MPI 程序设计

MPI（Message Passing Interface，消息传递接口）是一种消息传递编程模型。消息传递指用户必须通过显式地发送和接收消息来实现处理器间的数据交换。在这种并行编程中，每个控制流均有自己独立的地址空间，不同的控制流之间不能直接访问彼此的地址空间，必须通过显式的消息传递来实现。这种编程方式是大规模并行处理机（MPP）和机群（Cluster）采用的主要编程方式。由于消息传递程序设计要求用户很好地分解问题，组织不同控制流间的数据交换，并行计算粒度大，特别适合于大规模可扩展并行算法。

MPI 是基于进程的并行环境。进程拥有独立的虚拟地址空间和处理器调度，并且执行相互独立。MPI 设计为支持通过网络连接的机群系统，且通过消息传递来实现通信，消息传递是 MPI 的最基本特色。

1.1 MPI 的含义及功能

作为缩写的 MPI 具有两种含义：

- 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，并成为这种编程模型的代表和事实上的标准。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。
- MPI 是一个库，而不是一门语言。MPI 库共用三百多个函数调用，可以被 Fortran 77/90 和 C/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。目前 MPI 最新的标准是 3.0 版。MPI 只规定了标准并没有给出实现，目前主要的实现有 OpenMPI、Mvapich 和 MPICH，MPICH 相对比较稳定，而 OpenMPI 性能较好，Mvapich 则主要是为了 Infiniband 而设计。

1.1.1 MPI 功能

MPI 主要用于分布式存储的并行机，包括所有主流并行计算机。但是 MPI 也可以用于共享存储的并行机，如多核微处理器。编程实践证明 MPI 的可扩展性非常好，其应用范围从几个机器的

小集群到工业应用的上万节点的工业级集群。MPI 已在 MS Windows 上、所有主要的 Unix/Linux 工作站上和所有主流的并行机上得到实现。使用 MPI 作消息传递的 C 或 Fortran 并行程序可不加改变地运行在使用这些操作系统的工作站、以及各种并行机上。

MPI 既可用于功能分解，也可以用于数据分解，是一个比较通用的基于 CPU 的并行编程模式，MPI3.0 加入了异构并行计算的内容（MPI 调用可以传入指向加速器存储器地址的指针）。

MPI 支持 SPMD 编程模式，在 MPI 程序中定义的变量，只要不是在进程 id 块内定义，它在每个进程内都有一份拷贝。在运行时，由于具体的运行进度不一致，使得各个进程存储空间内的变量值可能不一致，在必要的时候要使用存储器栅栏来保证存储器的一致性，这点要特别注意。

1.1.2 错误处理宏

目前 MPI 实现提供了 C/C++/Fortran 语言的接口，分别在 mpi.h/mpif.h 头文件中。本文只关注于其 C 接口。除了有限的几个。MPI 中所有的函数以 MPI 开头，单词之间用下划线连接且首单词首字母大写。MPI 的每个函数的返回值都是错误码，利用它和 MPI_Error_string(int, char*, int*)可以获得错误的字符串描述。为了方便的检查错误，本节在头文件 MPIUtil.h 中设计了一个宏以检测 MPI 函数的返回值，代码如下所示。

```
   #pragma once


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


   #define checkMPIError(errno)  {\
      if(MPI_SUCCESS != errno){\
         char MPIErrorString[8096];\
         int rl;\
         MPI_Error_string(errno, MPIErrorString, &rl);\
      \
         printf("file = %s, line = %d, error = %s\n", __FILE__, __LINE__,
MPIErrorString);\
      }\
```

```
}
```

1.2 基本的 MPI 函数

本节主要介绍一些启动、结束并行环境、获取体系处理器及进程信息的函数、以及如何运行 MPI 程序。

1.2.1 启动结束 MPI 并行环境

函数 MPI_Init 启动 MPI 并行环境，它意味着并行代码的开始。在 MPI_Init 之前除了 MPI_Initated 不能调用任何 MPI 函数。而 MPI_Finalize 函数结束 MPI 并行环境，此后除了主进程外再无进程运行，在两者之间的代码区都由多个进程并行执行，在 MPI_Finalize 之后也不能调用任何 MPI 函数。这两个函数的原型如下：

```
#include <mpi.h>
int MPI_Init(int *argc, char*** argv);
int MPI_Finalize(void);
```

MPI_Init 有两个参数，类似于标准 C 中 main 函数的两个参数。它和 main 参数的差别在于，指针的级数更高，这是因为 MPI_Init 处理后会去掉 MPI 需要的参数，这样其它参数可直接交给 main 处理，这意味着编程时无须考虑 MPI 的命令行参数。

MPI_Initiated()函数查询 MPI_Init 是否已经调用过，其原型如下所示：

```
int MPI_Initiated(int *flag);
```

如果已经调用过 MPI_Init 则返回 flag = true, 否则返回 flag = false。这是唯一一个可以在 MPI_Init 前面调用的函数，表面上看来这个函数应当很有用，但是实际上几乎不用。

软件开发人员显式的调用 MPI_Abort 函数终止一个通信子的进程且返回一个错误码给 MPI 环境。其原型如下：

```
int MPI_Abort(MPI_Comm comm, int err);
```

comm 参数为通信域，err 参数为错误退出码，一般 0 为正常退出，1 为非正常退出，其值最好和 MPI 规定的错误码一致。

1.2.2 获取 MPI 进程相关信息

MPI 将进程组织成通信子，通信子可以看成一个通信集合，只有在这个集合中的进程之间才能通信。一个通信子是一个进程组和一个上下文的组合，上下文用于区分不同的通信子。在执行函数 MPI_Init 之后，一个 MPI 程序的所有进程形成一个缺省的组，这个组的通信子即 MPI_COMM_WORLD，该参数是 MPI 通信操作函数中必不可少的，用于限定参加通信的进程的范围。

在编写 MPI 程序时，常需要知道以下两个问题的答案：

- 任务由多少个进程来进行并行计算？
- 任务由多少个进程来进行并行计算？

MPI 提供了下列函数来回答这些问题：

- 函数 MPI_Comm_size 取得通信子内的进程数，其原型如下：

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

comm 参数指定通信域，一般是 MPI_COMM_WORLD；size 参数返回通信域内进程个数。

- 函数 MPI_Comm_rank 取得当前进程索引，其原型如下：

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

comm 参数表示通信子，rank 参数返回进程索引，进程索引从 0 开始到 size-1 结束，典型的 C 风格。可由 rank 唯一的标识通信子内的一个进程。

通常利用前两个函数以分别处理各进程的任务，因为索引唯一的标识了在某个通信子内的进程，这样就可以采用如果是 0 号进程，就做什么，如果是 1 号进程就做什么等等。

函数 MPI_Get_processor_name 取得当前处理器的名称，其原型如下：

```
MPI_Get_processor_name(char* name, int* size);
```

name 参数保存返回的处理器名称，size 参数返回名称的长度。

下面的例子简单的展示了如何获得总进程数、进程索引和进程使用的处理器名称。

```
#include <stdio.h>
#include "MPIUtil.h"
int main(int argc, char** argv) {
   int myrank, nprocs;
```

```
      char name[2048];
      int nameLen;

      checkMPIError(MPI_Init(&argc, &argv));
      checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &nprocs));
      checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &myrank));
      checkMPIError(MPI_Get_processor_name(name, &nameLen));

      printf("I am process %d, total %d processes, i use processor %s\n", myrank,
nprocs, name);

      checkMPIError(MPI_Finalize());
      return 0;
   }
```

1.2.3 计时函数

MPI 提供了函数 MPI_Wtick 和 MPI_Wtime 来获得计时的精度和计时，其原型如下：

```
   double MPI_Wtick();
   double MPI_Wtime();
```

MPI_Wtick 返回用作硬件计时的两次脉冲间的间隔，以秒为单位。MPI_Wtime 取得自过去某一时刻起到调用时的时间间隔。一般在开始运算时由某个进程执行 MPI_Wtime，结束时由某个进程执行 MPI_Wtime，使用任何机制给多控制流程序计时时，都要考虑同步的问题。

1.2.4 获取 MPI 版本

函数 MPI_Get_version 返回当时使用的 MPI 环境的版本号，其原型如下：

```
   int MPI_Get_version(int *major, int *minor);
```

参数 major 为主版本号，参数 minor 为次版本号。使用版本号可以确定 MPI 实现是否支持某些

特性。

1.3 运行 MPI 程序

MPI 程序的执行步骤一般为：

1. 编译以得到 MPI 可执行程序（若在同构的系统上，只需编译一次；若系统异构，则要在每一个异构系统上都对 MPI 源程序进行编译），
2. 将可执行程序拷贝到各个节点机上（业界一个常见的做法是使用网络文件系统，将可执行程序及其所需要的数据都保存在网络文件系统中），
3. 通过 mpirun 命令并行执行 MPI 程序，格式：mpirun －np N program

   其中：N 表示同时运行的进程数；program 指可执行 MPI 程序名。例如：

```
mpirun -np 6 cpi

mpirun -np 4 hello
```

   如上面的例子的运行命令为：mpirun --np 2 a.out。

   另一种方式：mpirun －machinefile file -np N program

   file 为配置文件，其格式为：

   机器名

   机器名

   机器名

   例如：

   node0

   node1

   node2

   node3

OpenMPI 还支持--byslot 命令选项，此时其 machinefile 格式为：机器名或 IP slots=x，x 表示在这个节点上启动的进程数。

无论是 OpenMPI、MPICH，还是 MVAPICH，都是按照 machinefile 中的顺序生成进程 id，通过进程号和 machinefile 即可确定进程在运行在那个节点上。

1.4 MPI 数据类型及转换

考虑到需要在不同的计算机系统之间传输数据，数据的表达就非常重要，且不同编译系统有不同的数据表示格式。MPI 预先定义一些基本数据类型，在实现过程中以这些基本数据类型为桥梁进行转换。MPI 定义了 MPI_Datatype 类型以统一表示数据类型，这样便可以使用数据类型 MPI_Datatype 作为参数，而实际传入更具体的类型。

在消息传递时，存在一个类型匹配的问题，通常要求发送和接收的数据类型完全一致（虽然依据 MPI 标准，即使数据类型不一致也有可能是可以的）。

MPI 中所有函数使用的数据类型均为 MPI_Datatype，具体可分为基本数据类型和自定义数据类型。

1.4.1 基本数据类型

基本数据类型是 MPI 原生支持的，和 C/Fortran 数据类型对应。在 MPI 程序中，除了基本数据类型，其它的任何扩展数据类型都需要通过调用函数基于已有的数据类型建立。为了能够在网络间传输字节，MPI 定义了 MPI_BYTE 类型。表 1-1 列出了 MPI 的基本数据类型和对应的 C 数据类型。

表1-1　　MPI 数据类型

MPI 数据类型	对应 C 数据类型
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
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
MPI_PACKED	无
MPI_LONG_LONG_INT	long long int

从表格中看，两者的对应关系一目了然。和 C 中的数据类型类似，MPI 的数据类型也可以转换，只是增加了许多限制。

和 C 类似，MPI 也没有规定不同类型数据的长度，虽然从可移植和跨机器考量，固定大小才是更可取的做法。

1.4.2 自定义数据类型

实践中，经常需要传输复杂的数据结构，如 C 中的 struct，如果只使用 MPI 的基本数据类型，那么在传输前需要进行转换，一方面这会占用时间，另一方面也占用存储空间，再者使用也不简洁方便。MPI 提供了自定义数据类型的能力，其基本步骤分为：

- 依据参数建立数据类型，常用的模式有，相同的数据类型重复多次，按一定的间隔重复，不同的数据类型等。
- 提交数据类型，数据类型只有提交后才能被 MPI 使用。
- 销毁。

1.4.3 建立数据类型

函数：

```
   int MPI_Type_contiguous(int count, MPI_Datatype old_type, MPI_Datatype
*new_type);
```

将 count 个 old_type 组合成一个新的数据类型 new_type。此函数使用相当简单，可用于将所有分量数据类型相同的结构体组成一个 MPI 可处理的数据类型。

函数 MPI_Type_vector 按步长取出相邻的多个数据以组成新的类型。其原型如下：

```
   int MPI_Type_vector(int count, int blocklen, int stride, MPI_Datatype old_type,
MPI_Datatype *newtype);
```

其中，count 表示重复的次数，blocklen 表示相邻的 old_type 数目，stride 表示间隔的步长。函数的含义是取 count 个重复的组件，每个组件由 stride 个 old_type 中取前 blocklen 个组成。如 count=2, blocklen=3, stride=4，old_type=MPI_INT 表示从每 4 个 int 中取出前 3 个，然后重复 2 次。

函数 MPI_Type_indexed 依据重复次数、偏移量将多个相同类型的数据组成一个新类型，其原型如下：

```
   int MPI_Type_indexed(int count, int blocklens[], int indices[],  MPI_Datatype
old_type, MPI_Datatype *newtype);
```

其中，count 表示 blocklens 和 indices 的长度，blocklens 每维大小表示 old_type 重复的次数，indices 每个元素表示对应的 blocklens 中第一个元素的起始地址。

假设 old_type 为 MPI_INT, count=2, blocklens={3, 2}, indices={0, 5}，此时建立的 newtype 表示从首地址开始连续的 3 个 int，然后忽略其后的 2 个 int，再加上紧接着的 2 个 int，即每 7 个 int 中，忽略第 3 第 4 个 int，其余的 int 组成的数据。

和 MPI_Type_indexed 类似，但更通用的一个函数是 MPI_Type_create_struct，其原型如下：

```
   int MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Aint
array_of_displacements[], MPI_Datatype array_of_types[], MPI_Datatype
*newtype);
```

其功能和 MPI_Type_indexed 不同的是，数据类型可以不相同。

1.4.4 获取数据类型的尺寸或地址

和 C 不同，MPI 基本和自定义数据类型的大小不能通过 sizeof 运算符得到，要通过函数 MPI_Type_size，其原型如下：

```
   int MPI_Type_size(MPI_Datatype datatype, int *size);
```

函数 MPI_Get_address 返回某个位置的绝对地址，其原型如下：

```
   int MPI_Get_address(const void *location, MPI_Aint *addr);
```

1.4.5 提交或销毁

建立后的类型只有在提交后才能使用，这由函数

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

负责，个人看来，这是 MPI 设计中不好的地方，类型应该在建立后就立刻可用，相信几乎所有的此函数调用都是在类型建立后立即出现。而且很多开发人员会忘记调用此函数。

类型使用完后，需要调用函数

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

释放其占用的空间。

1.5 消息传递

消息传递是 MPI 的核心内容，MPI 提供了许多功能强大的消息传递例程，如缓冲发送，阻塞发送，异步发送等等。在消息传递时，MPI 基本数据类型相应于宿主语言的基本数据类型。

MPI 的消息传递有阻塞和非阻塞之分。 阻塞是指一个进程须等待操作完成才返回，返回后用户可以重新使用调用中所占用的资源。非阻塞是指一个进程不必等待操作完成便可返回，但这并不意味着所占用的资源可立即被重用。

由于基于 GPU 的异构系统的广泛使用，MPI3.0 允许数据传输时传入的指针指向 GPU 存储器。 OpenMPI 和 Mvapich 已经提供了支持。通过使用库而不是程序员手动（将显存上的数据传回内存，然后再通过 MPI 传送出去）操作，能够获得更好的性能和可编程性。而 NVIDIA CUDA 引入的 GPUDirect RDMA 则从硬件的层次支持数据从显存通过 PCI-E 发送给其它网络节点，这节省了多次内存复制的开销。

1.5.1 点对点通信

点对点是指一个进程直接将消息发送给另一个进程，和对应的接收进程接收发送进程的数据。

如果发送操作的目的进程或接收操作的发送进程为 MPI_PROC_NULL，那么操作就不会执行，应用经常利用这一特性简化代码。

1. 阻塞发送接收

阻塞意味着如果消息没有发送，那么调用将会挂起 CPU，处于等待的状态。

函数 MPI_Send 阻塞发送消息，其原型如下：

```
   int MPI_Send(void* buf, int count, MPI_Datetype dataType, int recverId, int
tag, MPI_Comm comm);
```

buf 参数是发送内容数据的首地址指针，由于类型是 void*，故可以发送所有类型的数据；count 参数是发送的数据个数，独立于具体的数据类型；数据类型由 dataType 参数指定，不但可以是 MPI 内置的类型，也可以是自定义的类型；recverId 参数是接收进程的索引；tag 是此次通信使用的标签，必须和对应的接收语句相同；comm 是本次通信的通信子。

MPI_Recv 接收某个进程发送给当前进程的数据，其原型如下：

```
   int MPI_Recv(void* buf, int count, MPI_Datetype dataType, int recverId, int
tag, MPI_Comm comm, MPI_Status *status);
```

其参数和 MPI_Send 基本一致但多了一个 MPI_Status 且第四个参数为发送进程的 id，MPI_Status 参数放在 MPI_Recv 的最后。MPI_Status 保存了一次通信的许多信息，比如接收到的信息数量（可由 MPI_Get_count(MPI_Statue)获得）。

如果程序无须查询返回的 MPI_Status 信息，那么形参 status 可传入实参 MPI_STATUS_IGNORE 。

一次通信的两个进程使用的标签必须一致，否则可能会出错。

将函数 MPI_Send 和 MPI_Recv 结合起来使用，形成 MPI_Sendrecv 函数和函数 MPI_Sendrecv_replace，其中参数为两者之和，意义也一致，虽然远没有 MPI_Send 和 MPI_Recv 灵活，但是由编译器自动指定发送顺序，因此不会出现因为发送顺序导致的死锁。

要注意发送消息的顺序，不正确时会导致死锁。由于很多 MPI 实现的阻塞消息发送具有缓冲区，因此在数据量比较小不足以填满缓冲区的情况下，即使死锁也不会表现出来，但是那是个潜在的、难以发现的 Bug。为了防止发送和接收的顺序不对应造成的死锁，在必要的时候使用 MPI_Sendrecv*让编译器自己决定发送顺序。

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。

- 数据：起始地址、数据个数、数据类型。由 count 个类型为 datatype 的连续数据空间组成，起始地址为 buf，不是以字节数，而是以元素的个数指定消息的长度，count 可以是零，这种情况下消息的数据部分是空的。

- 信封：源/目的、标识、通信域。MPI 标识一条消息的信息包含四个域:
  - 发送进程：由进程的 rank 值唯一标识。
  - 接收进程：发送函数参数确定。
  - 标签：发送函数参数确定，用于识别不同的消息。
  - 通信子：缺省为 MPI_COMM_WORLD。

接收的 buf 必须至少等于发送的长度，如果接收 buf 太小，将导致溢出。如果接收函数的第四个参数为 MPI_ANY_SOURCE，此时可接收任意进程发送的数据(任意消息来源)。如果发送或接收函数的标签为 MPI_ANY_TAG，则匹配任意 tag 值的消息(任意 tag 消息)。消息传送中不允许发送和接收进程相同，否则会导致死锁。成对的接收和发送函数使用的通信子必须相同。

下面的例子简单的演示了如何使用阻塞发送和接收操作。

```
#include "MPIUtil.h"


int main(int argc, char *argv[]) {
    checkMPIError(MPI_Init(&argc, &argv));


    int id, size, buf;
    checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &id));
    checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &size));


    if(0 == id){
        buf = 30;
        checkMPIError(MPI_Send(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD));
    }else if(1 == id){
        MPI_Status status;
        buf = 5;
        checkMPIError(MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status));
        printf("i am %d, buf = %d\n", id, buf);
    }
```

```
    checkMPIError(MPI_Finalize());

    return 0;
}
```

运行命令为：mpirun --np 2 a.out

2. 非阻塞发送接收

非阻塞的发送：仅当调用了有关结束该发送的语句后才能重用发送缓冲区，否则将导致错误；对于接收方，与此相同，仅当确认该接收请求已完成后才能使用。所以对于非阻塞操作，要调用等待函数 MPI_Wait()或测试函数 MPI_Test()来等待操作结束或判断该操作是否已经完成，然后再向缓冲区中写入新内容或读取新内容。

阻塞发送将发生阻塞，直到通讯完成。非阻塞可将通讯交由后台处理，通信与计算可重叠。发送语句的前缀由 MPI_改为 MPI_I。

函数 MPI_Isend 异步发送一条消息，在发送操作还未结束此函数已经返回控制权，其原型如下：

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

其参数和 MPI_Send 基本一致，但多了最后一个参数 MPI_Request。由于异步，可以在通信的同时做其它的事情。

MPI_Irecv 异步接收一条消息，其原型如下：

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

其参数和 MPI_Isend 完全一致，此时如果要使用接收到的数据一定要使用相关的函数来保证已经接收到了数据。

发送的完成：代表发送缓冲区中的数据已送出，发送缓冲区可以重用，它并不代表数据已被接收方接收（数据有可能被缓冲）；接收的完成：代表数据已经写入接收缓冲区，接收者可访问接收

缓冲区。

为了保证需要使用传输过来的数据的时候数据已经可以使用，可使用 MPI_Wait 或者 MPI_Test（参数为一次通信使用的 MPI_Request）系列函数强制处理器等待。同时 MPI 也提供了一些等待某个或多个通信完成的函数，它们的参数也做了相应的调整。

下面的代码展示了如何使用异步通信函数及同步。

```
#include "MPIUtil.h"
#include <stdlib.h>
int main(int argc, char *argv[]) {
    checkMPIError(MPI_Init(&argc, &argv));

    int id, size, buf;
    checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &id));
    checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &size));

    MPI_Request req;
    if(0 == id){
        buf = 30;
        checkMPIError(MPI_Isend(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req));

    }else if(1 == id){
        buf = 5;
        checkMPIError(MPI_Irecv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &req));
    }
    MPI_Status status;
    checkMPIError(MPI_Wait(&req, &status));
    printf("i am %d, buf = %d\n", id, buf);
    checkMPIError(MPI_Finalize());

    return 0;
```

```
}
```

运行命令为：mpirun --np 2 a.out，其中 a.out 是使用 mpicc 编译后生成的目标程序。

3. 持久发送接收

某些情况下可能要多次调用参数相同但数据不同的 MPI_Send/MPI_Recv 对，此时可以使用持久通信函数。一方面它能够减少多次启动通信的代价，另一方面也更易于编程。

MPI_Send_init()/MPI_Recv_init（）让系统准备好发送或者接收，它们的参数与 MPI_Isend 一致。只有等到真正填充了数据后软件开发人员显式的指定发送时才开始发送，且可以多次使用同一缓冲区。

采用 MPI_Start 和 MPI_Start_all 函数执行持久通信操作，其原型如下：

```
int MPI_Start(MPI_Request *req);

int MPI_Start_all(MPI_Request *req, MPI_Statue *statue);
```

此时在使用数据时，也要保证已经接收到了数据。

持久化通信的请求需要被 MPI_Request_free 释放，这和其它的通信请求不一样。

```
int MPI_Request_free(MPI_Request *req);
```

下例给出了持久通信的一个简单例子，其中的 MPI_Wait 函数将会在下一节说明。

```
#include "MPIUtil.h"


int main(int argc, char *argv[]) {
    checkMPIError(MPI_Init(&argc, &argv));


    int id, size, buf;
    checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &id));
    checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &size));


    MPI_Request req;
    if(0 == id){
```

```
          buf = 30;
          checkMPIError(MPI_Send_init(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
&req));
      }else if(1 == id){
          MPI_Status status;
          buf = 5;
          checkMPIError(MPI_Recv_init(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
&req));
      }



      MPI_Status status;
      for(int i = 0; i < 5; i++){
          checkMPIError(MPI_Start(&req));
          checkMPIError(MPI_Wait(&req, &status));
          if(0 == id){
              buf++;
          }else if(1 == id){
              printf("i am %d, buf = %d\n", id, buf);
          }
          checkMPIError(MPI_Barrier(MPI_COMM_WORLD));
      }


      checkMPIError(MPI_Request_free(&req));


      checkMPIError(MPI_Finalize());


      return 0;
   }
```

运行命令为：mpirun --np 2 a.out

4. 检测异步通信是否完成

函数 MPI_Wait 和 MPI_Test 可用于等待或查询非阻塞通信是否完成，其原型如下：

```
int MPI_Wait(MPI_Request* req, MPI_Status* stat);
int MPI_Test(MPI_Request* req, MPI_Status* stat);
```

MPI_Wait 强制进程等待非阻塞通信完成。req 参数是要等待的通信，stat 是要等待的通信运行结束后的状态。该函数会阻塞直到前面调用的通信完成后才返回，MPI_Test 含义及参数和 MPI_Wait 一致，但此函数不会阻塞，调用后立马返回。这个函数非常方便的用于异步通信和计算重叠。

5. 查询等待接收的数据量

有时可能并不能提前知道等待接收的数据大小，需要在接收前查询对方发送的数据长度，MPI_Probe 和 MPI_Iprobe 可满足这一要求，其原型如下：

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status
*status);
```

只有 source 和 tag 都匹配的消息才会被查询，通过返回的 status 信息，可以知道等待接收的数据大小。MPI_Iprobe 是 MPI_Probe 的非阻塞版本。

6. 取消通信操作

如果需要取消某些通信，可以使用 MPI_Cancel 函数。原型如下：

```
int MPI_Cancel(MPI_Request *req);
```

1.5.2 集合通信

集合通信要求通信子中的所有进程都参与通信操作，即每一个进程都需要调用该操作函数，用于实现一对多、多对一、同步等操作。

在标准 3.0 以前，集合操作只支持同步版本，3.0 标准开始支持其异步版本。对于异步版本，无论操作是否完成进程都会接着执行，只有在操作执行完成后，其语义才会得到保证。

7. 存储器栅栏

函数 MPI_Barrier/MPI_Ibarrier 执行一次存储器栅栏，其原型如下：

```
int MPI_Barrier(MPI_Comm comm);
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *req);
```

MPI_Barrier 函数同步 comm 内的所有进程，除非它们全都到达此函数的调用点，否则必须等待。这个函数经常用来协调进程的进度。

MPI_Ibarrier 函数是 MPI_Barrier 的异步版本，无论通信子内的其它进程是否执行至此调用进程会立即返回。要保证其语义，程序必须确认操作已经完成。

8. 广播

函数 MPI_Bcast/MPI_Ibcast 做一次广播，其原型如下：

```
int MPI_Bcast(void* buf, int count, MPI_Datatype dateType, int src, MPI_Comm
comm);
int MPI_Ibcast(void* buf, int count, MPI_Datatype dateType, int src, MPI_Comm
comm, MPI_Request *req);
```

MPI_Bcast 函数把数据发送给通信子 comm 内的所有进程，这称为广播。其中，广播数据存放在 buf 参数中；数据的个数由 count 定义；数据类型由 dataType 指定；要广播的数据在 src 进程中。

MPI_Ibcast 函数是 MPI_Bcast 的异步版本，无论通信子内的其它进程是否执行至此调用进程会立即返回。要保证其语义，程序必须确认操作已经完成。

9. 归约

归约（MPI_Reduce/MPI_Ireduce）就是对一系列数据施加某种操作，如求最大最小，求和，求积。其原型如下：

```
   int MPI_Reduce(void* buf, void* b, int count, MPI_Datatype dataType, MPI_Op
op, int src, MPI_Comm comm);
   int MPI_Ireduce(void* buf, void* b,int count, MPI_Datatype dataType, MPI_Op
op, int src, MPI_Comm comm, MPI_Request *req);
```

其中，buf 参数为等待归约的数据；b 参数为归约结果存放的地址指针；count 指归约的数据个数；dataType 指结果的数据类型；op 为归约操作；MPI 中定义了相应的操作名。src 参数为归约后数据存放的进程 id。

MPI_Ireduce 函数是 MPI_Reduce 的异步版本，无论通信子内的其它进程是否执行至此调用进程会立即返回。要保证其语义，程序必须确认操作已经完成。

MPI 预定义的全局数据运算符 MPI_Op 有：MPI_MAX/MPI_MIN-求最大/最小；MPI_SUM-求和；MPI_PROD-求积；MPI_LAND-逻辑与；MPI_LOR-逻辑或。

如果要求所有的进程都拥有归约后的结果，则使用 MPI_Allreduce 或 MPI_Iallreduce，相比 MPI_Reduce，没有 src 参数。

注：可以自定义归约操作，这如两个函数有关，一是：MPI_Op_create，二是：MPI_function，这两个函数都要遵守一定的规则。

要注意的是：在 MPI 中很多函数好像让人用在如果进程号等于多少才调用，其实要求每一个进程都调用才能得到正确的结束，集合类函数很多就是如此。或许 MPI 在实现它们的时候并不是由这个进程向各个进程发，而是由各个进程到数据所在的进程取。

下面是经典的计算 π 例子，主要使用了归约。

```
   #include "MPIUtil.h"


   int main(int argc, char **argv) {
      checkMPIError(MPI_Init(&argc, &argv));
      int rank, size;


      checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &rank));
      checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &size));


      int num;
```

```
    if(0 == rank){
        printf("please input computing num:");
        fflush(stdout);
        scanf("%d", &num);
    }

    checkMPIError(MPI_Bcast(&num, 1, MPI_INT, 0, MPI_COMM_WORLD));

    double h = 1.0/(size*num);
    double partPI = 0.0;

    for(int i = rank*num; i < (rank+1)*num; i++){
        double x = h*(i + 0.5);
        partPI += 1.0/(1+x*x);
    }

    partPI *= h;

    double PI;
    checkMPIError(MPI_Reduce(&partPI, &PI, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD));

    if(0 == rank){
        printf("PI = %1.20f\n", PI*4);
    }

    checkMPIError(MPI_Finalize());

    return 0;
```

```
}
```

10. 散发

函数 MPI_Scatter/MPI_Iscatter 把某个进程的数据平均的散发给多个进程，其原型如下：

```
int MPI_Scatter(void* buf, int count, MPI_Datetype dataType, void* r, int c,
MPI_Datatype dataTypeS, int src, MPI_Comm comm);
int MPI_Iscatter(void* buf, int count, MPI_Datetype dataType, void* r, int c,
MPI_Datatype dataTypeS, int src, MPI_Comm comm, MPI_Request *req);
```

要散发的数据存在于 src 进程的 buf 参数中，要发送给某个进程数据个数存在于 count 参数中，它应该和 c 参数一致，数据类型由 dataType 参数决定，它和 dataTypeS 参数一致，src 参数说明拥有 buf 的进程 id。散发机制为按照进程号顺序排列，这应该是一个比较常用函数，其中比较复杂的散发机制由函数 MPI_Scatterv 来承担，这个函数相当复杂，应少使用。MPI_Scatterv 原型如下：

```
int MPI_Scatterv(void *sendbuf, int *sendcnts, int *displs, MPI_Datatype
sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm
comm);
int MPI_Iscatterv(void *sendbuf, int *sendcnts, int *displs, MPI_Datatype
sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm
comm, MPI_Request *req);
```

11. 收集

函数 MPI_Gather 和 MPI_Scatter 相反，它将多个进程的数据收集到一个进程中，原型如下：

```
int MPI_Gather(void* buf, int count, MPI_Datetype dataType, void* b, int c,
MPI_Datatype dataTypeS, int src, MPI_Comm comm);
int MPI_Igather(void* buf, int count, MPI_Datetype dataType, void* b, int c,
MPI_Datatype dataTypeS, int src, MPI_Comm comm, MPI_Request *req);
```

收集到的数据放在 src 进程的 b 参数中；参数 buf 是发送的数据，同样参数 count 指发送的数据个数，它应当和参数 c 一致；参数 dataType 和参数 dataTypeS 一致，指发送接收的数据类型；src 参数指收集的数据放到该进程中。同样也有更复杂的收集函数 MPI_Gatherv。

函数 MPI_Gatherv 函数是 MPI_Gather 的复杂版本，其原型如下：

```
   int MPI_Gatherv(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void
*recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype, int src, MPI_Comm
comm);
   int MPI_Igatherv(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void
*recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype, int src, MPI_Comm
comm, MPI_Request *req);
```

sendbuf 是发送缓冲区的起始位置；sendcount 发送元素个数；sendtype 发送数据类型；recvcounts 整型数组(大小等于组的大小)，用于指明从各进程要接收的元素的个数(仅对根进程有效)；displs 整型数组(大小等于组的大小). 其元素 i 指明要接收元素存放位置相对于接收缓冲区起始位置的偏移量 (仅在根进程中有效)；recvtype 接收数据类型；src 指明接收进程。

如果要求每一个进程都拥有收集后的数据，使用函数 MPI_Allgather 或 MPI_Iallgather，此时要去掉 MPI_Gather 的 src 参数。

12. 前缀和

MPI_Scan 参数与 MPI_Reduce 一致，但是它所做操作称为前缀和。前缀和是指运算后的数据是运算前该数据前面所有数据的和。依据运算后第一个数据是否为 0，又将其分为两类，其原型如下：

```
   int MPI_Scan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm);
   int MPI_Iscan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm, MPI_Request *req);
   int MPI_Exscan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm);
```

```
int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *req);
```

1.6 通信子操作

1.6.1 通信子分裂

有时需要将通信子分组，然后在不同的组之间建立通信子进行通信。将通信子分组由函数：

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI Comm *ncom);
```

确定。其中 color 由各进程指定，comm 中相同 color 的所有进程成为一个组，key 同样由进程指定，其相对大小确定了其在新建立的组中的顺序。

如果需要在不同的组之间进行通信，那么就需要在组之间建立通信子。这由函数 MPI_Intercomm_create 产生，其原型如下：

```
int MPI_Intercomm_create(MPI_Comm lcomm, int g, MPI_Comm pcomm, int pg, int tag, MPI_Comm *ncomm);
```

其中 pcomm 和 lcomm 指使用 MPI_Comm_split 时的原通信子和分裂后的新通信子；g 和 pg 指需要通信的两个组，tag 用来区分通信组。

使用完后，需要释放建立的通信子空间，这由：

```
int MPI_Comm_free(MPI_Comm comm);
```

函数负责。

1.6.2 进程拓扑

在计算多维空间数据时，通信需要对计算空间进行多维分解，需要使用一个进程计算空间的一小块。如果对应的进程组也能够分解成多维表示，那么进程间通信会变得简单，进程拓扑能够满足这个要求。函数：

```
int MPI_Cart_create(MPI_Comm comm, int ndims, int *dims, int *periods, int reorder, MPI_Comm *ncomm);
```

创建一个虚拟进程拓扑。其中 comm 是原先的通信子；ndims 是创建的通信子的维数；dims 指明了各维的进程数目，其长度为 ndims；periods 长度为 dims，其值表示进程是否周期化；reorder 表示是否打乱原来进程的顺序。

如果 comm 内进程数目大于 ncomm 内的总进程数，则 comm 内某些进程返回的 ncomm 为 MPI_COMM_NULL，如果相反，调用出错。

如果需要创建可以满足进程数目不定的虚拟拓扑，那么需要确定每个维度上的进程数目，函数：

```
int MPI_Dims_create(int nnodes, int ndims, int dims[]);
```

如果 dims 的某维值为正数，则函数不会改变该维值。因此调用前要将其初始化为 0 或需要的值。

如果需要知道多维虚拟拓扑进程的坐标，函数：

```
int MPI_Cart_rank(MPI_Comm comm, const int *coords, int *rank);
```

依据虚拟拓扑坐标确定通信子空间坐标。其中 corrds 是输入的虚拟拓扑坐标。

很多时候需要知道某个进程的上下，左右进程，函数：

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *source, int
*dest)
```

direction 指明是纵向还是横向，0 表示纵向，1 表示横向。disp 表示相隔的距离。source 表示移动到当前位置的进程。dest 表示当前进程移到的位置。

相反也应该能够依据进程的通信子坐标确定虚拟拓扑坐标。函数：

```
int MPI_Cart_coords (MPI_Comm comm, int rank, int ndims, int *coords);
```

在获得虚拟拓扑坐标前需要知道虚拟拓扑维数，函数：

```
int MPI_Cartdim_get(MPI Comm comm, int *ndims);
```

下面的函数能够给出虚拟拓扑的维数，是否周期化，及当前进程的虚拟拓扑坐标。

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int
*coords);
```

下面给出了二维进程和上下左右进程交换信息的例子。

```
#include"mpi.h"
```

```
#include<stdio.h>
#define NUM_DIMS 2

int main( int argc, char **argv ){

    int rank,size;
    int periods[NUM_DIMS] = {0,0};
    int dims[NUM_DIMS] = {0,0};
    int first, third;
    int data, buf=0;

    MPI_Comm my_comm;
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Dims_create(size, NUM_DIMS, dims);
    MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, 0, &my_comm);


    for(int dim = 0; dim < NUM_DIMS; dim++){
       MPI_Cart_shift(my_comm, dim, 1, &first, &third);
      if(0 <= first){
              MPI_Isend(&rank, 1, MPI_INT, first, 0, MPI_COMM_WORLD,
&request);
              printf("process %d send data %d to process %d\n", rank, rank,
first);
```

```
                MPI_Wait(&request, &status);
        }
        if(0 <= third){
                MPI_Isend(&rank, 1, MPI_INT, third, 0, MPI_COMM_WORLD,
&request);
                printf("process %d send data %d to process %d\n", rank, rank,
third);
                MPI_Wait(&request, &status);
        }

        if(0 <= first){
                MPI_Irecv(&buf, 1, MPI_INT, first, 0, MPI_COMM_WORLD,
&request);
                MPI_Wait(&request, &status);
                printf("process %d receive data %d from process %d\n", rank,
buf, first);
        }
        if(0 <= third){
                MPI_Irecv(&buf, 1, MPI_INT, third, 0, MPI_COMM_WORLD,
&request);
                MPI_Wait(&request, &status);
                printf("process %d receive data %d from process %d\n", rank,
buf, third);
        }
      }

      MPI_Finalize();
  }
  /*
```

```
process 0 receive data 1 from process 1

process 0 receive data 2 from process 2

process 0 send data 0 to process 1

process 0 send data 0 to process 2

process 1 receive data 0 from process 0

process 1 receive data 3 from process 3

process 1 send data 1 to process 0

process 1 send data 1 to process 3

process 2 receive data 0 from process 0

process 2 receive data 3 from process 3

process 2 send data 2 to process 0

process 2 send data 2 to process 3

process 3 receive data 1 from process 1

process 3 receive data 2 from process 2

process 3 send data 3 to process 1

process 3 send data 3 to process 2

*/
```

1.7 MPI 性能优化

有许多办法可以用于减小 MPI 的计算时间，主要有以下方面：

- 利用局部性和网络结构。程序访问的数据通常具有局部性的特点，而局部性能够提升缓存的利用效率。不同网络结构中不同的进程之间交换数据的时间是不同的，将网络结构和程序数据访问的局部性关联起来，以使得需要交换数据进程刚好是网络结构中访问数据耗时较小的，这就可提升性能。
- 应用和网络配置相匹配。不同的应用在通信和计算上具有不同的特点，有些应用需要进程之间频繁的交换数据，此时宜使用通信性能好的网络。有些应用如易平行具有很高计算但是通信很少，此时就应当使用计算性能好的网络。
- 并行时要考虑计算和通信的比例，计算的比例越大通常越有利于使用 MPI 并行，同时尽

量减少通信，提高每次通信的数据量。可以使用异步通信，持久通信来优化通信，全局通信尽量利用高效集合通信算法。

- 对于通信，减小通信次数和通信量，优先减少通信次数(一次多发送数据)，其次是通信量，尤其是计算节点间的通信。采用更好的通信协议。
- 使通信和计算重叠。
- 对于计算，设计更好的并行算法，并考虑系统结构，进而将算法更好的映射到硬件环境上。挖掘算法的并行度，减少串行部分热点比例。考虑系统的负载平衡，减少 CPU 空闲等待时间。通过引入重复计算来减少通信，即以计算换通信。