Parallel Computing for Science & Engineering Spring 2013: MPI collectives 2

Instructors:

Victor Eijkhout, Research Scientist, TACC

Kent Milfeld, Research Associate, TACC

Global Sum Example with MPI_Reduce and MPI_Scatter

- Processor 0 scatters data to everyone
- Everybody reduces back to 0 with a sum

MPI_Reduce Example

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define mpi_root 0

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

        int *myray,*send_ray;
        int i,psum, gsum, nlocal, ntotal;
        int  nrank,irank, ierr;

        MPI_Init(&argc,&argv);
        MPI_Comm_size( MPI_COMM_WORLD, &nrank );
        MPI_Comm_rank( MPI_COMM_WORLD, &irank);
```

MPI_Reduce Example

```
/* each processor will get nlocal elements from the root */

        nlocal=4;
        myray=(int*)malloc(nlocal*sizeof(int));

/* create the data to be sent from the root */

        if(irank == mpi_root){
            ntotal   =  nlocal*nrank;
            send_ray = (int*)malloc(ntotal*sizeof(int));
            for(i=0; i<ntotal; i++) send_ray[i]=i;
        }
```

MPI_Reduce Example

```
/* send a data section to each processor */

        ierr = MPI_Scatter(send_ray, nlocal, MPI_INT,
                                  myray, nlocal, MPI_INT,
                                mpi_root, MPI_COMM_WORLD);
/* partial sum */
        psum=0;
        for(i=0;i<nlocal;i++) psum+=myray[i];
        printf("irank= %d psum= %d\n ",irank,psum);

/* reduce partial  sums to the root */

    ierr = MPI_Reduce(&psum, &gsum, 1, MPI_INT,
                        MPI_SUM, mpi_root, MPI_COMM_WORLD);

/* the root prints the global sum */
        if(irank == mpi_root)printf("gsum= %d \n ",gsum);

    ierr = MPI_Finalize();
}
```

MPI_Reduce Example

```
program myreduce

   include "mpif.h"

   integer,parameter    :: mpi_root=0
   integer,allocatable  :: myray(:),send_ray(:)
   integer i,psum,gsum,      nlocal,     ntotal
   integer nrank,irank, ierr

   call MPI_Init(ierr)
   call MPI_Comm_size( MPI_COMM_WORLD, nrank, ierr)
   call MPI_Comm_rank( MPI_COMM_WORLD, irank, ierr))
```

MPI_Reduce Example

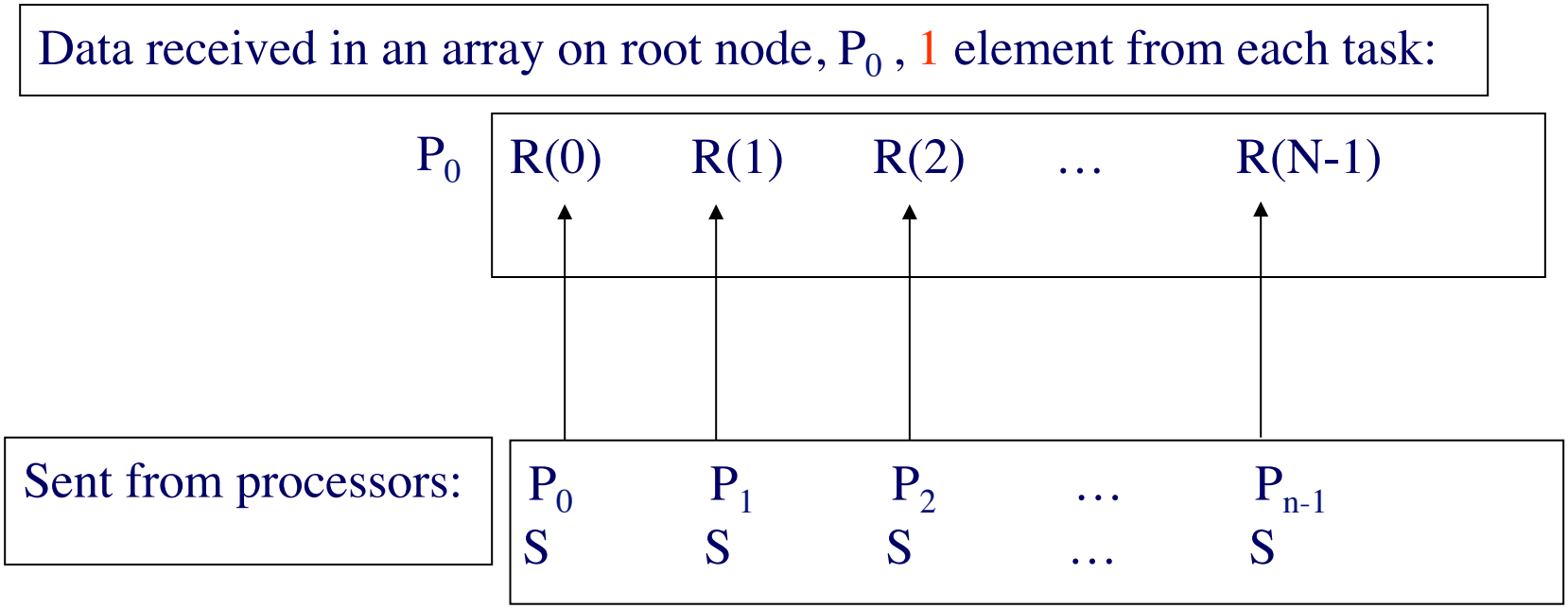
```
!  each processor will get nlocal elements from the root */
   nlocal=4
   allocate( myray(nlocal) )

!  create the data to be sent from the root */
   if(irank == mpi_root) then
      ntotal = nlocal*nrank
      allocate( send_ray(ntotal) )
       do i=1,ntotal
         send_ray(i)=I
       enddo
   endif
```

MPI_Reduce Example

```
!   send a data section to each processor */

    call MPI_Scatter(send_ray, nlocal, MPI_INTEGER, &
                        myray, nlocal, MPI_INTEGER, &
                     mpi_root, MPI_COMM_WORLD,ierr)
!   partial sum
    psum=0
    do i=1,nlocal
      psum=psum+myray(i)
    enddo
    print*,"irank:partial_sum",irank,psum

!   reduce partial sums to the root */

    call MPI_Reduce(psum, gsum, 1, MPI_INTEGER, MPI_SUM, &
                    mpi_root,      MPI_COMM_WORLD,    ierr)


    if(irank == mpi_root) print*,"gsum= ",gsum

    call MPI_Finalize(ierr)
end program
```

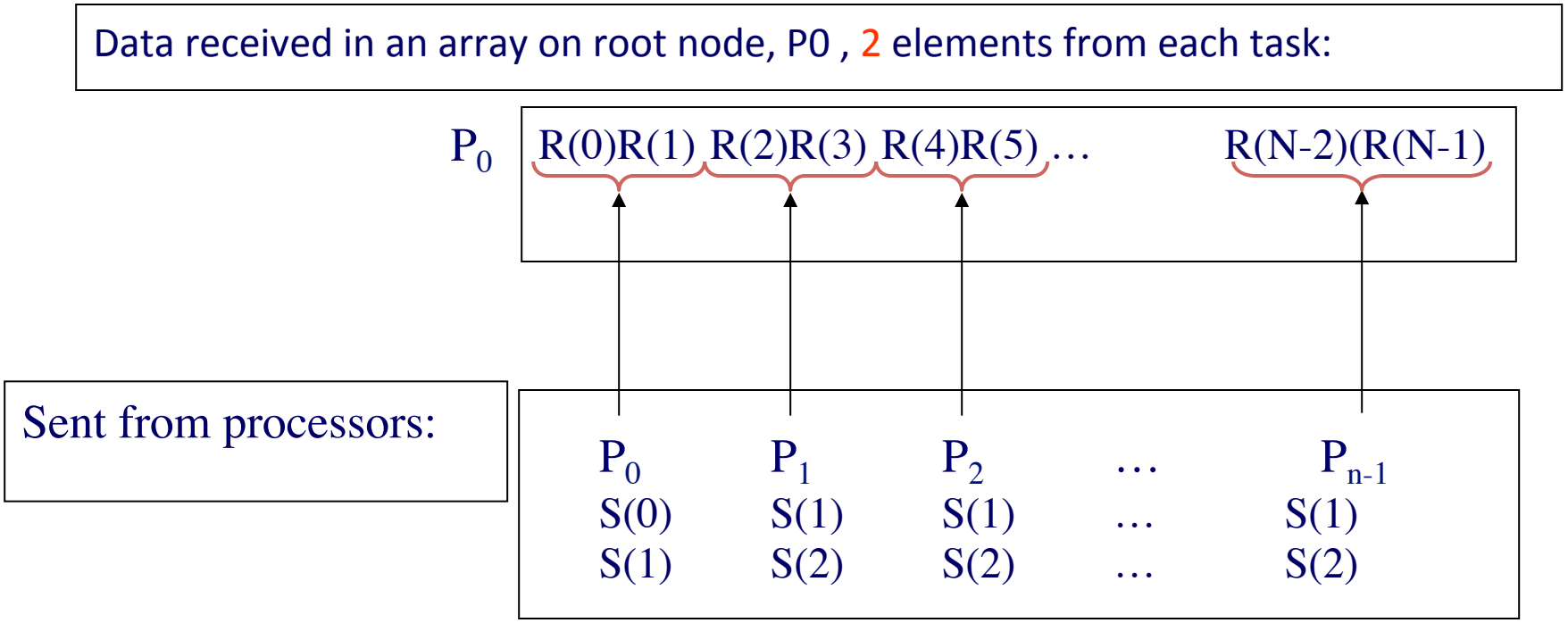
Gather Operation using MPI_Gather

- Inverse of Scatter—root receives a section of an array from each processor



Gather Operation using MPI_Gather

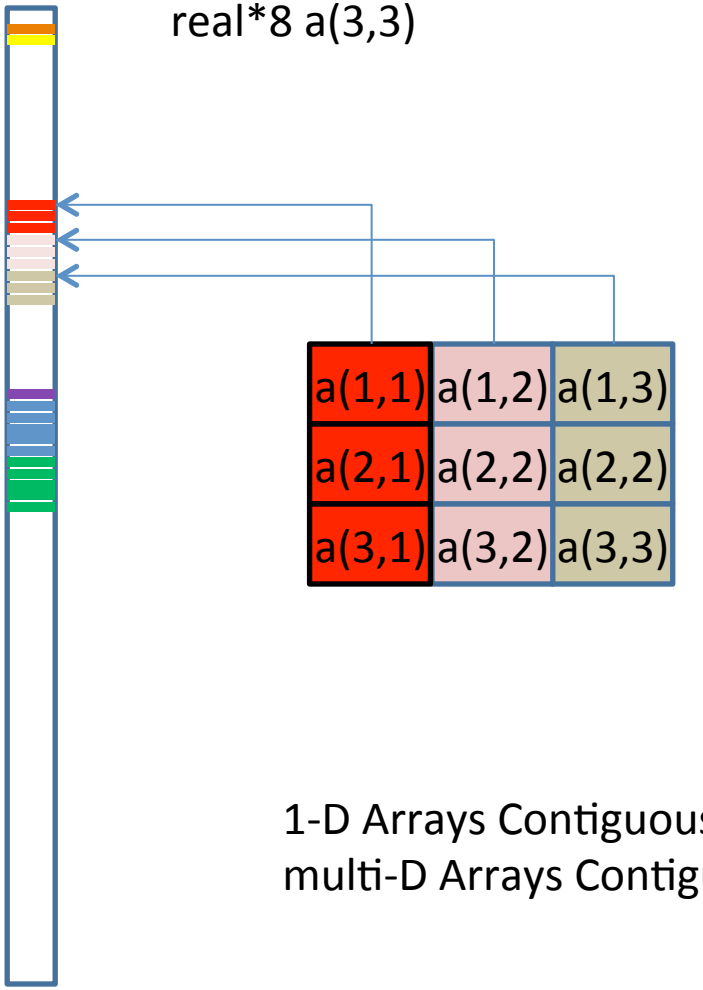
- 2-element version



Contiguous Data and Alignment

Fortran Language

```
real*8  sa, sb
real*8 sc, d1(5),d2(5)
real*8 a(3,3)
```

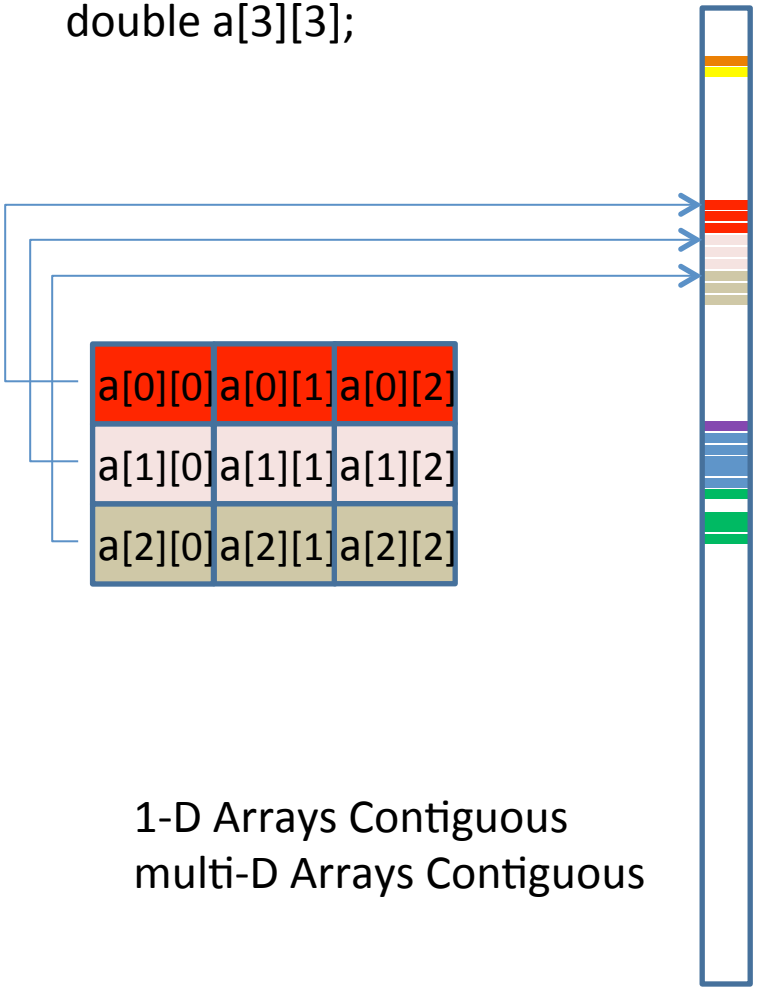


1-D Arrays Contiguous
multi-D Arrays Contiguous

Memory Layout for Compiled Program

C Language

```
double  sa, sb;
double sc, d1[5],d2[5];
double a[3][3];
```



1-D Arrays Contiguous
multi-D Arrays Contiguous

Memory Layout for Compiled Program

MPI_Gather

- C

  ierr=MPI_Gather(&sbuf[0], scnt, stype, &rbuf[0], rcnt, rtype, root, comm);

- Fortran

  call MPI_Gather(sbuf, scnt, stype, rbuf, rcnt, rtype, root, comm, ierr)

- Parameters
  - scnt = number of elements sent from each processor
  - sbuf = sending array of size sendcnts
  - rcnt = number of elements obtained from each processor (not the total)
  - rbuf = receiving array, size rcnt*np

e.g. MPI_Gather(S, 1, stype, R, 1, rtype, root, comm)
Scalar Array

Scatter → Work →Gather

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>


int numnodes, myid, mpi_err;         /*globals*/
#define mpi_root 0

void my_init(int  *argc, char ***argv) {
    mpi_err = MPI_Init(argc,argv);
    mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
    mpi_err = MPI_Comm_rank( MPI_COMM_WORLD, &myid);

}
```

Initialization pushed
Into a function

Scatter → Work →Gather

send_ray → myray → back_ray

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

   int *myray, *send_ray, *back_ray;
   int count, size,mysize,i,k,j,total;

   my_init(&argc,&argv);

   count=4;                          /*each task get 4 elements*/
   myray=(int*)malloc(count*sizeof(int));

   if(myid == mpi_root){             /*create send data*/
      size=count*numnodes;
      send_ray=(int*)malloc(    size*sizeof(int));
      back_ray=(int*)malloc(numnodes*sizeof(int));
      for(i=0;i<size;i++) send_ray[i]=i;
   }
```

Space allocation accommodates any number of tasks

Scatter → Work →Gather

```
mpi_err=MPI_Scatter(send_ray,count, MPI_INT,
                    myray,count, MPI_INT, mpi_root,MPI_COMM_WORLD);

total=0;                                          /*partial sum*/
for(i=0; i<count; i++) total=total+myray[i];
printf("myid= %d total= %d\n ",myid, total);
                                                  /*send back sum*/
mpi_err = MPI_Gather(&total,   1, MPI_INT,
                     back_ray, 1, MPI_INT, mpi_root, MPI_COMM_WORLD);

if(myid == mpi_root){
   total=0;
   for(i=0; i<numnodes; i++) total=total+back_ray[i];
   printf("results from all processors= %d \n ",total);
}
mpi_err = MPI_Finalize();}
```

Should free memory…

MPI_Allgather and MPI_Allreduce

- Gather and Reduce come in an "All" variation
- Results are returned to all processors
- The root parameter is missing from the call
- Similar to a gather or reduce followed by a broadcast
  - but is probably implemented differently

MPI_Allgather

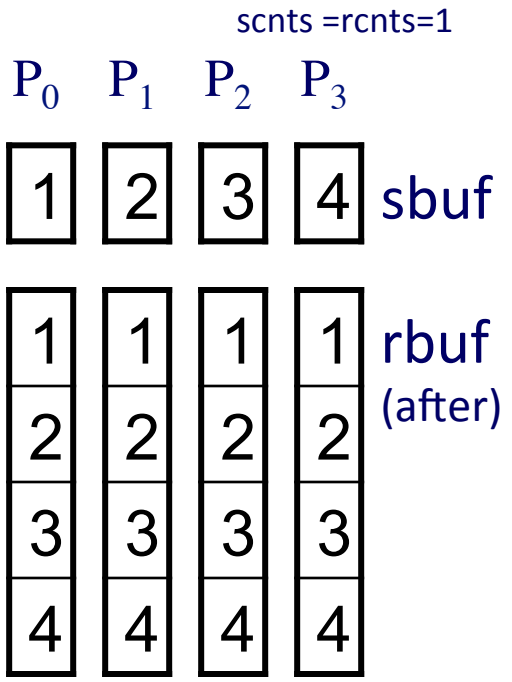
- C

```
ierr = MPI_Allgather(&sbuf[0], scnt, stype, &rbuf[0], rcnt, rtype, comm );
```

- Fortran

```
call    MPI_Allgather( sbuf,    scnt, stype,   rbuf,    rcnt, rtype, comm, ierr)
```

- Parameters
  - scnt = # of elements sent from each processor
  - sbuf = sending array of size scnt
  - rcnt = # of elements obtained from each proc.
  - rbuf = receiving array, size rcnt*np



MPI_Allreduce

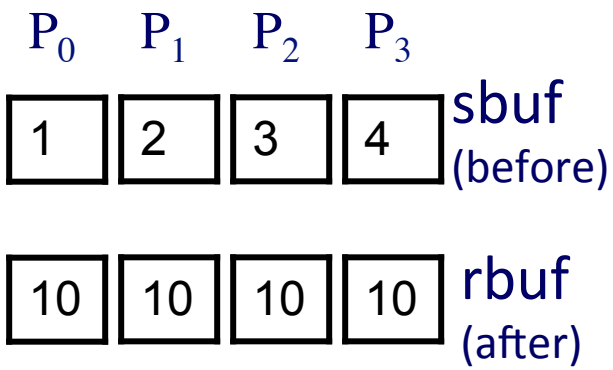
- C

```
ierr=MPI_Allreduce(&sbuf[0],&rbuf[0],cnt,type,op,comm)
```

- Fortran

```
call MPI_Allreduce( sbuf,   rbuf,   cnt,type,op,comm,ierr)
```

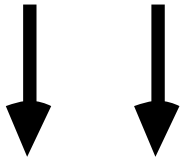
- Parameters
  - sbuf, array to reduce
  - rbuf, receive buffer
  - cnt = sbuf and rbuf size
  - type = datatype
  - operation = binary operator



Global Sum with MPI_Reduce

2d array spread across processors

	X(0)	X(1)	X(2)
TASK 1	A0	B0	C0
TASK 2	A1	B1	C1
TASK 3	A2	B2	C2

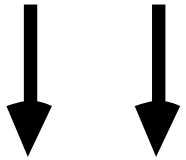


	X(0)	X(1)	X(2)
TASK 1	A0+A1+A2	B0+B1+B2	C0+C1+C2
TASK 2			
TASK 3			

Global Sum with MPI_Allreduce

2d array spread across processors

	X(0)	X(1)	X(2)
TASK 1	A0	B0	C0
TASK 2	A1	B1	C1
TASK 3	A2	B2	C2



	X(0)	X(1)	X(2)
TASK 1	A0+A1+A2	B0+B1+B2	C0+C1+C2
TASK 2	A0+A1+A2	B0+B1+B2	C0+C1+C2
TASK 3	A0+A1+A2	B0+B1+B2	C0+C1+C2

In place reduction

If there is a lot of data involved, you don't want to allocate twice on the root

```
void *buffer = malloc(size*sizeof(double));
// write data into the buffer
if (mytid==0) {
  sendbuf = MPI_IN_PLACE; recvbuf = buffer;
} else {
  sendbuf = buffer;       recvbuf = NULL;
}
MPI_Reduce(sendbuf,recvbuf,size,MPI_DOUBLE,MPI_MAX,0,comm);
```

Simpler for the Allreduce call

```
void *buffer = malloc(size*sizeof(double));
// write data into the buffer
sendbuf = MPI_IN_PLACE; recvbuf = buffer;
MPI_Reduce(sendbuf,recvbuf,size,MPI_DOUBLE,MPI_MAX,0,comm);
```

Should free memory…

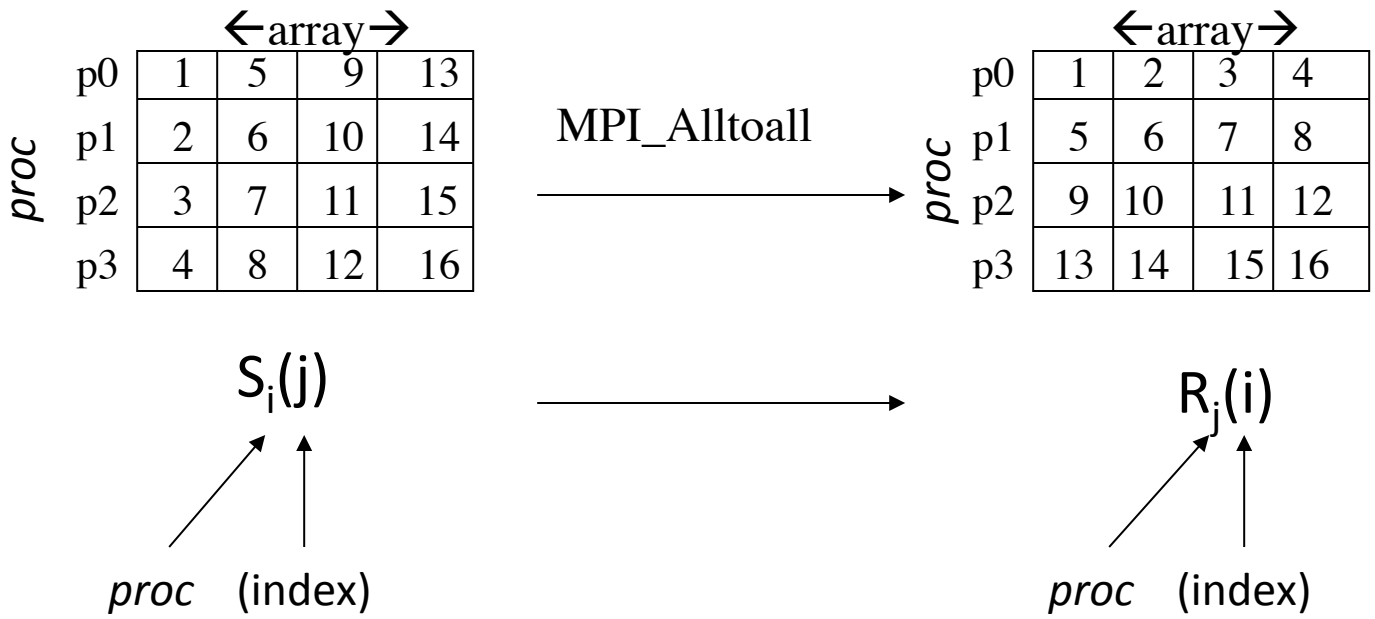
All to All communication with MPI_Alltoall

- Each processor sends and receives data to/from all others
- C

  ierr=MPI_Alltoall(&sbuf[0], scnt, stype, &rbuf[0], rcnt, rtype, comm);
- Fortran

  call MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, ierr)
- Parameters
  - scnt # of elements sent to each processor
  - sbuf is an array of size scnt*np (np=# of processes)
  - rcnts # of elements obtained from each processor
  - rbuf of size rcnt*np
- Note: send buffer and receive buffer must be of size = scnt * np

All to All with MPI_Alltoall



The variable or "V" operators

- The size of data in the send and receive buffers may vary on each processor.
- MPI_Gatherv: Gather different amounts of data from each processor to the root processor
- MPI_Allgatherv: Gather different amounts of data from each processor and sends all data to each
- MPI_Scatterv: Send different amounts of data to each processor from the root processor
- MPI_Alltoallv: Send and receive different amounts of data form all processors

MPI_Gatherv

- C

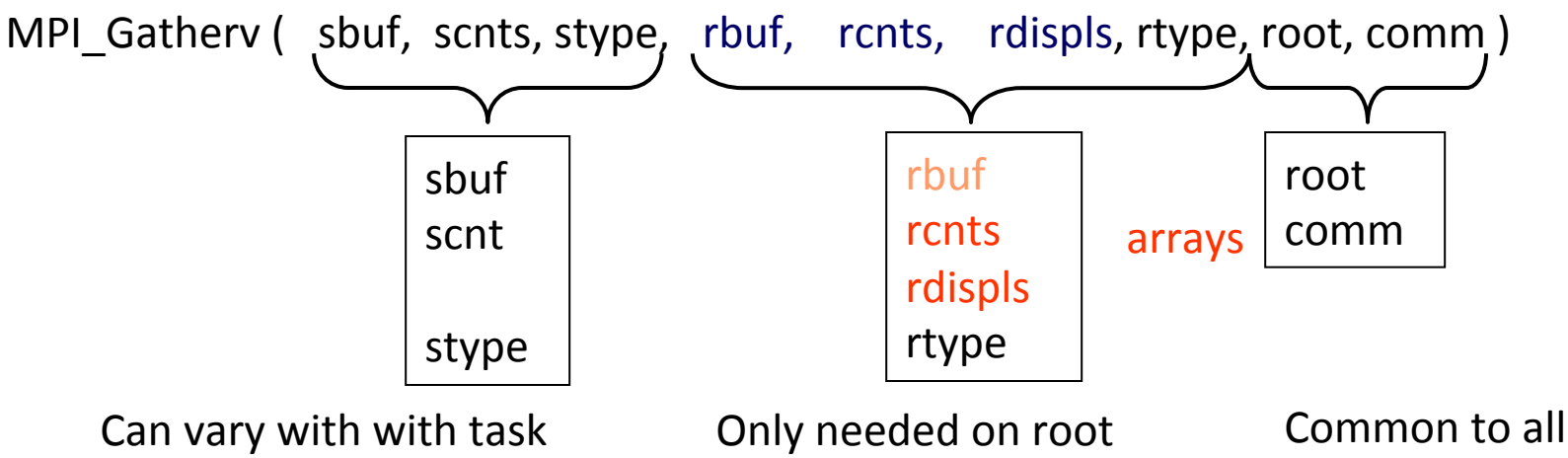
```
ierr=MPI_Gatherv(&sbuf[0], scnt,      stype,
                 &rbuf[0], &rcnts[0],&rdispls[0],rtype,
                 root, comm);
```

- Fortran

```
call MPI_Gatherv(sbuf, scnt,  stype,
                 rbuf, rcnts, rdispls, rtype,
                 root, comm,  ierr)
```

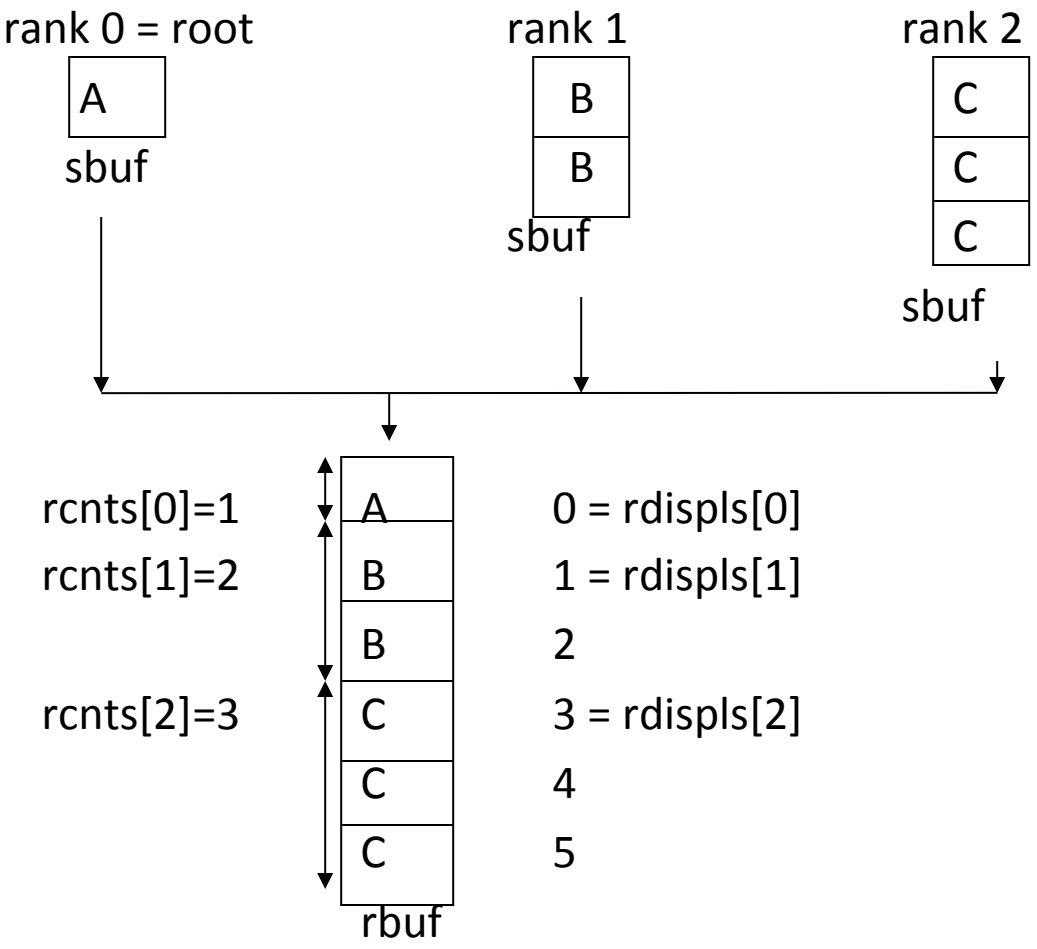
- Parameters:
  - rcnts is now an array of counts to be received from each processor—1st element # from processor 0, 2nd from processor 1, etc.
  - rdispls is an array of dis
  - ments (offsets)

MPI_Gatherv



- Messages (sbuf,scnts) are placed in rbuf in rank order:
  - rcnts(i) elements, starting at offset rdispls(i)
  - for i = {0,…,n-1} of group of n tasks.
- Size of data send by rank i and received in root rcnts(i) must be equal.
- "r" variables not "significant" on non-root

MPI_Gatherv


rank 0 = root
A
sbuf
rank 1
B
B
sbuf
rank 2
C
C
C
sbuf
rcnts[0]=1
rcnts[1]=2
rcnts[2]=3
A
B
B
C
C
C
rbuf
0 = rdispls[0]
1 = rdispls[1]
2
3 = rdispls[2]
4
5

MPI_Gatherv C code

```
#include <stdio.h>
#include   <mpi.h>
#define  N  8
#define  NP 2
#define  NPROC N/NP

main(int argc, char **argv){
/*          Build v from partial vectors, vp, in reverse order.
            MAP: v=[vp3,vp2,pv1,pv0]   4 processors, vp size=2
                    vpi = partial vector from processor i. */

   int npes, mype, ierr;
   double v[N], vp[NP];
   int  j,i, ivcnt[NPROC], ivdispl[NPROC];

   ierr = MPI_Init(&argc, &argv);
   ierr = MPI_Comm_size(MPI_COMM_WORLD, &npes);
   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &mype);
   if(npes != NPROC){ printf("Use %d PEs\n",N); exit(9);}

   for(i=0; i<npes; i++){
        ivcnt[i] = NP;              // get 2 (NP) from each rank
     ivdispl[i] = N-NP*(i+1); }  // reverse append order
```

MPI_Gatherv C code

```
    for(i=0; i<NP; i++) vp[i]=(double) mype;

    ierr=MPI_Gatherv(vp,  NP,      MPI_DOUBLE,
                     v,ivcnt,ivdispl,MPI_DOUBLE,0,MPI_COMM_WORLD);
    if(mype == 0){
       printf(" %d PEs;  partial vector length = %d.\n",npes,NP);
       printf(" Reversed storage, locations =");
       for(i=0;i<npes;i++) printf("%d ",ivdispl[i]); printf("\n");
       for(i=0;i<N;i++)    printf("%4.0f ",   v[i]); printf("\n");
    }
    ierr = MPI_Finalize();
}
```

4 PEs; partial vector length = 2.
Reversed storage, locations =6 4 2 0
 3 3 2 2 1 1 0 0

MPI_Gatherv Fortran code

```
program gather

      include "mpif.h"
      integer,parameter :: N=8, NP=2
      integer,parameter :: NPROC = N/NP
      integer           :: npes, mype, ierr, i,j
      real*8            :: v(N),vp(NP)
      integer           :: ivcnt(NPROC), ivdispl(NPROC)

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,mype,ierr)
      call mpi_comm_size(MPI_COMM_WORLD,npes,ierr)
      if(npes.ne.NPROC) stop
```

MPI_Gatherv Fortran code

```
ivcnt = NP                !// get 2 (NPs) from each rank
do i=1,npes; ivdispl(i)= N-NP*(i); enddo
do i=1,NP;          vp(i)= mype;      enddo

call mpi_gatherv(vp,NP,MPI_REAL8,          &
                 v,ivcnt,ivdispl,MPI_REAL8, 0,MPI_COMM_WORLD,ierr)

if(mype==0) print*, "Rev. Locs", ivdispl, "Rev. Vals",v

call mpi_finalize(ierr)

end program
```

Rev. Locs 6 4 2 0
Rev. Vals 3 3 2 2 1 1 0 0

MPI_Alltoallv

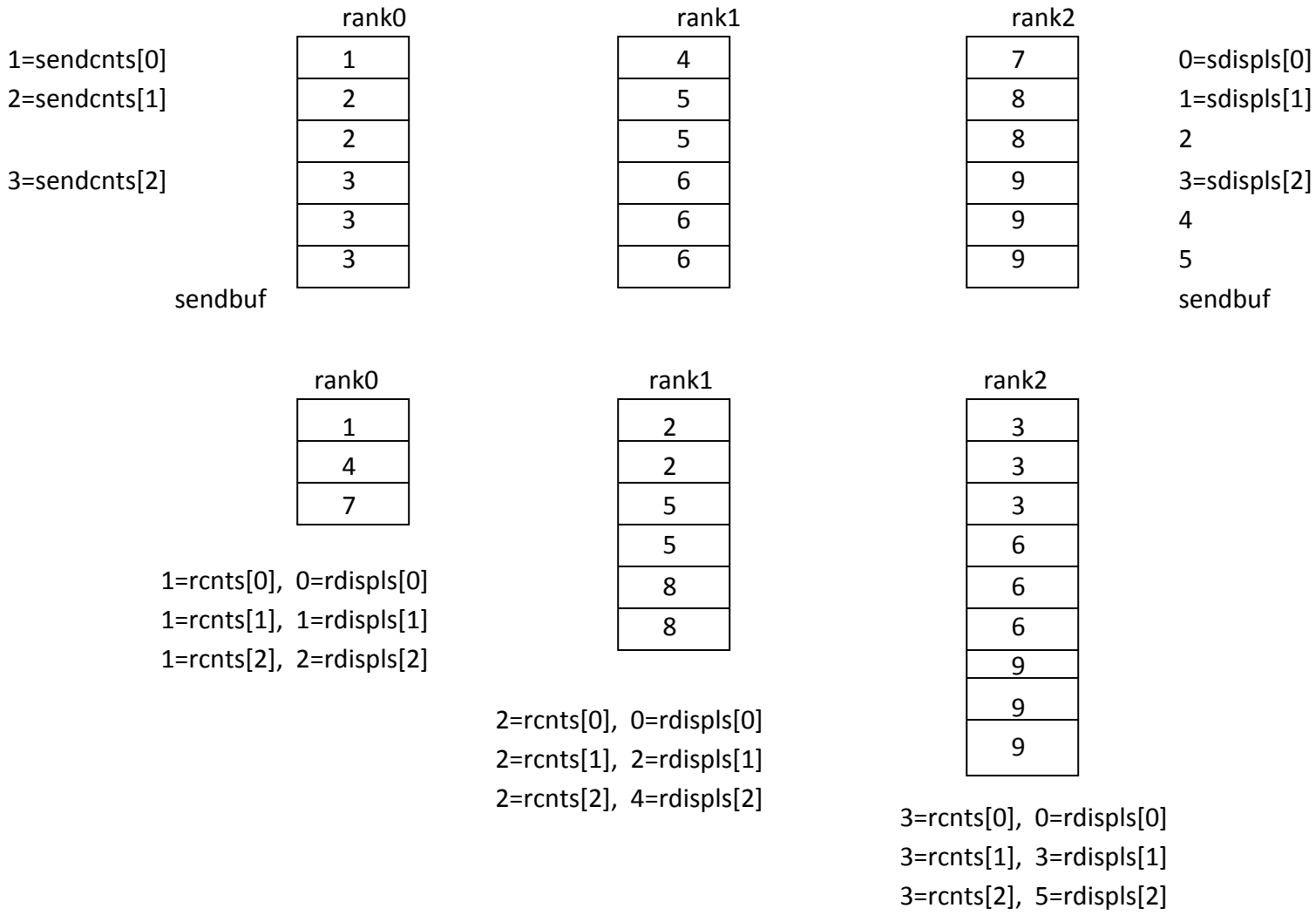
- Send and receive different amounts of data form all processors
- C

```
ierr=MPI_Alltoallv(&sbuf[0],&scnts[0],&sdispls[0],stype,
                   &rbuf[0],&rcnts[0],&rdispls[0],rtype,
                   comm);
```

- Fortran

```
Call MPI_Alltoallv(sbuf, scnts, sdispls, stype,
                   rbuf, rcnts, rdispls, rtype,
                   comm, ierr);
```

MPI_Alltoallv



Scan (parallel prefix)

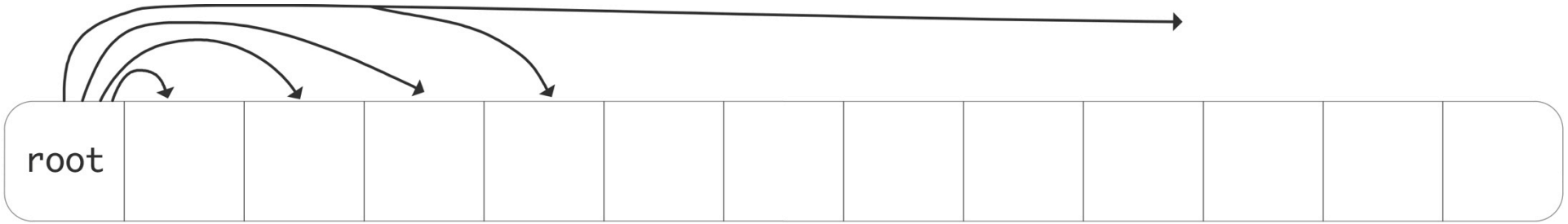
- Reduce computes *sum(x_i,i=0..P-1)*
- Scan computes on p_j partial *sum(x_i,i=0..j)*
- Two versions:
- Inclusive:
- **int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)**
- **MPI_Exscan** exclusive, same parameters.
- Why inclusive/exclusive? Plus/Mult vs Min/Max

Finally: barrier

- MPI_Barrier(comm)
- Simple to use, looks very useful, is not.
- There is almost no use for barriers, besides, collectives induce a synchronization

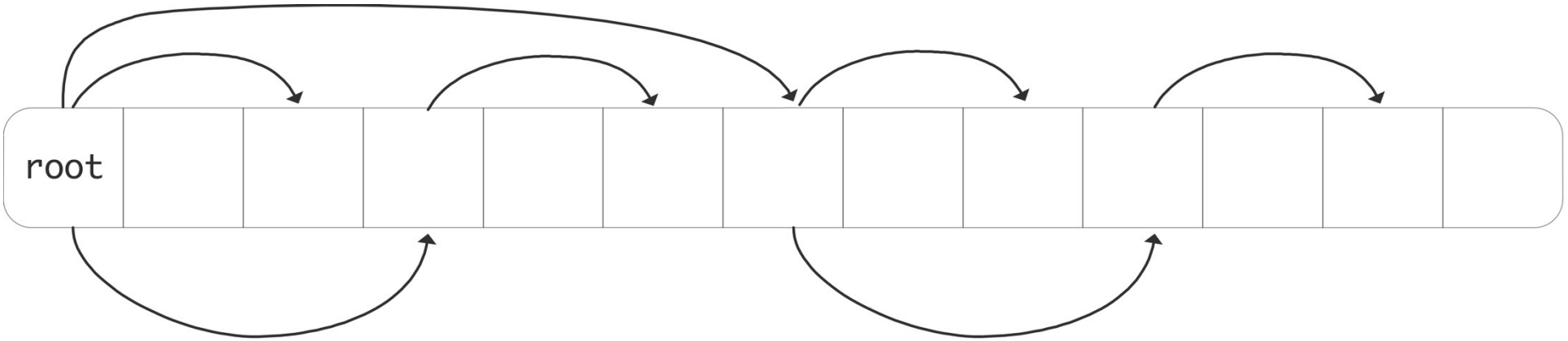
A word about implementation

How do you do a broadcast?



- Complexity?

How really to do a broadcast



- Complexity?