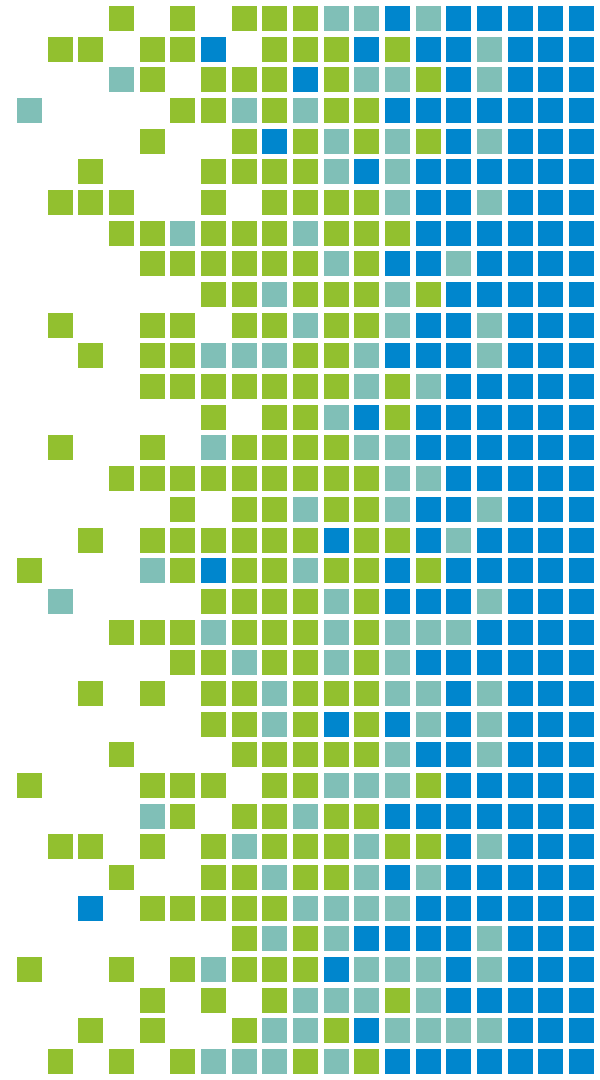


PRACE Course: Intermediate MPI

9-11 November 2022

MPI Collective Communication



Collective Communication

- Communications involving a group of processes.
- Must be called by all processes in a communicator.
- Examples:
 - Barrier synchronization.
 - Broadcast, scatter, gather.
 - Global sum, global maximum, etc.
 - Neighbor communication in a virtual process grid

From MPI-3.0

Characteristics of Collective Communication

- Optimised Communication routines involving a group of processes
- Collective action over a communicator, i.e. all processes must call the collective routine.
- Synchronization may or may not occur.
- No tags.
- Receive buffers must have exactly the same size as send buffers.
- In MPI-1.0 – MPI-2.2, all collective operations are blocking. Nonblocking versions since MPI-3.0.

Barrier Synchronisation

- C:
`int MPI_Barrier(MPI_Comm comm)`
- Fortran:
`MPI_BARRIER(COMM, IERROR)`
`TYPE(MPI_Comm) :: comm`
`INTEGER, OPTIONAL :: ierror`
- MPI_Barrier is normally never needed:
 - all synchronization is done automatically by the data communication; a process cannot continue before it has the data that it needs.
 - if used for debugging, please guarantee, that it is removed in production.

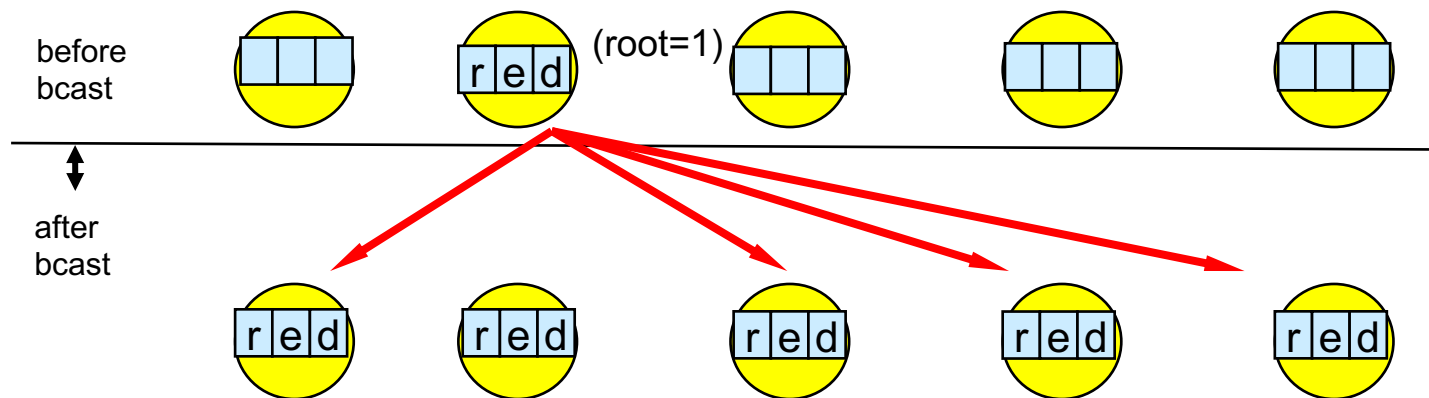
Broadcast

- C:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```
- Fortran:

```
MPI_Bcast(BUF, COUNT, DATATYPE, ROOT, COMM, IERROR)
  TYPE(*), DIMENSION(..) :: buf; TYPE(MPI_Datatype) :: datatype
  INTEGER COUNT, ROOT; TYPE(MPI_Comm) :: comm
  INTEGER, OPTIONAL :: ierror
```

MPI_Bcast(buf, 3, MPI_CHAR, 1, MPI_COMM_WORLD);



Scatter

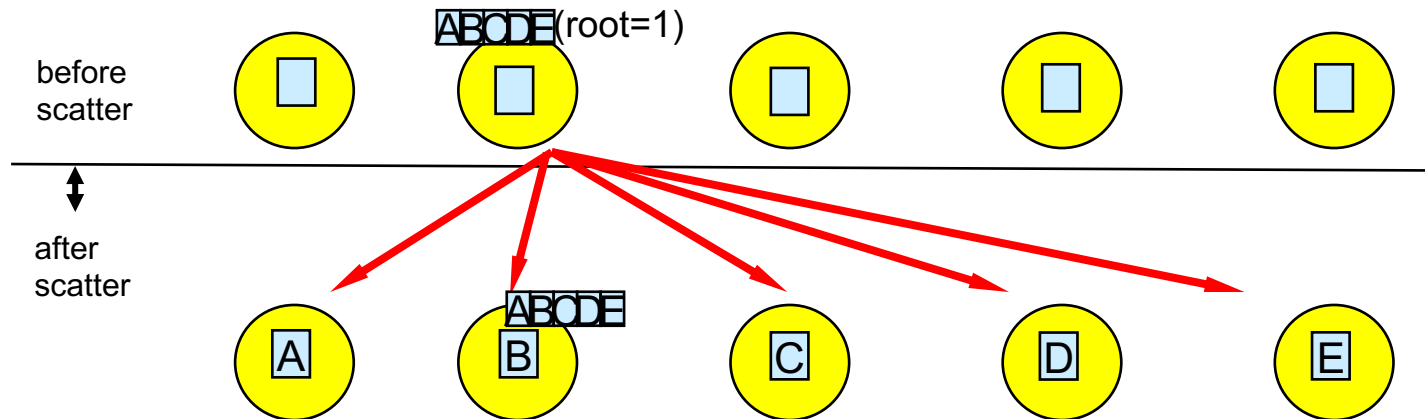
- C:

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

- Fortran:

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
RECVTYPE, ROOT, COMM, IERROR)
TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER,
OPTIONAL :: ierror
```

```
MPI_Scatter(sbuf, 1, MPI_CHAR, rbuf, 1, MPI_CHAR, 1, MPI_COMM_WORLD);
```



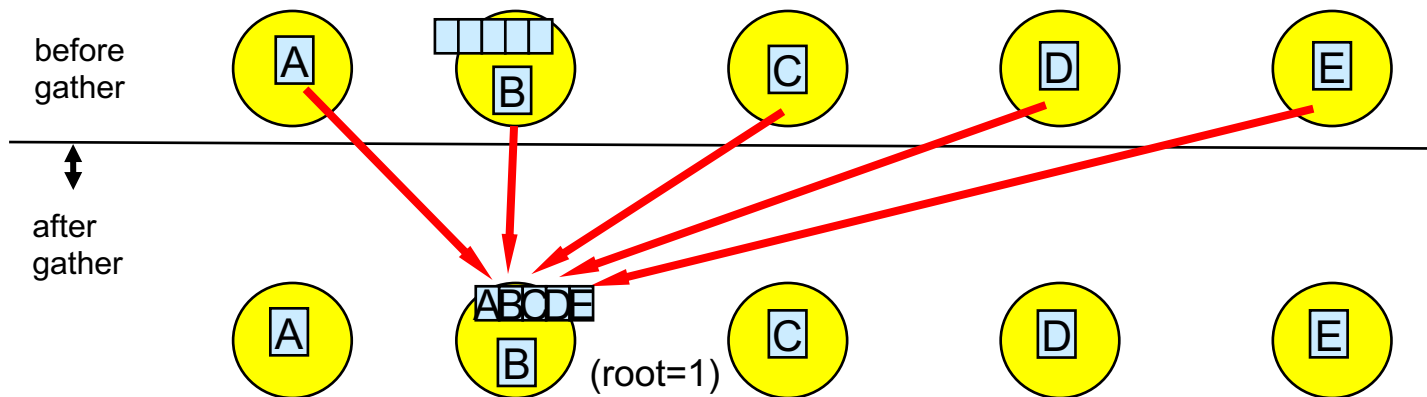
Gather

- C:

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```
- Fortran:

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: sendcount, recvcount, root;
TYPE(MPI_Datatype) :: sendtype, recvtype; TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror
```

MPI_Gather(sbuf, 1, MPI_CHAR, rbuf, 1, MPI_CHAR, 1, MPI_COMM_WORLD);



Example – MPI Scatter

```
#include <stdio.h>
#include <mpi.h>
#define SIZE 4

int main( int argc, char **argv ) {
    int myRank, uniSize, ierror;
    int sendbuf[SIZE][SIZE]={
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, 16},
    };
    int recvbuf[SIZE];

    ierror=MPI_Init(&argc, &argv);
    ierror=MPI_Comm_size(MPI_COMM_WORLD, &uniSize);
    ierror=MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if(uniSize==SIZE){
        ierror = MPI_Scatter(sendbuf, SIZE, MPI_INT, recvbuf, SIZE, MPI_INT, 0, MPI_COMM_WORLD );
        printf("rank= %d  Results: %d %d %d %d\n", myRank,recvbuf[0],recvbuf[1],recvbuf[2],recvbuf[3]);
    }

    ierror = MPI_Finalize();
    return ierror;
}
```

rank= 0	Results: 1.0 2.0 3.0 4.0
rank= 1	Results: 5.0 6.0 7.0 8.0
rank= 2	Results: 9.0 10.0 11.0 12.0
rank= 3	Results: 13.0 14.0 15.0 16.0

Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- $d_0 \circ d_1 \circ d_2 \circ d_3 \circ \dots \circ d_{s-2} \circ d_{s-1}$
 - d_i : data in process rank i (single variable, or vector)
 - \circ : associative operation
 - Example:
 - global sum or product
 - global maximum or minimum
 - global user-defined operation
- floating point rounding may depend on usage of associative law:
 - $[(d_0 \circ d_1) \circ (d_2 \circ d_3)] \circ [\dots \circ (d_{s-2} \circ d_{s-1})]$
 - $(((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \dots) \circ d_{s-2}) \circ d_{s-1})$

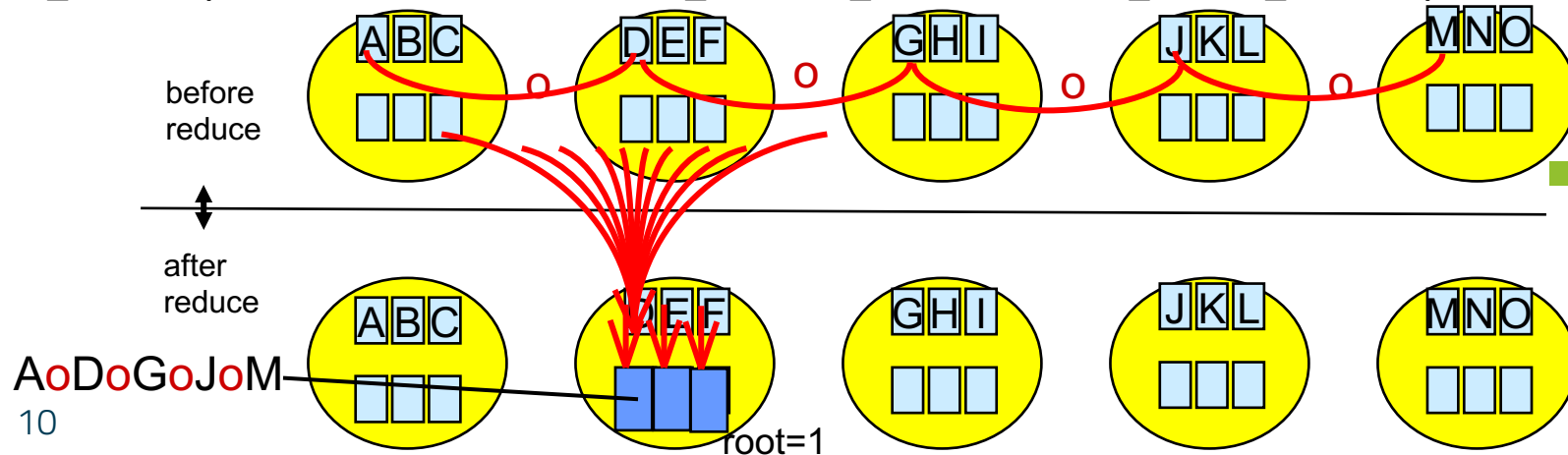
Reduce

- C:

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

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERR, OP)
  TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: count, root
  TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm;
  TYPE(MPI_Op) :: op; INTEGER, OPTIONAL :: ierror
```

MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);



Predefined Reduction Operation Handles

Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

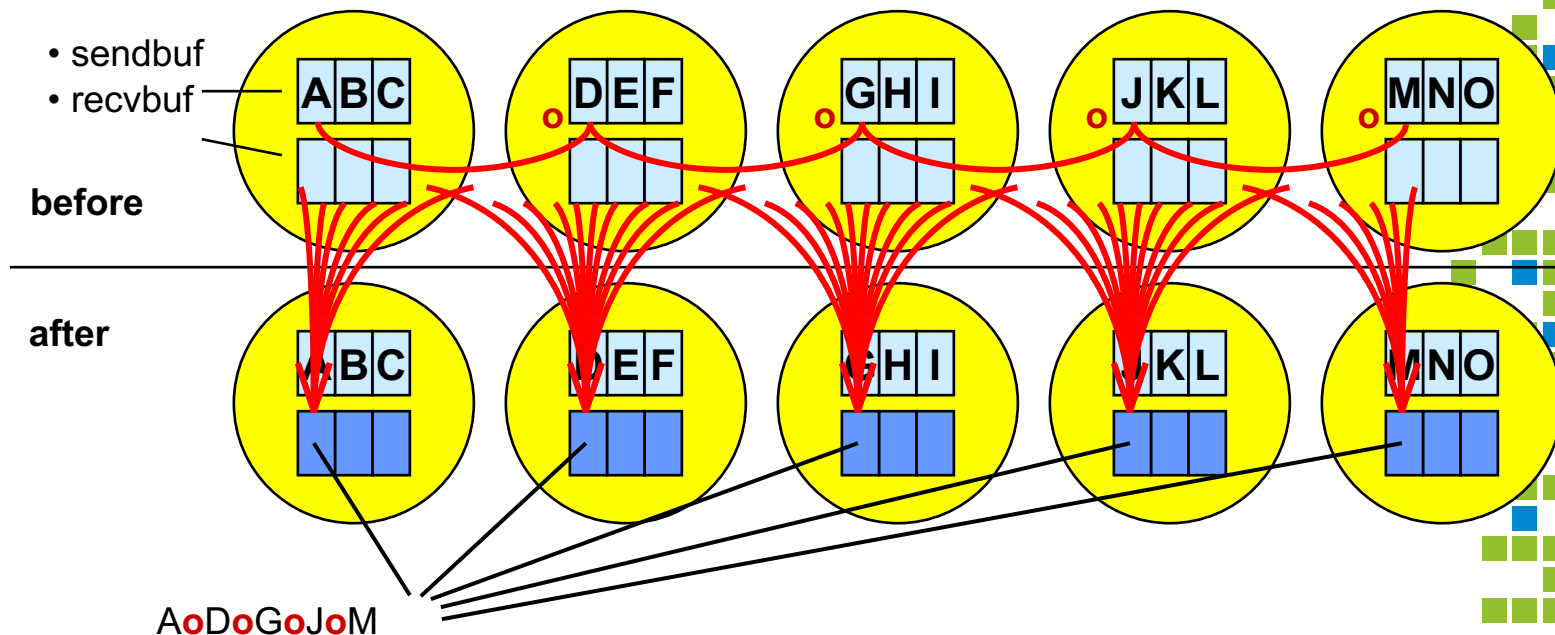
User-Defined Reduction Operations

- Operator handles
 - predefined – see table above
 - user-defined
- User-defined operation \square :
 - associative
 - user-defined function must perform the operation $\text{vector_A} \square \text{vector_B}$
- User-defined function: `MPI_User_function`, `FUNCTION USER_FUNCTION`
- Registering a user-defined reduction function:
 - C: `MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *op)`
 - Fortran: `MPI_OP_CREATE(FUNC, COMMUTE, OP, IERROR)`

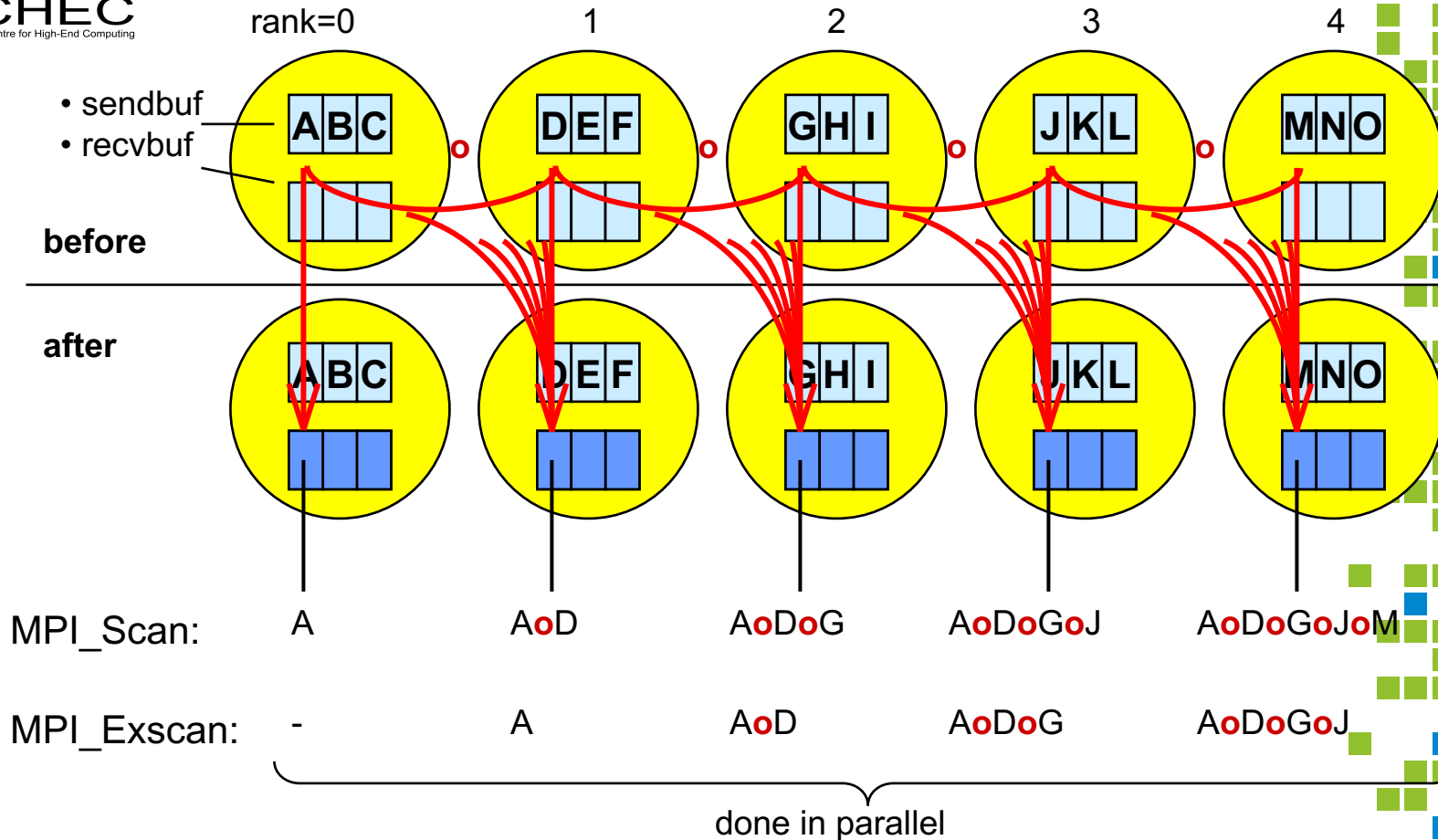
Variants of Reduction Operations

- `MPI_ALLREDUCE`
 - no root,
 - returns the result in all processes
- `MPI_REDUCE_SCATTER_BLOCK` and `MPI_REDUCE_SCATTER`
 - result vector of the reduction operation is scattered to the processes into the real result buffers
- `MPI_SCAN`
 - prefix reduction
 - result at process with rank i :=
reduction of inbuf-values from rank 0 to rank i
- `MPI_Exscan`
 - result at process with rank i :=
reduction of inbuf-values from rank 0 to rank $i-1$

MPI_Allreduce



MPI_Scan and MPI_Exscan



Other Collective Communication Routines

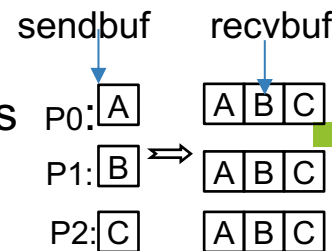
- MPI_Gatherv, MPI_Scatterv
 - Each message has a different count and displacement
 - array of counts and array of displs
- MPI_Allgather, MPI_Allgatherv
 - similar to MPI_Gather, but all processes receive the result vector
- MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw
 - each process sends messages to all processes

MPI_AllGather

- C:

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```
- Fortran:

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
RECVCOUNT, RECVTYPE, COMM, IERROR)
TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; TYPE(MPI_Comm) :: comm
INTEGER :: sendcount, recvcount; INTEGER, OPTIONAL :: ierror
TYPE(MPI_Datatype) :: sendtype, recvtype
```
- Gathers data from all members of a group and sends the data to all members of the group.
- similar to the MPI_Gather, except that it sends the data to all processes instead of only to the root.



MPI_Gatherv

- C:

```
int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,  
int *recvcounts, int *displs, MPI_Datatype recvttype, int root, MPI_Comm comm)
```

- Fortran:

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,  
RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, IERROR)  
TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; TYPE(MPI_Comm) :: comm  
INTEGER :: sendcount, recvcounts(*), displs(:), root;  
TYPE(MPI_Datatype) :: sendtype, recvttype; INTEGER, OPTIONAL :: ierror
```

- recvcounts: the number of elements that is received from each process. Each element in the array corresponds to the rank of the sending process.
- displs: The location, relative to the recvbuf parameter, of the data from each communicator process. The data that is received from process j is placed into the receive buffer of the root process offset displs[j] elements from the sendbuf pointer.

Example - MPI_Gatherv

```
recvcounts[4]={0, 1, 2, 3};  
int displs[4]={0, 0, 1, 3};
```

```
MPI_Gatherv(sendbuf, rank, MPI_INT, recvbuf, recvcounts, displs,  
MPI_INT, 0, MPI_COMM_WORLD);
```

```
rank=0: sendbuf[0]=0  
rank=1: sendbuf[0]=1  
rank=2: sendbuf[0]=2  
rank=2: sendbuf[1]=2  
rank=3: sendbuf[0]=3  
rank=3: sendbuf[1]=3  
rank=3: sendbuf[2]=3
```

```
recvbuf[0]=1  
recvbuf[1]=2  
recvbuf[2]=2  
recvbuf[3]=3  
recvbuf[4]=3  
recvbuf[5]=3
```

Nonblocking Collective Communication Routines

- Combines nonblocking functionality with a collective communication
- Preventing deadlocks, overlapping communication with computation and deferring synchronization together with a group communication.
- Nonblocking variants of all collective communication:
 - MPI_Ibarrier, MPI_Ibcast, MPI_Isscatter, MPI_Igather, MPI_Isscatter, MPI_Ireduce, ... (*Chapter 5 mpi31-report.pdf*)
- MPI_I... calls are local (i.e., not synchronizing), whereas the corresponding MPI_Wait collectively synchronizes in same way as corresponding blocking collective procedure
- The output request is the same request object with p2p comm.
- All completion calls are supported: MPI_Waitall, MPI_Waitany, MPI_Testall, ...

Nonblocking Collective Communication Routines

- Can not be canceled like nonblocking p2p operations.
- Nonblocking collective operations do not match with blocking collective operations
- Send and recv buffers must not be modified while the operation is in progress.
- May have multiple outstanding collective communications on same communicator
- Ordered initialization on each communicator

With point-to-point message passing, such matching is allowed

References: [1], [4] and [8]

Nonblocking Barrier

- C:
`int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)`
- Fortran:
`MPI_IBARRIER(COMM, REQUEST, IERROR)`
`TYPE(MPI_Comm) :: comm; TYPE(MPI_Request) :: request`
`INTEGER, OPTIONAL :: ierror`
- Performs a barrier synchronization across all members of a group in a non-blocking way
- MPI_Ibarrier is very useful routine:
 - each process starts the barrier after it finishes its local part and serves the requests of other processes until the barrier is reached by all others and completes



Example - Nonblocking Barrier

```
MPI_Request reqs[m], barrier_request; //We send m messages
```

```
int barrier_done=0, barrier_active=0;
```

Each as a sender loops over its neighbors, sending the data

```
for(i=0;i<m;i++) MPI_Issend(sbuf[i], size[i], type, dst[i], tag, comm, &reqs[i]);
```

```
while(!barrier_done)
```

Loop until all signaled that all receives are called

```
    MPI_Iprobe(MPI_ANY_Source, tag, comm, &flag, &stat);
```

```
    if(flag){ //allocate buffer and receive message }
```

Check if there is a message then MPI_Recv for this one msg

```
    if(!barrier_active){
```

```
        int flag;
```

```
        MPI_Testall(m, reqs, &flag, MPI_STATUSES_IGNORE);
```

Check whether all Issend calls are finished

```
        if(flag){
```

```
            MPI_Ibarrier(comm, &barrier_request);
```

```
            barrier_active=1;
```

Start MPI_Ibarrier to signal to all that all MPI_Issend of this process are already received

```
        }
```

```
    else{
```

```
        MPI_Test(&barrier_request, &barrier_done, MPI_STATUS_IGNORE);
```

```
    }
```

```
}
```

Nonblocking Reduce

- C:

```
int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,  
MPI_Request *request)
```
- Fortran:

```
MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,  
ROOT, COMM, REQUEST, IERROR)  
TYPE(*), DIMENSION(..) :: sendbuf, recvbuf; INTEGER :: count, root  
TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm;  
TYPE(MPI_Op) :: op; INTEGER, OPTIONAL :: ierror  
TYPE(MPI_Request) :: request
```
- Reduces values on all processes to a single value in a non-blocking way

MPI_Probe, MPI_Iprobe

- C:
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int **flag*, MPI_Status **status*)
- Fortran:
MPI_IPROBE(source, tag, comm, *flag*, *status*, *error*)
INTEGER :: source, tag; INTEGER, OPTIONAL :: error
TYPE(MPI_Comm) :: comm; TYPE(MPI_Status) :: status
LOGICAL :: flag
- flag = true if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm.
- flag = false leaves status undefined
- MPI_ANY_SOURCE: messages from an arbitrary source
- MPI_ANY_TAG: messages with an arbitrary tag

Example – MPI Iprobe

```
int main(int argc, char* argv[]){
    int myRank, ierror, a[5], i, flag=0;
    MPI_Status status;
    MPI_Request request;

    ierror=MPI_Init(&argc, &argv);
    ierror=MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if(myRank ==0){
        a[0]=2345; a[1]=654; a[2]=96574; a[3]=-12; a[4]=7676;
        int tag=myRank;
        printf("Process %d: sending the message.\n", myRank);
        ierror=MPI_Issend(a, 5, MPI_INT, 1, tag, MPI_COMM_WORLD,&request);
        ierror=MPI_Wait(&request,&status);
    }
    else if(myRank == 1){
        while (flag == 0){
            ierror=MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
            printf("After MPI_Iprobe, flag = %d\n", flag);
        }
        ierror=MPI_Recv(a, 5, MPI_INT, status.MPI_SOURCE, status.MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process %d: message received.\n", myRank);
        for(i=0; i<5; i++) printf("a[%d]=%d\n", i, a[i]);
    }

    ierror=MPI_Finalize();
    return ierror;
}
```