

Collective Communications

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

- ▶ Collective action over a communicator.
- ▶ All processes must communicate.
- ▶ Synchronisation may or may not occur.
- ▶ Standard collective operations are blocking.
 - non-blocking versions recently 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.

▶ C:

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

▶ Fortran:

```
MPI_BARRIER (COMM, IERROR)  
INTEGER COMM, IERROR
```

▶ C:

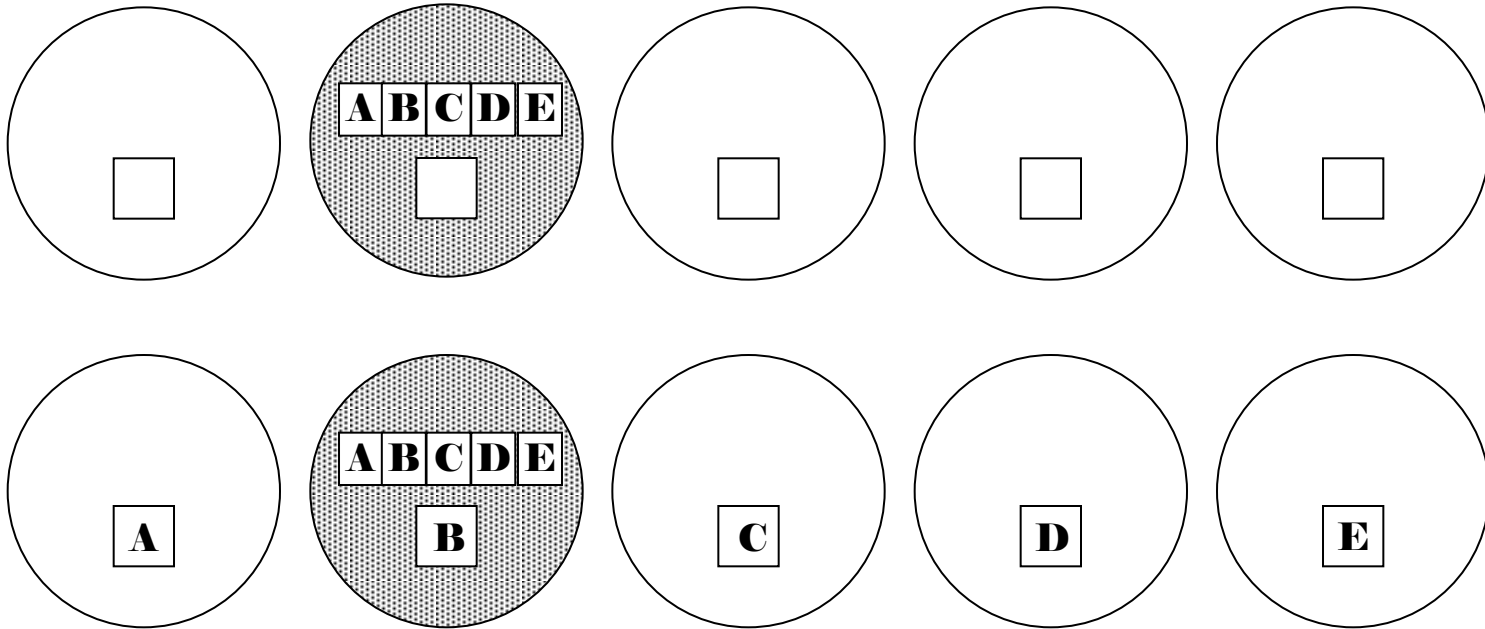
```
int MPI_Bcast (void *buffer, int count,  
              MPI_Datatype datatype, int root,  
              MPI_Comm comm)
```

▶ Fortran:

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

```
<type> BUFFER(*)
```

```
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```



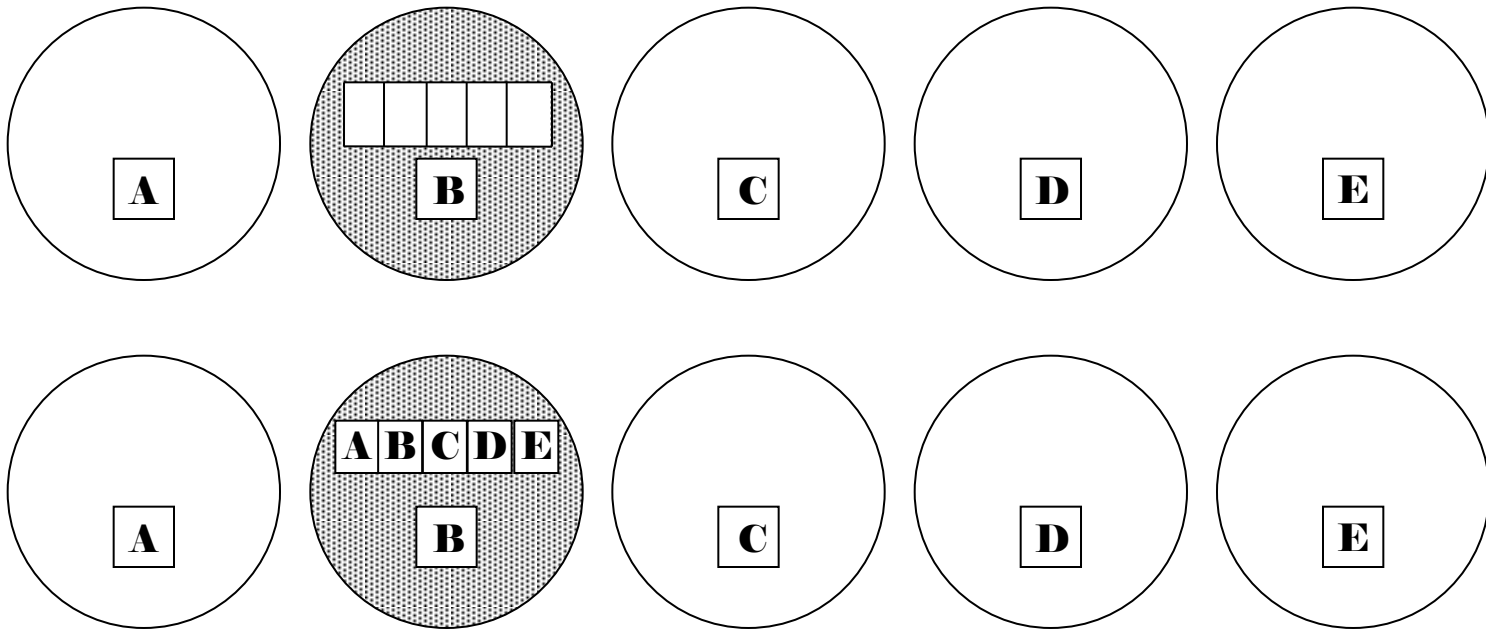
▶ C:

```
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
```



▶ 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
```

- ▶ 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

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

▶ C:

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

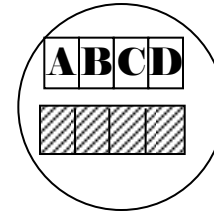
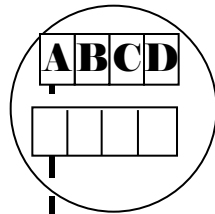
▶ Fortran:

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

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

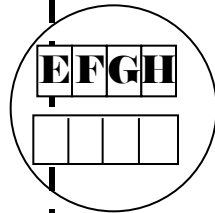
Rank

0

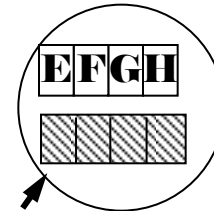
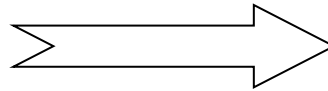


1

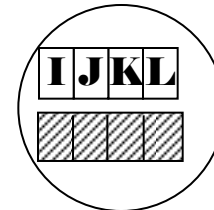
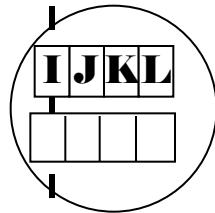
Root



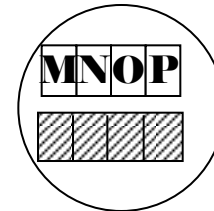
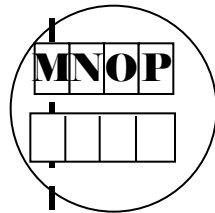
MPI_REDUCE



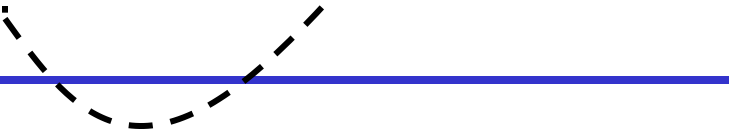
2



3



AoEoIoM



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 processor 0.

- ▶ Reducing using an arbitrary operator, ◦
- ▶ 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
```

- ▶ Operator function for \circ must act as:

```
for (i = 1 to len)
```

```
    inoutvec(i) = inoutvec(i)  $\circ$  invec(i)
```

- ▶ Operator \circ need not commute but must be associative.

- ▶ Operator handles have type `MPI_Op` or `INTEGER`
- ▶ C:

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

- ▶ Fortran:

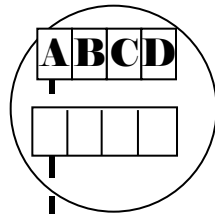
```
MPI_OP_CREATE (MY_OP, COMMUTE, OP, IERROR)
```

```
EXTERNAL MY_OP  
LOGICAL COMMUTE  
INTEGER OP, IERROR
```

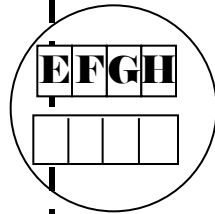
- ▶ `MPI_Allreduce` no root process
- ▶ `MPI_Reduce_scatter` result is scattered
- ▶ `MPI_Scan` “parallel prefix”

Rank

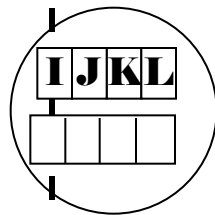
0



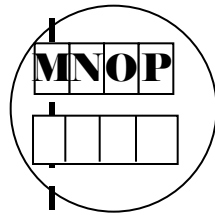
1



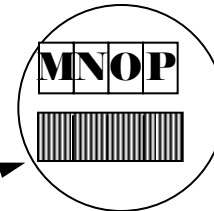
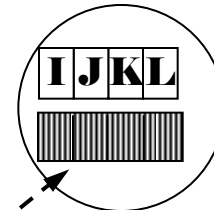
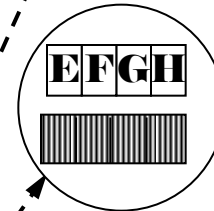
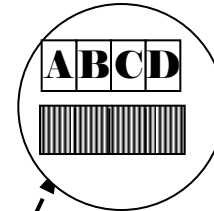
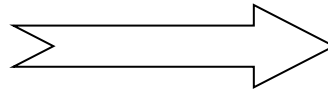
2



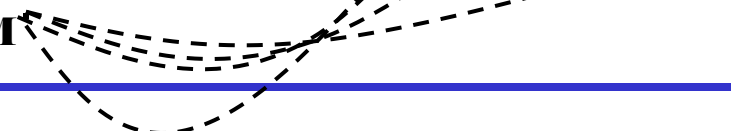
3



MPI_ALLREDUCE



AoEoIoM



Integer global sum

▶ C:

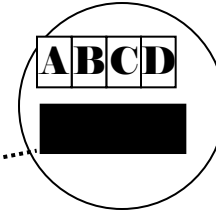
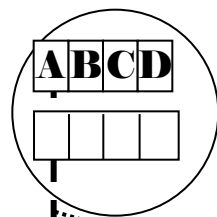
```
int MPI_Allreduce(void* sendbuf,  
                 void* recvbuf, int count,  
                 MPI_Datatype datatype,  
                 MPI_Op op, MPI_Comm comm)
```

▶ Fortran:

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

Rank

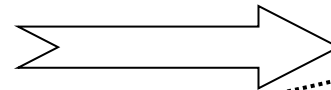
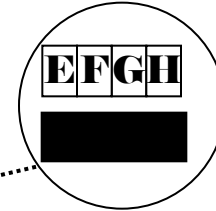
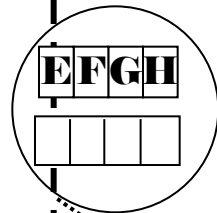
0



A

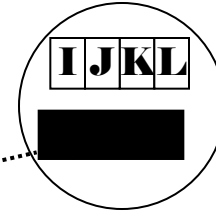
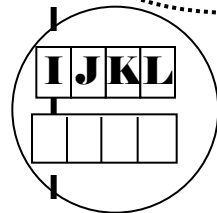
MPI_SCAN

1



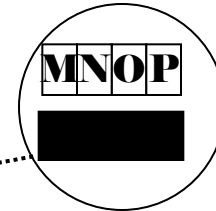
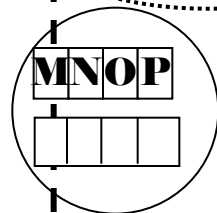
AoE

2

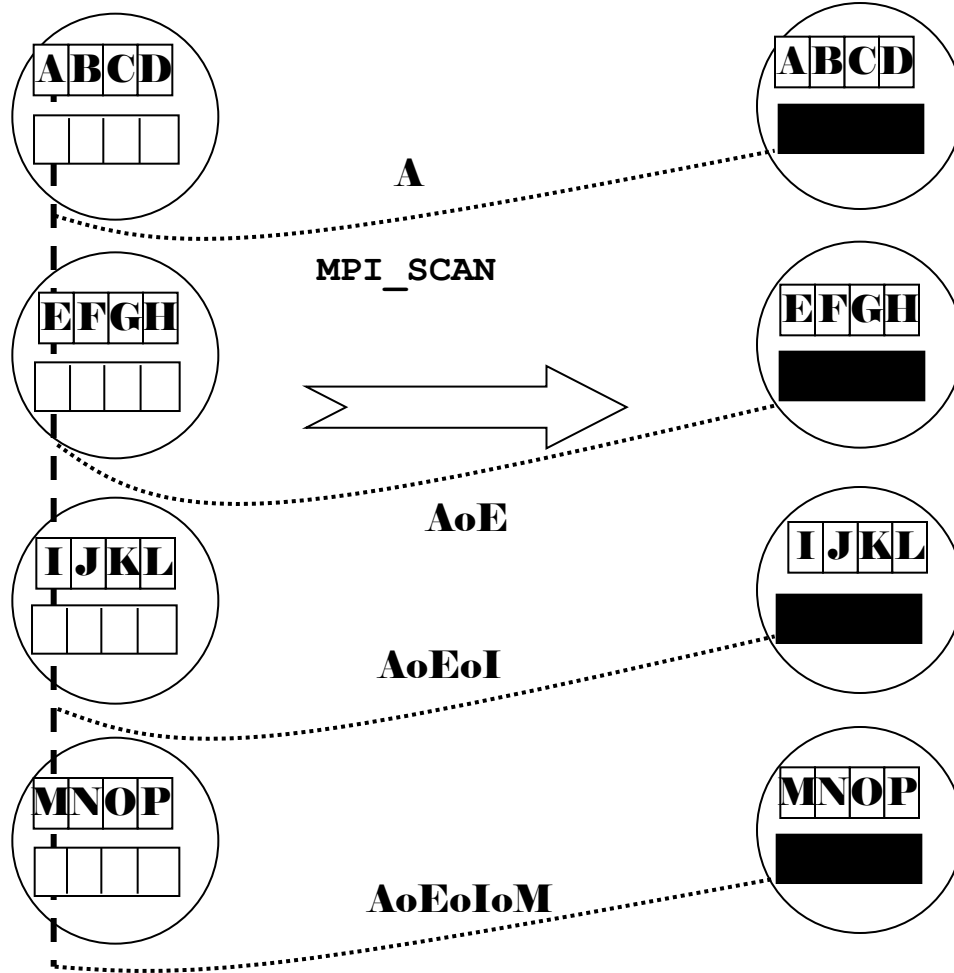


AoEoI

3



AoEoIoM



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)
```

- ▶ 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