# Parallel Computing for Science & Engineering
# Introduction to MPI, Simple Collectives
# Spring 2018

Instructors:
Charlie Dey, TACC
Lars Koesterke, TACC

# A little more about Buffers

**In C/C++**

General principle: buffer argument is address in memory of the data.

- Buffer is void pointer:
- write `&x` or `(void*)&x` for scalars
- write `x` or `(void*)x` for an array

# A little more about Buffers

**In Fortran**

General principle: buffer argument is address in memory of the data.

- Fortran always passes by reference:
- write `x` for a scalar
- write `x` for an array

# Experiment #2

Using your Ping-Pong program from Tuesday:

set up 8 tasks, and 8 integers: rank0 to rank7
initialize the integers to -1
each sending task will send it's rank to the receiving tasks corresponding integer variable

Have each task send a message back and forth to all the other tasks

Example, we have 8 tasks acquired,

task 0 will send and receive a message to tasks 1, 2, 3, 4, 5, 6, 7
task 1 will send and receive a message to tasks 2, 3, 4, 5, 6, 7, 0
task 2 will send and receive a message to tasks 3, 5, 6, 7, 8, 0, 1
…
…
…
…

print out the set of variable after each task has sent the message to all of its receiving tasks (hopefully you see the pattern)

# Experiment #3

Using your Ping-Pong program from Tuesday:

set up 8 tasks, and 8 integers: rank0 to rank7
initialize the integers to -1
each sending task will send it's rank to the receiving tasks corresponding integer variable

Have each task send a message back and forth to all the other tasks
Do this with non-blocking

Example, we have 8 tasks acquired,

task 0 will send and receive a message to tasks 1, 2, 3, 4, 5, 6, 7
task 1 will send and receive a message to tasks 2, 3, 4, 5, 6, 7, 0
task 2 will send and receive a message to tasks 3, 5, 6, 7, 8, 0, 1
…
…
…
…

print out the set of variable after each task has sent the message to all of its receiving tasks
(hopefully you see the pattern)

# Collectives

Gathering and spreading information:

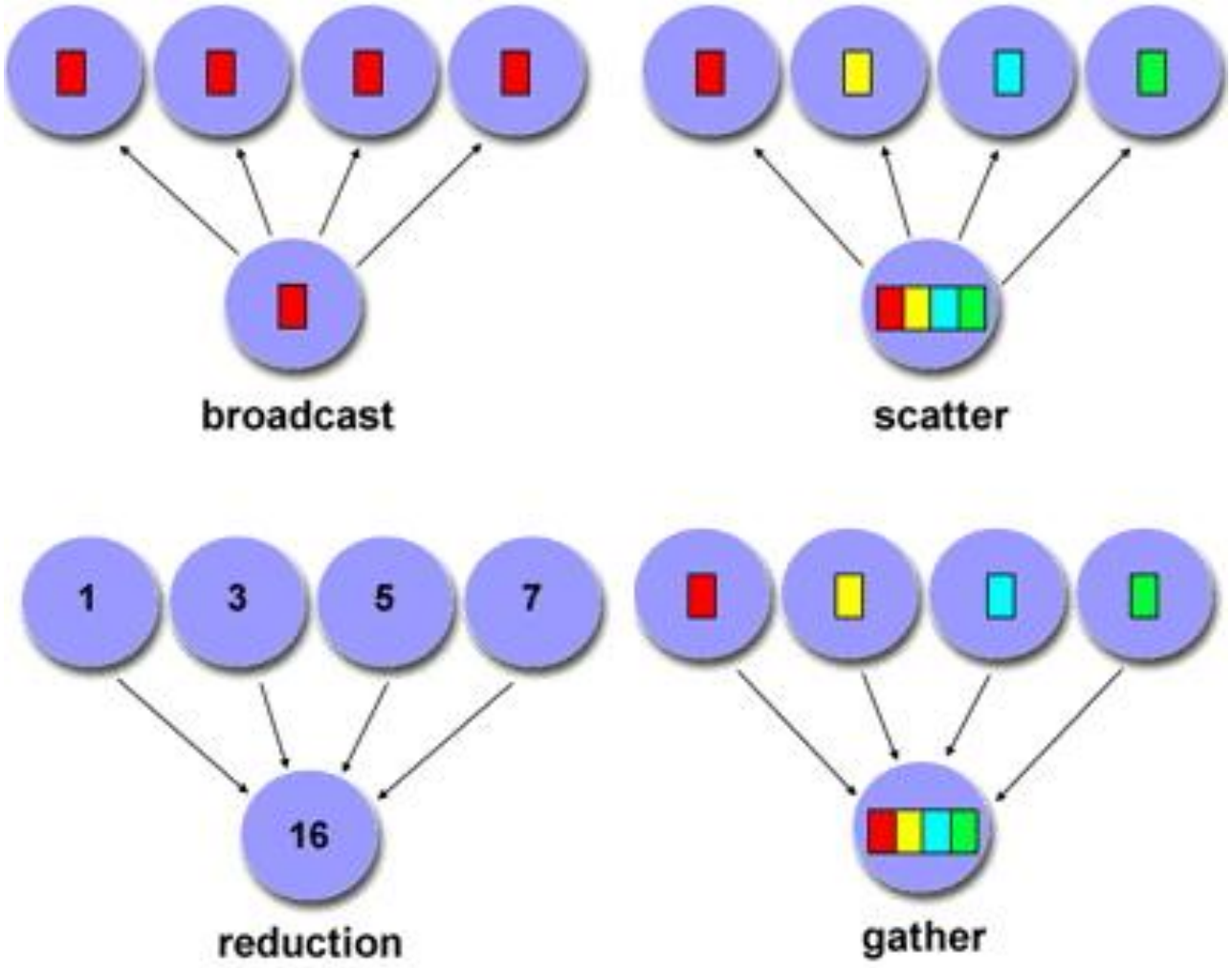
- Every process has data, you want to bring it together;
- One process has data, you want to spread it around.

**Root process: the one doing the collecting or disseminating.**

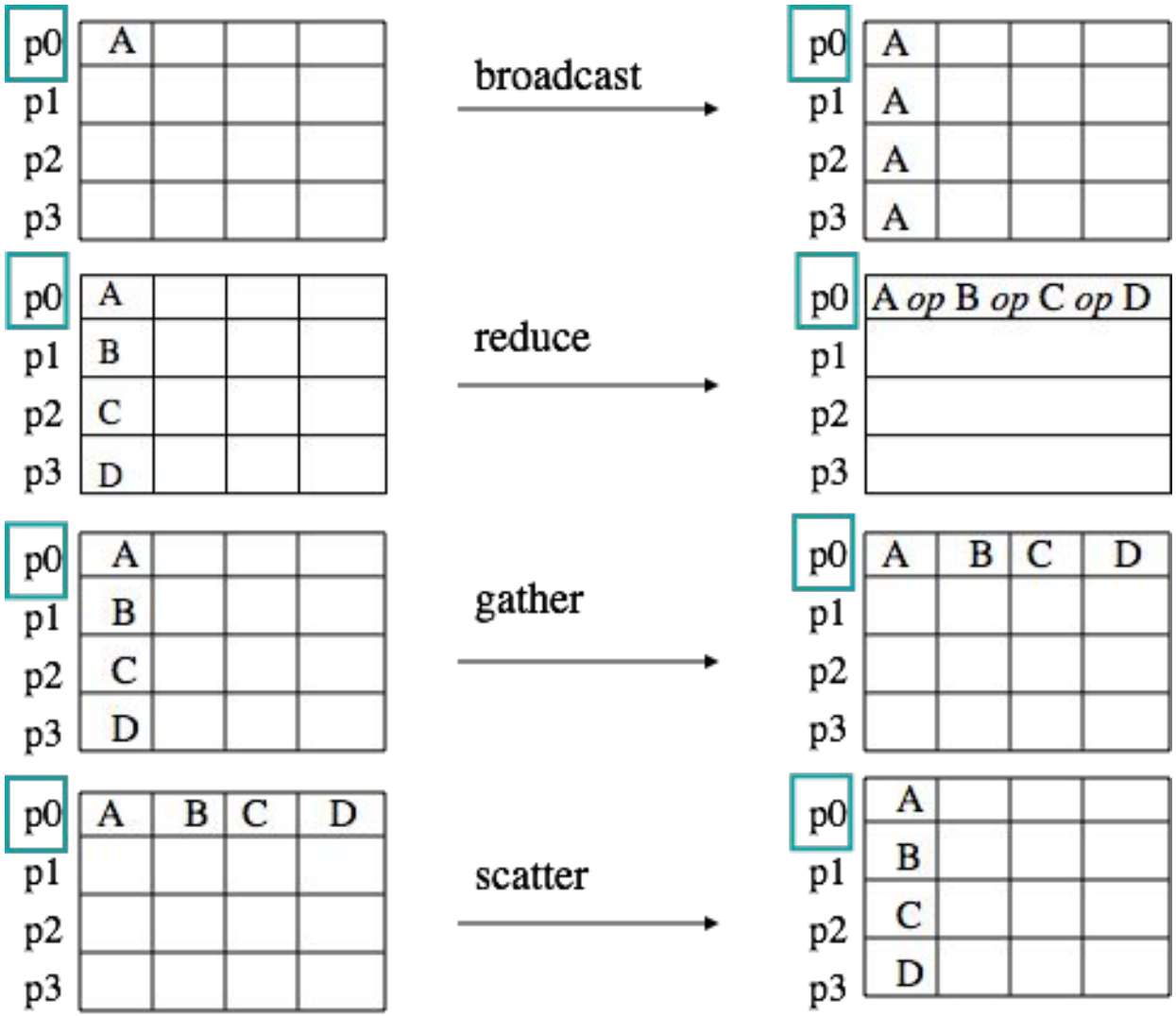
Basic cases:

- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.

# Collectives



# Collectives



# Collectives, Exercise

Given the following scenarios, which MPI collective would work best?

- Let each process compute a random number. You want to print the maximum of these numbers to your screen.
- Each process computes a random number again. Now you want to scale these numbers by their maximum.
- Let each process compute a random number. You want to print on what processor the maximum value is computed.

# Collectives, Broadcast

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

**C:**
```
ierr=MPI_Bcast(&dat[0], cnt, datatype,  root, comm);
```

**Fortran:**
```
call MPI_Bcast( dat, cnt, datatype,  root, comm, ierr)
```

- `root` is the rank of the process doing the broadcast
- Each process allocates buffer space;
  - root explicitly fills in values,
  - all others receive values through broadcast call.
- Datatype is `MPI_FLOAT`, `MPI_INT`, etc. (different between C/Fortran).
- comm is usually `MPI_COMM_WORLD`

# Collectives, Reduction

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

**C:**
```
ierr=MPI_Reduce(&sbuf[0], &rbuf[0], count,
    datatype, operator, root, comm)
```

**Fortran:**
```
call MPI_Reduce( sbuf, rbuf, count,
          datatype, operator, root, comm, ierr)
```

- recvbuf is ignored on non-root processes
- MPI_Op is `MPI_SUM`, `MPI_MAX`, etc.

# Collectives, Reduction

| | |
|---|---|
| `MPI_PROD` | Product |
| `MPI_SUM` | Sum |
| `MPI_LAND` | Logical `and` |
| `MPI_LOR` | Logical `or` |
| `MPI_LXOR` | Logical `exclusive or` |
| `MPI_BAND` | Bitwise `and` |
| `MPI_BOR` | Bitwise `or` |
| `MPI_BXOR` | Bitwise `exclusive or` |
| `MPI_MAX` | Maximum |
| `MPI_MIN` | Minimum |
| `MPI_MAXLOC` | Maximum value and location |
| `MPI_MINLOC` | Minimum value and location |

# Collectives, Gather/Scatter

```
int MPI_Gather(
  void *sendbuf, int sendcnt, MPI_Datatype sendtype,
  void *recvbuf, int recvcnt, MPI_Datatype recvtype,
  int root, MPI_Comm comm
);

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

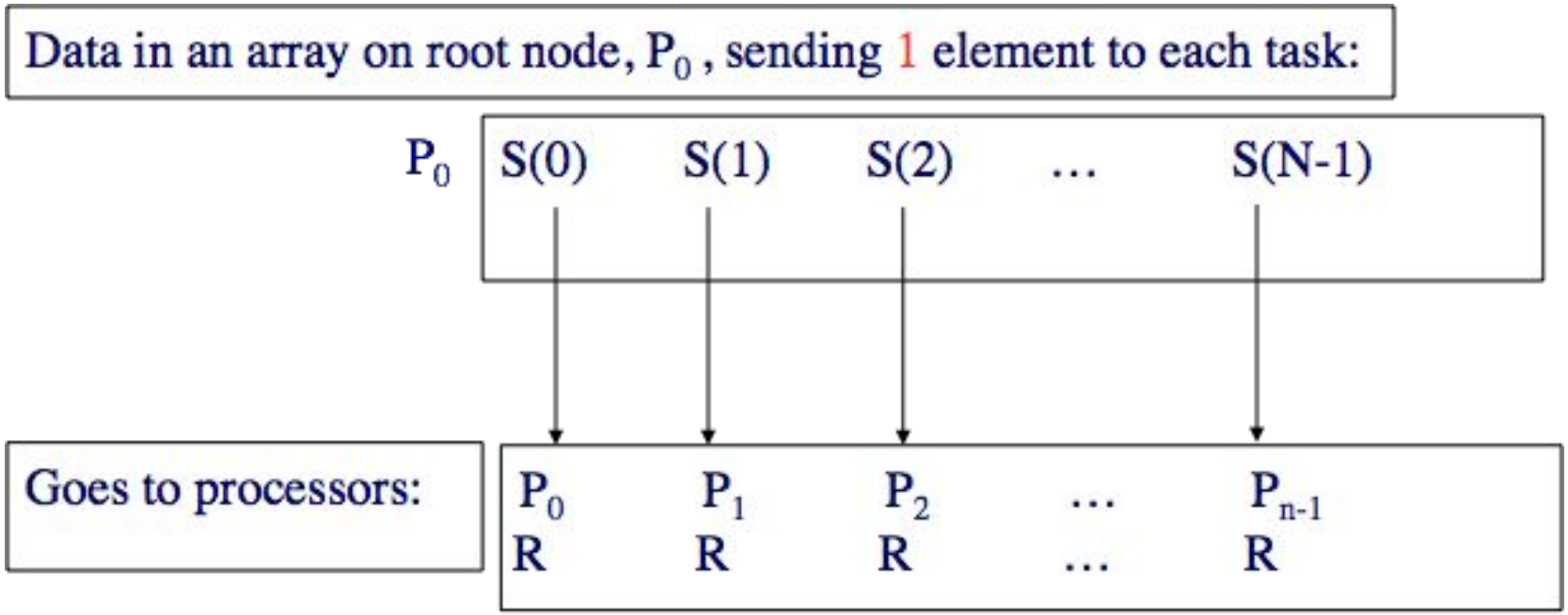
- Scatter: the `sendcount` / Gather: the `recvcount`
  - this is not the total length of the buffer
  - it is the amount of data to/from each process.

# Collectives, Gather/Scatter

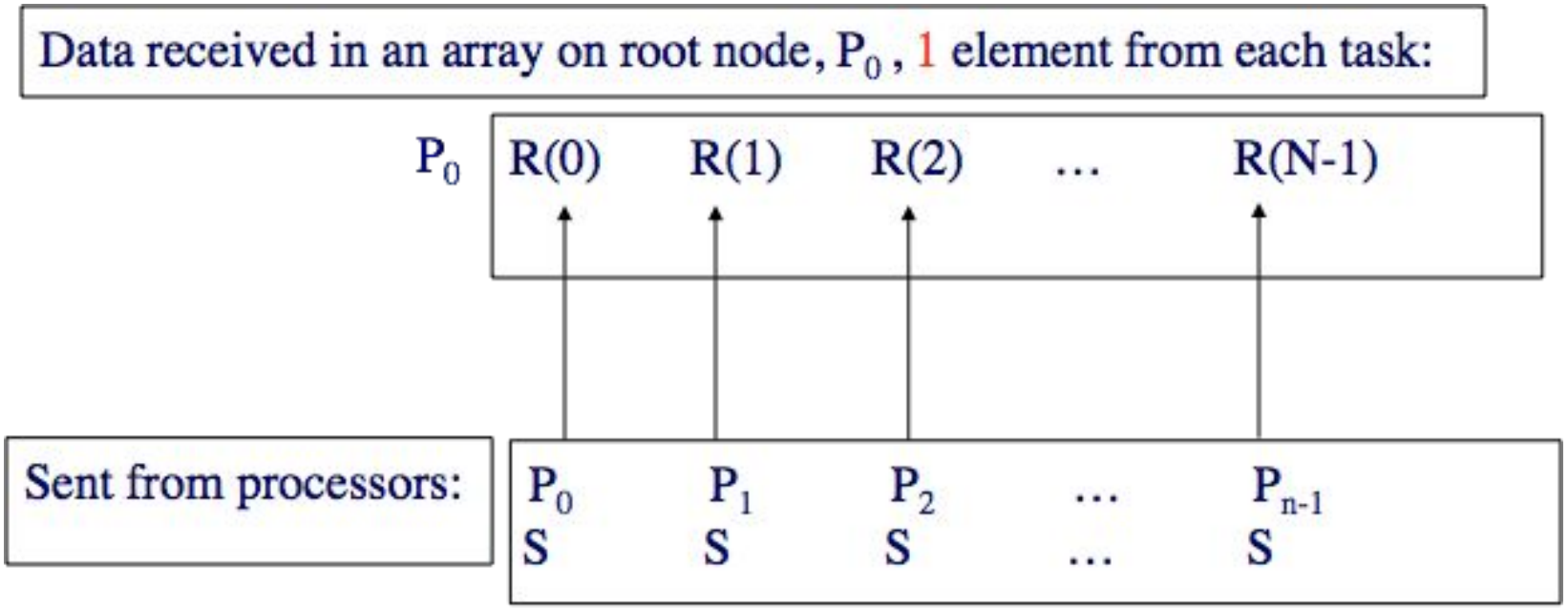
```
int MPI_Gather(
  void *sendbuf, int sendcnt, MPI_Datatype sendtype,
  void *recvbuf, int recvcnt, MPI_Datatype recvtype,
  int root, MPI_Comm comm
);

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

- Scatter: the `sendcount` / Gather: the `recvcount`
  - this is not the total length of the buffer
  - it is the amount of data to/from each process.

# Collectives, Scatter

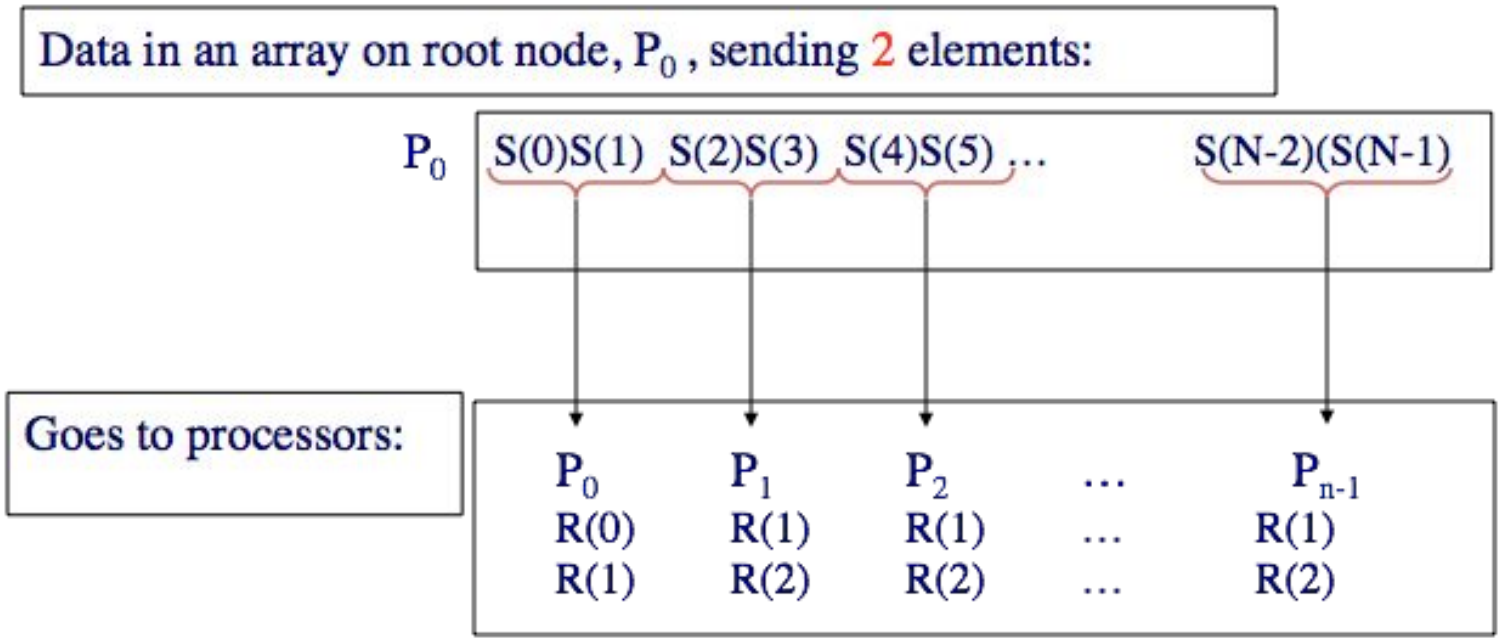


# Collectives, Gather



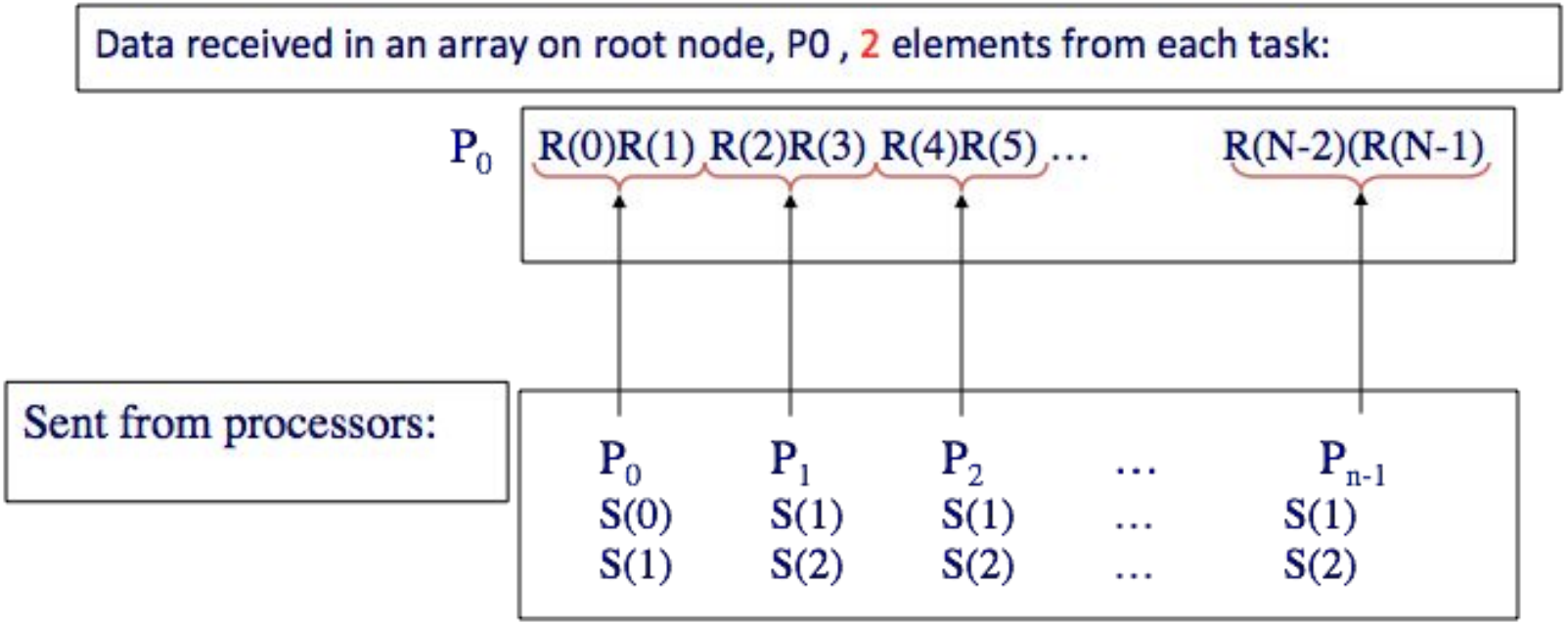
# Collectives, Scatter

## 2-elements per processor



# Collectives, Gather

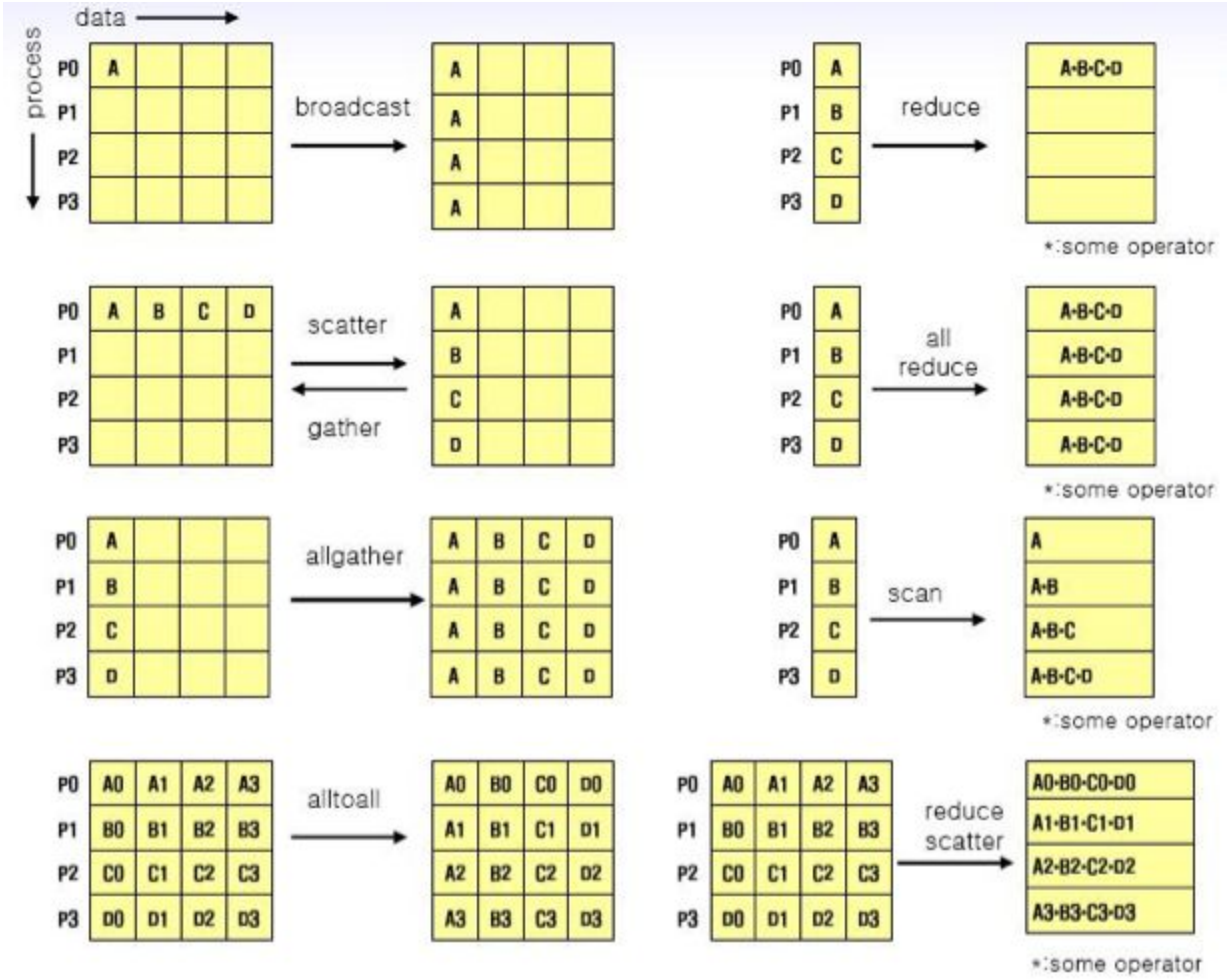
## 2-element version



# Collectives, Communication

- Every process MUST call the routine
  - All calls are blocking
  - A task may return when participation is complete
  - May or may not synchronize (implementation dependent)
- Must have “matching” arguments
  - no status
  - no tags
- Send and Receive sizes must match
  - mapping may vary
- Basic calls have a root—”all” versions don’t

# Collectives, Summary



# Collectives, Random Numbers

C:

```
// Initialize the random number generator
srand((int)(mytid*(double)RAND_MAX/ntids));
// compute a random number
randomfraction = (rand() / double)RAND_MAX);
```

Fortran:

```
integer :: randsize
integer,allocatable,dimension(:) :: randseed

real :: random_value
call random_seed(size=randsize)
allocate(randseed(randsize))
do i=1,randsize
   randseed(i) = 1023*mytid
end do
```

# In Class Lab #1

Write a short program where each process compute a random number. Printout which task computed the maximum value, and what that maximum value is.

test it out by having each task print out it's calculated value, and the root task print out the max value reported back

What collective would you use?