_Info info, char * port_name)
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
INTEGER INFO, IERROR
CHARACTER *(*) SERVICE_NAME, PORT_NAME

MPI**调用接口** 139 MPI_UNPUBLISH_NAME

当通信完成后，服务端进程可以取消服务名字和某一端口标识的联系。

19.4 基于socket的通信

MPI还提供了将socket通信转化为组间通信域通信的调用。

MPI_COMM_JOIN(fd, intercomm) IN fd 已建立连接的socket文件句柄 OUT intercomm 根据该socket返回的组间通信域 int MPI_Comm_join(int fd, MPI_Comm * intercomm) MPI_COMM_JOIN(FD, INTERCOMM,IERROR) INTEGER FD, INTERCOMM, IERROR

MPI调用接口 140 MPI_COMM_JOIN

通过MPI_COMM_JOIN调用，将原来通过socket连接的进程，包含在一个组间通信域之内，这样原来socket方式的通信就可以用基于MPI的方式来进行。

19.5 小结

正如本章一开始所介绍的那样，组间通信域是动态进程管理中的一个核心的部分，只要掌握了它，就可以很容易地掌握各种动态进程的管理方法。

掌握使用组间通信域进行点到点和组通信的基本含义之后，其实剩下的任务就是了解如何创建组间通信域，本章给出了三种方式：派生新进程、结合其它程序的进程和通过socket通信来建立。一旦建立了组间通信域，则不管建立前有什么不同，建立后使用组间通信域进行通信的方式是相同的。

第20章 远程存储访问

远程存储访问即直接对非本地的存储空间进行访问。本章介绍MPI-2提供的三种基本的远程存储访问方式即“读”、“写”和“累计”。远程存储访问的最大特点是不需要对方的进程参与通信。MPI-2是通过“窗口”来实现远程存储访问的。

20.1 简介

在MPI-2中增加远程存储访问的能力，主要是为了使MPI在编写特定算法和通信模型的并行程序时，更加自然和简洁。因为在许多情况下，都需要一个进程对另外一个进程的存储区域进行直接访问。

MPI-2对远程存储的访问主要是通过“窗口”来进行的。为了进行远程存储访问，首先需要定义一个“窗口”，该“窗口”开在各个进程的一段本地进程存储空间，其目的是为了让其它的进程可以通过这一窗口来访问本地的数据。

定义好窗口之后，就可以通过窗口来访问远程存储区域的数据了，MPI-2提供了三种基本的访问形式，即“读”、“写”和“累计”。“读”操作只是从远端的窗口获取数据，并不对远端数据进行任何修改；“写”操作将本地的内容写入远端的窗口，它修改远端窗口的内容；而“累计”操作就更复杂一些，它将远端窗口的数据和本地的数据进行某种指定方式的运算之后，再将运算的结果写入远端窗口。

MPI-2就是通过“读”、“写”和“累计”三种操作来实现对远程存储的访问和更新的。除了基本的窗口操作之外，MPI-2还提供了窗口管理功能，用来实现对窗口操作的同步管理。MPI-2对窗口的同步管理有三种方式：1 栅栏方式（fence），在这种方式下，对窗口的操作必须放在一对栅栏语句之间，这样可以保证当栅栏语句结束之后，其内部的窗口操作可以正确完成；2 握手方式，在这种方式下，调用窗口操作的进程需要将具体的窗口调用操作放在以MPI_WIN_START开始，以MPI_WIN_COMPLETE结束的调用之间，相应的，被访问的远端进程需要以一对调用MPI_WIN_POST和MPI_WIN_WAIT与之相适应。MPI_WIN_POST允许其它的进程对自己的窗口进行访问，而MPI_WIN_WAIT调用结束之后可以保证对本窗口的调用操作全部完成。MPI_WIN_START申请对远端进程窗口的访问，只有当远端窗口执行了MPI_WIN_POST操作之后才可以访问远端窗口，MPI_WIN_COMPLETE完成对远端窗口访问操作；3 锁方式，在这种方式下，不同的进程通过对特定的窗口加锁来实现互斥访问，当然用户根据需要可以使用共享的锁，这是就可以允许使用共享锁的进程对同一窗口同时访问。

远端存储的访问，窗口是具体的实现形式，通过窗口操作实现来实现单边通信，通过对窗口的管理操作来实现对窗口操作的同步控制。

20.2 窗口的创建与窗口操作

20.2.1 创建窗口

```
MPI_WIN_CREATE(base, size, disp_unit, info, comm ,win)
IN      base        窗口空间的初始地址
IN      size        以字节为单位的窗口空间大小
IN      disp_unit   一个偏移单位对应的字节数
IN      info        传递给运行时的信息
IN      comm        通信域
OUT     win         返回的窗口对象
int MPI_Win_create(void * base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm,
          MPI_Win  * win)
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
```

MPI调用接口 141 MPI_WIN_CREATE

MPI_WIN_CREATE在本地的一片特定的存储空间之上开辟一个“窗口”，其它的进程就可以通过这一窗口来直接访问该窗口限定的存储空间。该空间的基地址是base，以字节为单位的空间大小是size，在该窗口上以字节为单位的数据类型偏移为disp_unit，传递给运行时的信息为info。如果给定窗口大小size=0，则表示该窗口没有存储区域可以被其它进程访问。

该调用是一个组调用，所有comm通信域之内的进程都要执行，它返回一个窗口对象，该窗口并不只是表示本地窗口，它是建立在整个组上的窗口，通过这一窗口对象并指定组内的特定进程，就可以访问组内的任何一个进程的提供给窗口的存储区域，实现远程存储访问。

```
MPI_WIN_FREE(win)
INOUT     win     窗口对象，输入为要释放的窗口，返回为空
int MPI_Win_free(MPI_Win * win)
MPI_WIN_FREE(WIN, IERROR)
   INTEGER    WIN, IERROR
```

MPI调用接口 142 MPI_WIN_FREE

与MPI_WIN_CREAT 操作相对应，MPI_WIN_FREE将前面创建的窗口对象释放掉，它也是一个组调用，组内的所有进程都需要执行它，它返回后将原来的窗口对象置为空

MPI_WIN_NULL。**在所有的远程存储访问完成后，执行这一操作释放掉不再使用的对象。**

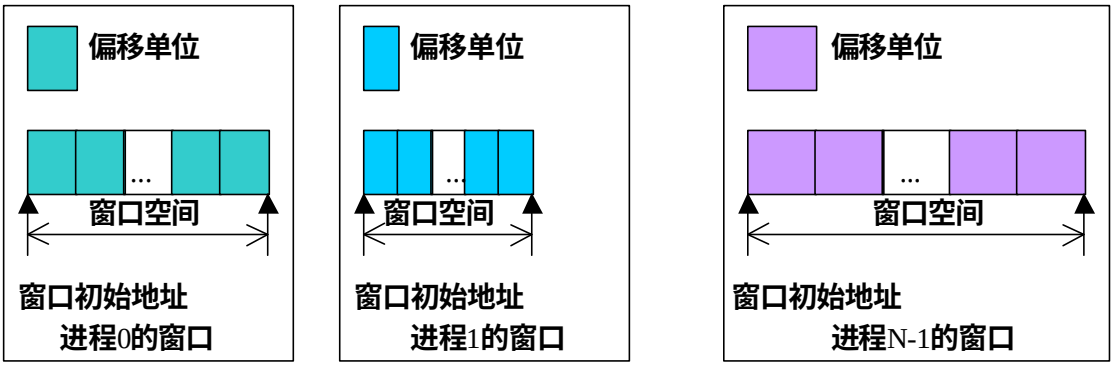


图 81 各进程创建的可供其它进程直接访问的窗口

窗口创建后得到的窗口对象并不是指本进程的窗口，而是指整个组调用的窗口，使用该窗口对象可以访问任何一个进程组内的窗口。

20.2.2 向窗口写

MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
target_count, target_datatype, win)

IN origin_addr **本地发送缓冲区起始地址**

IN origin_count **本地发送缓冲区中将要写到窗口内的数据个数**

IN origin_datatype **本地发送缓冲区中的数据类型**

IN target_rank **目标进程标识**

IN target_disp **相对于写窗口起始地址的偏移单位，从该位置开始写**

IN target_count **以指定的数据类型为单位，写入窗口的数据的个数**

IN target_datatype **写数据的数据类型**

IN win **窗口对象**

```
int MPI_Put(void * origin_addr, int origin_count, MPI_Datatype origin_datatype,
            int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype
            target_datatype, MPI_Win win)
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
            TARGET_RANK, TARGET_DISP, TARGET_COUNT,
            TARGET_DATATYPE, WIN, IERROR)
  <type> ORIGIN_ADDR(*)
  INTEGER (KIND=MPI_ADDRESS_KIND) TARGET_DISP
  INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
            TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR
```

MPI调用接口 143 MPI_PUT

MPI_PUT将本进程中从地址origin_addr开始数据类型为origin_datatype 的origin_count个数据，写入进程号为target_rank的窗口空间，具体位置是从相对与该窗口的起始位置的第target_disp个偏移位置开始， target_address=base+target_disp*disp_unit, 写入target_count个target_datatype类型的数据。其中base和disp_unit是在创建窗口是指定的。

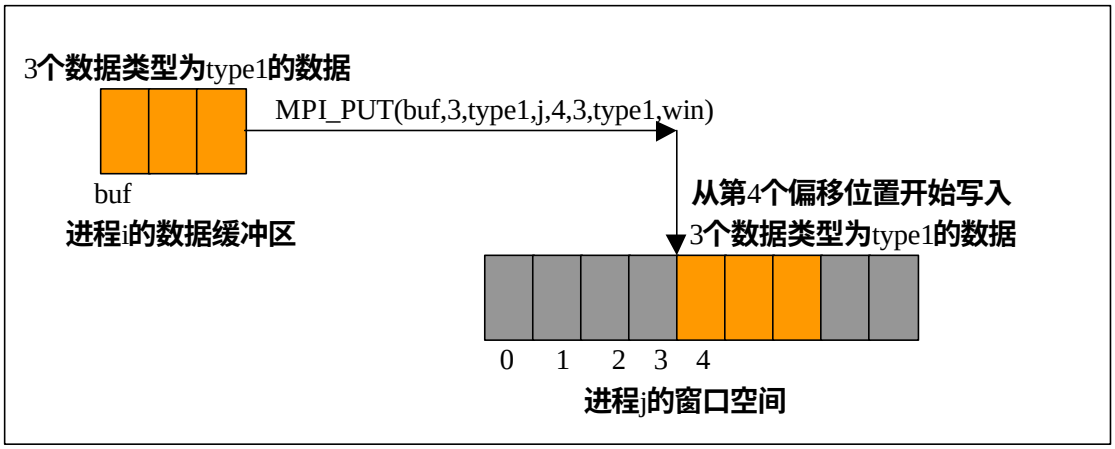


图 82 MPI_PUT操作图示

20.2.3 从窗口读

MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

OUT	origin_addr	本地接收缓冲区的起始地址
IN	origin_count	以指定的数据类型为单位，接收数据的个数
IN	origin_datatype	接收数据的数据类型
IN	target_rank	将要读的窗口所在的进程标识
IN	target_disp	读取位置相对于窗口起始地址偏移单位的个数
IN	target_count	以指定的数据类型为单位，读取数据的个数
IN	target_datatype	读取数据的数据类型
IN	win	窗口对象

```
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int
        target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype
        target_datatype, MPI_Win win)
MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
        TARGET_RANK)
    <type> ORIGIN_ADD(*)
    INTEGER (KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
        TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR
```

MPI调用接口 144 MPI_GET

MPI_GET与MPI_PUT是类似的，只不过数据传送的方向正好相反，它从其它进程的窗口读数据到自己的缓冲区中。具体地， 它从target_rank进程的第target_disp个偏移位置开始，读取target_count个类型为target_datatype的数据，放到本地以origin_add开始的缓冲区中，接收数据的个数为origin_count个，数据类型为origin_datatype。

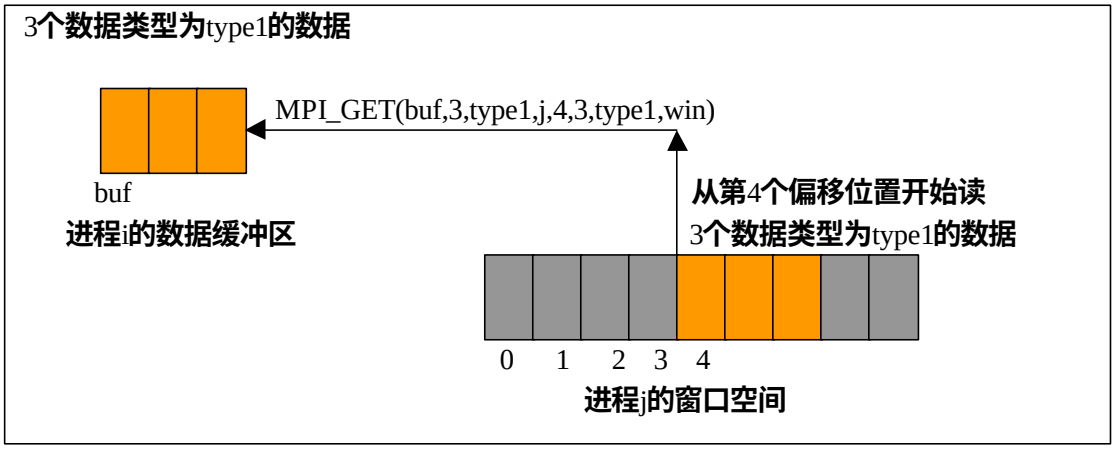


图 83 MPI_GET操作图示

20.2.4 对窗口数据的运算

MPI_ACCUMULATE相对于窗口读或窗口写来说更复杂一些，其基本含义是

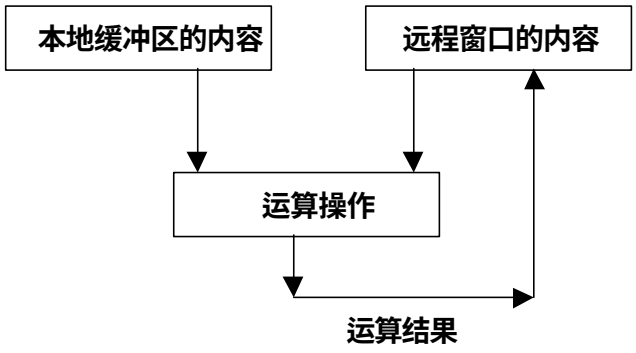


图 84 对窗口数据的运算操作图示

其中窗口待修改数据的个数和类型必须和本地缓冲区中的数据个数和类型相一致，具体的运算操作可以是预定义的各种归约操作

窗口数据运算的具体含义是：将本地缓冲区中从origin_addr开始的origin_count个数据类型为origin_datatype的数据，和目标进程target_rank的窗口内，从target_disp个偏移开始的个数为target_count，数据类型为target_datatype的数据，进行op运算，然后将运算结果存入窗口数据原来的位置。

MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp target_count, target_datatype, op, win)

IN	origin_addr	**本地缓冲区起始地址**
IN	origin_count	**指定数据个数**
IN	origin_datatype	**数据类型**
IN	target_rank	**累计窗口所在的进程标识**
IN	target_disp	**累计数据起始位置相对于窗口开始位置的偏移**
IN	target_count	**窗口中累计数据个数**
IN	target_datatype	**累计数据的数据类型**
IN	op	**具体的累计操作**
IN	win	**窗口对象**

int MPI_Accumulate(void * origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)

<type> ORIGIN_ADDR(*)

INTEGER (KIND=MPI_ADDRESS_KIND) TARGET_DISP

INTEGER ORIGIN_ADDR, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR

MPI**调用接口** 145 MPI_ACCUMULATE

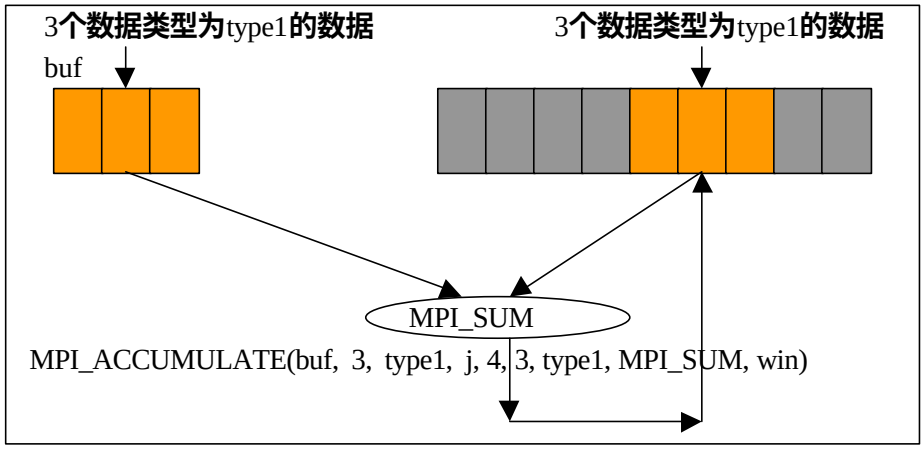


图 85 MPI_ACCUMULATE**操作图示**

此外，还有一些与窗口操作相关的操作，如

```
MPI_WIN_GET_GROUP(win, group)
IN      win      窗口对象
OUT     group    该窗口对象所对应的进程组
int MPI_Win_get_group(MPI_Win win, MPI_Group * group)
MPI_WIN_GET_GROUP(WIN,GROUP, IERROR)
    INTEGER   WIN,  GROUP,  IERROR
```

MPI调用接口 146 MPI_WIN_GET_GROUP

MPI_WIN_GET_GROUP**调用根据窗口对象**win，**返回该对象所对应的进程组**group，**该进程组就是创建窗口对象**win**时所使用的通信域所对应的进程组**。

20.3 窗口同步管理

以上对窗口的操作都是非阻塞操作，窗口数据的一致性需要程序员通过调用特定的窗口同步管理语句来实现。

对窗口的同步管理有三种相互独立的方式，即栅栏方式、握手方式和锁方式，下面分别对它们进行详细介绍。

20.3.1 栅栏方式

它是一种松同步方式，使用这一方式时，对窗口的操作必须放在一对栅栏语句之间，这样，当程序走出栅栏语句划定的区域之后，可以保证栅栏语句之间所有对窗口的操作都已经完成。

```
MPI_WIN_FENCE(assert, win)
IN      assert    程序的声明
IN      win       窗口对象
int MPI_Win_fence(int assert, MPI_Win win)
MPI_WIN_FENCE(ASSERT,WIN, IERROR)
    INTEGER  ASSERT, WIN, IERROR
```

MPI调用接口 147 MPI_WIN_FENCE

MPI_WIN_FENCE**是一个组调用，即**win**进程组内的所有进程，不管它有没有显式的窗口数据操作，都必须得执行这一调用。**

进程0	进程1	进程N-1
MPI_WIN_FENCE	MPI_WIN_FENCE	MPI_WIN_FENCE
MPI_GET		MPI_PUT
MPI_WIN_FENCE	MPI_WIN_FENCE	MPI_WIN_FENCE

图 86　MPI_WIN_FENCE的同步方式

可以把第一个MPI_WIN_FENCE**调用看作是允许下面后续的程序执行窗口操作，而第二个**MPI_WIN_FENCE**调用看作是上面所有关于窗口的操作都已经完成，即窗口数据已经修改完成，或者从窗口读取的数据已经达到本地缓冲区，从而可以使用该缓冲区中的数据或者释放该数据缓冲区。**

MPI_WIN_FENCE**是用作窗口的同步管理，是组调用，但是它不要求所有的组内进程都执行窗口操作，如图中进程**1**就可以没有任何的窗口操作。**

20.3.2 握手方式

握手方式涉及到一对关于窗口做作的进程，即对窗口进行操作的进程和窗口所在的进程，通过二者的握手来达到对窗口的同步管理。

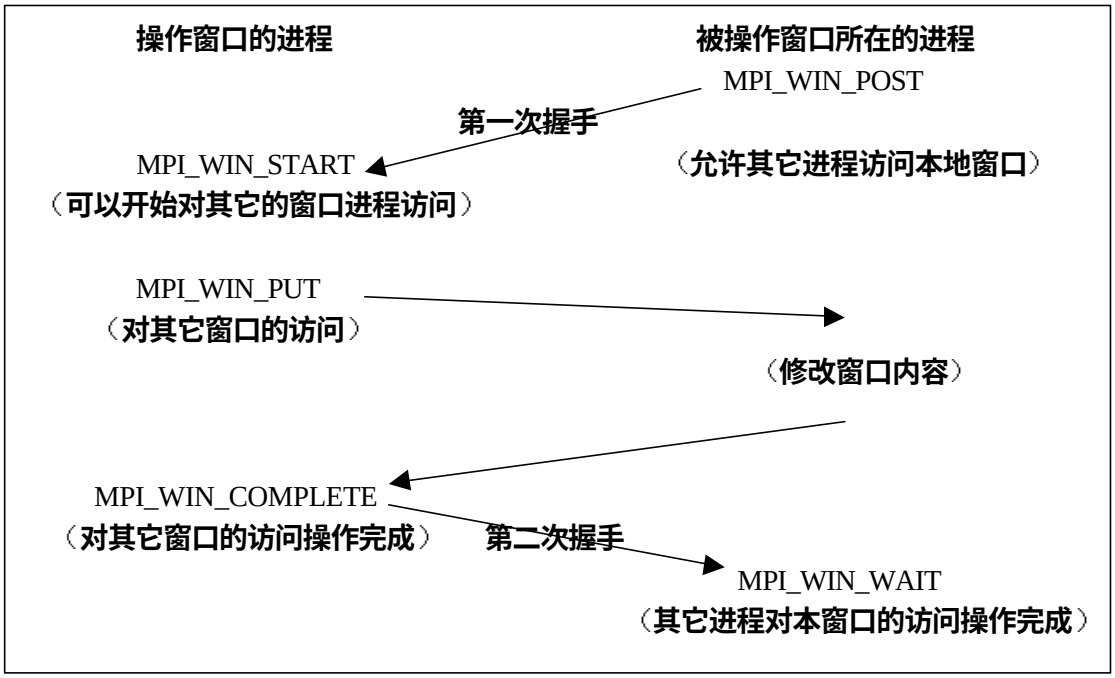


图 87 窗口握手同步方式图示

```
MPI_WIN_START(group, assert, win)
IN      group          进程组
IN      assert         程序的声明
IN      win            窗口对象
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
   INTEGER GROUP, ASSERT, WIN, IERROR
```

MPI调用接口 148 MPI_WIN_START

在握手方式下，只有当调用了MPI_WIN_START**后，才可以对进程组**group**内其它进程的窗口进行不同的窗口操作。**assert**主要是用于实现的优化。**

```
MPI_WIN_COMPLETE(win)
IN          win     窗口对象
int MPI_Win_complete(MPI_Win win)
MPI_WIN_COMPLETE(WIN, IERROR)
     INTEGER  WIN,  IERROR
```

MPI调用接口 149 MPI_WIN_COMPLETE

在握手方式下，对于每一个MPI_WIN_START**调用，都必须有一个相应的**MPI_WIN_COMPLETE**调用与之相匹配，在这两个调用之间，是对窗口的具体操作语句。当这一调用结束后，意味着前面对窗口的各种操作都已经完成。**

```
MPI_WIN_POST(group, assert, win)
IN      group
IN      assert
IN      win
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
MPI_WIN_POST(GROUP, ASSERT, WIN)
     INTEGER   GROUP,  ASSER5T, WIN, IERROR
```

MPI调用接口 150 MPI_WIN_POST

当某进程执行了MPI_WIN_POST**调用之后，意味着从该调用之后，本地的窗口向其它进程开启，其它的进程可以对本进程的窗口进行远程访问，**MPI_WIN_POST**调用和试图访问本地窗口的**MPI_WIN_START**相握手，握手成功则意味着双方达成一致，一方允许远程访问，一方可以进行远程访问。**

```
MPI_WIN_WAIT(win)
IN      win      窗口对象
int MPI_Win_wait(MPI_Win win)
MPI_WIN_WAIT(WIN, IERROR)
    INTEGER  WIN,  IERROR
```

MPI调用接口 151 MPI_WIN_WAIT

MPI_WIN_WAIT完成从前面一个MPI_WIN_POST开始的对本地窗口的所有远程访问操作，该调用的结束意味着前面所有对本进程远程窗口访问的完成。它和远程访问本进程的进程中的MPI_WIN_COMPLETE调用相握手，一旦握手成功则意味着双方的远程窗口访问操作都已经成功完成。

```
MPI_WIN_TEST(win,flag)
IN      win     窗口对象
OUT     flag    完成标志
int MPI_Win_test(MPI_Win win, int * flag)
MPI_WIN_TEST(WIN,FLAG,IERROR)
      INTEGER   WIN,  IERROR
      LOGICAL   FLAG
```

MPI调用接口 152 MPI_WIN_TEST

MPI_WIN_TEST操作和前面介绍的非阻塞通信中的各种TEST操作相类似，它探测关于窗口的非阻塞操作是否完成。如果该调用结束后，返回标志flag=true，则其效果和MPI_WIN_WAIT完全一样，若flag=false，则表示对窗口的非阻塞操作还没有结束，当然也就不能访问窗口中的数据，可以等待一段时间后再进行探测或者直接调用MPI_WIN_WAIT完成对窗口的操作。

20.3.3 锁方式

锁方式用于多个进程对同一个窗口进行远程存储访问，为了协调这多个进程之间的关系而采取的一种方法。这种方法的基本思想在各种不同的领域都得到广泛的应用。

在一定程度上，它类似于操作系统中临界区的概念，可以把待访问的窗口看作是临界资源，在同一个时刻只允许一个进程对它进行访问，在访问前，该进程必须为该窗口加上一把锁，这样在它访问该窗口期间，其它的进程就不能访问该窗口，当这一进程对该窗口的访问结束后，再把锁打开，允许其它的进程再对该窗口进行加锁和访问。

通过加锁方式，可以避免多个进程同时对同一窗口访问造成数据的不一致。

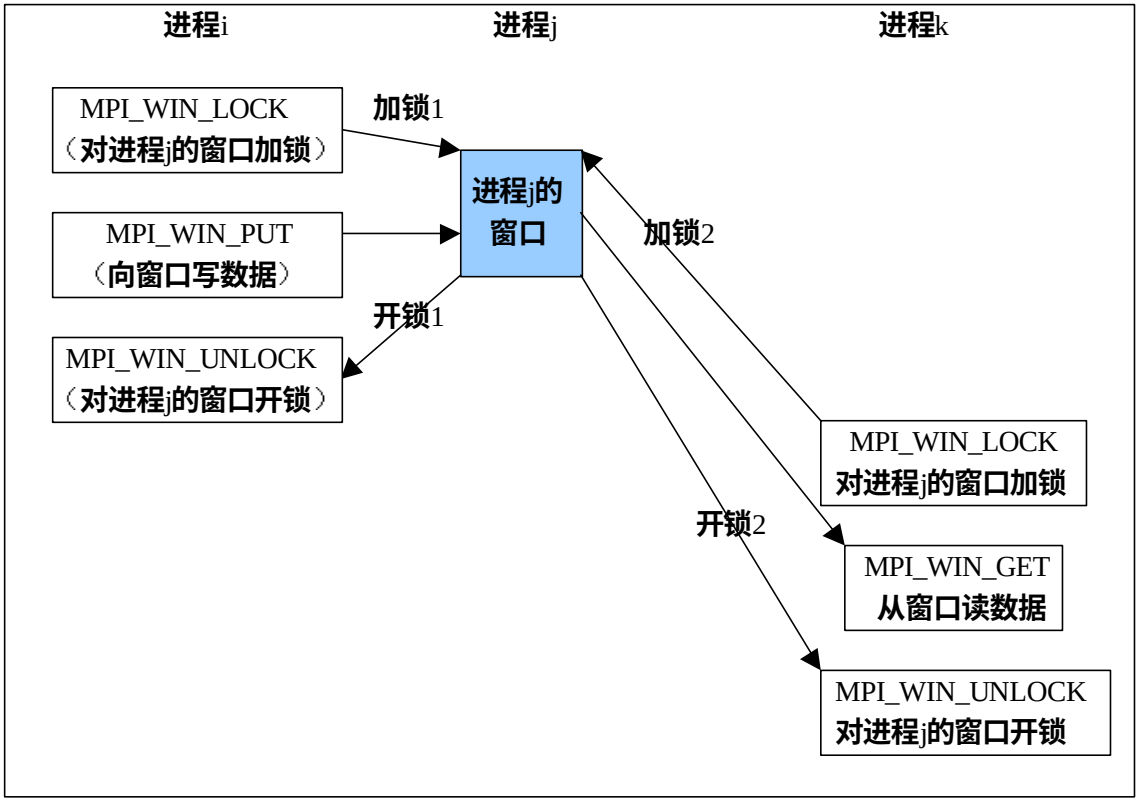


图 88 通过加锁与开锁实现对同一窗口的互斥访问

MPI_WIN_LOCK(lock_type, rank, assert, win)
IN lock_type **锁类型**
IN rank **加锁窗口所在的进程标识号**
IN assert **程序的声明**
IN win **窗口对象**
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR

MPI**调用接口** 153 MPI_WIN_LOCK

MPI_WIN_LOCK**对指定进程的窗口加锁，一旦加锁成功，该进程就可以对另一个进程的远程窗口进行访问。加锁的类型有两种，一种是互斥型，一种是共享型。对于互斥型的锁，一旦加上就不允许其它的进程对该远程窗口进行任何访问操作，这样可以确保当本进程在访问该窗口时，不会因为其它进程的介入而造成数据的不一致。如果是共享型的锁，则其它的进程只能加共享的锁，由程序员来保证多个进程对该窗口共享访问的一致性（比如可以有多个进程同时对一个窗口进行读取，当不运行其它的进程对该窗口进行写操作）。**

MPI_WIN_UNLOCK(rank, win)
IN rank **被开锁窗口的进程标识号**
IN win **窗口对象**
int MPI_Win_unlock(int rank, MPI_Win win)
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
INTEGER RANK, WIN, IERROR

MPI**调用接口** 154 MPI_WIN_UNLOCK

MPI_WIN_UNLOCK**打开由本进程对**rank**进程窗口的加锁，从而允许其它的进程对该窗口进行加锁和访问操作。它和**MPI_WIN_LOCK**是严格匹配的，有一个加锁操作，就必然有一个相应的开锁操作。**

20.4 小结

窗口是远程存储访问中的重要概念，其实MPI-2**的远程存储访问就是各进程将自己的一部分内存区域开辟成其它所有进程都可以访问的窗口，从而使其它的进程实现对自己数据的远程访问，窗口操作是相对简单的，对窗口访问的同步控制是需要注意的问题。**

第21章 并行I/O

在MPI-1中对并行文件I/O根本就没有定义，其原因就在于并行I/O过于复杂，很难找到一个统一的标准。但是，I/O是许多应用中不可缺少的部分，MPI-2在大量实践的基础上，终于提出了一套并行I/O的标准接口。

本章介绍对文件的各种并行操作和管理方法，主要包括直接对文件指定位置数据的读写、对文件视口的读写以及共享文件的读写。在本章的最后，针对分布式数组，给出了分布式数组文件的读写方法。

21.1 概述

MPI-2提供的关于并行文件I/O的调用十分丰富，根据读写定位方法的不同，可以分为三种：1 指定显式的偏移，这种调用没有文件指针的概念，每次读写操作都必须明确指定读写文件的位置；2 各进程拥有独立的文件指针，这种方式的文件操作不需要指定读写的位置，每一个进程都有一个相互独立的文件指针，读写的起始位置就是当前指针的位置，读写完成后文件指针自动移到下一个有效数据的位置。这种方式的文件操作，需要每一个进程都定义各自在文件中的“视口”，视口数据是文件连续或不连续的一部分，各个进程对视口的操作，就如同是对一个打开的独立的连续文件的操作一样；3 共享文件指针，在这种情况下，每一个进程对文件的操作，都是从当前共享文件指针的位置开始，操作结束后共享文件指针自动转移到下一个位置，共享指针位置的变化对所有进程都是可见的，各进程使用的是同一个文件指针，任何一个进程对文件的读写操作都会引起其它所有进程文件指针的改变。

根据同步机制的不同，对文件的操作又可以分为阻塞和非阻塞两大类。对于阻塞调用，对文件的读写调用返回后，可以使用读入数据缓冲区中的数据或者文件已经被更新，但是对于非阻塞文件读写，如同非阻塞通信类似，读写调用的返回并不意味着读写调用的完成，需要调用相应的完成语句来保证读写操作的完成。对于非阻塞通信，又进一步细分为单步法和两步法，单步法的含义是指MPI只提供非阻塞文件读写的开始操作，不提供完成操作；对于两步法，MPI明确提供非阻塞文件读写的开始和完成语句，将对文件读写操作的调用分为明显的两步。对于单步法，其实也需要完成调用，只不过它使用的是和非阻塞通信一样的MPI_WAIT之类的完成方式，而不是特别的对文件操作的完成方式。只有对组读写，才可以使用两步法。

根据对参加读写操作的进程的限制，可以分为独立读写和组读写。所谓独立读写就是单个进程可以实现的读写操作，不需要其它进程的参与；而组读写则要求所有的进程都必须执行相同的读写调用，但是提供给该调用的读写参数可以不同。

以上任意一种读写定位方式，任意一种同步机制，都有独立读写和组读写的调用。但是，在MPI-2中，对于非阻塞的组读写，只有两步法，不存在单步法调用。这是因为对于非阻塞的组调用，使用MPI提供的将读写明显分开的两步法，可以提供给MPI更多的优化机会，有利于对非阻塞组读写的高效实现。

注意在文件的各种组调用中，并没有给出进程组或通信域，该调用所适用的进程组是由调用使用的文件句柄决定的。因为文件打开时，需要给出通信域参数，因此与文件句柄相联系的通信域就是组读写所使用的通信域。

下面的表格对本章介绍的MPI文件读写调用进行了简单地分类汇总和对比。

表格 20 各种并行文件I/O调用

读写定位方法	同步机制		各进程间的关系	
			独立读写	组读写
指定显式偏移	阻塞		READ_AT WRITE_AT	READ_AT_ALL WRITE_AT_ALL
	非阻塞	单步法	IREAD_AT IWRITE_AT	
		两步法		READ_AT_ALL_BEGIN READ_AT_ALL_END WRITE_AT_ALL_BEGIN WRITE_AT_ALL_END
独立的文件指针	阻塞		READ WRITE	READ_ALL WRITE_ALL
	非阻塞	单步法	IREAD IWRITE	
		两步法		READ_ALL_BEGIN READ_ALL_END WRITE_ALL_BEGIN WRITE_ALL_END
共享文件指针	阻塞		READ_SHARED WRITE_SHARED	READ_ORDERED WRITE_ORDERED
	非阻塞	单步法	IREAD_SHARED IWRITE_SHARED	
		两步法		READ_ORDERED_BEGIN READ_ORDERED_END WRITE_ORDERED_BEGIN WRITE_ORDERED_END

21.2 并行文件管理的基本操作

MPI_FILE_OPEN(comm, filename, amode, info, fh)
IN comm **组内通信域**
IN filename **将打开的文件名**
IN amode **打开方式**
IN info **传递给运行时的信息**
OUT fh **返回的文件句柄**
int MPI_File_open(MPI_Comm comm, char * filename, int amode, MPI_Info info,
MPI_File * fh)
MPI_FILE_OPEN(COMM,FILENAME, AMODE, INFO, FH,IERROR)
CHARACTER *(*) FILENAME
INTEGER COMM, AMODE, INFO, FH, IERROR

MPI**调用接口** 155 MPI_FILE_OPEN

MPI_FILE_OPEN**是一个组调用，即通信域**comm**中的所有进程都必须按给定的文件名**

filename来执行它，各进程使用的文件名filename要相同，给定的打开方式amode也要相同，但是各进程传递给运行时的信息info可以互不相同。该调用返回一个文件句柄fh，以后各进程对文件的具体操作都是通过文件句柄fh来实现的。

根据用途的不同，打开方式可以是表格 21所示的任何9种方式之一。

表格 21 文件打开方式

打开方式	含义
MPI_MODE_RDONLY	只读
MPI_MODE_RDWR	读写
MPI_MODE_WRONLY	只写
MPI_MODE_CREATE	若文件不存在则创建
MPI_MODE_EXCL	创建不存在的新文件，若存在则错
MPI_MODE_DELETE_ON_CLOSE	关闭时删除
MPI_MODE_UNIQUE_OPEN	不能并发打开
MPI_MODE_SEQUENTIAL	文件只能顺序存取
MPI_MODE_APPEND	追加方式打开，初始文件指针指向文件尾

```
MPI_FILE_CLOSE(fh)
INOUT     fh     前面打开的文件句柄
int MPI_File_close(MPI_File * fh)
MPI_FILE_CLOSE(FH,IERROR)
     INTEGER  FH,  IERROR
```

MPI调用接口 156 MPI_FILE_CLOSE

MPI_FILE_CLOSE关闭前面已经打开的与句柄fh相联系的文件，它也是一个组调用，即所有打开该文件的进程，也必须都执行关闭操作。这里虽然没有明确指出通信域或进程组，但是，文件句柄fh已包含了进程组的信息。

```
MPI_FILE_DELETE(filename, info)
IN        filename        将删除的文件名
IN        info            传递给运行时的信息
int MPI_File_delete(char * filename, MPI_Info info)
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
     CHARACTER *(*) FILENAME
    INTEGER  INFO,  IERROR
```

MPI调用接口 157 MPI_FILE_DELETE

MPI_FILE_DELETE删除指定的文件filename。

```
MPI_FILE_SET_SIZE(fh,size)
INOUT     fh          文件句柄
IN        size        指定新的文件大小（以字节为单位）
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
   INTEGER FH, IERROR
  INTEGER (KIND=MPI_OFFSET_KIND) SIZE
```

MPI调用接口 158 MPI_FILE_SET_SIZE

MPI_FILE_SET_SIZE强制指定句柄fh对应文件的大小为size。这是一个组调用，即组内所有的进程都执行该操作，并且各进程必须使用相同的文件大小size。

```
MPI_FILE_PREALLOCATE(fh, size)
INOUT     fh        将要预分配空间的文件句柄
IN        size      预分配空间的大小
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
  INTEGER   FH, IERROR
 INTEGER (KIND=MPI_OFFSET_KIND) SIZE
```

MPI调用接口 159 MPI_FILE_PREALLOCATE

MPI_FILE_PREALLOCATE确保与句柄fh相联系的文件分配到大小为size字节的空间。它是一个组调用，所有组内的进程都执行它并且给定相同的size大小。若原来文件的大小已经大于size，在该调用对原来的文件没有影响；若原来文件的大小小于size，则扩展文件的大小为size，扩展的部分看作是写入了未定义的数据。

```
MPI_FILE_GET_SIZE(fh,size)
IN        fh          想知道大小的文件句柄
OUT       size        返回的文件大小
int MPI_File_get_size(MPI_File fh, MPI_Offset * size)
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
    INTEGER   FH, IERROR
   INTEGER (KIND=MPI_OFFSET_KIND) SIZE
```

MPI调用接口 160 MPI_FILE_GET_SIZE

MPI_FILE_GET_SIZE返回以字节为单位的文件大小。

```
MPI_FILE_GET_GROUP(fh,group)
IN      fh      文件句柄
OUT     group   返回的与该句柄联系的进程组
int  MPI_File_get_group(MPI_File fh, MPI_Group * group)
MPI_FILE_GET_GROUP( FH, GROUP, IERROR)
   INTEGER FH, GROUP, IERROR
```

MPI调用接口 161 MPI_FILE_GET_GROUP

MPI_FILE_GET_GROUP返回预文件句柄fh相联系的进程组group，该进程组就是打开该文件时通信域comm对应的进程组。

```
MPI_FILE_GET_AMODE(fh, amode)
IN      fh      文件句柄
OUT     amode   返回的该文件的打开模式
int  MPI_File_get_amode(MPI_File fh, int * amode)
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER   FH,  AMODE, IERROR
```

MPI调用接口 162 MPI_FILE_GET_AMODE

MPI_FILE_GET_AMODE返回文件句柄fh打开时所使用的打开模式amode。

```
MPI_FILE_SET_INFO(fh, info)
INOUT   fh    文件句柄
IN      info  将设置的运行时信息
int MPI_File_set_info(MPI_File fh, MPI_Info info)
MPI_FILE_SET_INFO(FH, INFO, IERROR)
   INTEGER FH, INFO, IERROR
```

MPI调用接口 163 MPI_FILE_SET_INFO

MPI_FILE_SET_INFO将优化和提示信息传递给对象fh，在对文件进行操作时，fh可以根据运行时信息进行特定的优化。

```
MPI_FILE_GET_INFO(fh, info_used)
IN      fh          文件句柄
OUT     info_used   返回的运行时信息
int MPI_File_get_info(MPI_file fh, MPI_Info * info_used)
MPI_FILE_GET_INFO(FH, INFO_USED,IERROR)
  INTEGER FH,INFO_USED, IERROR
```

MPI调用接口 164 MPI_FILE_GET_INFO

MPI_FILE_GET_INFO**返回与文件句柄**fh**相联系的运行时信息**info_used。

21.3 显式偏移的并行文件读写

一种并行文件读写的方法是每一个进程都清楚地知道它将要处理的数据在文件种的准确位置，从而所有的进程可以同时对文件的不同部分进行读写，实现文件的并行操作。

在显式偏移方式下，读取位置是作为一个参数传递给读写调用的，不需要文件指针，当然也不需要文件指针的更新。即在这种方式下，是没有文件指针的概念的，当然也不需要文件指针的移动操作SEEK**，它可以直接从任意指定的位置开始读写。**

21.3.1 阻塞方式

MPI_FILE_READ_AT(fh, offset,buf,count,datatype,status)

IN	fh	**文件句柄**
IN	offset	**读取位置相对于文件头的偏移**
OUT	buf	**读取数据存放的缓冲区**
IN	count	**读取数据个数**
IN	datatype	**读取数据的数据类型**
OUT	status	**返回的状态参数**

int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void * buf, int count,
MPI_Datatype datatype, MPI_Statye * status)

MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT,DATATYPE,STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI**调用接口** 165 MPI_FILE_READ_AT

MPI_FILE_READ_AT **从文件**fh**中，从指定的偏移位置**offset**开始，读取**count**个数据类型为**datatype**的数据，存放到数据缓冲区**buf**之中，**status**是该读写操作完成后返回的状态参数。**

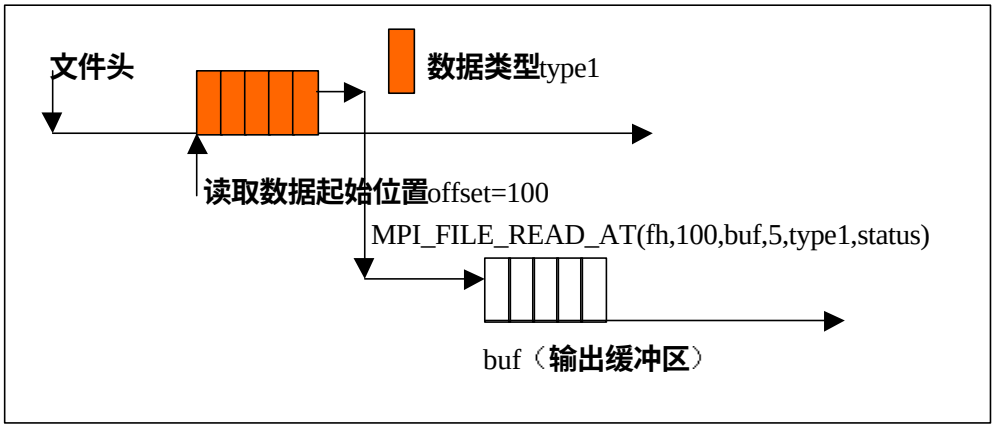


图 89　MPI_FILE_READ_AT 图示

MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype,status)

INOUT	fh	**文件句柄**
IN	offset	**写入文件数据的起始偏移地址**
IN	buf	**将写入数据存放缓冲区的起始地址**
IN	count	**写入数据的个数**
IN	datatype	**写入数据的数据类型**
OUT	status	**写入操作完成后返回的状态信息**

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void * buf, int count,
      MPI_Datatype datatype, MPI_Status * status)
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS,
       IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
  INTEGER (KIND=MPI_OFFSET_KIND) OFFSET
```

MPI调用接口 166　MPI_FILE_WRITE_AT

MPI_FILE_WRITE_AT **与**MPI_FILE_READ_AT **相对应，它向文件句柄**fh **对应中，从指定的位置**offset **开始，将数据缓冲区**buf **中**count **个类型为**datatype **的数据，写入到该文件中，其中**status **是返回的状态参数。**

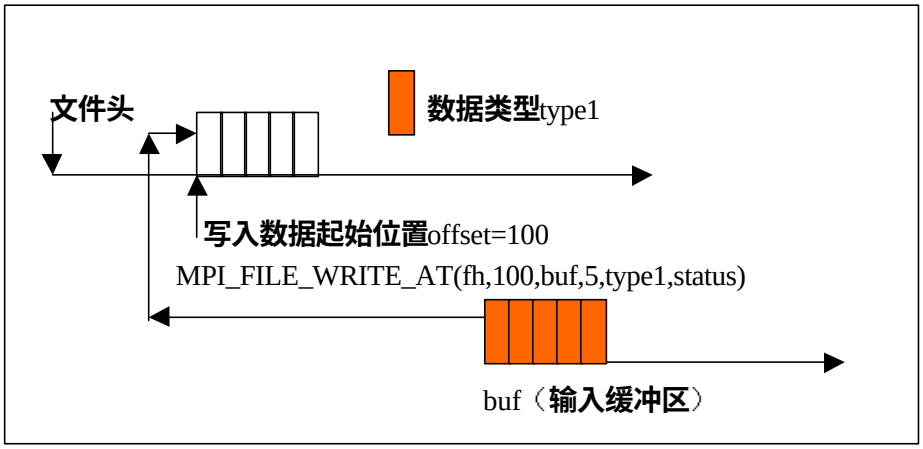


图 90 MPI_FILE_WRITE_AT 图示

MPI_FILE_READ_AT_ALL(fh, offset,but,count,datatype,status)

IN	fh	**读取文件的文件句柄**
IN	offset	**读取数据的起始偏移位置**
OUT	buf	**存放读取数据的缓冲区**
IN	count	**读取数据个数**
IN	datatype	**读取数据的数据类型**
OUT	status	**读取操作完成后的状态**

```
int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count,
      MPI_Datatype datatype, MPI_Status * status)
MPI_FILE_READ_AT_ALL(FH, OFFSET,BUF,COUNT,DATATYPE,STATUS, IERROR)
      <type>   BUF(*)
      INTEGER   FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
      INTEGER  (KIND=MPI_OFFSET_KIND)   OFFSET
```

MPI调用接口 167 MPI_FILE_READ_AT_ALL

MPI_FILE_READ_AT_ALL**是组调用，即与文件句柄**fh**相联系的进程组中的所有进程都要执行此读取调用，就如同每个进程都执行了一个相应的**MPI_FILE_READ_AT **操作**。

MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)

INOUT	fh	**写入文件的句柄**
IN	offset	**写入数据的开始位置**
IN	buf	**存放将写入数据的缓冲区**
IN	count	**写入数据的个数**
IN	datatype	**写入数据的数据类型**
OUT	status	**写入完成后返回的状态**

```
int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void * buf, int count,
      MPI_Datatype datatype, MPI_Status * status)
MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE,
      STATUS, IERROR)
      <type> BUF(*)
      INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE),
                  IERROR
      INTEGER (KIND=MPI_OFFSET_KIND) OFFSET
```

MPI**调用接口** 168 MPI_FILE_WRITE_AT_ALL

MPI_FILE_WRITE_AT_ALL**操作和**MPI_FILE_READ_AT_ALL**相对应，它也是一个组调用，即与文件句柄**fh**相联系的所有进程都必须执行此调用，它如同每一个进程都执行了一个相应的**MPI_FILE_WRITE_AT **操作一样**。

21.3.2 非阻塞方式

和通信的非阻塞方式类似，文件的非阻塞调用返回后，并不意味着对文件的读写操作已经完成，只有当对相应的非阻塞对象进行测试，发现该非阻塞操作已经结束后，才表示对文件的读写操作全部完成。这时才可以使用从文件读取的数据或释放写入文件的输出缓冲区。前面非阻塞通信中的各种对非阻塞对象的操作在对文件的非阻塞调用中得到的非阻塞对象上也同样适用。

MPI_FILE_IREAD_AT **是**MPI_FILE_READ_AT **的非阻塞形式，它从文件**fh**中读取数据，读取数据的起始位置是**offset**，个数为**count**个，数据类型为**datatype**，读取的数据放到数据缓冲区**buf**中，返回一个非阻塞读取完成对象**request**。同所有的非阻塞调用一样，该调用执行后立即返回，不管数据读取是否已经完成。读取操作的完成是通过对**request**对象调用**MPI_WAIT**实现的。如同通信操作的非阻塞调用一样，也可以用**MPI_TEST**调用来查看该非阻塞读取是否完成**。

MPI_FILE_IREAD_AT(fh, offset,buf, count, datatype, request)
IN fh **读取文件的句柄**
IN offset **读取数据的偏移位置**
OUT buf **存放读取数据的缓冲区**
IN count **读取数据个数**
IN datatype **读取数据类型**
OUT request **返回的非阻塞读取完成对象**
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void * buf, int count,
MPI_Datatype datatype, MPI_Request * request)
MPI_FILE_IREAD_AT(FH,OFFSET,BUF,COUNT,DATATYPE,REQUEST,IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
INTEGER (KIND=MPI_OFFSET_KIND) OFFSET

MPI**调用接口** 169 MPI_FILE_IREAD_AT

MPI_FILE_IWRITE_AT(fh, offset,buf, count, datatype, request)
INOUT fh **写入文件的句柄**
IN offset **写入的起始位置**
IN buf **写入数据缓冲区**
IN count **写入数据的个数**
IN datatype **写入数据的数据类型**
OUT request **返回的非阻塞写入完成对象**
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void * buf, int count,
MPI_Datatype datatype, MPI_Request * request)
MPI_FILE_IWRITE_AT(FH,OFFSET,BUF,COUNT,DATATYPE,REQUEST,IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
INTEGER (KIND=MPI_OFFSET_KIND) OFFSET

MPI**调用接口** 170 MPI_FILE_IWRITE_AT

MPI_FILE_IWRITE_AT **是**MPI_FILE_WRITE_AT **的非阻塞形式，它向文件**fh**中写入数据，写入数据在文件中的起始位置是**offset**，写入数据的数据个数是**count**，写入数据的数据类型是**datatype**，将写入文件中的数据存放在缓冲区**buf**中，该调用返回的非阻塞写入完成对象是**request**。这一调用执行后立即返回，真正写入操作的完成可以通过对**request**对象调用**MPI_WAIT**操作实现。**

21.3.3 两步非阻塞组调用

对于非阻塞的组读写调用，MPI-2提供了特殊的调用形式，即两步非阻塞组调用形式。其含义是，在非阻塞组读写调用的开始，执行“开始”读写组调用语句，在非阻塞组读写调用的结束，执行“完成”读写组调用语句。在功能上，“完成”读写组调用语句和相应的MPI_WAIT语句是非常接近的。

所谓的两步非阻塞组调用，就是将原来一个完整的组调用分成两步，第一布是启动该非阻塞组调用，第二步是完成该非阻塞组调用。由于有了第二步的调用，在两步非阻塞调用中，就不需要再象其它的非阻塞调用那样执行MPI_WAIT之类的操作来完成非阻塞调用。

两步非阻塞组调用是一种形式严格的非阻塞组调用方法，使用这种方法有助于对这一调用的优化实现。

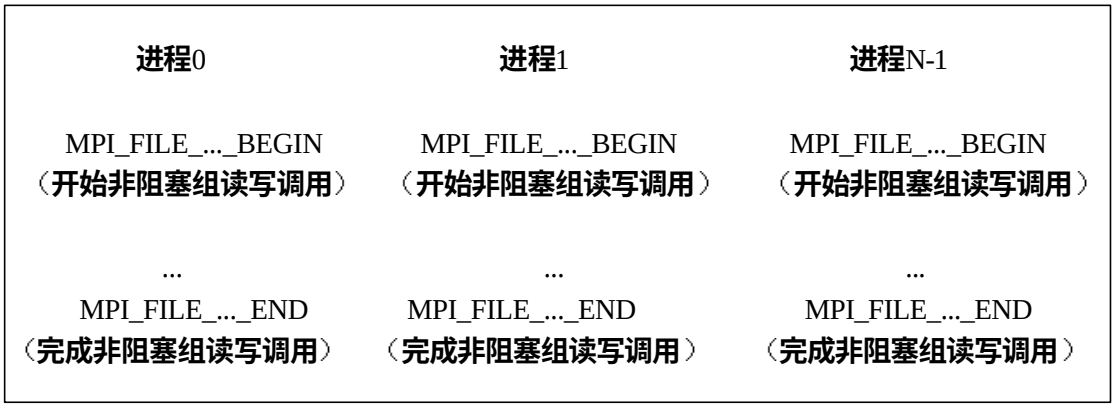


图 91 两步非阻塞组调用图示

MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

IN	fh	读取文件的文件句柄
IN	offset	读取数据的偏移位置
OUT	buf	读取数据将要存放的缓冲区
IN	count	读取数据个数
IN	datatype	读取数据的数据类型

```
int  MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void * buf,
       int count, MPI_Datatype datatype)
MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE,
                  IERROR)
       <type>   BUF(*)
       INTEGER   FH,  COUNT, DATATYPE, IERROR
       INTEGER  (KIND=MPI_OFFSET_KIND)  OFFSET
```

MPI调用接口 171 MPI_FILE_READ_AT_ALL_BEGIN

MPI_FILE_READ_AT_ALL_BEGIN“开始”一个非阻塞的读组调用，与文件句柄fh对应的进程组内的进程都从各自进程指定的偏移位置offset开始，读取count个类型为datatype的

数据，并且将结果存放在buf中。这一语句的完成要通过进程组内各进程都执行MPI_FILE_READ_AT_ALL_END来实现，即只有当执行了MPI_FILE_READ_AT_ALL_END调用，各进程才可以访问buf缓冲区中的数据。

```
MPI_FILE_READ_AT_ALL_END(fh, buf, status)
IN              fh          读取数据的文件句柄
OUT             buf         读取数据存放的缓冲区
OUT             status      该调用完成后返回的状态信息
int  MPI_File_read_at_all_end(MPI_File fh, void * buf, MPI_Status *status)
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
        <type>    BUF(*)
        INTEGER    FH,  STATUS(MPI_STATUS_SIZE), IERROR
```

MPI调用接口 172　MPI_FILE_READ_AT_ALL_END

MPI_FILE_READ_AT_ALL_END调用完成前面由MPI_FILE_READ_AT_ALL_BEGIN开始的非阻塞组调用。其中fh和buf和MPI_FILE_READ_AT_ALL_BEGIN中应完全一样。本调用结束后，从文件中读取的数据，已全部存放在buf中，下面的代码就可以使用这些数据了。

```
MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
INOUT       fh              写入文件的文件句柄
IN          offset          写入数据的起始位置
IN          buf             写入数据存放的缓冲区
IN          count           写入数据的个数
IN          datatype        写入数据的数据类型
int  MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void * buf,
        int count, MPI_Datatype datatype)
MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE,
                IERROR)
        <type>    BUF(*)
        INTEGER    FH,  COUNT, DATATYPE, IERROR
        INTEGER  (KIND=MPI_OFFSET_KIND)  OFFSET
```

MPI调用接口 173　MPI_FILE_WRITE_AT_ALL_BEGIN

MPI_FILE_WRITE_AT_ALL_BEGIN“开始”一个非阻塞的写组调用，与文件句柄fh对应的进程组内的进程都从各自进程指定的偏移位置offset开始，将buf中count个类型为datatype的数据，写入到文件中。它和MPI_FILE_READ_AT_ALL_BEGIN的用法完全一样，只不过在意义上，一个执行写，一个执行读罢了。这一语句的完成要通过进程组内各进程都执行MPI_FILE_WRITE_AT_ALL_END来实现。

MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)

INOUT	fh	**写入文件的文件句柄**
IN	buf	**写入数据存放的缓冲区**
OUT	status	**该调用完成后返回的状态信息**

int MPI_File_write_at_all_end(MPI_File fh, void * buf, MPI_Status *status)

MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)

<type> BUF(*)

INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI**调用接口** 174 MPI_FILE_WRITE_AT_ALL_END

MPI_FILE_WRITE_AT_ALL_END**完成前面**MPI_FILE_WRITE_AT_ALL_BEGIN**开始的非阻塞写操作，该调用结束后，可以保证**buf**中的数据已经全部写到文件的指定位置。**

21.4 多视口的并行文件并行读写

对于前面介绍的文件读写方法，都不涉及文件指针，文件读写的位置都是作为参数明确给出的，而这一部分介绍的文件读取都是从一个特定的文件视口中，从文件指针的当前位置，对文件进行读写操作。

不同进程对应的文件指针可以是互不相同的，它们可以分别指向同一文件的不同位置。而视口是相对于某一进程来说的，它是特定进程所能看到的文件，某一进程的文件视口可以是整个文件，但多数情况下，文件视口只是整个文件的一个或几个部分。文件视口在整个文件中对应的部分可以是不连续的，但各个进程看到的其文件视口中的数据却是连续的。

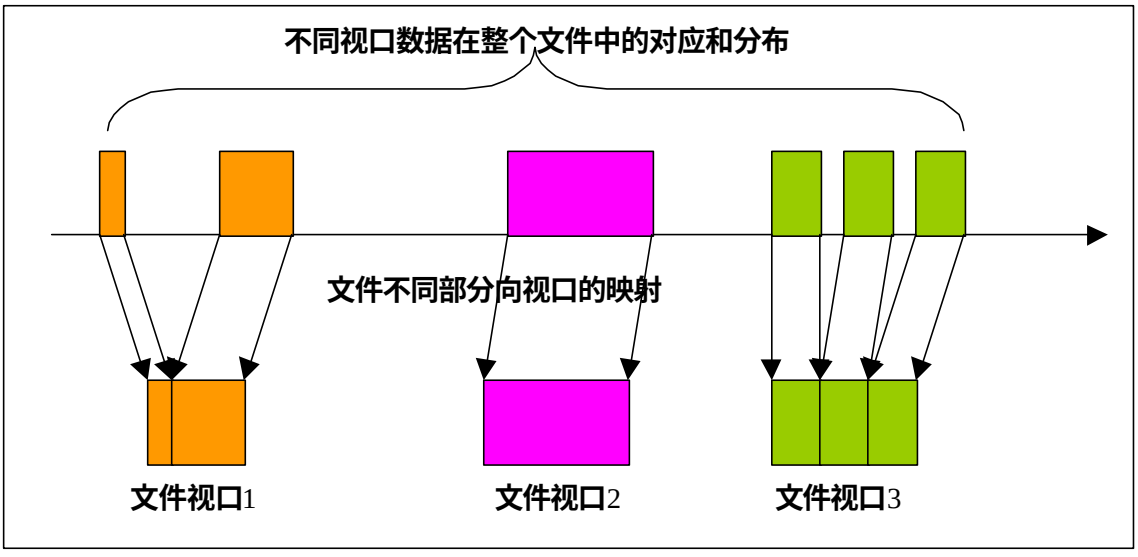


图 92 文件与视口的关系图示

21.4.1 文件视口与指针

文件视口可以用一个三元组来表达：

<起始偏移，基本类型，文件类型>

其中偏移是指该视口在文件中的起始位置，该位置度量是以字节为单位的；而基本类型是视口数据存取的基本单位，基本类型可以是MPI的预定义数据类型或派生数据类型；文件类型或者就是基本类型，或者是从基本类型派生出来的其它类型，文件类型真正限定了文件中哪些数据可以被视口访问，哪些数据对视口是不可见的。文件视口就是文件中从特定的偏移开始，连续多个直至文件结束的特定文件类型组成的。

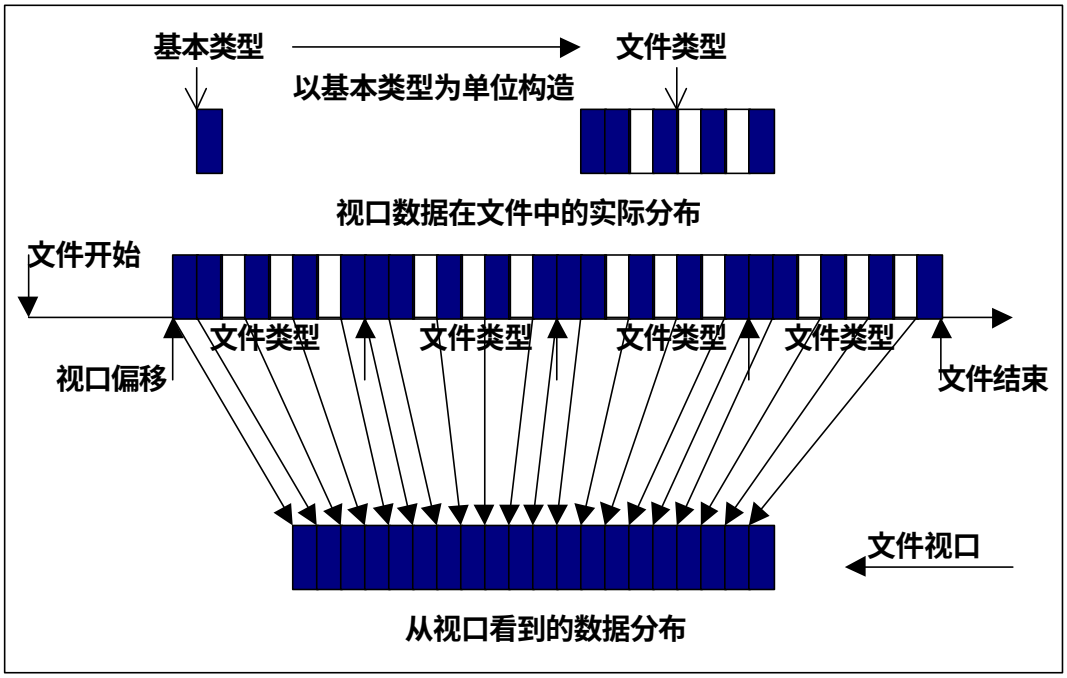


图 93 视口与基本类型、文件类型和文件的关系图示

MPI_FILE_SET_VIEW设置文件视口，它是一个组IO调用，所有与fh相联系的进程组中的进程都执行这一调用。调用进程在fh对应的文件中设置本进程的文件视口，该视口相对于文件头的偏移是disp，即视口首先从文件中跳过disp个字节，然后给出了视口数据的基本数据单位etype，以后所有对该视口的访问必须以etype为单位来进行，而filetype则在etype的基础上，通过以etype为单位定义数据类型filetype，将视口不需要的数据排斥在外，即一般filetype定义的数据类型是有“空穴”的，这些空穴是视口无需访问的数据。

由上不难看出，文件视口其实就是一种特殊的数据类型，它指定的位置不象前面定义派生数据类型那样是在内存中，而是在文件中。它的另一个约束是都必须以基本的数据单位etype为基础来进行定义，而不是可以任意使用不同的数据类型来定义；视口包含数据的多少其实是通过定义内容不连续的数据类型filetype来实现的，该类型中不连续的部分是视口不需要访问的部分，手段指定本视口包括哪些数据；从偏移disp开始，连续重复N次知道文件结束，由数据类型filetype得到的新的派生数据类型，才是文件视口对应数据类型。

以后当进程对它们各自的文件视口进行访问时，可以认为该文件中只包含视口对应的数据，而且数据之间是没有空隙的。

不同的进程，通过在相同的文件上定义互不交叉的文件视口，就可以实现对文件的并行

访问。

```
MPI_FILE_SET_VIEW(fh, disp,etype,filetype,datarep,info)
INOUT   fh          视口对应文件的文件句柄
IN      disp        视口在文件中的偏移位置
IN      etype       视口基本数据类型
IN      filetype    视口文件类型
IN      datarep     视口数据的表示方法
IN      info        传递给运行时的信息
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
    MPI_Datatype filetype, char * datarep, MPI_Info info)
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO,
    IERROR)
    CHARACTER *(*) DATAREP
    INTEGER FH, ETYPE, FILETYPE,INFO,IERROR
```

MPI**调用接口** 175 MPI_FILE_SET_VIEW

MPI_FILE_SET_VIEW**调用完成后，原来的文件句柄**fh**就不再代表该文件，而是代表本调用产生的文件视口，以后使用**fh**对文件的所有操作都是对其视口的操作。**

其中视口数据的表示方法共有三种：native、internal**和**external32**，定义数据表示是为了高效解决**MPI**的一致性问题，因为不同类型的计算机，其数据的表示方法是不同的。**

其中native**数据表示的含义是数据在文件中的存储方式和在内存中的完全一样，这样在进行文件存取时，就没有数据转换的开销，对文件访问的效率和精度没有损失。显然这种方式在由不同类型的计算机组成的异构环境是行不通的，使用**native**数据表示虽然效率高，但存在移植性的问题。**

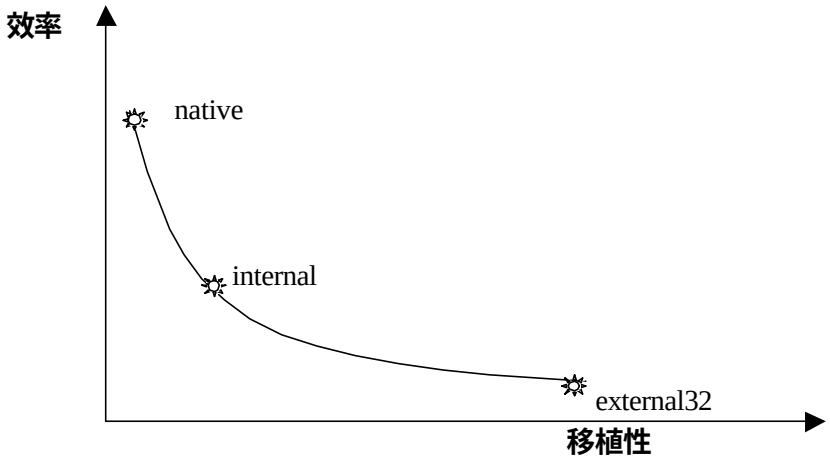


图 94 不同的数据表示和效率与移植性的关系

internal**数据表示是由具体的实现来定义的，比如相同的**MPI**实现可以在不同类型的机器上实现数据转换，它是为了解决**native**数据表示的不可移植问题，通过在某一个具体的实现上提供特定的手段，来实现一定程度的移植性，它解决可移植问题并不彻底。**

external32**数据表示是为了彻底解决任何不同类型机器之间的数据移植问题，用**external32

数据表示产生的文件，可以被任何MPI**实现，在任何计算机上进行访问。但是**external32**数据表示的一个缺点就是效率的问题，对每次不同的文件的读写，都必须进行一次同一格式的数据转换。**

MPI_FILE_GET_VIEW(fh, disp,etype,filetype,datarep)
IN fh **视口文件句柄**
OUT disp **返回的视口在文件中的起始偏移**
OUT etype **视口数据单元类型**
OUT filetype **视口文件类型**
OUT datarep **视口的数据表示**
int MPI_File_get_view(MPI_File fh, MPI_Offset * disp, MPI_Datatype * etype,
MPI_Datatype * filetype, char * datarep)
MPI_FILE_GET_VIEW(FH, DISP,ETYPE,FILETYPE,DATAREP,IERROR)
INTEGER FH, ETYPE,FILETYPE,IERROR
CHARACTER *(*) DATAREP,
INTEGER (KIND=MPI_OFFSET_KIND) DISP

MPI**调用接口** 176 MPI_FILE_GET_VIEW

MPI_FILE_GET_VIEW**是一个查询调用，它返回文件视口的各种参数，**fh**是给定的文件视口句柄，**disp**是该视口在文件中的起始偏移位置，**etype**是文件视口的基本数据单位类型，**filetype**是文件视口的文件类型，**datarep**是文件视口的数据表示方法。以上各个参数对应于**MPI_FILE_SET_VIEW**调用时所给出的各种参数。**

MPI_FILE_SEEK(fh, offset, whence)
INOUT fh **文件句柄**
IN offset **相对偏移位置**
IN whence **指出**offset**的参照位置**
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
MPI_FILE_SEEK(FH,OFFSET,WHENCE,IERROR)
INTEGER FH, WHENCE, IERROR
INTEGER (KIND=MPI_OFFSET_KIND) OFFSET

MPI**调用接口** 177 MPI_FILE_SEEK

MPI_FILE_SEEK**将文件的指针移动到给定的位置。其中**fh**是文件句柄，**offset**是相对于**whence**的偏移位置，它的数据可正可负。其中**whence**的取值可以为**MPI_SEEK_SET、MPI_SEEK_CUR**和**MPI_SEEK_END**。不同参数取值的含义如表格** 22**所示。**

表格 22 **不同文件位置参照点的含义**

参照位置取值	**调用后文件指针的位置**
MPI_SEEK_SET	offset
MPI_SEEK_CUR	**当前指针位置**+offset
MPI_SEEK_END	**文件结束位置**+offset

MPI_FILE_GET_POSITION(fh, offset)
IN fh **文件句柄**
OUT offset **偏移位置**
int MPI_File_get_position(MPI_File fh, MPI_Offset * offset)
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
INTEGER FH, IERROR
INTEGER (KIND=MPI_OFFSET_KIND) OFFSET

MPI**调用接口** 178 MPI_FILE_GET_POSITION

MPI_FILE_GET_POSITION**返回当前文件视口指针相对视口起始位置的偏移，该偏移是以视口的基本数据类型**etype**为单位来度量的。注意这里是相对于视口的偏移，因此不考虑它们对应的数据在实际文件中的空隙所占用的空间。**

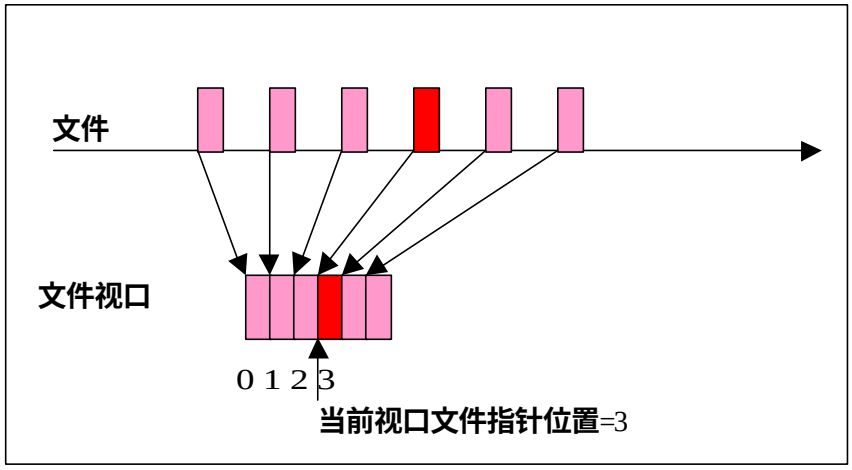


图 95 当前文件视口位置图示

MPI_FILE_GET_BYTE_OFFSET(fh, offset,disp)
IN fh **视口文件句柄**
IN offset **在视口中的相对偏移**
OUT disp **以字节为单位，在文件中的绝对偏移位置**
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset, MPI_Offset * disp)
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
INTEGER FH, IERROR
INTEGER (KIND=MPI_OFFSET_KIND) OFFSET, DISP

MPI**调用接口** 179 MPI_FILE_GET_BYTE_OFFSET

MPI_FILE_GET_BYTE_OFFSET**返回相对于文件视口的偏移位置**offset**在文件中对应的**

绝对位置，该绝对位置的度量是以字节为单位的。

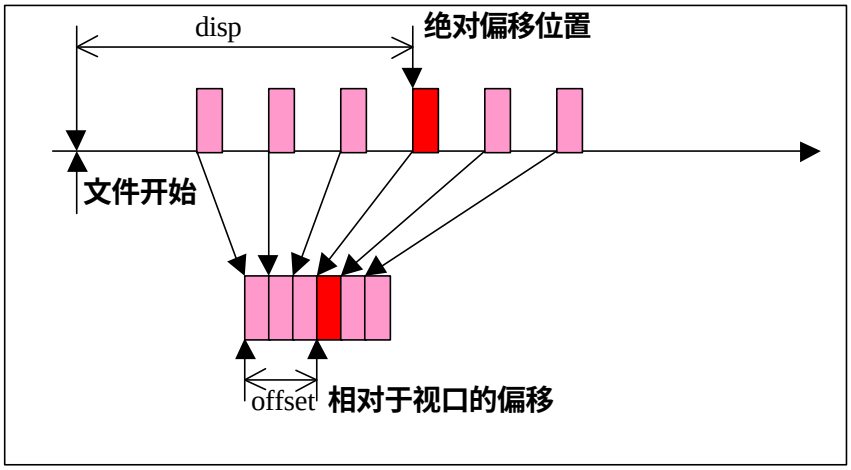


图 96 相对于视口的偏移和相对于文件的绝对位置关系图示

21.4.2 阻塞方式的视口读写

MPI_FILE_READ(fh, buf,count,datatype,status)
INOUT fh 文件视口句柄
OUT buf 读出数据存放的缓冲区
IN count 读出数据个数
IN datatype 读出数据的数据类型
OUT status 操作完成后返回的状态
int MPI_File_read(MPI_file fh, void * buf, int count, MPI_Datatype datatype,
MPI_Status * status)
MPI_FILE_READ(FH, BUF, COUNT,DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH,COUNT,DATATYPE,STATUS,IERROR

MPI调用接口 180 MPI_FILE_READ

MPI_FILE_READ从视口文件fh中读取数据，读取数据的类型是datatype，读取数据的个数是count，读取的数据放到buf缓冲区中，该操作完成后返回的状态放到status中。这里没有指明读取的位置，该位置是隐含指定的，它就是当前视口文件句柄指针的位置。该读取操作完成后，视口文件句柄指针自动指向下一个视口内的基本数据类型的位置。

MPI_FILE_WRITE(fh, buf,count,datatype,status)
INOUT fh **视口文件句柄**
IN buf **将要写入文件的数据存放的缓冲区**
IN count **写入数据的个数**
IN datatype **写入数据的数据类型**
OUT status **该调用返回的状态参数**
int MPI_File_write(MPI_file fh, void * buf, int count, MPI_Datatype datatype,
MPI_Status * status)
MPI_FILE_WRITE(FH, BUF, COUNT,DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH,COUNT,DATATYPE,STATUS,IERROR

MPI**调用接口** 181 MPI_FILE_WRITE

MPI_FILE_WRITE**将**buf**中的**count**个数据类型为**datatype**的数据写入到视口文件中，返回状态是**status。MPI_FILE_WRITE**和**MPI_FILE_READ**是对应的，写入位置也是由当前视口文件句柄的指针隐含指定的，写入完成后，当前视口文件句柄指针指向视口中下一个基本数据单元的位置。**

MPI_FILE_READ_ALL(fh, buf,count,datatype,status)
INOUT fh **视口文件句柄**
OUT buf **读出数据存放的缓冲区**
IN count **读出数据的个数**
IN datatype **读出数据的数据类型**
OUT status **该调用返回的状态信息**
int MPI_File_read_all(MPI_file fh, void * buf, int count, MPI_Datatype datatype,
MPI_Status * status)
MPI_FILE_READ_ALL(FH, BUF, COUNT,DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH,COUNT,DATATYPE,STATUS,IERROR

MPI**调用接口** 182 MPI_FILE_READ_ALL

MPI_FILE_READ_ALL**是一个组调用，与句柄**fh**相联系的进程组中的所有进程都要执行此调用，它如同进程组内的所有进程都执行了一个**MPI_FILE_READ**调用一样，从视口文件中读取**count**个数据类型为**datatype**的数据，放到缓冲区**buf**中，**status**是返回的状态参数。**

注意这里不同进程所使用的视口文件指针是不同的，调用结束后各个视口文件的指针都自动指向本视口下一个基本数据单元的位置。

MPI_FILE_WRITE_ALL(fh, buf,count,datatype,status)
INOUT fh **视口文件句柄**
IN buf **写入数据存放的缓冲区**
IN count **写入数据的个数**
IN datatype **写入数据的数据类型**
OUT status **写入调用返回的状态信息**
int MPI_File_write_all(MPI_file fh, void * buf, int count, MPI_Datatype datatype,
MPI_Status * status)
MPI_FILE_WRITE_ALL(FH, BUF, COUNT,DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH,COUNT,DATATYPE,STATUS,IERROR

MPI**调用接口** 183 MPI_FILE_WRITE_ALL

MPI_FILE_WRITE_ALL**也是组调用，和**MPI_FILE_READ_ALL**相对应，一个是读，一个是写。与句柄**fh**相联系的进程组中的所有进程都要执行此调用，它如同进程组内的所有进程都执行了一个**MPI_FILE_WRITE**调用一样，将数据缓冲区**buf**中的**count**个数据类型为**datatype**的数据数据，写入到视口文件中，**status**是返回的状态参数。**

各进程写入的位置可以是互不相同的，因为不同进程所使用的视口文件指针是独立的，调用结束后各个视口文件的指针都自动指向本视口下一个基本数据单元的位置。

21.4.3 非阻塞方式的视口读写

前面介绍的是阻塞方式的视口文件读写，视口文件的读写还有非阻塞的形式。即首先执行视口读写的请求操作，然后立即返回，返回后并不意味着读写操作的完成，读写的真正完成还需要对非阻塞读写完成对象执行调用MPI_WAIT**，这和非阻塞通信操作是类似的。**

MPI_FILE_IREAD(fh, buf, count,datatype,request)
INOUT fh **视口文件句柄**
OUT buf **读取数据存放的缓冲区**
IN count **读取数据个数**
IN datatype **读取数据的类型**
OUT request **返回的非阻塞读取完成对象**
int MPI_File_iread(MPI_File fh, void * buf, int count,Datatype datatype,
MPI_Request * request)
MPI_FILE_IREAD(FH,BUF,COUNT,DATATYPE,REQUEST,IERROR)
<type> BUF (*)
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI**调用接口** 184 MPI_FILE_IREAD

从句柄fh**对应的视口文件中读取**count**个数据类型为**datatype**的数据放到**buf**中，这是一个非阻塞读取调用，该调用不必等到读取操作完成便可以立即返回，同时返回一个非阻塞读取完成对象**request**，读取操作的完成可以通过对**request**对象执行**MPI_WAIT**调用，该调用结束后，可以保证读取操作已经完成。**

```
MPI_FILE_IWRITE(fh, buf, count,datatype,request)
INOUT     fh          视口文件句柄
IN        buf         写入数据存放的缓冲区
IN        count       写入数据的个数
IN        datatype    写入数据的数据类型
OUT       request     返回的非阻塞写入完成对象
int  MPI_File_iwrite(MPI_File fh, void * buf, int count,Datatype datatype,
          MPI_Request  *  request)
MPI_FILE_IWRITE(FH,BUF,COUNT,DATATYPE,REQUEST,IERROR)
     <type>  BUF  (*)
     INTEGER  FH,  COUNT,  DATATYPE,  REQUEST,  IERROR
```

MPI调用接口 185　MPI_FILE_IWRITE

MPI_FILE_IWRITE**向句柄**fh**对应的视口文件写入数据，数据存放在**buf**中，共有**count**个，数据类型为**datatype**，这是一个非阻塞读取调用，该调用不必等到写入操作完成便可以立即返回，同时返回一个非阻塞写入完成对象**request**，写入操作的完成可以通过对**request**对象执行**MPI_WAIT**调用，该调用结束后，可以保证写入操作已经完成。**

21.4.4 两步非阻塞视口组调用方式

MPI-2**对组调用的非阻塞视口文件读写采取了特殊的形式，将非阻塞视口文件的读写明确分为两步，一是非阻塞视口读写组调用的开始，二是非阻塞视口读写组调用的结束。其实第二步和**MPI_WAIT**的功能是一致的。**

```
MPI_FILE_READ_ALL_BEGIN(fh, buf,count,datatype)
INOUT     fh          视口文件句柄
OUT       buf         读取数据存放的缓冲区
IN        count       读取数据的个数
IN        datatype    读取数据的数据类型
int MPI_File_read_all_begin(MPI_File fh, void * buf, int count,
          MPI_Datatype  datatype)
MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
          <type>    BUF(*)
          INTEGER    FH, COUNT, DATATYPE, IERROR
```

MPI调用接口 186　MPI_FILE_READ_ALL_BEGIN

MPI_FILE_READ_ALL_BEGIN**开始一个视口文件的非阻塞组调用，文件句柄**fh**对应的**

进程组内的所有进程都需要执行此调用，但是各进程分别拥有自己独立的视口文件指针，各进程分别从自己的视口文件指针所在的当前位置开始读取，读取数据的个数是count个，读取数据的数据类型是datatype。该读取调用执行后立即返回，不必等到真正读取操作的完成，数据从视口文件中读出并且放到buf中是MPI_FILE_READ_ALL_END调用后的结果。

```
MPI_FILE_READ_ALL_END(fh, buf,status)
INOUT    fh        视口文件句柄
OUT      buf       读取数据存放的缓冲区
OUT      status    返回的状态信息
int MPI_File_read_all_end(MPI_File fh, void * buf, MPI_Status * status)
MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
        <type>  BUF(*)
        INTEGER  FH, STATUS(MPI_STATUS_SIZE) , IERROR
```

MPI调用接口 187 MPI_FILE_READ_ALL_END

MPI_FILE_READ_ALL_END也是一个组调用，它和MPI_FILE_READ_ALL_BEGIN结合起来，实现非阻塞视口文件的组读取。当MPI_FILE_READ_ALL_END调用完成后，前面启动的非阻塞组文件读取操作才真正完成，数据缓冲区中的数据才可以被各个进程访问。

```
MPI_FILE_WRITE_ALL_BEGIN(fh, buf,count,datatype)
INOUT    fh          视口文件句柄
IN       buf         读取数据存放的缓冲区
IN       count       读取数据的个数
IN       datatype    读取数据的数据类型
int MPI_File_write_all_begin(MPI_File fh, void * buf, int count,
         MPI_Datatype  datatype)
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
        <type>  BUF(*)
        INTEGER  FH, COUNT, DATATYPE, IERROR
```

MPI调用接口 188 MPI_FILE_WRITE_ALL_BEGIN

MPI_FILE_WRITE_ALL_BEGIN和MPI_FILE_READ_ALL_BEGIN类似，它启动一个非阻塞的组写入操作，视口文件句柄fh对应的进程组中的所有进程都需要执行该调用，它将数据缓冲区buf中的count个数据类型为datatype的数据，写入到句柄fh对应的视口文件中。该调用不必等到写入操作的完成就可以立即返回，写操作的完成是通过调用另一个组调用语句MPI_FILE_WRITE_ALL_END实现的。

```
MPI_FILE_WRITE_ALL_END(fh, buf,status)
INOUT     fh          视口文件句柄
IN        buf         写入数据存放的缓冲区
OUT       status      返回的状态信息
int MPI_File_write_all_end(MPI_File fh, void * buf, MPI_Status * status)
MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
        <type>   BUF(*)
        INTEGER   FH, STATUS(MPI_STATUS_SIZE) , IERROR
```

MPI调用接口 189 MPI_FILE_WRITE_ALL_END

MPI_FILE_WRITE_ALL_END**也是一个组调用，它和**MPI_FILE_WRITE_ALL_BEGIN**结合起来，实现非阻塞视口文件的组写入。当**MPI_FILE_WRITE_ALL_END**调用完成后，前面启动的非阻塞组文件写入操作才真正完成，数据缓冲区才可以被释放或更新。**

21.5 共享文件读写

对于独立视口文件的读写，每一个进程都拥有一个独立的视口文件指针，各个进程对自己视口文件的读写操作只会改变本视口文件指针的位置，对其它视口的文件指针没有任何影响，就如同是在对两个互不相干的文件进行操作。

但是对于共享文件的读写，该共享文件只有一个文件指针，所有的进程都共享它，即任何一个进程对该指针的修改都会对其它的进程产生影响。

```
MPI_FILE_SEEK_SHARED(fh, offset, whence)
INOUT     fh          共享文件句柄
IN        offset      偏移的相对位置
IN        whence      偏移相对的绝对位置
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
    INTEGER    FH, WHENCE, IERROR
    INTEGER  (KIND=MPI_OFFSET_KIND)   OFFSET
```

MPI调用接口 190 MPI_FILE_SEEK_SHARED

MPI_FILE_SEEK_SHARED **的含义同**MPI_FILE_SEEK**类似，只不过前者移动的是共享文件指针，该调用执行完后所有的进程都会看到指针位置的变化，而**MPI_FILE_SEEK**只移动当前进程的指针。**

MPI_FILE_GET_POSITION_SHARED**同**MPI_FILE_GET_POSITION**类似，它返回共享文件指针相对起始位置的偏移。**

```
MPI_FILE_GET_POSITION_SHARED(fh, offset)
IN          fh
OUT         offset
int  MPI_File_get_position(MPI_File fh, MPI_Offset * offset)
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
      INTEGER FH, IERROR
      INTEGER (KIND=MPI_OFFSET_KIND) OFFSET
```

MPI**调用接口** 191 MPI_FILE_GET_POSITION_SHARED

21.5.1 阻塞共享文件读写

```
MPI_FILE_READ_SHARED(fh, buf,count,datatype,status)
INOUT     fh          共享文件句柄
OUT       buf         读取数据存放的缓冲区
IN        count       读取数据的个数
IN        datatype    读取数据的数据类型
OUT       status      返回的状态信息
int MPI_File_read_shared(MPI_File fh, void * buf, int count, MPI_Datatype datatype,
        MPI_Status   *  status)
MPI_FILE_READ_SHARED(FH, BUF, COUNT,DATATYPE, STATUS,IERROR)
     <type> BUF(*)
     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI**调用接口** 192 MPI_FILE_READ_SHARED

MPI_FILE_READ_SHARED**从句柄**fh**对应的共享文件中读取**count**个数据类型为**datatype**的数据，放到**buf**中，返回的状态是**status**。这一调用使用的文件指针是共享指针，即该读取操作完成后，共享指针自动移到下一个数据单元的位置，其它进程再对该共享文件进行操作时，文件指针的位置是本调用完成后指针移动过的位置。**

MPI_FILE_WRITE_SHARED**向句柄**fh**对应的共享文件中写入数据，数据存放在**buf**中，共**count**个，数据类型为**datatype**。返回的状态信息是**status。MPI_FILE_WRITE_SHARED**调用和**MPI_FILE_READ_SHARED**一样，写入位置是共享文件指针对应的位置，该写入操作完成后，共享文件指针自动指向下一个数据单元的位置，指针的移动在所有的进程中都会体现出来，本写入操作完成后，其它进程对共享文件的操作是在本操作完成后指针所在的位置开始的。**

```
MPI_FILE_WRITE_SHARED(fh, buf,count,datatype,status)
INOUT     fh          共享文件句柄
IN        buf         写入数据存放的缓冲区
IN        count       写入数据的个数
IN        datatype    写入数据的数据类型
OUT       status      返回的状态信息
int MPI_File_write_shared(MPI_File fh, void * buf, int count, MPI_Datatype datatype,
        MPI_Status   *  status)
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT,DATATYPE, STATUS,IERROR)
     <type> BUF(*)
     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI调用接口 193 MPI_FILE_WRITE_SHARED

```
MPI_FILE_READ_ORDERED(fh, buf, count, datatype, status)
INOUT     fh          共享文件的句柄
OUT       buf         读取数据存放的缓冲区
IN        count       读取数据的个数
IN        datatype    读取数据的数据类型
OUT       status      该调用返回的状态信息
int MPI_File_read_ordered(MPI_File fh, void * buf, int count, MPI_Datatype datatype,
        MPI_Status  * status)
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
      <type>  BUF(*)
     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI调用接口 194 MPI_FILE_READ_ORDERED

MPI_FILE_READ_ORDERED调用是组调用，就如同每一个进程组内的进程都执行了一个MPI_FILE_READ_SHARED调用。各个进程对共享文件的读取是按续进行的，根据各个进程的rank标识，从小到大，0，1，...，N-1 号进程依次对文件进行读取。上一个进程读取完成后，文件指针自动指向下一个数据单元的位置，每个进程都是从共同的文件视口中读取count个datatype数据类型的数据，存放到各自的buf中。其中status是返回的状态信息。

MPI_FILE_WRITE_ORDERED调用是组调用，就如同每一个进程组内的进程都执行了一个MPI_FILE_WRITE_SHARED调用。各个进程对共享文件的写入是按续进行的，根据各个进程的rank标识，从小到大，0，1，...，N-1 号进程依次对文件进行写入。上一个进程写入完成后，文件指针自动指向下一个数据单元的位置，每个进程都是向共同的文件视口中写入存放在各自buf中的count个datatype数据类型的数据。其中status是返回的状态信息。

MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)

INOUT	fh	**视口文件句柄**
IN	buf	**写入数据存放的缓冲区**
IN	count	**写入数据个数**
IN	datatype	**写入数据的数据类型**
OUT	status	**返回的状态信息**

int MPI_File_write_ordered(MPI_File fh, void * buf, int count, MPI_Datatype datatype,
MPI_Status * status)

MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI**调用接口** 195 MPI_FILE_WRITE_ORDERED

21.5.2 非阻塞共享文件读写

MPI_FILE_IREAD_SHARED**与**MPI_FILE_READ_SHARED**一样，也是从句柄**fh**对应的共享文件中读取**count**个数据类型为**datatype**的数据，存放到**buf**中，但是这一调用和**MPI_FILE_READ_SHARED**不同之处就在于它是非阻塞调用，它不必等到文件读取完成就可以立即返回，而文件读取的最终完成是通过使用本调用返回的非阻塞读取完成对象**request**执行**MPI_WAIT**实现的。**

MPI_FILE_IREAD_SHARED(fh, buf,count,datatype,request)

INOUT	fh	**共享文件句柄**
OUT	buf	**读取数据存放的缓冲区**
IN	count	**读取数据的个数**
IN	datatype	**读取数据的数据类型**
OUT	request	**非阻塞读取完成对象**

int MPI_File_iread_shared(MPI_File fh, void * buf, int count, MPI_Datatype datatype,
MPI_Request * request)

MPI_FILE_IREAD_SHARED(FH, BUF, COUNT,DATATYPE, REQUEST,IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI**调用接口** 196 MPI_FILE_IREAD_SHARED

MPI_FILE_IWRITE_SHARED**向句柄**fh**对应的共享文件中写入存放在**buf**中的数据，写入数据的个数是**count**个，写入数据的数据类型是**datatype**。该调用执行后立即返回，不必等到写入操作完成，返回的非阻塞写入完成对象是**request**，写入操作的最终完成是通过对**request**调用**MPI_WAIT**实现的。**

MPI_FILE_IWRITE_SHARED(fh, buf,count,datatype,request)

INOUT	fh	**共享文件句柄**
IN	buf	**写入数据存放的缓冲区**
IN	count	**写入数据个数**
IN	datatype	**写入数据类型**
OUT	request	**非阻塞写入完成对象**

int MPI_File_iwrite_shared(MPI_File fh, void * buf, int count, MPI_Datatype datatype,
MPI_Request * request)

MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT,DATATYPE, REQUEST,IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI**调用接口** 197 MPI_FILE_IWRITE_SHARED

21.5.3 两步非阻塞共享文件组读写

共享文件的非阻塞组读写，也是分成两步来进行的，第一步是启动非阻塞的组读写，然后立即返回；第二步是完成非阻塞的组读写。

MPI_FILE_READ_ORDERED_BEGIN(fh, buf, count, datatype)

INOUT	fh	**文件句柄**
OUT	buf	**读取数据存放的缓冲区**
IN	count	**读取数据的个数**
IN	datatype	**读取数据的数据类型**

int MPI_File_read_ordered_begin(MPI_File fh, void * buf, int count,
MPI_Datatype datatype, MPI_Status * status)
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, IERROR

MPI**调用接口** 198 MPI_FILE_READ_ORDERED_BEGIN

MPI_FILE_READ_ORDERED_BEGIN**开始一个非阻塞的共享文件组读写，它的作用是让与**fh**相联系的进程组内的每一个进程按照其进程编号**rank**的大小，从小到大，依次从共享文件中读取**count**个数据类型为**datatype**的数据，存放在各自的缓冲区**buf**中，但是该调用不必读取操作完成就可以立即返回，而该读取操作的完成是通过调用**MPI_FILE_READ_ORDERED_END**实现的。**

MPI_FILE_READ_ORDERED_END**完成前面启动的非阻塞共享文件组读取调用，其中**fh**是文件句柄，**buf**是读取数据存放的缓冲区，**status**是返回的状态信息。当这一调用结束后，各进程才可以使用从文件中读取的缓冲区**buf**中的数据。**

MPI_FILE_READ_ORDERED_END(fh, buf, status)
INOUT fh **文件句柄**
OUT buf **读取数据存放的缓冲区**
OUT status **返回的状态信息**
int MPI_File_read_ordered_end(MPI_File fh, void * buf, MPI_Status * status)
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, STATUS, IERROR

MPI**调用接口** 199 MPI_FILE_READ_ORDERED_END

MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
INOUT fh **共享文件句柄**
IN buf **写入数据存放的缓冲区**
IN count **写入数据的个数**
IN datatype **写入数据的数据类型**
int MPI_File_write_ordered_begin(MPI_File fh, void * buf, int count,
MPI_Datatype datatype)
MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
<type> BUF(*)
INTEGER FH, COUNT,DATATYPE, IERROR

MPI**调用接口** 200 MPI_FILE_WRITE_ORDERED_BEGIN

MPI_FILE_WRITE_ORDERED_BEGIN**开始一个非阻塞的共享文件组写入，它的作用是让与**fh**相联系的进程组内的每一个进程按照其进程编号**rank**的大小，从小到大，依次将各自缓冲区**buf**中**count**个数据类型为**datatype**的数据，写入到共享文件中，但是该调用不必写入操作完成就可以立即返回，而该写入操作的完成是通过调用** MPI_FILE_WRITE_ORDERED_END**实现的。**

MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
INOUT fh **文件句柄**
IN buf **写入数据存放的缓冲区**
OUT status **返回的状态信息**
int MPI_File_write_ordered_end(MPI_File fh, void * buf, MPI_Status * status)
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI**调用接口** 201 MPI_FILE_WRITE_ORDERED_END

MPI_FILE_WRITE_ORDERED_END**完成前面启动的非阻塞共享文件组写入调用，其中**fh**是文件句柄，**buf**是写入数据存放的缓冲区，**status**是返回的状态信息。当这一调用结束后，各进程才可以释放缓冲区**buf**中的数据。**

```
MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)
IN        fh          文件句柄
IN        datatype    数据类型
OUT       extent      返回的类型跨度
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype, MPI_Aint * extent)
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE,EXTENT, IERROR)
    INTEGER    FH, DATATYPE, IERROR
    INTEGER  (KIND=MPI_ADDRESS_KIND)    EXTENT
```

MPI**调用接口** 202　MPI_FILE_GET_TYPE_EXTENT

MPI_FILE_GET_TYPE_EXTENT**返回文件句柄**fh**对应的文件中的数据类型**datatype**的跨度大小**extent**。若当前文件视口使用的是用户定义的数据表示，则应使用**dtype_file_extent_fn**来计算数据类型的跨度。**

```
MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn,
        dtype_file_extent_fn,extra_state)
IN        datarep                数据表示
IN        read_conversion_fn     读取时使用的数据转换函数
IN        write_conversion_fn    写入时使用的数据转换函数
IN        dtype_file_extent_fn   查询文件数据类型的跨度时使用的函数
IN        extra_state
int MPI_Register_datarep(char * datarep, MPI_Datarep_conversion_function *
read_conversion_fn, MPI_Datarep_conversion_function * write_conversion_fn,
MPI_Datarep_extent_function * dtype_file_extent_fn, void * extra_state)
MPI_REGISTER_DATAREP(DATAREP,READ_CONVERSION_FN,
 WRITE_CONVERSION_FN,DTYPE_FILE_EXTENT_FN,EXTRA_STATE,IERROR)
  EXTERNAL  READ_CONVERSION_FN, WRITE_CONVERSION_FN
                DTYPE_FILE_EXTENT_FN
   INTEGER    (KIND=MPI_ADDRESS_KIND)  EXTRA_STATE
   INTEGER    IERROR
```

MPI**调用接口** 203　MPI_REGISTER_DATAREP

MPI_REGISTER_DATAREP**注册了一个新的数据表示**datarep，**以后就可以作为参数传递给**MPI_FILE_SET_VIEW，**对新的数据类型**datarep，**在以后进行读取时，使用的读取函数为**read_conversion_fn，**它将文件中的数据表示转换为本地内存中的表示；在进行写入时，使用的写入函数是**write_conversion_fn，**它将本地内存中的数据转换为文件中的数据表示；而计算文件数据类型的跨度时使用的计算函数是**dtype_file_extent_fn。

```
MPI_FILE_SET_ATOMICITY(fh, flag)
INOUT          fh          文件句柄
IN             flag        标志信息
int MPI_File_set_atomicity(MPI_File fh, int flag)
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
   INTEGER  FH,  IERROR
   LOGICAL  FLAG
```

MPI调用接口 204　MPI_FILE_SET_ATOMICITY

MPI_FILE_SET_ATOMICITY是一个组调用，它设置同样是通过组调用方式打开的文件句柄fh的访问模式。当flag=ture时，它设置对文件的访问为原子访问模式，若flag=false，则取消对文件的原子访问模式。通过设置对文件的访问模式，可以满足对文件存取的一致性语义要求。

```
MPI_FILE_GET_ATOMICITY(fh, flag)
IN             fh       文件句柄
OUT            flag     返回的标志信息
int MPI_File_set_atomicity(MPI_File fh, int  * flag)
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
   INTEGER  FH,  IERROR
   LOGICAL  FLAG
```

MPI调用接口 205　MPI_FILE_GET_ATOMICITY

MPI_FILE_GET_ATOMICITY得到文件句柄fh对应文件的访问模式信息，结果放在flag中。这一结果就是调用MPI_FILE_SET_ATOMICITY设置的结果，当flag=true时，表示文件已被设置为原子访问模式，若flag=false，则文件被取消原子访问模式。

```
MPI_FILE_SYNC(fh)
INOUT          fh       文件句柄
int MPI_File_sync(MPI_File fh)
MPI_FILE_SYNC(FH, IERROR)
   INTEGER  FH,  IERROR
```

MPI调用接口 206　MPI_FILE_SYNC

MPI_FILE_SYNC是一个组调用，与句柄fh相联系的进程组中的所有进程都必须执行它。其作用就是将前面所有对fh写操作需要写入的数据都写到文件存放的存储设备上。对于非阻塞写入，编程者需要保证此调用在非阻塞调用已经完成后才执行。

21.6 分布式数组文件的存取

在许多情况下，MPI要处理的数组是分布在各个处理器上的，并且其分布方式也是知道的，可以用文件来存储这样的数组数据，当需要把这些分布式数组写入文件或从文件中读入时，可以通过使用分布式数组或分布式数组片段构造符，来定义分布式数组或分布式数组片段，构造分布式数组文件类型，从而实现读分布式数组文件的高效、简单的操作。

首先介绍数组的三种基本的分布方式：块分布、循环分布和循环块分布。

设有数组A1(100)和含有4个处理器的处理器阵列P1(4)，则数组A1在P1上进行块分布的结果是

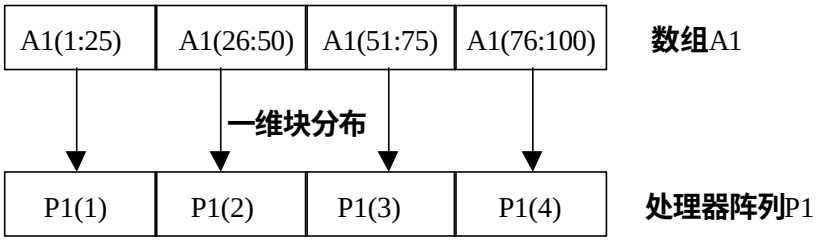


图 97 一维块分布

数组A1在P1上的循环分布为：

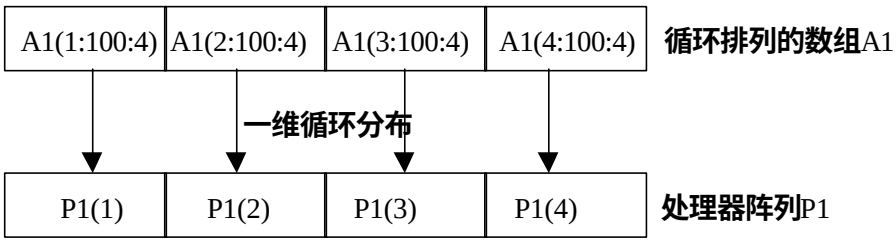


图 98 一维循环分布

数组A1在P1上的循环块分布为（设块的大小为2）：

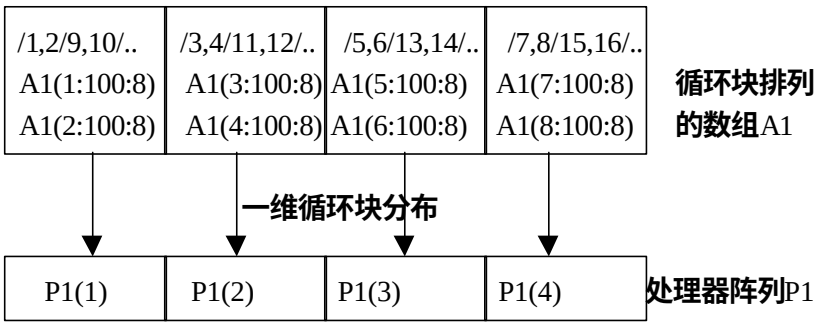


图 99 一维循环块分布

对于多维的情况，就如同在每一维上进行指定的一维分布。下面介绍分布式数组类型和分布式数组片段类型的定义。

MPI_TYPE_CREATE_DARRAY(size,rank,ndims,array_of_gsizes,array_of_distribs,
array_of_dargs,array_of_psizes,order,oldtype,newtype)

IN	size	存放分布式数组文件的处理器个数
IN	rank	当前进程标识
IN	ndims	分布式数组的维数
IN	array_of_gsize	全局数组每一维的大小
IN	array_of_distribs	数组每一维的分布方式
IN	array_of_dargs	数组每一维分布方式的参数
IN	array_of_psizes	数组每一维分布的处理器个数
IN	older	数组存放以C方式的行优先或FORTRAN方式的列优先
IN	oldtype	数组元素的类型
OUT	newtype	返回的分布式文件数组的数据类型

int MPI_Type_create_darray(int size, int rank, int ndims, int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype * newtype)

MPI_TYPE_CREATE_DARRAY(SIZE,RANK,NDIMS,ARRAY_OF_GSIZES,
ARRAY_OF_DISTRIBS,ARRAY_OF_DARGS, ARRAY_OF_PSIZE,ORDER,
ORDER, OLDTYPE, NEWTYPE, IERROR)

INTEGER SIZE,RANK,NDIMS, ARRAY_OF_GSIZE(*), ARRAY_OF_DISTRIBS(*),
ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE,

MPI调用接口 207 MPI_TYPE_CREATE_DARRAY

MPI_TYPE_CREATE_DARRAY是一个新的数据类型构造符，用来定义分布式数组类型。对于一个ndims维的全局数组，它每一维的大小由数组array_of_gsize定义，而每一维在处理器阵列上的分布方式和分布参数由数组array_of_distribs和array_of_dargs定义，只有对于循环块分布array_of_dargs才有意义，对于块分布和循环分布，使用却省的分布参数MPI_DISTRIBUTE_DFLT_DARG。总的处理器个数是size，处理器阵列每一维的处理器个数由数组array_of_psizes定义。处理器阵列总是行优先的，但是局部数组在内存的存储顺序是由older来定义的，它只有两种取值，要么是FORTRAN形式的列优先MPI_ORDER_FORTRAN，要么是C形式的行优先MPI_ORDER_C。oldtype是数组元素的类型，而返回的newtype是分布式数组类型。

下面的程序片段是使用分布式数组构造符的一个例子。对于一个m*n的2维全局浮点数组，每一维都按块分布，6个处理器构成阵列2*3的阵列，局部数组在内存中以行优先来排列，通过调用MPI_Type_create_darray得到一个分布式数组类型filetype，然后用filetype作为文件类型来定义文件视口，这样将内存中连续排列的局部数组写入文件视口时，就会以正确的方式将该块局部数据在全局数组文件的指定部分进行更新，这样程序员就不必去计算复杂的全局与局部数组的对应关系，这也是定义分布式数组文件类型的一个优点。

```
gsizes[0]=m;
gzises[1]=n;
distribs[0]=MPI_DISTRIBUTE_BLOCK;
distribs[1]=MPI_DISTRIBUTE_BLOCK;
```

```
dargs[0]=MPI_DISTRIBUTE_DFLT_DARG;
dargs[1]=MPI_DISTRIBUTE_DFLT_DARG;
psizes[0]=2;
psizes[1]=3;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Type_create_darray(6,rank,2,gsizes,distribs,dargs,psizes,MPI_ORDER_C,MPI_FLOAT,
&filetype);
MPI_Type_commit(&filetype);
MPI_File_open(MPI_COMM_WORLD, "datafile",MPI_MODE_CREATE|
              MPI_MODE_WRONLY,MPI_INFO_NULL,&fh);
MPI_File_set_view(fh,0,MPI_FLOAT,filetype,"native",MPI_INFO_NULL);
local_array_size=num_local_rows*num_local_cols;
MPI_File_write_all(fh,local_array,local_array_size,MPI_FLOAT,&status);
MPI_File_close(&fh);
```

程序 67 分布式数组的定义

```
MPI_TYPE_CREATE_SUBARRAY(ndims,array_of_sizes, array_of_subsizes,
      array_of_starts, order, oldtype, newtype)
IN     ndims              数组维数
IN     array_of_sizes     每一维数组的大小
IN     array_of_subsizes  子数组每一维的大小
IN     array_of_starts    子数组起始坐标在数组中每一维的相对偏移
IN     order              与创建分布式数组类型时使用的优先方式相同
IN     oldtype            数组元素的类型
OUT    newtype            返回的子数组类型
int  MPI_Type_create_subarray(int ndims, int array_of_sizes[], int array_of_subsizes[],
     int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype * newtype)
MPI_TYPE_CREAT_SUBARRAY(NDIMS, ARRAY_OF_SIZES,
       ARRAY_OF_SUBSIZES,ARRAY_OF_STARTS,ORDER,OLDTYPE,NEWTYPE,
        IERROR)
INTEGER  NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
            ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```

MPI调用接口 208 MPI_TYPE_CREATE_SUBARRAY

MPI_TYPE_CREATE_SUBARRAY是子数组数据类型构造符，它定义数组的一部分作为一个新的数据类型。对于一个ndims维数组，数组每一维的大小由array_of_sizes定义，而子数组相应维的大小由array_of_subsizes定义，子数组第一个元素相对于整个数组在每一维上的相对偏移由array_of_starts定义，子数组在内存的排列次序（行优先或列优先）由order定义，数组元素的类型为oldtype，而返回的新数据类型newtype是子数组类型。

下面的程序片段是使用子数组构造符的一个例子。对于一个m*n的2维全局浮点数组，分布在6个处理器构成阵列2*3的阵列上，子数组每一维的大小分别为m/2和n/3，子数组在内存中以行优先来排列，子数组第一个元素的相对偏移为start_indices[0]=coords[0]*lsizes[0]和

start_indices[1]=coords[1]*lsizes[1]，通过调用MPI_Type_create_subarray得到一个分布式子数组类型filetype，然后用filetype作为文件类型来定义文件视口，这样将内存中连续排列的局部数组写入文件视口时，就会以正确的方式将该块局部数据在全局数组文件的指定部分进行更新，这样程序员就不必去计算复杂的全局与局部数组的对应关系，这也是定义分布式数组文件类型的一个优点。

下面的程序片段

```
gsizes[0]=m;
gsizes[1]=n;
psizes[0]=2;
psizes[3]=3;
lsizes[0]=m/psizes[0];
lsizes[1]=n/psizes[1];
dims[0]=2;
dims[1]=3;
periods[0]=periods[1]=1;
MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods, 0 &comm);
MPI_Comm_rank(comm,&rank);
MPI_Cart_coords(comm,rank,2,coords);
start_indices[0]=coords[0]*lsizes[0];
start_indices[1]=coords[1]*lsizes[1];
MPI_Type_create_subarray(2,gsizes,lsizes,start_indices,MPI_ORDER_C,MPI_FLOAT,&filetype);
MPI_Type_commit(&filetype);
MPI_File_open(MPI_COMM_WORLD,"datafile",MPI_MODE_CREATE|
MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_setview(fh,0,MPI_FLOAT,filetype,"native",MPI_INFO_NULL);
memsizes[0]=lsizes[0]+8;
memsizes[1]=lsizes[1]+8;
/*  内存数组两侧都预留出4个单元*/
start_indices[0]=start_indices[1]=4;
MPI_Type_subarray(2,memsizes,lsizes,start_indices,MPI_ORDER_C,MPI_FLOAT,&memtype);
/*定义内存数组的子数组*/
MPI_Type_commit(&memtype);
MPI_File_write_all(fh,local_array,1,memtype,&status);
MPI_File_close(&fh);
```

程序 68 子数组的定义

21.7 小结

并行文件I/O是并行计算的一个重要内容，MPI作为一种标准的并行开发环境，必然对这部分的内容进行规范，本章的调用比较多，这些调用是在对已有的各种并行文件系统和并行文件操作进行充分考察后提出来的，它充分说明了并行I/O的复杂性和已有的并行I/O的成果。

网上资源

主要的MPI主页及MPI标准

http://www.mpi-forum.org	MPIF**主页**
http://www.mcs.anl.gov/mpi	MPI**主页**
http://www.netlib.org/mpi/index.html	netlib**上的**MPI**站点**

MPI**的实现**

http://www-unix.mcs.anl.gov/mpi/mpich/	ANL/MSU**实现的**MPICH**主页**
http://www.mcs.anl.gov/mpi/mpich	MPICH**实现**
ftp://ftp.mcs.anl.gov/pub/mpi	MPICH**实现**
http://www.mpi.nd.edu/MPI	**其它的**MPI**实现**
http://www.lsc.nd.edu/MPI2	MPI**实现列表**
http://www.erc.msstate.edu/mpi	MSU**的**MPI**项目**
http://www.mpi.nd.edu/lam/	LAM**的**MPI

关于MPI的文档、讨论与例子

http://www-unix.mcs.anl.gov/mpi/tutorial/	MPI**相关材料**
http://www.erc.msstate.edu/mpi/mpi-faq.html	**常见问题**
http://www-unix.mcs.anl.gov/mpi/mpich/faq.html	**常见问题解答**
http://www.mpi-forum.org/docs	**勘误**
comp.parallel.mpi	**关于**MPI**的新闻组**
ftp://ftp.mpi-forum.org/pub/docs/	MPI**文挡**
http://www.mcs.anl.gov/mpi/usingmpi	MPI**教程**
http://www.mcs.anl.gov/mpi/usingmpi2	MPI**教程**
ftp://ftp.mcs.anl.gov/pub/mpi/using/examples	MPI**例子程序**
ftp://ftp.mcs.anl.gov/pub/mpi/using2/examples	MPI**例子程序**
http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl/contents.html	MPI**例子程序**

参考文献

[Ado98] Jean-Marc Adamo. Multi-threaded object-oriented MPI-based message passing interface: the ARCH library. Boston : Kluwer Academic, 1998. ISBN 0792381653.

[Ads97] Jeanne C.Adams. Fortran 95 Handbook. 1997.

[Akn87] Edited by Akinori Yonezawa and Mario Tokoro. Object-oriented concurrent programming. Cambridge, Mass. : MIT Press, 1987. ISBN 0262240262.

[Akl89] Selim G. Akl. The design and analysis of parallel algorithms. Englewood Cliffs, N.J. : Prentice Hall, 1989. ISBN 0132000563.

[Alv98] Vassil Alexandrov, Jack Dongarra (eds.). Recent advances in parallel virtual machine and message passing interface : 5th European PVM/MPI User's Group Meeting, Liverpool, UK, September 7-9, 1998 : proceedings. Berlin ; New York : Springer, 1998. ISBN 3540650415.

[Ans91] Gregory R.Andrews. Concurrent programming : principles and practice. Redwood City, Calif. : Benjamin/Cummings Pub. Co., 1991. ISBN 0805300864.

[Bab88] Edited by Robert G. Babb. Programming parallel processors. Reading, Mass. : Addison-Wesley Pub. Co., 1988. ISBN 0201117215.

[Bar92] Barr E. Bauer. Practical parallel programming. San Diego : Academic Press, 1992. ISBN 0120828103.

[Brc93] Lester, Bruce P. The art of parallel programming .Englewood Cliffs, N.J. : Prentice Hall, 1993. ISBN 0130459232.

[Bus88] Alan Burns. Programming in Occam 2. Wokingham [Berkshire] England Reading, Mass. Addison-Wesley, 1988. ISBN 0201173719.

[Car90] Nicholas Carriero, David Gelernter. How to write parallel programs : a first course. Cambridge, Mass. : MIT Press, 1990. ISBN 026203171X.

[Chay88] K. Mani Chandy, Jayadev Misra. Parallel program design : a foundation. Reading, Mass. : Addison-Wesley Pub. Co., 1988. ISBN 0201058669.

[Chs92] Cheese, A. (Andrew). Parallel execution of Parlog. Berlin : Springer-Verlag, 1992. ISBN 0387553827 (New York).

[Con92] Michael H. Coffin. Parallel programming : a new approach. Summit, NJ : Silicon Press, 1992. ISBN 0929306139.

[Fok95] Lloyd D.Fosdick ... [et al]. An Introduction to High-Performance Scientific Computing. 1995.

[For94] Ian Foster . Designing and building parallel programs : concepts and tools for parallel software engineering. Reading, Mass. : Addison-Wesley, 1994. ISBN 0201575949.

[Gen88] Narain Gehani, Andrew McGettrick.Concurrent programming. Wokingham, England : Addison-Wesley, 1988. ISBN 0201174359.

[Gen89] Narain Gehani, William D. Roome. The Concurrent C programming language. Summit, NJ, USA : Silicon Press, 1989. ISBN 0929306007.

[Gej97] Robert A.van de Geijn. Using PLAPACK: Parallel Linear Algebra Package. 1997.

[Get94] Al Geist ... [et al]. PVM: Parallel Virtual Machine-- A Users's Guide and Tutorial for Network Parallel Computing. 1994.

[Gry87] Steve Gregory. Parallel logic programming in PARLOG : the language and its

implementation. Wokingham, England ; Reading, Mass. : Addison-Wesley Pub. Co., 1987. ISBN 0201192411, 0201192412.

[Grp99] William Gropp, Ewing Lusk, Anthony Skjellum. Using MPI : portable parallel programming with the message-passing interface Cambridge, Mass. : MIT Press, 1999. 2nd edition. ISBN 0262571323.

[Grp99] William Gropp, Ewing Lusk, Rajeev Thakur. Using MPI-2 : advanced features of the message-passing interface. Cambridge, Mass. : MIT Press, 1999. ISBN 0262571331.

[Har91] Philip J. Hatcher and Michael J. Quinn. Data-parallel programming on MIMD computers . Cambridge, Mass. : MIT Press, c1991. ISBN 0262082055.

[Kol94] Charles H.Koelbel, David B.Loveman ...[et al]. The High Performance Fortran Handbook. 1994

[Pen96] Guy-RenPerrin, Alain Darte, (eds.). The data parallel programming model : foundations, HPF realization, and scientific applications，Berlin ; New York : Springer, 1996。ISBN 3540617361 (Berlin : acid-free paper)。

[Pet87] R.H. Perrott. Parallel programming. Wokingham, England : Addison-Wesley Pub. Co., 1987. ISBN 0201142317.

[Pos88] Constantine D. Polychronopoulos. Parallel programming and compilers. Boston : Kluwer Academic, 1988. ISBN 0898382882.

[Ral91] Susann Ragsdale, editor. Parallel programming. New York : McGraw-Hill, 1991. ISBN 0070511861.

[Sat88] Gary Sabot. The paralation model : architecture-independent parallel programming. Cambridge, Mass. : MIT Press, 1988. ISBN 0262192772.

[Snr97] Marc Snir, Steve Otto, Steven Huss Lederman, David Walker, Jack Dongarra. MPI: the Complete Reference. the MIT Press, 1997

[Snr98] Marc Snir ... [et al.]. MPI--the complete reference. Cambridge, Mass. : MIT Press, 1998. 2nd edtion. ISBN 0262692155, 0262692163.

[Snw92] C.R. Snow. Concurrent programming. New York : Combridge University Press, 1992. ISBN 0521327962.

[Tik91] Evan Tick. Parallel logic programming. Cambridge, Mass. : MIT Press, 1991. ISBN 0262200872.

[Wim96] William H. Press ... [et al.] Numerical recipes in Fortran 90 : the art of parallel scientific computing. Cambridge [England] ; New York : Cambridge University Press, 1996. Edition 2nd ed. ISBN 0521574390 (hardcover).

[Wis90] Shirley A. Williams. Programming models for parallel systems. New York : J. Wiley, 1990. ISBN 0471923044.

[Win96] Edited by Gregory V.Wilson ... [et ..al]. Parallel Programming Using C++.1996.

[Win95] Gregory V. Wilson. Practical Parallel Programming.1995

[Yag87] Rong Yang. P-Prolog, a parallel logic programming language. Singapore : World Scientific, 1987. ISBN 9971505088.

[Yun93] C.K. Yuen ... [et al.]. Parallel lisp systems : a study of languages and architectures. London : Chapman & Hall, 1993. ISBN 0442315686.

中英文术语对照

Aliased Argument	别名参数
Asynchronous Communication	异步通信
Attributes	属性
Bandwidth	带宽
Blocking Communication	阻塞通信
Blocking Receive	阻塞接收
Blocking Send	阻塞发送
Buffer	缓冲区
Buffered Communication Mode	缓存通信通信
Caching of Attributes	属性缓冲
Cartesian Topology	笛卡儿拓扑
Collective Communication	组通信
Communication Modes	通信模式
Communication Processor	通信处理机
Communicator	通信域
Context	上下文
Contiguous Data	数据连续
Datatypes	数据类型
Deadlock	死锁
Event	事件
Graph Topology	图拓扑
Group	进程组
Heterogeneous Computing	异构计算
InterCommunicator	组间通信域
IntraCommunicator	组内通信域
Latency	延时
MPI	Message Passing Interface
MPIF	MPI论坛
Multicomputer	多计算机
NoBlocking Communication	非阻塞通信
NoBlocking Receive	非阻塞接收
NoBlocking Send	非阻塞发送
Node	结点
Persistent Requests	重复通信对象
Physical Topology	物理拓扑
Point-to-Point Communication	点到点通信
Portability	移植性
Process	进程
Processor	处理器
PVM	Parallel Virtual Machine
Rank	进程标识号
Ready	就绪
Ready Communication Mode	就绪通信模式

Reduce	**归约**
Request Object	**归约对象**
Safe Programs	**安全程序**
Standard Communication Mode	**标准模式**
Status Object	**状态对象**
Subgroup	**子进程组**
Synchronization	**同步**
Synchronous Communication Mode	**同步通信模式**
Thread	**线程**
Topology	**拓扑**
Type Map	**类型图**
Type Signature	**类型表**
User-Defined Topology	**用户定义拓扑**
Virtual Shared Memory	**虚拟共享内存**
Virtual Topology	**虚拟拓扑**

本书介绍的MPI调用索引

附录1　MPI常量列表

1. C **数据类型常量**

MPI的C数据类型常量	对应的C类型
MPI_CHAR	char
MPI_BYTE	
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_LONG_DOUBLE	long double (some systems may not implement)

2. MPI_MAXLOC**和**MPI_MINLOC**在**C**中用到的类型**

MPI数据类型	对应的C类型
MPI_FLOAT_INT	struct { float, int }
MPI_LONG_INT	struct { long, int }
MPI_DOUBLE_INT	struct { double, int }
MPI_SHORT_INT	struct { short, int }
MPI_2INT	struct { int, int }
MPI_LONG_DOUBLE_INT	struct { long double, int };**可选**
MPI_LONG_LONG_INT	struct { long long, int };**可选**

3. **特别的数据类型**

MPI数据类型	用途
MPI_PACKED	For MPI_Pack and MPI_Unpack
MPI_UB	For MPI_Type_struct; an upper-bound indicator
MPI_LB	For MPI_Type_struct; a lower-bound indicator

4. Fortran**数据类型**

MPI数据类型	对应的Fortran数据类型
MPI_REAL	REAL
MPI_INTEGER	INTEGER
MPI_LOGICAL	LOGICAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	complex*16**或**complex*32

5. FORTRAN**中可选数据类型**

MPI**数据类型**	Fortran**数据类型**
MPI_INTEGER1	integer*1
MPI_INTEGER2	integer*2
MPI_INTEGER4	integer*4
MPI_REAL4	real*4
MPI_REAL8	real*8

6. MPI_MAXLOC**和**MPI_MINLOC**在**Fortran**中用到的类型，它们对应两个基本的**Fortran**数据类型**

MPI**数据类型**	Fortran**数据类型**
MPI_2INTEGER	INTEGER,INTEGER
MPI_2REAL	REAL, REAL
MPI_2DOUBLE_PRECISION	DOUBLE PRECISION, DOUBLE PRECISION
MPI_2COMPLEX	COMPLEX, COMPLEX
MPI_2DOUBLE_COMPLEX	complex*16, complex*16

7. **通信域，在**C**中其类型为**MPI_Comm**，在**Fortran**中为**Fortran

可用的预定义通信域	**含义**
MPI_COMM_WORLD	**包含所有进程**
MPI_COMM_SELF	**只包含调用进程本身**

8. **进程组，进程组在**C**中是**MPI_Group**类型，在**Fortran**是**INTEGER**类型**

预定义进程组	**含义**
MPI_GROUP_EMPTY	**不包含任何成员的进程组**

9. **进程组比较结果**

比较返回的结果	**含义**
MPI_IDENT	**完全相同**
MPI_CONGRUENT	**进程组完全相同**
MPI_SIMILAR	**成员相同但顺序不同**
MPI_UNEQUAL	**不同**

10. **组调用，一些组调用要进行运算操作**（MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, and MPI_SCAN）**，该操作的类型在**C**中是**MPI_Op**类型，在**Fortran**中是**INTEGER**类型，**

预定义组调用操作类型	**含义**
MPI_MAX	**最大值**
MPI_MIN	**最小值**
MPI_SUM	**求和**
MPI_PROD	**求积**
MPI_LAND	**逻辑与**

MPI_BAND	**按位与**
MPI_LOR	**逻辑或**
MPI_BOR	**按位或**
MPI_LXOR	**逻辑异或**
MPI_BXOR	**按位异或**
MPI_MINLOC	**返回最小值和位置**
MPI_MAXLOC	**返回最大值和位置**

11. **关键字值**，C**和**Fortran **的数据类型相同**

关键字	**含义**
MPI_TAG_UB	**最大的**tag**标识数值**
MPI_HOST	**主进程标识**
MPI_IO	I/O**进程标识**
MPI_WTIME_IS_GLOBAL	**若**MPI_WTIME**全局同步则返回**1

12. **无效对象**

无效对象标识	**含义**
MPI_COMM_NULL	**无效通信域**
MPI_OP_NULL	**无效操作**
MPI_GROUP_NULL	**无效进程组**
MPI_DATATYPE_NULL	**无效数据类型**
MPI_REQUEST_NULL	**无效对象**
MPI_ERRHANDLER_NULL	**无效错误句柄**

13. **预定义常量**

常量名	**含义**
MPI_MAX_PROCESSOR_NAME	**机器名最大长度**
MPI_MAX_ERROR_STRING	**错误字符串最大长度**
MPI_UNDEFINED	**用于无定义或不知道的整数值**
MPI_UNDEFINED_RANK	**不知道的进程标识**
MPI_KEYVAL_INVALID	**未初始化的无效关键字值**
MPI_BSEND_OVERHEAD	**缓存发送时增加的空间**
MPI_PROC_NULL	**空的进程标识**
MPI_ANY_SOURCE	**从任何源接收**
MPI_ANY_TAG	**匹配任何**tag**标识**
MPI_BOTTOM	**地址空间的绝对起始地址**

14. **拓扑类型**

拓扑类别	**含义**
MPI_GRAPH	**通用图拓扑**
MPI_CART	**笛卡儿拓扑**

15. MPI **状态**，MPI_Status**数据类型时一个结构，它包括三个成员。**

状态成员	**含义**
MPI_SOURCE	**发送消息的进程**
MPI_TAG	**该消息的标识**
MPI_ERROR	**返回的错误代码**

16. **其它类型与函数**

MPI_Aint	C**中的地址类型**
MPI_Handler_function	C**中处理错误的函数**
MPI_User_function	C**中的用户定义函数**
MPI_Copy_function	**属性拷贝函数**
MPI_NULL_COPY_FN	**预定义拷贝函数**
MPI_Delete_function	**删除属性函数**
MPI_NULL_DELETE_FN	**删除函数**
MPI_DUP_FN	**复制函数**
MPI_ERRORS_ARE_FATAL	**强制退出的错误句柄**
MPI_ERRORS_RETURN	**返回错误代码的句柄**

17. MPI **错误代码**

错误代码	**含义**
MPI_SUCCESS	**成功**
MPI_ERR_BUFFER	**无效缓冲区指针**
MPI_ERR_COUNT	**无效记数参数**
MPI_ERR_TYPE	**无效数据类型**
MPI_ERR_TAG	**无效**tag**标识**
MPI_ERR_COMM	**无效通信域**
MPI_ERR_RANK	**无效进程标识**
MPI_ERR_ROOT	**无效**ROOT**根进程**
MPI_ERR_GROUP	**传递给函数的组为空**
MPI_ERR_OP	**无效操作**
MPI_ERR_TOPOLOGY	**无效拓扑**
MPI_ERR_DIMS	**不合法的维**
MPI_ERR_ARG	**无效参数**
MPI_ERR_UNKNOWN	**未知的错误**
MPI_ERR_TRUNCATE	**接收时消息被截断**
MPI_ERR_OTHER	**用错误字符串表示的其它错误**
MPI_ERR_INTERN	**内部错误代码**
MPI_ERR_IN_STATUS	**错误在**status**中**
MPI_ERR_PENDING	**孤立挂起对象**
MPI_ERR_REQUEST	**不合法的对象句柄**
MPI_ERR_LASTCODE	**最后一个错误代码**

附录2　MPICH 1.2.1函数列表

1. MPI **调用列表**

MPI_Abort	MPI_Address
MPI_Allgather	MPI_Allgatherv
MPI_Allreduce	MPI_Alltoall
MPI_Alltoallv	MPI_Attr_delete
MPI_Attr_get	MPI_Attr_put
MPI_Barrier	MPI_Bcast
MPI_Bsend	MPI_Bsend_init
MPI_Buffer_attach	MPI_Buffer_detach
MPI_Cancel	MPI_Cart_coords
MPI_Cart_create	MPI_Cart_get
MPI_Cart_map	MPI_Cart_rank
MPI_Cart_shift	MPI_Cart_sub
MPI_Cartdim_get	MPI_CHAR
MPI_Comm_compare	MPI_Comm_create
MPI_Comm_dup	MPI_Comm_free
MPI_Comm_group	MPI_Comm_rank
MPI_Comm_remote_group	MPI_Comm_remote_size
MPI_Comm_size	MPI_Comm_split
MPI_Comm_test_inter	MPI_Dims_create
MPI_DUP_FN	MPI_Errhandler_create
MPI_Errhandler_free	MPI_Errhandler_get
MPI_Errhandler_set	MPI_Error_class
MPI_Error_string	MPI_File_c2f
MPI_File_close	MPI_File_delete
MPI_File_f2c	MPI_File_get_amode
MPI_File_get_atomicity	MPI_File_get_byte_offset
MPI_File_get_errhandler	MPI_File_get_group
MPI_File_get_info	MPI_File_get_position
MPI_File_get_position_shared	MPI_File_get_size
MPI_File_get_type_extent	MPI_File_get_view
MPI_File_iread	MPI_File_iread_at
MPI_File_iread_shared	MPI_File_iwrite
MPI_File_iwrite_at	MPI_File_iwrite_shared
MPI_File_open	MPI_File_preallocate
MPI_File_preallocate	MPI_File_read
MPI_File_read_all	MPI_File_read_all_begin
MPI_File_read_all_end	MPI_File_read_at
MPI_File_read_at_all	MPI_File_read_at_all_begin
MPI_File_read_at_all_end	MPI_File_read_ordered
MPI_File_read_ordered_begin	MPI_File_read_ordered_end
MPI_File_read_shared	MPI_File_seek
MPI_File_seek_shared	MPI_File_set_atomicity
MPI_File_set_errhandler	MPI_File_set_info
MPI_File_set_size	MPI_File_set_view
MPI_File_sync	MPI_File_write
MPI_File_write_all	MPI_File_write_all_begin
MPI_File_write_all_end	MPI_File_write_at

MPI_File_write_at_all	MPI_File_write_at_all_begin
MPI_File_write_at_all_end	MPI_File_write_ordered
MPI_File_write_ordered_begin	MPI_File_write_ordered_end
MPI_File_write_shared	MPI_Finalize
MPI_Finalized	MPI_Gather
MPI_Gatherv	MPI_Get_count
MPI_Get_elements	MPI_Get_processor_name
MPI_Get_version	MPI_Graph_create
MPI_Graph_get	MPI_Graph_map
MPI_Graph_neighbors	MPI_Graph_neighbors_count
MPI_Graphdims_get	MPI_Group_compare
MPI_Group_difference	MPI_Group_excl
MPI_Group_free	MPI_Group_incl
MPI_Group_intersection	MPI_Group_range_excl
MPI_Group_range_incl	MPI_Group_rank
MPI_Group_size	MPI_Group_translate_ranks
MPI_Group_union	MPI_Ibsend
MPI_Info_c2f	MPI_Info_create
MPI_Info_delete	MPI_Info_dup
MPI_Info_f2c	MPI_Info_free
MPI_Info_get	MPI_Info_get_nkeys
MPI_Info_get_nthkey	MPI_Info_get_valuelen
MPI_Info_set	MPI_Init
MPI_Init_thread	MPI_Initialized
MPI_Int2handle	MPI_Intercomm_create
MPI_Intercomm_merge	MPI_Iprobe
MPI_Irecv	MPI_Irsend
MPI_Isend	MPI_Issend
MPI_Keyval_create	MPI_Keyval_free
MPI_NULL_COPY_FN	MPI_NULL_DELETE_FN
MPI_Op_create	MPI_Op_free
MPI_Pack	MPI_Pack_size
MPI_Pcontrol	MPI_Probe
MPI_Recv	MPI_Recv_init
MPI_Reduce	MPI_Reduce_scatter
MPI_Request_c2f	MPI_Request_free
MPI_Rsend	MPI_Rsend_init
MPI_Scan	MPI_Scatter
MPI_Scatterv	MPI_Send
MPI_Send_init	MPI_Sendrecv
MPI_Sendrecv_replace	MPI_Ssend
MPI_Ssend_init	MPI_Start
MPI_Startall	MPI_Status_c2f
MPI_Status_set_cancelled	MPI_Status_set_elements
MPI_Test	MPI_Test_cancelled
MPI_Testall	MPI_Testany
MPI_Testsome	MPI_Topo_test
MPI_Type_commit	MPI_Type_contiguous
MPI_Type_create_darray	MPI_Type_create_indexed_block
MPI_Type_create_subarray	MPI_Type_extent
MPI_Type_free	MPI_Type_get_contents
MPI_Type_get_envelope	MPI_Type_hindexed
MPI_Type_hvector	MPI_Type_indexed
MPI_Type_lb	MPI_Type_size
MPI_Type_struct	MPI_Type_ub

MPI_Type_vector	MPI_Unpack
MPI_Wait	MPI_Waitall
MPI_Waitany	MPI_Waitsome
MPI_Wtick	MPI_Wtime
MPIO_Request_c2f	MPIO_Request_f2c
MPIO_Test	MPIO_Wait

2. MPE**调用列表**

CLOG_commtype	CLOG_cput
CLOG_csync	CLOG_Finalize
CLOG_get_new_event	CLOG_get_new_state
CLOG_Init	CLOG_init_buffers
CLOG_mergelogs	CLOG_mergend
CLOG_msgtype	CLOG_newbuff
CLOG_nodebuffer2disk	CLOG_Output
CLOG_procbuf	CLOG_reclen
CLOG_rectype	CLOG_reinit_buff
CLOG_treesetup	MPE
MPE_Add_RGB_color	MPE_CaptureFile
MPE_CaptureFile	MPE_Close_graphics
MPE_Comm_global_rank	MPE_Counter_create
MPE_Counter_free	MPE_Counter_nxtval
MPE_Decomp1d	MPE_Describe_event
MPE_Describe_state	MPE_Draw_circle
MPE_Draw_line	MPE_Draw_logic
MPE_Draw_point	MPE_Draw_points
MPE_Draw_string	MPE_Fill_circle
MPE_Fill_rectangle	MPE_Finish_log
MPE_Get_mouse_press	MPE_GetTags
MPE_Iget_mouse_press	MPE_Iget_mouse_press
MPE_Init_log	MPE_Initialized_logging
MPE_IO_Stdout_to_file	MPE_Line_thickness
MPE_Log_event	MPE_Log_get_event_number
MPE_Log_receive	MPE_Log_send
MPE_Make_color_array	MPE_Num_colors
MPE_Open_graphics	MPE_Print_datatype_pack_action
MPE_Print_datatype_unpack_action	MPE_ReturnTags
MPE_Seq_begin	MPE_Seq_end
MPE_Start_log	MPE_Stop_log
MPE_TagsEnd	MPE_Update

```
/* test program to find out how much buffering a system supplies */

#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];
{
      int    myid, numprocs;
      int    namelen;
      char  processor_name[MPI_MAX_PROCESSOR_NAME];
      char  *buf;
      int   bufsize, other, done, i;
      double t1, t2, tbase;
      MPI_Status  status;

       MPI_Init(&argc,&argv);
       MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
       MPI_Comm_rank(MPI_COMM_WORLD,&myid);

     /* Output processor names in rank order */
       MPI_Get_processor_name(processor_name,&namelen);
     if (myid > 0)
       MPI_Recv( MPI_BOTTOM, 0, MPI_INT, myid - 1, 5, MPI_COMM_WORLD,
                 &status );
      fprintf(stderr,"Process  %d on %s\n", myid, processor_name);
       fflush( stderr );
     if (myid + 1 < numprocs)
       MPI_Send( MPI_BOTTOM, 0, MPI_INT, myid + 1, 5, MPI_COMM_WORLD );


      bufsize = 1024;
      other     = (myid + 1) % 2;
      done      = 0;

     while (!done && bufsize < 1024*1024*16) {
       if ((buf = (char *) malloc (bufsize)) == NULL) {
            fprintf(stderr, "%d could not malloc %d bytes\n", myid, bufsize );
             MPI_Abort( MPI_COMM_WORLD, 1 );
             exit(-1);
       }
       /* fprintf(stderr,"%d sending %d to %d\n", myid, bufsize, other ); */
```

```
    if ((myid % 2) == 0) {
        MPI_Send( MPI_BOTTOM, 0, MPI_INT, other, 1, MPI_COMM_WORLD );
        MPI_Recv( MPI_BOTTOM, 0, MPI_INT, other, 2, MPI_COMM_WORLD,
                  &status );
        /* Compute a time to send when the receive is waiting */
         t1 = MPI_Wtime();
        MPI_Send( buf, bufsize, MPI_CHAR, other, 100, MPI_COMM_WORLD );
         t2 = MPI_Wtime();
         tbase = t2 - t1;
        MPI_Recv( MPI_BOTTOM, 0, MPI_INT, other, 2, MPI_COMM_WORLD,
                  &status );
        /* Compute a time when the receive is NOT waiting */
         t1 = MPI_Wtime();
        MPI_Send( buf, bufsize, MPI_CHAR, other, 100, MPI_COMM_WORLD );
         t2 = MPI_Wtime();
        if (t2 - t1 > 1.5 && t2 - t1 > 2.0 * tbase) {
          printf( "MPI_Send blocks with buffers of size %d\n",
                bufsize );
          done = 1;
          }
    }
    else {
        MPI_Recv( MPI_BOTTOM, 0, MPI_INT, other, 1, MPI_COMM_WORLD,
                  &status );
         t1 = MPI_Wtime();
        MPI_Send( MPI_BOTTOM, 0, MPI_INT, other, 2, MPI_COMM_WORLD );
        MPI_Recv( buf, bufsize, MPI_CHAR, other, 100, MPI_COMM_WORLD,
                  &status );
        MPI_Send( MPI_BOTTOM, 0, MPI_INT, other, 2, MPI_COMM_WORLD );
         while (MPI_Wtime() - t1 < 2.0) ;
        MPI_Recv( buf, bufsize, MPI_CHAR, other, 100, MPI_COMM_WORLD,
                  &status );
    }
    fprintf(stderr,"%d received %d fr %d\n", myid, bufsize, other );
    free( buf );
    i = done;
    MPI_Allreduce( &i, &done, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD );
    bufsize *= 2;
    }
    MPI_Finalize();
    return 0;
}
```

```
/*************************************************/

      program main
      double precision a(8,8), alocal(4,4)
      integer i, j, r, rank, size, sizeofdbl, ierr
      integer stype, t(2), vtype
      integer displs(2)
      integer blklen(2)
      integer sendcount(4), sdispls(4)
      include 'mpif.h'

      call MPI_Init( ierr )
      call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
      call MPI_Comm_size( MPI_COMM_WORLD, size, ierr )

      if (size .ne. 4) then
          print *, '"This program requires exactly four processors'
         call MPI_Abort( MPI_COMM_WORLD, 1, ierr )
      endif

      if (rank .eq. 0) then
C  Initialize the matrix.  Note that C has row-major storage
          do 10 j=1,8
              do 10 i=1,8
 10           A(i,j) = 1.0 + i / 10.0d0 + j / 100.0d0
C Form the vector type for the submatrix
              call MPI_Type_vector( 4, 4, 8, MPI_DOUBLE_PRECISION,
     *                              vtype, ierr )
C Set an UB so that we can place this in the matrix
          t(1) = vtype
          t(2) = MPI_UB
          displs(1) = 0
          call MPI_Type_size( MPI_DOUBLE_PRECISION, sizeofdbl, ierr )
          displs(2) = 4 * sizeofdbl
          blklen(1) = 1
          blklen(2) = 1
          call MPI_Type_struct( 2, blklen, displs, t, stype, ierr )
          call MPI_Type_commit( stype, ierr )
C Setup the Scatter values for the send buffer
          sendcount(1) = 1
          sendcount(2) = 1
          sendcount(3) = 1
          sendcount(4) = 1
```

```
      sdispls(1) = 0
      sdispls(2) = 1
      sdispls(3) = 8
      sdispls(4) = 9
      call MPI_Scatterv( A, sendcount, sdispls, stype, alocal, 4*4,
     *                   MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr )
      else
         call MPI_Scatterv( MPI_BOTTOM, sendcount, sdispls, stype,
     *                   alocal, 4*4, MPI_DOUBLE_PRECISION, 0,
     *                   MPI_COMM_WORLD, ierr )
      endif

C Everyone can now print their local matrix
      do r=0, size-1
         if (rank .eq. r) then
            print *, "Output for process ", r
              do i=1,4
            print *, (alocal(i,j),j=1,4)
              enddo
         endif
        call MPI_Barrier( MPI_COMM_WORLD, ierr )
      enddo

      call MPI_Finalize( ierr )
      stop
      end
```

```
/%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%/
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
int argc;
char *argv[];
{
     double  A[8][8], alocal[4][4];
     int i, j, r, rank, size;
     MPI_Datatype stype, t[2], vtype;
     MPI_Aint        displs[2];
     int             blklen[2];
     int             sendcount[4], sdispls[4];
```

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
 MPI_Comm_size( MPI_COMM_WORLD, &size );
if (size != 4) {
 fprintf( stderr, "This program requires exactly four processors\n" );
 MPI_Abort( MPI_COMM_WORLD, 1 );
 }
if (rank == 0) {
 /* Initialize the matrix.  Note that C has row-major storage */
 for (j=0; j<8; j++)
       for (i=0; i<8; i++)
        A[i][j] = 1.0 + i / 10.0 + j / 100.0;
 /* Form the vector type for the submatrix */
 MPI_Type_vector( 4, 4, 8, MPI_DOUBLE, &vtype );
 /* Set an UB so that we can place this in the matrix */
 t[0] = vtype;
 t[1] = MPI_UB;
 displs[0] = 0;
 displs[1] = 4 * sizeof(double);
 blklen[0] = 1;
 blklen[1] = 1;
 MPI_Type_struct( 2, blklen, displs, t, &stype );
 MPI_Type_commit( &stype );
 /* Setup the Scatter values for the send buffer */
 sendcount[0] = 1;
 sendcount[1] = 1;
 sendcount[2] = 1;
 sendcount[3] = 1;
 sdispls[0] = 0;
 sdispls[1] = 1;
 sdispls[2] = 8;
 sdispls[3] = 9;
 MPI_Scatterv( &A[0][0], sendcount, sdispls, stype,
              &alocal[0][0], 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD );

 }
 else {
 MPI_Scatterv( (void *)0, (void *)0, (void *)0, MPI_DATATYPE_NULL,
              &alocal[0][0], 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD );
 }

/* Everyone can now print their local matrix */
for (r = 0; r<size; r++) {
```

```
    if (rank == r) {
        printf( "Output for process %d\n", r );
        for (j=0; j<4; j++) {
          for (i=0; i<4; i++)
              printf( "%.2f ", alocal[i][j] );
          printf( "\n" );
          }
          fflush( stdout );
    }
    MPI_Barrier( MPI_COMM_WORLD );
    }

    MPI_Finalize( );
    return 0;
}

/*****************************************************************************/
```