第3章

消息传递系统并行编程

消息传递系统又被称为分布式内存系统，这种系统通常使用互连网络将多个计算节点连到一起，每个节点拥有独立的内存空间，并运行独立的操作系统。消息传递系统的可扩展性好，适合运行大规模并行程序。在此类系统中，程序以多进程并行的方式运行在各个计算节点，进程之间通过消息传输的方式实现数据交换和同步，以协同完成整个计算任务。正由于可扩展性好，当前几乎所有的高性能计算机系统都采用此种架构。

消息传递系统采用的最主流并行编程模型是 MPI（message passing interface），本章将重点讨论如何使用 MPI 进行并行编程。除了 MPI 编程之外，近年来还出现了一类新的用于消息传递系统的编程语言——PGAS 语言，这类语言试图将共享内存编程的一些特点引入消息传递系统中，以改善消息传递系统的可编程性和软件生产率。本章最后还将介绍主流的作业管理系统 Slurm，它通常被安装运行在高性能计算机系统中，以实现多个用户共享使用高性能计算资源之目的。

3.1 MPI 简介

3.1.1 MPI 是什么？

以集群系统为代表的消息传递系统在 20 世纪 80 年代后期兴起，为了在这类系统上编写并行程序，消息传递型编程接口开始出现，早期比较有代表性的是 PVM（parallel virtual machine），后来 MPI（message passing interface）接口规范于 1994 年发布，凭借其更强的消息通信功能和灵活的特性逐渐占据主流。

MPI 是一种消息传递型并行编程接口规范，它定义了一系列在进程间进行消息传输和同步的编程接口。换句话说，MPI 不是一种编程语言，而是编程接口规范。目前 MPI 支持 C/C++ 语言和 FORTRAN 语言，对于 C/C++ 语言来说，MPI 对应一系列接口函数；对于 FORTRAN 语言来说，MPI 对应的则是一系列子程序（subroutine）。

在没有特殊说明的情况下，本章后面部分的内容都针对 C 语言。后面介绍的绝大多数编程接口和数据结构都有对应的 FORTRAN 语言版本，感兴趣的读者可以在 MPI 编程参考手册中找到相应的内容。

目前存在多种按照 MPI 规范实现的 MPI 程序库，主要有开源软件和厂商软件两类。经典的开源 MPI 程序库有 MPICH、OpenMPI、LAM 等；一些处理器和计算机厂商推出了针对其处理器和系统的 MPI 程序库，如 Intel MPI、IBM MPI 等。

目前，MPI 被广泛应用于从小规模集群系统到超级计算机的各种消息传递型系统中，已成为高性能计算机系统中最主流的并行编程接口。历经数十年发展以来，虽然 MPI 接口规范不断尝试加入一些新的特性，版本也从最初的 MPI v1 升级到 MPI v5，但其本质仍然是以进程间消息发送 / 接收为核心。随着计算机体系结构的不断演变，以及并行程序规模不断增大，MPI 的一些缺点也逐渐显露出来，突出的一点是：较为原始的底层消息接口导致可编程性不好，降低了并行程序开发的生产率（productivity）。因此，业内也出现了一些新的编程模型和编程语言，如 Charm++ 和本章后面将要介绍的 PGAS，但这些语言都处于相对小众的地位，难以动摇 MPI 在高性能计算机中的主流编程接口地位。形成这种局面的主要原因有两个：一是 MPI 编程接口虽然“原始”“可编程性差”，但它具有相对优越的性能，以及很好的灵活性和包容性，例如，MPI 自身不支持多线程编程，也不支持 GPU 异构编程，但它可以很好地与这些新型编程接口融合，进而可以方便地实现多线程或调用 GPU 核函数；第二个原因是使用习惯和历史积累，如今已经有大量基于 MPI 开发的软件和算法库在应用中，特别是很多科学 / 工程计算软件已经在各自应用领域得到广泛应用，进而形成了 MPI 的稳固地位。

3.1.2　MPI 的并行模式

MPI 程序执行的并行模式被称为**单程序多数据**（**single program multiple data, SPMD**）。如图 3-1 所示，一个 MPI 程序在被执行时将创建多个 MPI 进程，每个进程执行相同的程序，即“单程序”；数据则被分成多块，分别由不同的进程处理，从而实现多个进程的并行计算，即“多数据”；这些进程被分派在各个节点上执行，并通过调用 MPI 接口函数实现进程间的消息发送 / 接收，以及进程间的同步；程序执行时创建的 MPI 进程个数，以及每个节点上运行的进程个数，与可用的计算资源，即节点个数和处理器个数之间没有强制性关联，很多情形下，人们习惯按照每个处理器（核）一个进程的方式分配计算资源和创建进程，但这并不是强制性的，例如，可以给每个处理器（核）分派多个进程，或者节点中的进程个数少于处理器（核）数。

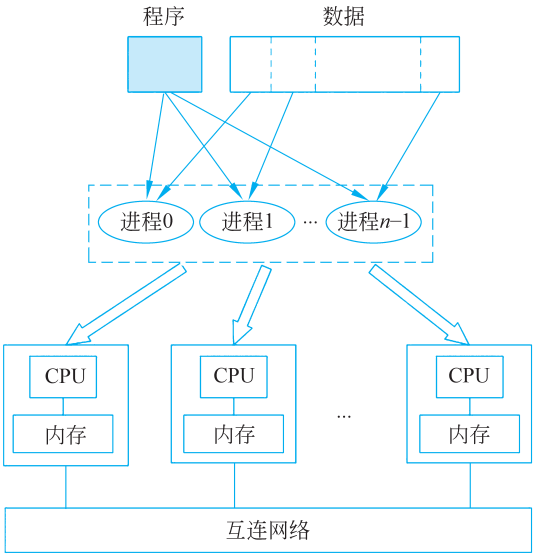


图 3-1　MPI 的 SPMD 并行模式

需要说明的是，上述多个 MPI 进程是在 MPI 程序启动时被创建的，而进程个数则由程序的启动参数指定。也就是说，无须在程序中显式调用接口函数创建 MPI 进程，程序在编写时也并不知道会以多少个进程执行，实际上，每次执行程序时可以指定不同的进程

个数。

如上所述，MPI 程序的并行单位是进程，其消息发送 / 接收都是在进程之间进行的，因此，可以认为 MPI 是一种多进程编程接口。那么，MPI 和第 2 章介绍的多线程编程是何关系？简略地说，MPI 不排斥多线程，但它并不提供与线程相关的编程接口。如果要在 MPI 程序中实现多线程，则可以通过 OpenMP、Pthreads 等多线程接口在 MPI 进程中实现。在 3.6 节编者将专门讨论这一问题。

3.1.3 一个简单的 MPI 程序

为了帮助读者理解 MPI 程序的基本构成，本节首先介绍 MPI 版的 Hello World 程序，见代码清单 3-1。

代码清单 3-1 MPI 版本的 Hello World 程序。

```
1    #include <stdio.h>
2    #include "mpi.h"
3
4    int main(int argc, char **argv)
5    {
6       int rank, numProcs;
7
8       MPI_Init(&argc,&argv);
9       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10      MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
11      printf("Hello World! I am rank %d, there are %d processes\n",
12               rank, numProcs );
13      MPI_Finalize( );
14      return 0;
15   }
```

编写 MPI 程序需要引用头文件 mpi.h（第 2 行），该文件定义了所有 MPI 接口函数的原型、数据类型及常量定义等。所有的 MPI 接口函数都以字符串“MPI_”开头，MPI 程序在开始时需要调用 MPI_Init( ) 函数（第 8 行），该函数是通知 MPI 环境进行初始化操作，包括为进程指定全局 rank、分配消息缓冲区等。需要注意的是，程序在调用 MPI_Init( ) 之前不能调用其他 MPI 接口函数。

在程序的最后，需要调用 MPI_Finalize( ) 函数（第 13 行），该函数是通知 MPI 环境回收分配给进程的所有资源。在调用 MPI_Finalize( ) 函数之后，程序将无法再调用其他 MPI 接口函数。

MPI_Init( ) 和 MPI_Finalize( ) 函数的定义如下。

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

在 MPI_Init( ) 和 MPI_Finalize( ) 之间，程序可以进行计算，也可以调用 MPI 接口函数实现进程间通信 / 同步。

在执行上述程序时，每个进程都会输出一条 Hello World 信息，该信息还包含进程号

（rank）和进程个数，这两个数值通过调用函数 MPI_Comm_rank( ) 和 MPI_Comm_size( ) 获得，后面会详细介绍这两个函数的功能。

假设以 4 个进程运行该程序，则执行结果如下。

```
Hello World! I am rank 0, there are 4 processes
Hello World! I am rank 1, there are 4 processes
Hello World! I am rank 2, there are 4 processes
Hello World! I am rank 3, there are 4 processes
```

通过程序执行结果可以看出，按照 MPI 的 SPMD 并行模式，该程序的每个进程都执行相同的程序，并通过 printf( ) 语句显示一条信息。事实上，由于多个进程并行执行，其显示信息的顺序并不确定。

3.1.4　MPI 基本环境

1. 进程管理

在 MPI 环境中，MPI 进程的创建、启动，以及运行时的管理都是通过进程管理器（process manager, PM）来完成的。进程管理器就是 MPI 环境与操作系统的接口。不同的 MPI 程序库可能使用不同的进程管理器，例如，MPICH 的默认进程管理器是 hydra。

MPI 程序一般都是大规模并行程序，所以进程会被分布在多个节点上执行，进程管理器也需要管理节点信息，在程序运行过程中，每个节点上部署的 MPI 环境需要彼此交换信息，以协同完成计算工作。所以在 MPI 库安装之后，需要配置各节点间的 SSH（secure shell）无口令登录。MPI 环境指定节点配置文件（典型文件名为 machinefile），该文件里包含需要执行程序的节点列表。

为了便于对节点及计算任务进行管理，高性能计算机系统通常会使用作业管理系统，如 Slurm、PBS 等。本章后面将介绍如何使用作业管理系统。

2. MPI 程序编译

MPI 环境提供了针对 C、C++、FORTRAN77、FORTRAN90 程序的编译脚本，通过封装 gcc、g++、gfortran 等编译器可以实现 MPI 程序的编译和链接，如表 3-1 所示。

表 3-1　MPI 程序编译脚本

编 译 脚 本	对应编程语言
mpicc	C
mpic++	C++
mpif77	FORTRAN77
mpif90	FORTRAN90

下面给出用 mpicc 编译 C 语言程序 helloworld.c 的命令示例，该命令将生成可执行文件 helloworld。可以看出，编译命令和选项与 gcc 相同。

```
$ mpicc helloworld.c -o helloworld
```

较大规模的 MPI 程序往往包含多个源程序文件，为便于编译和维护程序，可以建立

工程文件 makefile，在工程文件中使用 mpicc 编译 MPI 程序；或者还可以编写脚本并使用 CMake 工具生成跨平台的编译工程文件。

3. MPI 程序运行

MPI 环境提供 mpiexec 命令来启动 MPI 应用程序（早期 MPI 使用 mpirun 命令），启动命令格式如下。

```
mpiexec -f machinefile -n <num> <executable>
```

程序启动命令的参数说明如下。

（1）-f machinefile：该参数用于指定节点配置文件。machinefile 即为节点配置文件的文件名，该文件是一个格式简单的文本文件，其中包含执行程序使用的节点清单。图 3-2 给出了一个简单的节点配置文件样例，该文件的节点清单中包含 4 个节点，用节点名或 IP 地址标识。大规模集群系统通常使用作业管理系统统一管理和分配计算节点，每次执行程序时，分配到的节点可能不同，在这种情形下，该文件通常在作业脚本中动态生成。

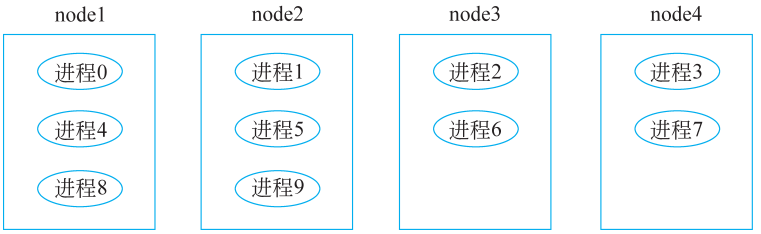
```
node1
node2
node3
node4
```

图 3-2　节点配置文件 machinefile 样例

（2）-n num：该参数用于指定希望启动的进程个数，由整数 num 给出。

（3）executable：指定要执行的程序文件名，可以是 MPI 程序，也可以是一个普通的非 MPI 程序。无论何种，MPI 环境都将通过进程管理器为其创建指定个数的进程，每个进程都执行该程序，但如果是非 MPI 程序，则进程间没有 MPI 通信。

在使用 mpiexec/mpirun 命令在多个节点上启动应用程序时，MPI 可以根据需要指定程序运行的进程个数，那么，这些进程是如何在多个节点上分派的呢？在默认情况下，MPI 将按照顺序轮转分派的方式在各个节点启动进程。图 3-3 给出了在 4 个节点上启动 10 个 MPI 进程的示例，假设使用前面所述的节点配置文件（node1~node4），本次启动 10 个进程（-n 10），这些进程将按顺序被分派在这 4 个节点上，直到所有进程被分派完毕。



mpiexel -f machinefile -n 10 helloworld

图 3-3　10 个 MPI 进程在 4 个节点上的分派

还需说明的一点是，MPI 要求执行程序的每个节点都能够访问到该程序文件。由于大规模集群系统通常使用共享外存系统，用户的程序和数据都被存放在共享外存系统中，可以被所有节点访问到，所以上述条件是满足的。但如果是没有共享外存系统的小规模集群系统，则需要配置节点间的共享文件系统（如利用 NFS），把用户程序和数据都存放在该共享文件系统中。

4. MPI 程序调试

MPI 集成了与 Linux 系统主流调试工具 gdb 的接口，在使用 mpiexec 命令启动并行程序时，可以添加 -gdb 参数，就可以使用 gdb 调试程序。

有些 MPI 库的进程管理工具并不支持 -gdb 参数，这种情况下可以使用一种小技巧使 gdb 装载程序。例如，编程人员可修改源程序，在 MPI_Init( ) 函数之后添加一个有条件判断的死循环函数，如代码清单 3-2 所示。当进程运行至循环体时，会停留在循环体中，用户通过查询进程号以使用 gdb 挂载欲调试的进程，再通过 gdb 修改变量 i 的值，使其跳出循环体。

代码清单 3-2　添加循环使 gdb 挂载程序示例。

```
1    #include <stdio.h>
2    #include"mpi.h"
3
4    int main(int argc, char **argv)
5    {
6       MPI_Init(&argc,&argv);
7       int i=1;
8       While (i)
9           sleep(1);
10      ...
11   }
```

3.1.5　通信子、进程组、进程号

在 MPI 编程模型中，**通信子**（**communicator**）是一个非常重要的概念。通信子是 MPI 环境管理进程及通信的基础设施，它定义了**一个可以相互间通信的进程集合**，进程间的消息传递都是在通信子中进行的。一个可供类比的例子是：程序在访问文件的时候需要使用文件句柄（handle），在进行网络通信的时候需要使用套接字（socket），而在需要进行 MPI 消息传输的时候就需要使用通信子，所有的 MPI 消息通信接口函数都将通信子作为参数之一。

提出通信子概念的背景是，MPI 程序执行时会生成多个进程，通信既可以在所有这些进程之间进行，也可以在进程子集中进行，即把进程分组，仅在组内进程间通信。为了便于管理这些通信，就需要使用进程组（group）和通信子的概念。进程组（很多时候被简称为“组”）对应程序执行时的进程全集或子集，而通信子就是组内进程间通信的基础设施。

上述描述试图用简单直观的方式帮助读者理解通信子这一概念。实际上，这里描述的通信子被称为“组内通信子”，MPI 程序的绝大多数通信都使用这种通信子。除此之外，还有一种“组间通信子”，本章后文部分将专门介绍。

在 MPI 程序启动后，MPI 环境会创建两个默认的通信子，一个是全局通信子 MPI_COMM_WORLD，它对应本次程序启动的所有进程；另一个是 MPI_COMM_SELF，该通信子只包含进程自身。

进程组（group）是与通信子关系密切的重要组成部分，定义一个通信子时，也就指定了一组共享该空间的进程组。进程组的概念侧重的是对进程的分组，将进程划分成不同

的组以完成不同的任务。通信子的概念侧重的是对通信空间的划分，通信子除了包括一组可以相互通信的进程组，还包括通信上下文、虚拟处理器拓扑等，其更体现了与通信相关的所有元素的集合。

每个进程在其所属的通信子中都有唯一的进程号（rank），进程号标识了该进程在此通信子中的序号，也可以被理解为进程在进程组中的序号。前面 Hello World 示例程序中的进程号就是进程在全局通信子 MPI_COMM_WORLD 中的进程号。在一个通信子中，进程号从 0 开始编号，为一个连续整数序列。需要强调的是，进程号仅对某个通信子有效，一个进程在不同的通信子中可能拥有不同的进程号，通过二元组 **< 通信子，进程号 >** 可以唯一地标识一个 MPI 进程。

MPI 提供两个接口函数用于获取进程和通信子的相关信息，定义如下。

```
int MPI_Comm_size( MPI_Comm comm, int &size );
int MPI_Comm_rank( MPI_Comm comm, int &rank );
```

函数 MPI_Comm_size( ) 用于获取指定通信子中所包含的进程个数，而函数 MPI_Comm_rank( ) 则用于获取进程在指定通信子中的进程号。这两个函数的第一个参数都是通信子，其类型是 MPI 定义的通信子数据类型 MPI_Comm；第二个参数是输出型参数，即期待返回的值。

如果用通信子 MPI_COMM_WORLD 作为参数调用这两个函数，就可以获取本次程序启动的进程总数和本进程的全局进程号。按照 MPI 的 SPMD 并行模式，程序运行时启动的进程个数由启动参数指定，程序在编写时并不知道它会以多少个进程被执行，而且每个进程都将执行相同的程序。为了在进程间进行任务和数据的划分，程序必须知道本次程序执行的进程个数，以及每个进程的身份（即进程号），然后根据这些信息划分数据，由各进程并行处理，这一需求可以通过上述两个函数来实现。

几乎所有的 MPI 程序在调用 MPI_Init( ) 完成初始化后，都会调用 MPI_Comm_size( ) 和 MPI_Comm_rank( ) 获取本次程序的运行信息（进程个数和自身进程号），然后根据获取到的信息进行数据的初始化和分配（例如，根据进程总数计算应由每个进程负责的数据量和起止区间），并完成进程间的工作分配（例如，根据进程号让不同的进程执行不同的函数或语句）。

3.1.6 MPI 数据类型

1. 主要的预定义数据类型

MPI 预定义了很多数据类型，主要的预定义数据类型与 C 语言数据类型的对应关系如表 3-2 所示。在程序中，使用 MPI 数据类型或者 C 语言数据类型定义变量都是被允许的。

表 3-2 MPI 常用数据类型与 C 语言数据类型的对应关系

MPI 数据类型	对应的 C 语言数据类型
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int

续表

MPI 数据类型	对应的 C 语言数据类型
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_INT8_T	Int8_t
MPI_INT16_T	Int16_t
MPI_INT32_T	Int32_t
MPI_INT64_T	Int64_t
MPI_UINT8_T	uInt8_t
MPI_UINT16_T	uInt16_t
MPI_UINT32_T	uInt32_t
MPI_UINT64_T	uInt64_t
MPI_C_BOOL	bool

2. MPI 类型匹配规则

在通信过程中，通信双方对数据类型的指定必须一致，发送方进程和接收方进程在调用通信函数时，必须指定相同的 MPI 数据类型。有一点需要特别注意的是，由于 MPI 数据类型与宿主语言数据类型有对应关系，会存在不同 MPI 数据类型对应相同宿主语言类型的情况，例如，MPI_LONG_LONG_INT 和 MPI_LONG_LONG 两种整数类型都对应的 C 语言的 usigned long long int 类型，在这种情况下，也必须使用相同的 MPI 数据类型匹配收发消息，不允许出现发送 / 接收双方分别使用 MPI_LONG_LONG_INT 和 MPI_LONG_LONG 类型的情况，即便这两者底层的实现都是 C 语言的 usigned long long int 类型。

3. 自定义数据类型

除了预定义数据类型之外，MPI 还支持自定义数据类型。一种比较典型的应用场景是：在程序中用 MPI 消息传输结构体（即 C 语言的 struct）。此时有两种可选的解决方案：一是将结构体数据当作无符号字符构成的数组以进行传输（即表 3-2 中的 MPI_UNSIGNED_CHAR 类型）；二是自定义数据类型，并在 MPI 消息中直接使用这种自定义的数据类型。

在 MPI 中，开发者需要调用接口函数来自定义数据类型。为此 MPI 提供了多种接口函数，其中最通用的是 MPI_Type_create_struct( )。该函数的定义如下。

```
int MPI_Type_create_struct(
        int count,                                  // 该数据类型中的块数
        const int array_of_blocklengths[],          // 每一块的元素个数
        const MPI_Aint array_of_displacements[], // 每一块的字节偏移
        const MPI_Datatype array_of_types[],        // 每一块中元素的数据类型
        MPI_Datatype *newtype                       // 新数据类型指针
);
```

下面通过一个简单示例展示如何在 MPI 中自定义数据类型。假设程序中定义了一个结构体类型 S，如下所示。

```
struct S {
       char rank;
       double value;
       }
```

现在将其定义为 MPI 自定义数据类型，以便使用 MPI 消息传输该结构体数据，示例程序如代码清单 3-3 所示。由于结构体类型 S 中有 2 个元素，程序第 6 行调用 MPI_Type_create_struct( ) 时的第 1 个参数值设为 2，其后三个参数分别指定了这两个元素的变量个数（blockLen[]）、字节偏移量（displacements[]）和数据类型（types[]）。需要说明的是，字节偏移量数组 displacements[] 中没有使用数值 0 和 1，而是使用了 C 语言标准宏 offsetof( ) 以返回两个元素的偏移值，这主要是考虑字节对齐因素的影响。例如，C 语言的 char 类型变量只占用 1 字节，但很多平台和编译器默认为其分配 4 字节或 8 字节内存，以便提高访存性能。因此，使用 offsetof( ) 获取各元素偏移量可以使程序具有更好的适应性。调用 MPI_Type_create_struct( ) 将返回自定义的数据类型 myType，但在正式使用该数据类型之前，需要调用 MPI_Type_commit( ) 进行提交（第 7 行）。

代码清单 3-3 自定义数据类型的程序示例。

```
1    int blockLen[] = { 1, 1 };
2    MPI_Aint displacements[]={offsetof(S,rank), offsetof(S,value)};
3    MPI_Datatype types[] = { MPI_CHAR, MPI_DOUBLE };
4    MPI_Datatype myType;
5
6    MPI_Type_create_struct(2,blockLen,displacements,types,&myType);
7    MPI_Type_commit( &myType );
```

3.1.7 MPI 通信简介

MPI 通信是 MPI 编程模型的核心部分，本小节概述性地介绍 MPI 通信的一些基本概念。

1. 通信类型

MPI 提供了多种通信类型，按照参与通信的对象区分可以将之分为两大类，分别是点对点通信和集合通信。

点对点通信（point-to-point communication）是指两个进程间的通信，一个进程向另一个进程发送消息。点对点通信根据通信模式的不同也可以分成几类，这在 3.2 节中会详细

介绍。

集合通信（collective communication）是指在一组进程间发送 / 接收消息。例如，广播就是集合通信的一种操作，即组内某个进程向组内其他所有进程发送消息。关于集合通信的详细介绍见 3.3 节。

2. MPI 消息

MPI 的消息由**消息信封**和**消息数据**组成。消息信封可以唯一标识某一条消息，发送进程和接收进程在收发某一条消息时，MPI 环境通过匹配消息信封以确保消息的正确接收。消息信封可以表示成如下三元组。

```
<src/dest, tag, comm>
```

上述三元组中，comm 是指发生该通信行为的通信子；src/dest 是消息的源进程（src）或目的进程（dest）的进程号；tag 是该消息的标识，为一非负整数，用于区分同一对进程之间可能的多条消息。

消息数据由消息内容组成，可以由如下三元组表示。

```
<buf, count, datatype>
```

上述三元组中，buf 是存放消息的缓冲区，这个缓冲区可以是发送缓冲区，也可以是接收缓冲区；count 是消息中数据的个数；datatype 是数据的类型。

3.2　点对点通信

点对点通信是 MPI 通信中最常用的模式，两个进程之间可以通过点对点通信完成消息的传递。

消息的传递涉及数据的复制和同步，为满足不同的需求，MPI 定义了多种点对点通信模式。从进程调用消息传输接口函数后是否阻塞的角度可以将点对点通信分成两类：**阻塞通信**和**非阻塞通信**。阻塞通信是指进程调用通信函数时，函数需要等待某些操作完成之后才返回；非阻塞通信则是通信函数将请求告知 MPI 环境后就返回。

为了支持更加灵活的消息通信，MPI 还定义了四种通信模式供开发者根据需要选用，包括**标准模式、缓存模式、就绪模式**和**同步模式**。这些编程模式可以与阻塞通信、非阻塞通信配合使用，这样就形成了多种点对点通信接口函数。表 3-3 给出了 MPI 提供的点对点通信接口函数一览表。

表 3-3　MPI 提供的点对点通信接口函数一览表

分　类	通信模式	发送函数	接收函数
阻塞通信	标准模式	MPI_Send( )	MPI_Recv( ) MPI_Irecv( ) MPI_Recv_init( )
	缓存模式	MPI_Bsend( )	
	就绪模式	MPI_Rsend( )	
	同步模式	MPI_Ssend( )	

续表

分类		通信模式	发送函数	接收函数
非阻塞通信	非持续	标准模式	MPI_Isend( )	MPI_Recv( ) MPI_Irecv( ) MPI_Recv_init( )
		缓存模式	MPI_Ibsend( )	
		就绪模式	MPI_Irsend( )	
		同步模式	MPI_Issend( )	
	持续	标准模式	MPI_Send_init( )	MPI_Recv( ) MPI_Irecv( ) MPI_Recv_init( )
		缓存模式	MPI_Bsend_init( )	
		就绪模式	MPI_Rsend_init( )	
		同步模式	MPI_Ssend_init( )	

从表 3-3 中可以看出，发送函数的种类较多，有 12 种发送操作，其命名采用 MPI_xxsend( ) 的形式，其中的 xx 可以是字母 B、R、S，分别表示缓存（Buffered）模式、就绪模式（Ready）、同步模式（Synchronous），还可以进一步与表示非阻塞通信的字母 I 组合（I 表示 Immediate），形成 Ib、Ir、Is，即四种模式的非阻塞通信，或添加进一步后缀“_init”，形成持续的非阻塞通信（专门用于在循环中重复发送消息）。

与发送操作相比，接收函数的种类较少。MPI 共有 3 种消息接收操作，分别是阻塞接收 MPI_Recv( )、非阻塞接收 MPI_Irecv( )，以及持续接收 MPI_Recv_init( )。需要说明的是，发送函数和接收函数之间可以灵活组合，例如，发送方使用阻塞发送函数，而接收方使用非阻塞接收函数，或者双方使用不同的通信模式，这些都是允许的。后面的几小节将详细介绍这几种通信模式。

3.2.1 标准通信模式

1. 发送和接收函数

标准通信模式（standard mode）是 MPI 点对点通信最常用的方式。标准通信模式的发送和接收都是阻塞式的。关于此处“阻塞式”的具体含义，或者说发送和接收函数何时返回等将在后文详细讨论。标准通信模式的发送函数是 MPI_Send( )，接收函数是 MPI_Recv( )。MPI_Send( ) 函数与 MPI_Recv( ) 函数的定义如下。

1）MPI_Send（）：标准通信发送函数

```
int MPI_Send(
    void *buffer,          // 发送数据缓冲区指针
    int count,             // 发送数据元素数
    MPI_Datatype type,     // 发送数据类型
    int dest,              // 接收进程号
    int tag,               // 识别该消息的标识
    MPI_COMM comm          // MPI 通信子
);
```

2）MPI_Recv（）：阻塞式接收函数

```
int MPI_Recv(
```

```
    void *buffer,          // 接收数据缓冲区指针
    int count,             // 接收数据元素数
    MPI_Datatype type,     // 接收数据类型
    int source,            // 发送进程号
    int tag,               // 识别该消息的标识
    MPI_COMM comm,         // MPI 通信子
    MPI_Status *status     // 接收操作状态指针
);
```

消息发送 / 接收函数的参数主要包含消息信封和消息数据两部分，如图 3-4 所示。消息数据包括消息缓冲区、数据个数、数据类型三部分。消息信封涉及的信息包括源进程 / 目的进程号、tag、通信子，其中，通信子用于指定一组进程，源进程号和目的进程号仅对此通信子有效；tag 是一个由用户定义的非负整数，用于唯一地识别发送 – 接收进程间的一条消息（由于 MPI 允许在同一对发送 – 接收进程间连续传输多条消息，为了区分这些消息，开发者需要给不同的消息赋予不同的 tag 值）。MPI 环境通过匹配消息信封以确保消息的正确传输。例如，发送函数启动了一次发送操作，MPI 环境会检测有没有匹配的接收操作，如果匹配成功才会启动数据传输。

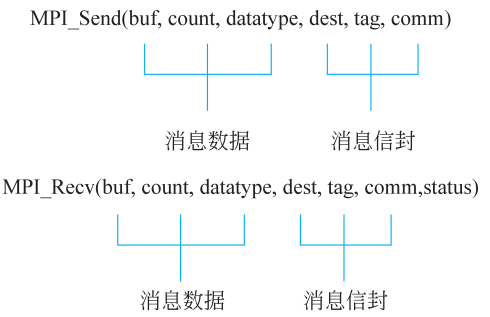


图 3-4　标准发送和接收函数的参数含义

接收进程在调用接收函数时可以指明具体的源进程号和 tag，即指定接收某个进程发送的某条特定的消息，也可以使用通配符作为消息信封的参数。例如，用 MPI_ANY_SOURCE 表示任意源进程，MPI_ANY_TAG 表示任意 tag。使用这些通配符就表示该接收操作可以接收来自任意进程发送的任意标识的消息。需要注意的是，发送操作必须指明接收进程的进程号及消息标识，不能使用通配符。

MPI_Recv( ) 函数的参数除了消息数据和消息信封之外，还有一个参数 status，用于返回接收消息的结果。该参数的类型是 MPI_Status, 是一个结构体，定义如下。

```
typedef struct MPI_Status(
        int count,             // 实际发送 / 接收的字节数
        int cancelled,         // 该通信是否被取消
        int MPI_SOURCE,        // 通信对端的进程号
        int MPI_TAG,           // 消息标识
        int MPI_ERROR
        );
```

程序可以通过 status 的相关字段来获取最终的通信结果，尤其是对使用了通配符的接收操作，可以通过 MPI_SOURCE 和 MPI_TAG 获取消息的来源和标识，并通过 count 获取实际传输的数据大小。除了直接访问 status 变量，用户也可以通过 MPI_Get_count( ) 函数从 status 中获取该次通信实际传输的数据大小。

2. 一个简单的标准模式通信程序示例

代码清单 3-4 给出了一个简单的 MPI 标准模式通信示例程序，其功能是 0 号进程接收其他进程发来的消息。程序中的 main( ) 函数在完成 MPI 初始化后，调用 MPI_Comm_rank( ) 和 MPI_Comm_size( ) 函数来获取调用进程的进程号和本次启动的进程个数，之后在第 32~35 行根据进程号分别调用不同的函数，即 0 号进程调用 ProcRecv( )，而其他进程调用 ProcSend( )。函数 ProcSend( ) 使用标准通信模式向 0 号进程发送一条消息，而函数 ProcRecv( ) 则使用标准通信模式从除 0 号进程之外的其他进程各接收一条消息。

代码清单 3-4 MPI 标准通信程序示例。

```
1   #include <stdio.h>
2   #include"mpi.h"
3   #define BUF_SIZE 10
4   int rank, numProcs, sbuf[BUF_SIZE], rbuf[BUF_SIZE];
5   MPI_Status status;
6
7   int ProcSend( )
8   {
9      /* 向 0 号进程发送消息 */
10     printf("process:%d of %d sending...\n", rank, numProcs );
11     MPI_Send( sbuf, BUF_SIZE, MPI_INT, 0, 1, MPI_COMM_WORLD );
12  }/*ProcSend( )*/
13
14  int ProcRecv( )
15  {
16     int source;
17     /* 从 0 号进程以外的每个进程接收消息 */
18     printf("process:%d of %d receiving..n", rank, numProcs );
19     for ( source = 1; source< numProcs; source++ )
20          MPI_Recv( rbuf, BUF_SIZE, MPI_INT, source, 1,
21                    MPI_COMM_WORLD, &status );
22  }/*ProcRecv( )*/
23
24  int main( int argc, char *argv[] )
25  {
26     int i;
27     MPI_Init( &argc, &argv );
28     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
29     MPI_Comm_size( MPI_COMM_WORLD, &numProcs );
30     for ( i = 0; i< BUF_SIZE; i++ )
31         sbuf[i] = rank + i;
32     if (rank == 0) //0 号进程调用 ProcRecv( ), 其他进程调用 ProcSend( )
33         ProcRecv( );
34     else
35         ProcSend( );
36     MPI_Finalize( );
37     return 0;
38  }
```

图 3-5 给出了以上示例程序在启动 4 个进程时的消息传输示意图。在图中，进程 0 从进程 1、2、3 各接收一条消息。

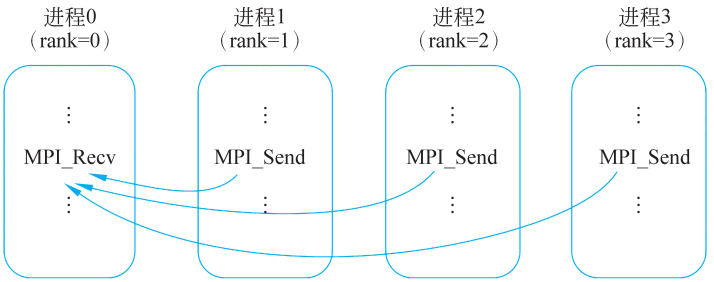


图 3-5　示例程序在 4 个进程时的通信示意图

3. 阻塞式通信的含义

前面已经提到，标准通信模式是阻塞式的，对于发送函数 MPI_Send( ) 来说，“阻塞式”的含义与通常理解的有一定差异，在这里将进行专门的讨论和介绍。

通常理解的“阻塞式”是指函数返回时操作已经完成，这对于接收函数 MPI_Recv( ) 是成立的，该函数返回意味着已经收到了消息（除非通信出错），接收缓冲区中存放着收到的消息数据；但对于发送函数 MPI_Send( ) 来说，函数返回仅意味着消息数据和信封已被妥善保存，程序可以修改发送缓冲区中的内容了。换句话说，函数返回并不意味着消息已被发送给接收进程，甚至都不能保证消息已被发出（就是说可能还存放在发送节点的本地缓冲区中）。之所以出现这种情形，与标准发送函数的以下两种处理方式有关。

处理方式一：将待发送消息复制到 MPI 环境的内部缓冲区中，然后函数返回，后续的消息发送操作由 MPI 环境自行完成。在这种方式下，发送函数可以很快返回，发送进程可以接着进行后续的计算，这有助于提升程序执行效率和性能；但这种方式也有不足之处，一是需要在内存中复制消息数据，引入了额外的开销，二是需要 MPI 环境提供足够的缓冲区，当发送的消息过长，或者消息数量很多时，可能会出现缓冲区不足的情况。

处理方式二：MPI 环境不缓存消息，直接将消息发送给接收方。这通常发生在两种情形下，一是 MPI 环境的内部缓冲区可用空间不足，无法缓存消息，因而只能等待直接把消息发送出去；二是接收方进程已经启动了接收操作，此时，跳过消息缓存步骤而直接发送消息显然是更高效的选择。

根据以上两种方式，在调用 MPI_Send( ) 函数时，可能出现以下三种情形。

（1）接收方已经启动了接收操作，则立即启动消息发送，完成消息发送后返回，如图 3-6（a）所示。

（2）接收方尚未启动接收操作，且 MPI 环境有足够的缓冲区，则将消息存入缓冲区后返回；随后接收方启动接收操作后，由 MPI 环境自行完成消息传输，如图 3-6（b）所示。

（3）接收方尚未启动接收操作，且 MPI 环境内部缓冲区不足，则一直等到接收方启动接收操作，发送消息完毕后返回，如图 3-6（c）所示。

图 3-6 描述了标准模式通信中发送进程和接收进程的三种状态。

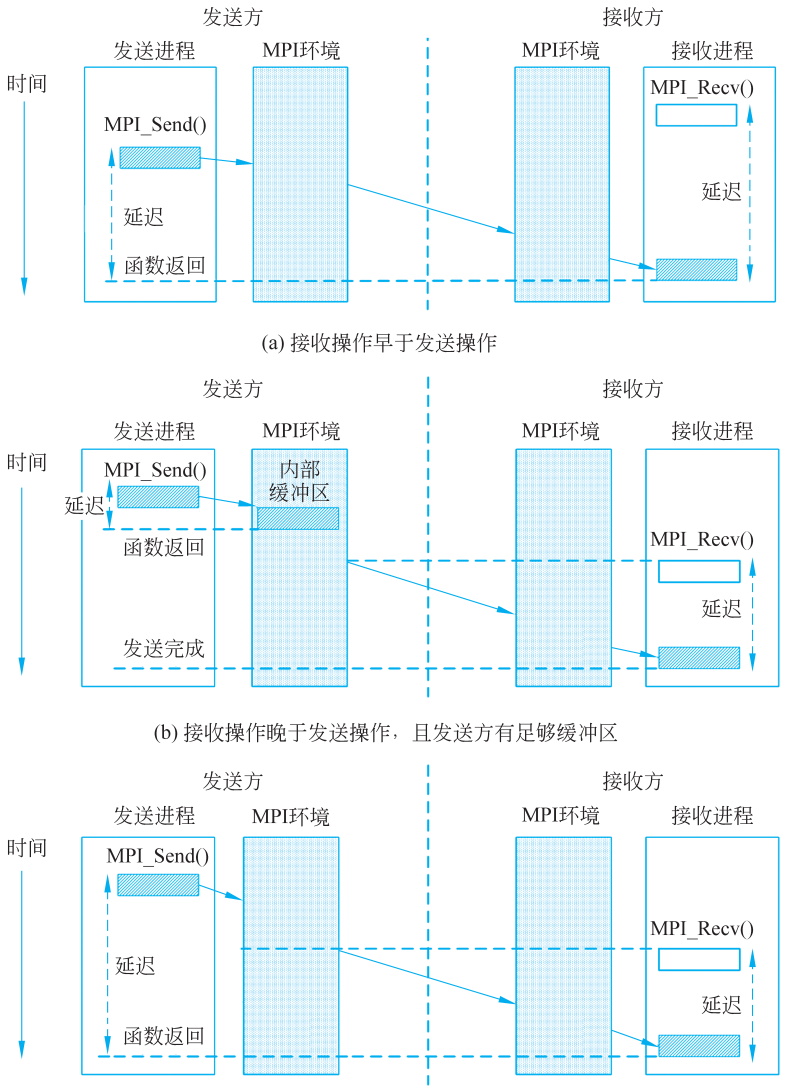


图 3-6　标准通信中发送进程和接收进程的三种情形

3.2.2　缓存通信模式

对于上一节介绍的标准通信模式，将由 MPI 环境决定是否将待发送消息缓存，如果不缓存消息，则发送进程可能需要长时间等待，也就是发送方程序调用发送函数后，须经历很长时间才能返回。为了进一步提高编程的灵活性，MPI 提供了专门的缓存通信模式（buffered mode），使用户可以对通信缓冲区进行控制。

在缓存通信模式下，发送方进程调用发送函数时，消息将被存入缓冲区，然后函数立即返回，而不会造成发送进程和接收进程之间的等待，这就解开了阻塞式通信时发送和接收之间的耦合关系。由于 MPI 环境的缓冲区容量有限，为确保有足够缓冲区存放消息，

缓存模式采用“自备缓冲区”的实现方式，即由发送进程准备足够大的缓冲区，并将其注册到 MPI 环境，在通信过程中使用，待不再使用时再卸载缓冲区并释放。

缓存模式发送函数为 MPI_Bsend( )，与标准模式发送函数相比，函数名称不同，但参数一样，具体定义如下。

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

如果用户正确申请了缓冲区，则在消息存入缓冲区后，发送函数将立即返回。如果用户缓冲区不足以缓存消息数据，则函数将返回错误。

为了支持消息缓存，用户必须自行分配消息缓冲区，并将其注册到 MPI 环境中，以供后续缓存通信使用。为此 MPI 提供了三个配套使用的函数，分别用于计算消息所需缓冲区大小、将自定义缓冲区装配到 MPI 环境、从 MPI 环境卸载自定义缓冲区。三个函数的定义如下。

```
int MPI_Pack_size(
    int count,          // 数据个数
    MPI_Datatype type,  // 数据类型
    MPI_COMM comm,      // MPI 通信子
    int *size           // 缓冲区大小 ( 字节数 )
);

int MPI_Buffer_attach(
    void *buffer,       // 缓冲区地址
    int count,          // 缓冲区大小
);

int MPI_Buffer_detach(
    void *buffer,       // 缓冲区地址
    int count,          // 缓冲区大小
);
```

用户在使用缓存发送模式时，首先使用 MPI_Pack_size( ) 计算待发送消息数据所需的缓冲区大小，该函数的前三个参数与后续调用的 MPI_Bsend( ) 函数一致，第四个参数 size 是一个输出型参数，存放所需缓冲区大小；获知需要的消息缓冲区大小后，程序就可以使用 malloc( ) 函数分配缓冲区。需要注意的是，指定分配的缓冲区大小时不能只使用 MPI_Pack_size( ) 返回的值，因为 MPI 环境在缓存消息时不仅需要缓存消息数据，也需要缓存一些附加信息，例如，消息信封信息、一些指针等。MPI 提供了一个名为 MPI_BSEND_OVERHEAD 的常量，该常量定义了一次缓存发送操作所需要的附加空间的上界值，所以在指定 malloc 分配的缓存大小时，应该使用 MPI_Pack_size( ) 返回的值加上常量 MPI_BSEND_OVERHEAD 的值，以确保为缓存通信提供足够大小的缓冲区。此外，如果用户需要连续进行多次缓存模式发送，例如连续发送 n 次，那么分配的总的缓存大小应该是对每次需要发送的数据进行 MPI_Pack_size( ) 得出数据所需要的空间大小后再累加，最后加上 n 倍的常量 MPI_BSEND_OVERHEAD。

MPI_Buffer_attach( ) 用于将用户分配的缓冲区注册到 MPI 环境，使 MPI 环境装配该缓冲区。该函数第一个参数 buffer 是用户通过 malloc( ) 函数分配的缓冲区指针，第二个参

数同样是 MPI_Pack_size( ) 计算得出的空间大小与常量 MPI_BSEND_OVERHEAD 之和。

在将缓冲区提交给 MPI 环境后，用户即可使用 MPI_Bsend( ) 以缓存模式发送消息。待程序不再使用缓存模式发送消息时，可调用 MPI_Buffer_detach( ) 函数将缓冲区从 MPI 环境中卸载。MPI_Buffer_detach( ) 函数是一个阻塞操作，会一直等到该缓冲区中的消息发送完成后才返回。MPI_Buffer_detach( ) 返回后，用户可以使用 MPI_Buffer_attach( ) 再次装载该缓冲区，也可以将该缓冲区释放。

代码清单 3-5 给出了一段注册和卸载缓冲区的示例代码。这段程序分配固定大小的缓冲区并注册到 MPI 环境，在开发者明确知道程序发送消息小于该长度时，可以采用这种简单的实现方式。

代码清单 3-5 缓存模式几个函数的使用示例。

```
#define BUF_SIZE 10000                  // 定义缓冲区大小
int size;
char *buf;
buf = (char *)malloc( BUF_SIZE );    // 申请缓冲区
MPI_Buffer_attach( buf, BUFFSIZE ); // 注册缓冲区

… // 此时程序可使用缓存模式发送消息，可用缓存容量 10 000 字节

MPI_Buffer_detach( &buf, &size);     // 卸载缓冲区
```

3.2.3 同步通信模式

同步通信模式（synchronous mode）是指发送方进程需要等待接收方开始接收数据之后才返回，即发送 / 接收双方达到一个确定的同步点后，发送方进程的发送函数才返回。在同步通信模式下，发送方首先向接收方发送一个消息发送请求，接收方的 MPI 环境将该请求保存下来，等待接收方进程的接收动作启动后，接收方将向发送方返回一个消息发送许可，表示已经准备好接收数据，发送方收到许可后开始发送消息，等到待发送的数据已经全部被系统缓冲区缓存并开始发送后，发送进程才从发送函数返回。也就是说，在同步模式下，发送函数返回意味着接收方已经开始接收消息，这实际上在发送进程和接收进程之间进行了一次同步。

同步通信模式的发送函数是 MPI_Ssend( ), 其定义如下。

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);
```

可以看出，MPI_Ssend( ) 函数的参数与 MPI_Send( ) 函数完全一致。

3.2.4 就绪通信模式

就绪通信模式（ready mode）要求发送方启动发送操作时，接收进程已经启动了接收操作（处于就绪状态）。这是一种比较严格的时序要求，如果得不到满足，也就是发送方启动发送操作时，接收方尚未启动接收，那么发送函数就会返回错误。

就绪通信模式的发送函数是 MPI_Rsend( )，其定义如下。

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);
```

MPI_Rsend( ) 函数的参数仍然与 MPI_Send( ) 函数完全一致。

就绪模式与之前三种通信模式最大的区别是：开发者要确保接收操作的启动早于发送操作。这种严格时序要求换来的好处是，发送函数可以立即启动消息传输，从而省去了消息的缓冲以及发送 / 接收双方的握手操作，因此提高了消息传输效率。

3.2.5　四种通信模式小结

1. 四种通信模式的特点比较

表 3-4 对 MPI 的四种通信模式进行了比较。

表 3-4　四种点对点通信模式的比较

通信模式类别	消息发送的特点
标准模式	根据接收操作是否已启动，以及 MPI 缓冲区状态决定发送行为： 如接收操作已启动，则立即启动发送操作； 如接收操作未启动，且 MPI 缓冲区够用，则缓冲消息后返回； 如接收操作未启动，且 MPI 缓冲区不足，则等待接收操作启动
缓存模式	消息存入 MPI 缓冲区后即返回，消息传输由 MPI 环境自行完成
同步模式	仅当发送操作和接收操作匹配后，发送函数才返回； 如果接收操作启动较晚，则发送函数一直等待
就绪模式	接收操作启动需早于发送操作，发送方和接收方可立即匹配并开始消息传输， 其通信效率在四种模式中最高，但对发送方和接收方有严格的时序要求

针对上述四种通信模式的特点，开发者可以根据需要选用不同的通信模式，需要说明的是，这些特点主要是针对发送方的，发送方调用不同的发送函数，其行为可能是不同的。下面讨论接收方的接收函数。

2. 统一的接收函数 MPI_Recv（ ）

四种通信模式所对应的发送函数各不相同，但无论哪种通信模式，其接收函数都是一样的，即 MPI_Recv( ) 函数。

MPI_Recv( ) 是一个阻塞操作，仅当接收进程的缓冲区中收到了期待的数据才返回。如果 MPI_Recv( ) 的调用早于发送方的发送操作，那么接收进程也将一直等待，直到收到数据后才返回。

3. 死锁的可能性

消息的发送和接收涉及多个进程，且消息需在发送方和接收方之间配对，加之接收操作和有些发送操作是阻塞式的，因此存在死锁的可能性。

图 3-7 中的程序片段给出了一个有可能出现死锁的例子。这段示例程序假设有两个进程，其中，进程 0 向进程 1 发送两条消息，消息的 tag 分别为 tag1、tag2。需要注意的是，进程 0 发送消息的顺序是 tag1、tag2，而进程 1 接收消息的顺序是 tag2、tag1。这样，进

程 0 发送的第一条消息将与进程 1 的第二条接收操作对应，第二条消息将与进程 1 的第一条接收操作对应。由于 MPI_Recv( ) 是阻塞操作，所以进程 1 只有在收到消息 tag2 后，才能从第一个接收操作 MPI_Recv( ) 返回。而进程 0 的第一条发送操作是发送 tag1 消息，并不是发送进程 1 所等待的消息 tag2。由于进程 0 的发送操作采用标准模式，如果 MPI 环境的可用缓冲区不足以缓存消息 tag1，那么进程 0 的 MPI_Send( ) 将一直等待进程 1 接收该消息，而此时进程 1 在等待接收消息 tag2，这就出现了进程 0 和进程 1 相互等待的情形，即发生了死锁。如果进程 0 发送消息时 MPI 环境的可用缓冲区足以缓存该消息，则进程 0 可以立即从第一条 MPI_Send( ) 返回，进而通过第二条 MPI_Send( ) 发送进程 1 期待的消息 tag2, 进程 1 成功接收消息 tag2 后，通过第二条 MPI_Send( ) 接收消息 tag1，这种情况下死锁将不会发生。

```
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
if ( rank == 0 )
{
   MPI_Send( buf1, count, MPI_REAL, 1, tag1, MPI_COMM_WORLD );
   MPI_Send( buf2, count, MPI_REAL, 1, tag2, MPI_COMM_WORLD );
}
else
{
   MPI_Recv( buf1, count, MPI_REAL, 0, tag2, MPI_COMM_WORLD, &status );
   MPI_Recv( buf2, count, MPI_REAL, 0, tag1, MPI_COMM_WORLD, &status );
}
```

图 3-7　可能死锁的 MPI 程序示例

该示例程序可能出现死锁的主要原因是：程序在调用发送 / 接收函数时没有严格地按照消息的顺序匹配发送函数和接收函数的调用，导致死锁的发生。另外还存在其他通信死锁的情形，例如，多个进程间彼此发送消息出现循环等待。由于死锁的发生存在不确定性，这将增大程序调试的难度，需要开发者在使用 MPI 通信时给予充分重视。

4. 缓冲区的使用

MPI 定义了三种缓冲区，分别如下。

（1）应用缓冲区：在应用程序中定义的消息缓冲区。

（2）系统缓冲区：MPI 环境为通信所准备的存储空间。

（3）用户向系统注册的缓冲区：用户使用缓存通信模式时，在程序中显式申请的存储空间，然后注册到 MPI 环境中供通信所用。

在缓存模式下，由用户程序申请缓冲区并将之提交给 MPI 环境，这种情形就是使用的上述第三种缓冲区；在标准模式下，由 MPI 环境提供系统缓冲区，如果系统缓冲区充足，则缓冲待发送消息后进程可以返回，如果缓冲区不足，发送进程将等待直到通信操作完成后才返回。所以，从缓冲区的使用角度来看，标准模式介于缓存模式与同步模式之间，其与缓存模式的主要区别是缓冲区的提供者不同。

3.2.6　组合发送接收

MPI 提供了一种组合了发送和接收的通信操作 MPI_Sendrecv( )，将发送一条消息并接收一条消息合并到了一个接口函数。组合发送与接收在语义上等同于分别调用了一个发

送操作和接收操作，但相比于分别调用一条发送和接收操作来说，MPI 环境会优化发送操作和接收操作的执行顺序，有效地避免不合理的通信顺序，从而在一定程度上避免死锁的产生。

MPI_Sendrecv( ) 函数的定义如下。

```
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);
```

组合发送接收操作的发送缓冲区和接收缓冲区必须是分开的，发送操作和接收操作必须是同一个通信子内的操作。组合发送接收操作不是对称的，一个 MPI_Sendrecv( ) 调用不一定需要与另一个 MPI_Sendrecv( ) 匹配，MPI_Sendrecv( ) 发送的消息可以由一个普通的接收操作（如 MPI_Recv ）接收，也可以从一个普通的发送操作（如 MPI_Send ）接收消息。另外需要说明的是，dest 参数用于指定消息的接收方，source 参数用于指定消息的发送方，也就是说，当进程调用该函数时，与它通信的进程可以是两个不同的进程（一个接收，另一个发送），也可以是同一个进程（dest==source ）。

3.2.7　非阻塞通信

在进程间传输消息会产生较高的时延。除了互连网络引入的延迟外，这种时延还常涉及发送节点和接收节点的 MPI 环境之间的握手。对于阻塞式消息传输来说，在调用接收函数和有些发送函数时，调用进程将被阻塞，此时处理器处于空闲状态，因而这会降低程序的整体并行效率，进而影响程序性能。

“重叠通信和计算”是解决上述问题的常用方法，即当进程执行阻塞式发送或接收操作时，让通信在后台进行，在此期间，程序继续执行并进行计算，从而提高程序的并行效率和性能。重叠通信和计算的实现方法一般有两种，一种采用多线程机制，当一个线程由于通信而被阻塞时，系统将调度其他线程执行，从而使处理器不至于处于空闲状态。然而，如本章开头所述，MPI 自身并不支持多线程，因此，本小节介绍的是 MPI 支持的另一种重叠通信和计算的机制——非阻塞式通信。

MPI 的非阻塞通信机制主要被用于实现通信和计算的重叠，其发送和接收过程如图 3-8 所示。发送进程调用非阻塞发送函数时，在启动该发送操作后就返回，不会等待发送操作完成。同样，接收进程调用非阻塞接收函数时，在启动接收操作后就返回，也不会等待接收操作完成。在调用非阻塞通信函数后，消息传输过程由 MPI 环境在后台完成，进程可以继续进行计算，从而实现通信和计算的重叠。由于在非阻塞通信机制中，发送 / 接收函数返回并不意味着消息传输已经完成，为了查询通信是否完成或等待通信完成，MPI 提供了一系列通信测试函数。程序在调用非阻塞通信函数后将先执行计算语句，然后调用这些函数，检测通信状态或等待通信完成。

非阻塞发送操作也有四种模式，它们与阻塞发送的四种模式一一对应。四种模式的接口函数分别是标准模式 MPI_Isend( )、缓存模式 MPI_Ibsend( )、同步模式 MPI_Issend( ) 和就绪模式 MPI_Irsend( )。接口函数名称中的前缀 I 表示 Immediate，意指立即返回。