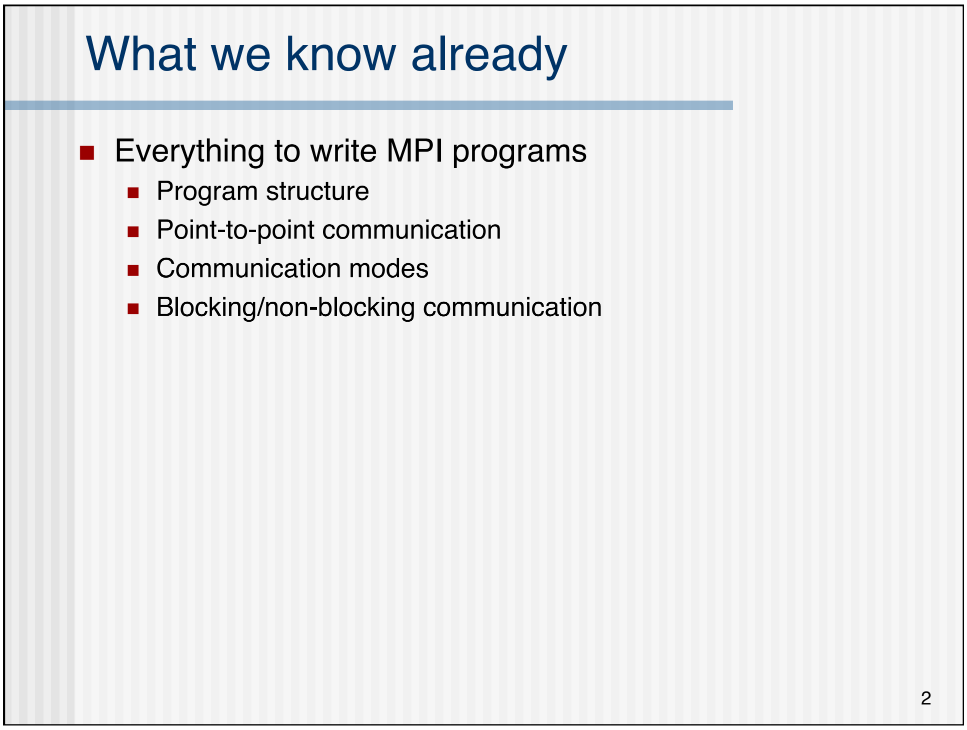
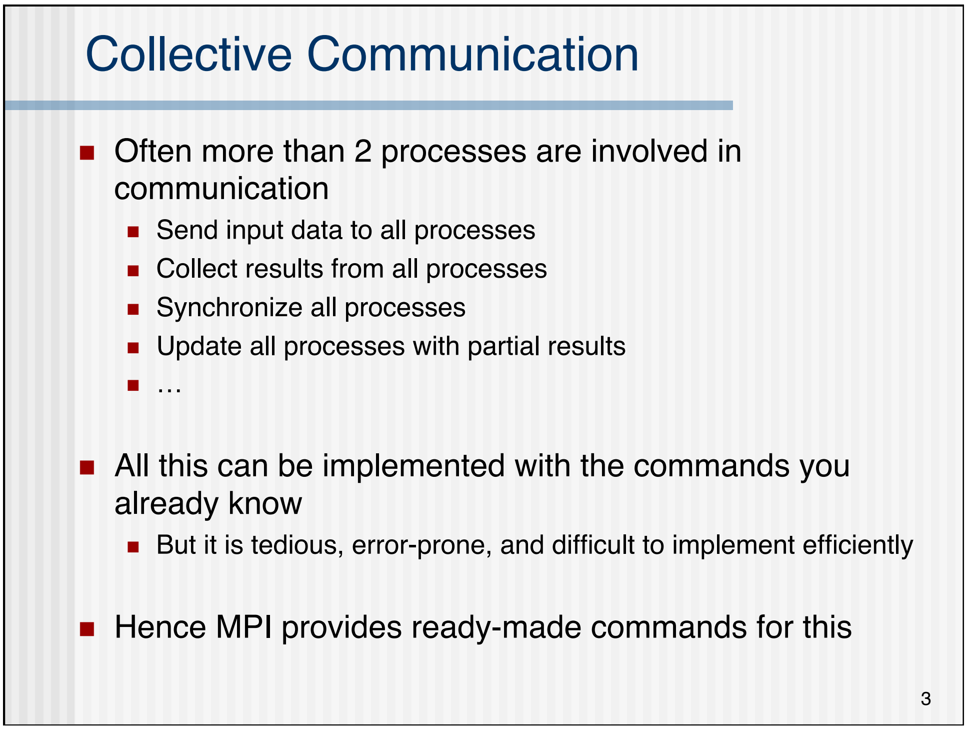
Basic MPI
Collective Communication

Erwin Laure
Director PDC



What we know already

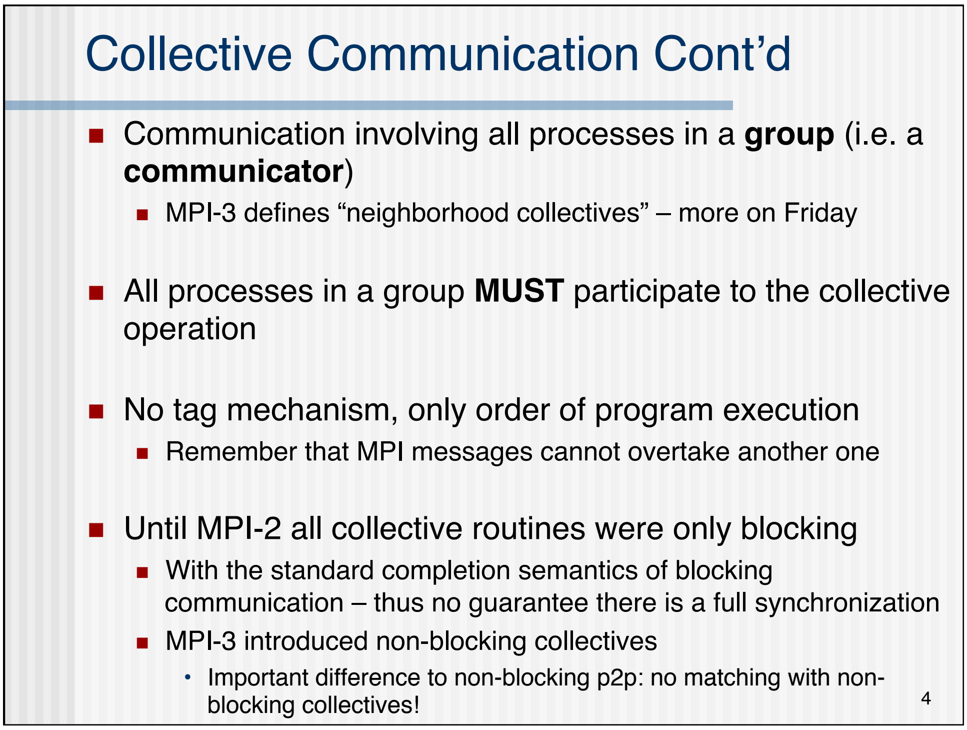
- Everything to write MPI programs
  - Program structure
  - Point-to-point communication
  - Communication modes
  - Blocking/non-blocking communication



Collective Communication

- Often more than 2 processes are involved in communication
  - Send input data to all processes
  - Collect results from all processes
  - Synchronize all processes
  - Update all processes with partial results
  - …
- All this can be implemented with the commands you already know
  - But it is tedious, error-prone, and difficult to implement efficiently
- Hence MPI provides ready-made commands for this

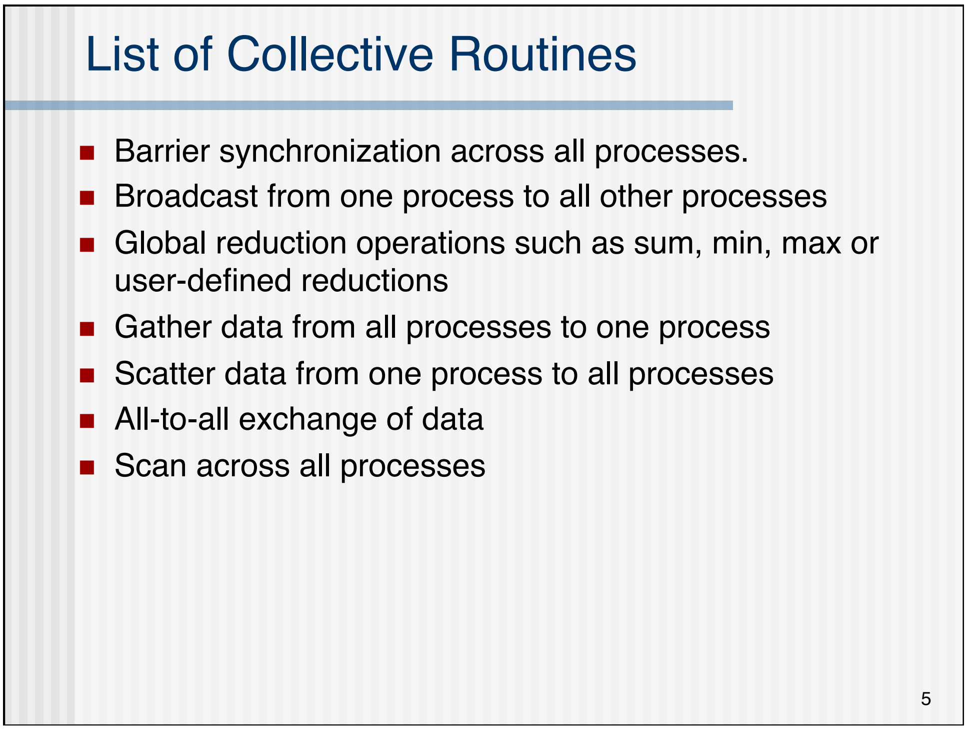
3



Collective Communication Cont’d

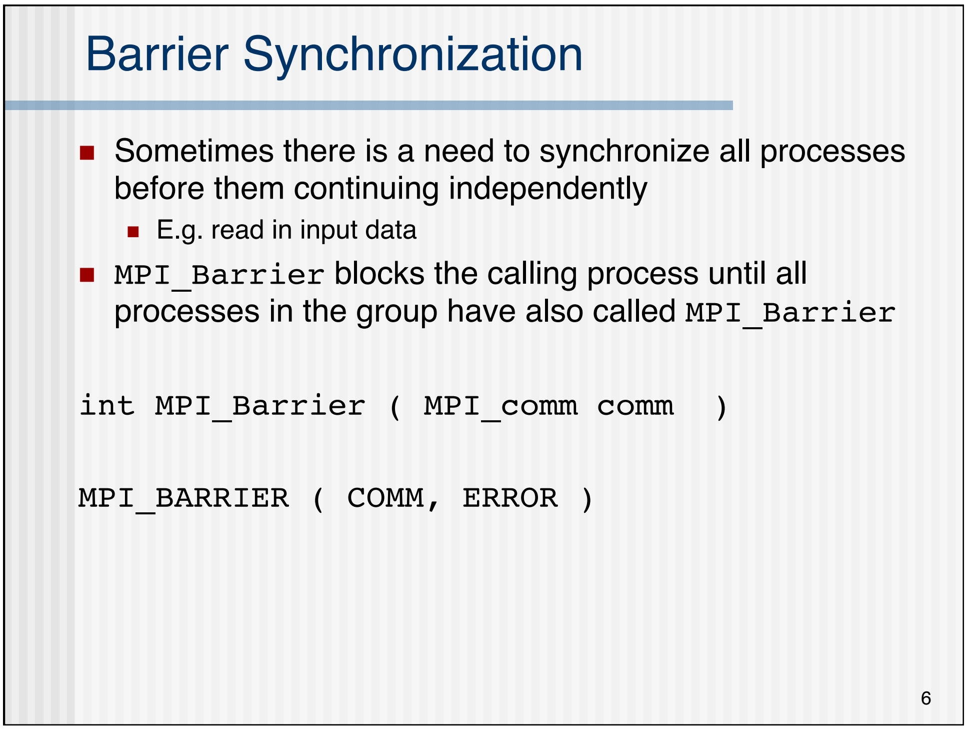
- Communication involving all processes in a **group** (i.e. a **communicator**)
  - MPI-3 defines “neighborhood collectives” – more on Friday
- All processes in a group **MUST** participate to the collective operation
- No tag mechanism, only order of program execution
  - Remember that MPI messages cannot overtake another one
- Until MPI-2 all collective routines were only blocking
  - With the standard completion semantics of blocking communication – thus no guarantee there is a full synchronization
  - MPI-3 introduced non-blocking collectives
    - Important difference to non-blocking p2p: no matching with non-blocking collectives!

4



List of Collective Routines

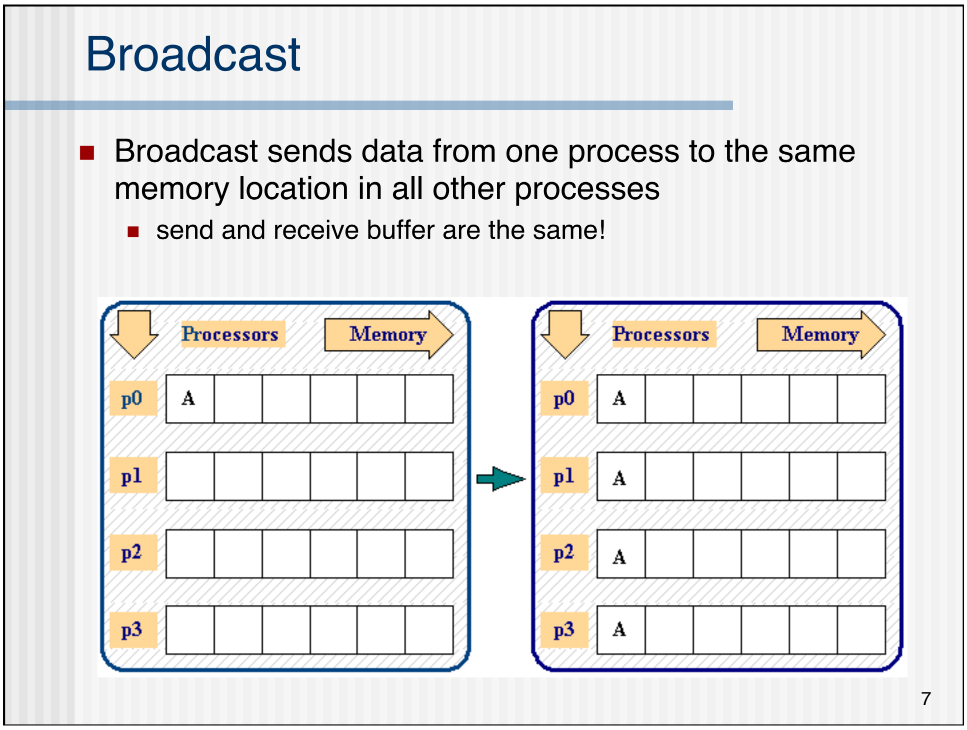
- Barrier synchronization across all processes.
- Broadcast from one process to all other processes
- Global reduction operations such as sum, min, max or user-defined reductions
- Gather data from all processes to one process
- Scatter data from one process to all processes
- All-to-all exchange of data
- Scan across all processes



Barrier Synchronization

- Sometimes there is a need to synchronize all processes before them continuing independently
  - E.g. read in input data
- `MPI_Barrier` blocks the calling process until all processes in the group have also called `MPI_Barrier`

```
int MPI_Barrier ( MPI_comm comm  )

MPI_BARRIER ( COMM, ERROR )
```



Broadcast

- Broadcast sends data from one process to the same memory location in all other processes
  - send and receive buffer are the same!

Processors → Memory

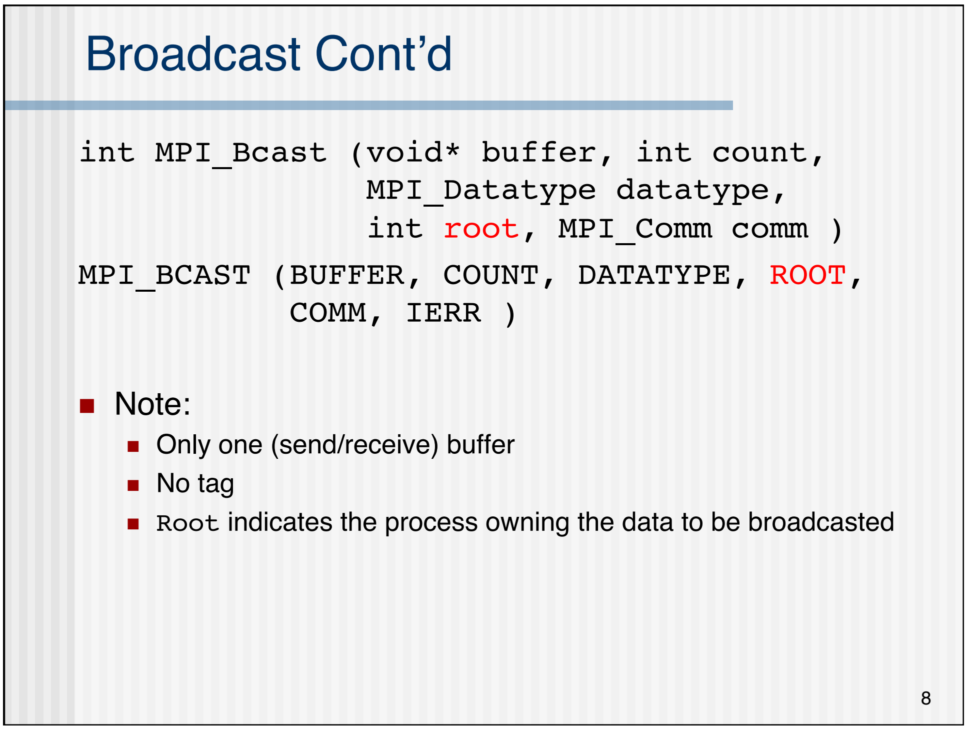
p0	A					
p1						
p2						
p3						

→

Processors → Memory

p0	A					
p1	A					
p2	A					
p3	A					

7



Broadcast Cont’d

```
int MPI_Bcast (void* buffer, int count,
               MPI_Datatype datatype,
               int root, MPI_Comm comm )
MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT,
           COMM, IERR )
```

- Note:
  - Only one (send/receive) buffer
  - No tag
  - `Root` indicates the process owning the data to be broadcasted

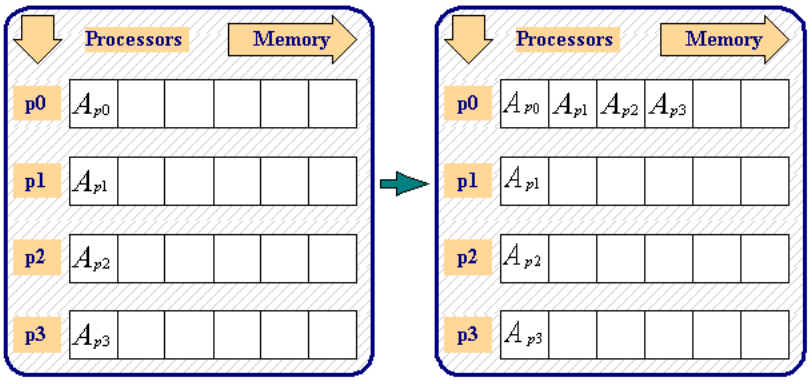
8

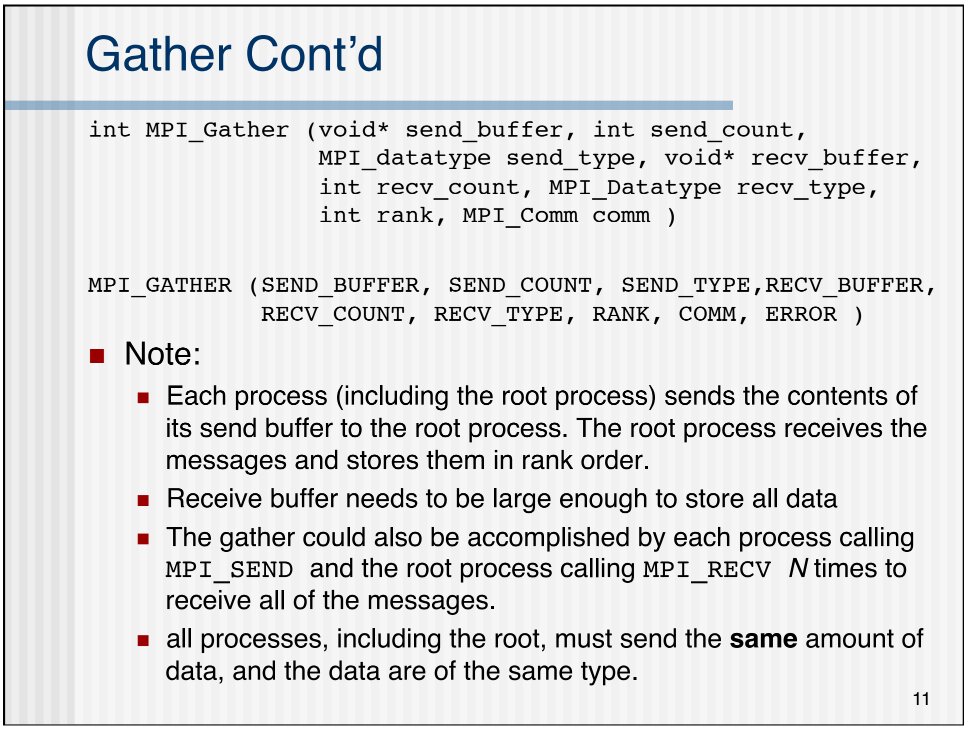
Broadcast Example

```
#include <mpi.h>
void 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();
}
```

Gather

- Gather is a all-to-one operation that collects the data from all processes in target process

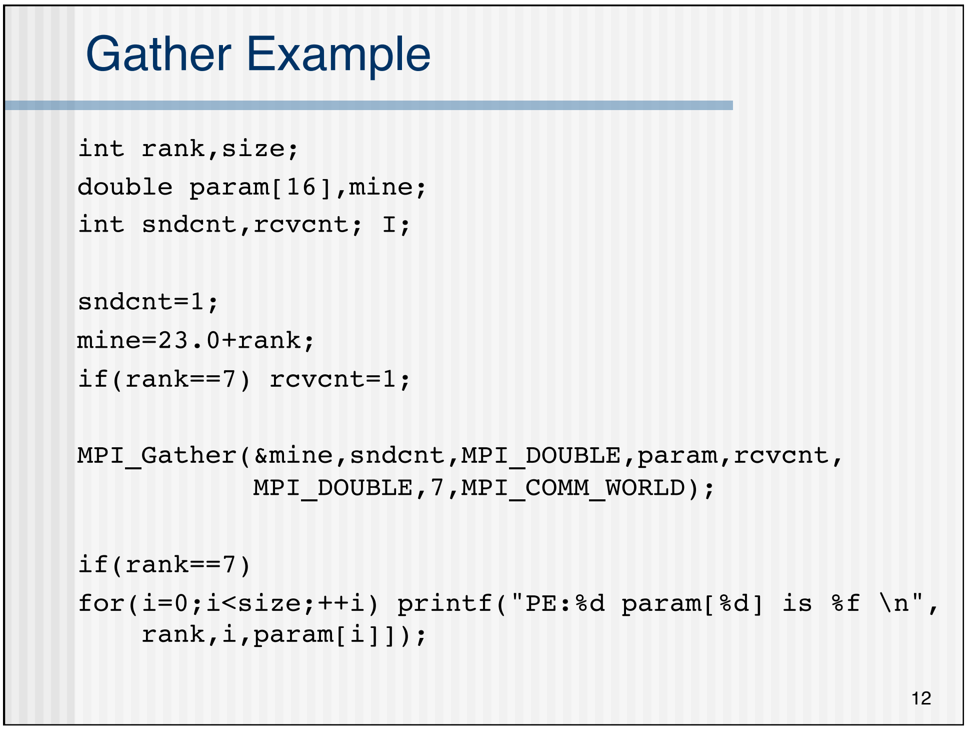




Gather Cont'd

```
int MPI_Gather (void* send_buffer, int send_count,
                MPI_datatype send_type, void* recv_buffer,
                int recv_count, MPI_Datatype recv_type,
                int rank, MPI_Comm comm )


MPI_GATHER (SEND_BUFFER, SEND_COUNT, SEND_TYPE,RECV_BUFFER,
            RECV_COUNT, RECV_TYPE, RANK, COMM, ERROR )
```

- Note:
  - Each process (including the root process) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order.
  - Receive buffer needs to be large enough to store all data
  - The gather could also be accomplished by each process calling `MPI_SEND` and the root process calling `MPI_RECV` *N* times to receive all of the messages.
  - all processes, including the root, must send the **same** amount of data, and the data are of the same type.

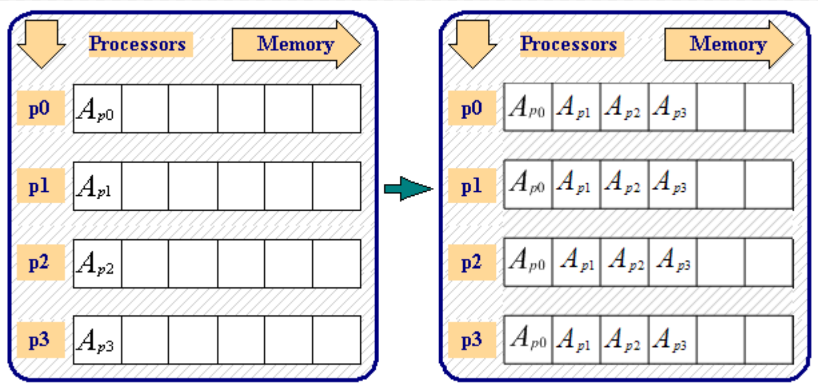


Gather Example

```
int rank,size;
double param[16],mine;
int sndcnt,rcvcnt; I;

sndcnt=1;
mine=23.0+rank;
if(rank==7) rcvcnt=1;

MPI_Gather(&mine,sndcnt,MPI_DOUBLE,param,rcvcnt,
           MPI_DOUBLE,7,MPI_COMM_WORLD);

if(rank==7)
for(i=0;i<size;++i) printf("PE:%d param[%d] is %f \n",
    rank,i,param[i]]);
```

Allgather

- Sometimes it is also useful to gather the data not only into one process but all
- Equivalent to `MPI_Gather` plus `MPI_Bcast`
- `MPI_Allgather` has same syntax as `MPI_Gather`

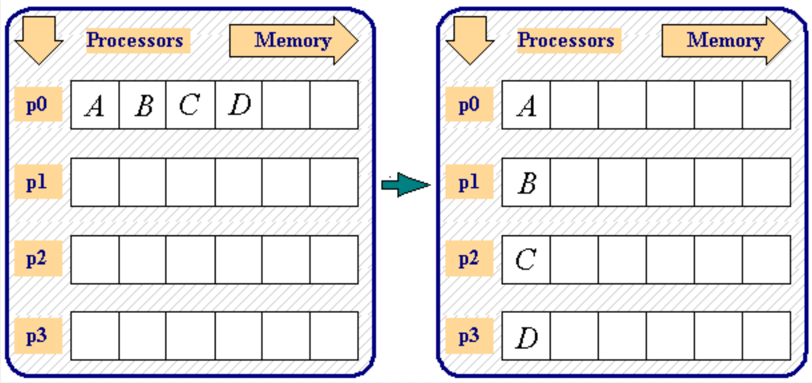


p0	A_{p0}					
p1	A_{p1}					
p2	A_{p2}					
p3	A_{p3}					

p0	A_{p0}	A_{p1}	A_{p2}	A_{p3}		
p1	A_{p0}	A_{p1}	A_{p2}	A_{p3}		
p2	A_{p0}	A_{p1}	A_{p2}	A_{p3}		
p3	A_{p0}	A_{p1}	A_{p2}	A_{p3}		

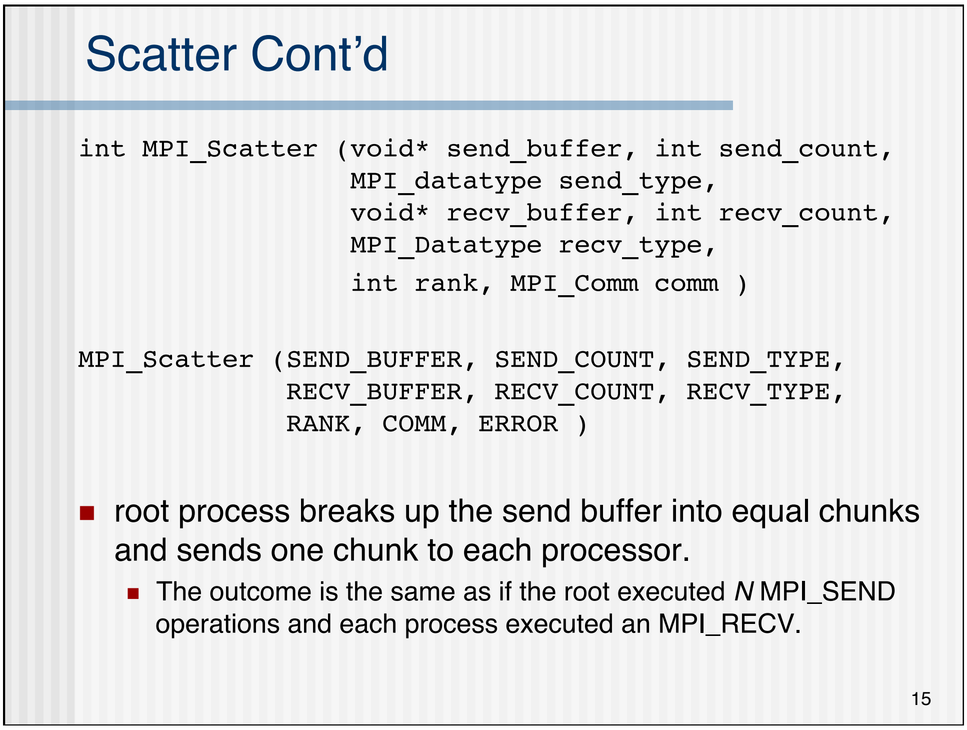
Scatter

- Distribute data to all processes – one-to-all communication
- Inverse to gather



p0	A	B	C	D		
p1						
p2						
p3						

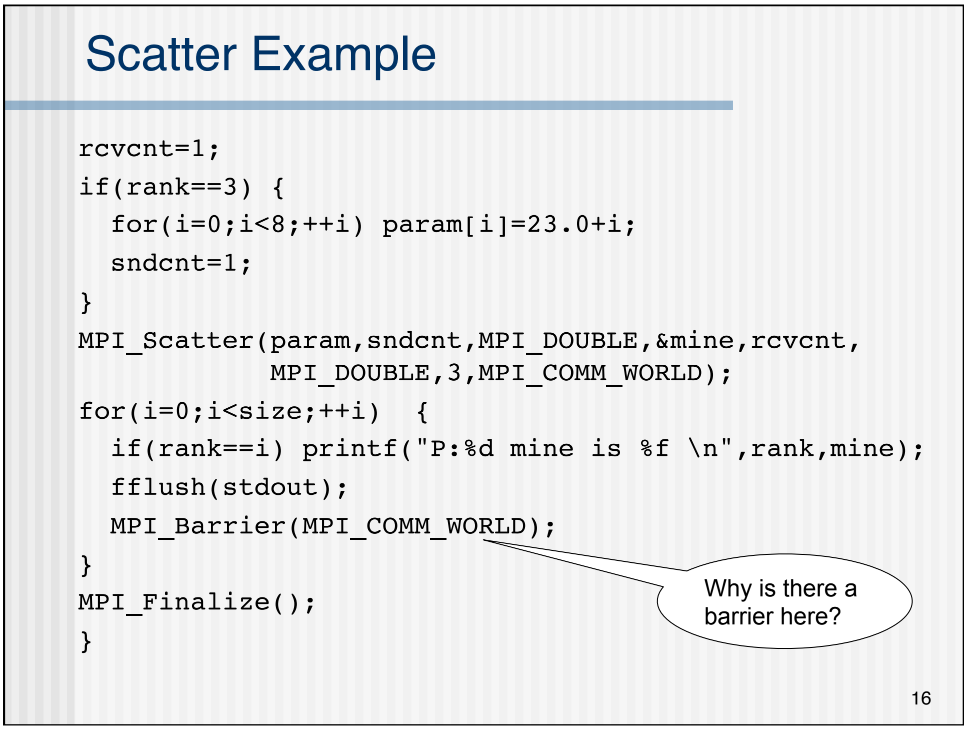
p0	A					
p1	B					
p2	C					
p3	D					



Scatter Cont’d

```
int MPI_Scatter (void* send_buffer, int send_count,
                 MPI_datatype send_type,
                 void* recv_buffer, int recv_count,
                 MPI_Datatype recv_type,
                 int rank, MPI_Comm comm )

MPI_Scatter (SEND_BUFFER, SEND_COUNT, SEND_TYPE,
             RECV_BUFFER, RECV_COUNT, RECV_TYPE,
             RANK, COMM, ERROR )
```

- root process breaks up the send buffer into equal chunks and sends one chunk to each processor.
  - The outcome is the same as if the root executed *N* MPI_SEND operations and each process executed an MPI_RECV.



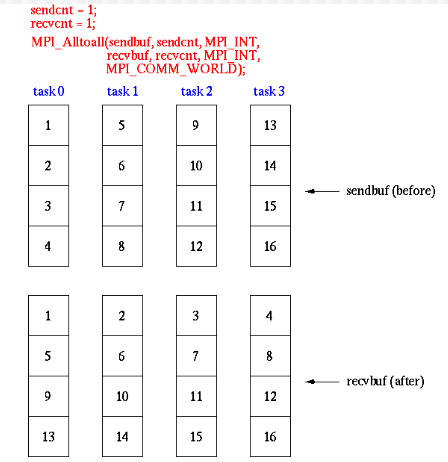
Scatter Example

```
rcvcnt=1;
if(rank==3) {
  for(i=0;i<8;++i) param[i]=23.0+i;
  sndcnt=1;
}
MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,rcvcnt,
            MPI_DOUBLE,3,MPI_COMM_WORLD);
for(i=0;i<size;++i)  {
  if(rank==i) printf("P:%d mine is %f \n",rank,mine);
  fflush(stdout);
  MPI_Barrier(MPI_COMM_WORLD);
}
MPI_Finalize();
}
```

Why is there a barrier here?

Other Gather/Scatter Variants

- Gather/Scatter is also defined over vectors
  - `MPI_GATHERV` and `MPI_SCATTERV` allow a varying count of data from/to each process.
- `MPI_ALLTOALL`
  - Every process performs a scatter



```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
             recvbuf, recvcnt, MPI_INT,
             MPI_COMM_WORLD);
```

task 0	task 1	task 2	task 3
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

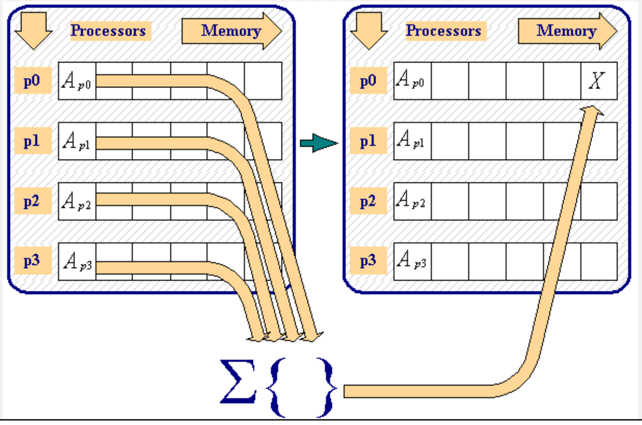
sendbuf (before)

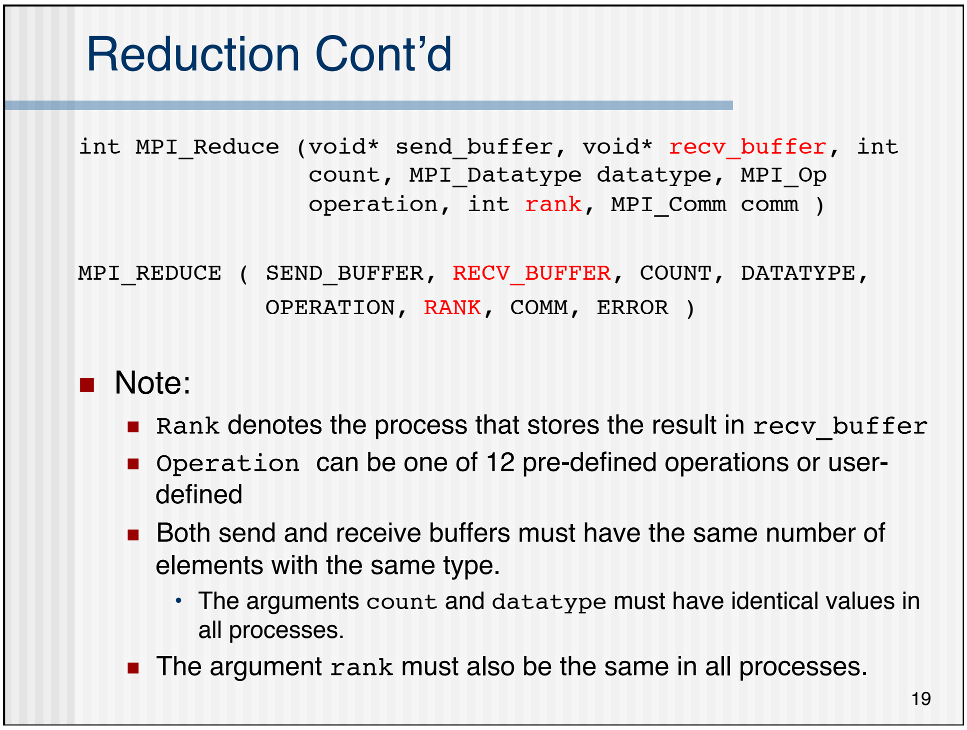
task 0	task 1	task 2	task 3
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

recvbuf (after)

Reduction

- Collect data from each processor
- Reduce these data to a single value (such as a sum or max)
- Store the reduced result on the root processor

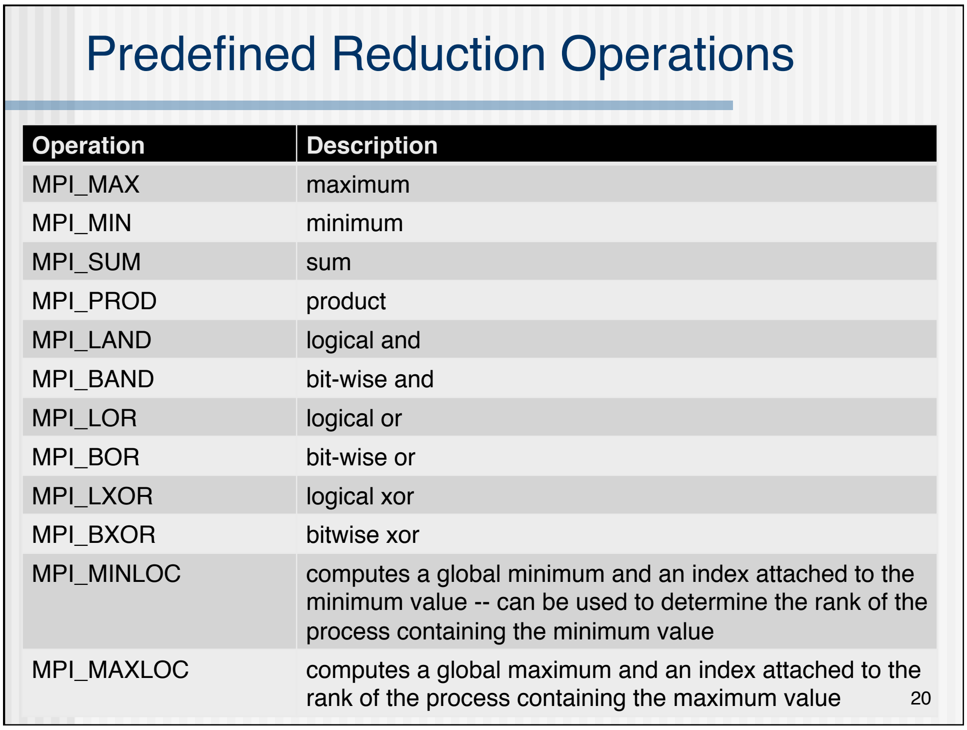




Reduction Cont'd

```
int MPI_Reduce (void* send_buffer, void* recv_buffer, int
                count, MPI_Datatype datatype, MPI_Op
                operation, int rank, MPI_Comm comm )

MPI_REDUCE ( SEND_BUFFER, RECV_BUFFER, COUNT, DATATYPE,
             OPERATION, RANK, COMM, ERROR )
```

- Note:
  - `Rank` denotes the process that stores the result in `recv_buffer`
  - `Operation` can be one of 12 pre-defined operations or user-defined
  - Both send and receive buffers must have the same number of elements with the same type.
    - The arguments `count` and `datatype` must have identical values in all processes.
  - The argument `rank` must also be the same in all processes.



Predefined Reduction Operations

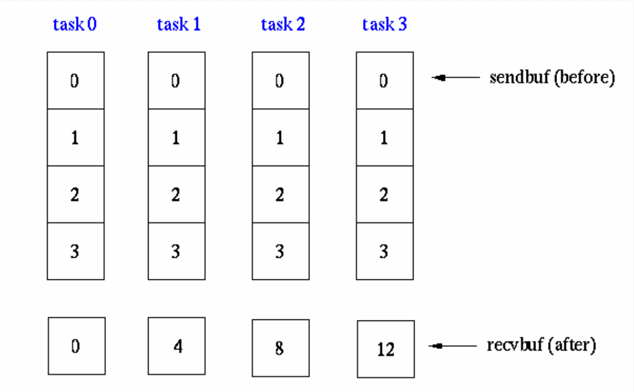
Operation	Description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bitwise xor
MPI_MINLOC	computes a global minimum and an index attached to the minimum value -- can be used to determine the rank of the process containing the minimum value
MPI_MAXLOC	computes a global maximum and an index attached to the rank of the process containing the maximum value

Reduction Example

```
#include    <stdio.h>
#include    <mpi.h>
void main(int argc, char *argv[]) {
  int rank;
  int source,result,root;

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

  root=7;
  source=rank+1;

  MPI_Reduce(&source,&result,1, MPI_INT, MPI_PROD, root,
             MPI_COMM_WORLD);
  if(rank==root) printf("P:%d MPI_PROD result is %d \n", rank,
                         result);

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
}
```

Reduce Variations

- `MPI_Allreduce` makes the result available in the receive buffers of all processes
  - Equivalent to `MPI_Reduce` plus `MPI_Bcast`
- `MPI_Reduce_scatter` scatters the result vector across the processes in the group

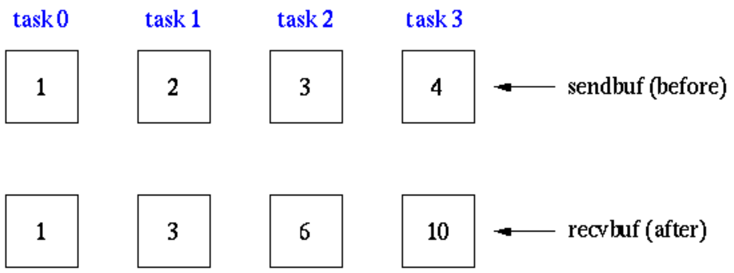


task 0	task 1	task 2	task 3	
0	0	0	0	← sendbuf (before)
1	1	1	1	
2	2	2	2	
3	3	3	3	
0	4	8	12	← recvbuf (after)

Reduce Variations Cont'd

- `MPI_Scan` performs a partial reduction in which process i receives data from processes 0 through i, inclusive

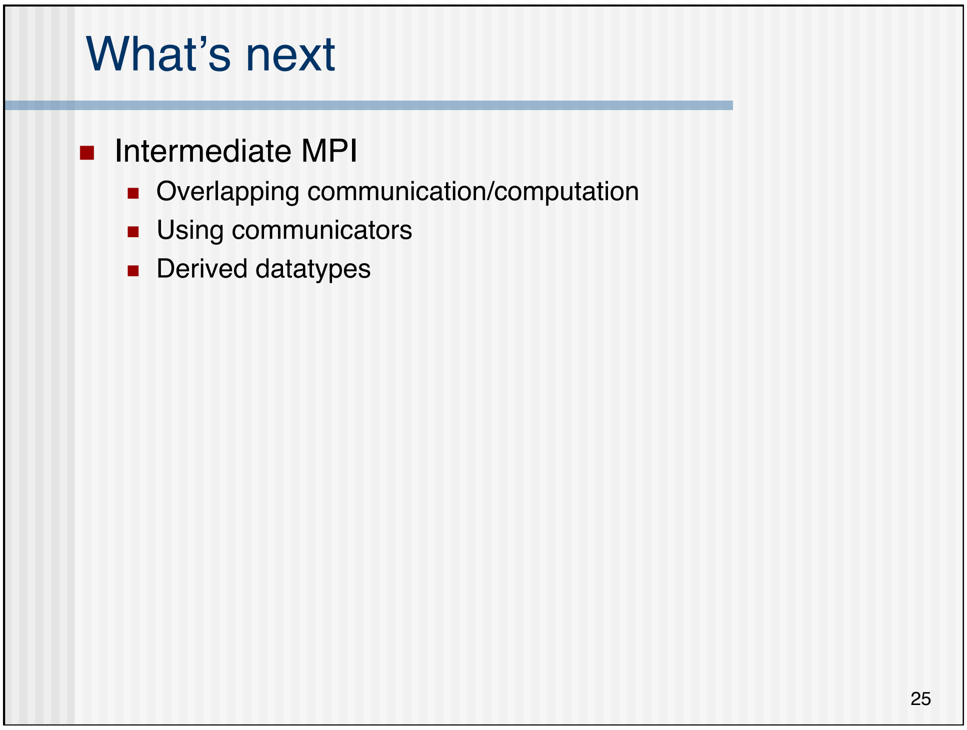
```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
                MPI_COMM_WORLD);
```



task 0	task 1	task 2	task 3	
1	2	3	4	← sendbuf (before)
1	3	6	10	← recvbuf (after)

Summary

- Collective communication routines provide convenient calls for standard communication patterns
- Depending on the implementation they may be much more efficient than hand-coding (or not)
  - Synchronization overhead might be substantial
- Collective communication makes extensive use of groups/ communicators



What’s next

- Intermediate MPI
  - Overlapping communication/computation
  - Using communicators
  - Derived datatypes

25