華東師範大學 | 数学科学学院
School of Mathematical Sciences, East China Normal University

消息传递编程接口 MPI

（二）消息传递

目录页

Contents

目录页

Contents

1 MPI 点对点通信

- 阻塞型通信
- 非阻塞型通信
- 检测函数

阻塞型通信

MPI 中的通信分为：阻塞型和非阻塞型

阻塞型 (blocking)

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

阻塞型通信

非阻塞型 (non blocking)

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
- 在某些并行系统上，使用非阻塞型函数可以实现计算与通信的重叠
- 常用的非阻塞型通信函数为 MPI_ISEND 和 MPI_IRECV

非阻塞型发送函数

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

C

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

F77

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG,
             COMM, REQUEST, IERR)
<type>  BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG,
             COMM, REQUEST, IERR
```

- 这里 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 指针，FORTRAN 中为整型

非阻塞型接收函数

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

C

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

F77

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE,
                TAG, COMM, REQUEST, IERR)
<type>  BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG,
                COMM, REQUEST, IERR
```

- 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！

非阻塞型通信检测

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)
F77	MPI_WAIT(REQUEST, STATUS, IERR) INTEGER REQUEST, IERR, STATUS(MPI_STATUS_SIZE)

- 该函数是阻塞型的，它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

非阻塞型通信检测

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)
F77	MPI_TEST(REQUEST, FLAG, STATUS, IERR) LOGICAL FLAG INTEGER REQUEST, IERR, STATUS(MPI_STATUS_SIZE)

- 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)
F77	MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERR) INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, IERR, STATUS(MPI_STATUS_SIZE)

- 所有请求句柄中至少有一个已经完成才返回，阻塞型函数
- 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)`

F77	`MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS,`
	`INDEX, FLAG, STATUS, IERR)`
	`INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX,`
	`FLAG, IERR, STATUS(MPI_STATUS_SIZE)`

- 功能同 MPI_WAITANY，非阻塞型函数

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 所有消息的状态数组
C	int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)
F77	MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERR) INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR, ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)

- 当所有的通信操作全部完成后才返回，否则将一直等待
- 阻塞型函数

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,`
	`MPI_Status *array_of_statuses)`

F77	`MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS,`
	`ARRAY_OF_STATUSES, IERR)`
	`INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR,`
	`ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)`

- 非阻塞型，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)`
F77	`MPI_WAITSOME( INCOUNT,ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERR)` `INTEGER INCOUNT, OUTCOUNT, IERR, ARRAY_OF_REQUESTS(*), ARRAY_OF_INDICES(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)`

- 阻塞型，返回所有已经完成的通信，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)`

F77	`MPI_TESTSOME( INCOUNT,ARRAY_OF_REQUESTS,`
	`OUTCOUNT, ARRAY_OF_INDICES,`
	`ARRAY_OF_STATUSES, IERR)`
	`INTEGER INCOUNT, OUTCOUNT, IERR,`
	`ARRAY_OF_REQUESTS(*), ARRAY_OF_INDICES(*),`
	`ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)`

- 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

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

F77	`MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERR)`
	`INTEGER  SOURCE, TAG, COMM, IERR,`
	`  STATUS(MPI_STATUS_SIZE)`

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

F77	`MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERR)`
	`LOGICAL  FLAG`
	`INTEGER  SOURCE, TAG, COMM, IERR,`
	`  STATUS(MPI_STATUS_SIZE)`

MPI 释放通信请求函数

MPI_REQUEST_FREE(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 取消通信函数

MPI_CANCEL(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(status,flag)

- 检测通信请求是否被取消

目录页

Contents

2 MPI 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）

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

- 持久通信请求用于以完全相同的方式重复收发消息，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。

- 持久通信请求的创建（非阻塞型持久通信请求）

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

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

持久通信请求的创建

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

- 参数含义与 MPI_ISEND 相同
- 该函数并不开始消息的实际发送，而只是创建一个请求句柄，通过参数 request 返回给用户程序，留待以后实际发送时用
- 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 和 MPI_RSEND_INIT, 分别对应于缓冲、同步和就绪模式的非阻塞型消息发送

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

持久通信请求

- 持久通信请求的收发

  MPI_START(request)

  MPI_STARTALL(count, array_of_requests)

† 可反复调用 MPI_START 或 MPI_START_ALL 来进行多次通信。

- 持久通信请求的完成与释放

  MPI_WAIT (request, status)

  MPI_REQUEST_FREE (request)

目录页

Contents

3 MPI 聚合通信

- 一对多、多对一、多对多
- 通信，同步和计算

聚合通信

- 聚合通信 (collective communication) 是指多个进程之间的通信
- 根据数据的流向，聚合通信可分为三种类型：

  一对多、多对一、多对多
- 根进程 (root)：在一对多和多对一中的 “一”
- 聚合通信一般实现三个功能：通信，同步和计算

障碍同步

MPI_BARRIER(comm)

参数	IN comm 通信器
C	`int MPI_Barrier(MPI_Comm comm)`
F77	`MPI_BARRIER(COMM, IERR)` `INTEGER COMM, IERR`

- 这是 MPI 提供的唯一的一个同步函数
- 当 COMM 通信器中的所有进程都执行这个函数时才返回，
  如果有一个进程没有执行此函数，其余进程将处于等待状态。
  在执行完这个函数之后，所有进程将同时执行其后的任务

广播

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

参数	INOUT buf	通信消息缓冲区的起始地址
	IN count	将广播出去/或接收的数据个数
	IN datatype	广播/接收数据的数据类型
	IN root	广播数据的根进程的标识号
	IN comm	通信器
C	`int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`	

F77	`MPI_BCAST(BUF, COUNT, DATATYPE, ROOT, COMM, IERR)`
	`<type> BUF(*)`
	`INTEGER COUNT,DATATYPE,ROOT,COMM,IERR`	

- root 进程将 buf 中的内容广播发送给通信器内的所有进程（包括自身）

数据收集

- 数据收集指各个进程（包括根进程）将自己的一块数据发送给根进程，根进程将这些数据合并成一个更大的数据块
  - 收集相同长度的数据块：**MPI_GATHER**
  - 收集不同长度的数据块：**MPI_GATHERV**

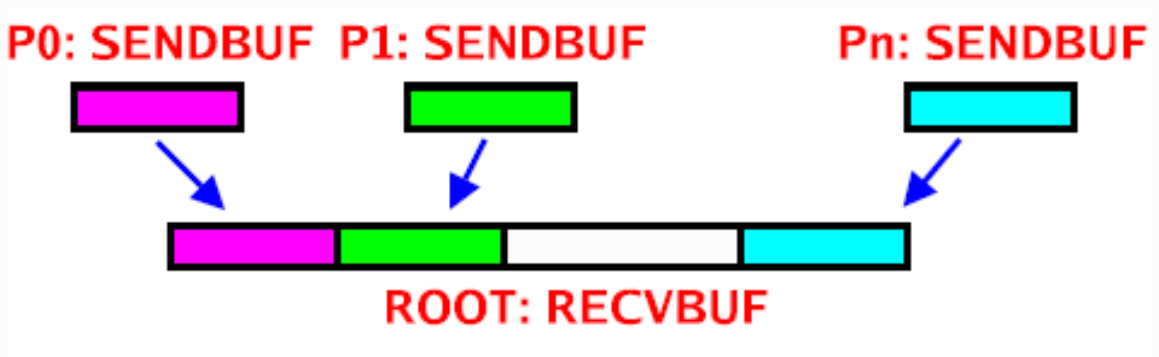
MPI_GATHER

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

- 根进程从通信器中的每个进程（含根进程）接收一个相同长度、相同类型的数据块，并按发送进程的进程号依次存放到自己的 **recvbuf** 中，合并成一个更大的数据块
- 参数 **recvbuf**、**recvcount**、**recvtype** 仅对根进程有意义
- 其作用就象一个进程组中的所有进程（包括 **root**）都执行了一个发送调用，同时根进程执行了 **np** 次接收调用，即

```
MPI_Send(sendbuf, sendcount, sendtype, root, ...)
if (myid==root)
    for(i=0; i<np, i++)
        mpi_recv(recvbuf+i*recvcount*extent(recvtype), recvcount,
                 recvtype, i, ...)
```

MPI_GATHER



C

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

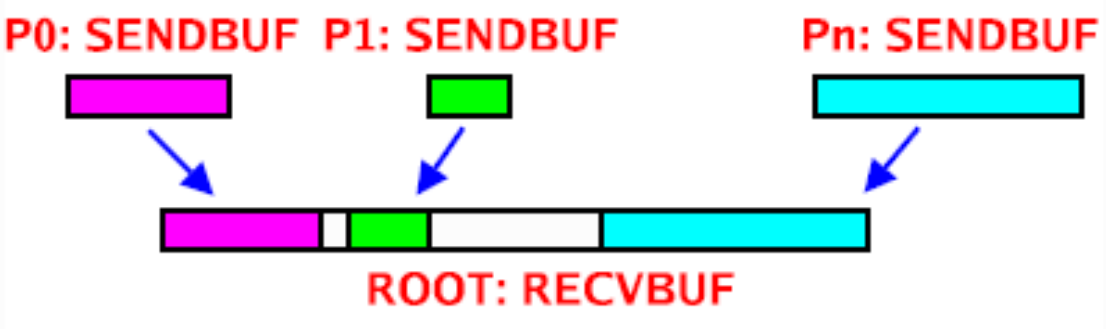
F77

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE,
           RECVBUF, RECVCOUNT, RECVTYPE,
           ROOT, COMM, IERR)
<type>   SENDBUF(*), RECVBUF(*)
INTEGER  SENDCOUNT, SENDTYPE, RECVCOUNT,
         RECVTYPE,ROOT, COMM, IERR
```

MPI_GATHERV

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

- 这个函数是 **MPI_GATHER** 的扩充，它允许从不同的进程中接收不同长度的消息。接收到的消息的存放位置由 **recvbuf** 和位置偏移参数 **displs**（数组）决定。
- 根进程中 **recvcounts(i)** 与第 i 个进程的 **sendcount** 一致



root 进程从第 i 个进程接收 **recvcounts(i)** 个数据，存放到以 **recvbuf+displs(i)*extent(recvtype)** 为首地址的缓冲区中

MPI_GATHERV

C	`int MPI_Gatherv(void *sendbuf, int sendcount,`
	`MPI_Datatype sendtype,`
	`void *recvbuf, int *recvcounts,`
	`int *displs, MPI_Datatype recvtype,`
	`int root, MPI_Comm comm)`

F77	`MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE,`
	`RECVBUF, RECVCOUNTS,DISPLS,`
	`RECVTYPE, ROOT, COMM, IERR)`
	`<type> SENDBUF(*), RECVBUF(*)`
	`INTEGER SENDCOUNT,SENDTYPE,RECVTYPE,ROOT,`
	`COMM,IERR,RECVCOUNTS(*),DISPLS(*)`

- 注：这里的 **recvcounts** 和 **displs** 是长度为 **np** 的数组

数据散发

- 数据散发是指根进程将一个大的数据块分成小块分别散发给各个进程（包括根进程自己），是数据收集的逆操作
  - 散发相同长度的数据块：**MPI_SCATTER**
  - 散发不同长度的数据块：**MPI_SCATTERV**

MPI_SCATTER

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

- 根进程的 **sendbuf** 中有 **np** 个连续存放的数据块，每个数据块包含 **sendcout** 个 **sendtype** 类型的数据，根进程将这些数据块按进程的标识号依次分发给各个进程（包括 **root** 进程）
- 参数 **sendbuf**、**sendcount**、**sendtype** 仅对根进程有意义

C

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

F77

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

MPI_SCATTERV

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

- 是 **MPI_SCATTER** 的扩展，
  它允许根进程向各个进程发送的长度不同且不一定是连续存放的数据块

C

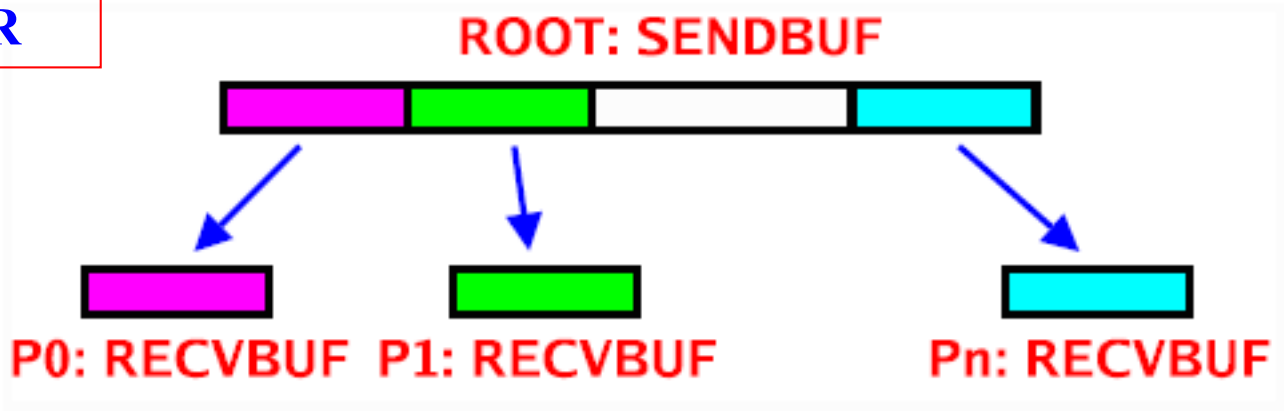
```
int MPI_Scatterv(void *sendbuf, int sendcounts,
        int *displs, MPI_Datatype sendtype,
        void *recvbuf, int *recvcount,
        MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```

F77

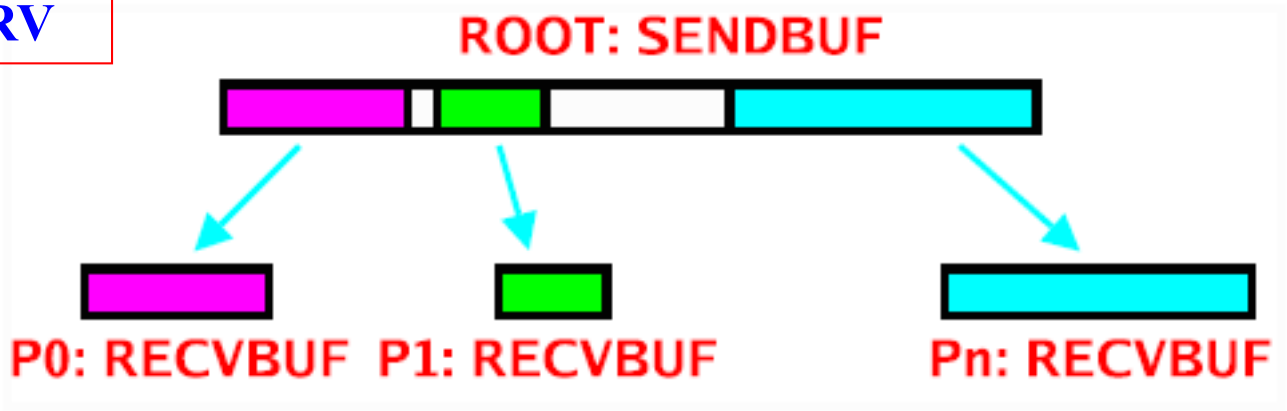
```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS,
          SENDTYPE, RECVBUF, RECVCOUNT,
          RECVTYPE, ROOT, COMM, IERR)
<type>   SENDBUF(*), RECVBUF(*)
INTEGER  SENDCOUNT,SENDTYPE,RECVTYPE,ROOT,
         COMM,IERR,RECVCOUNTS(*),DISPLS(*)
```

SCATTER/V

MPI_SCATTER



MPI_SCATTERV



多点对多点的通信

全收集

- 与数据收集类似，区别是所有进程都接收数据
- 相同长度数据块的全收集：**MPI_ALLGATHER**
- 不同长度数据块的全收集：**MPI_ALLGATHERV**

全收集散发

- 每个进程散发自己的一个数据块，并且收集拼装所有进程散发过来的数据块
- 它既可以认为是数据全收集的扩展，也可以被认为是数据散发的扩展。
- 相同数据长度的全收集散发：**MPI_ALLTOALL**
- 不同数据长度的全收集散发：**MPI_ALLTOALLV**

MPI_ALLGATHER

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

等价于依次以每个进程为根进程进 **np** 次 **MPI_GATHER**，或以任意进程为根进程调用一次 **MPI_GATHER**，紧接着再对收集到的数据进行一次广播

C

```
int MPI_Allgather(void *sendbuf,
        int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount,
        MPI_Datatype recvtype, MPI_Comm comm)
```

F77

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE,
              RECVBUF, RECVCOUNT, RECVTYPE,
              COMM, IERR)
<type>    SENDBUF(*), RECVBUF(*)
INTEGER   SENDCOUNT, SENDTYPE, RECVCOUNT,
          RECVTYPE, COMM, IERR
```

MPI_ALLGATHERV

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

- 不同长度数据块的全收集，参数与 **MPI_GATHERV** 类似
- 进程 **j** 的 **sendcount** = 所有进程的 **recvcounts(j)**

C

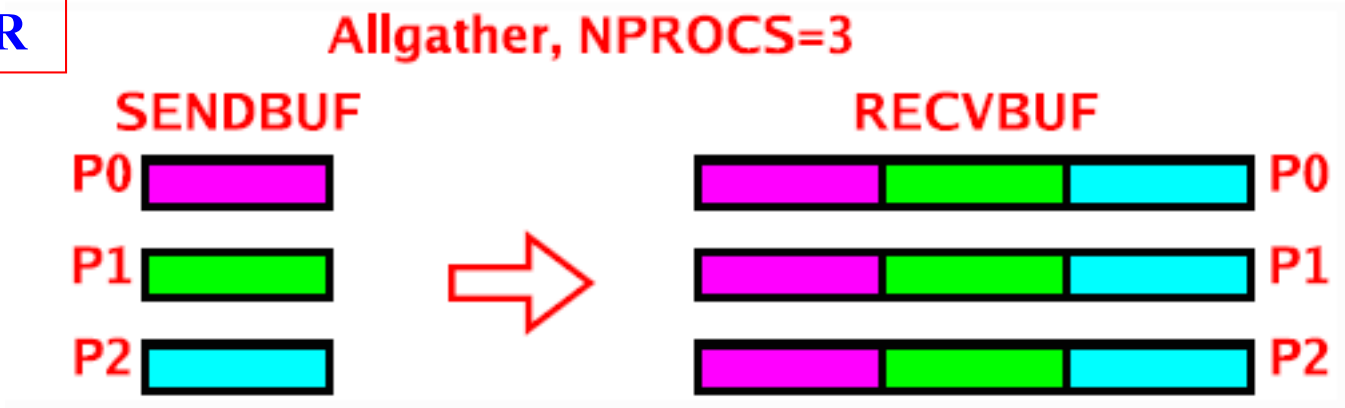
```
int MPI_Allgatherv(void *sendbuf,
        int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcounts,
        int * displs,MPI_Datatype recvtype,
        MPI_Comm comm)
```

F77

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE,
                RECVBUF, RECVCOUNTS, DISPLS,
                RECVTYPE, COMM, IERR)
<type>   SENDBUF(*), RECVBUF(*)
INTEGER  SENDCOUNT, SENDTYPE, RECVCOUNTS(*),
         DISPLS(*), RECVTYPE, COMM, IERR
```

ALLGATHER/V

MPI_ALLGATHER



MPI_ALLGATHERV

MPI_ALLTOALL

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

C

```
int MPI_Alltoall(void *sendbuf,
      int sendcount, MPI_Datatype sendtype,
      void* recvbuf, int recvcount,
      MPI_Datatype recvtype, MPI_Comm comm)
```

F77

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE,
             RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERR)
<type>   SENDBUF(*), RECVBUF(*)
INTEGER  SENDCOUNT, SENDTYPE, RECVCOUNT,
         RECVTYPE, COMM, IERR
```

- 相同长度数据块的全收集散发
- 进程 **i** 将 **sendbuf** 中的第 **j** 块数据发送至进程 **j** 的 **recvbuf** 中的第 **i** 个位置，**sendbuf** 和 **recvbuf** 均有 **np** 个连续的数据块构成
- 该操作相对于将数据/进程进行一次转置

MPI_ALLTOALLV

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

C

```
int MPI_Alltoallv(void *sendbuf,
      int sendcounts, int sdispls,
      MPI_Datatype sendtype, void* recvbuf,
      int recvcounts, int rdispls,
      MPI_Datatype recvtype, MPI_Comm comm)
```

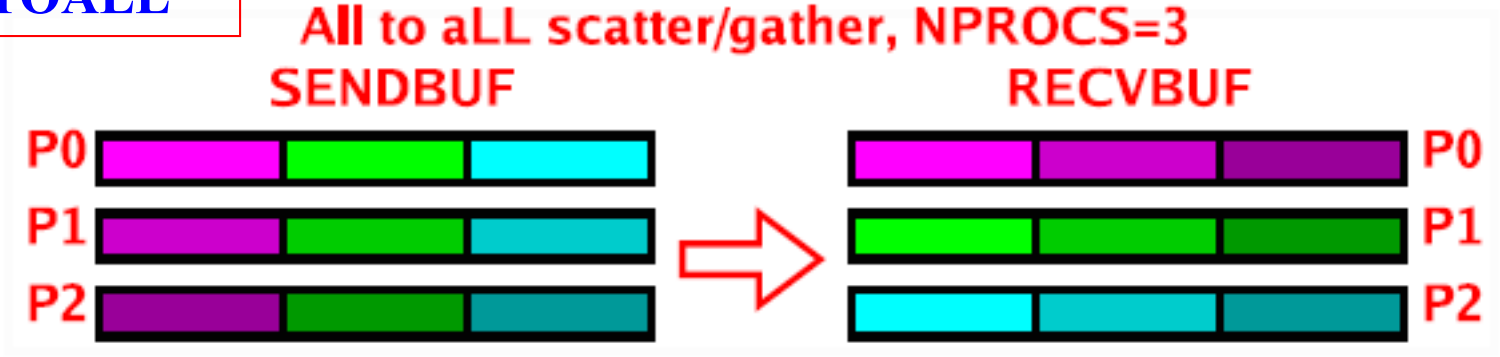
F77

```
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS,
              SENDTYPE, RECVBUF, RECVCOUNTS,
              RDISPLS, RECVTYPE, COMM, IERR)
<type>   SENDBUF(*), RECVBUF(*)
INTEGER  SENDCOUNT, SENDTYPE, COMM, IERR,
    RECVCOUNT,RECVTYPE,SDISPLS(*),RDISPLS(*)
```

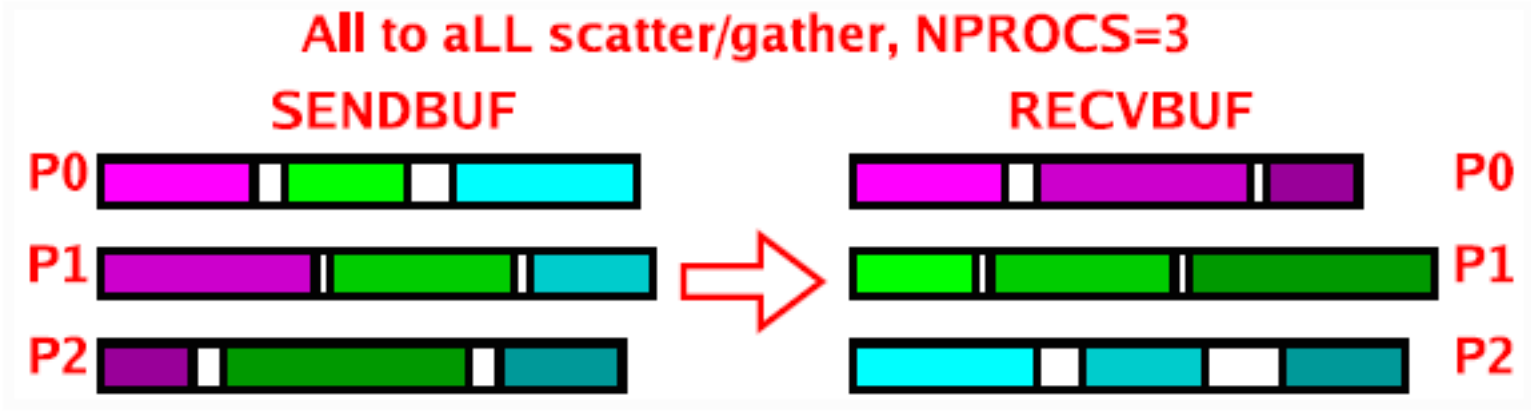
- 不同长度数据块的全收集散发

ALLTOALL/V

MPI_ALLTOALL



MPI_ALLTOALLV



归约

- 归约：**MPI_REDUCE**
  - 该函数将通信器内每个进程输入缓冲区（**sendbuf**）中的数据按给定的操作进行简单运算，并将其结果返回到根进程的输出缓冲区（**recvbuf**）中
  - 归约运算可以是 MPI 预定义的运算操作，也可以是用户自定义的运算

- 自定义归约运算操作：**MPI_OP_CREATE**
  - 用户自己创建一个新的归约运算

- 释放自定义的归约操作：**MPI_OP_FREE**
  - 自定义的归约运算不再需要时，可以将其释放，以释放其占用的资源

MPI_REDUCE

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

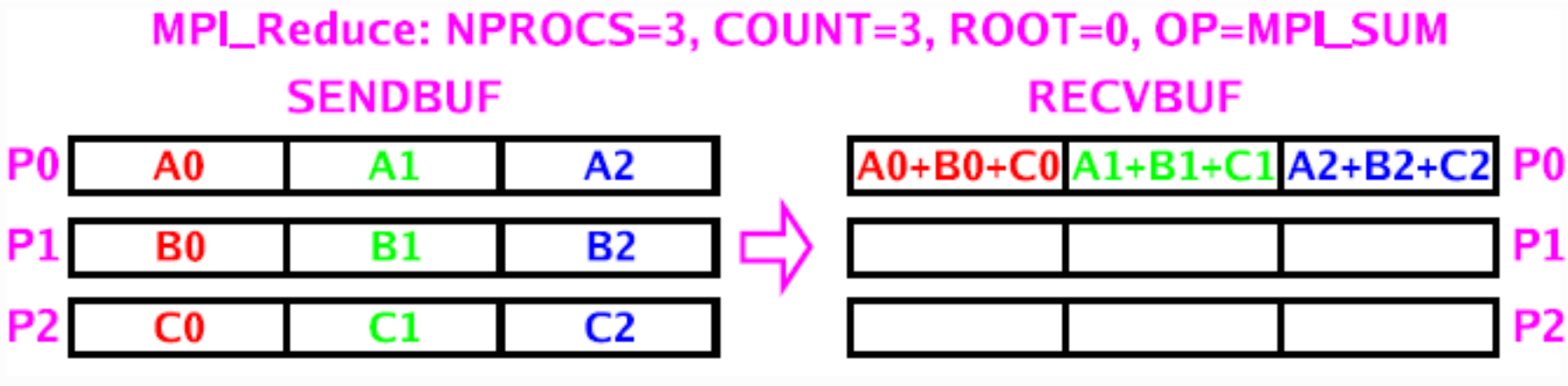
- 参数 **recvbuf** 只对根进程有意义
- 所有进程所提供的数据长度相同、类型相同
- 参数 **op** 用来指定归约所使用的运算，可以是 MPI 预定义的，也可以是用户自行定义的，但必须满足结合律

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

F77	`MPI_REDUCE(SENDBUF,RECVBUF,COUNT,DATATYPE,`
	`  OP,ROOT,COMM,IERR)`
	`<type>  SENDBUF(*), RECVBUF(*)`
	`INTEGER  COUNT,DATATYPE,OP,ROOT,COMM,IERR`

MPI_REDUCE

MPI_REDUCE



归约运算

- MPI 中预定义的运算操作（op）

操作名	含义	操作名	含义
MPI_MAX	求最大	MPI_LXOR	逻辑异或
MPI_MIN	求最小	MPI_BAND	二进制按位与
MPI_SUM	求和	MPI_BOR	二进制按位或
MPI_PROD	求积	MPI_BXOR	二进制按位异或
MPI_LAND	逻辑与	MPI_MAXLOC	求最大值和所在位置
MPI_LOR	逻辑或	MPI_MINLOC	求最小值和所在位置

归约运算

- 每种归约运算操作所允许的数据类型

运算操作 OP	允许的数据类型
MPI_MAX, MPI_MIN	整型和实型
MPI_SUM, MPI_PROD	整型、实型和复型
MPI_LAND, MPI_LOR, MPI_LXOR	C的整型和Fortran的逻辑型
MPI_BAND, MPI_BOR, MPI_BXOR	整型和二进制型(MPI_BYTE)

- MPI_MAXLOC 和 MPI_MINLOC 是两个特殊的运算，它们不仅求最值，而且还会记下最值的位置，所以需要一种特殊的数据类型，即数对。

规约举例

- 修改计算 pi 的程序，使用 **MPI_REDUCE**

```
sum = 0.0;
for (i=myid; i<n; i+=np)
{
    x = h*(i+0.5);
    sum += f(x);
}
mypi = h*sum;

MPI_Barrier(MPI_COMM_WORLD);

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

自定义归约操作

MPI_OP_CREATE(func, commute, op)

参数	IN func 用户自定义的函数 IN commute 自定义的运算是否满足交换律 OUT op 归约运算操作名
C	`int MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *op)`

F77	`MPI_OP_CREATE(FUNC, COMMUTE, OP, IERR)`
	`EXTERNAL FUNC`
	`LOGICAL COMMUTE`
	`INTEGER OP, IERR`

- 参数中 **func** 是用户提供的用于完成该运算的外部函数名
- **MPI_OP_CREATE** 将自定义函数 **func** 和 **op** 联系起来

自定义归约操作

- MPI 对用户自定义的外部函数 **func** 有严格要求，必须是具有如下形式的接口：

C	`void func(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype)`

F77	`FUNCTION FUNC(INVEC,INOUTVEC,LEN,DATATYPE)`
	`<type> INVEC(LEN), INOUTVEC(LEN)`
	`INTEGER LEN, DATATYPE`

† **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个数，**datatype** 指出归约对象的数据类型，函数的返回结果保存在 **inoutvec** 中。

MPI_OP_FREE

MPI_OP_FREE(op)

- 当一个用户自定义的运算不再需要时，可以将其释放，以释放其占用的系统资源

C	`int MPI_Op_free(MPI_Op *op)`
F77	`MPI_OP_FREE(OP, IERR)` `INTEGER  OP, IERR`

其它归约操作

- 全归约：**MPI_ALLREDUCE**
  - 作用相当于在 **MPI_REDUCE** 后再将结果进行一次广播
- 归约散发：**MPI_REDUCE_SCATTER**
  - 将归约结果分散给所有的进程（每个进程存储部分结果）
- 扫描：**MPI_SCAN**
  - 可以看作是一种特殊的归约，每一个进程都对排在它前面的进程进行归约操作，操作结束后，第 **i** 个进程中的 **recvbuf** 中将包含前 **i** 个进程的归约结果

全归约

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

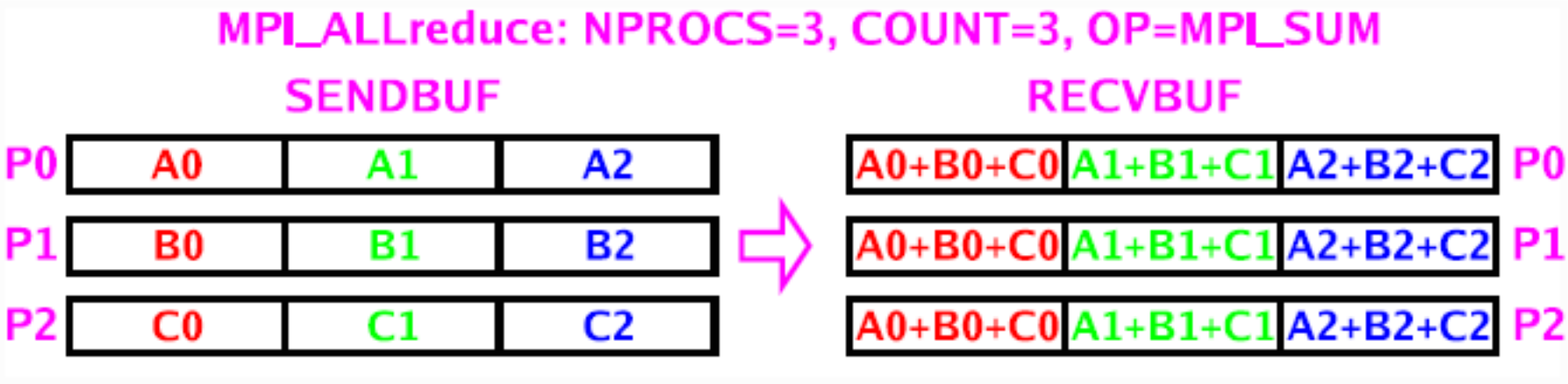
C	`int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

F77	`MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERR)`
	`<type> SENDBUF(*), RECVBUF(*)`
	`INTEGER COUNT, DATATYPE, OP, COMM, IERR`

- 所有进程的 **recvbuf** 将同时获得归约运算的结果

全归约

MPI_ALLREDUCE



归约散发

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

C

```
int MPI_Reduce_scatter(void *sendbuf,
            void *recvbuf, int *recvcounts,
            MPI_Datatype datatype,
            MPI_Op op, MPI_Comm comm)
```

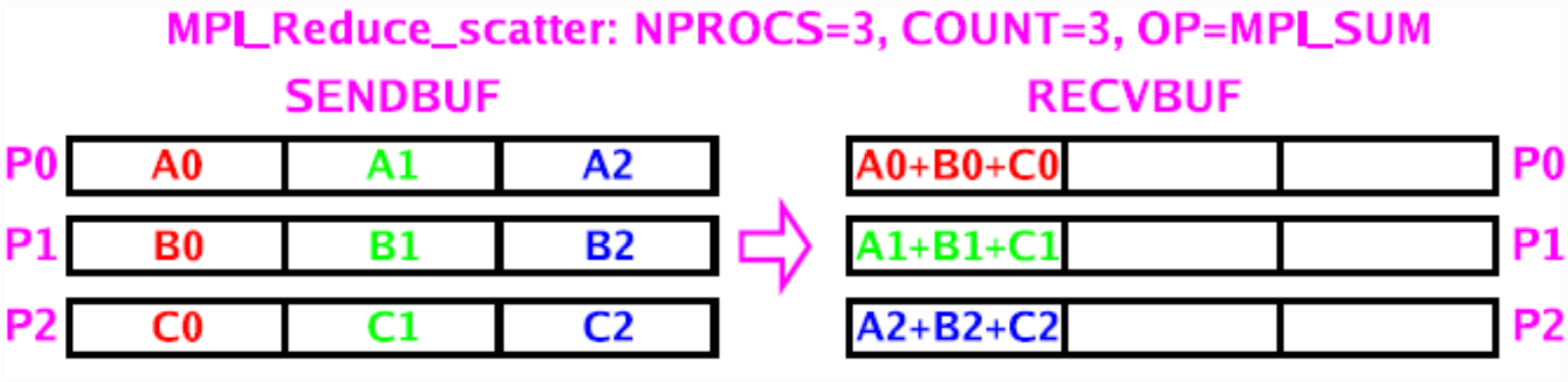
F77

```
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF,
                   RECVCOUNTS, DATATYPE,
                   OP, COMM, IERROR)
<type>   SENDBUF(*), RECVBUF(*)
INTEGER  DATATYPE,OP,COMM,IERR,RECVCOUNTS(*)
```

- **MPI_REDUCE_SCATTER** 对由 **sendbuf**, **count** 和 **datatype** 指定的数据进行归约操作，这里 **count**=$\sum_{i=1}^{np}$recvcounts(***i***)
- 然后将归约结果散发给所有的进程，其中进程 **i** 获得的数据块长度为 **recvcounts(i)**

归约散发

MPI_REDUCE_SCATTER



扫描

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

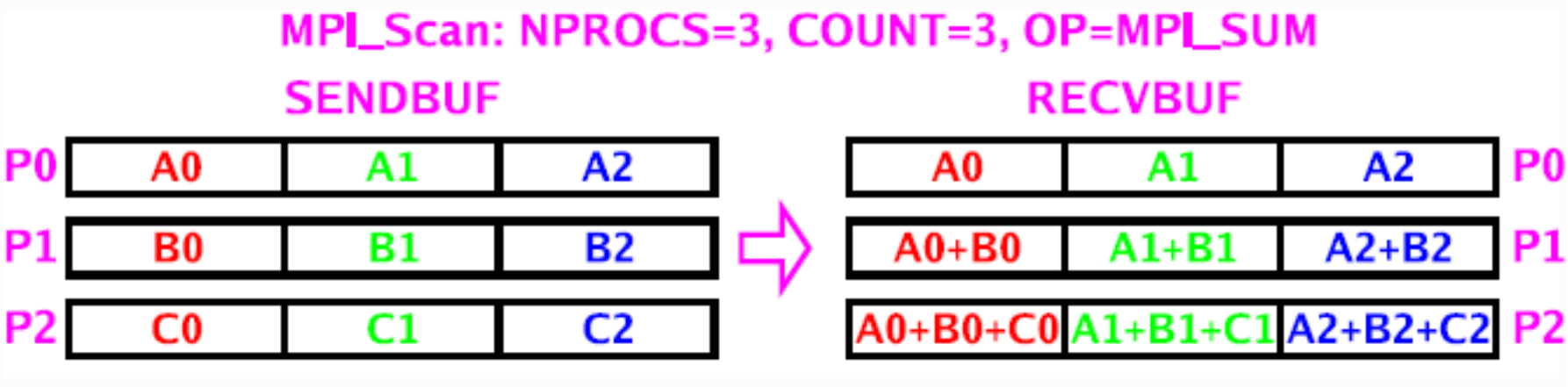
C	`int MPI_Scan(void *sendbuf, void *recvbuf,`
	`  int count, MPI_Datatype datatype,`
	`  MPI_Op op, MPI_Comm comm)`

F77	`MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE,`
	`  OP, COMM, IERR)`
	`<type>  SENDBUF(*), RECVBUF(*)`
	`INTEGER  COUNT, DATATYPE, OP, COMM, IERR`

- 每一个进程都对排在它前面的进程进行归约操作，操作结束后，第 **i** 个进程中的 **recvbuf** 中将包含前 **i** 个进程的归约结果
- **0** 号进程 接收缓冲区中的数据就是其发送缓冲区的数据

扫描

MPI_SCAN


MPI_Scan: NPROCS=3, COUNT=3, OP=MPI_SUM
SENDBUF
RECVBUF
P0
A0
A1
A2
P1
B0
B1
B2
P2
C0
C1
C2
A0
A1
A2
P0
A0+B0
A1+B1
A2+B2
P1
A0+B0+C0
A1+B1+C1
A2+B2+C2
P2

本讲函数小结

点对点	MPI_SEND、MPI_ISEND、MPI_BSEND、MPI_IBSEND、MPI_SSEND、MPI_ISSEND、MPI_RSEND、MPI_IRSEND、MPI_RECV、MPI_IRECV、MPI_PROBE、MPI_IPROBE、MPI_WAIT、MPI_TEST、MPI_WAITALL、MPI_TESTALL、MPI_WAITANY、MPI_TESTANY、MPI_WAITSOME、MPI_TESTSOME、MPI_REQUEST_FREE、MPI_CANCEL、MPI_TEST_CANCELLED
持久通信	MPI_SEND_INIT、MPI_RECV_INIT、MPI_START、MPI_STARTALL
一对多	MPI_BCAST、MPI_SCATTER、MPI_SCATTERV
多对一	MPI_GATHER、MPI_GATHERV、MPI_REDUCE
多对多	MPI_ALLGATHER、MPI_ALLGATHERV、MPI_ALLTOALL、MPI_ALLTOALLV、MPI_ALLREDUCE、MPI_REDUCE_SCATTER、MPI_SCAN
其它	MPI_BARRIER