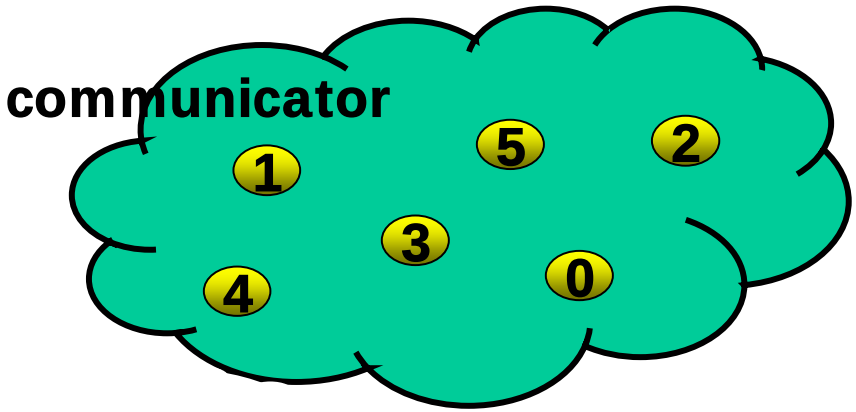
MPI并行编程入门

中国科学院计算机网络信息中心
超级计算中心

聚合通信

- 定义
- 三种通信方式
- 聚合函数列表
- 同步
- 广播
- 收集
- 散发
- 全散发收集
- 归约

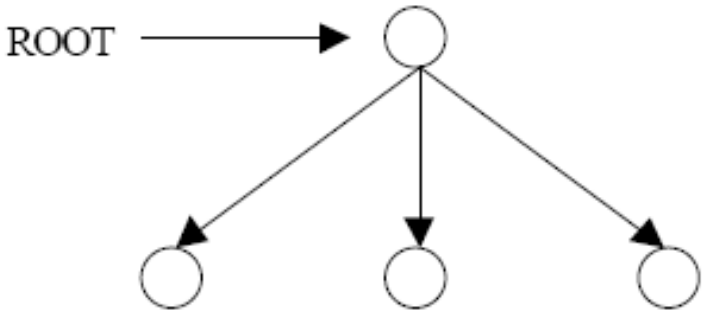
定义

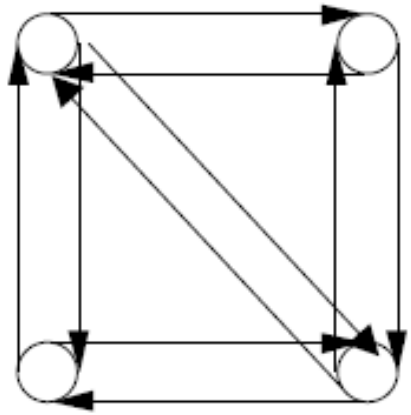


- 一个通信器的所有进程参与，所有进程都调用聚合通信函数
- MPI系统保证聚合通信函数与点对点调用不会混淆
- 聚合通信不需要消息标号
- 聚合通信函数都为阻塞式函数
- 聚合通信的功能：通信、同步、计算等

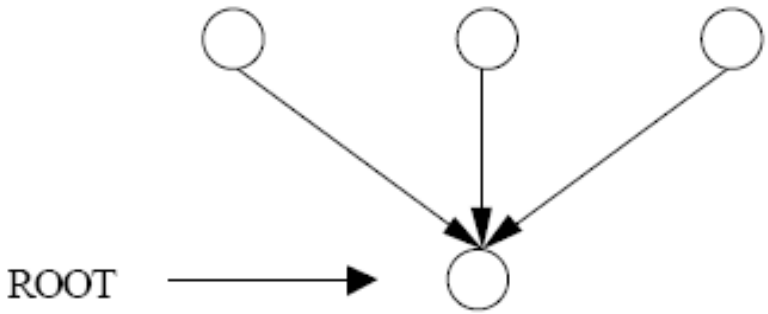
三种通信方式

一对多





多对一



多对多

聚合函数列表

- MPI_Barrier
- MPI_Bcast
- MPI_Gather/MPI_Gatherv
- MPI_Allgather/MPI_Allgatherv
- MPI_Scatter/MPI_Scatterv
- MPI_Alltoall/MPI_Alltoallv
- MPI_Reduce/MPI_Allreduce/MPI_Reduce_scatter
- MPI_Scan

同步

C

```
int MPI_Barrier(MPI_Comm comm )
```

Fortran 77

```
MPI_BARRIER(COMM, IERR)
INTEGER COMM, IERR
```

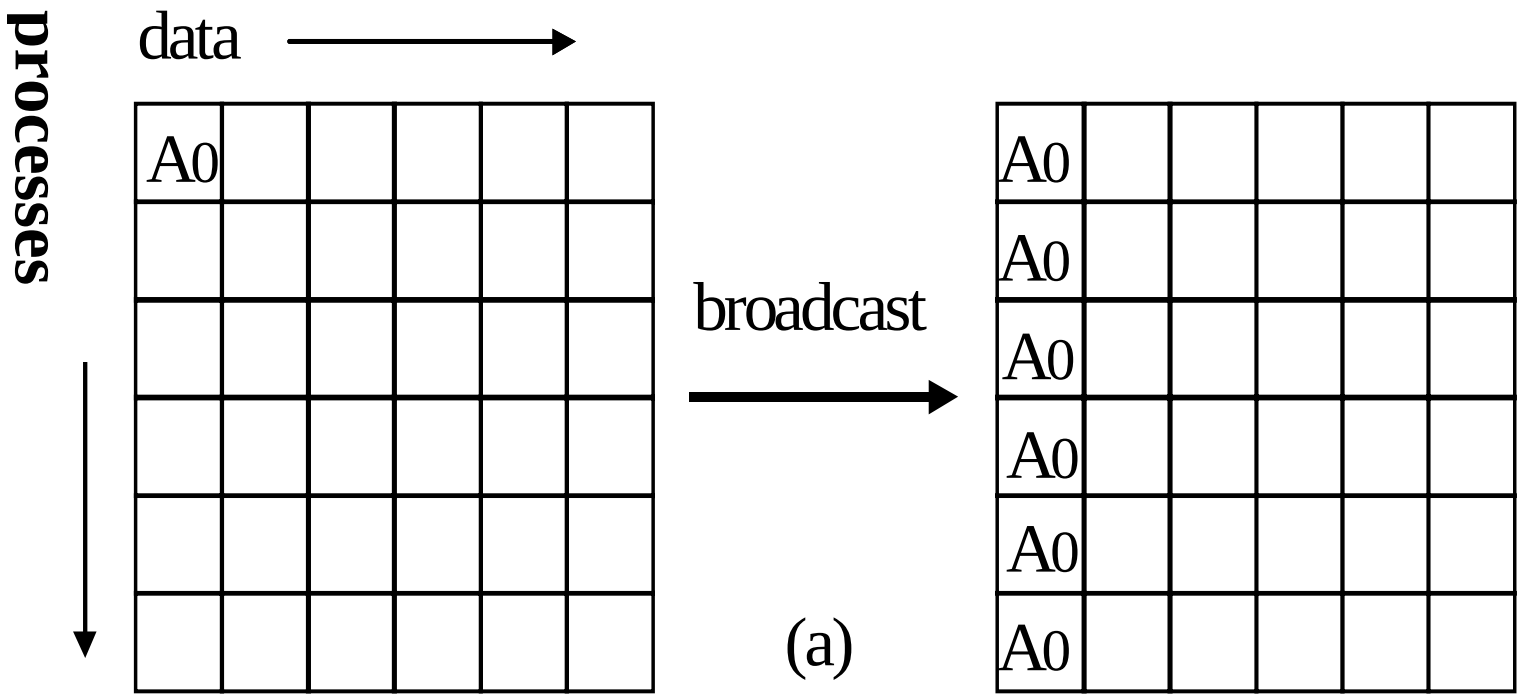
该函数用于进程同步，即一个进程调用该函数后需等待通信器内所有进程调用该函数后返回

Sample - Fortran

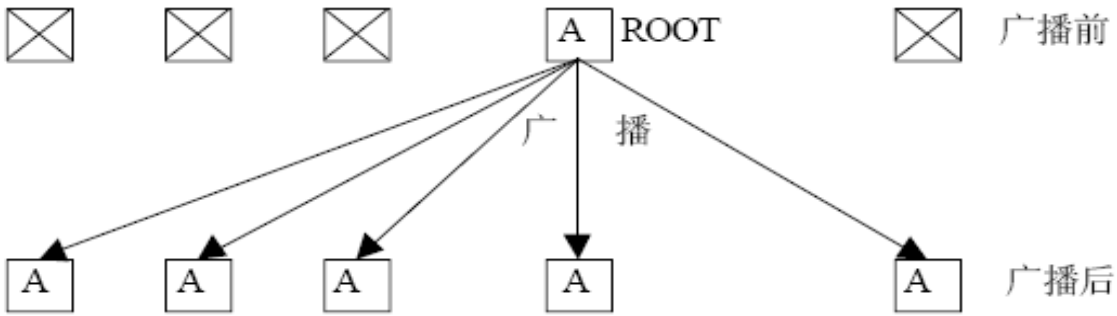
```
……
CALL MPI_COMM_RANK(COMM,RANK,IERR)
IF(RANK.EQ.0) THEN
   CALL WORK0(……)
ELSE
   CALL WORK1(……)
CALL MPI_BARRIER(COMM,IERR)
CALL WORK2(……)
……
```

```
……
CALL MPI_COMM_RANK(COMM,RANK,IERR)
IF(RANK.EQ.0) THEN
   CALL WORK0(……)
   CALL MPI_BARRIER(COMM,IERR)
ELSE
   CALL WORK1(……)
CALL WORK2(……)
……
```

广播



广播



C

```
int MPI_Bcast(void *buffer, int count,
        MPI_Datatype datatype, int root, MPI_Comm comm )
```

Fortran 77

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

广播

- 通信器中**root**进程将自己**buffer**内的数据发给通信器内所有进程
- 非**root**进程用自己的**buffer**接收数据

```
IF ( MYRANK .EQ. ROOT ) THEN
   DO I=0, NPROCS-1
      IF ( I .NE. ROOT ) THEN
         CALL MPI_SEND(BUFFER, COUNT, DATATYPE, I, ...
      ENDIF
   ENDDO
ELSE
   CALL MPI_RECV(BUFFER, COUNT, DATATYPE, ROOT, ...
ENDIF
```

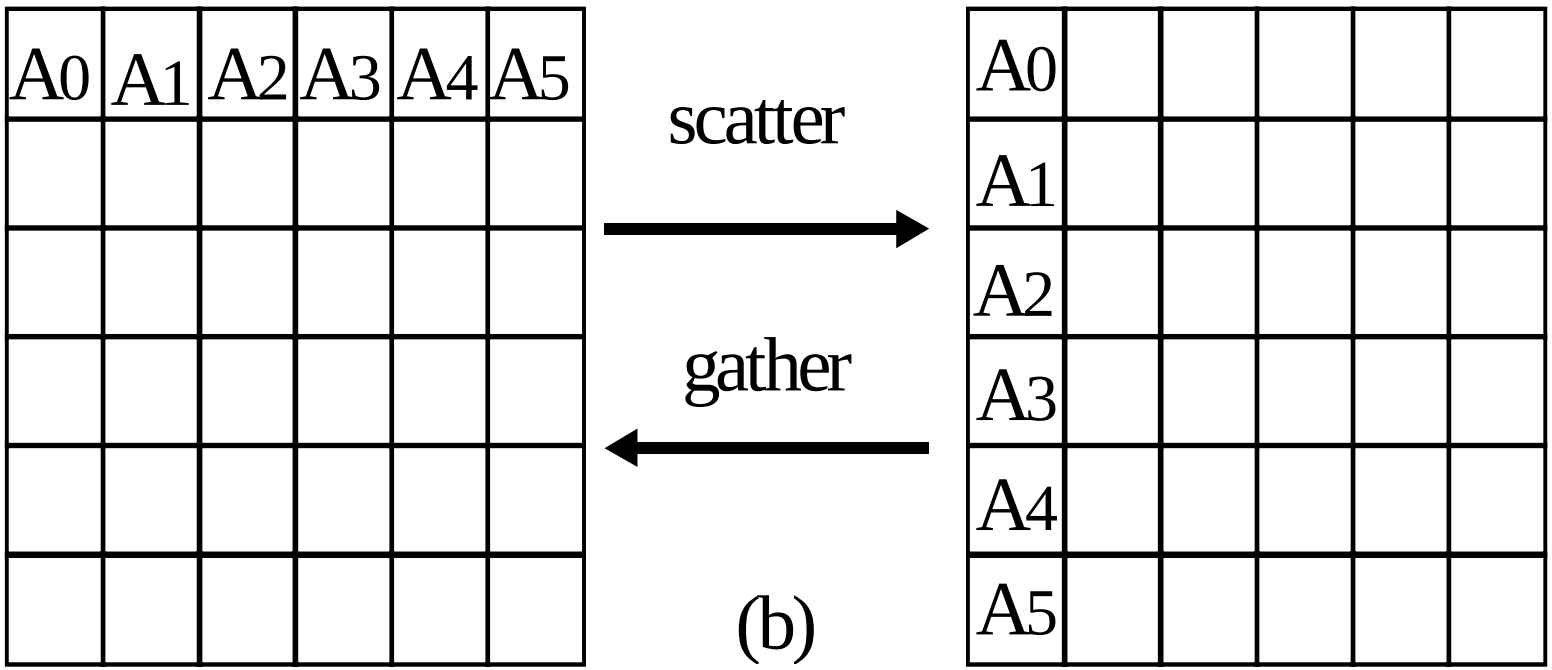
Sample - C

```
#include<mpi.h>

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

    int rank;

    double param;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    if(rank==5) param=23.0;

    MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);

    printf("P:%d after broadcast parameter is %f\n",rank,param);

    MPI_Finalize();

}
```

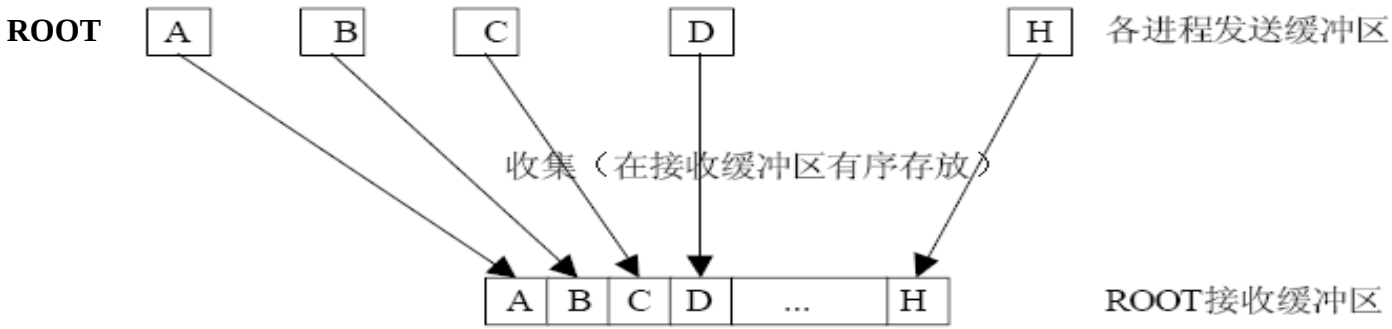
Program Output

```
P:0 after broadcast parameter is 23.000000
P:6 after broadcast parameter is 23.000000
P:5 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000
P:7 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:4 after broadcast parameter is 23.000000
```

收集&散发



收集 (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)
```

Fortran 77

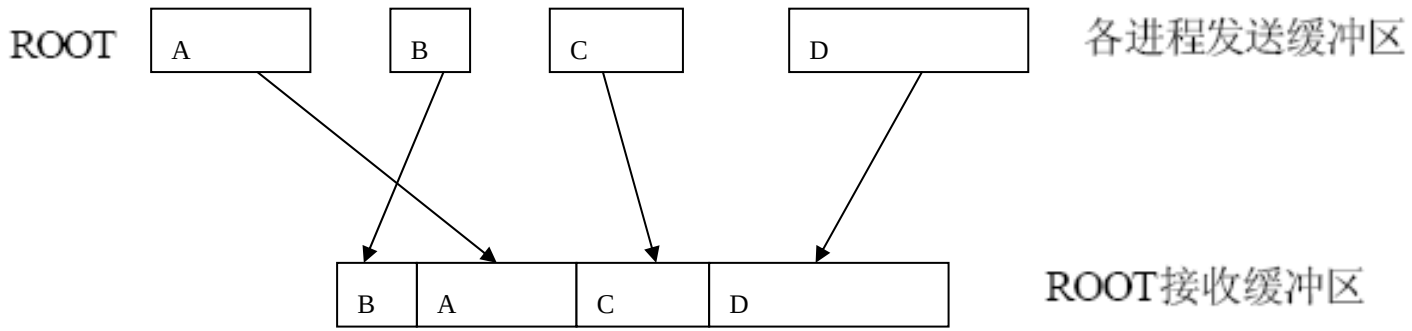
```
 MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+        RECVCOUNT, RECVTYPE, ROOT, COMM, IERR)
 <type>  SENDBUF(*), RECVBUF(*)
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
+        COMM, IERR
```

收集(MPI_Gather)

- 所有进程（包括根进程）将**sendbuf**的数据传输给根进程；根进程按着进程号顺序依次接收到**recvbuf**
- 发送与接收的数据类型相同；**sendcount**和**recvcount**相同
- 非根进程接收消息缓冲区被忽略，但需要提供

```
 CALL MPI_SEND(SENDBUF, SENDCOUNT, SENDTYPE, ROOT, ...)
 IF ( MYRANK .EQ. ROOT ) THEN
    DO I=0, NPROCS-1
       CALL MPI_RECV( RECVBUF + I*RECVCOUNT*extent(RECVTYPE),
+                     RECVCOUNT, RECVTYPE, I, ...)
    ENDDO
 ENDIF
```

收集(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)
```

Fortran 77

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

收集(MPI_Gatherv)

- 每个进程发送的数据个数不同
- 根进程接收不一定连续存放
- 数组**recvcounts**和**displs**的元素个数等于进程总数,并与进程顺序对应
- 数据的个数与位移都以**recvtype**为单位

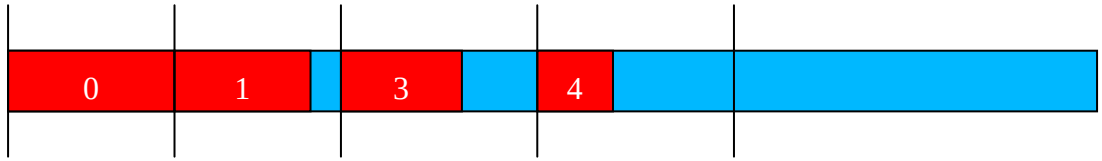
```
INTEGER DISPLS(0:NPROCS-1), RECVCOUNTS(0:NPROCS-1)
... ...
CALL MPI_SEND(SENDBUF, SENDCOUNT, SENDTYPE, ROOT, ...)
IF ( MYRANK .EQ. ROOT ) THEN
   DO I=0, NPROCS-1
      CALL MPI_RECV( RECVBUF + DISPLS(I)*extent(RECVTYPE),
+                    RECVCOUNTS(I), RECVTYPE, I, ...)
   ENDDO
ENDIF
```

Sample – Fortran

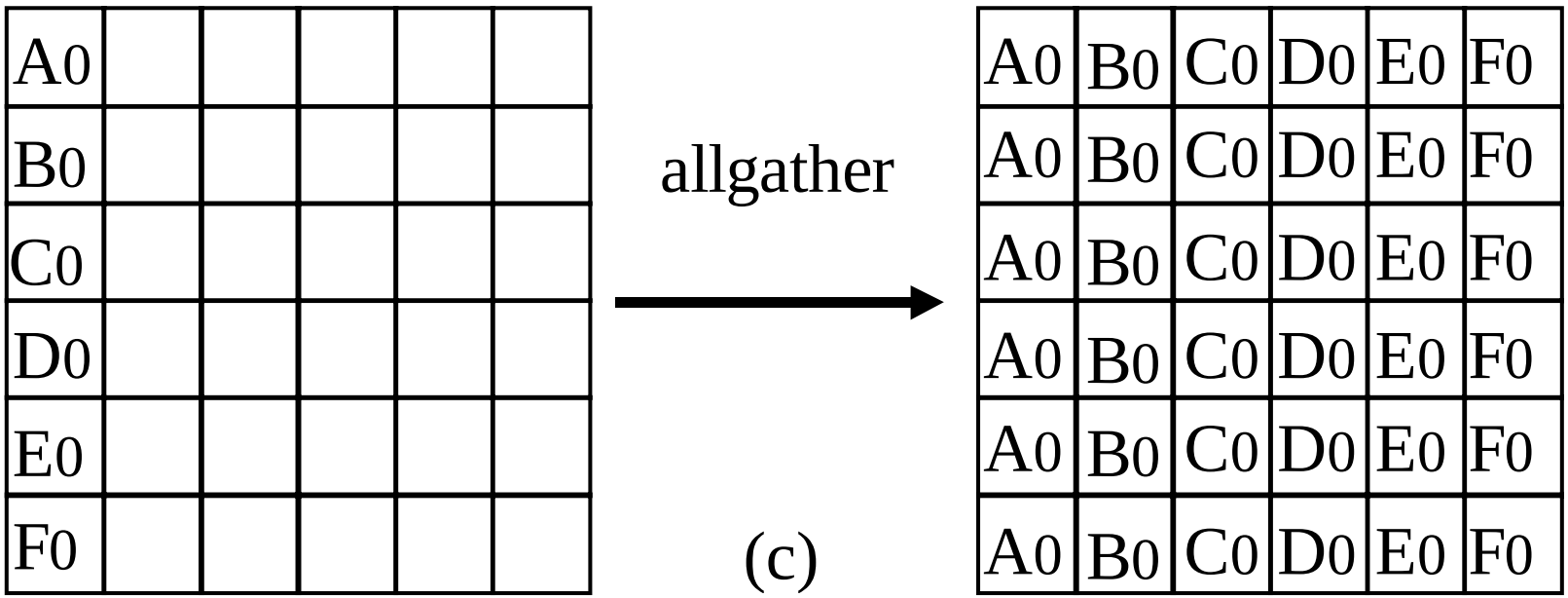
```
……
INTEGER A(100),RBUF(10000),SIZE,ROOT,RTYPE,RANK,RECS(100),DISP(100)
CALL MPI_COMM_SIZE(COMM,SIZE,IERR)
CALL MPI_COMM_RANK(COMM,RANK,IERR)
IF(100*SIZE .GT. 10000) THEN
   PRINT*, “NOT ENOUGH RECEIVING BUF”
   CALL MPI_FINALIZE(IERR)
ELSE
   ROOT=0
   IF(RANK.EQ.0) THEN
      DO I=0,SIZE-1
         RECS(I)=100-I
         DISP(I)=I*100
      ENDDO
   ENDIF
   CALL MPI_GATHERV(A,100-RANK,MPI_INTEGER,RBUF,RECS,DISP,MPI_INTERGER,ROOT,COMM,IERR)
ENDIF
……
```

进程 i 向进程 0 发送 100-i 个整型数，

每隔100个整型数依次存储消息

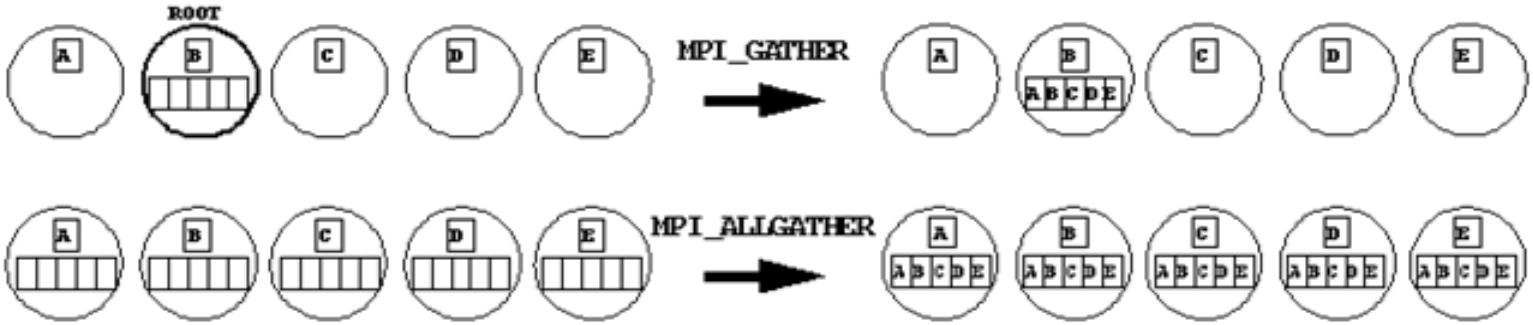


收集(MPI_Allgather)



(c)

收集(MPI_Allgather)



C

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

Fortran 77

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

收集(MPI_Allgather)

MPI_Allgather 等价于依次以每个进程为根进程调用 NPROCS 次普通数据收集函数 MPI_Gather:

```
 DO I=0, NPROCS-1
    CALL MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+                   RECVCOUNT, RECVTYPE, I, COMM, IERR)
 ENDDO
```

也可以认为 MPI_Allgather 相当于以任一进程为根进程调用一次普通收集, 紧接着再对收集到的数据进行一次广播, 例如:

```
 ROOT=0
 CALL MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+                RECVCOUNT, RECVTYPE, ROOT, COMM, IERR)
 CALL MPI_BCAST(RECVBUF, RECVCOUNT*NPROCS, RECVTYPE, ROOT,
+               COMM, IERR)
```

收集(MPI_Allgatherv)

C

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

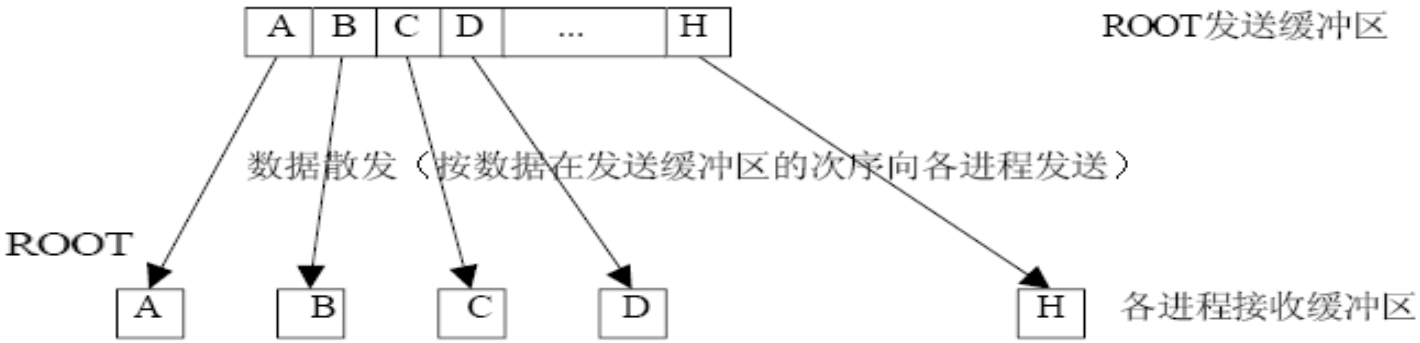
Fortran 77

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

MPI_Allgatherv 等价于依次以每个进程为根进程调用 NPROCS 次 MPI_Gatherv:

```
DO I=0, NPROCS-1
   CALL MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+           RECVCOUNTS, DISPLS, RECVTYPE, I, COMM, IERR)
ENDDO
```

散发(MPI_Scatter)



C

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

Fortran 77

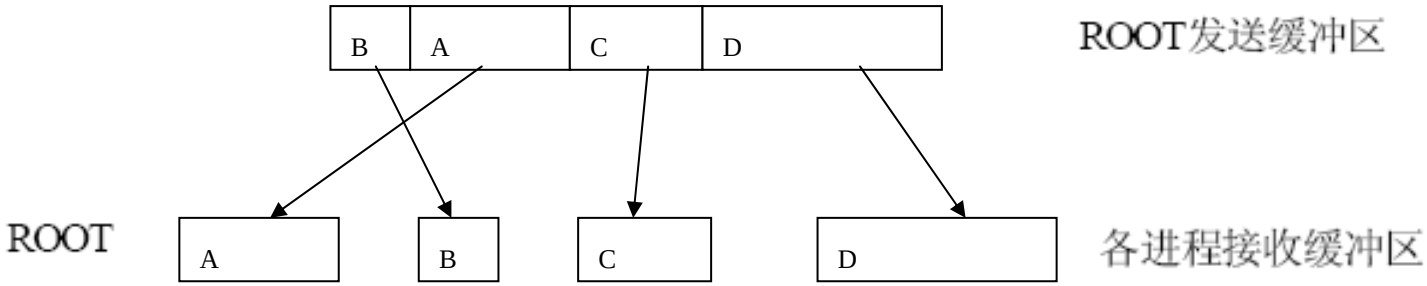
```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+        RECVCOUNT, RECVTYPE, ROOT, COMM, IERR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
+        COMM, IERR
```

散发(MPI_Scatter)

- 根进程有**np**个数据块，每块包含**sendcount**个类型为**sendtype**的数据；根进程将这些数据块按着进程号顺序依次散发到各个进程（包含根进程）的**recvbuf**
- 发送与接收的数据类型相同；**sendcount**和**recvcount**相同
- 非根进程发送消息缓冲区被忽略，但需要提供

```
 IF ( MYRANK .EQ. ROOT ) THEN
    DO I=0, NPROCS-1
       CALL MPI_SEND( SENDBUF + I*SENDCOUNT*extent(SENDTYPE),
+                     SENDCOUNT, SENDTYPE, I, ...)
    ENDDO
 ENDIF
 CALL MPI_RECV(RECVBUF, RECVCOUNT, RECVTYPE, ROOT, ...)
```

散发(MPI_Scatterv)



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)
```

Fortran 77

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

散发(MPI_Scatterv)

- 根进程向各个进程发送的数据个数不等
- 根进程散发各个进程的数据，其缓存区不一定连续
- 数组**recvcounts**和**displs**的元素个数等于进程总数,并与进程顺序对应
- 数据的个数与位移都以**sendtype**为单位

```
 INTEGER DISPLS(0:NPROCS-1), SENDCOUNTS(0:NPROCS-1)
 ... ...
 IF ( MYRANK .EQ. ROOT ) THEN
    DO I=0, NPROCS-1
       CALL MPI_SEND( SENDBUF + DISPLS(I)*extent(SENDTYPE),
+                     SENDCOUNTS(I), SENDTYPE, I, ...)
    ENDDO
 ENDIF
 CALL MPI_RECV(RECVBUF, RECVCOUNT, RECVTYPE, ROOT, ...)
```

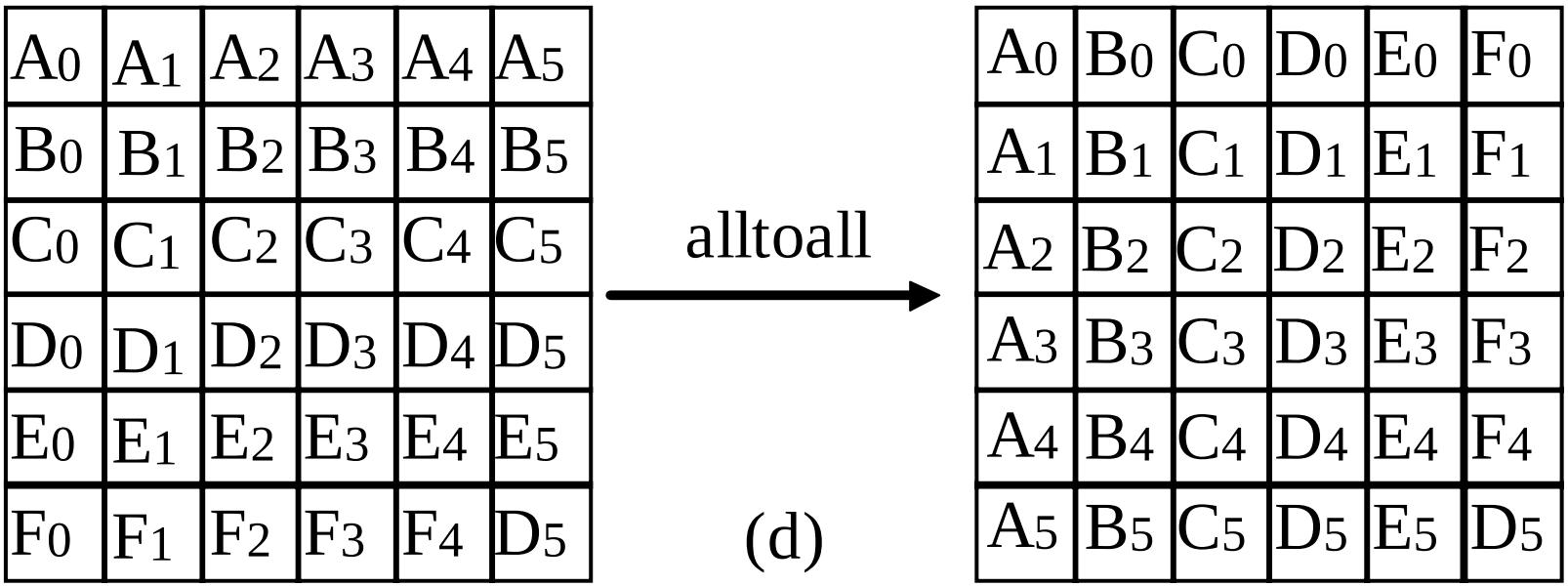
Sample - C

```
#include <mpi.h>
int main (int argc, char *argv[])
{
   int rank,size,i,j;
   double param[400],mine;
   int sndcnt,revcnt;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   revcnt=1;
   if(rank==3)
   {
     for(i=0;i<size;i++) param[i]=23.0+i;
     sndcnt=1;
   }
   MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,revcnt,MPI_DOUBLE,3,MPI_COMM_WORLD);
   printf("P:%d mine is %f\n",rank,mine);
   MPI_Finalize();
 }
```

根进程向所有进程次序分发1个数组元素

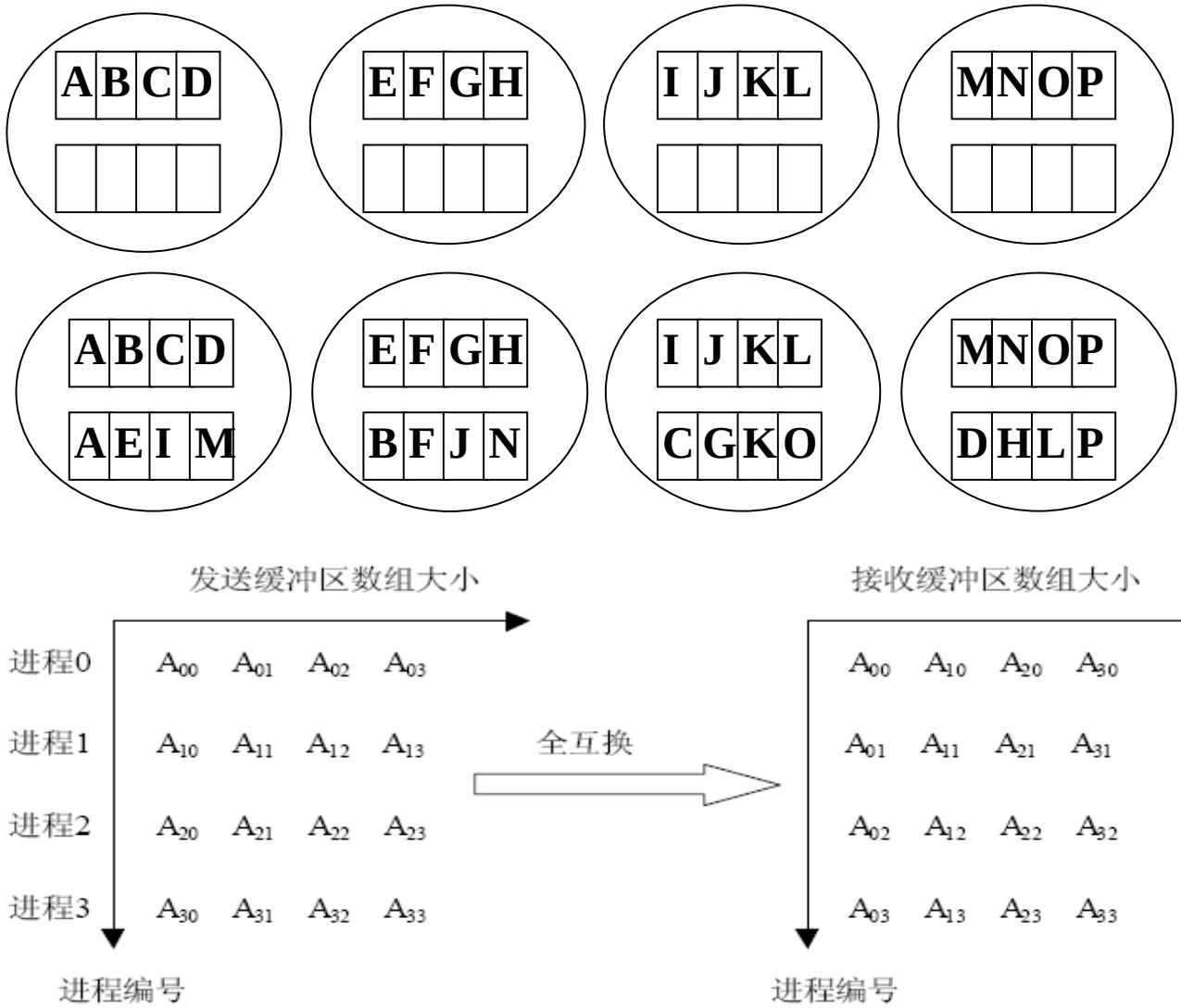
```
            Program Output
P:0 mine is 23.000000
P:1 mine is 24.000000
P:2 mine is 25.000000
P:3 mine is 26.000000
```

全散发收集(MPI_Alltoall)



(d)

全散发收集(MPI_Alltoall)



全散发收集(MPI_Alltoall)

C

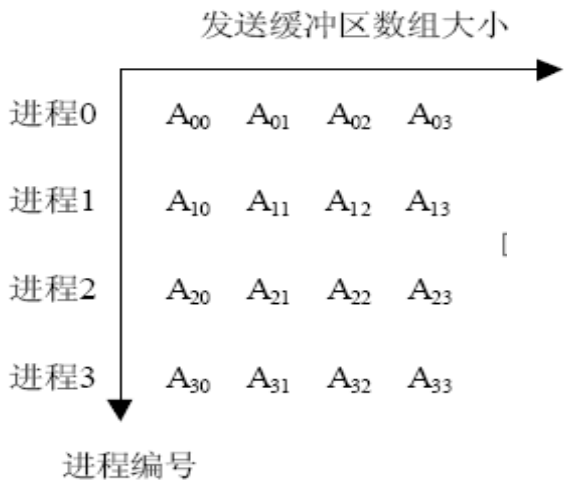
```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype,
        MPI_Comm comm)
```

Fortran 77

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

```
Do I=0,NPROCS-1
  CALL MPI_SCATTER(SENDBUF(I), SENDCOUNT, SENDTYPE,
  + RECVBUF+I*RECVCOUNT*extent(RECVTYPE), RECVCOUNT, RECVTYPE, I, COMM, IERR)
ENDDO
```

全散发收集(MPI_Alltoallv)



- 任意行散发，参照**sdispls**
- 任意列收集，参照**rdispls**

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)
```

Fortran 77

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

全散发收集(MPI_Alltoallv)

- 每个进程如同根进程一样，执行一次**MPI_Scatterv**发送

```
Do I=0,NPROCS-1
    CALL MPI_SCATTERV(SENDBUF(I), SENDCOUNTS, SDISPLS, SENDTYPE,
  +   RECVBUF+RDISPLS(I)*extent(RECVTYPE), RECVCOUNTS(I), RECVTYPE, I,…)
ENDDO
```

- 每个进程如同根进程一样，执行一次**MPI_Gatherv**接收

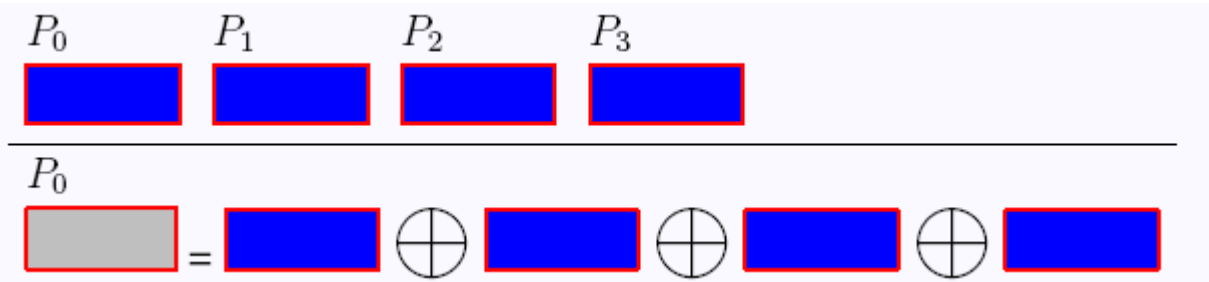
```
Do I=0,NPROCS-1
    CALL MPI_GATHERV(SENDBUF+SDISPLS(I)*extent(RECVTYPE), SENDCOUNTS(I),
  +    SENDTYPE, RECVBUF(I), RECVCOUNTS, RDISPLS, RECVTYPE, I,…)
ENDDO
```

归约

假设一个通信器中有 p 个进程, 每个进程均有一个 n 个元素的数组. 设 $\{a_{k,i}, k = 1, \ldots, n\}$ 为第 i 个进程中的数组, $i = 0, \ldots, p-1$. 又设 $\oplus$ 为关于这些数组元素的二目运算, 则相应的归约操作 (reduction) 结果定义为数组 $\{\mathrm{res}_k, k = 1, \ldots, n\}$, 其中:

$$\mathrm{res}_k = a_{k,0} \oplus a_{k,1} \cdots \oplus a_{k,p-1},$$

MPI 的归约函数要求运算 $\oplus$ 满足结合律, 但可以不满足交换律. 运算可以是 MPI 预定义的, 也可以是用户自行定义的.



归约(MPI_Reduce)

C

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

Fortran 77

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

- 各进程提供数据(sendbuf,count,datatype)
- 归约结果存放在root进程的缓冲区recvbuf

设进程数为 NPROCS, 则 MPI_Reduce 相当于在根进程 (root) 中计算:

```
DO K=1, COUNT
   RECVBUF(K) = SENDBUF(K) of process 0
   DO I=1, NPROCS-1
      RECVBUF(K) = RECVBUF(K) op ( SENDBUF(K) of process I )
   ENDDO
ENDDO
```

归约

MPI 预定义的运算及对数据类型的限制

MPI 运算符	含义	允许的数据类型	
		C	Fortran 77
MPI_MAX	求最大	整型, 实型	整型, 实型
MPI_MIN	求最小	整型, 实型	整型, 实型
MPI_SUM	求和	整型, 实型, 复型	整型, 实型, 复型
MPI_PROD	求积	整型, 实型, 复型	整型, 实型, 复型
MPI_LAND	逻辑与	整型	逻辑型
MPI_BAND	二进制按位与	整型, 字节	整型, 字节
MPI_LOR	逻辑或	整型	逻辑型
MPI_BOR	二进制按位或	整型, 字节	整型, 字节
MPI_LXOR	逻辑异或	整型	逻辑型
MPI_BXOR	二进制按位异或	整型, 字节	整型, 字节
MPI_MAXLOC	最大值及位置	*	*
MPI_MINLOC	最小值及位置	*	*

归约

MPI_MINLOC 和 MPI_MAXLOC 是两个特殊的运算, 它们要求由数对 (连续存放的两个数) 构成的一类特殊数据类型. MPI 为它们定义了下面一些数据类型:

C

$$\texttt{MPI\_FLOAT\_INT} = \{\texttt{float}, \texttt{int}\}$$

$$\texttt{MPI\_DOUBLE\_INT} = \{\texttt{double}, \texttt{int}\}$$

$$\texttt{MPI\_LONG\_INT} = \{\texttt{long}, \texttt{int}\}$$

$$\texttt{MPI\_2INT} = \{\texttt{int}, \texttt{int}\}$$

$$\texttt{MPI\_SHORT\_INT} = \{\texttt{short}, \texttt{int}\}$$

$$\texttt{MPI\_LONG\_DOUBLE\_INT} = \{\texttt{long double}, \texttt{int}\}$$

Fortran 77

$$\texttt{MPI\_2REAL} = \{\texttt{REAL}, \texttt{REAL}\}$$

$$\texttt{MPI\_2DOUBLE\_PRECISION} = \{\texttt{DOUBLE PRECISION}, \texttt{DOUBLE PRECISION}\}$$

$$\texttt{MPI\_2INTEGER} = \{\texttt{INTEGER}, \texttt{INTEGER}\}$$

归约

设 $x = (u, i)$, $y = (v, j)$, 则 MPI_MINLOC$(x, y) = (w, k)$, 其中:

$$w = \min(u, v), \qquad k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

设 $x = (u, i)$, $y = (v, j)$, 则 MPI_MAXLOC$(x, y) = (w, k)$, 其中:

$$w = \max(u, v), \qquad k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

Sample - C

```
#include <mpi.h>
/* Run with 16 processes */
int main (int argc, char *argv[])
{
  int rank, root=7;
  struct
  {
    double value;
    int rank;
  } in, out;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  in.value=rank+1;
  in.rank=rank;
  MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
  if(rank==root) printf("P :%d max=%lf at rank %d\n",rank,out.value,out.rank);
  MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MINLOC,root,MPI_COMM_WORLD);
  if(rank==root) printf("P :%d min=%lf at rank %d\n",rank,out.value,out.rank);
  MPI_Finalize();
}
```

数对的归约操作

(1.000000,0) (2.000000,1)…(16.000000,15)

```
        Program Output
P:7 max = 16.000000 at rank 15
P:7 min = 1.000000 at rank 0
```

Sample - Fortran

```
      PROGRAM MaxMin
C     Run with 8 processes
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      integer in(2),out(2)
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
      CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
      in(1)=rank+1
      in(2)=rank
      call MPI_REDUCE(in,out,1,MPI_2INTEGER,MPI_MAXLOC, 7,MPI_COMM_WORLD,err)
      if(rank.eq.7) print *,"P:",rank," max=",out(1)," at rank ",out(2)
      call MPI_REDUCE(in,out,1,MPI_2INTEGER,MPI_MINLOC, 2,MPI_COMM_WORLD,err)
      if(rank.eq.2) print *,"P:",rank," min=",out(1)," at rank ",out(2)
      CALL MPI_FINALIZE(err)
      END
```

```
     Program Output
P:2 min=1 at rank 0
P:7 max=8 at rank 7
```

全归约(MPI_Allreduce)

C

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

Fortran 77

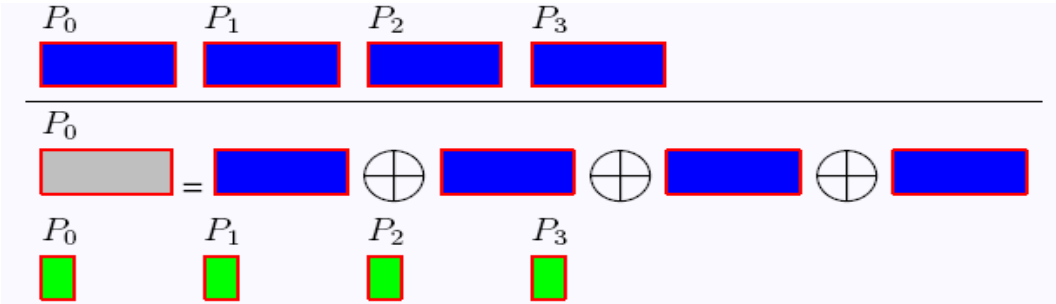
```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,
+        COMM, IERR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERR
```

全归约函数与普通归约函数的操作类似, 但所有进程将同时获得归约运算的结果. MPI_Allreduce 除了比 MPI_Reduce 少了一个 root 参数外, 其它参数及含义与后者一样.

MPI_Allreduce 相当于在 MPI_Reduce 后马上再将结果进行一次广播, 因此它等价于:

```
ROOT=0
CALL MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT,
+        COMM, IERR)
CALL MPI_BCAST(RECVBUF, COUNT, DATATYPE, ROOT, COMM, IERR)
```

归约散发(MPI_Reduce_scatter)



C

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

Fortran 77

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

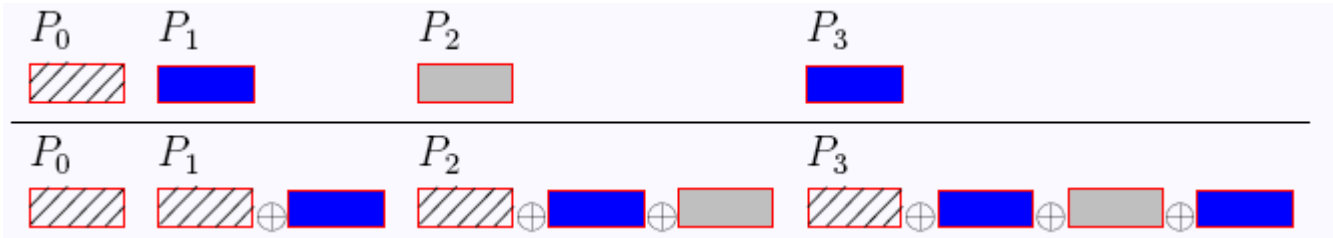
归约散发函数首先进行一次 COUNT $= \sum_{i=0}^{\text{NPROCS}-1}$ recvcounts(i) 的归约操作, 然后再对归约结果进行散发操作, 散发给第 i 个进程的数据块长度为 recvcounts(i). 其余参数的含义与 MPI_Reduce 一样.

归约散发(MPI_Reduce_scatter)

设 NPROCS 为进程数, MYRANK 为进程号, 则 MPI_Reduce_scatter 相当于

```
 INTEGER ROOT,COUNT,DISPLS(0:NPROCS-1),RECVCOUNTS(0:NPROCS-1)
 <type>  TMPBUF(*)

 COUNT = RECVCOUNTS(0)
 DISPLS(0) = 0
 DO I=1, NPROCS-1
    COUNT = COUNT + RECVCOUNTS(I)
    DISPLS(I) = DISPLS(I-1) + RECVCOUNTS(I-1)
 ENDDO
 ROOT=0
 CALL MPI_REDUCE(SENDBUF, TMPBUF, COUNT, DATATYPE, OP, ROOT,
+         COMM, IERR)
 CALL MPI_SCATTERV(TMPBUF, RECVCOUNTS, DISPLS, DATATYPE,
+         RECVBUF, RECVCOUNTS(MYRANK), DATATYPE, ROOT, COMM,
+         IERR)
```

前缀归约(MPI_Scan)



C

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

Fortran 77

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

设进程号为 MYRANK, 则 MPI_Scan 相当于每个进程分别计算:

```
DO K=1, COUNT
   RECVBUF(K) = SENDBUF(K) of process 0
   DO I=1, MYRANK
      RECVBUF(K) = RECVBUF(K) op ( SENDBUF(K) of process I )
   ENDDO
ENDDO
```

归约

- 创建新运算

C

```
int MPI_Op_create(MPI_User_function *func, int commute,
        MPI_Op *op)
```

Fortran 77

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

- func是用户提供的用于完成运算的外部函数
- commute用来指明所定义的运算是否满足交换律
- 一个运算创建后和MPI预定义的运算一样使用

归约

- 用户自定义函数

C

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

Fortran 77

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

- invec与inoutvec分别指出要被归约的数据所在缓冲的首地址
- datatype指出归约对象的数据类型
- len给出了invec与inoutvec中包含的元素个数（相当于归约函数中的count）
- 函数返回时，运算结果储存在inoutvec中

```
DO I=1,LEN
   INOUTVEC(I) = INVEC(I) op INOUTVEC(I)
ENDDO
```

归约

当一个用户定义的运算不再需要时, 可以调用 MPI_Op_free 将其释放, 以便释放它所占用的系统资源.

C

```
int MPI_Op_free(MPI_Op *op)
```

Fortran 77

```
MPI_OP_FREE(OP, IERR)
INTEGER OP, IERR
```

Sample - Fortran

```
      PROGRAM UserOP
C     Run with 8 processes
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      integer source, reslt
      EXTERNAL digit
      LOGICAL commute
      INTEGER myop
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
      CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
      commute= true
      call MPI_OP_CREATE(digit,commute,myop,err)
      source=(rank+1)**2
      call MPI_BARRIER(MPI_COM_WORLD,err)
      call MPI_SCAN(source,reslt,1,MPI_INTEGER,myop,MPI_COMM_WORLD,err)
      print *,"P:",rank," my result is ",reslt
      CALL MPI_OP_FREE(myop,err)
      CALL MPI_FINALIZE(err)
  END
  integer function digit(in,inout,len,type)
      integer len,type
      integer in(len),inout(len)
      do i=1,len
         inout(i)=mod((in(i)+inout(i)),10)
      end do
      digit=5
end
```

source={ 1,4,9,16,25,36,49,64}

```
  Program Output
 P:6 my result is 0
 P:5 my result is 1
 P:7 my result is 4
 P:1 my result is 5
 P:3 my result is 0
 P:2 my result is 4
 P:4 my result is 5
 P:0 my result is 1
```