TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

MPI Programming

IHPCSS

Collectives

July 7-12, 2019

PRESENTED BY:

John Cazes

cazes@tacc.utexas.edu

Lecture and Lab slides available at: tinyurl.com/tacc-2019-ihpcss

Outline

Advantages of Message Passing
Background on MPI
Basic Information
Point to Point Communication
Nonblocking Communication
Wildcards
Probing
Collective Communications
"V" Operations
Derived Datatypes
Communicators

MPI Collective Communications (CC)

Barrier

Broadcast

Reduce

Gather/Scatter

All to All

AllReduce

AllGather

MPI Collective Communications (CC)

- All processes in the group MUST call the CC
- Must have matching arguments
- Amount of data sent must match amount received (Mapping may vary) (No recv status)
- CC calls can (not required) return as soon as their participation is complete—the call may/may-not synchronize all processes of the group
  (If you need to sync, use a barrier)

MPI_Barrier

Blocks the caller until all members in the communicator have called it.

Used as a synchronization tool.

C

```
ierr = MPI_Barrier(comm);
```

Fortran

```
call MPI_Barrier(comm, ierr)
```

Parameter

Comm: communicator (often `MPI_COMM_WORLD`)

Broadcast Operation: MPI_Bcast

All tasks call MPI_Bcast

One task (root) sends a message all others receive the message

C

```
ierr=MPI_Bcast(&buffer,count,datatype,root,communicator);
```

Fortran

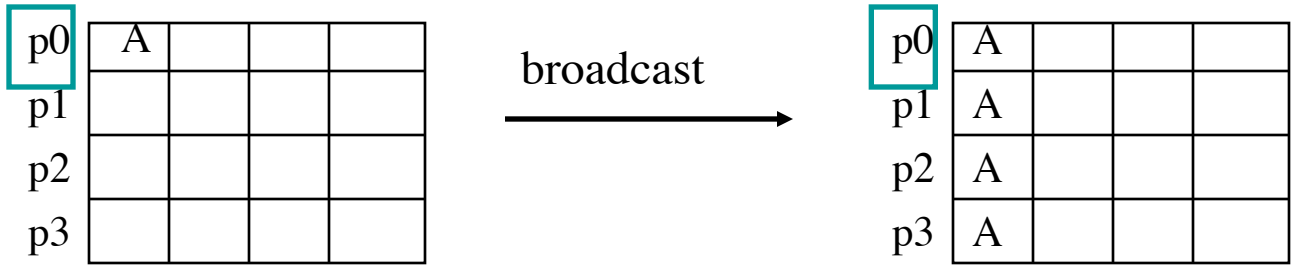
```
call MPI_Bcast(buffer,count,datatype,root,communicator,ierr)
```

Root is task that sends the message

Broadcast Example

Write a parallel program to broadcast data using MPI_Bcast

- Initialize MPI
- Have processor 0 broadcast an integer
- Have all processors print the data
- Quit MPI



Broadcast Example – C

```
#include "mpi.h"
#include <math.h>

/* Broadcast Example */
int main(argc,argv)
int argc;
char *argv[];
{

  int mytask, ntasks;
  int i, ierr, source, count;
  int buffer[4];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&mytask);
```

Broadcast Example – C

```
  source=0;
  count=4;
  if (mytask == source) {
    for (i=0;i<count;i++) buffer[i]=i+1 ;
  }

  ierr = MPI_Bcast(buffer,count,MPI_INTEGER,source,MPI_COMM_WORLD);
  printf( "%3d : ",mytask);
  for (i=0;i<count;i++) printf( " %2d ",buffer[i]);
  printf( "\n");

  ierr = MPI_Finalize();
}
```

Broadcast Example – F90

```
Program Main
include "mpif.h"

! Broadcast Example
  Integer :: i,mytask,ntasks
  Integer :: source,count
  Integer :: buffer(4)
  Integer :: status(MPI_STATUS_SIZE)

  call MPI_Init(ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,ntasks,ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,mytask,ierr)
```

Broadcast Example – F90

```
source=0
  count=4
  if (mytask == source) then
    forall (i=1:count) buffer(i)=i
  end if

  call MPI_Bcast(buffer,count,MPI_INTEGER,source,MPI_COMM_WORLD,ierr)
  print*, mytask," : ", (buffer(i), i=1,count)

  call MPI_Finalize(ierr)
End Program Main
```

Reduction Operations

Used to combine partial results from all processors

Result returned to root processor

Several types of operations available

Works on single elements and arrays

MPI_Reduce

C

```
ierr=MPI_Reduce(&sendbuf, &recvbuf, count, datatype,
                operation, root, communicator);
```

Fortran

```
call MPI_Reduce(sendbuf, recvbuf, count, datatype, &
                operation, root, communicator, ierr)
```

Parameters

- Like MPI_Bcast, a root is specified.
- Operation is a type of mathematical operation

Predefined Operations for MPI_Reduce

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or

Predefined Operations for MPI_Reduce

OR define your own reduction operator to work with your custom data type

	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum value and location
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or

Global Sum with MPI_Reduce

C

```
double sum_partial, sum_global;
sum_partial = ...;
ierr = MPI_Reduce(&sum_partial, &sum_global,
           1, MPI_DOUBLE_PRECISION,
           MPI_SUM,root, MPI_COMM_WORLD);
```

Fortran

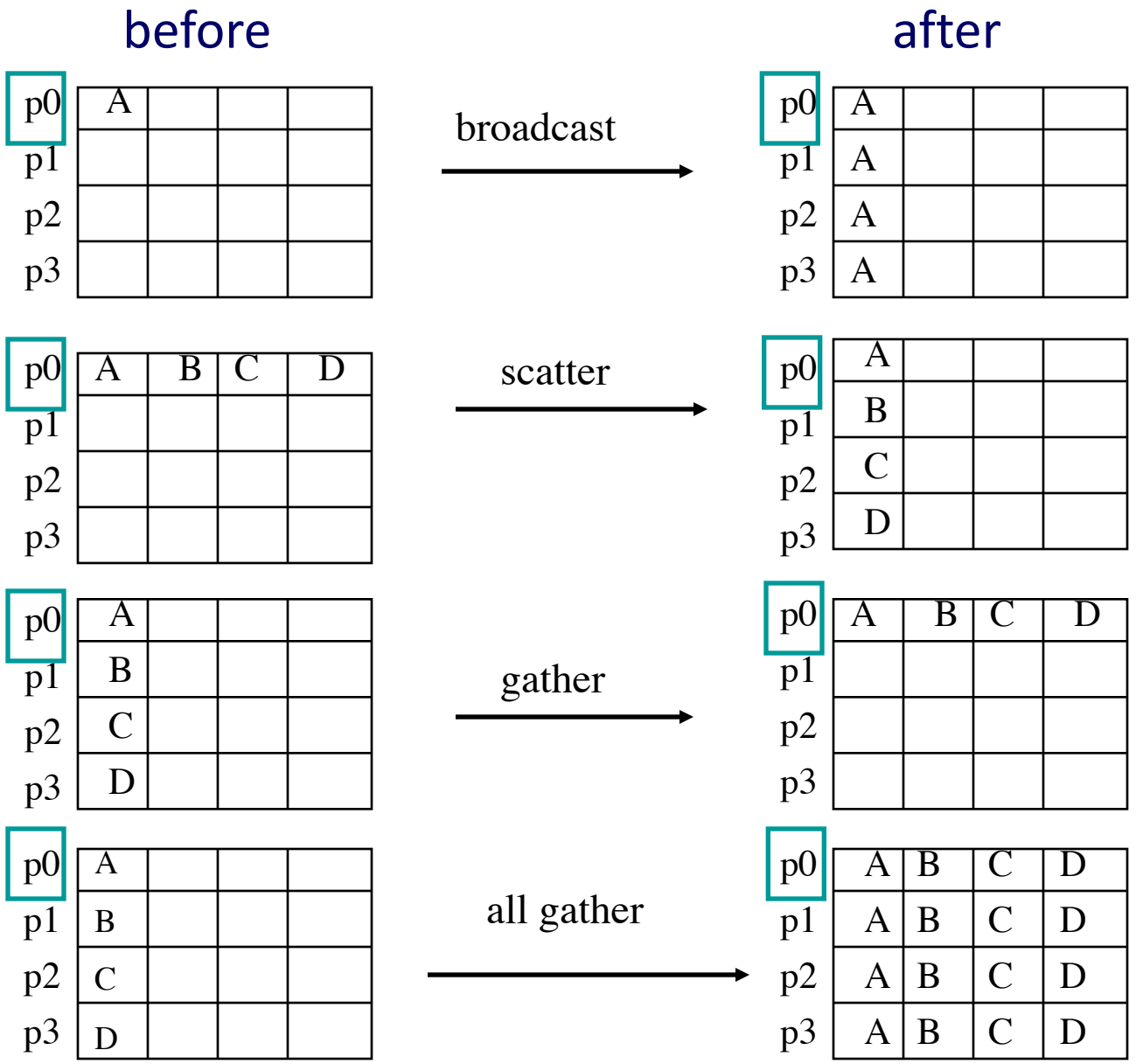
```
double precision sum_partial, sum_global
sum_partial = ...
call MPI_Reduce(sum_partial, sum_global, &
        1, MPI_DOUBLE_PRECISION,          &
        MPI_SUM,root, MPI_COMM_WORLD, ierr)
```

Scatter and Gather

Root

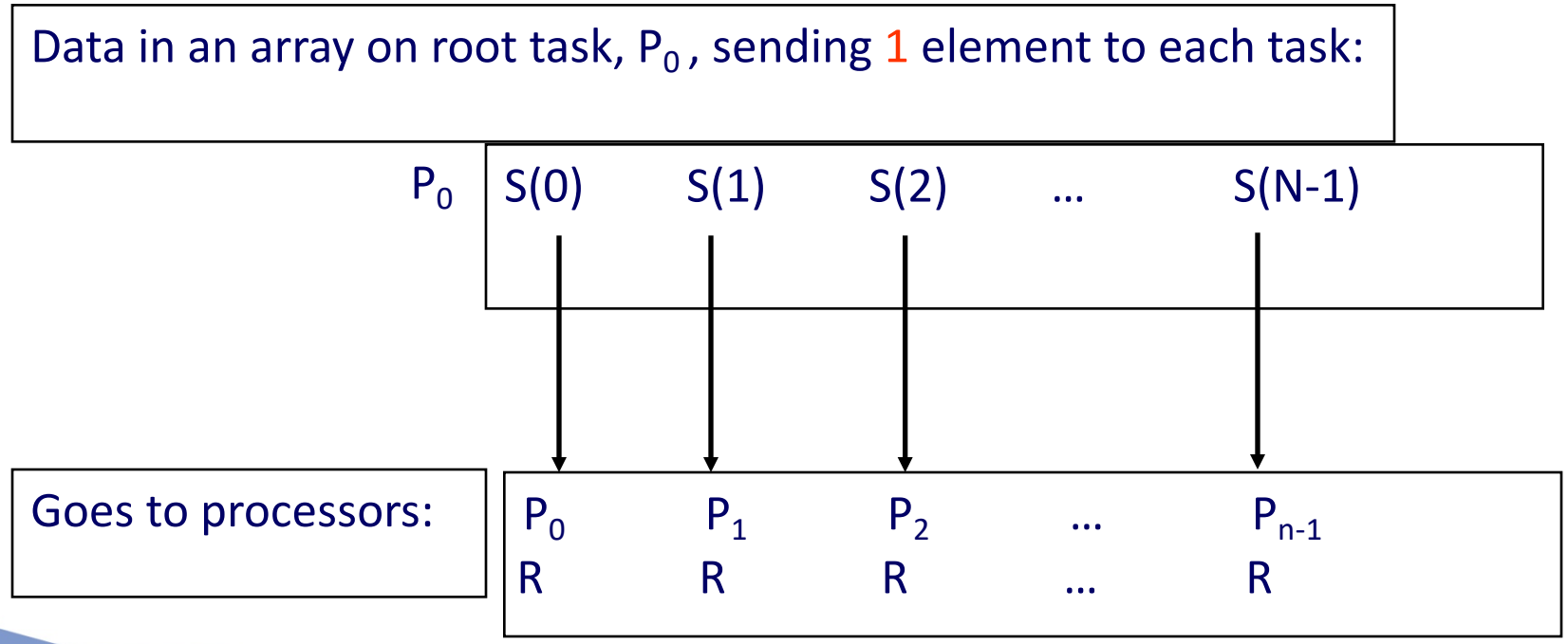
"send" array element or single variable

task or processor



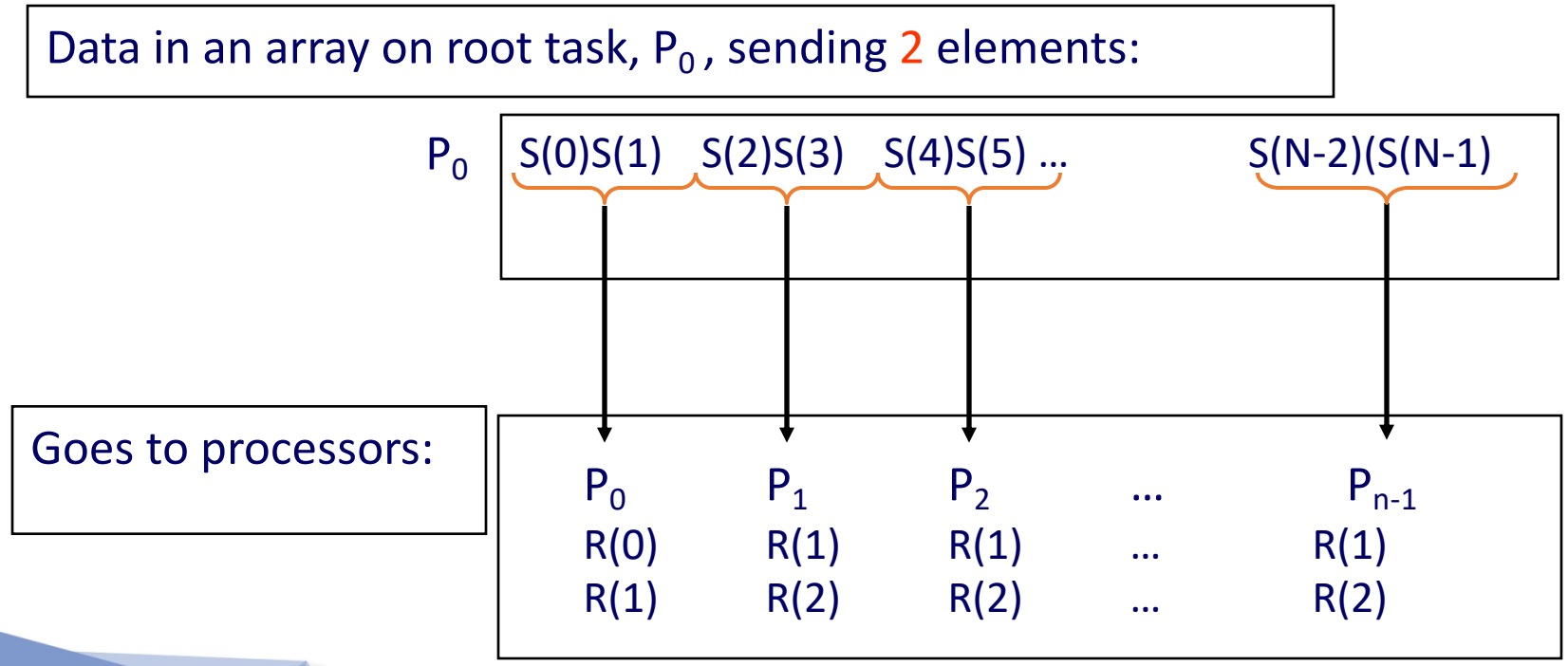
Scatter Operation using MPI_Scatter

Similar to Broadcast but sends a section of an array to each processors



Scatter Operation using MPI_Scatter

Similar to Broadcast but sends a section of an array to each processors



MPI_Scatter Syntax

C

```
ierr = MPI_Scatter(&sbuf, scnt, stype, &rbuf, rcnt,
rtype, root, comm );
```

Fortran

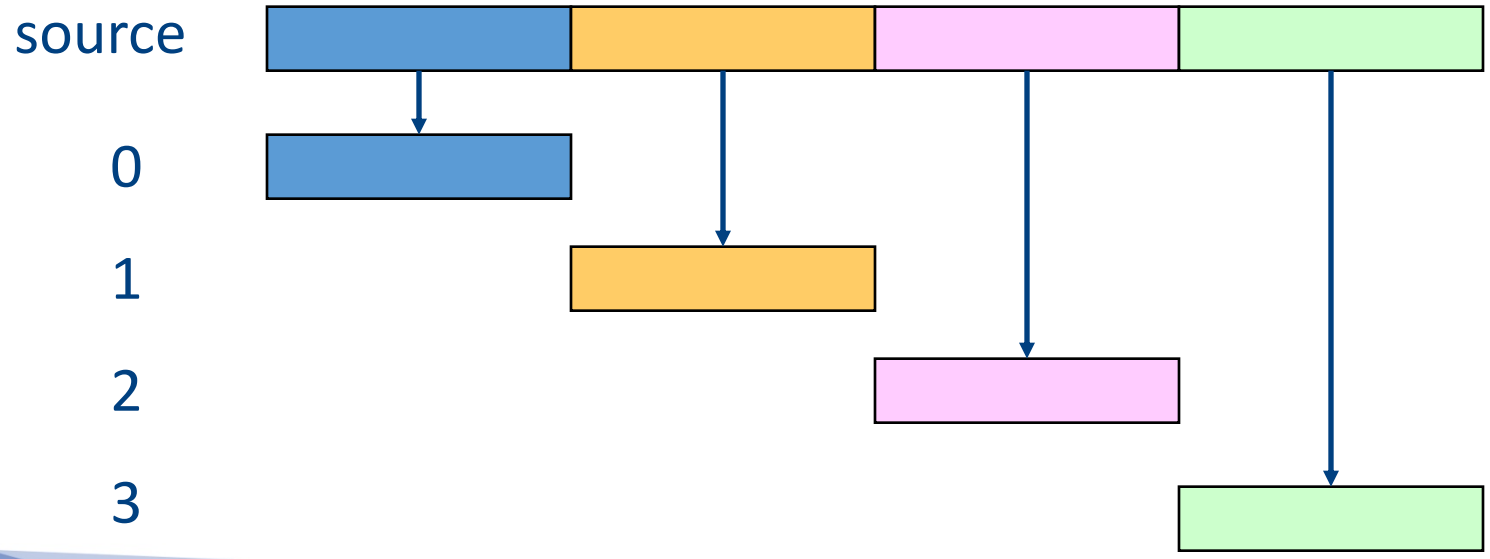
```
call MPI_Scatter(sbuf, scnt, stype, rbuf, rcnt, &
rtype, root,comm,ierr)
```

Parameters

sbuf = array of size (number processors*sendcnts)
scnt = number of elements sent to each processor
rcnt = number of element(s) obtained from the root processor
rbuf = element(s) obtained from the root processor

MPI_Scatter and MPI_Reduce example

- This program shows how to use MPI_Scatter and MPI_Reduce
- Each processor gets different data from the root processor
- The data is summed and then sent back using MPI_Reduce
- The root processor then prints the global sum



MPI_Scatter/Reduce F90 code

```
Program Main
   include “mpif.h”
   Integer, Parameter :: mpi_root=0
   Integer :: count,ierr,mytask,ntasks
   Integer :: size,mysize,i
   Real, allocatable :: myray,send_ray,total,gtotal

   call MPI_Init(ierr)
   call MPI_Comm_size(MPI_COMM_WORLD,ntasks,ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD,mytask,ierr)
!** each processor will get count elements from the root
   count=4
   allocate(myray(count))
!** create the data to be sent from root
   if (mytask == mpi_root) then
      size=count*ntasks
      allocate(send_ray(size))
      forall(i=1:size) send_ray(i)=real(i)
   end if
```

```
!** send different data to each processor
        call MPI_Scatter(send_ray,count MPI_REAL,myray,count, &
                         MPI_REAL,mpi_root,MPI_COMM_WORLD,ierr)
/* each processor does a local sum */
        total=sum(myray)
        print*, “mytask=“, mytask,  “total=", total)
!** send the local sums back to the root
        call MPI_Reduce(total, gtotal, 1, MPI_REAL, MPI_SUM, &
                         mpi_root, MPI_COMM_WORLD, ierr);
!** the root prints the global sum
        if (myid == mpi_root) then
           print*, "results from all processors=", gtotal
           deallocate(sendray)
        end if
        deallocate(myray)
    call MPI_Finalize(ierr)
End Program Main
```

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

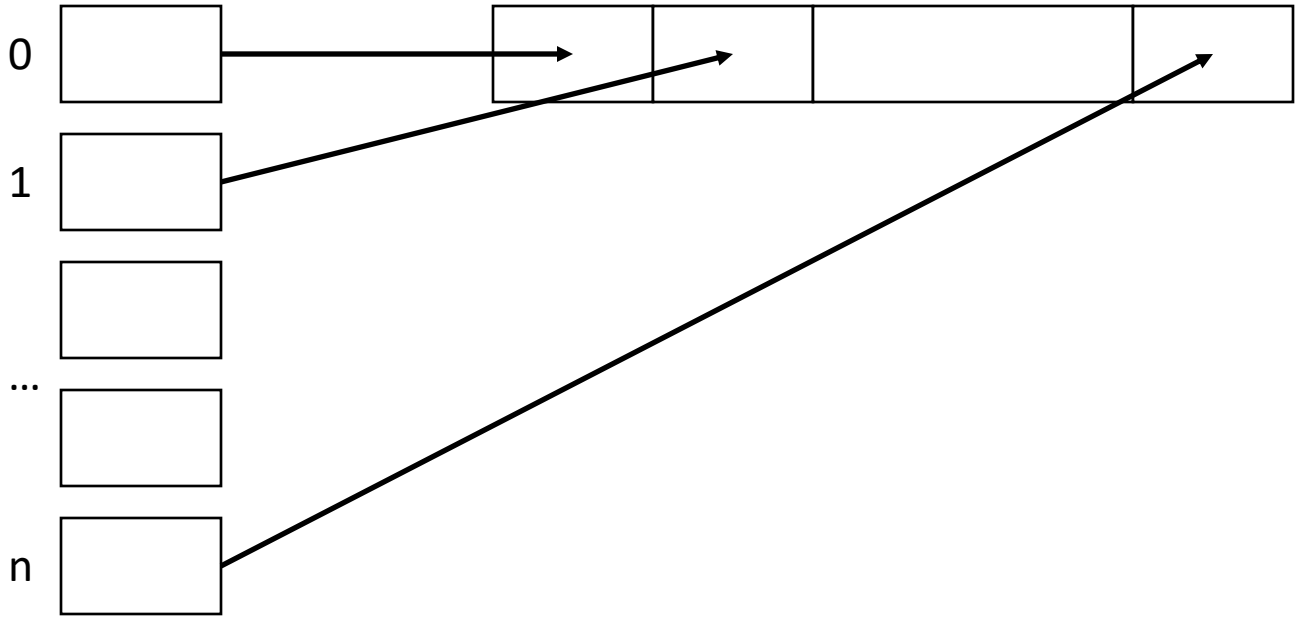
sbuf = sending array of size sendcnts

rcnt = number of elements obtained from each processor NOT TOTAL

rbuf = receiving array, size recvcnts*number of processors

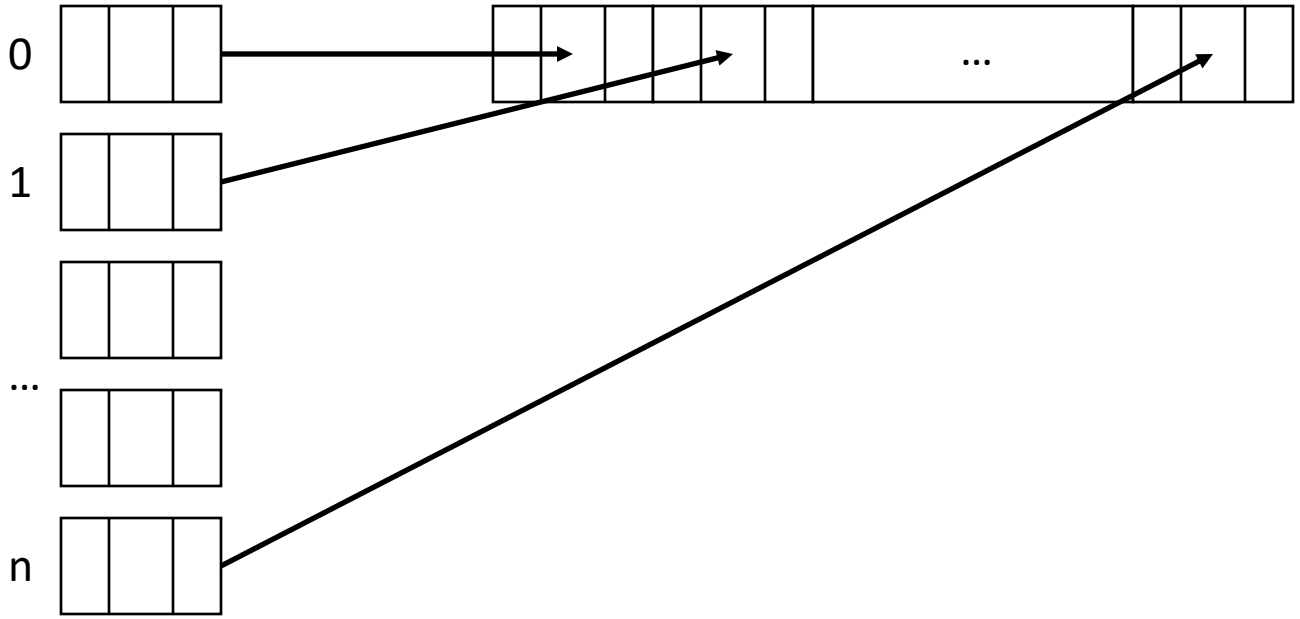
e.g. MPI_Gather(S, 1, stype, R, 1, rtype, root, comm)

Gather on Scalars



MPI_Gather(S, 1, stype, R, 1, rtype, root, comm)

Gather on Vectors



MPI_Gather(S, 3, stype, R, 3, rtype, root, comm)

Building a Matrix with MPI_Gather in C

```
#include <stdlib.h>
#include <stdio.h>
#include  <mpi.h>
#define  N 4
main(int argc, char **argv){
/* Build matrix A from ROW vectors v; 4 processors, A=4x4.
   MAP: A = [v0,
             v1,
             v2,
             v3]  vi = vector ROW from process i.            */
  int npes, mype, ierr;
  int i, j;
  double a[N][N], v[N];

  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &npes);
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &mype);

  if(npes != N){ printf("Use %d PEs\n",N); exit(9);}

  for(i=0; i<N; i++) v[i] = (double) mype ; /* Fill v with PE# */
  /*Gather up ROW vecs into matrix "a" on PE 0.*/
  ierr = MPI_Gather(v,N,MPI_DOUBLE,a,N,MPI_DOUBLE, 0,MPI_COMM_WORLD);
  if(mype == 0)
    for(i=0; i<N; i++){
      for(j=0; j<N; j++) printf("%5f ", a[i][j]);
      printf("\n");
    }
  ierr = MPI_Finalize();
}
```

Building a Matrix with MPI_Gather in F90

```
Program Main
! Build matrix A from column vectors v; 4 processors, A=4x4.
! MAP: A = [v0,v1,v2,v3]  vi = column vector from process I.
  use mpi
  integer,parameter :: N=4
  real*8            :: a(N,N),v(N)

  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.N) stop
  v=mype
  call mpi_gather(v,N,MPI_REAL8,a,N,MPI_REAL8,0,MPI_COMM_WORLD,ierr)
  if(mype.eq.0) write(6,'(4f5.0)') ((a(i,j),j=1,N),i=1,4)

  call mpi_finalize(ierr)
End Program Main
```

MPI_Allgather and MPI_Allreduce

Gather and Reduce come in an "ALL" variations

Results are returned to all processors

The root parameter is missing from the call

Similar to a gather or reduce followed by a broadcast

MPI_Allgather

C

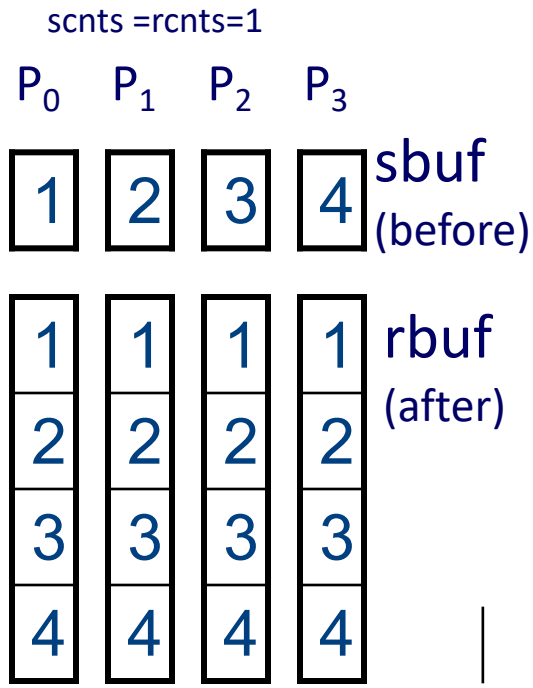
```
ierr = MPI_Allgather(&sbuf,scnt,stype,&rbuf,rcnt,rtype,comm );
```

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*number of processors



MPI_Allreduce

C

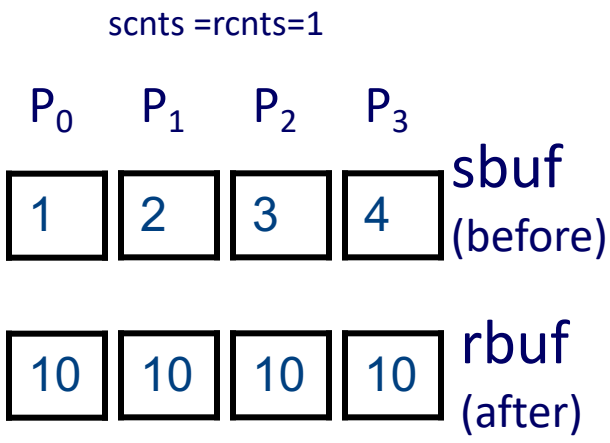
```
ierr = MPI_Allreduce(&sbuf,&rbuf,cnt,datatype,operation,comm);
```

Fortran

```
call MPI_Allreduce(sbuf,rbuf,cnt,datatype,operation,comm,ierr)
```

Parameters

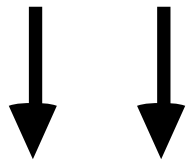
sbuf = sending array
rbuf = receiving array
cnt = # of array elements (sbuf & rbuf)
type = datatype
operation = binary operator



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

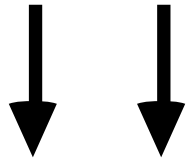


	X(0)	X(1)	X(2)
TASK 1	A0+A1+A2	B0+B1+B2	C0+C1+C2
TASK 2			
TASK 3			

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

All-to-all communication with MPI_Alltoall

- MPI_Alltoall works like MPI_Allgather, except each process sends a distinct data to each of the receivers.
- Each task sends its j^{th} block to task j
- Each task gets different data

Non-blocking collectives

Collectives are blocking.(sort of)

Compare blocking sends:

MPI_Send -> MPI_Isend

Non-blocking collectives:

MPI_Bcast -> MPI_Ibcast

Use for overlap communication/computation

Use of non-blocking collectives

Similar calls, but output a request object:

MPI_Isomething(<usual arguments>, **MPI_Request** *req);

Calls return immediately

- Send buffers can be read but not modified
- recv buffers should not be accessed

Multiple collectives can complete in any order

No guaranteed progress.

Nonblocking tests:

- Test, Testany, Testall, Testsome

Blocking wait:

- Wait, Waitany, Waitall, Waitsome

Restrictions of non-blocking collectives

No tags, in-order matching

Send and vector buffers may not be touched during operation

MPI_Cancel not supported

No matching with blocking collectives

MPI_Iallreduce

C

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

Fortran

```
call MPI_Iallreduce( sbuf,   rbuf,   cnt,type,op,comm, &request,ierr)
```

Parameters

sbuf = array to reduce
rbuf = receive buffer
cnt = sbuf and rbuf size
type = datatype
op = binary operator
request = communication request (the ID)

e.g.

```
MPI_Iallreduce( ......., &request);
……
MPI_Wait(request);
```

The variable or “V” operators

The size of data in the send and receive buffers may vary on each proce

- 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
- 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

Only MPI_Gatherv and MPI_Alltoallv are reviewed

MPI_Gatherv

C

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

Fortran

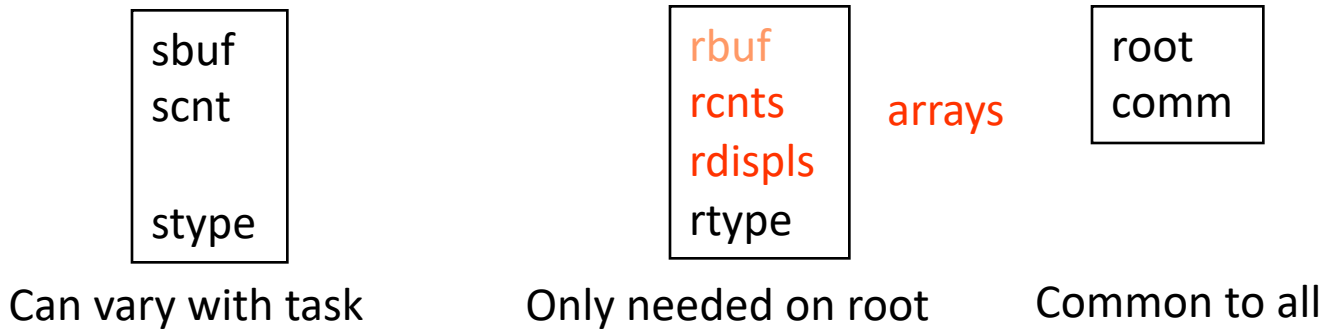
```
Call MPI_Gatherv(sbuf,scnt,stype, &
                 rbuf,rcnts,rdispls, &
                 rtype,comm,root,ierr)
```

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 displacements (offsets)

MPI_Gatherv

MPI_Gatherv (sbuf, scnts, stype, rbuf, rcnts, rdispls, rtype, root, comm)



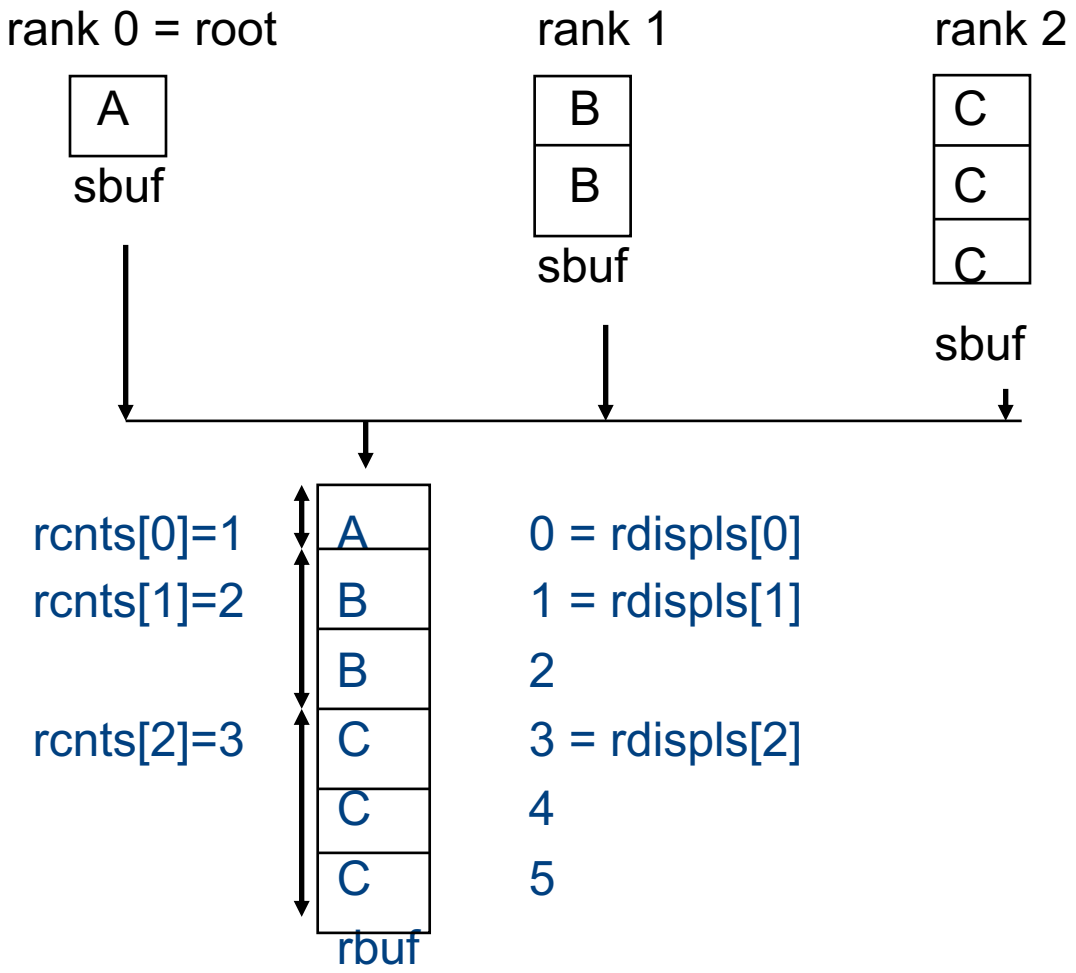
Messages (sbuf,scnts) are placed in rbuf in root order:

- rcnts(i) elements, starting at offset rdispls(i)
- For i = {0,…,n-1} of group of n tasks.

Size of data sent 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,vp1,vp0]   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 from all processors

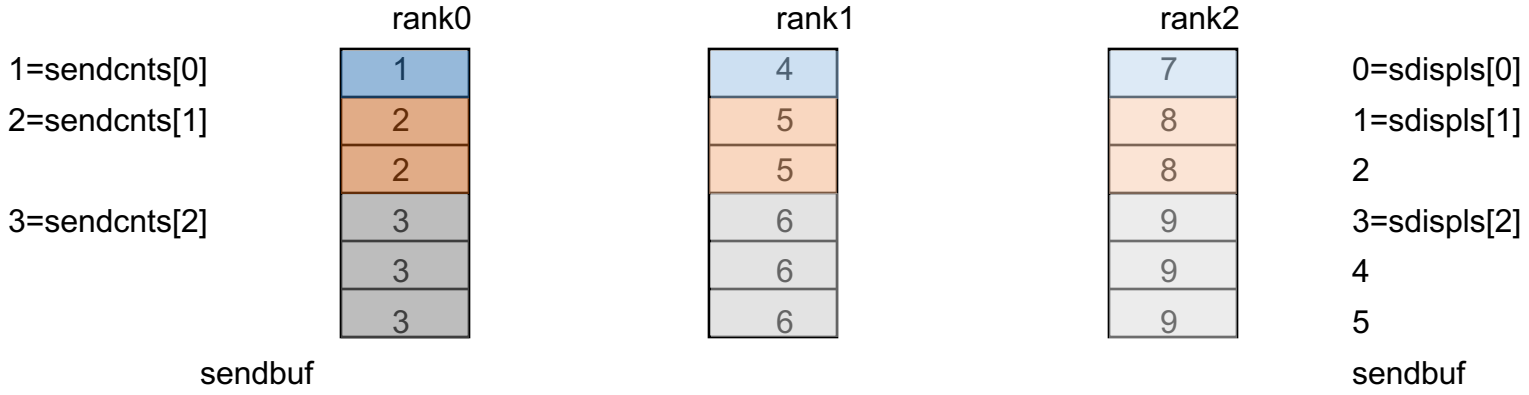
C

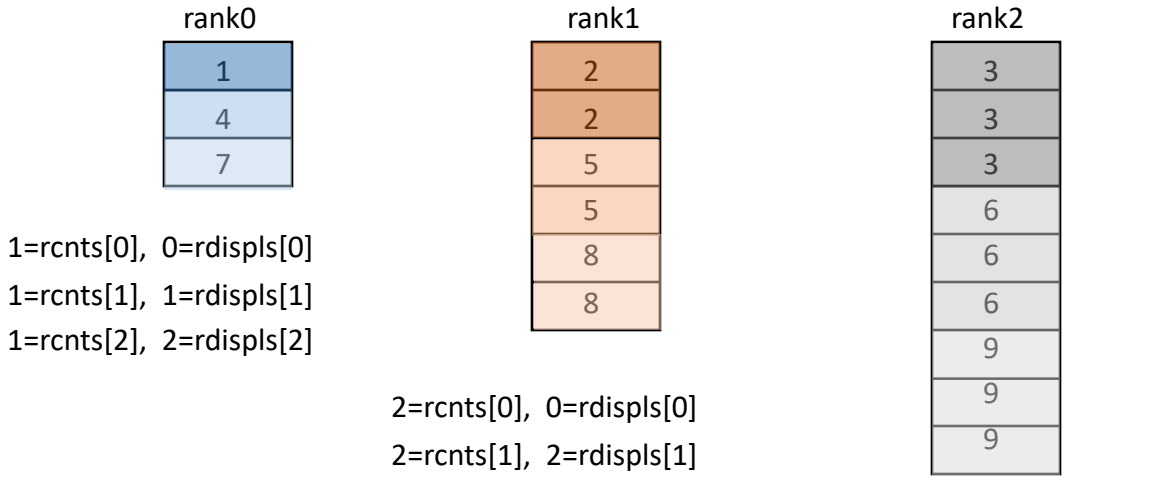
```
ierr=MPI_Alltoallv(&sbuf[0],&scnts[0],&sdispls[0],stype,
                   &rbuf[0],&rcnts[0],&rdispls[0],rtype,
                   comm);
```

Fortran

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

MPI_Alltoallv





License