Collective Communication

- Collective communication
- Barrier synchronization
- Broadcast*
- Scatter*
- Gather
- Gather/Scatter variations
- Summary illustration
- Global Reduction operations
- Predefined reduction operations
- `MPI_Reduce`
- Minloc and Maxloc*
- User-Defined reduction operations
- Reduction operator functions
- Registering a user-defined reduction operator*
- Variants of `MPI_Reduce`
- Class Exercise: Last Ring

*includes sample C and Fortran programs

Collective communication

- Communications involving a group of processes
- Called by **all** processes in a communicator
- Examples:
  - Barrier (synchronization)
  - Broadcast, scatter, gather (data distribution)
  - Global sum, global maximum, etc. (collective operations)

Characteristics of collective communication

- Collective communication will not interfere with point-to-point communication and vice-versa
- All processes must call the collective routine
- Synchronization not guaranteed (except for barrier)
- No non-blocking collective communication
- No tags
- Receive buffers must be exactly the right size

Barrier synchronization

- Red light for each processor: turns green when all processors have arrived

C:

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

Fortran:

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

Broadcast

- One-to-all communication: same data sent from root process to all the others in the communicator

- C:

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

- Fortran:

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

- All processes must specify same root rank and communicator

Sample Program #5 - C

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

```
Program Output
P:0 after broadcast parameter is 23.000000
P:6 after broadcast parameter is 23.000000
P:5 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000
P:7 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:4 after broadcast parameter is 23.000000
```

Sample Program #5 - Fortran

```
PROGRAM broadcast
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
if(rank.eq.5) param=23.0
call MPI_BCAST(param,1,MPI_REAL,5,MPI_COMM_WORLD,err)
print *,"P:",rank," after broadcast param is ",param
CALL MPI_FINALIZE(err)
END
```

```
Program Output
P:1 after broadcast parameter is 23.
P:3 after broadcast parameter is 23.
P:4 after broadcast parameter is 23
P:0 after broadcast parameter is 23
P:5 after broadcast parameter is 23.
P:6 after broadcast parameter is 23.
P:7 after broadcast parameter is 23.
P:2 after broadcast parameter is 23.
```

Scatter

- One-to-all communication: different data sent to each process in the communicator (in rank order)

C:

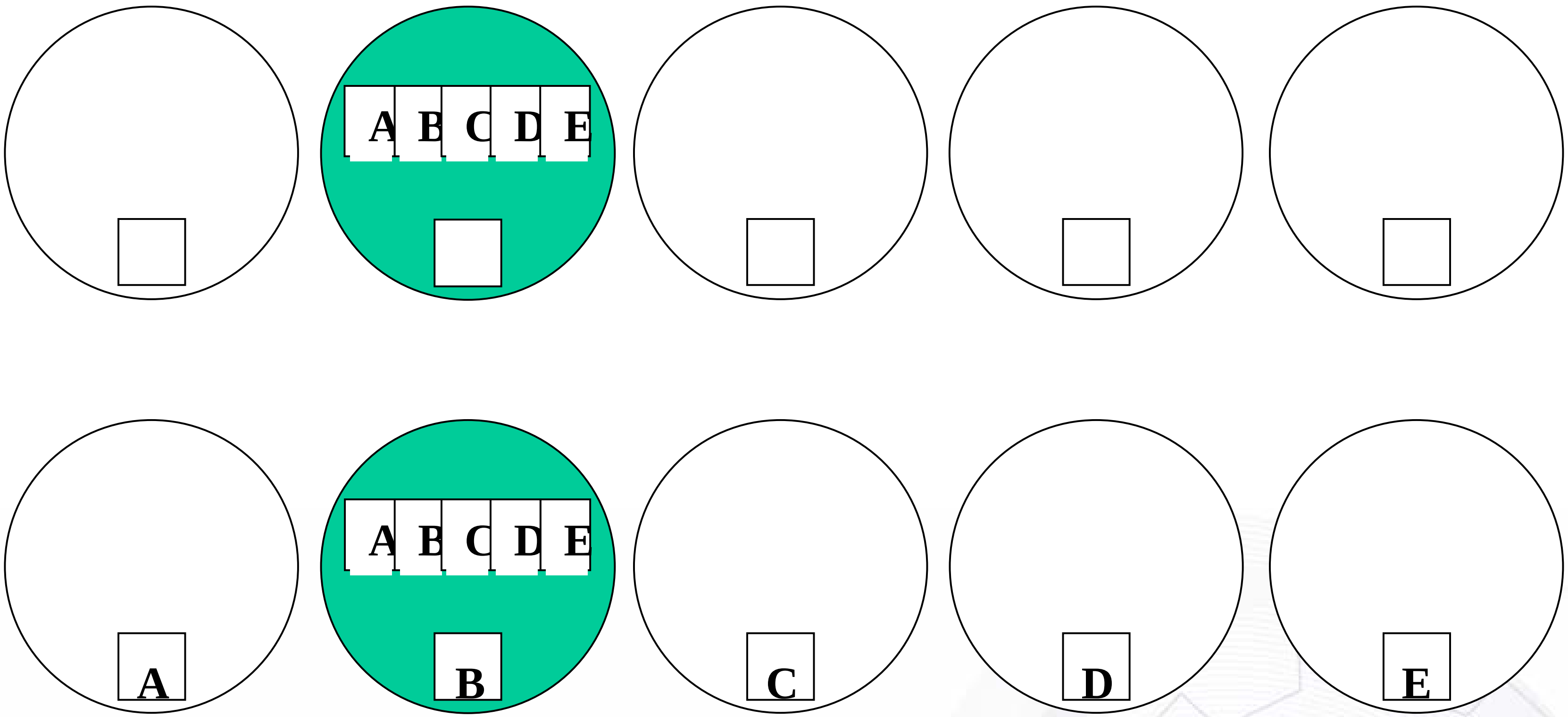
```
int MPI_Scatter(void* sendbuf, int sendcount,
    MPI_Datatype sendtype, void* recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Fortran:

```
CALL MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
    RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
```

- `sendcount` is the number of elements sent to each process, not the "total" number sent
  - send arguments are significant only at the root process

Scatter example



Sample Program #6 - C

```
#include <mpi.h>
 int main (int argc, char *argv[]) {
    int rank,size,i,j;
    double param[4],mine;
    int sndcnt,revcnt;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    revcnt=1;
    if(rank==3){
       for(i=0;i<4;i++) param[i]=23.0+i;
       sndcnt=1;
    }
    MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,revcnt,MPI_DOUBLE,3,
                MPI_COMM_WORLD);
    printf("P:%d mine is %f\n",rank,mine);
    MPI_Finalize();
 }
```

```
     Program Output
P:0 mine is 23.000000
P:1 mine is 24.000000
P:2 mine is 25.000000
P:3 mine is 26.000000
```

Sample Program #6 - Fortran

```
PROGRAM scatter
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param(4), mine
integer sndcnt,rcvcnt
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
rcvcnt=1
if(rank.eq.3) then
   do i=1,4
      param(i)=23.0+i
   end do
   sndcnt=1
end if
call MPI_SCATTER(param,sndcnt,MPI_REAL,mine,rcvcnt,MPI_REAL,
&            3,MPI_COMM_WORLD,err)
print *,"P:",rank," mine is ",mine
CALL MPI_FINALIZE(err)
END
```

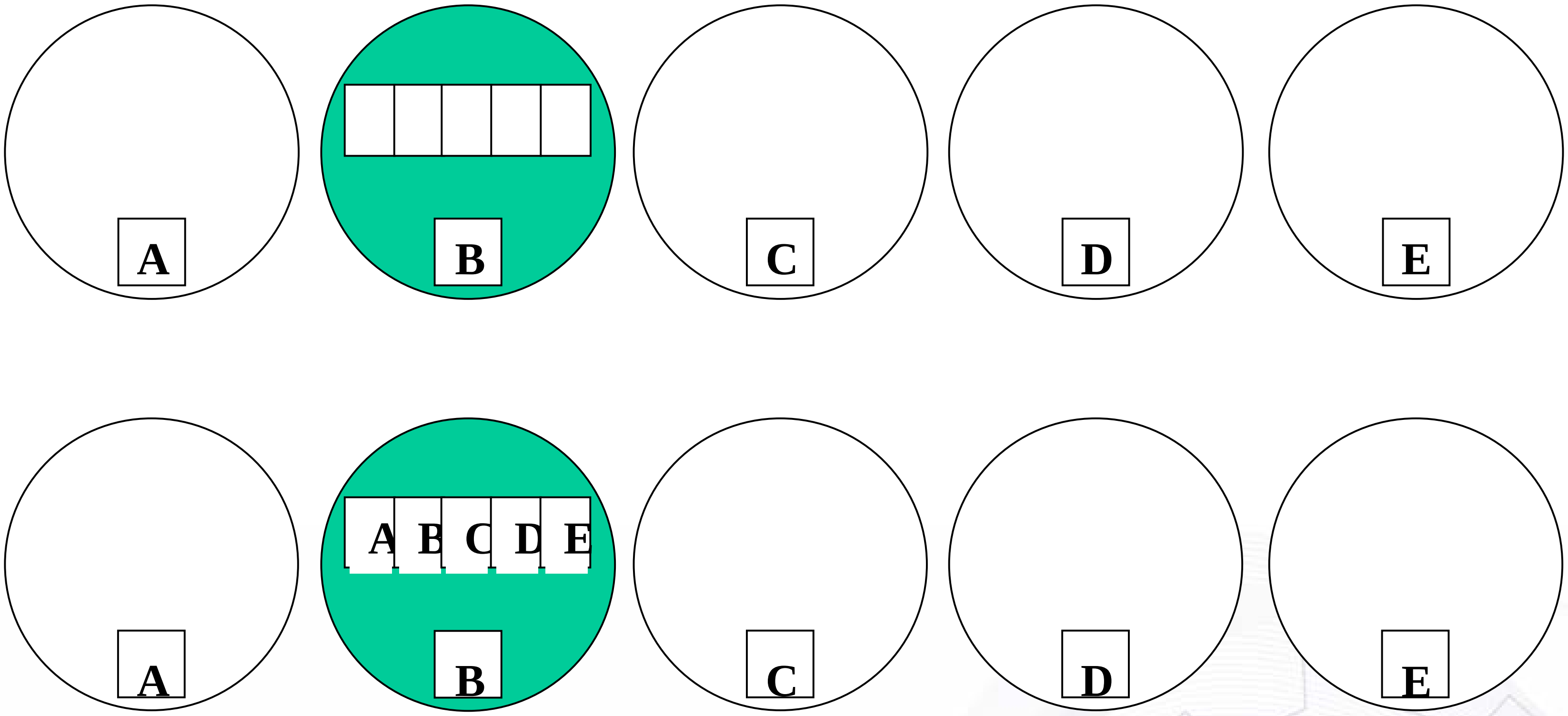
```
  Program Output
P:1 mine is 25.
P:3 mine is 27.
P:0 mine is 24.
P:2 mine is 26.
```

Ohio Supercomputer Center

Gather

- All-to-one communication: different data collected by root process
  - Collection done in rank order
- `MPI_GATHER` & `MPI_Gather` have same arguments as matching scatter routines
- Receive arguments meaningful only at the root process

Gather example



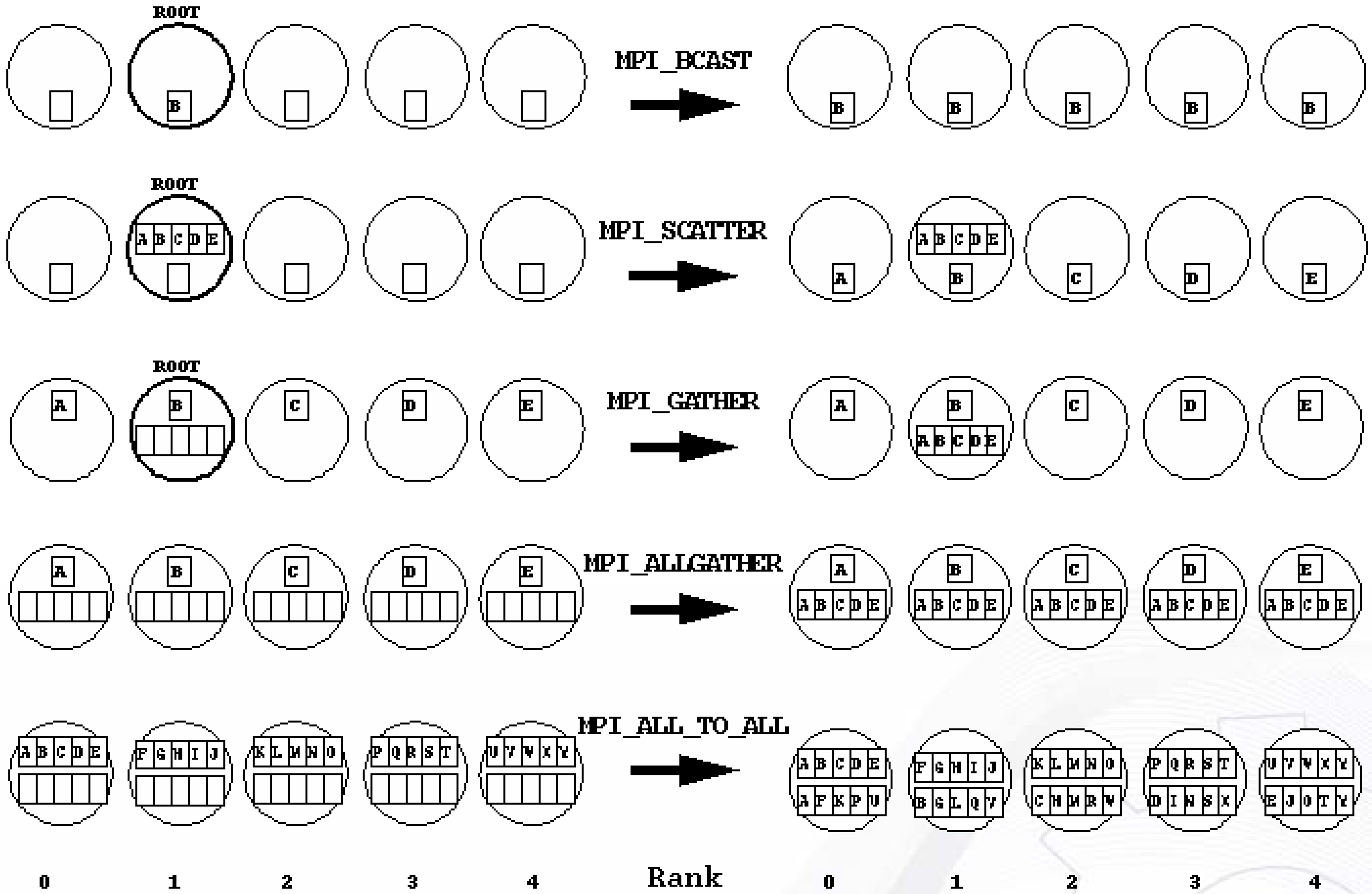
Ohio Supercomputer Center

Gather/Scatter variations

- `MPI_Allgather`
- `MPI_Alltoall`

- No root process specified: all processes get gathered or scattered data
- Send and receive arguments significant for all processes

Summary


ROOT
MPI_BCAST
MPI_SCATTER
MPI_GATHER
MPI_ALLGATHER
MPI_ALL_TO_ALL
Rank
0
1
2
3
4

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

Global Reduction Operations

C:

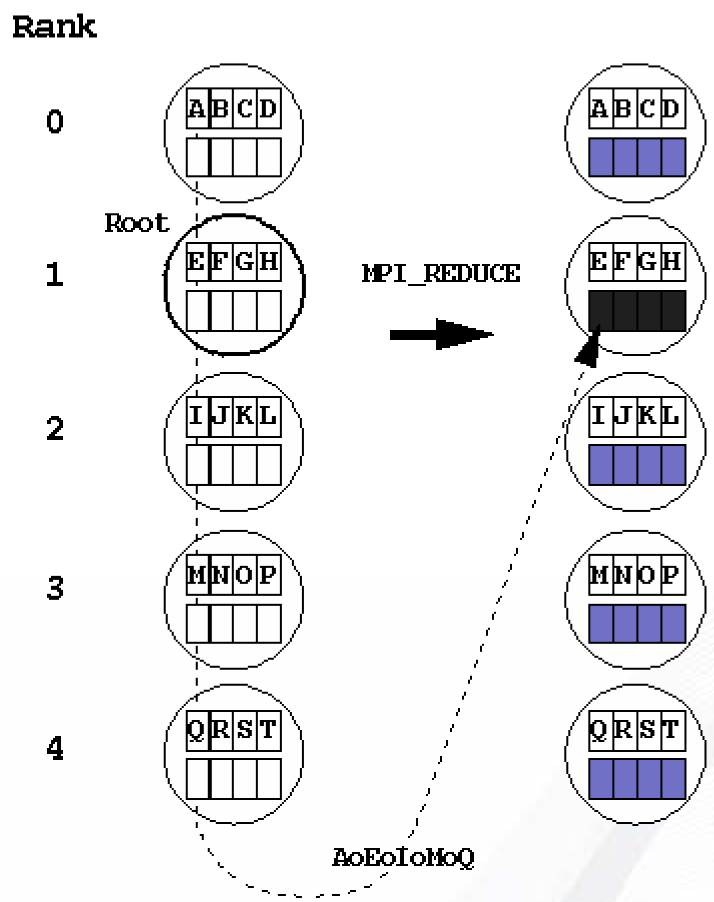
```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
       MPI_Datatype datatype, MPI_Op op, int root,
       MPI_Comm comm)
```

Fortran:

```
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
CALL MPI_REDUCE(SENDBUF,RECVBUF,COUNT,DATATYPE,OP,ROOT,COMM,
                IERROR)
```

- op is an associative operator that takes two operands of type `datatype` and returns a result of the same type
- `count` is the number of "*ops*" done on consecutive elements of `sendbuf` (it is also size of `recvbuf`)

MPI_Reduce



Rank
0
1
2
3
4
Root
A B C D
E F G H
I J K L
M N O P
Q R S T
MPI_REDUCE
AoEoIoMoQ

MPI_MINLOC, MPI_MAXLOC

- Designed to compute a global minimum/maximum and an index associated with the extreme value
  - Common application: index is the processor rank (see sample program)
- If more than one extreme, get the first
- Designed to work on operands that consist of a value and index pair
- MPI_Datatypes include:

C:

```
MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT,
   MPI_SHORT_INT, MPI_LONG_DOUBLE_INT
```

Fortran:

```
MPI_2REAL, MPI_2DOUBLEPRECISION, MPI_2INTEGER
```

Ohio Supercomputer Center

Sample Program #7 - C

```
#include <mpi.h>
/* Run with 16 processes */
int main (int argc, char *argv[]) {
   int rank;

   struct {
     double value;
     int rank;
   } in, out;

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

   in.value=rank+1;
   in.rank=rank;
   root=7;

   MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
   if(rank==root) printf("PE:%d max=%lf at rank %d\n",rank,out.value,out.rank);
   MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MINLOC,root,MPI_COMM_WORLD);
   if(rank==root) printf("PE:%d min=%lf at rank %d\n",rank,out.value,out.rank);
   MPI_Finalize();
}
```

```
        Program Output
P:7 max = 16.000000 at rank 15
P:7 min = 1.000000 at rank 0
```

Ohio Supercomputer Center

Sample Program #7 - Fortran

```
      PROGRAM MaxMin
C
C Run with 8 processes
C
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      integer in(2),out(2)
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
      CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
      in(1)=rank+1
      in(2)=rank
      call MPI_REDUCE(in,out,1,MPI_2INTEGER,MPI_MAXLOC,
     &                7,MPI_COMM_WORLD,err)

      if(rank.eq.7) print *,"P:",rank," max=",out(1)," at rank ",out(2)

      call MPI_REDUCE(in,out,1,MPI_2INTEGER,MPI_MINLOC,
     &                2,MPI_COMM_WORLD,err)

      if(rank.eq.2) print *,"P:",rank," min=",out(1)," at rank ",out(2)

      CALL MPI_FINALIZE(err)
      END
```

```
   Program Output
P:2 min=1 at rank 0
P:7 max=8 at rank 7
```

User-Defined Reduction Operators

- Reducing using an arbitrary operator op

C function of type `MPI_User_function`:

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

Fortran function of type:

```
FUNCTION MY_OPERATOR (INVEC(*), INOUTVEC(*), LEN, DATATYPE)

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

Reduction Operator Functions

- Operator function for `op` must act as:

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

- Operator `op` need not commute
- `inoutvec` argument acts as both a second input operand as well as the output of the function

Registering a User-Defined Reduction Operator

- Operator handles have type `MPI_Op` or `INTEGER`
- If `commute` is `TRUE`, reduction may be performed faster

C:

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

Fortran:

```
EXTERNAL FUNC
INTEGER OP,IERROR
LOGICAL COMMUTE
MPI_OP_CREATE (FUNC, COMMUTE, OP, IERROR)
```

Sample Program #8 - C

```
#include <mpi.h>
 typedef struct {
    double real,imag;
 } complex;

 void cprod(complex *in, complex *inout, int *len, MPI_Datatype *dptr) {
    int i;
    complex c;
    for (i=0; i<*len; ++i) {
       c.real=(*in).real * (*inout).real - (*in).imag * (*inout).imag;
       c.imag=(*in).real * (*inout).imag + (*in).imag * (*inout).real;
       *inout=c;
       in++;
       inout++;
    }
 }

 int main (int argc, char *argv[]) {
    int rank;
    int root;
    complex source,result;
```


Ohio Supercomputer Center

Sample Program #8 - C (cont.)

```
   MPI_Op myop;
    MPI_Datatype ctype;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_contiguous(2,MPI_DOUBLE,&ctype);
    MPI_Type_commit(&ctype);
    MPI_Op_create(cprod,TRUE,&myop);
    root=2;
    source.real=rank+1;
    source.imag=rank+2;
    MPI_Reduce(&source,&result,1,ctype,myop,root,MPI_COMM_WORLD);
    if(rank==root) printf ("PE:%d result is %lf + %lfi\n",rank, result.real,
   result.imag);
    MPI_Finalize();
}

--------------------------------------------

P:2 result is -185.000000 + -180.000000i
```

Sample Program #8 - Fortran

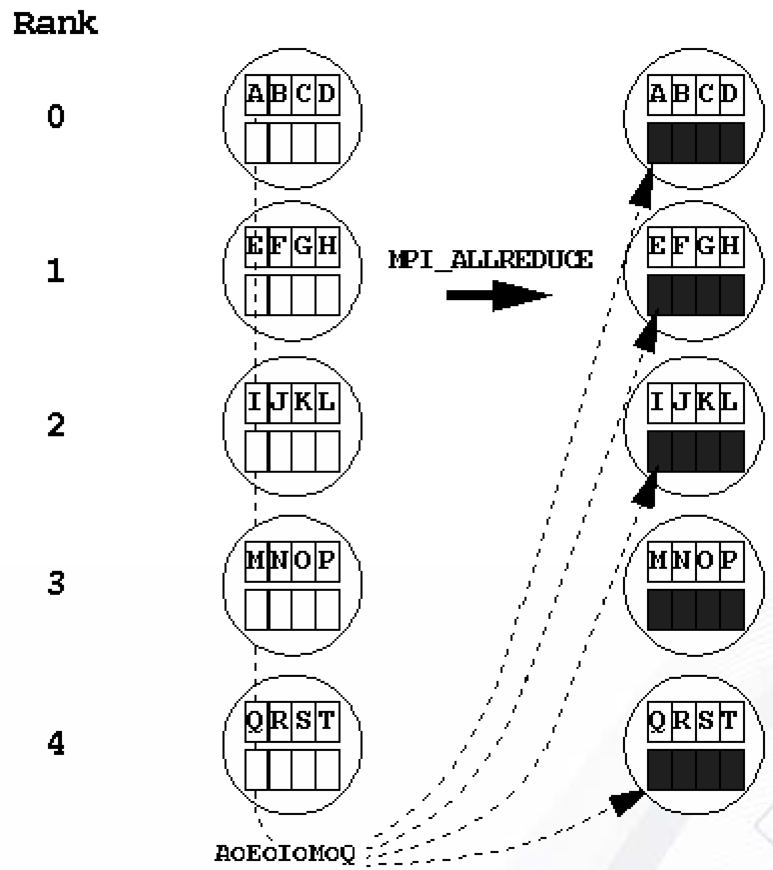
```
PROGRAM UserOP
  INCLUDE 'mpif.h'
  INTEGER err, rank, size
  integer source, reslt
  external digit
  logical commute
  integer myop
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
  CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
  commute=.true.
  call MPI_OP_CREATE(digit,commute,myop,err)
  source=(rank+1)**2
  call MPI_BARRIER(MPI_COM_WORLD,err)
  call MPI_SCAN(source,reslt,1,MPI_INTEGER,myop,MPI_COMM_WORLD,err)
  print *,"P:",rank," my result is ",reslt
  CALL MPI_FINALIZE(err)
END
integer function digit(in,inout,len,type)
  integer in(len),inout(len)
  integer len,type
  do i=1,len
      inout(i)=mod((in(i)+inout(i)),10)
  end do
  digit = 5
end
```

```
 Program Output
P:6 my result is 0
P:5 my result is 1
P:7 my result is 4
P:1 my result is 5
P:3 my result is 0
P:2 my result is 4
P:4 my result is 5
P:0 my result is 1
```

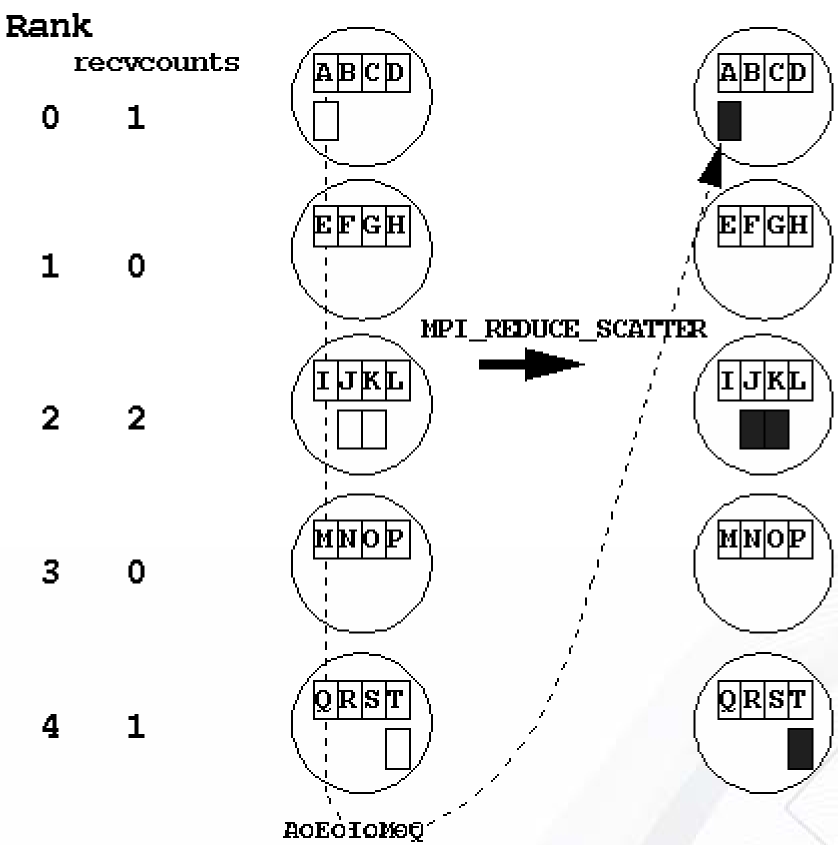
Variants of MPI_REDUCE

- MPI_ALLREDUCE - no root process (all get results)
- MPI_REDUCE_SCATTER - multiple results are scattered
- MPI_SCAN - “parallel prefix”

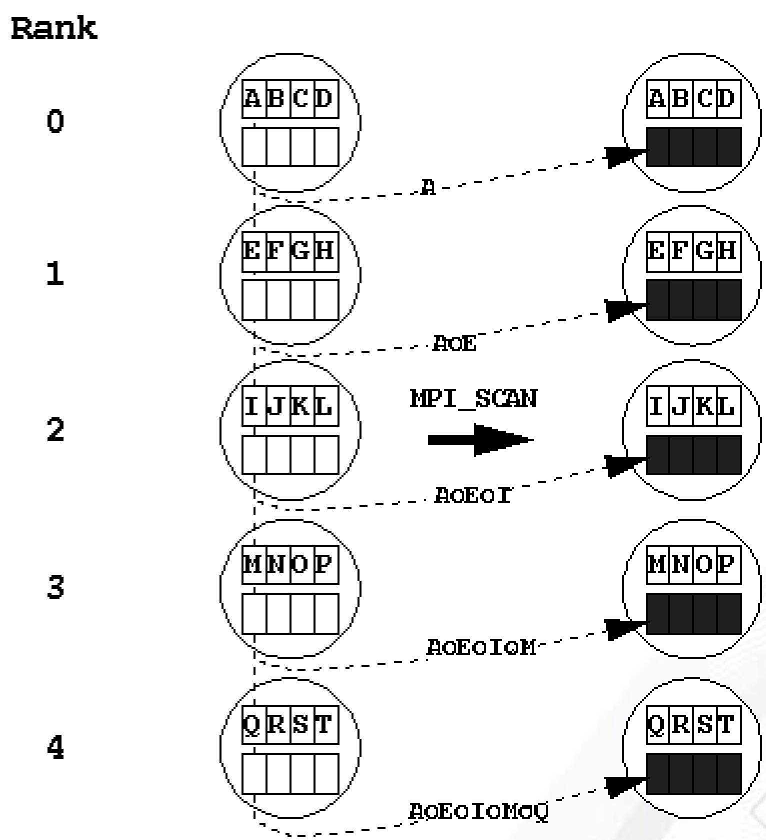
MPI_ALLREDUCE



MPI_REDUCE_SCATTER



MPI_SCAN



Rank
0
1
2
3
4
A B C D
E F G H
I J K L
M N O P
Q R S T
MPI_SCAN
A
AoE
AoEoI
AoEoIoM
AoEoIoMoQ

Class Exercise: Collective Ring

- Rewrite the "Structured Ring" program to use MPI global reduction to perform its global sums
- Extra credit: Rewrite it so that each process computes a partial sum
- Extra extra credit: Rewrite this so that each process prints out its partial result in rank order