

MPI: a message-passing parallel programming model

Eduard Ayguadé and Daniel Jiménez
Computer Architecture Department (UPC)



**Barcelona
Supercomputer
Center**

National Supercomputer Facility

Message Passing Interface (MPI)

- **Environment**
- **Barrier synchronization**
- **Point-to-point communication**
- **Collective communication**
- **More examples**

Message Passing Interface (MPI)

■ Environment

- MPI_INIT
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_FINALIZE
- MPI_ABORT

■ Barrier synchronization

■ Point-to-point communication

■ Collective communication

■ More examples

Very Simple Program (VSP)

```
#include "mpi.h"

int rank;
int nproc;

int main( int argc, char* argv[] ) {
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* Nothing to do */
    printf("Process: %d out of %d:\\\n\n
           Hola mundo!\\n", rank, nproc);

    MPI_Finalize();
}
```

■ Usage

- ```
int MPI_Init(int* argc_ptr, /* in */
 char** argv_ptr[]); /* in */
```

## ■ Description

- Initialize MPI
- All MPI programs must call this routine once and only once before any other MPI routines

## ■ Usage

- ```
int MPI_Comm_size( MPI_Comm comm, /* in */  
                  int* size );    /* out */
```

■ Description

- Return the number of processes (`size`) in the group associated with a communicator `comm`
- Communicator
 - ✓ Context for a communication operation
 - ✓ Messages are always received within the context they were sent
 - ✓ Messages sent in different contexts do not interfere
 - ✓ `MPI_COMM_WORLD`
- Process group
 - ✓ Set of processes that share a communication context

■ Usage

- ```
int MPI_Comm_rank (MPI_Comm comm, /* in */
 int* rank); /* out */
```

## ■ Description

- Returns the identifier of the local process in the group associated with a communicator `comm`
- The identifier (`rank`) of the process is in the range from 0 ... `size - 1`

## ■ Usage

- `int MPI_Finalize (void);`

## ■ Description

- Terminates all MPI processing
- Make sure this routine is the last MPI call
- All pending communications involving a process have completed before the process calls `MPI_FINALIZE`



## ■ Usage

- ```
int MPI_Abort( MPI_Comm comm, /* in */  
               int errorcode ); /* in */
```

■ Description

- Forces all processes of an MPI job to terminate

Message Passing Interface (MPI)

- Environment
- **Barrier synchronization**
- Point-to-point communication
- Collective communication
- More examples

VSP: barrier synchronization

```
#include "mpi.h"

int rank;
int nproc;

int main( int argc, char* argv[] ) {
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    // Ensure that all processes arrive
    // here before timing
    MPI_Barrier(MPI_COMM_WORLD);
    /* Something to do in local */

    MPI_Finalize();
}
```

■ Usage

- `int MPI_Barrier(MPI_Comm comm); /* in */`

■ Description

- Blocks each process in communicator `comm` until all processes have called it

Message Passing Interface (MPI)

- Environment
- Barrier synchronization
- **Point-to-point communication**
- Collective communication
- More examples

MPI: Point-to-point communication

■ Blocking

- Return from the procedure indicates the user is allowed to reuse resources specified in the call

■ Non-blocking

- The procedure may return before the operation completes, and before the user is allowed to reuse resources specified in the call

■ List of some basic routines:

- `MPI_SEND`
- `MPI_RECV`
- `MPI_ISEND`
- `MPI_Irecv`
- `MPI_WAIT`
- `MPI_TEST`
- `MPI_GET_COUNT`

VSP: blocking operations

```
#include "mpi.h"

int rank, nproc;

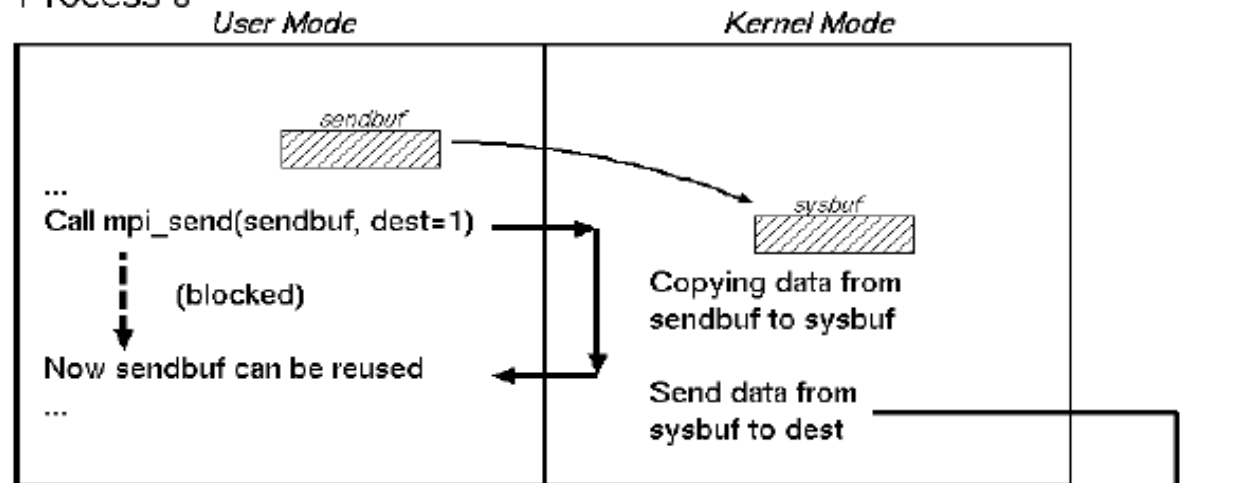
int main( int argc, char* argv[] ) {
    int isbuf, irbuf;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

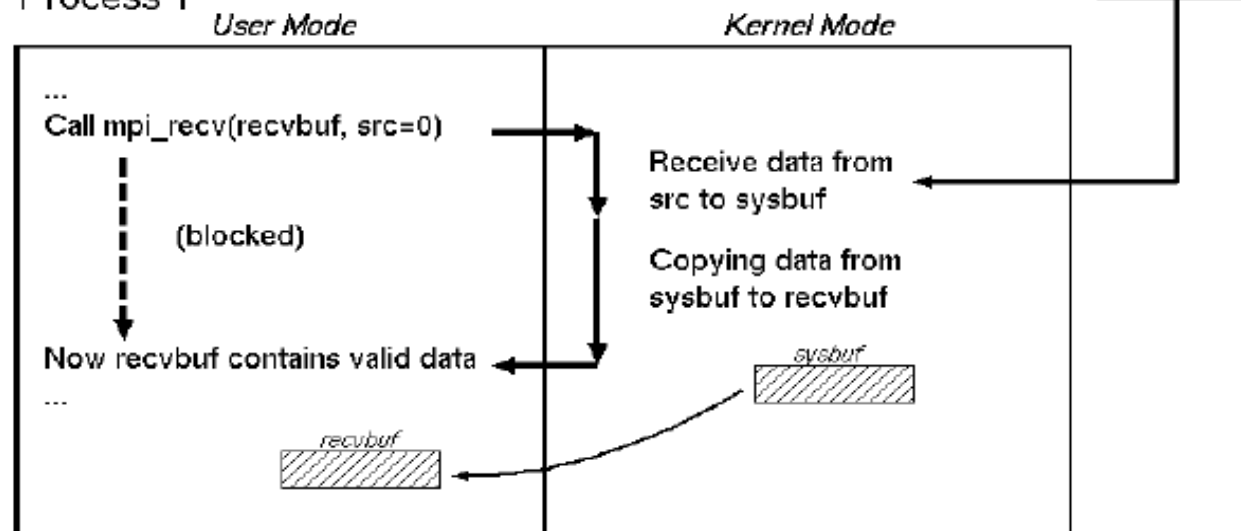
    if(rank == 0) {
        isbuf = 9;
        MPI_Send( &isbuf, 1, MPI_INTEGER, 1, 1, MPI_COMM_WORLD);
    } else if(rank == 1) {
        MPI_Recv( &irbuf, 1, MPI_INTEGER, 0, 1, MPI_COMM_WORLD,
                  &status);
        printf( "%d\n", irbuf );
    }
    MPI_Finalize();
}
```

MPI: blocking operations

Process 0



Process 1



■ Usage

- ```
int MPI_Send(void* buf, /* in */
 int count, /* in */
 MPI_Datatype datatype, /* in */
 int destination, /* in */
 int tag, /* in */
 MPI_Comm comm); /* in */
```

## ■ Description

- Performs a blocking send operation
- The message can be received by either `MPI_RECV` or `MPI_IRECV`
- Message envelope
  - ✓ Information used to distinguish messages and selectively receive them
  - ✓ `<destination, tag, comm>`

## ■ Usage

- ```
int MPI_Recv( void* buf,          /* out */
              int count,          /* in */
              MPI_Datatype datatype, /* in */
              int source,         /* in */
              int tag,            /* in */
              MPI_Comm comm,      /* in */
              MPI_Status* status ); /* out */
```

■ Description

- Performs a blocking receive operation
- The message received must be less than or equal to the length of the receive buffer `buf`
- `MPI_RECV` can receive a message sent by either `MPI_SEND` or `MPI_ISEND`
- **Message envelope:** `<source, tag, comm>`

■ Usage

- ```
int MPI_Sendrecv(void *sendbuf, /* in */
 int sendcount,
 MPI_Datatype sendtype,
 int dest,
 int sendtag,
 void *recvbuf, /* out */
 int recvcount,
 MPI_Datatype recvtype,
 int source,
 int recvtag,
 MPI_Comm comm,
 MPI_Status *status);
```

## ■ Description

- Sends and receives a message
- Blocking exchange

## ■ MPI\_Datatype can be one of the following:

- MPI\_CHAR
- MPI\_SHORT
- MPI\_INT
- MPI\_LONG
- MPI\_UNSIGNED\_CHAR
- MPI\_UNSIGNED\_SHORT
- MPI\_UNSIGNED
- MPI\_UNSIGNED\_LONG
- MPI\_FLOAT
- MPI\_DOUBLE
- MPI\_LONG\_DOUBLE
- MPI\_BYTE
- MPI\_PACKED

# VSP: non-blocking operations

```
#include "mpi.h"

int main(int argc, char* argv[])
{
 int rank, nproc;
 int isbuf, irbuf, count;
 MPI_Request request;
 MPI_Status status;

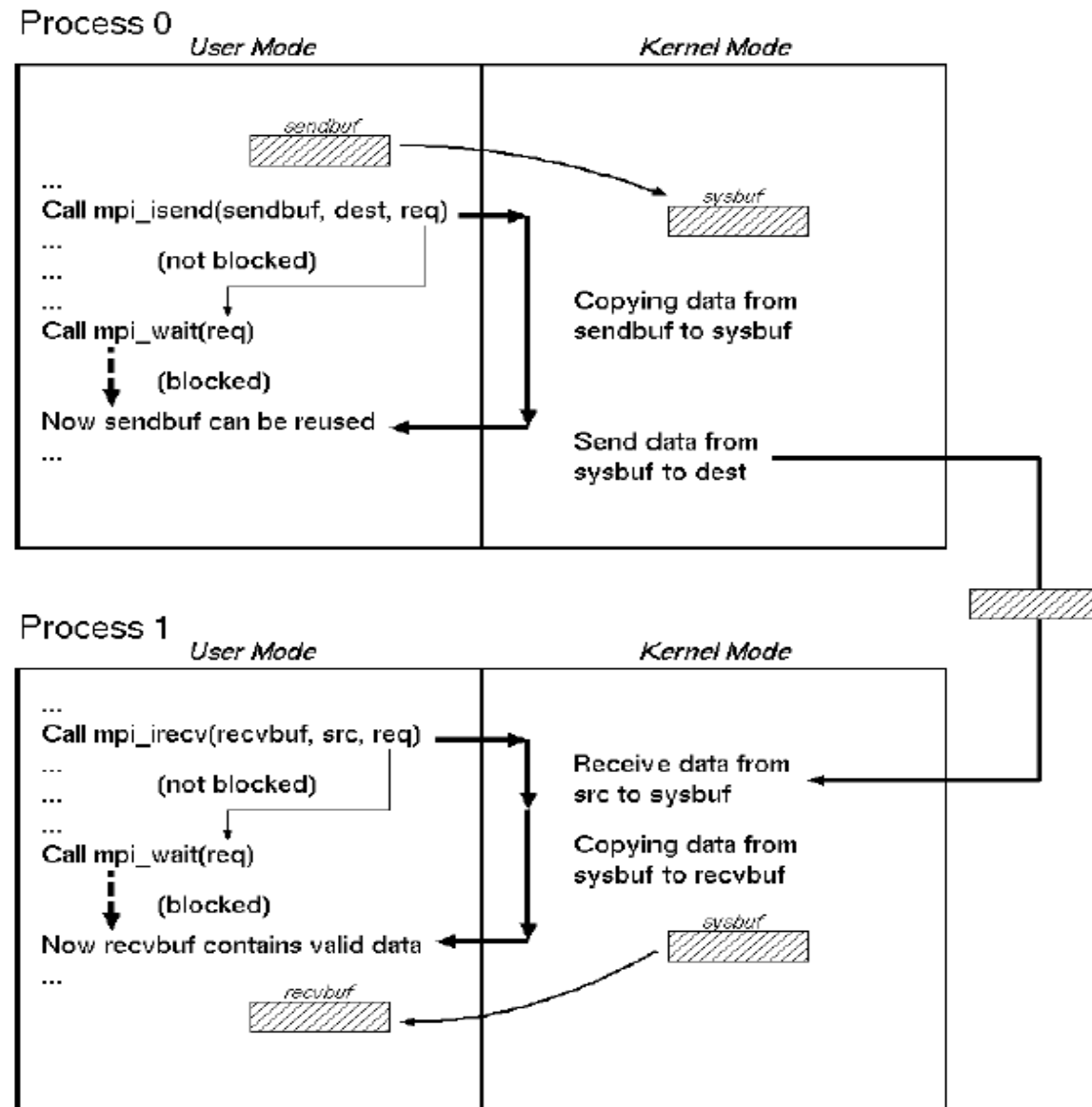
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 if(rank == 0) {
 isbuf = 9;
 MPI_Isend(&isbuf, 1, MPI_INTEGER, 1, 1,
 MPI_COMM_WORLD, &request);
 }
}
```

## VSP: non-blocking operations

```
} else if(rank == 1) {
 MPI_Irecv(&irbuf, 1, MPI_INTEGER, MPI_ANY_SOURCE,
 MPI_ANY_TAG, MPI_COMM_WORLD, &request);
 /* OTHER WORK TO DO */
 MPI_Wait(&request, &status);
 MPI_Get_count(status, MPI_INTEGER, &count);
 printf("irbuf = %d source = %d tag = %d
 count = %d\n", irbuf, status.MPI_SOURCE,
 status.MPI_TAG, count);
}
MPI_Finalize();
}
```

# MPI: non-blocking operations



## ■ Usage

- ```
int MPI_Isend( void* buf,           /* in */
               int count,          /* in */
               MPI_Datatype datatype, /* in */
               int dest,           /* in */
               int tag,            /* in */
               MPI_Comm comm,      /* in */
               MPI_Request* request ); /* out */
```

■ Description

- Performs a non-blocking send operation
- `request` is an identifier for later enquiry with `MPI_WAIT` or `MPI_TEST`
- The send buffer `buf` may not be modified until the request has been completed by `MPI_WAIT` or `MPI_TEST`
- The message can be received by either `MPI_RECV` or `MPI_IRECV`

■ Usage

- ```
int MPI_Irecv(void* buf, /* out */
 int count, /* in */
 MPI_Datatype datatype, /* in */
 int source, /* in */
 int tag, /* in */
 MPI_Comm comm, /* in */
 MPI_Request* request); /* out */
```

## ■ Description

- Performs a non-blocking receive operation
- Do not access any part of the receive buffer `buf` until the receive is completed by `MPI_WAIT` or `MPI_TEST`
- The message received must be less than or equal to the length of the receive buffer `buf`
- `MPI_IRecv` can receive a message sent by either `MPI_SEND` or `MPI_ISEND`

## ■ Usage

- ```
int MPI_Wait( MPI_Request* request,    /* inout */  
              MPI_Status* status );    /* out */
```

■ Description

- Waits for a non-blocking operation to complete, with identifier stored in `request`
- Information on the completed operation is found in `status`
- If wildcards (`MPI_ANY_SOURCE`, `MPI_ANY_TAG`) were used by the receive for either the `source` or `tag`, the actual source and tag can be retrieved from `status` → `MPI_SOURCE` and `status` → `MPI_TAG`

■ Usage

- ```
int MPI_Test (MPI_Request* request, /* inout */
 int *flag, /* out */
 MPI_Status* status); /* out */
```

## ■ Description

- Test for the completion of a send or receive
- `flag` equals `MPI_SUCCESS` if MPI routine completed successfully

## ■ Usage

- ```
int MPI_Get_count( MPI_Status status,      /* in */
                  MPI_Datatype datatype,  /* in */
                  int* count );           /* out */
```

■ Description

- Returns the number of elements in a message (indicated by `status`)
- The `datatype` argument and the argument provided by the call that set the `status` variable should match

■ Usage

- ```
int MPI_Probe(int source, /* input */
 int tag, /* input */
 MPI_Comm comm, /* input */
 MPI_Status *status); /* out */
```

## ■ Description

- Blocking call that returns only after a matching message is found
- Wildcards can be used to wait for messages coming from any source (`MPI_ANY_SOURCE`) or with any tag (`MPI_ANY_TAG`)
- There is a non-blocking `MPI_Iprobe`

# Message Passing Interface (MPI)

- Environment
- Barrier synchronization
- Point-to-point communication
- **Collective communication**
- More examples

## ■ Collective

- If all processes in a process group need to invoke the procedure

## ■ List of some routines:

- MPI\_BCAST
- MPI\_REDUCE
- MPI\_SCATTER
- MPI\_SCATTERV
- MPI\_GATHER
- MPI\_GATHERV
- MPI\_ALLGATHER
- MPI\_ALLTOALL
- MPI\_ALLTOALLV

## PI: broadcast and reduce

```
#include <mpi.h>
void main (int argc, char *argv[])
{
 int i, my_id, numprocs, num_steps;
 double x, pi, step, sum = 0.0 ;

 MPI_Init(&argc, &argv) ;
 MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
 MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
 if (my_id==0) scanf("%d",&num_steps);
 MPI_Bcast(&num_steps, 1, MPI_INT, 0, MPI_COMM_WORLD)
 step = 1.0/(double) num_steps ;
 my_steps = num_steps/numprocs ;

 for (i=my_id*my_steps; i<(my_id+1)*my_steps; i++){
 x = (i+0.5)*step;
 sum += 4.0/(1.0+x*x);
 }
 sum *= step ;
 MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE,
 MPI_SUM, 0, MPI_COMM_WORLD) ;
 MPI_Finalize() ;
}
```

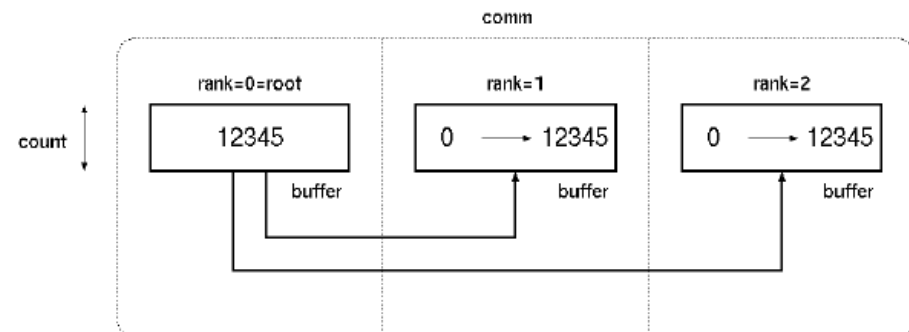


## ■ Usage

- ```
int MPI_Bcast( void* buffer,          /* inout */
               int count,             /* in */
               MPI_Datatype datatype, /* in */
               int root,              /* in */
               MPI_Comm comm);        /* in */
```

■ Description

- Broadcasts a message from `root` to all processes in communicator `comm`
- The type signature of `count` and `datatype` on any process must be equal to the type signature of `count` and `datatype` at the `root`



■ Usage

- ```
int MPI_Reduce(void* sendbuf, /* in */
 void* recvbuf, /* out */
 int count, /* in */
 MPI_Datatype datatype, /* in */
 MPI_Op op, /* in */
 int root, /* in */
 MPI_Comm comm); /* in */
```

## ■ Description

- Applies a reduction operation to the vector `sendbuf` over the set of processes specified by communicator `comm` and places the result in `recvbuf` on `root`

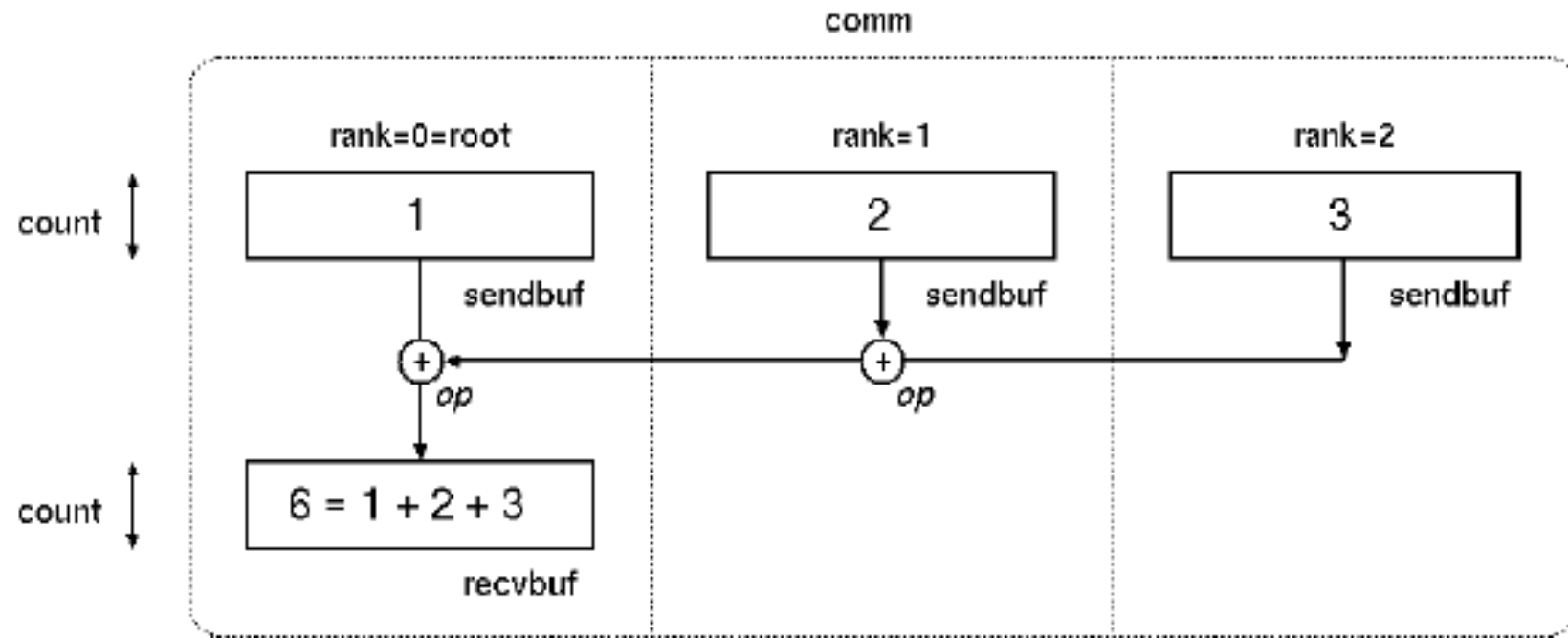
### ■ Description (Cont'd)

- Both the input and output buffers have the same number of elements with the same type
- Users may define their own operations or use the predefined operations provided by MPI

### ■ Predefined operations

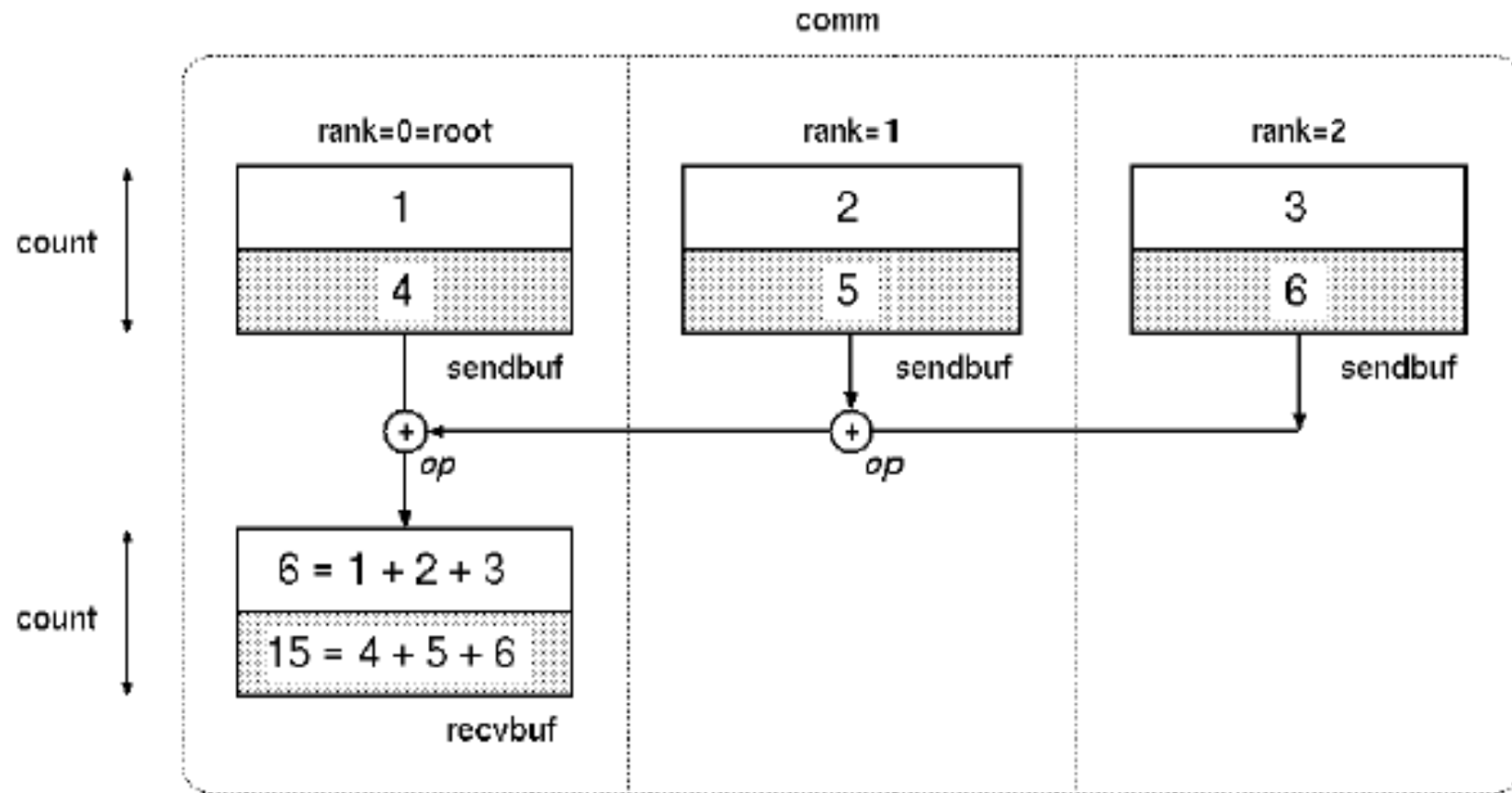
- `MPI_SUM`, `MPI_PROD`
- `MPI_MAX`, `MPI_MIN`
- `MPI_MAXLOC`, `MPI_MINLOC`
- `MPI_LAND`, `MPI_LOR`, `MPI_LXOR`
- `MPI_BAND`, `MPI_BOR`, `MPI_BXOR`

## MPI\_REDUCE (cont'd)



MPI\_REDUCE for scalars

## MPI\_REDUCE (cont'd)



MPI\_REDUCE for arrays

## VSP: scatter and gather

```
int gsize, localbuf[100];
int root=0, rank, *rootbuf;
...
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == root)
 rootbuf = (int *)malloc(nproc*100*sizeof(int));

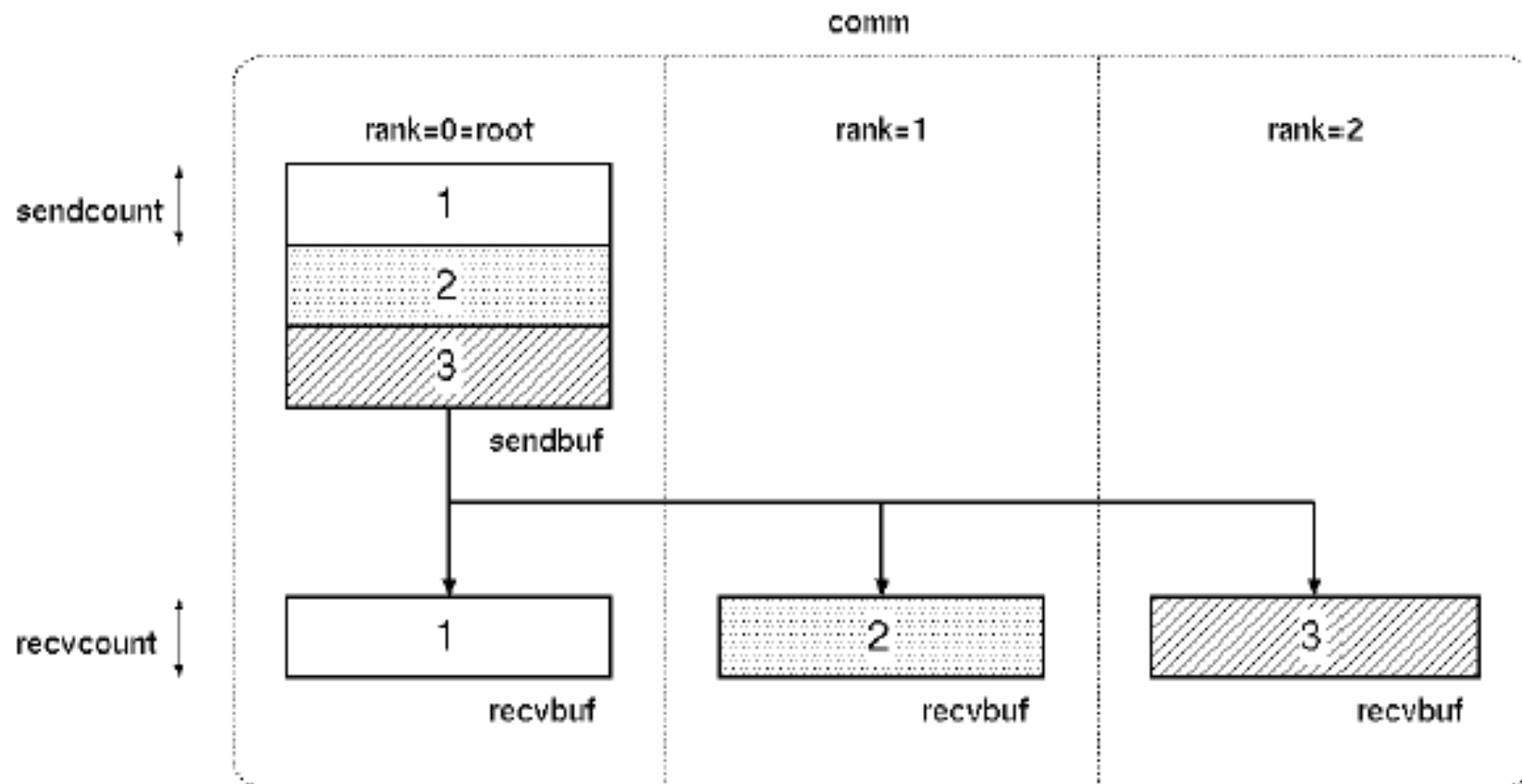
/* MATRIX INITIALIZED IN ROOT */
MPI_Scatter (rootbuf, 100, MPI_INT, localbuf, 100,
 MPI_INT, root, comm);

/* DO WORK WITH DATA */

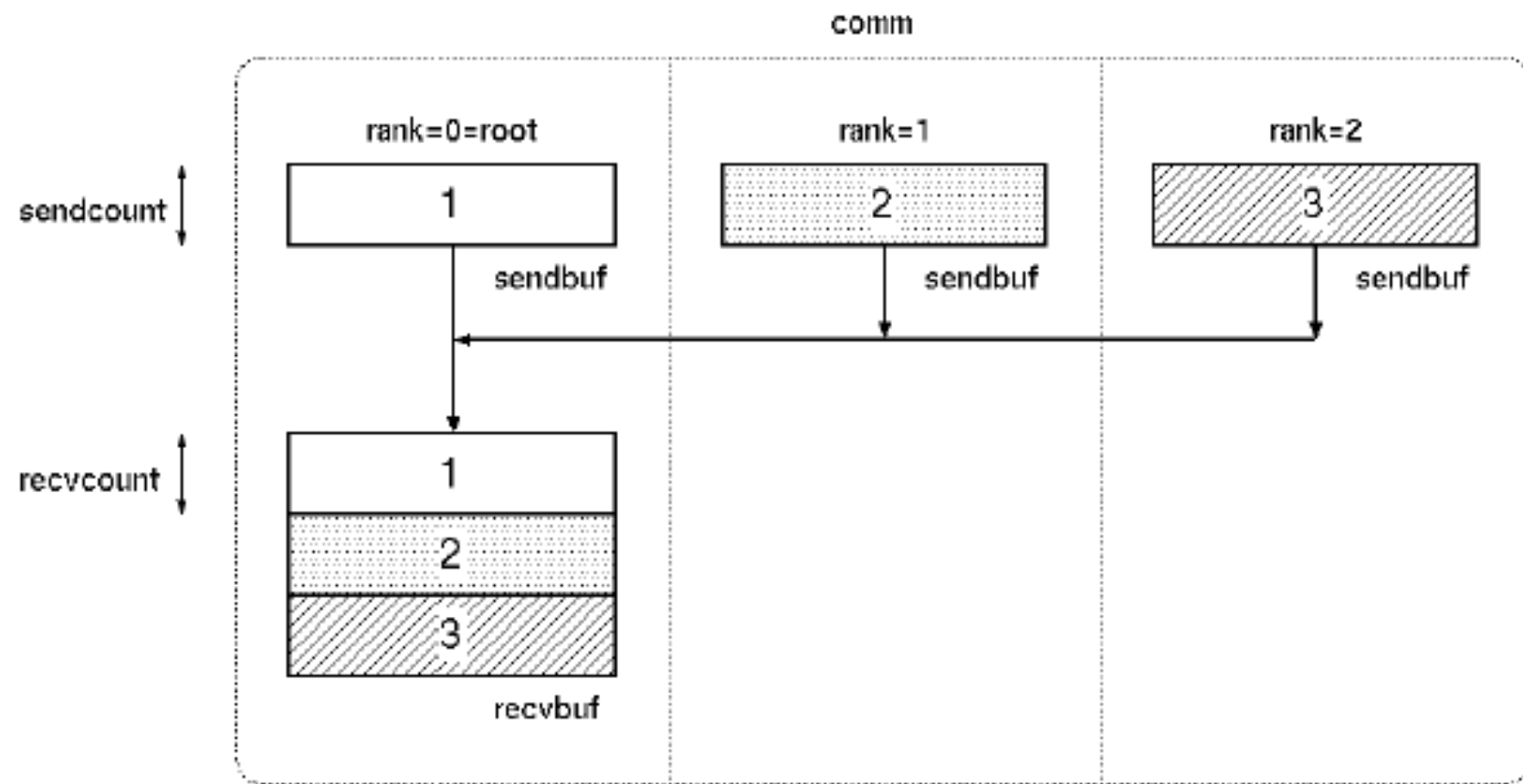
MPI_Gather (localbuf, 100, MPI_INT, rootbuf, 100,
 MPI_INT, root, comm);

/* RESULTS BACK IN ROOT */
```

# MPI\_SCATTER



# MPI\_GATHER





## ■ Usage

- ```
int MPI_Scatter( void* sendbuf,      /* in */
                 int sendcount,      /* in */
                 MPI_Datatype sendtype, /* in */
                 void* recvbuf,      /* out */
                 int recvcount,      /* in */
                 MPI_Datatype recvtype, /* in */
                 int root,           /* in */
                 MPI_Comm comm);     /* in */
```

■ Description

- Distribute individual messages from `root` to each process in communicator
- Inverse operation to `MPI_GATHER`

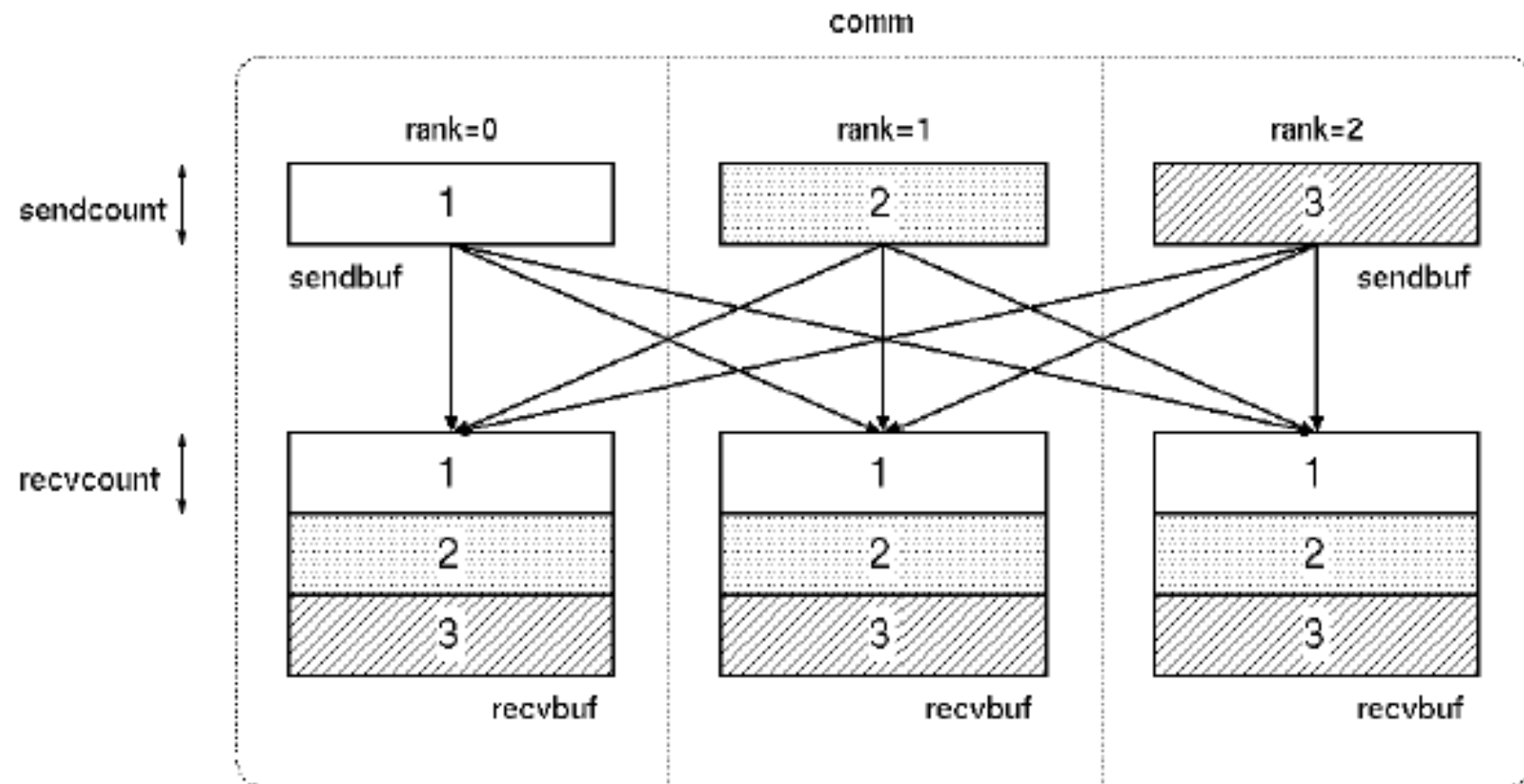
■ Usage

- ```
int MPI_Gather(void* sendbuf, /* in */
 int sendcount, /* in */
 MPI_Datatype sendtype, /* in */
 void* recvbuf, /* out */
 int recvcount, /* in */
 MPI_Datatype recvtype, /* in */
 int root, /* in */
 MPI_Comm comm); /* in */
```

## ■ Description

- Collects individual messages from each process in communicator `comm` to the `root` process and store them in rank order

# MPI\_ALLGATHER



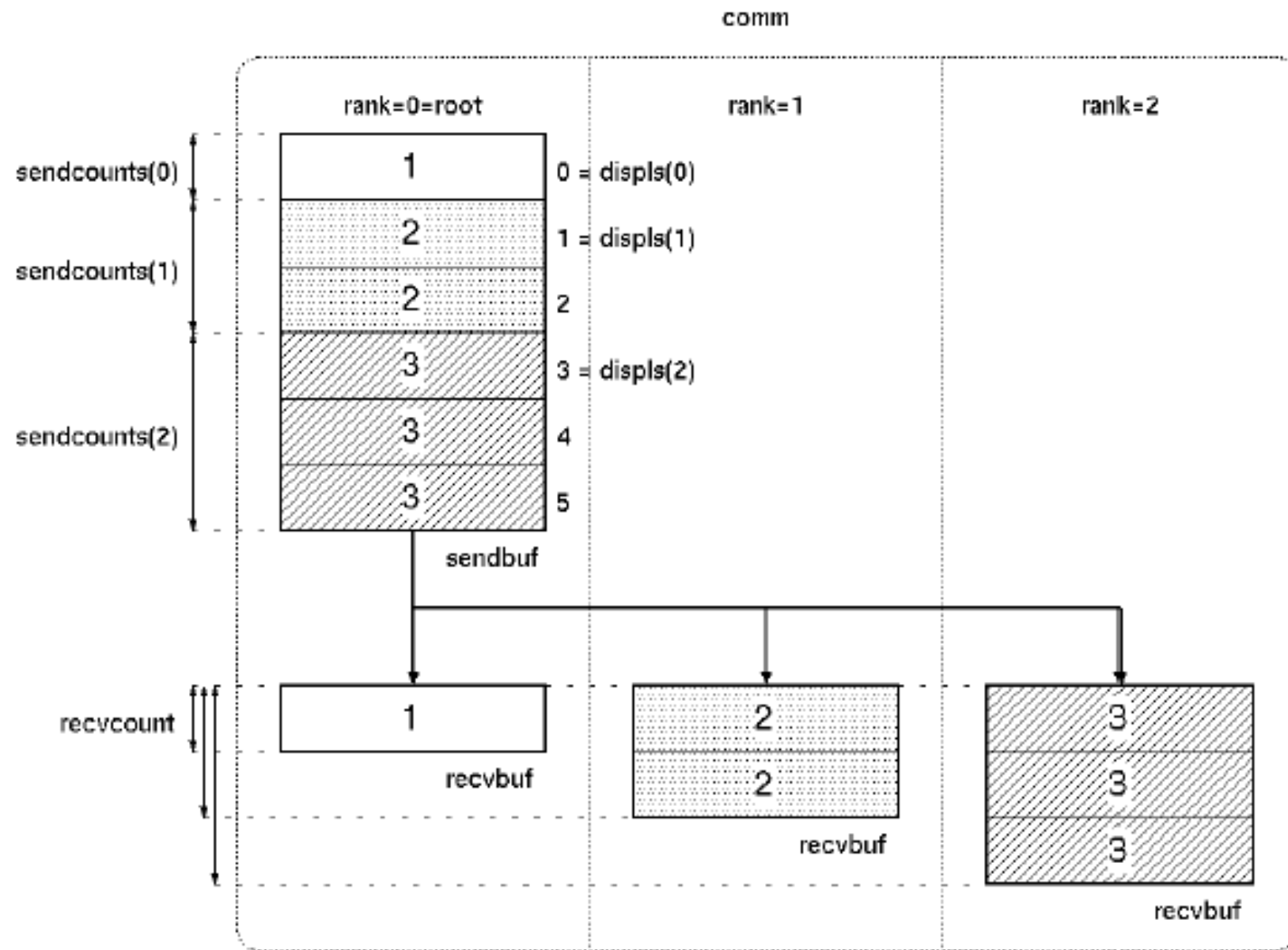
## ■ Usage

- ```
int MPI_Allgather( void* sendbuf,          /* in */
                  int sendcount,          /* in */
                  MPI_Datatype sendtype,  /* in */
                  void* recvbuf,          /* out */
                  int recvcount,          /* in */
                  MPI_Datatype recvtype,  /* in */
                  MPI_Comm comm);         /* in */
```

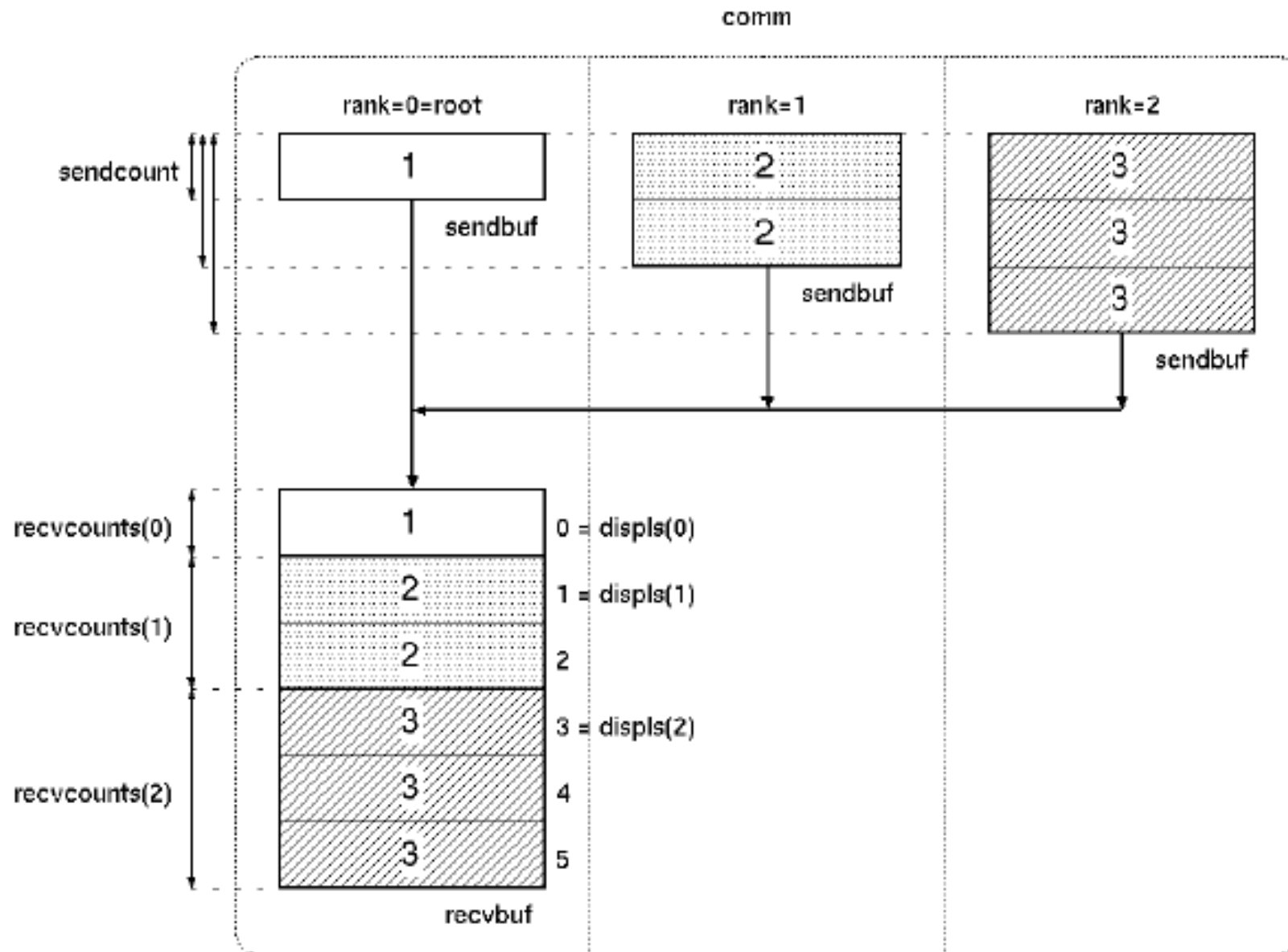
■ Description

- Gathers individual messages from each process in communicator `comm` and distributes the resulting message to each process
- Similar to `MPI_GATHER` except that all processes receive the result

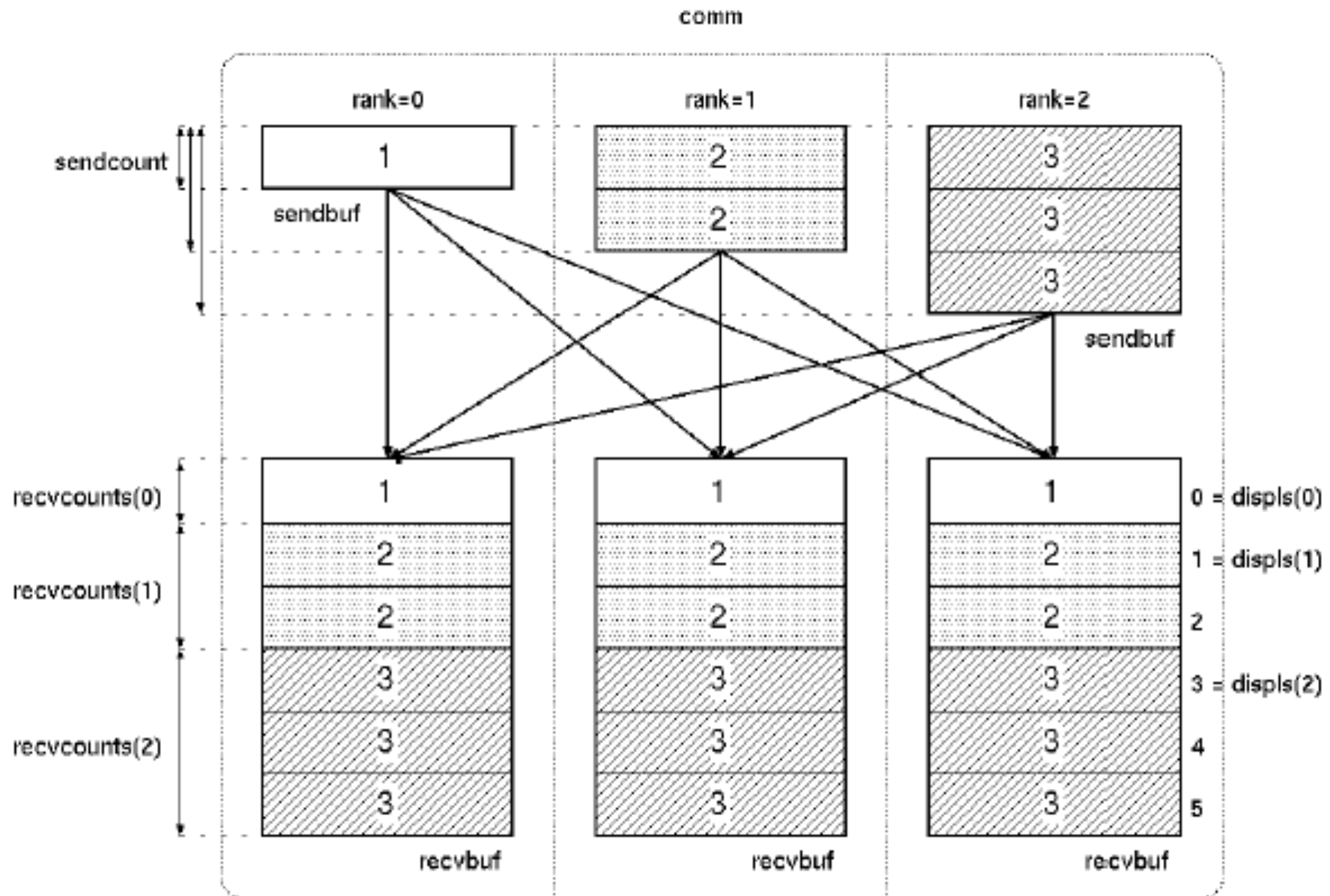
MPI_SCATTERV



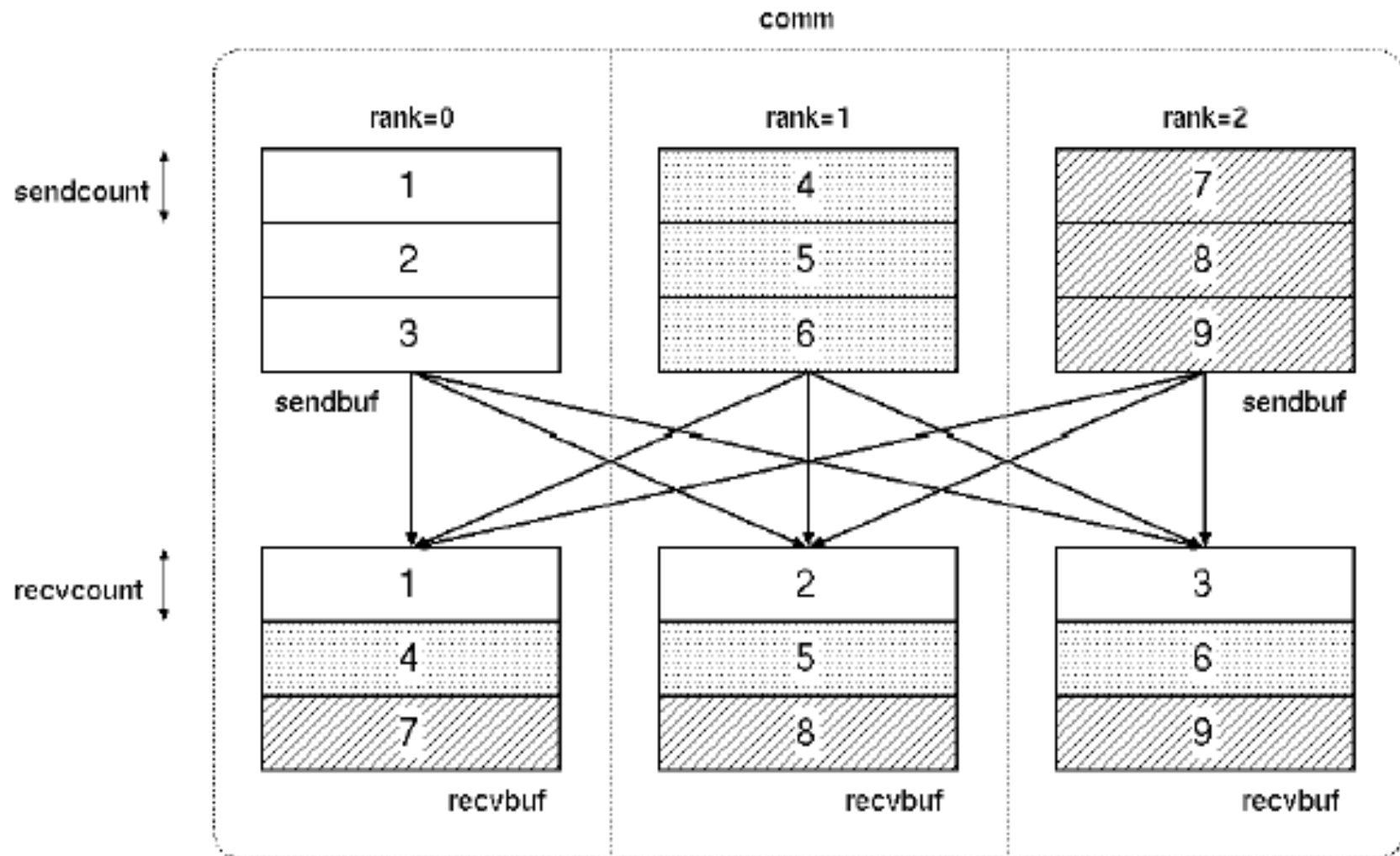
MPI_GATHERV



MPI_ALLGATHERV



MPI_ALLTOALL



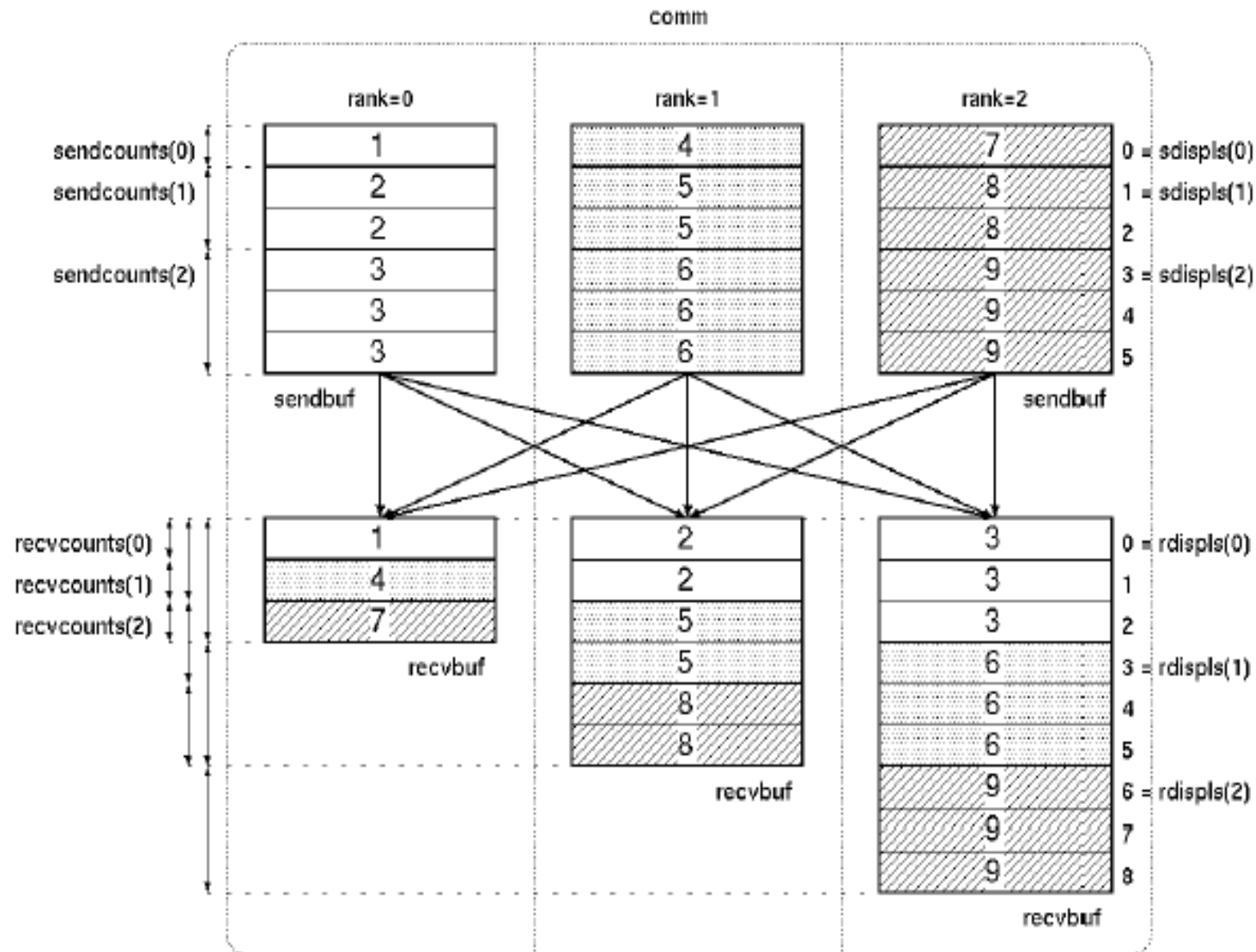
■ Usage

- ```
int MPI_Alltoall(void* sendbuf, /* in */
 int sendcount, /* in */
 MPI_Datatype sendtype, /* in */
 void* recvbuf, /* out */
 int recvcount, /* in */
 MPI_Datatype recvtype, /* in */
 MPI_Comm comm); /* in */
```

## ■ Description

- Sends a distinct message from each process to every other process
- The j-th block of data sent from process i is received by process j and placed in the i-th block of the buffer recvbuf
- Useful to implement, for example, transpositions

# MPI\_ALLTOALLV



# MPI: a message-passing parallel programming model

Eduard Ayguadé and Daniel Jiménez  
Computer Architecture Department (UPC)



**Barcelona  
Supercomputer  
Center**

*National Supercomputer Facility*