消息传递接口MPI

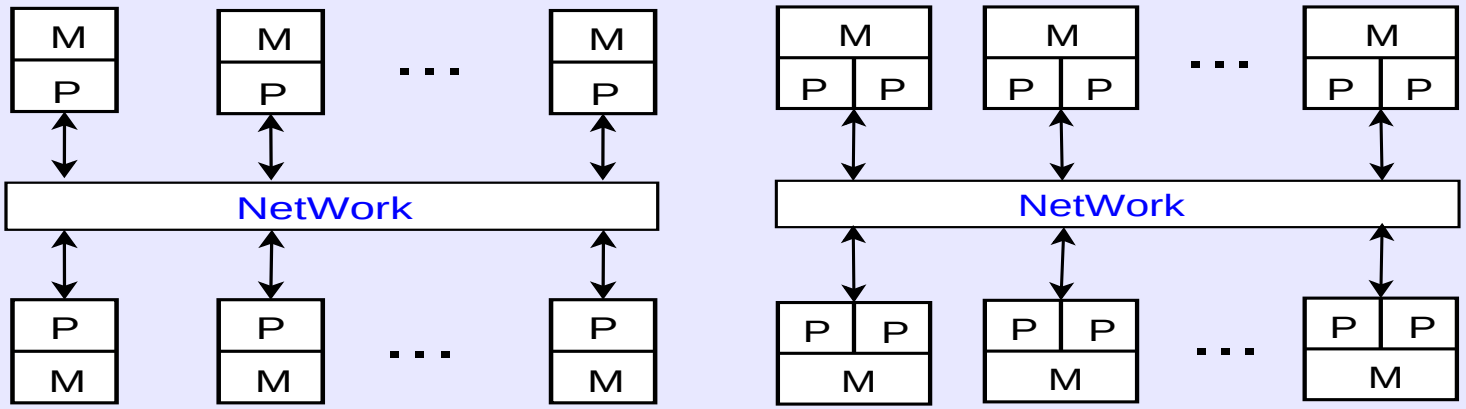
陈俊清

jqchen@math.tsinghua.edu.cn

清华大学数学科学系

March 15, 2013

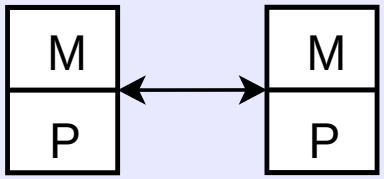
消息传递编程模型



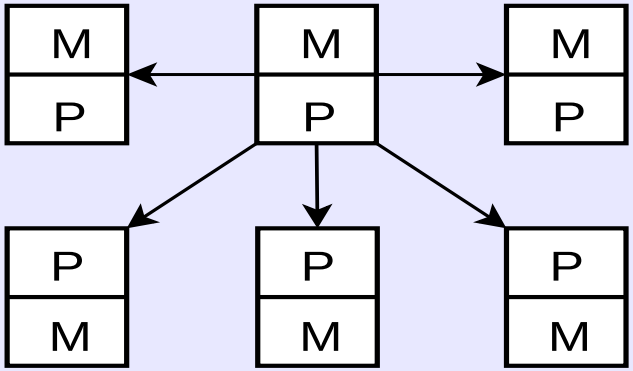
特点:

- 每个进程有独立的内存地址空间，数据分块独立存放;
- 编程复杂，需要显式并行操作;
- 可移植性强，可扩展性好。

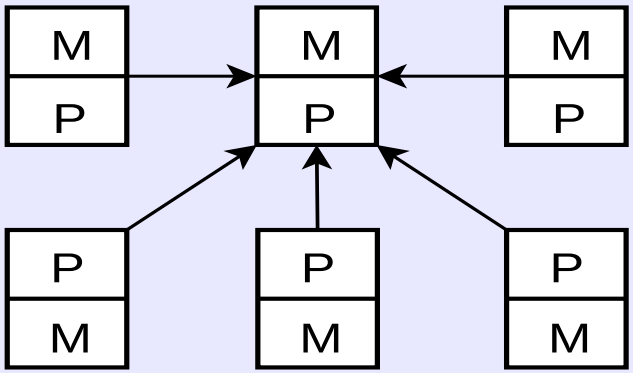
常见消息传递模式



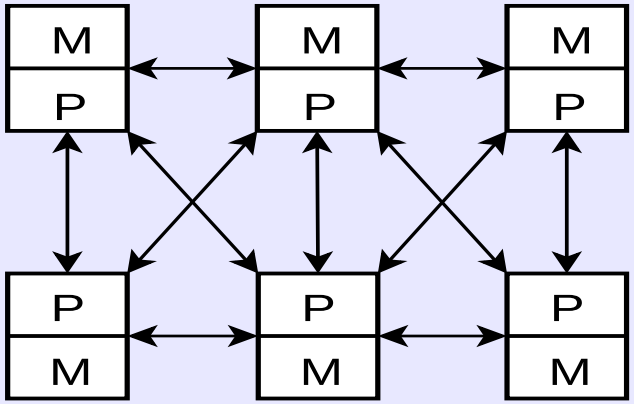
(c) 点对点通信



(d) 一对多通信（广播、散发）



(e) 多对一通信（收集、归约）



(f) 多对多通信（全收集、全交换、全归约、归约分发）

Part I

点对点通信

点对点通信

点对点通信（point to point communicatoin）是指一对进程之间进行的消息收发操作：一个进程发送，另一个进程接收。

- 阻塞型通信函数;
- 非阻塞型通信函数;
- 持久通信函数（非阻塞型）;

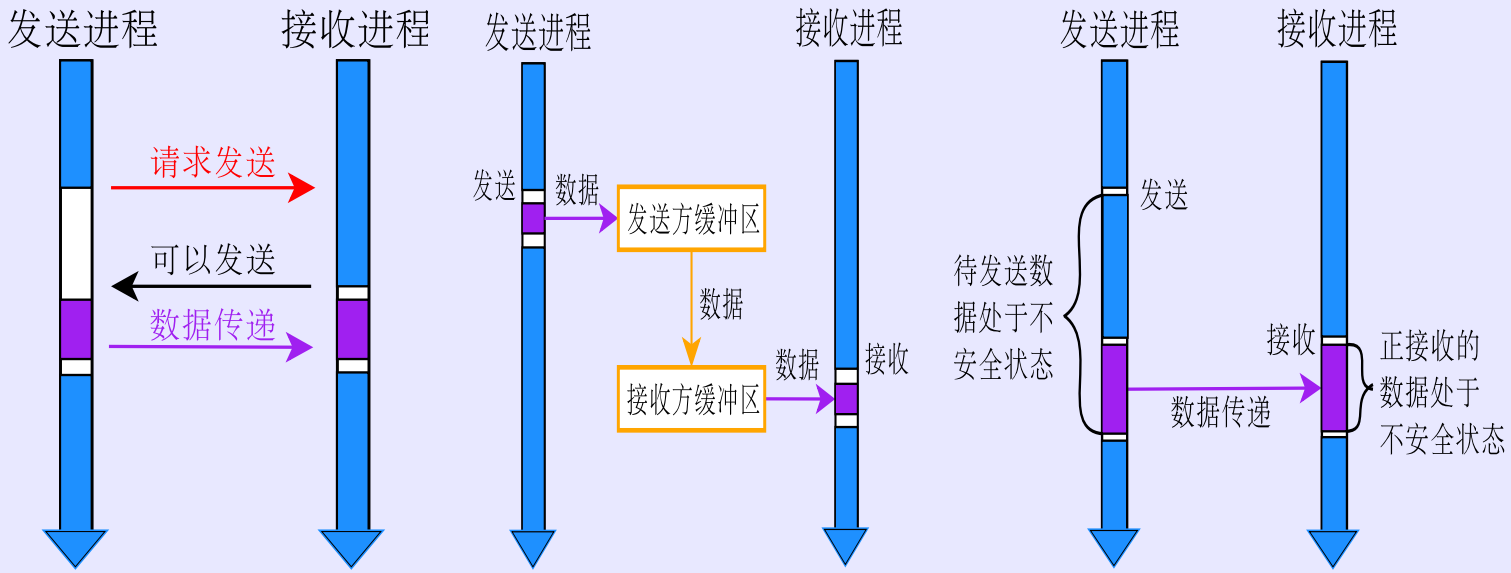
点对点通信共有12对函数，分别对应以上三种类型，每种4对。

阻塞型与非阻塞型函数

- **阻塞型（blocking）**：阻塞型函数需要等待指定操作的实际完成，或至少所涉及的数据已经被MPI系统安全地备份后才返回。阻塞型函数的操作一般是非局部的，它的完成可能需要与其它进程进行通信。阻塞型函数使用不当很容易引起程序的死锁。
- **非阻塞型（nonblocking）**：非阻塞型函数的调用总是立即返回，而实际操作则由MPI系统在后台进行。用户必须随后调用其它函数来等待或查询操作的完成情况。在操作完成之前对相关数据区的操作是不安全的，因为随时可能与正在后台进行的操作发生冲突。非阻塞型函数调用是局部的，它的完成不需要与其它进程进行通信。

MPI 通信模式

- 阻塞型：无缓冲区(1(g))、带缓冲区(1(h))
- 非阻塞型(1(i))



(g) 阻塞型　(h) 阻塞型　(i) 非阻塞型

Outline

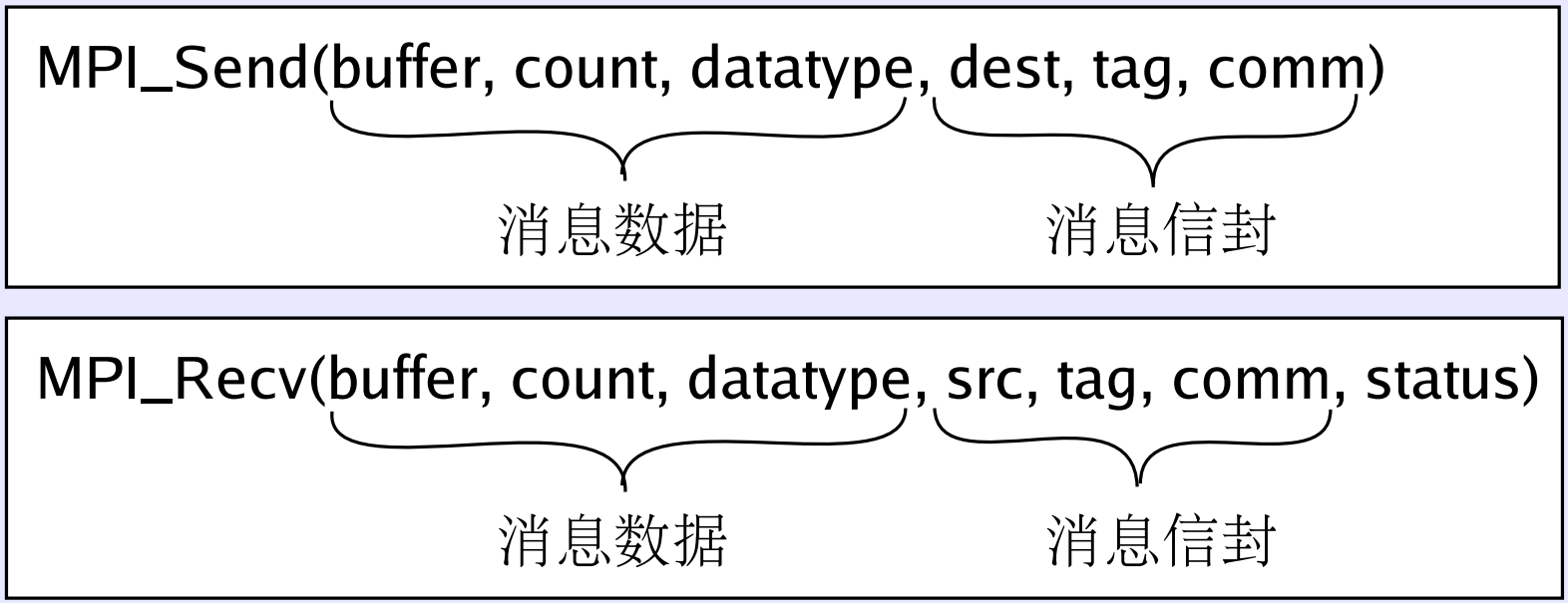
Introduction

例: ex3.c

```
char message[20];
if(myrank == 0){
    strcpy(message,"Hello,there");
    MPI_Send(message,strlen(message)+1, MPI_CHAR, 1, 99,
                        MPI_COMM_WORLD);
}
else if (myrank == 1){
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
                    &status);
    printf("Received: %s \n ",message);
}
```

MPI消息的构成

- 消息信封：⟨ 消息源／目的地、消息标识、通信域 ⟩
- 消息数据：⟨ 消息起始地址、数据个数、数据类型 ⟩



Outline

标准阻塞发送

```
int MPI_Send(void *buff, int count,
        MPI_Datatype datatype, int dest, int tag,
        MPI_Comm comm)
```

buff：待发送数据的地址；count：发送的数据量；datatype：发送的数据类型； tag的取值范围为0－MPI_TAG_UB;
dest的取值范围为0－np-1（np为通信器comm中的进程数)或MPI_PROC_NULL;
count 是指定数据类型的个数，而不是字节数。

标准阻塞接收

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

tag的取值范围为0－MPI_TAG_UB，或MPI_ANY_TAG;
source的取值范围为0－np-1（np为通信器comm中的进程数），
或MPI_ANY_SOURCE或MPI_PROC_NULL;
count给出接收缓冲区的大小（指定数据类型的个数），它是接收数据长度的上界。具体接收数据长度可以通过调用MPI_Get_count函数得到。 如需接收未知长度的数据，需要利用 MPI_Probe函数。

返回状态status

status返回有关接收到的消息的信息，它的类型是由MPI系统定义的，其结构如下（在C语言中status是一个结构）：

```
typedef struct {
    ... ...
    int MPI_SOURCE;  消息源地址
    int MPI_TAG;     消息标签
    int MPI_ERROR;   错误码
}MPI_Status;
```

查询接收到的消息长度

```
int MPI_Get_count(MPI_Status *status,
        MPI_Datatype datatype, int *count);
```

该函数在count中返回消息的长度（数据类型个数），数据类型必须与接收的信息中数据类型一致。

FORTRAN 的标准阻塞点对点函数

Fortran 函数

```
    MPI_SEND(BUFF, COUNT, DATATYPE, DEST, TAG, COMM, ERR)
    <type> BUFF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, ERR
```

```
    MPI_RECV(BUFF, COUNT, DATATYPE, SOURCE, TAG, COMM,
 + STATUS, IERR)
    <type> BUFF(*)
   INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, IERR
   INTEGER STATUS(MPI_STATUS_SIZE)
```

在Fortran中，STATUS各元素的定义如下：
STATUS(MPI_SOURCE)：消息源地址
STATUS(MPI_TAG)：消息标签
STATUS(MPI_ERROR)：错误码

消息传递三阶段

1. 在发送进程中，封装消息，从发送缓冲区到MPI系统;
2. 基于通信器，消息从发送进程到接收进程;
3. 在接收进程中，拆卸消息，从MPI系统到接收缓冲区。

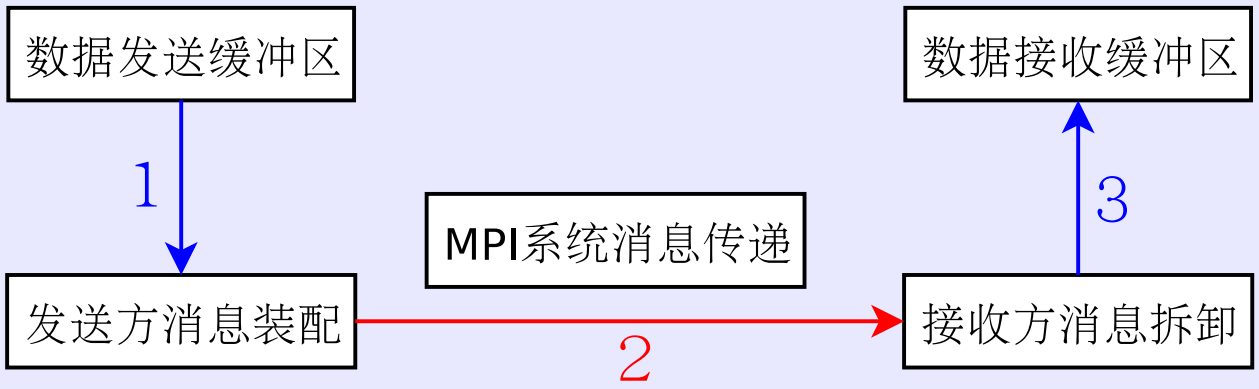


Figure: 每个阶段都要保证数据类型匹配

数据匹配

消息传递过程中的每一步都要符合数据类型匹配。

- 编程语言与MPI原始数据类型间的匹配（C语言）； int $\longleftrightarrow$ MPI_INT;
  double $\longleftrightarrow$ MPI_DOUBLE;
  char $\longleftrightarrow$ MPI_CHAR;
  ......
  例外： MPI_BYTE, MPI_PACKED
  int,double, char,... $\longleftrightarrow$ MPI_BYTE
- 接收与发送的匹配;
  MPI_INT$\longleftrightarrow$ MPI_INT
  MPI_DOUBLE $\longleftrightarrow$ MPI_DOUBLE
  ......

数据匹配

正确的匹配(MPI_DOUBLE a, b):

```
MPI_Status status;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank == 0)
    MPI_Send(a,10,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
else if(myrank == 1)
    MPI_Recv(b,15,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status)
```

数据匹配

错误的匹配:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
if(myrank == 0)
   MPI_Send(a,10,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
else if(myrank == 1)
   MPI_Recv(b,120,MPI_BYTE,0,tag,MPI_COMM_WORLD,&status);
```

正确的匹配:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
if(myrank == 0)
   MPI_Send(a,80,MPI_BYTE,1,tag,MPI_COMM_WORLD);
else if(myrank == 1)
   MPI_Recv(b,120,MPI_BYTE,0,tag,MPI_COMM_WORLD,&status);
```

数据匹配

归纳起来，数据类型匹配可归结为：

- 有类型数据的通信，发送方的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。

消息传递的顺序：tag的作用

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0)
    MPI_Send (buf1, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    MPI_Send (buf2, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
else
    MPI_Recv (buf1, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD
    MPI_Recv (buf2, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat)
... ...
```

消息传递的顺序：tag的作用

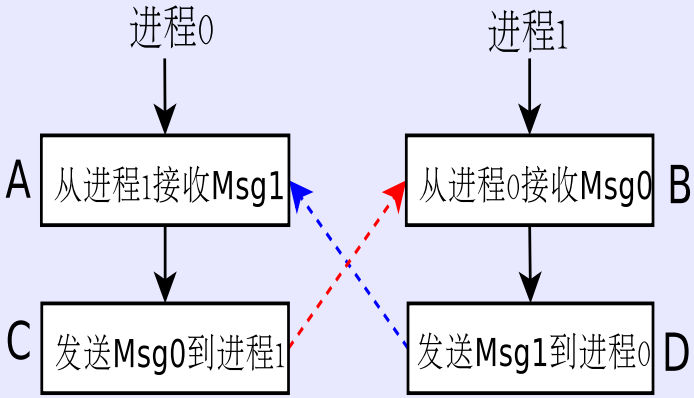
```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0)
    MPI_Send (buf1, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    MPI_Send (buf2, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
else
    MPI_Recv (buf2, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat)
    MPI_Recv (buf1, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat)
... ...
```

消息传递的顺序：tag的作用

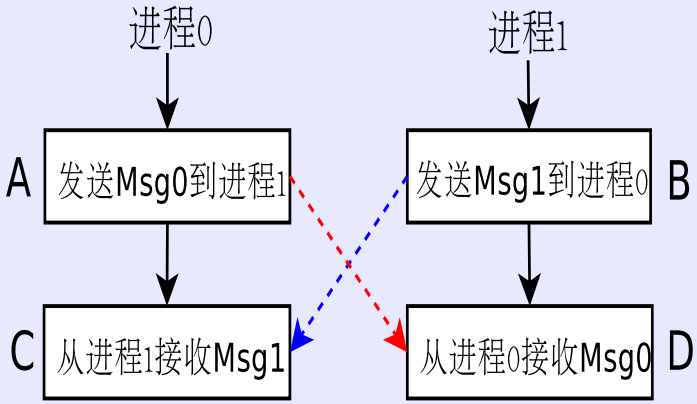
```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0)
    MPI_Send (buf1, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
    MPI_Send (buf2, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD);
else
    MPI_Recv (buf2, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, &stat
    MPI_Recv (buf1, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, &stat
... ...
```

参看例子程序:ex3-0.c

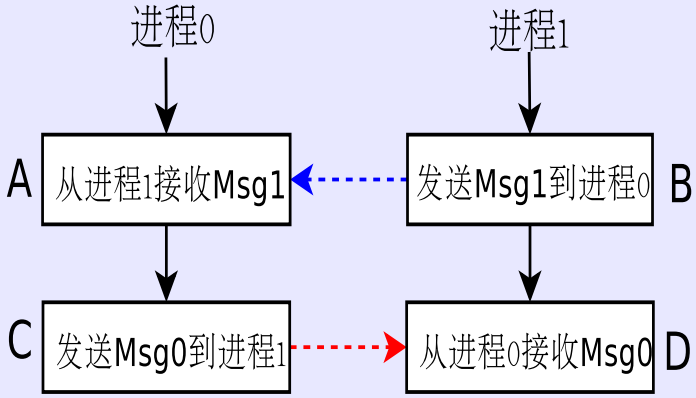
死锁现象



(a) 一定死锁



(b) 可能死锁



(c) 不会死锁

消息传递实例

- 阻塞型消息传递：ex3-1.c
- 消息收发顺序导致的程序死锁：ex3-2.c
- 长消息导致的程序死锁：ex3-3.c
- 0号进程先发，其它进程先收：ex3-4.c
- 偶数号进程先发，奇数号进程先收：ex3-5.c。当nprocs=1时有可能发生死锁。

如果将接收和求和中的数组B改成A，会得到什么结果?

其它一些阻塞型消息传递函数

- 发送与接收组合进行

```
int MPI_Sendrecv(void *sendbuff, int sendcount,
        MPI_Datatype sendtype, int dest, int sendtag,
        void *recvbuff, in recvcount,
        MPI_Datatype recvtype, int source, int recvtag
        MPI_Comm comm, MPI_Status *status)
```

本函数将一次发送调用和一次接收调用合并进行。它使得MPI更加简洁，更重要的是，MPI的实现通常能够保证使用MPI_Sendrecv的程序不会出现之前例子中出现的消息收发配对不好而造成的死锁。
例子(ex3-6.c):用MPI_Sendrecv替换MPI_Send和 MPI_Recv.

其它一些阻塞型消息传递函数

- 发送与接收组合进行，收发使用同一个缓冲区：

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

Outline

消息发送模式

- **标准模式(standard mode)**:由MPI系统决定是否将消息copy到缓冲区然后立即返回 （此时消息的发送由MPI在后台进行），还是等待将数据发送出去再返回。 大部分MPI系统都预留了一定大小的缓冲区，当发送消息的长度小于缓冲区长度的时候，会将消息缓存然后立即返回， 否则当消息部分或全部发送完再返回。标准模式发送被称为是非局部的，因为它的完成可能需要与接收方联络，标准模式的阻塞发送函数为MPI_Send.
- **缓冲模式(buffered mode)**:MPI系统将消息拷贝到用户提供的一个缓冲区然后立即返回， 消息的传递由MPI在后台进行。用户必须保证提供的缓冲区大小足以容下缓冲模式传递的消息。 缓冲模式发送操作被称为是局部的，因为它不需要与接收方联络即可完成（返回）。 缓冲模式的标准阻塞发送函数为： MPI_Bsend.

消息发送模式

- **同步模式(synchronous mode)**:实际上就是标准模式基础上， 还要求接收方开始接收数据后函数调用才返回。 同步模式发送是非局部的。同步模式的阻塞发送函数是：MPI_Ssend.
- **就绪模式(ready mode)**:调用就绪模式发送消息的时候必须确保接收方已经处于就绪状态 （正在等件接收消息），否则该调用将产生一个错误。 该模式设立的目的是在一些以同步方式工作的并行系统上由于发送时可以假设接收方已经在接收 而减少一些消息发送的开销。如果一个使用就绪模式的MPI程序是正确的， 则将其中所有就绪模式的消息发送改为标准模式后也应该是正确的。 就绪模式的阻塞发送函数是MPI_Rsend.

阻塞型缓冲模式消息发送函数

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

各参数的含义与MPI_Send同。该函数将buf中的数据拷贝到一个用户指定的缓冲区中然后立即返回，实际发送由MPI系统在后台进行。
用户在调用该函数前必须调用MPI_Buffer_attach来为它提供一个缓冲区。

```
int MPI_Buffer_attach(void *buf, int size);
```

阻塞型缓冲模式消息发送函数

MPI只允许提交一个供MPI_Bsend使用的缓冲区，用户应该保证该缓冲区足够容下所有可能同时用MPI_Bsend发送的消息（每个消息所需的缓冲区的长度通常等于消息中数据的长度加上一个常量，用户可以调用函数MPI_Type_size来查询一个消息实际需要的缓冲区长度）。

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

MPI_Buffer_detach撤销用MPI_Buffer_attach提交的缓冲区。在调用前用户不应该修改缓冲区中的内容，以免破坏正在发送的消息。该函数的返回亦表明所有缓冲模式的消息发送均已完成。

```
int MPI_Buffer_detach(void *buffer_addr, int *size);
```

注意，上述函数中buffer_addr和size均为输出参数，其中buffer_addr返回所撤销缓冲区的地址，size返回所涉及缓冲区的长度。

阻塞缓冲型的例子

参看： ex3-7.c

阻塞型通信小结

- 对于各种发送模式，接收操作均为：MPI_Recv。MPI_Recv是一个阻塞型操作，当接收缓冲区中收到了期待的数据才返回；
- 缓冲通信模式主要用于解开发送和接收之间的耦合。有了缓冲机制，即使接收端没有启动相应的接收，在完成数据缓存后，发送端也可以返回。增加了内存到内存复制的开销；
- 数据类型必须匹配；
- 同一个进程所执行的通信动作总是有序的：

```
    if(myrank == 0){
MPI_Bsend(buf1, count, MPI_DOUBLE, 1, tag1, comm);
MPI_Ssend(buf2, count, MPI_DOUBLE, 1, tag2, comm);
    }
    else if(myrank == 1){
MPI_Recv(buf1, count, MPI_DOUBLE, 0,tag2,comm);
MPI_Recv(buf2, count, MPI_DOUBLE, 0, tag1,comm);
    }
```

Outline

非阻塞型通信

将通信与计算进行重叠是提高性能的一个重要方法。非阻塞通信是达到这一目的方法。
非阻塞通信需要通过发送start操作(send start call)来启动一个发送请求，不真正启动发送， 但可以在系统从发送数据区取走数据之前就返回。还需要一个发送操作complete(send complete call)来完成通信。 发送操作complete操作主要目的是确认数据已经从发送数据区复制走。
在start和complete中间可以并行进行数据传输和计算。

非阻塞型点对点通信函数

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

该函数提交一个消息发送请求，要求MPI系统在后台完成消息的发送。MPI_Isend函数为该发送创建一个请求并将请求的句柄通过request变量返回给MPI进程， 供随后查询／等待消息发送的完成用。 与阻塞消息发送一样，非阻塞消息发送也有四种模式： 标准模式（Standard）、缓冲模式（Buffer）、同步模式（Synchonous）和就绪模式(Ready)，后三种模式较少使用， 我们在此不作介绍。

非阻塞接收

```
int MPI_Irecv(void *buff, int count,
        MPI_Datatype datatype, int src, int tag,
        MPI_Comm comm, MPI_Request *request)
```

该函数提交一个消息接收请求，要求MPI系统在后台完成消息的接收。MPI_Isend函数为该接收创建一个请求并将请求的句柄通过request变量返回给MPI进程，供随后查询／等待消息接收的完成用。

通信请求的完成与检测

等待、检测一个通信请求的完成:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *
```

MPI_Wait是阻塞型函数，它必须等待通信的完成才返回，MPI_Test是对应的非阻塞型函数。 MPI_Wait等待指定的通信请求完成然后返回。成功返回时，status中包含关于所完成的通信的信息， 相应的通信请求被释放，request被置成MPI_REQUEST_NULL. MPI_Test检测指定的通信请求，不论通信是否完成都立即返回。 如果通信已经完成则flag=true,status中包含关于所完成的通信的信息，相应的通信请求被释放，request被置成MPI_REQUEST_NULL。如果通信尚未完成则返回flag=false. 对于接收请求，status返回的内容与MPI_Recv返回的一样。对于发送请求，status 的返回值不确定。

通信请求的完成与检测

等待检测一组通信请求中某一个的完成:

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

等待或检测array_of_requests所给出的通信请求中任何一个的完成。 成功返回时，index中包含所完成的通信请求在数组array_of_requests中的位置。 其它参数与MPI_Wait和MPI_Test同。

通信请求的完成与检测

等待检测一组通信请求的全部完成:

```
int MPI_Waitall(int count,
        MPI_Request *array_of_requests,
        MPI_Status *array_of_statuses)
int MPI_Testall(int count,
        MPI_Request *array_of_requests,
        MPI_Status *array_of_statuses)
```

当函数返回值等于MPI_ERR_IN_STATUS时表明部分通信请求的处理出错， 此时用户应检查array_of_statuses的每个元素中MPI_ERROR域的值来得到这些通信请求的错误码。

通信请求的完成与检测

等待检测一组通信请求中一部分的完成:

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

outcount中返回的是成功完成的通信请求个数。 array_of_indices的前outcount个元素给出的是已完成的通信请求在数组array_of_requests中的位置。 当指定的通信请求中至少有一个已经完成时或发生错误时MPI_Waitsome才返回， 而MPI_Testsome当所有指定的通信请求都未完成时在outcount中返回0。 当函数返回值等于MPI_ERR_IN_STATUS时表明部分通信请求的处理出错， 此时用户应检查array_of_statuses的每个元素中MPI_ERROR域的值来得到这些通信请求的错误码。

通信请求的释放

```
int MPI_Request_free(MPI_Request *request)
```

MPI_Request_free释放指定的通信请求（及所占用的内存）。如果与该通信请求相关联的通信尚未完成，则MPI_Request_free会等待通信的完成。函数成功返回后request的值将被置成MPI_REQUEST_NULL。

执行顺序

```
MPI_Comm_rank(MPI_COMM_WORLD, myrank);
if(myrank == 0){
    MPI_Isend(a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, req1);
    MPI_Isend(b, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, req2);
}/*进程0先发a再发b*/
else if(myrank ==1){
    MPI_Irecv(a, 1, MPI_INT, 0, MPI_ANY_TAG,
                    MPI_COMM_WORLD, req1);
   /*进程1一定先收a，哪怕b先到*/
    MPI_Irecv(b, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, req2);
    /*进程1然后再接收b*/
}
MPI_Wait(req1,status);
MPI_Wait(req2,status);
```

执行顺序

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.

非阻塞消息传递的例子

程序例子为：

`ex3-8.c`

消息探测与通信请求的取消

消息探测:

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

例如：在接收未知长度的消息的时候，可以先用MPI_Probe和MPI_Get_count得到消息的长度。
MPI_Probe属于阻塞型函数，它等待直到一个符合条件的消息到达后才返回。对应的非阻塞型函数为MPI_Iprobe,它不论是否有符合条件的消息都立即返回：如果探测到符合条件的消息flag=true, 否则flag=false.

```
int MPI_Iprobe(int source, int tag,
        int *flag, MPI_Status *status)
```

消息探测的例子

例:ex3-9.c

消息探测与通信请求的取消

通信请求的取消:

```
int MPI_Cancel(MPI_Request *request)
int MPI_Test_cancelled(MPI_Status *status, int *flag)
```

MPI_Cancel用于取消一个尚未完成的通信请求，它在MPI系统中设置一个取消该通信请求的标志然后立即返回，具体的取消操作由MPI系统在后台完成。 调用MPI_Cancel后，仍需调用MPI_Wait、 MPI_Test或MPI_Request_free等函数来完成并释放该通信请求。
MPI_Test_cancelled可用来检测一个通信请求是否已经被取消，如果通信被取消，返回flag=true.

点对点通信函数汇总

函数类型	通信模式	阻塞型	非阻塞型
消息发送函数	标准模式	MPI_Send	MPI_Isend
	缓冲模式	MPI_Bsend	MPI_Ibsend
	同步模式	MPI_Ssend	MPI_Issend
	就绪模式	MPI_Rsend	MPI_Irsend
消息接收函数		MPI_Recv	MPI_Irecv
消息检测函数		MPI_Probe	MPI_Iprobe
等待／查询函数		MPI_Wait	MPI_Test
		MPI_Waitall	MPI_Testall
		MPI_Waitany	MPI_Testany
		MPI_Waitsome	MPI_Testsome
释放通信请求		MPI_Request_free	
取消通信			MPI_Cancel
			MPI_Test_cancelled

Outline

持久通信请求

持久通信请求（persistent communication request）可以用于以完全相同的方式（相同的通信器、收发缓冲区、数据类型与长度、源／目的地址和消息标签）重复收发消息。目的是减少处理消息的开销及简化MPI程序。

创建持久发送请求

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

创建一个持久消息发送的请求，该函数并不开始实际消息的发送，而只是创建一个请求句柄并返回给用户程序，留待以后实际消息发送时用。MPI_Send_init 对应于标准模式的非阻塞发送。相应的还有另外三个函数MPI_Bsend_init、 MPI_Ssend_init和 MPI_Rsend_init,分别对应缓冲模式、同步模式和就绪模式。

创建持久消息接收请求

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

创建一个持久消息接收请求，该函数并不开始实际消息的接收，而是创建一个请求句柄返回给用户程序，留待以后实际消息接收时用。

```
int MPI_Start(MPI_Request *request)
int MPI_Startall(int count,
        MPI_Request *array_of_requests)
```

每次调用MPI_Start相当于调用一次与MPI_Xxxx_init相对应的非阻塞型通信函数（MPI_Isend,MPI_Irecv等）。MPI_Startall的作用与MPI_Start类似，但它一次启动多个通信，用一个MPI_Xxxx_init创建的持久通信请求可反复调用MPI_Start或MPI_Startall来完成多次通信。

持久通信请求的完成与释放

与普通非阻塞型通信函数一样，通过持久通信请求启动的通信也需要调用前面介绍的函数来等待或检测通信的完成情况或取消通信，或用MPI_Request_free来释放一个持久通信请求。

例使用持久通信请求函数：`ex3-10.c`

点到点通信总结

关于缓冲方式:

- 标准的Send:利用MPI系统缓冲数据;
- Bsend将MPI系统提供的缓冲区交给用户来管理;
- Rsend相当于不要缓冲区，但发送端不能提前等待;
- Ssend也相当于不要缓冲区，但允许等待。

关于死锁

预防办法:

- 确保发送数据小于MPI系统提供的缓冲区;
- 调整收发顺序,用正确的执行顺序防死锁;
- 利用组合发送接收，严格配对防死锁;
- 使用缓冲模式发送;
- 使用非阻塞通信（作业）.

关于阻塞与非阻塞

- 阻塞发送返回意味着发送缓冲区可以被再次使用（修改、删除等），不影响接收端的接收结果，但不意味接收完成。
- 阻塞接收操作当且仅当接收完成才返回。
- 非阻塞发送与接收工作行为类似，调用后可立即返回，而不用再管通信相关的事情。
- 非阻塞仅对MPI系统提出一个通信请求，在可能的时候启动通信，用户并没有办法预测什么时候启动。
- 非阻塞情形，在确定通信之前，任何对缓冲区的修改都是危险的。

执行顺序与公平性

- 顺序规定：
  - 先发送的消息在接收端不会发生顺序颠倒；
  - 发送进程向同一个目标发送了m1和m2两条消息，即使都与接收函数匹配，也只能按顺序来。
  - 对同一条消息，同一个进程上先启动的接收动作先接到消息。
- MPI系统不提供公平性措施，用户必须自己在程序设计中解决这个问题。 例如：进程0与进程1同时向进程2发送消息，并都与其中一个接收函数匹配，则只有一个进程的发送成功。

作业

- 针对例子程序中发生死锁的情况，非阻塞方式解决死锁。
- 数组A[8]={0,1,2,3,4,5,6,7}，存放于进程0中，将A的前两个分量发送给进程0,三四两个分量发送给进程1， 第五六两个分量发给进程2，将最后两个分量发给进程3.编程实现这个过程，提交程序。