Collective Communications

Reusing this material

Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronisation.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

Characteristics of Collective Comms

- Collective action over a communicator.
- All processes must communicate.
- Synchronisation may or may not occur.
- Standard collective operations are blocking.
  - non-blocking versions introduced into MPI 3.0
  - may be useful in some situations but not yet commonly employed
  - obvious extension of blocking version: extra request parameter
- No tags.
- Receive buffers must be exactly the right size.

Barrier Synchronisation

- C:

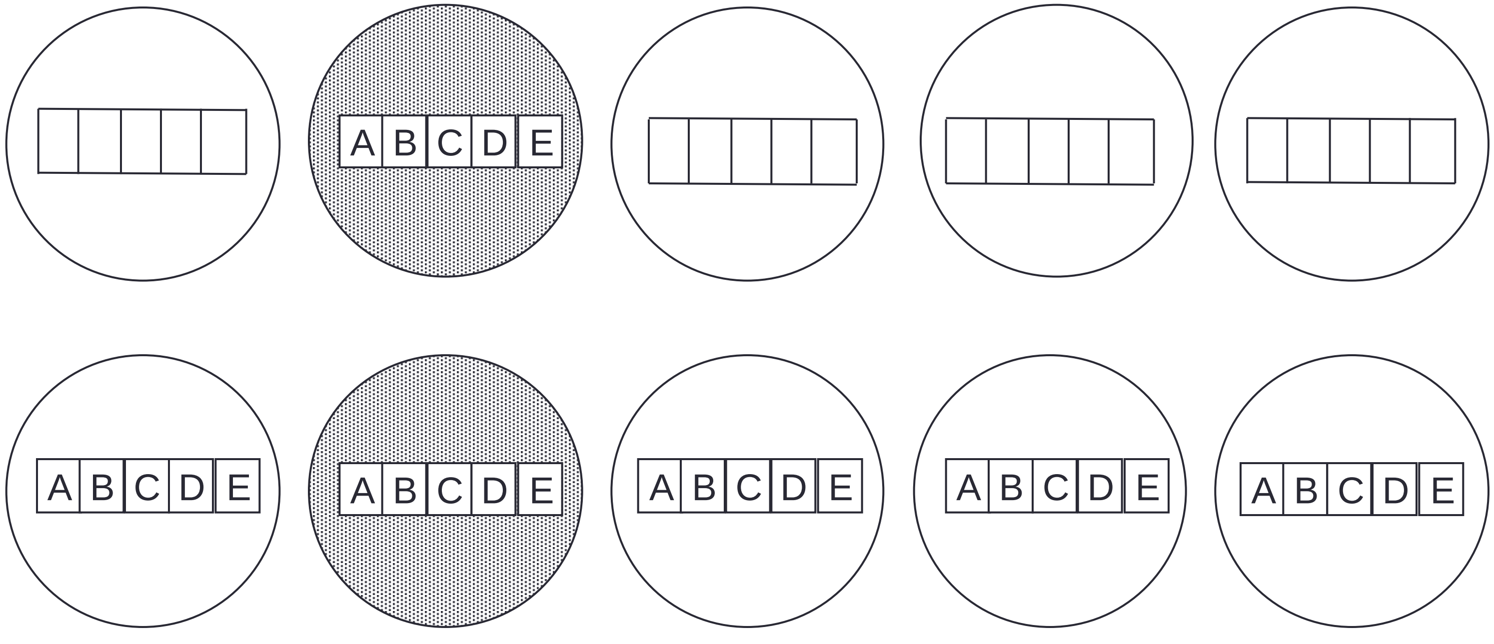
```
int MPI_Barrier (MPI_Comm comm)
```

- Fortran:

```
MPI_BARRIER (COMM, IERROR)

INTEGER COMM, IERROR
```

Broadcast



Broadcast

- C:

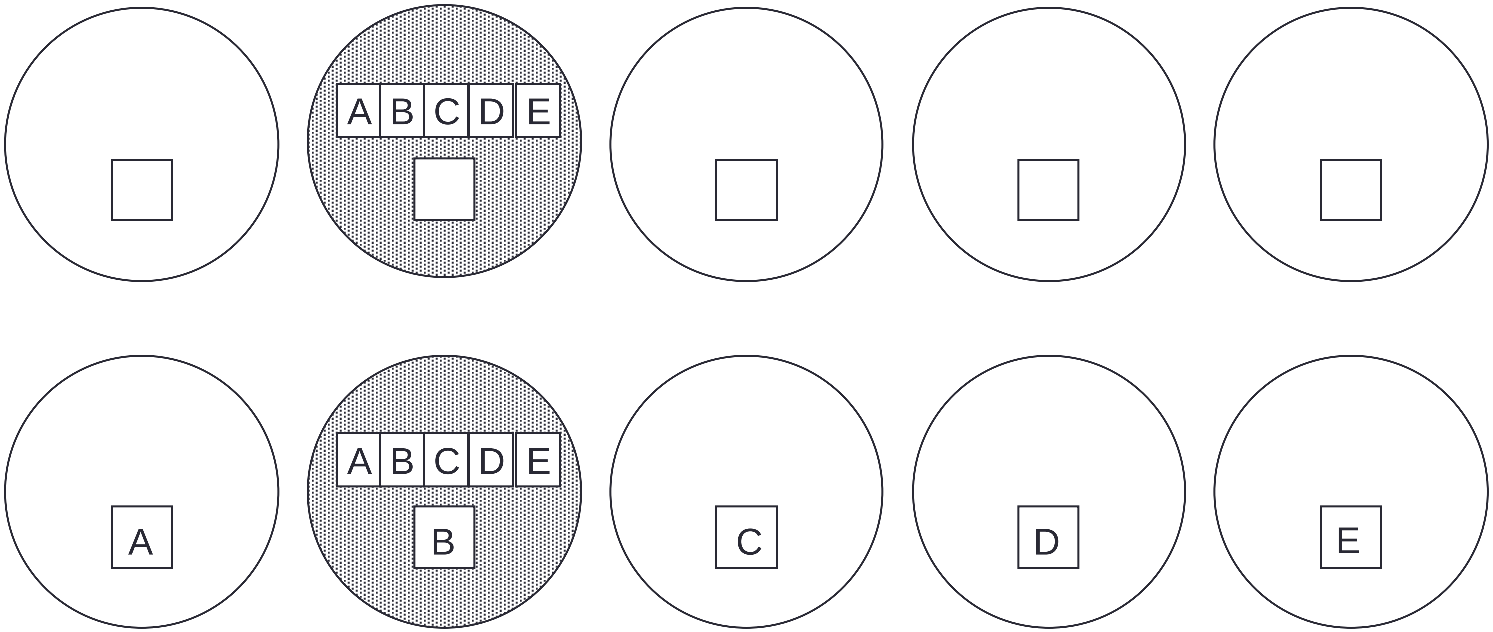
```
int MPI_Bcast (void *buffer, int count,
               MPI_Datatype datatype, int root,
               MPI_Comm comm)
```

- Fortran:

```
MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT,
           COMM, IERROR)

<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

Scatter



Scatter

- C:

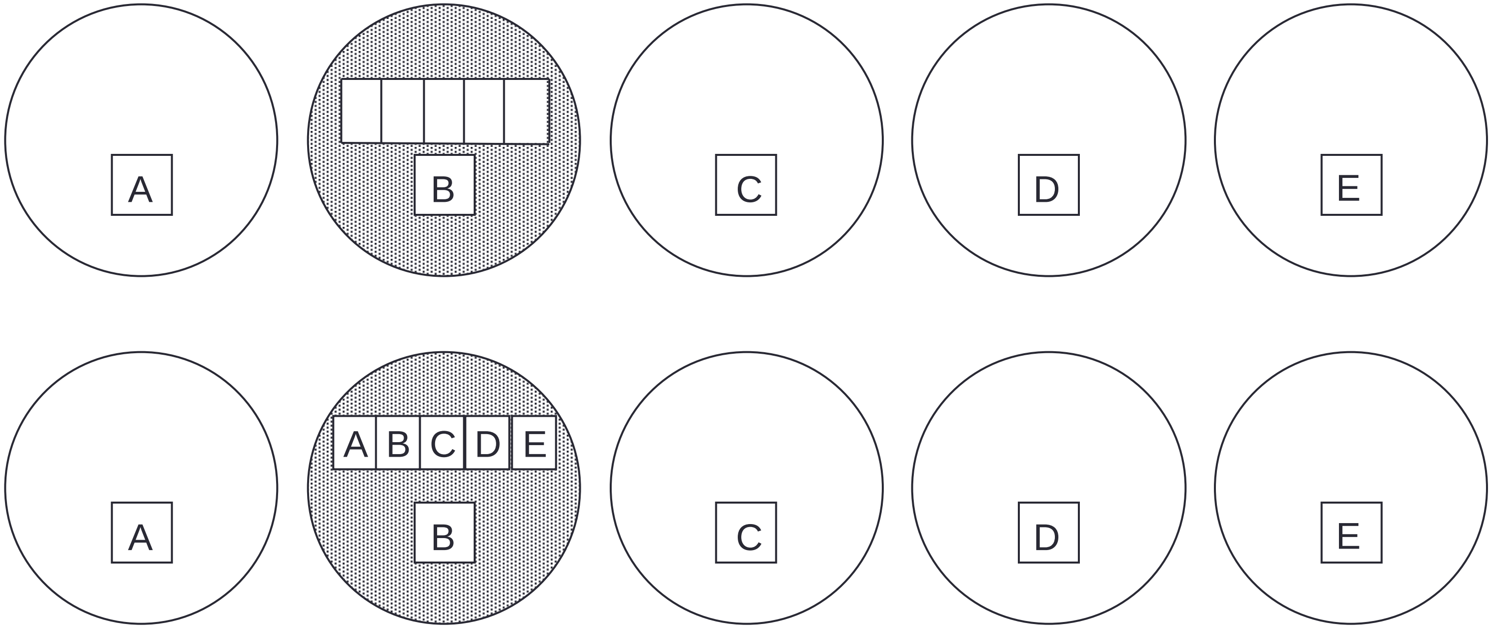
```
int MPI_Scatter(void *sendbuf,
      int sendcount, MPI_Datatype sendtype,
      void *recvbuf, int recvcount,
      MPI_Datatype recvtype, int root,
      MPI_Comm comm)
```

- Fortran:

```
      MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE,
                  RECVBUF, RECVCOUNT, RECVTYPE,
                  ROOT, COMM, IERROR)

      <type> SENDBUF, RECVBUF
      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
      INTEGER RECVTYPE, ROOT, COMM, IERROR
```

Gather



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>  SENDBUF, RECVBUF
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

More general routines

- Basic scatter and gather routines are quite restrictive
  - fixed amount of data transferred between every pair pf processes
  - all data assumed to be contiguous in the larger array
- More general "vector" versions exist
  - **`MPI_Scatterv()`** and **`MPI_Gatherv()`**
  - additional integer arrays (i.e. "vectors") with **`size`** entries specifying
    - different counts for each process
    - different locations of data for each process (i.e. index into the buffer)
- Even more complicated patterns can be achieved
  - using derived datatypes for sendtype and/or recvtype
  - beyond the scope of this course

Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.
- Examples:
  - global sum or product
  - global maximum or minimum
  - global user-defined operation

Predefined Reduction Operations

MPI Name	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
`MPI_MINLOC`	Minimum and location

MPI_Reduce

- C:

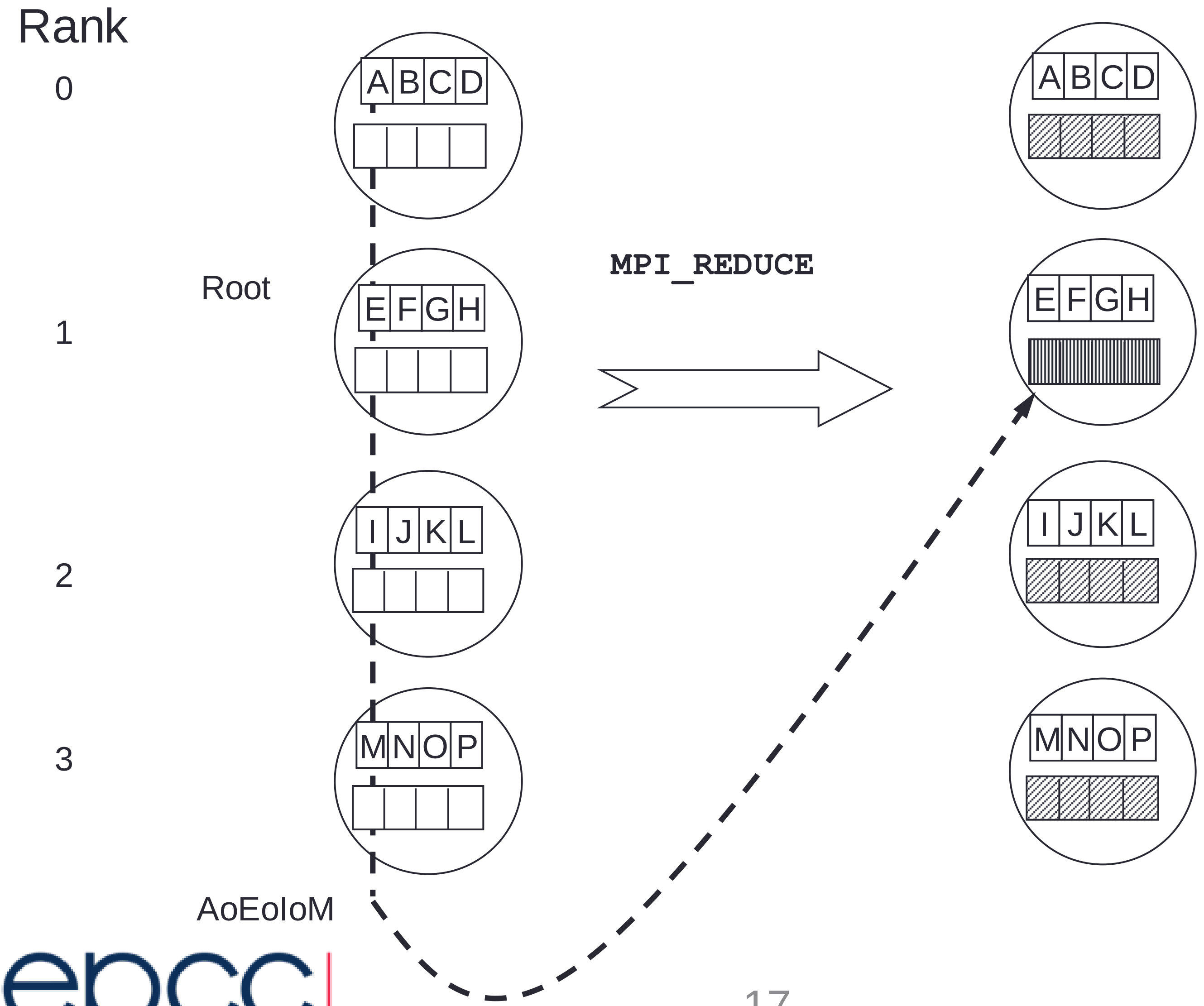
```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)
```

- Fortran:

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT,
           DATATYPE, OP, ROOT, COMM, IERROR)

<type>    SENDBUF, RECVBUF
INTEGER   SENDCOUNT, SENDTYPE, RECVCOUNT
INTEGER   RECVTYPE, ROOT, COMM, IERROR
```

MPI_REDUCE


Rank
0
1
2
3
Root
A B C D
E F G H
I J K L
M N O P
MPI_REDUCE
A B C D
E F G H
I J K L
M N O P
AoEoIoM

Example of Global Reduction

Integer global sum

- C:

```
MPI_Reduce(&x, &result, 1, MPI_INT,
           MPI_SUM,0, MPI_COMM_WORLD)
```

- Fortran:

```
CALL MPI_REDUCE(x, result, 1, MPI_INTEGER,
                MPI_SUM, 0,
                MPI_COMM_WORLD, IERROR)
```

- Sum of all the **`x`** values is placed in **`result`**.
- The result is only placed there on process 0.

User-Defined Reduction Operators

- Reducing using an arbitrary operator, o
- C - function of type **`MPI_User_Function`**:

```
void my_op (void *invec, void *inoutvec, int *len,
            MPI_Datatype *datatype)
```

- Fortran - external subprogram of type

```
SUBROUTINE MY_OP(INVEC(*), INOUTVEC(*), LEN,
                 DATATYPE)
<type>   INVEC(LEN), INOUTVEC(LEN)
INTEGER  LEN, DATATYPE
```

Reduction Operator Functions

- Operator function for o must act as

```
for (i = 1 to len)
  inoutvec(i) = inoutvec(i)  o  invec(i)
```

- Operator o need not commute, but must be associative

Registering User-Defined Operator

- Operator handles have type **`MPI_Op`** or **`INTEGER`**

- C:

```
int MPI_Op_create(MPI_User_function *my_op,
                  int commute, MPI_Op *op)
```

- Fortran:

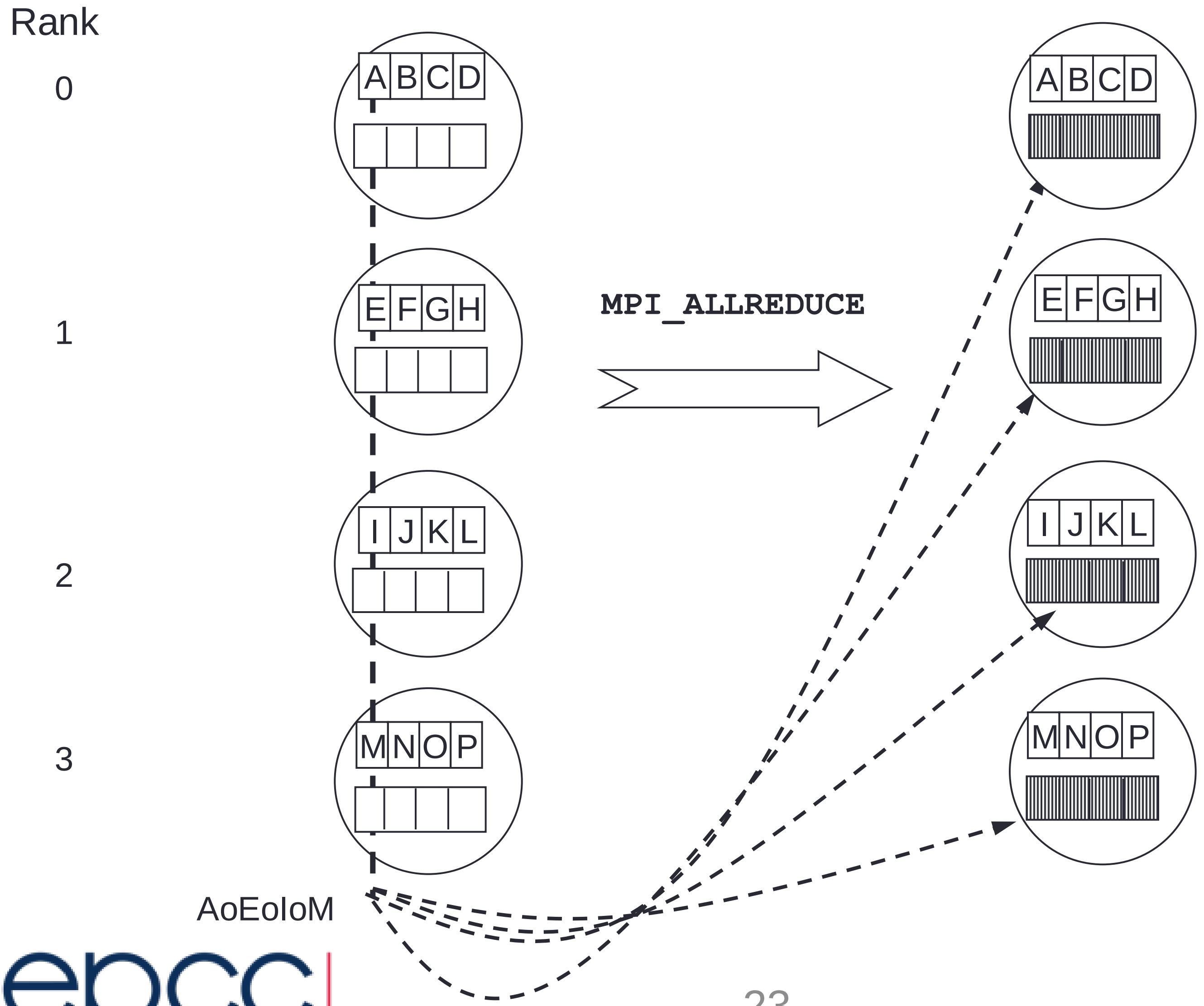
```
MPI_OP_CREATE (MY_OP, COMMUTE, OP, IERROR)

EXTERNAL MY_OP
LOGICAL  COMMUTE
INTEGER  OP, IERROR
```

Variants of MPI_REDUCE

- **`MPI_Allreduce`** no root process
- **`MPI_Reduce_scatter`** result is scattered
- **`MPI_Scan`** “parallel prefix”

MPI_ALLREDUCE



MPI_ALLREDUCE

Integer global sum

- C:

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

- Fortran:

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT,
              DATATYPE, OP, COMM, IERROR)
```

Allreduce example

Integer global sum

- C:

```
MPI_Allreduce(&x, &result, 1, MPI_INT,
              MPI_SUM, MPI_COMM_WORLD)
```

- Fortran:

```
CALL MPI_ALLREDUCE(x, result, 1, MPI_INTEGER,
                   MPI_SUM,
                   MPI_COMM_WORLD, IERROR)
```

- Sum of all the **`x`** values is placed in **`result`**.
- The result is stored on every process

Vector reductions (Fortran)

```
double precision, dimension(3) :: localdata, globaldata

localdata(1) = pressure
localdata(2) = temperature
localdata(3) = rainfall

call mpi_allreduce(localdata, globaldata, 3,          &
                   MPI_DOUBLE_PRECISION, MPI_SUM,   &
                   MPI_COMM_WORLD, ierr)

write(*,*) "global P, T and R = ", globaldata(:)
```

Vector reductions (C)

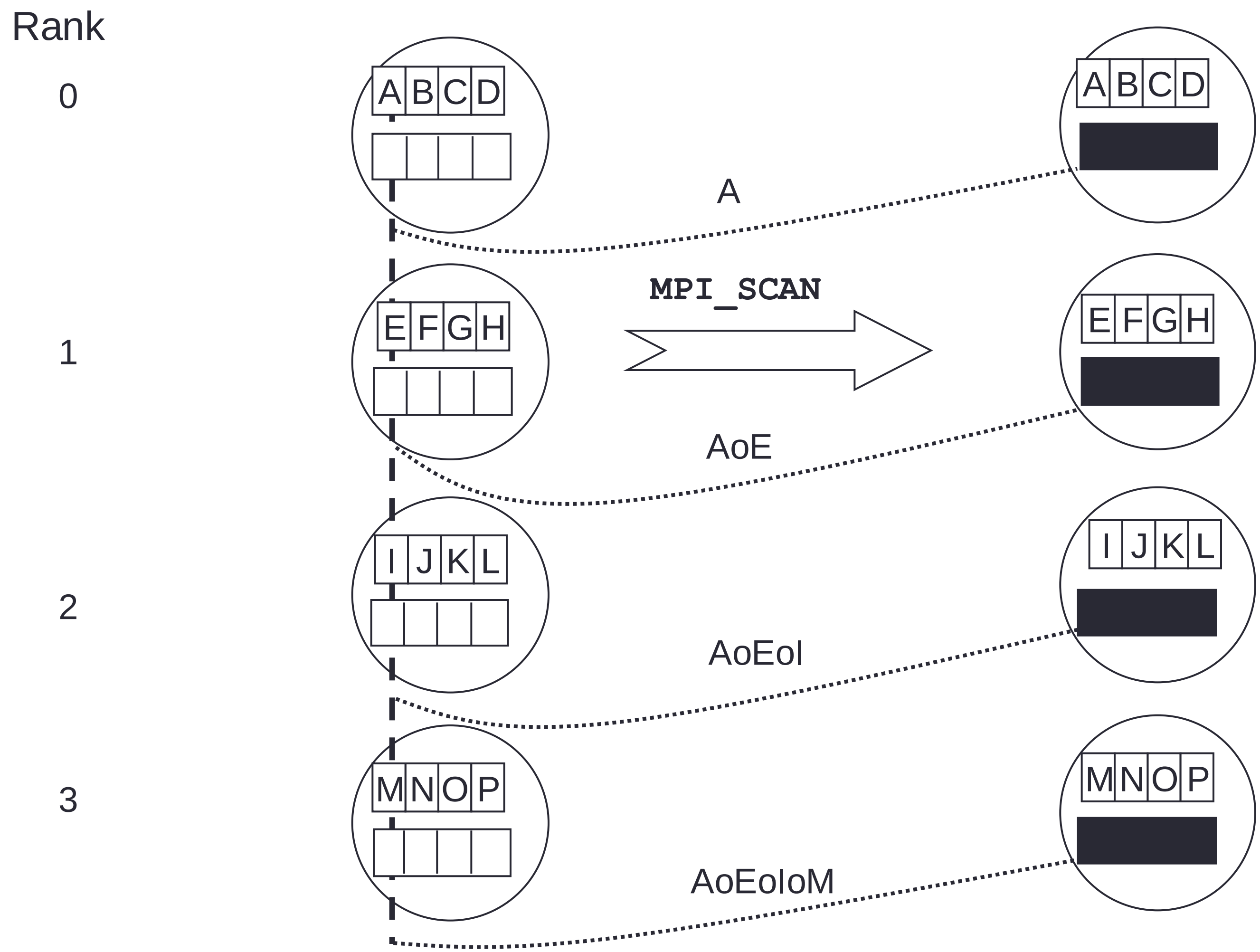
```
double localdata[3], globaldata[3];

localdata[0] = pressure;
localdata[1] = temperature;
localdata[2] = rainfall;

MPI_Allreduce(localdata, globaldata, 3,
              MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

printf("global P, T and R = %f, %f, %f\n",
       globaldata[0], globaldata[1], globaldata[2] );
```

MPI_SCAN


Rank
0
1
2
3
A B C D
E F G H
I J K L
M N O P
A B C D
E F G H
I J K L
M N O P
MPI_SCAN
A
AoE
AoEoI
AoEoIoM

MPI_SCAN

Integer partial sum

- C:

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

- Fortran:

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT,
         DATATYPE, OP, COMM, IERROR)
```

- Scan is inclusive
  - *includes* the result on calling process
  - for *exclusive* scan use **`MPI_Exscan()`**

Exercise

- See Exercise 5 on the sheet
- Rewrite the pass-around-the-ring program to use MPI global reduction to perform its global sums.
- Then rewrite it so that each process computes a partial sum