

Collective communication

Slides Pekka Manninen



Introduction

- Collective communication transmits data among all processes in a process group
 - These routines must be called by all the processes in the group
- Collective communication includes
 - data movement
 - collective computation
 - synchronization

Example

MPI_Barrier that makes each task hold until all tasks have called it

```
int MPI_Barrier(comm)
MPI_BARRIER(comm, rc)
```

Introduction

- Collective communication may outperform point-to-point communication
 - Depends on implementation and case
- Code is more compact and easier to read:

```
if (my_id == 0) then
  do i = 1, ntasks-1
    call mpi_send(a, 1048576, &
      MPI_REAL, i, tag, &
      MPI_COMM_WORLD, rc)
  end do
else
  call mpi_recv(a, 1048576, &
    MPI_REAL, 0, tag, &
    MPI_COMM_WORLD, status, rc)
end if
```

```
call mpi_bcast(a, 1048576, &
  MPI_REAL, 0, &
  MPI_COMM_WORLD, rc)
```

Communicating a vector a consisting of 1M float elements from the task 0 to all other tasks with point-to-point and collective communication

Introduction

- Amount of sent and received data must match
- Only blocking routines are available in MPI-2
 - Routine returns as soon as its participation in the overall communication is complete
- No tag arguments
 - Order of execution must coincide across processes

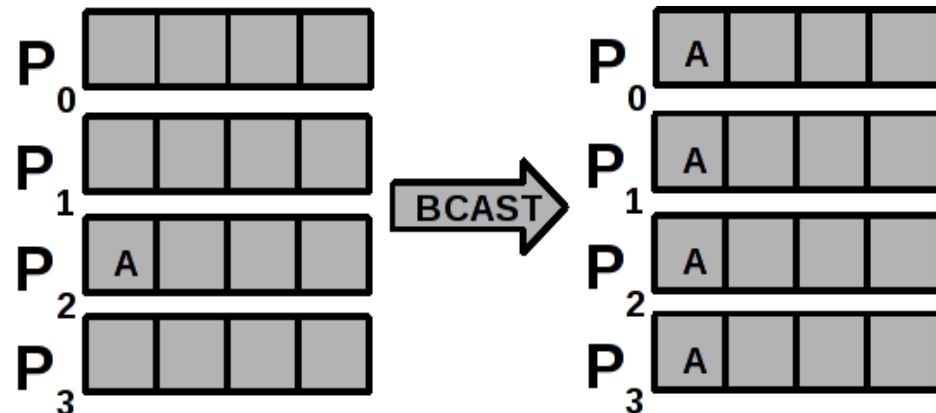
Part I: One-to-many communication



Broadcasting

- With **MPI_Bcast**, the task *root* sends a *buffer* of data to all other tasks

MPI_Bcast(*buffer*, *count*, *datatype*, *root*, *comm*)
buffer data to be distributed
count number of entries in buffer
datatype data type of buffer
root rank of broadcast root
comm communicator



Broadcasting

- C & Fortran bindings

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

```
MPI_BCAST(buffer, count, datatype, root, comm, ierror)  
type buffer(*)  
integer count, datatype, root, comm, ierror
```

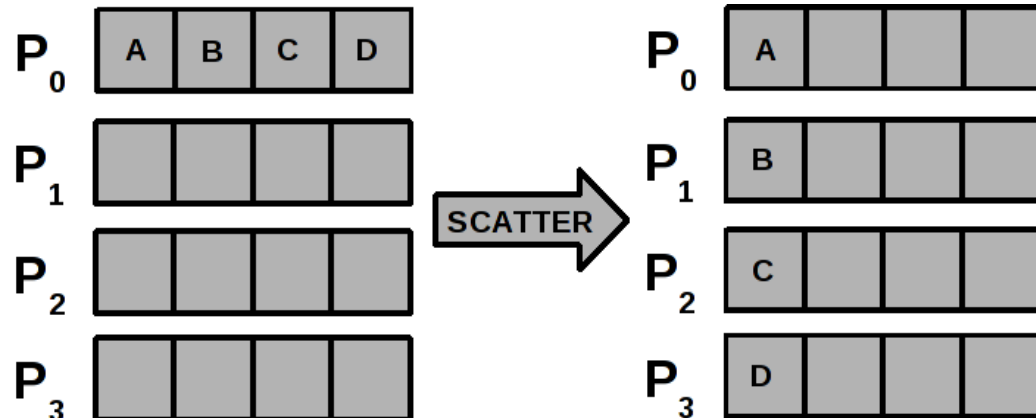


Scattering

- **MPI_Scatter**: Task *root* sends an equal share of data (*sendbuf*) to all other processes

MPI_Scatter(*sendbuf*, *sendcount*, *sendtype*, *recvbuf*, *recvcount*, *recvtype*, *root*, *comm*)

sendbuf send buffer (data to be scattered)
sendcount number of elements sent to each process
sendtype data type of send buffer elements
recvbuf receive buffer
recvcount number of elements in receive buffer
recvtype data type of receive buffer elements
root rank of sending process
comm communicator



Not the size of the whole *sendbuf*, but the chunk of data to be sent

Scattering



- C & Fortran bindings

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

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

```
type sendbuf(*), recvbuf(*)
integer sendcount, recvcount, sendtype, recvtype,
        root, comm, ierror
```



One-to-all example

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if

call mpi_bcast(a, 16, &
  MPI_INTEGER, 0, &
  MPI_COMM_WORLD, rc)

if (my_id==3) print *, a(:)
```

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if
call mpi_scatter(a, 4, &
  MPI_INTEGER, &
  aloc, 4, MPI_INTEGER, &
  MPI_COMM_WORLD, rc)

if (my_id==3) print *, aloc(:)
```

Assume 4 MPI tasks. What would the (full) program print?

```
> aprun -n 4 ./a.out
```

```
 1      2      3      4
 5      6      7      8
 9     10     11     12
13     14     15     16
```

```
> aprun -n 4 ./a.out
```

```
13      14      15      16
```

Variable-width scatter

- Like MPI_Scatter, but messages can have different sizes and displacements

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

<i>sendbuf</i>	send buffer
<i>sendcounts</i>	array (of length group size) specifying the number of elements to send to each processor
<i>displs</i>	array (of length group size). Entry <i>i</i> specifies the displacement (relative to <i>sendbuf</i>)
<i>sendtype</i>	data type of send buffer elements
<i>recvbuf</i>	receive buffer
<i>recvcount</i>	number of elements in receive buffer
<i>recvtype</i>	data type of receive buffer elements
<i>root</i>	rank of sending process
<i>comm</i>	communicator

Variable-width scatter

- C & Fortran bindings

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

```
MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf,  
             recvcount, recvtype, root, comm, ierror)  
type sendbuf(*), recvbuf(*)  
integer sendcounts(*), displs(*), recvcount, sendtype, recvtype,  
root, comm, ierror
```



Scatterv example

```
if (my_id==0) then
  do i = 1, 10
    a(i) = i
  end do
  sendcounts = (/ 1, 2, 3, 4 /)
  displs = (/ 0, 1, 3, 6 /)
end if
call mpi_scatterv(a, sendcnts,
&
  displs, MPI_INTEGER,&
  aloc, 4, MPI_INTEGER, &
  0, MPI_COMM_WORLD, rc)
```

Assume 4 MPI tasks. What are the values in **aloc** in different tasks?

1	0	0	0	TASK 0
2	3	0	0	TASK 1
4	5	6	0	TASK 2
7	8	9	10	TASK 3

Part II: Many-to-one communication



Gathering

- **MPI_Gather** collects individual data from each task to the *root* task

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
           recvcount, recvtype, root, comm)
```

sendbuf send buffer

sendcount number of elements in send buffer

sendtype data type of send buffer elements

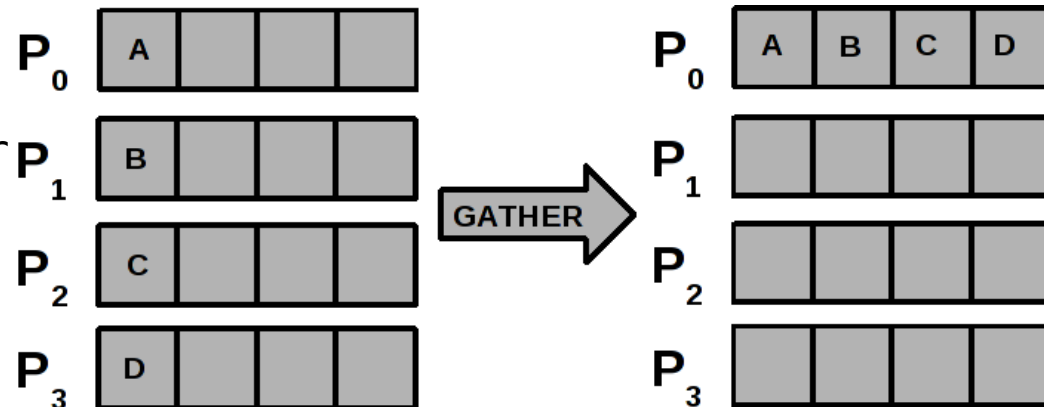
recvbuf receive buffer

recvcount number of elements for any single receive

recvtype data type of recv buffer elements

root rank of
receiving
process

comm communicator



Gathering



- C and Fortran bindings

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

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

```
type sendbuf(*), recvbuf(*)
```

```
integer sendcount, recvcount, sendtype, recvtype, root,
        comm, ierror
```



Variable-width gather

- MPI_Gatherv is similar to MPI_Gather, but allows for varying amounts of data

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf,  
            recvcnts, displs, recvtype, root, comm)
```

sendbuf send buffer

sendcount number of elements in send buffer

sendtype data type of send buffer elements

recvbuf receive buffer

recvcnts array of number of elements to be receive from each task

displs integer array (of length group size). Entry *i* specifies the displacement relative to *recvbuf*

at

which to place the incoming data from process *i*

recvtype data type of recv buffer elements

root rank of receiving process

comm communicator

Variable-width gather

- C & Fortran bindings

```
int MPI_Gatherv ( void *sendbuf, int sendcnt, MPI_Datatype
                 sendtype, void *recvbuf, int *recvcnts, int *displs,
                 MPI_Datatype recvtype, int root, MPI_Comm comm
                 )
```

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

```
type sendbuf(*), recvbuf(*)
```

```
integer sendcount, recvcounts(*), displs(*), sendtype,
            recvtype, root, comm, ierror
```

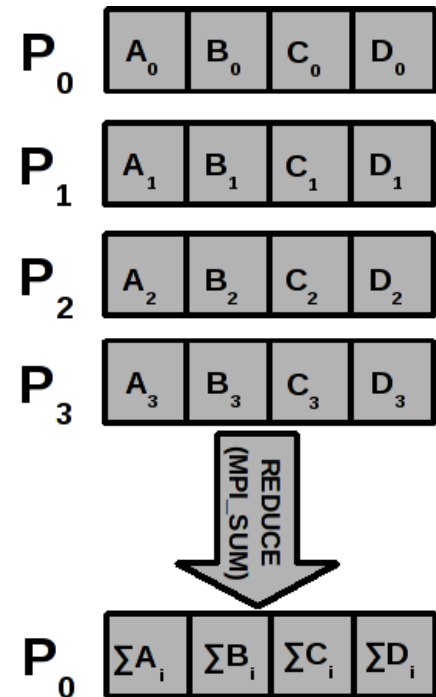


Reduce operation

- Applies a reduction operation *op* to *sendbuf* over the set of tasks and places the result in *recvbuf* on *root*

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

sendbuf send buffer
recvbuf receive buffer
count number of elements in send buffer
datatype data type of elements of send buffer
op reduce operation
root rank of root process
comm communicator



Reduce operation

- C & Fortran bindings

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

```
MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root,  
comm, ierror)  
type sendbuf(*), recvbuf(*)  
integer count, datatype, op, root, comm, ierror
```

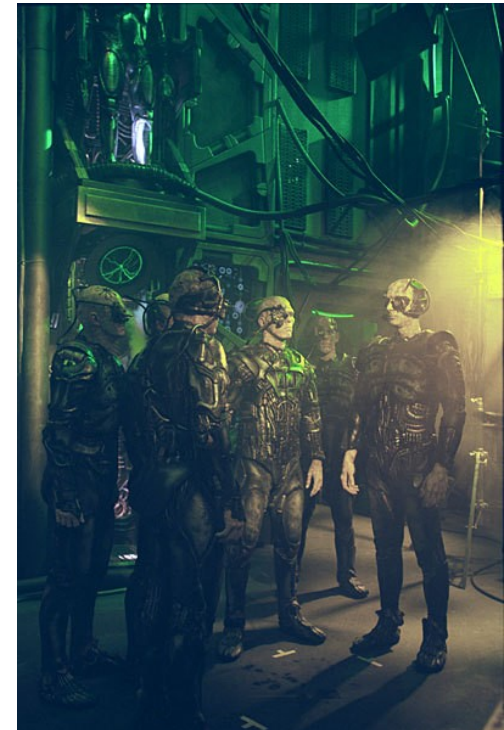


Reduce operation

- Available reduction operations (*op*)

<u>Operation</u>	<u>Meaning</u>
MPI_MAX	Max value
MPI_MIN	Min value
MPI_SUM	Sum
MPI_PROD	Product
MPI_MAXLOC	Max value + location
MPI_MINLOC	Min value + location
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bitwise XOR

Part III: Many-to-many communication

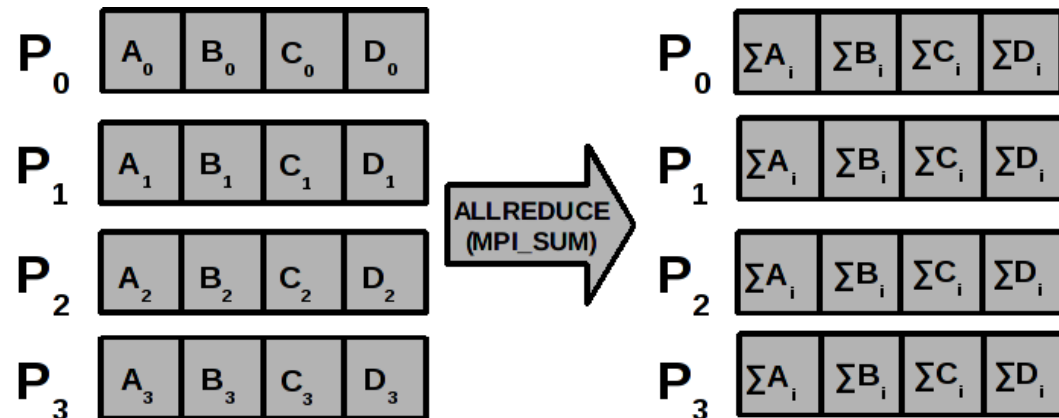


Global reduce operation

- **MPI_Allreduce** Combines values from all processes and distribute the result back to all processes
 - Compare: MPI_Reduce + MPI_Bcast

`MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)`

sendbuf starting address of send buffer
recvbuf starting address of receive buffer
count number of elements in send buffer
datatype data type of elements of send buffer
op operation
comm communicator



Global reduce operation



- C & Fortran bindings

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



Allreduce example

```
real :: a(1024), aloc(128)
real :: rloc, r
integer :: i, my_id, ntasks, rc

call mpi_init(rc)
call mpi_comm_rank(MPI_COMM_WORLD,my_id,rc)
if (my_id==0) then
  call random_number(a)
end if

call mpi_scatter(a, 128, MPI_INTEGER, &
  aloc, 128, MPI_INTEGER, &
  0, MPI_COMM_WORLD, rc)
rloc = dot_product(aloc,aloc)
call mpi_allreduce(rloc, r, 1, MPI_REAL, &
  MPI_SUM, MPI_COMM_WORLD, rc)

print *, 'id=', my_id, 'local=', rloc, &
  'global=', r
call mpi_finalize(rc)
```

Parallel $r = \mathbf{a} \cdot \mathbf{a}$

```
> mpirun -np 8 a.out
id= 6 local= 39.68326   global= 338.8004
id= 7 local= 39.34439   global= 338.8004
id= 1 local= 42.86630   global= 338.8004
id= 3 local= 44.16300   global= 338.8004
id= 5 local= 39.76367   global= 338.8004
id= 0 local= 42.85532   global= 338.8004
id= 2 local= 40.67361   global= 338.8004
id= 4 local= 49.45086   global= 338.8004
```

All-to-one plus one-to-all

- MPI_Allgather gathers data from each task and distributes the resulting data to each task

– Compare: MPI_Gather + MPI_Bcast

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

sendbuf send buffer

sendcount number of elements in send buffer

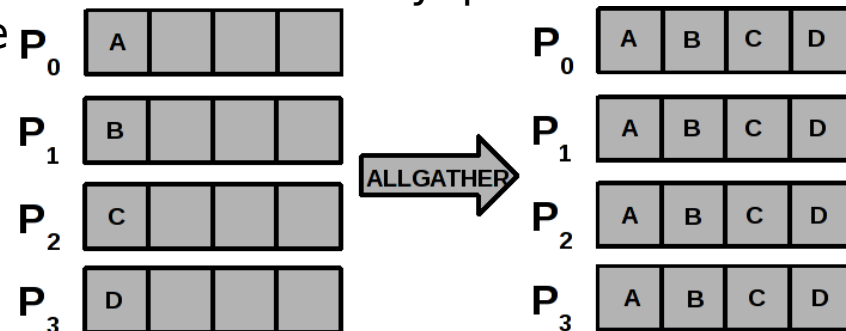
sendtype data type of send buffer elements

recvbuf receive buffer

recvcount number of elements received from any process

recvtype data type of receive buffer elements

comm communicator



MPI_Allgather

- C & Fortran bindings

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

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

```
type sendbuf(*), recvbuf(*)
```

```
integer sendcount, recvcount, sendtype, recvtype, comm,  
        ierror
```



All-to-one plus one-to-all

- **MPI_Reduce_scatter** applies a reduction operation to *sendbuf* over the tasks and scatters the result according to the values in *recvcounts*

– Compare: MPI_Reduce + MPI_Scatter

```
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts,  
                  datatype, op, comm)
```

sendbuf send buffer

recvbuf receive buffer

recvcounts array specifying the number of elements in
 result distributed to each process

datatype data type of elements of input buffer

op operation

comm communicator

This array
must be
identical on
all processes

Reduce & scatter

- C & Fortran bindings

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

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

```
type sendbuf(*), recvbuf(*)
```

```
integer recvcounts(*), datatype, op, comm, ierror
```

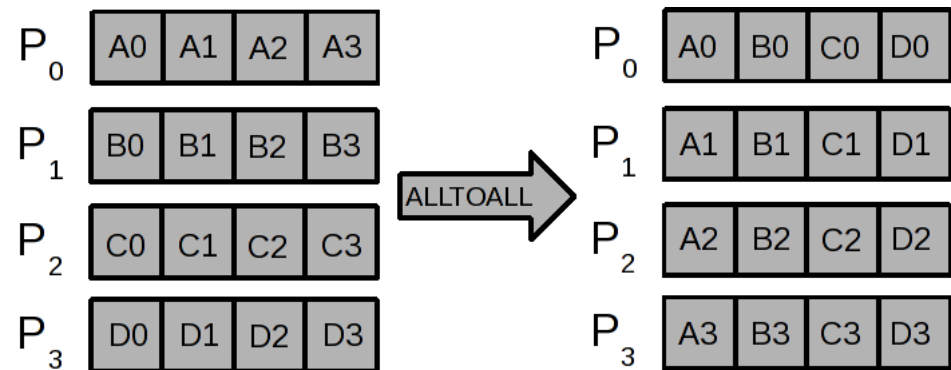


From each to every

- **MPI_Alltoall** sends a distinct message from each task to every task
 - Compare: “All scatter”

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

sendbuf send buffer
sendcount number of elements to send to each process
sendtype data type of send buffer elements
recvbuf receive buffer
recvcount number of elements received from any process
recvtype data type of receive buffer elements
comm communicator



From each to every

- C & Fortran bindings

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

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

```
type sendbuf(*), recvbuf(*)
```

```
integer sendcount, recvcount, sendtype, recvtype, comm,  
ierror
```



All-to-all example

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if

call mpi_bcast(a, 16, MPI_INTEGER, 0, &
  MPI_COMM_WORLD, rc)

call mpi_alltoall(a, 16, MPI_INTEGER, aloc, 4, &
  MPI_INTEGER, MPI_COMM_WORLD, rc)
```

What does happen here? Assume 4 MPI tasks.

Variable-width all-to-all

- **MPI_Alltoallv** is similar to MPI_Alltoall, but messages can have different sizes and displacements

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

<i>sendbuf</i>	send buffer
<i>sendcounts</i>	array (dim: comm size) specifying the number of elements to send to each processor
<i>sdispls</i>	array (of length group size). Entry j specifies the displacement (relative to sendbuf)
<i>sendtype</i>	data type of send buffer elements
<i>recvbuf</i>	receive buffer
<i>recvcounts</i>	array (dim: comm size) specifying the maximum number of elements that can be received from each processor
<i>rdispls</i>	array (dim: comm size). Entry i specifies the displacement (relative to recvbuf)
<i>recvtype</i>	data type of receive buffer elements
<i>comm</i>	communicator

Variable-width all-to-all

- C & Fortran bindings

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

```
MPI_ALLTOALLV(sendbuf,sendcounts, sdispls, sendtype,
recvbuf, recvcounts, rdispls, recvtype, comm, ierror)
type sendbuf(*), recvbuf(*)
integer sendcounts(*), recvcounts(*), sdispls(*),
rdispls(*),
sendtype, recvtype, comm, ierror
```



Common mistakes with collectives



- ⊘ Using a collective operation within one branch of an if-test of the rank

```
IF (my_id == 0) CALL MPI_BCAST(...
```

- All processes, both the root (the sender or the gatherer) and the rest (receivers or senders), must call the collective routine

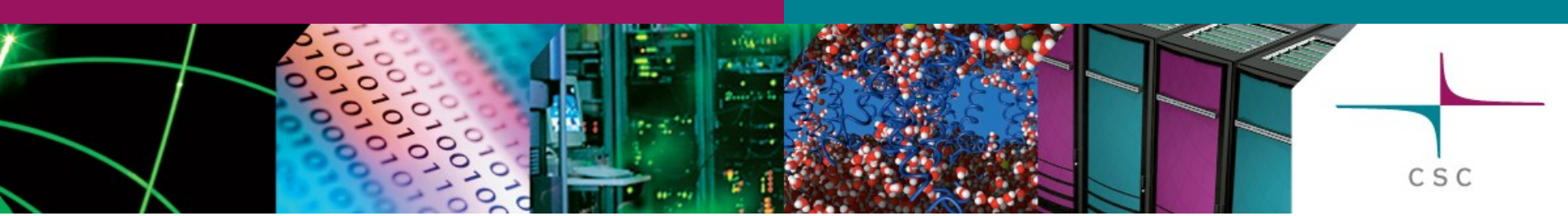
- ⊘ Assuming that all processes making a collective call would complete at the same time

- ⊘ Using the input buffer as the output buffer

```
CALL MPI_ALLREDUCE(a, a, n, MPI_REAL, MPI_SUM, ...
```

Collectives wrap-up

- Collective operations make the code more transparent and compact
- Use when appropriate, i.e. all processes are involved
 - All processes in a communicator must call them, too
- They become even more useful with user-defined communicators



User-defined communicators

Slides Pekka Manninen

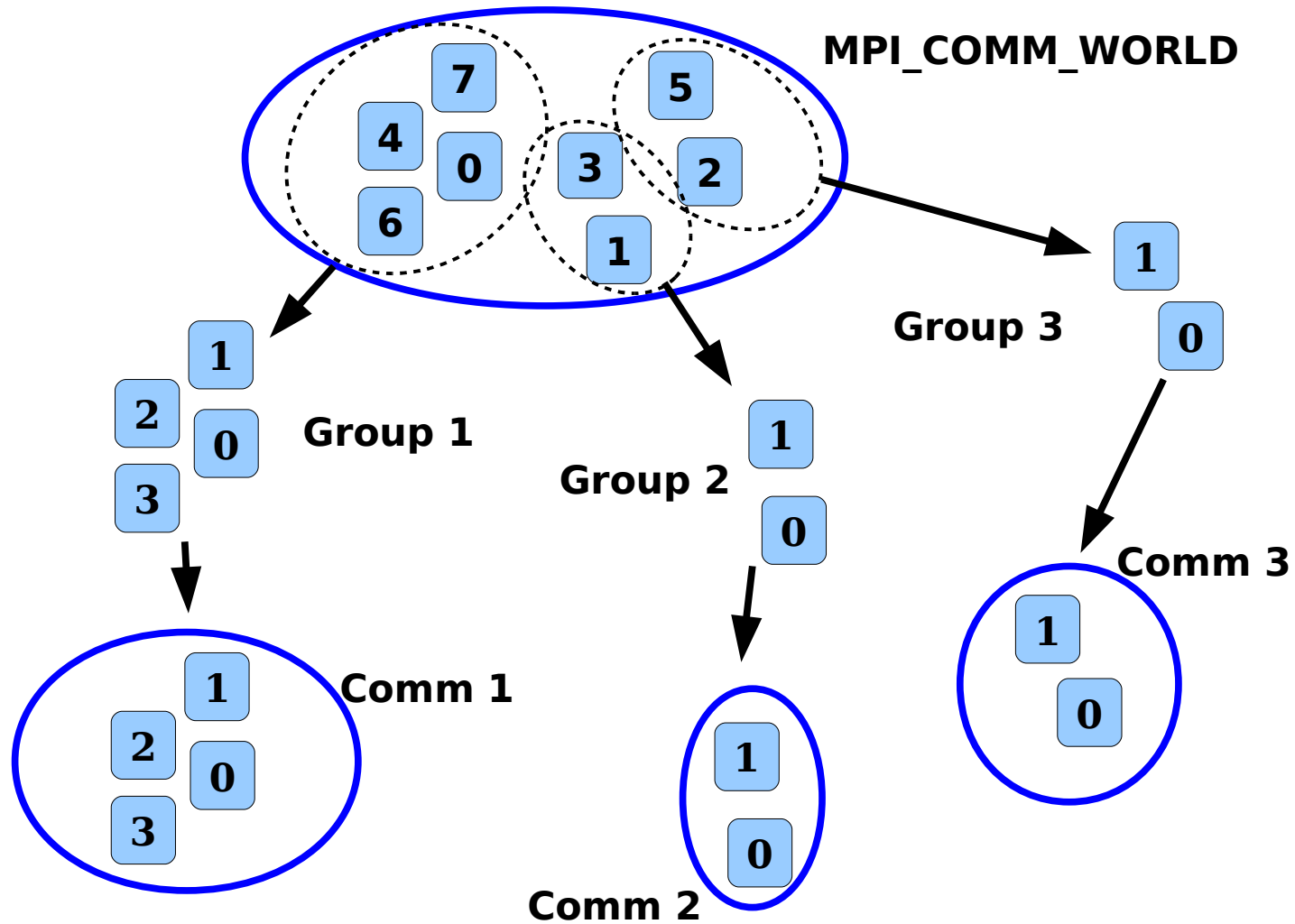


Communicators and process groups



- So far: `MPI_COMM_WORLD`
- Sometimes useful to divide processes into subgroups
 - Task level parallelism with process groups performing separate tasks
 - Improving the data mapping onto tasks
 - Parallel I/O
- The group members communicate primarily within themselves

Communicators and process groups



Communicators



- The communicator determines the scope and the "communication universe"
 - The source and destination of a message is identified by process rank within that group
- Communicators are dynamic
- A task can belong simultaneously to several communicators
 - In each of them it has a unique ID, however
- MPI features a large set of operations for communicator management

Creating a communicator



```
...  
  
if (myid%2 == 0) {  
    color = 1;  
} else {  
    color = 2;  
}  
  
MPI_Comm_split(MPI_COMM_WORLD, color, myid, &subcomm);  
MPI_Comm_rank(subcomm, &mysubid);  
  
printf ("I am rank %d in MPI_COMM_WORLD, but %d in Comm %d.\n",  
        myid, mysubid, color);  
  
...
```

```
> mpirun -np 8 ./comm  
I am rank 2 in MPI_COMM_WORLD, but 1 in Comm 1.  
I am rank 7 in MPI_COMM_WORLD, but 3 in Comm 2.  
I am rank 0 in MPI_COMM_WORLD, but 0 in Comm 1.  
I am rank 4 in MPI_COMM_WORLD, but 2 in Comm 1.  
I am rank 6 in MPI_COMM_WORLD, but 3 in Comm 1.  
I am rank 3 in MPI_COMM_WORLD, but 1 in Comm 2.  
I am rank 5 in MPI_COMM_WORLD, but 2 in Comm 2.  
I am rank 1 in MPI_COMM_WORLD, but 0 in Comm 2.
```

Splitting the MPI tasks into two communicators basing whether their ID in MPI_COMM_WORLD is odd or even

Creating a communicator

- `MPI_Comm_split` creates new communicators based on 'colors' and 'keys'

```
MPI_Comm_split(comm, color, key, newcomm)
```

<code>comm</code>	communicator handle
<code>color</code>	control of subset assignment, processes with the same color belong to the same new communicator
<code>key</code>	control of rank assignment
<code>newcomm</code>	new communicator handle

If `color = MPI_UNDEFINED`, a process does not belong to any of the new communicators

Creating a communicator



- C and Fortran bindings

```
int MPI_Comm_split (MPI_Comm comm, int color, int key,  
    MPI_Comm newcomm)
```

```
MPI_COMM_SPLIT (comm, color, key, newcomm, rc)
```

```
INTEGER comm, color, key, newcomm, rc
```

- Return code values

`MPI_SUCCESS` No error; MPI routine completed successfully.

`MPI_ERR_COMM` Invalid communicator. A common error is to use a null communicator in a call

`MPI_ERR_INTERN` This error is returned when some part of the implementation is unable to acquire memory



Another example



```
if (my_id == 0) then
  write (*,*) 'n?'
  read (*,*) n
  allocate(x(n))
  call random_number(x)
end if
call mpi_bcast(n, 1, &
  MPI_INTEGER, 0, &
  MPI_COMM_WORLD, rc)
nloc = n/ntasks
allocate (xloc(nloc))
call mpi_scatter(x, nloc, &
  MPI_REAL, xloc, nloc, &
  MPI_REAL, 0, MPI_COMM_WORLD,
  rc)
```

```
if (modulo(my_id,2)==0) then
  color = 1
  operation = MPI_MAX
  loc = maxval(xloc(:))
else
  color = 2
  operation = MPI_MIN
  loc = minval(xloc(:))
end if

call mpi_comm_split(MPI_COMM_WORLD,&
  color, my_id, subcomm, rc)
call mpi_comm_rank(subcomm,&
  my_subid, rc)

call mpi_reduce(loc, sval, 1,&
  MPI_REAL, operation, 0,&
  subcomm, rc)
```

Other communicator defines a global minimum of a set of real numbers and another a maximum.

Communicator manipulation



MPI_Comm_size	Returns number of processes in communicator's group
MPI_Comm_rank	Returns rank of calling process in communicator's group
MPI_Comm_compare	Compares two communicators
MPI_Comm_dup	Duplicates a communicator
MPI_Comm_free	Marks a communicator for deallocation

Communicator manipulation

- Communicator comparison

```
MPI_Comm_compare(comm1, comm2, result)
```

```
comm1    communicator 1 handle
```

```
comm2    communicator 2 handle
```

```
result   integer with a value MPI_IDENT if the contexts and  
groups are the same, MPI_CONGRUENT if different  
contexts but identical groups, MPI_SIMILAR if  
different contexts but similar groups, and  
MPI_UNEQUAL otherwise
```

- Communicator duplication

```
MPI_Comm_dup(comm, newcomm)
```

```
comm     old communicator
```

```
newcomm  duplicated communicator
```

- Communicator deallocation

```
MPI_Comm_free(comm)
```

```
comm     communicator to be destroyed
```

Communicator manipulation



- C and Fortran bindings

```
int MPI_Comm_compare ( MPI_Comm comm1, MPI_Comm comm2, int
    result )
```

```
MPI_COMM_COMPARE ( comm1, comm2, result, rc )
INTEGER comm1, comm2, result, rc
```

```
int MPI_Comm_dup ( MPI_Comm comm, MPI_Comm newcomm )
```

```
MPI_COMM_DUP ( comm, newcomm, rc )
INTEGER comm, newcomm, rc
```

```
int MPI_Comm_free ( MPI_Comm comm )
```

```
MPI_COMM_FREE ( comm, rc )
INTEGER comm, rc
```



Summary

- The processes can be divided into subgroups with user defined communicators
- Useful
 - In task level parallelism
 - When communication happens between subgroups of processes e.g. with multiple levels of parallelization