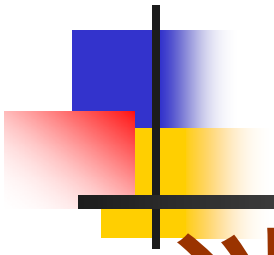


第五讲

消息传递编程接口 MPI

二、 MPI 消息传递

MPI 消息传递

MPI 点对点通信类型

- 阻塞型和非阻塞型

MPI 消息发送模式

- 标准模式、缓冲模式、同步模式、就绪模式

MPI 聚合通信

- 多个进程间的通信

主要内容

- 非阻塞型通信接口
- MPI 消息发送模式
- 持久通信
- 聚合通信

阻塞型和非阻塞型通信

阻塞型 (blocking) 和非阻塞型 (non blocking) 通信

- 阻塞型通信函数需要等待指定的操作实际完成，或所涉及的数据被 MPI 系统安全备份后才返回
  - 阻塞型通信是非局部操作，它的完成可能涉及其它进程
  - MPI_SEND 和 MPI_RECV 都是阻塞型的

- 非阻塞型通信函数总是立即返回，实际操作由 MPI 后台进行，需要调用其它函数来查询通信是否完成
  - 非阻塞型通信是局部操作
  - 在实际操作完成之前对相关数据区域的操作是不安全的
  - 在有些并行系统上，使用非阻塞型函数可以实现计算与通信的重叠进行
  - 常用的非阻塞型通信函数为 MPI_ISEND 和 MPI_IRECV

MPI_SEND

MPI_SEND(buf,count,datatype,dest,tag,comm)

参数	IN	buf	所发送消息的首地址
	IN	count	将发送的数据的个数
	IN	datatype	发送数据的数据类型
	IN	dest	接收消息的进程的标识号
	IN	tag	消息标签
	IN	comm	通信器

C

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

- 阻塞型消息发送接口
- 最基本的点对点通信函数之一

MPI_SEND

MPI_SEND(buf,count,datatype,dest,tag,comm)

- MPI_SEND 将缓冲区中 count 个 datatype 类型的数据发给进程号为 dest 的目的进程
- 这里 count 是元素个数，即指定数据类型的个数，不是字节数，数据的起始地址为 buf
- datatype 是 MPI 数据类型
- 本次发送的消息标签是 tag，使用标签的目的是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
- dest 的取值范围为0~np-1 (np 表示通信器 comm 中的进程数) 或 MPI_PROC_NULL，tag 的取值为 0~ MPI_TAG_UB
- 该函数可以发送各种类型的数据，如整型、实型、字符等

- 点对点通信是 MPI 通信机制的基础

MPI_RECV

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

参数	OUT buf	接收消息数据的首地址
	IN count	接收数据的最大个数
	IN datatype	接收数据的数据类型
	IN source	发送消息的进程的标识号
	IN tag	消息标签
	IN comm	通信器
	OUT status	返回状态
C	int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,MPI_Comm comm, MPI_Status *status)	

- 阻塞型消息接收接口
- 最基本的点对点通信函数之一

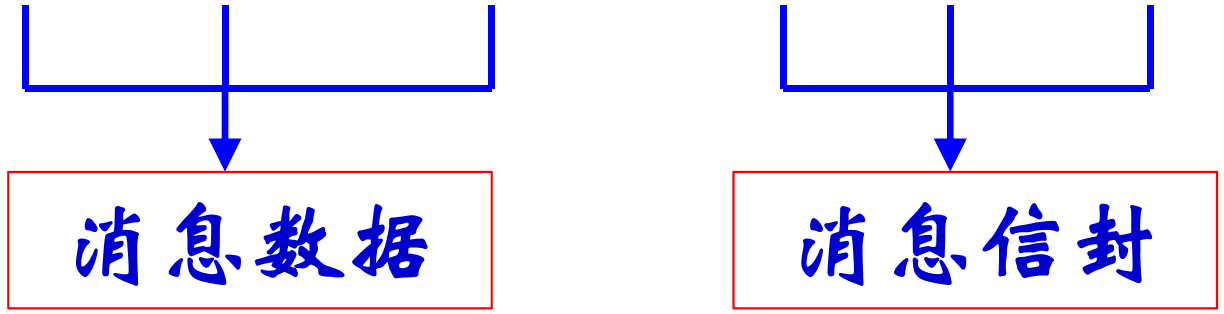
MPI_RECV

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

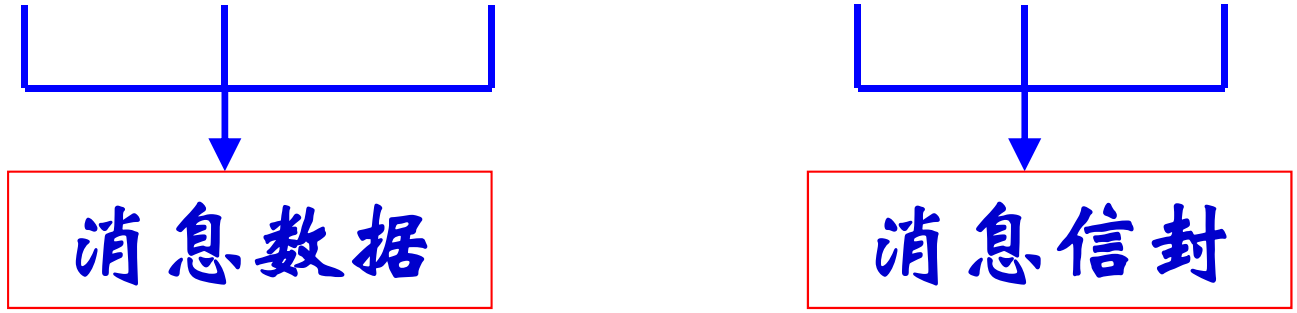
- 从指定的进程 source 接收不超过 count 个 datatype 类型的数据，并把它放到缓冲区中，起始位置为 buf，本次消息的标识为 tag
- source 取值范围为 0 ~np-1，或 MPI_ANY_SOURCE，或 MPI_PROC_NULL，tag 取值为 0~MPI_TAG_UB 或 MPI_ANY_TAG
- 接收消息时返回的状态 STATUS，在 C 语言中是用结构定义的，可供查询的成员包括 MPI_SOURCE，MPI_TAG 和 MPI_ERROR。
- 此外，STATUS 还包含接收消息元素的个数，但它不是显式给出的，需要调用函数 MPI_GET_COUNT 查询

MPI 发送接收

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




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



在使用阻塞型函数传递消息时要避免死锁！

C：结构

```
status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
```

MPI_SENDRECV

```
MPI_SENDRECV(sendbuf, sendcount, sendtype,
             dest, sendtag,
             recvbuf, recvcount, recvtype,
             source, recvtag, comm, status)
```

参数	参见 MPI_SEND 和 MPI_RECV

C	`int MPI_Sendrecv(void *sendbuf,`
	`  int sendcount, MPI_Datatype sendtype,`
	`  int dest, int sendtag,`
	`  void *recvbuf, int recvcount,`
	`  MPI_Datatype recvtype, int source,`
	`  int recvtag, MPI_Comm comm,`
	`  MPI_Status *status)`

- 发送消息和接收消息组合在一起

MPI_SENDRECV

```
MPI_SENDRECV(sendbuf, sendcount, sendtype,
             dest, sendtag,
             recvbuf, recvcount, recvtype,
             source, recvtag, comm, status)
```

- 好处是不用考虑先发送还是先接收消息，从而可以避免消息传递过程中可能的死锁
- sendbuf 和 recvbuf 必须指向不同的缓冲区

MPI_SENDRECV_REPLACE

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

参数	参见 MPI_SEND 和 MPI_RECV

C	`int MPI_Sendrecv(void *buf, int count,`
	`MPI_Datatype datatype,`
	`int dest, int sendtag,`
	`int source, int recvtag,`
	`MPI_Comm comm,`
	`MPI_Status *status)`

- 功能与 MPI_SENDRECV 类似，但收发消息使用的是同一个缓冲区

非阻塞通信接口

- MPI_ISEND
- MPI_IRECV
- MPI_WAIT、MPI_TEST
- MPI_WAITANY、MPI_TESTANY
- MPI_WAITALL、MPI_TESTALL
- MPI_WAITSOME、MPI_TESTSOME
- MPI_PROBE、MPI_IPROBE
- MPI_REQUEST_FREE
- MPI_CANCEL、MPI_TEST_CANCELLED

MPI_ISEND

MPI_ISEND(buf,count,datatype,dest,tag,comm,request)

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

- 非阻塞型发送函数
- 其中 request 是输出参数，为请求句柄，以备将来查询，其它参数含义与 MPI_SEND 相同
- 在 C 中 request 的数据类型是 MPI_Request

MPI_IRECV

```
MPI_IRECV(buf,count,datatype,source,tag,comm,request)
```

参数	略

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

- 非阻塞型接收函数
- 参数中没有 status，消息的查询使用 request

阻塞型/非阻塞型通信函数使用时要保持一致性！

MPI_WAIT

```
MPI_WAIT(request, status)
```

参数	IN request 通信请求 OUT status 消息状态
C	int MPI_Wait(MPI_Request *request, MPI_Status *status)

- 阻塞型通信检测函数：必须等待指定的通信请求完成后才能返回，与之相应的非阻塞型函数是 MPI_TEST
- 成功返回时，status 中包含关于所完成的通信的消息，相应的通信请求被释放，即 request 被置成 MPI_REQUEST_NULL

MPI_TEST

```
MPI_TEST(request, flag, status)
```

参数	OUT flag 操作是否完成标志 (其它参数含义同 MPI_WAIT)
C	int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

- 非阻塞型通信检测函数：不论通信是否完成都立刻返回
- 功能同 MPI_WAIT

MPI_WAITANY

```
MPI_WAITANY(count, array_of_requests, index, status)
```

参数	IN count 请求句柄的个数 INOUT array_of_requests 请求句柄数组 OUT index 已完成通信操作的句柄指标 OUT status 消息状态
C	int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)

- 阻塞型通信检测函数：所有请求句柄所对应的通信操作中至少有一个已经完成才返回
- 若有多个请求句柄已完成，则随机选择其中一个并立即返回

MPI_TESTANY

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

参数	OUT flag 操作是否完成标志 (其它参数含义同 MPI_WAITANY)
C	int MPI_Testany(int count, MPI_Request *array_of_requests, int *index, int *flag, MPI_Status *status)

- 非阻塞型通信检测函数
- 功能同 MPI_WAITANY

MPI_WAITALL

```
MPI_WAITALL(count, array_of_requests,
                   array_of_statuses)
```

参数	IN count 请求句柄的个数 INOUT array_of_requests 请求句柄数组 OUT array_of_statuses 所有消息状态数组
C	int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)

- 阻塞型通信检测函数
- 当所有的通信操作全部完成后才返回，否则将一直等待

MPI_TESTALL

```
MPI_TESTALL(count, array_of_requests, flag,
                                array_of_statuses)
```

参数	OUT flag 操作是否完成标志 (其它参数含义同 MPI_WAITALL)
C	int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses)

- 非阻塞型通信函数：无论所有的通信操作是否全部完成，都将立即返回
- 若有一个通信操作没有完成，则 flag 为 0（假）

MPI_WAITSOME

```
MPI_WAITSOME(incount, array_of_requests,
             outcount, array_of_indices,
             array_of_statuses)
```

参数	IN	incount	请求句柄的个数
	INOUT	array_of_requests	请求句柄数组
	OUT	outcount	已完成通信请求个数
	OUT	array_of_indices	已完成请求的下标数组
	OUT	array_of_statuses	所有消息的状态数组

C

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

- 阻塞型通信检测函数：至少有一个通信操作完成才返回

MPI_TESTSOME

```
MPI_TESTSOME(incount, array_of_requests, outcount,
             array_of_indices, array_of_statuses)
```

参数	参数含义同 MPI_WAITSOME
C	int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)

- 非阻塞型，若一个通信操作都没完成，则 outcount=0

MPI 消息检测函数

- **MPI_PROBE(source,tag,comm,status)**
  - 该函数用于检测一个符合条件的消息是否到达。它是阻塞型函数，必须等到一个符合条件的消息到达后才返回
  - 参数的含义与 **MPI_RECV** 相同

- **MPI_IPROBE(source,tag,comm,flag,status)**
  - 非阻塞型消息检测函数
  - **flag** 在 C 中为整型，在 FORTRAN 中为逻辑型
  - 如果符合条件的消息已到达，则 **flag** 为非零值/真，否则为 **0**/假

这两个函数中的参数 source 可以是 MPI_ANY_SOURCE，tag 也可以是 MPI_ANY_TAG，但必须指定通信器

MPI_PROBE/IPROBE

MPI_PROBE(source,tag,comm,status)

C

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

MPI_IPROBE(source,tag,comm,flag,status)

C

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

MPI_REQUEST_FREE

MPI_REQUEST_FREE(request)

C	MPI_Request_free(MPI_Request request)

- 释放指定的通信请求（及所占用的内存资源）
- 若该通信请求相关联的通信操作尚未完成，则等待通信的完成，因此通信请求的释放并不影响该通信的完成
- 该函数成功返回后 request 被置为 MPI_REQUEST_NULL
- 一旦执行了释放操作，该通信请求就无法再通过其它任何的调用访问

MPI_CANCEL

MPI_CANCEL(request)

C	MPI_Cancel(MPI_Request request)

- 非阻塞型，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后立即返回，具体的取消操作由 MPI 系统在后台完成。
- MPI_CANCEL 允许取消已调用的通信请求，但并不意味着相应的通信一定会被取消：若相应的通信请求已经开始，则它会正常完成，不受取消操作的影响；若相应的通信请求还没开始，则可以释放通信占用的资源。
- 调用 MPI_CANCEL 后，仍需用 MPI_WAIT，MPI_TEST 或 MPI_REQUEST_FREE 来释放该通信请求

MPI_TEST_CANCELLED

```
MPI_TEST_CANCELLED(status,flag)
```

C	`MPI_Test_cancelled(MPI_Status status, int *flag)`

- 检测通信请求是否被取消

MPI 消息发送模式

- MPI 提供四种点对点消息发送模式
  - 标准模式 (standard mode)
  - 缓冲模式 (buffered mode)
  - 同步模式 (synchronous mode)
  - 就绪模式 (ready mode)

每种发送模式都有相应的阻塞型和非阻塞型函数

MPI 消息发送模式

■ 标准模式 (standard mode)

- 由系统决定是先将数据复制到一个缓存区，然后返回；还是等待数据发送出去后才返回
- 通常 MPI 系统会预留一定大小的缓存区
- 标准模式阻塞型/非阻塞型函数: **MPI_SEND/MPI_ISEND**

■ 缓冲模式 (buffered mode)

- 将数据复制到一个用户指定的缓存区，然后立即返回
- 消息的发送由 MPI 系统后台进行
- 用户必须保证提供的缓存区足以保存所需发送的数据
- 缓冲模式阻塞型/非阻塞型函数: **MPI_BSEND/MPI_IBSEND**

MPI 消息发送模式

同步模式 (synchronous mode)

- 在标准模式的基础上要求确认接收方开始接收数据后才返回
- 同步模式阻塞型/非阻塞型函数: **MPI_SSEND/MPI_ISSEND**

就绪模式 (ready mode)

- 发送时假设接收方已经处于就绪状态，否则产生一个错误
- 缓冲模式阻塞型/非阻塞型函数: **MPI_RSEND/MPI_IRSEND**

MPI_BSEND、MPI_SSEND、MPI_RSEND 的用法同 MPI_SEND
MPI_IBSEND、MPI_ISSEND、MPI_IRSEND 的用法同 MPI_ISEND

点对点通信函数小结

函数类型	模式	阻塞型	非阻塞型
消息发送	标准	MPI_SEND	MPI_ISEND
	缓冲	MPI_BSEND	MPI_IBSEND
	同步	MPI_SSEND	MPI_ISSEND
	就绪	MPI_RSEND	MPI_IRSEND
消息接收		MPI_RECV	MPI_IRECV
消息检测		MPI_PROBE	MPI_IPROBE
等待/查询		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

持久通信请求

- 持久通信请求用于以完全相同的方式重复收发的消息，目的是减少处理消息时的开销，并简化 MPI 程序
- 持久通信请求收发步骤：先创建一个请求，然后进行收发
- 持久通信请求相关接口
  - 持久通信请求的创建（非阻塞型持久通信请求）

    `MPI_SEND_INIT、MPI_RECV_INIT`

  - 持久通信请求的收发

    `MPI_START、MPI_STARTALL`

- 持久通信请求可反复使用，进行多次通信

MPI_SEND_INIT

```
MPI_SEND_INIT(buf,count,datatype,dest,
              tag,comm,request)
```

C

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

- 创建一个非阻塞型消息发送请求
- 参数含义与 MPI_ISEND 相同
- 该函数并不进行消息的实际发送，只是创建一个请求句柄，通过 request 返回给用户程序，留待以后实际发送时使用
- MPI_SEND_INIT 对应标准的非阻塞型消息发送，相应地有 MPI_BSEND_INIT、MPI_SSEND_INIT、MPI_RSEND_INIT

MPI_RECV_INIT

```
MPI_RECV_INIT(buf,count,datatype,source,
              tag,comm,request)
```

C

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

- 创建一个非阻塞型消息接收请求
- 参数含义与 MPI_IRECV 相同
- 该函数并不进行消息的实际接收，只是创建一个请求句柄，通过 request 返回给用户程序，留待以后实际接收时使用

MPI_START/STARTALL

MPI_START(request)

C	`MPI_Start(MPI_Request *request)`

- 启动基于持久通信请求的通信，相当于调用相应的非阻塞型通信函数，需调用 **MPI_Wait** 函数来等待通信的完成

- **MPI_STARTALL(count, array_of_requests)**

C	`MPI_Startall(int count,` `MPI_Request *array_of_requests)`

- 启动多个基于持久通信请求的通信