Parallel Programming with MPI and OpenMP

Tuesday

- Morning – MPI
  - Review of programming exercises
  - MPI – point-to-point communication (send and receive modes)
  - MPI – collective communication, Communicator splitting
  - MPI – Reduction operations
  - *Exercise:* Communicate around a ring
- Afternoon
  - Review of programming exercise
  - MPI – non-blocking communication
  - Laplace equation II: Implementation with MPI
  - Thinking Parallel II: Performance considerations

Review from yesterday

Fundamentals

- MPI is a library; link at compile time
- Always include header file or Fortran module
- In C, MPI defines some typedefs (MPI_Status, MPI_Comm, MPI_Request, MPI_Datatype). In Fortran these are Integers
- MPI has its own data types to be specified in communication routines
- MPI functions return a success-value (check with MPI_SUCCESS). Status of recv is not about success

Review from yesterday (II)

Basic functions

- `MPI_Init, MPI_Finalize,`
- `MPI_Comm_size, MPI_Comm_rank`
- `MPI_Abort`

These functions are sufficient to write functional parallel programs

Messages

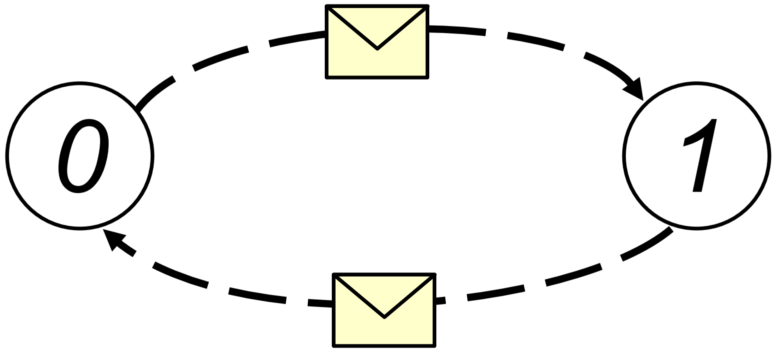
- Involves a pair of processes
- Is performed within a communicator
- `MPI_Send,  MPI_Recv`

Timers

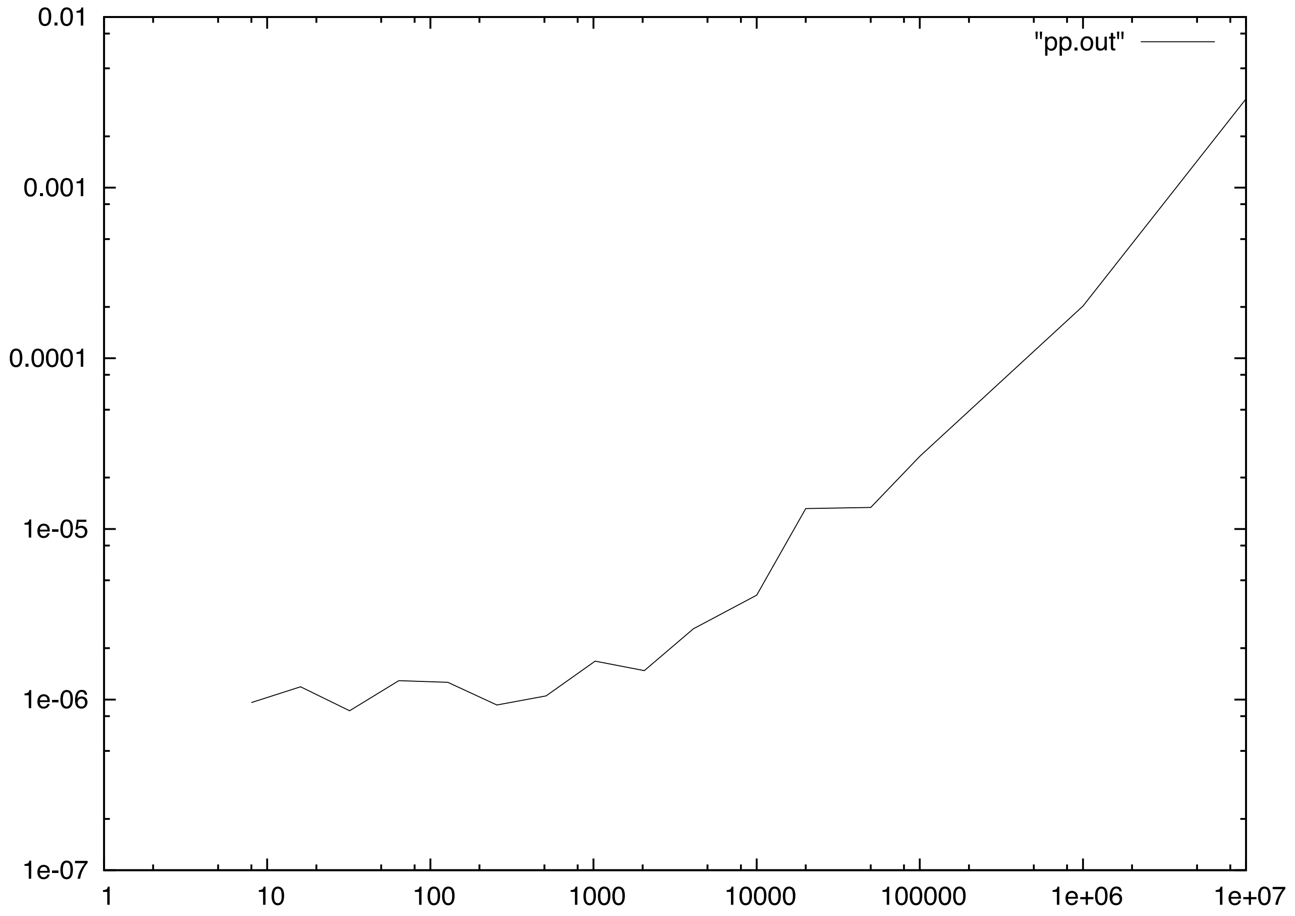
- `MPI_Wtime,   MPI_Wtick`

Exercise 1: Ping pong

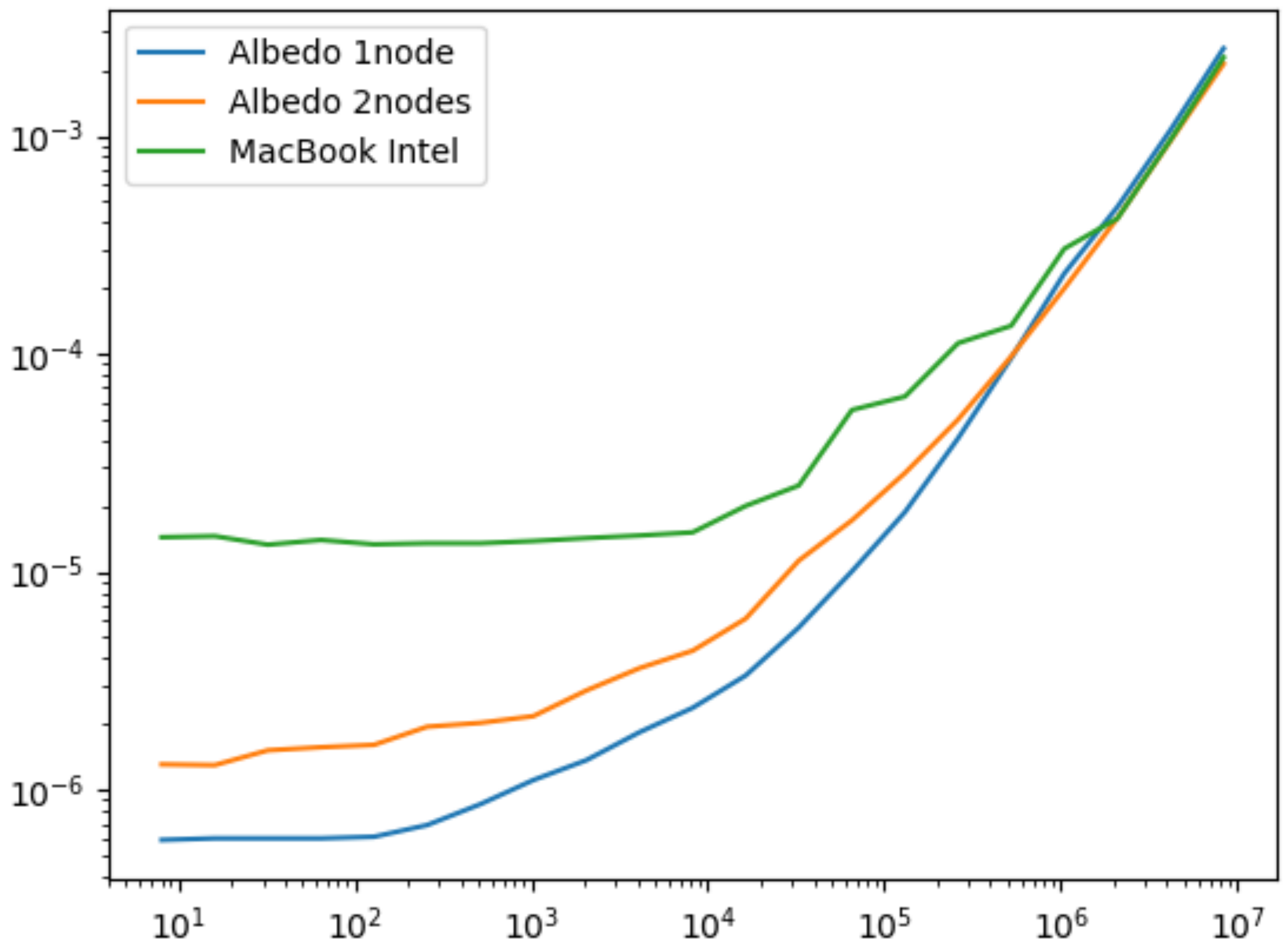
- Write a program in which two processes repeatedly pass a message back and forth.
- Insert timing calls to measure the time taken for one message (send+receive).
- Investigate how the time taken varies with the size of the message



Ping pong: result from Notebook



Ping pong: Notebook vs. HPC

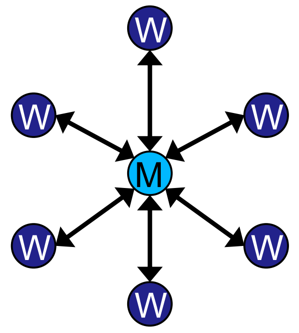


Example: Master-Worker Program

- 2 Sets of processes:
  - Master (rank=0) distributes work and collects results
  - Workers (rank>0) get work task from Master, process it and return results.

- Allows for some load balancing if pieces of work > number of workers

- Communication scheme
  - Point-to-point communication between master and all workers
  - Only MPI_Send and MPI_Recv required



Master-Worker: Communication protocol

- Master:
  - Wait for request from Worker
  - Send task to Worker
  - If all tasks completed: When flag for completion

- Worker:
  - Send request for work to Master (empty message)
  - Receive answer from Master
  - End if completion flag

Example: Master-Worker Program

- Control program:

```
int main(int argc, char *argv[])
{
    int maxtask = 40;
    int rank;

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

    if (rank == 0)
        master(maxtask);
    else
        worker();

    MPI_Finalize();
    return 0;
}
```

Example: Master-Worker Program

- Master:

```
void master(const int maxtask)
{
  MPI_Status status;
  int size, msg, task, dest;

  MPI_Comm_size(MPI_COMM_WORLD, &size);

  for (task = 1; task <= maxtask; task++)
  {
    MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    MPI_Send(&task, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
  }

  task = -1;
  for (dest = 1; dest < size; dest++)
  {
    MPI_Recv(&msg, 1, MPI_INT, dest, 0, MPI_COMM_WORLD, &status);
    MPI_Send(&task, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
  }
}
```

Example: Master-Worker Program

- Worker:

```
void worker(void)
{
  MPI_Status status;
  int rank, msg, task;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  do
  {
    MPI_Send(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&task, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

    if (task >= 0)
    {
      printf("rank %d: working on task %d\n", rank, task);
      system("sleep 1");
    }
  }
  while (task >= 0);
}
```

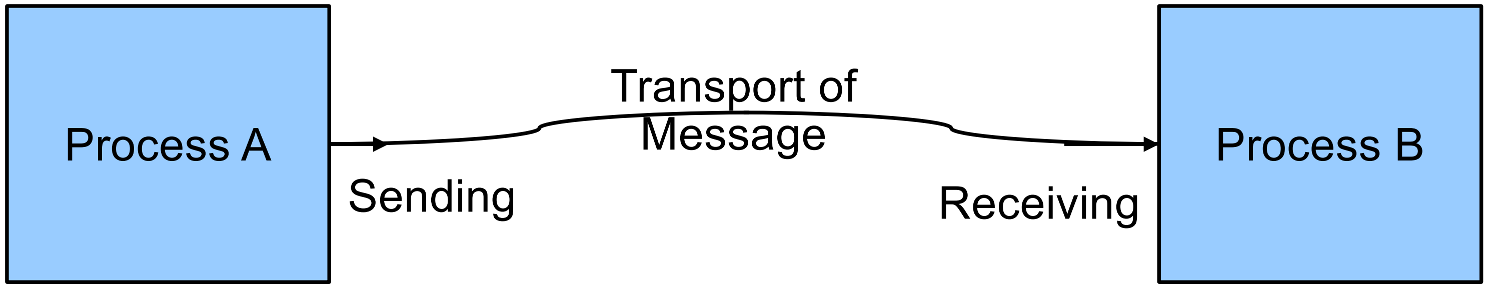
Point-to-Point Communication

MPI: “Large” and “Small”

- MPI is *“Large”*
  - MPI 1.2 has 128 functions
  - MPI 2.0 has 152 functions
  - MPI 3.0 has ~300 functions (MPI 4 and 5 added more)
- MPI is *“Small”*
  - Many programs need to use only about 6 MPI functions.
- MPI is the *“Right Size”*
  - Offers enough flexibility so user's don't need to master >300 functions to use it properly.

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another

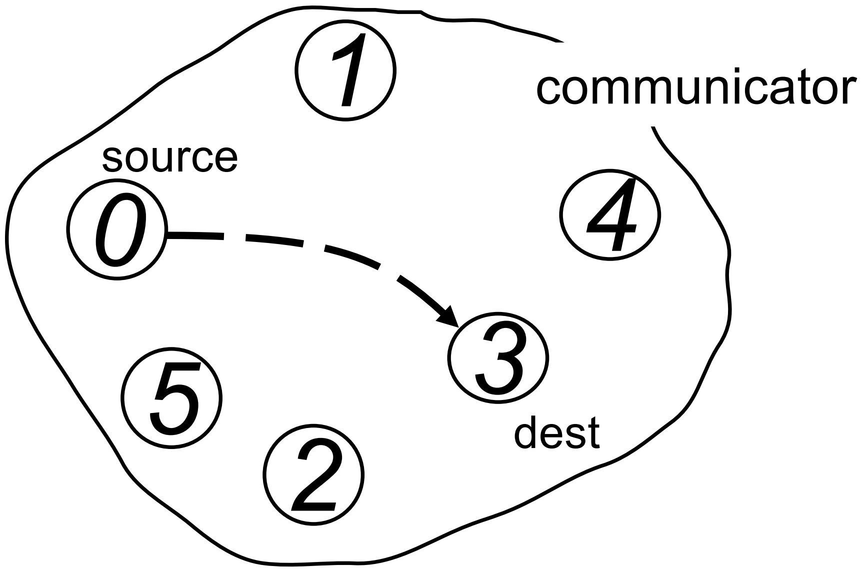


- Different types of point-to-point communication

Point-to-Point Communication

- Basic features:
  - In basic MPI (MPI-1) only “two-sided” communications are allowed, requiring an explicit **send** and **receive**.
  - Point-to-point (P2P) communication is explicitly two-sided, and the message will not be transferred without the **active** participation of both processes.
  - A *message* generically consists of a *body* (data being transferred) and an *envelope* (tags indicating, e.g., source and destination)
  - Fundamental – almost all of the MPI communications are built around P2P operations.

Point-to-Point Communication



- Communication takes place within a communicator.
- Source process identified by its rank in the communicator
- Destination process is identified by its rank in the communicator.
- Message is identified by a 'tag’
  (Just set it to 0, unless you know that you need a different value)

Standard mode: Synchronous Blocking Message-Passing

- Processes synchronize (handshake).
- Blocking - both processes wait until the transaction has completed.
  - Send-buffer can be re-used upon completion

Requirements for successful communication

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message types must match.
- Receiver's buffer must be large enough.

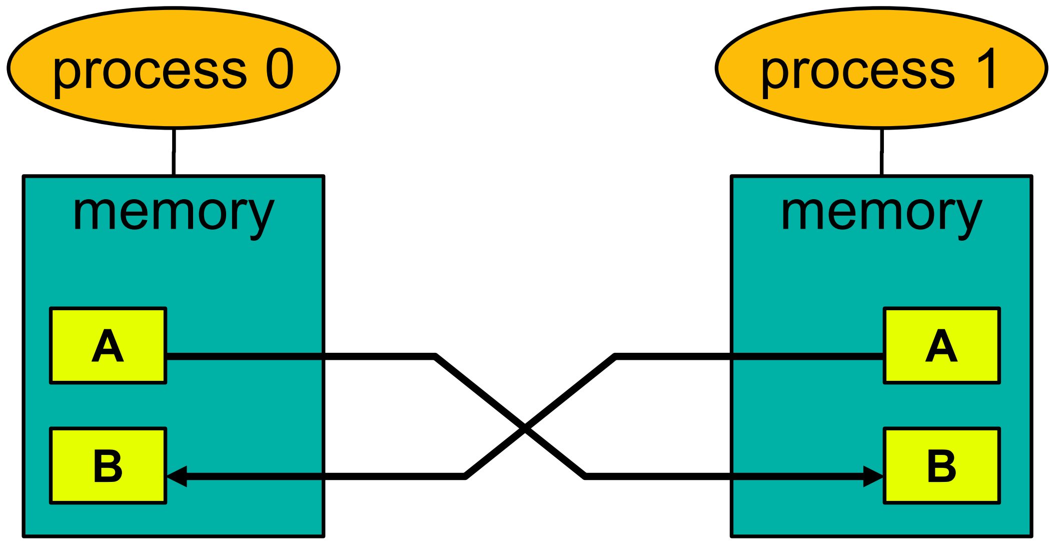
Wildcarding

- Receiver can wildcard
- To receive from any source: `MPI_ANY_SOURCE`
- To receive with any tag: `MPI_ANY_TAG`
- Actual source, tag, and amount of data are returned in the receiver's *status* parameter.

- Can make communication more efficient
- Use with care! More prone to produce errors.

Exchanging Data (I)

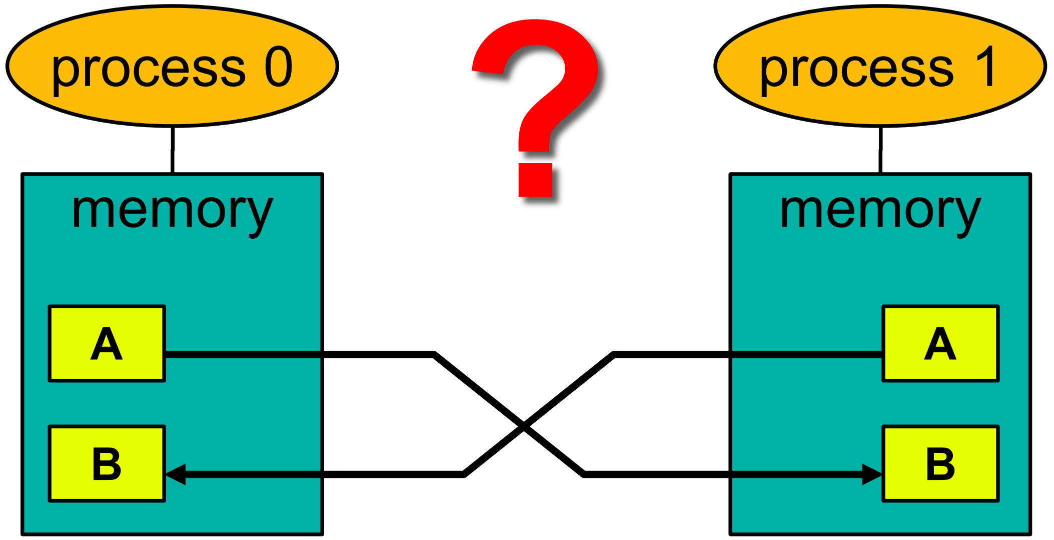
- Example with two processes: 0 and 1
- General data exchange is very similar



```
MPI_Send(A, ...)           MPI_Send(A, ...)
MPI_Recv(B, ...)           MPI_Recv(B, ...)
```

Exchanging Data (II)

- Example with two processes: 0 and 1
- General data exchange is very similar



```
MPI_Send(A, ...)          MPI_Send(A, ...)
MPI_Recv(B, ...)          MPI_Recv(B, ...)
```

Why is this is wrong?

Exchanging Data (III)

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
```

Process 0:

```
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &status)
```

Process 1:

```
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &status)
```

The problem:

- `MPI_Send` is a non-local operation – may not complete until a matching receive is posted.
- Both processes may block in `MPI_Send`, waiting for a corresponding receive. This is called **deadlock**.

One Solution to Deadlock

Switch the order of Send/Recv on one process.

Process 0:

```
  MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
  MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &status)
```

Process 1:

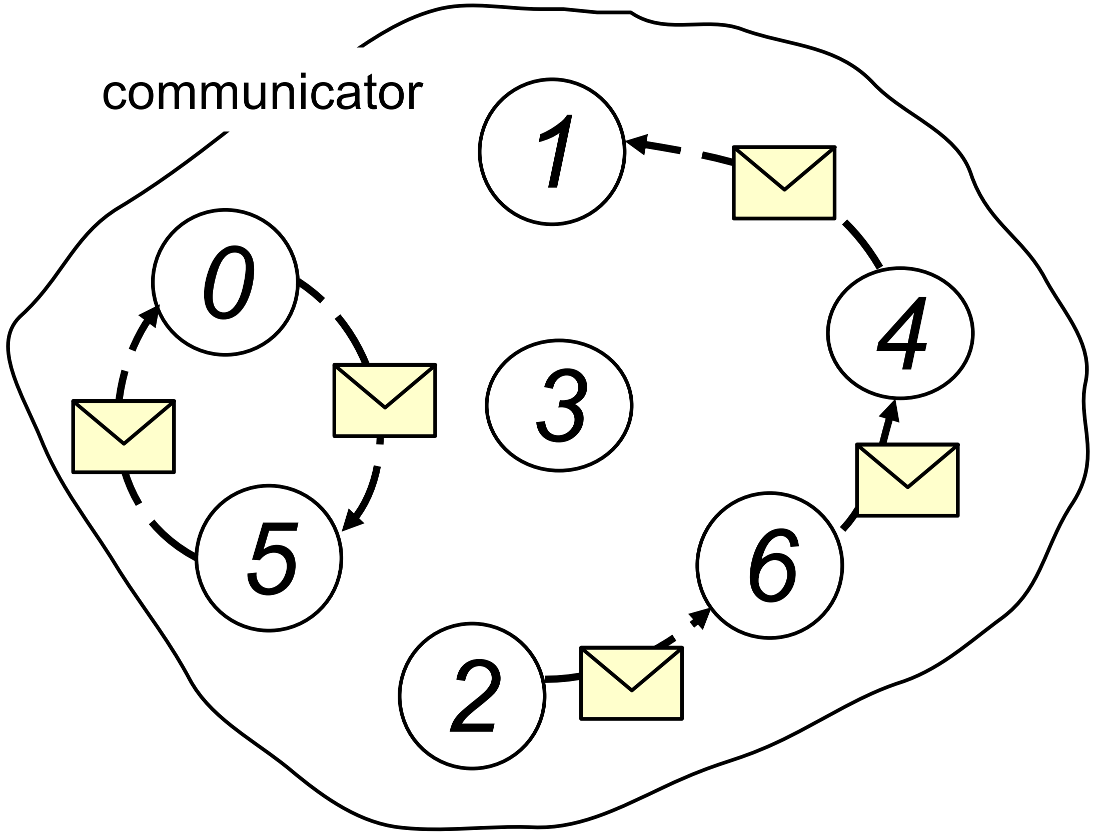
```
  MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &status)
  MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
```

This seems to solve the problem, but:

- It works only in simple cases.
- Even in this case, does not allow use of bi-directional hardware.
- In more complicated cases, it may avoid deadlock but serialize communication.
- Makes SPMD code non-symmetric and harder to follow.

Recommendation: **Use another approach**

Combined Send and Receive (I)



- Examples where deadlock or serialization is possible

Combined Send and Receive (II)

- C:

```
int MPI_Sendrecv
    (void *sendbuf, int sendcount,
     MPI_Datatype sendtype, int dest, int sendtag,

     void *recvbuf, int recvcount,
     MPI_Datatype recvtype, int source, int recvtag,

     MPI_Comm comm, MPI_Status *status)
```

Combined Send and Receive (III)

- Fortran:

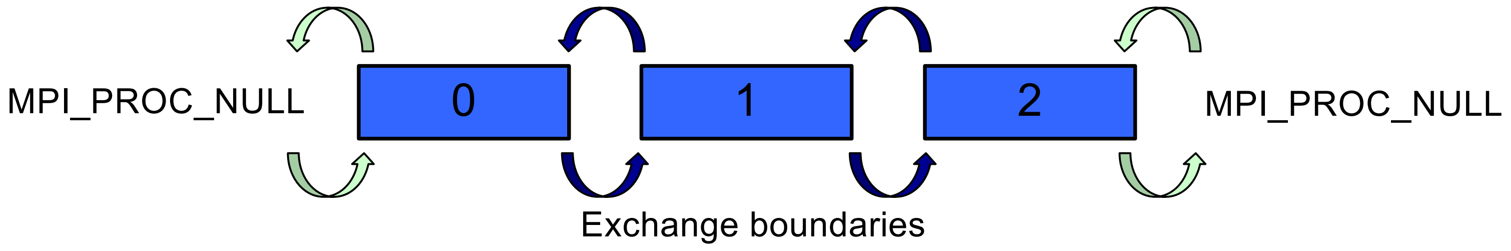
```
CALL MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE,
     DEST, SENDTAG,
     RECVBUF, RECVCOUNT, RECVTYPE,
     SOURCE, RECVTAG,
     COMM, STATUS, IERROR)

<type> SENDBUF(*), RECVBUF(*)

INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG,
     RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM,
     IERROR, STATUS(MPI_STATUS_SIZE)
```

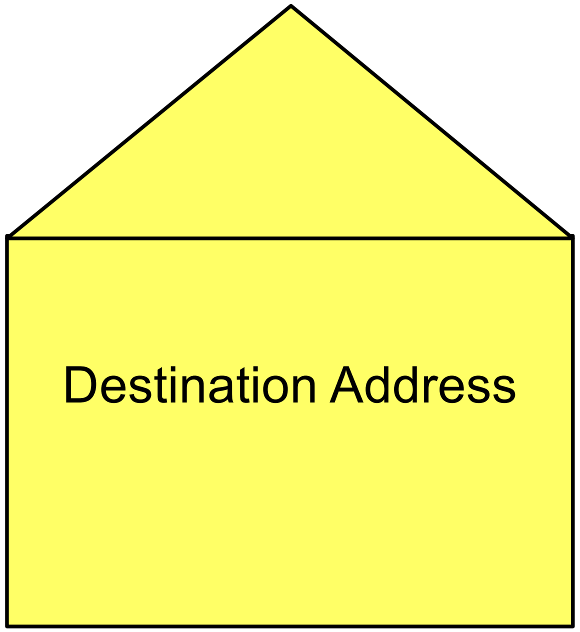
Combined Send and Receive (IV)

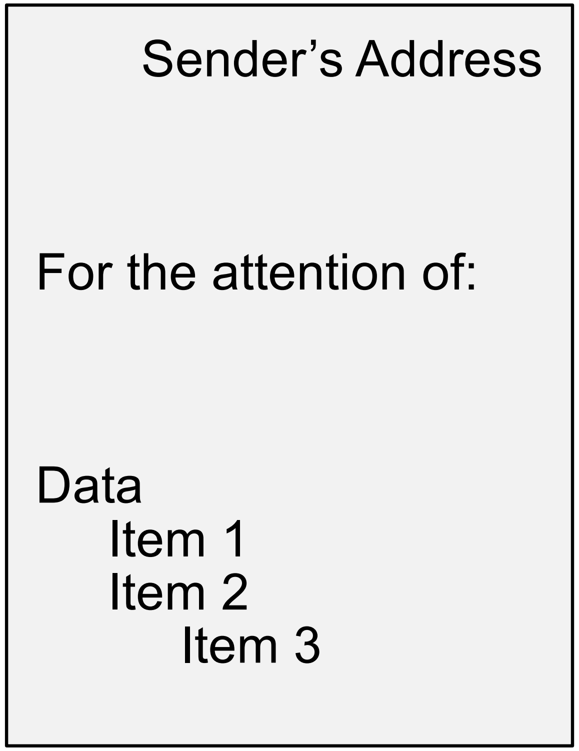
- Extension of point-to-point communication
- Combines send and receive into a single call
- Send and receive happen simultaneously
- Avoids deadlocks
- Blocking
- Can involve 3 processes at a time
- Can involve a non-existing process (`MPI_PROC_NULL`)



Communication Envelope

- Communication include more than data:
  Communication envelope





- Distinguish different messages

Communication Envelope Information

- Envelope information is returned from `MPI_Recv` as `status`
- Information includes:
  - Source: `status.MPI_SOURCE` or `status(MPI_SOURCE)`
  - Tag: `status.MPI_TAG` or `status(MPI_TAG)`
  - Count: `MPI_Get_count` or `MPI_GET_COUNT`
  - also: Destination, Communicator

Received Message Count

- `'count'` in `MPI_Recv` is size of receive buffer, not amount of data received

- C:

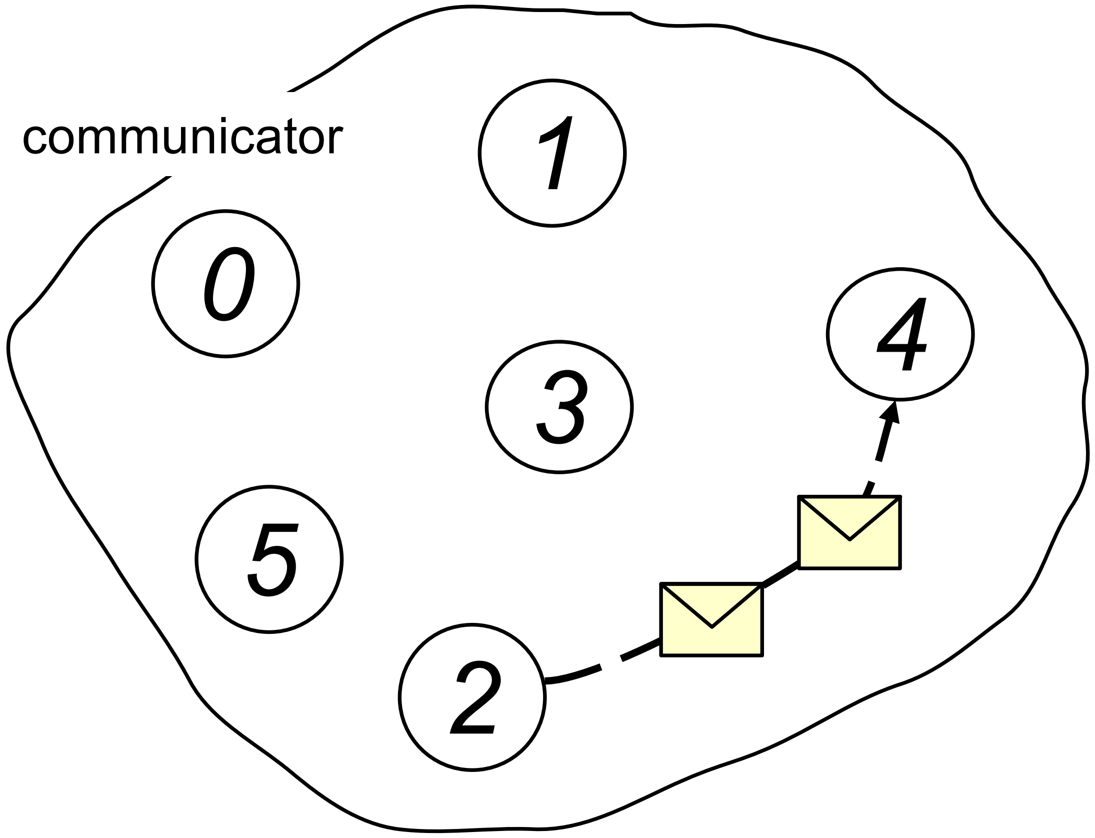
```
int MPI_Get_count (MPI_Status status,
         MPI_Datatype datatype, int *count)
```

- Fortran:

```
CALL MPI_GET_COUNT (STATUS, DATATYPE, COUNT,
         IERROR)

INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE,
         COUNT, IERROR
```

Message Order Preservation



- Messages do not overtake each other.

Communication Modes

Communication modes

- Relate to the receipt of the message
- Specify type of send operation
- Affect reuse of the send buffer

Communication mode	Notes
Synchronous send	Only completes when the receive has completed
Buffered send	Always completes (unless an error occurs), irrespective of receiver
Standard send	Either synchronous or buffered
Ready send	Always completes (unless an error occurs), irrespective of whether the receive has completed
Receive	Completes when a message has arrived

MPI P2P Communication Routines

- One routine for each mode

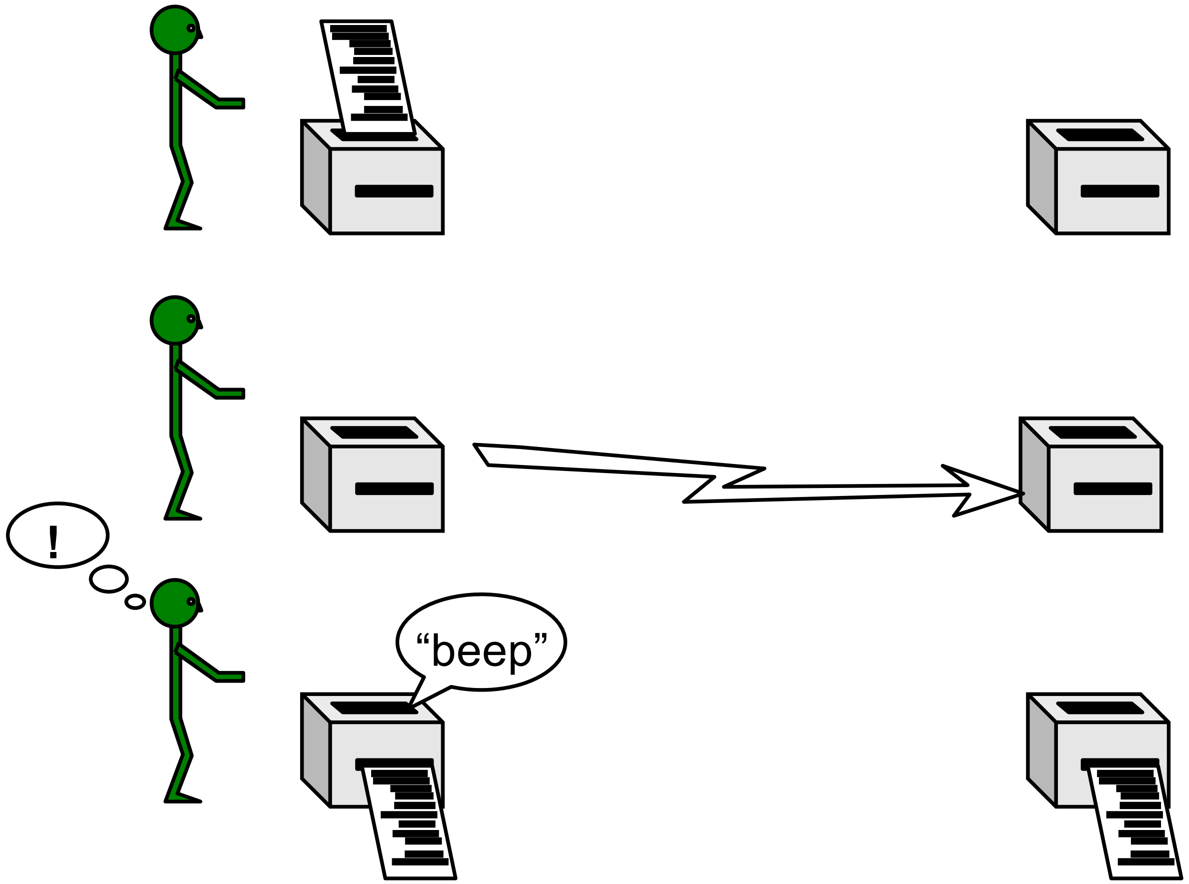
Operation	MPI call
Standard send	MPI_Send
Synchronous send	MPI_Ssend
Buffered send	MPI_Bsend
Ready send	MPI_Rsend

- Only one routine for receiving

Operation	MPI call
Receive	MPI_Recv

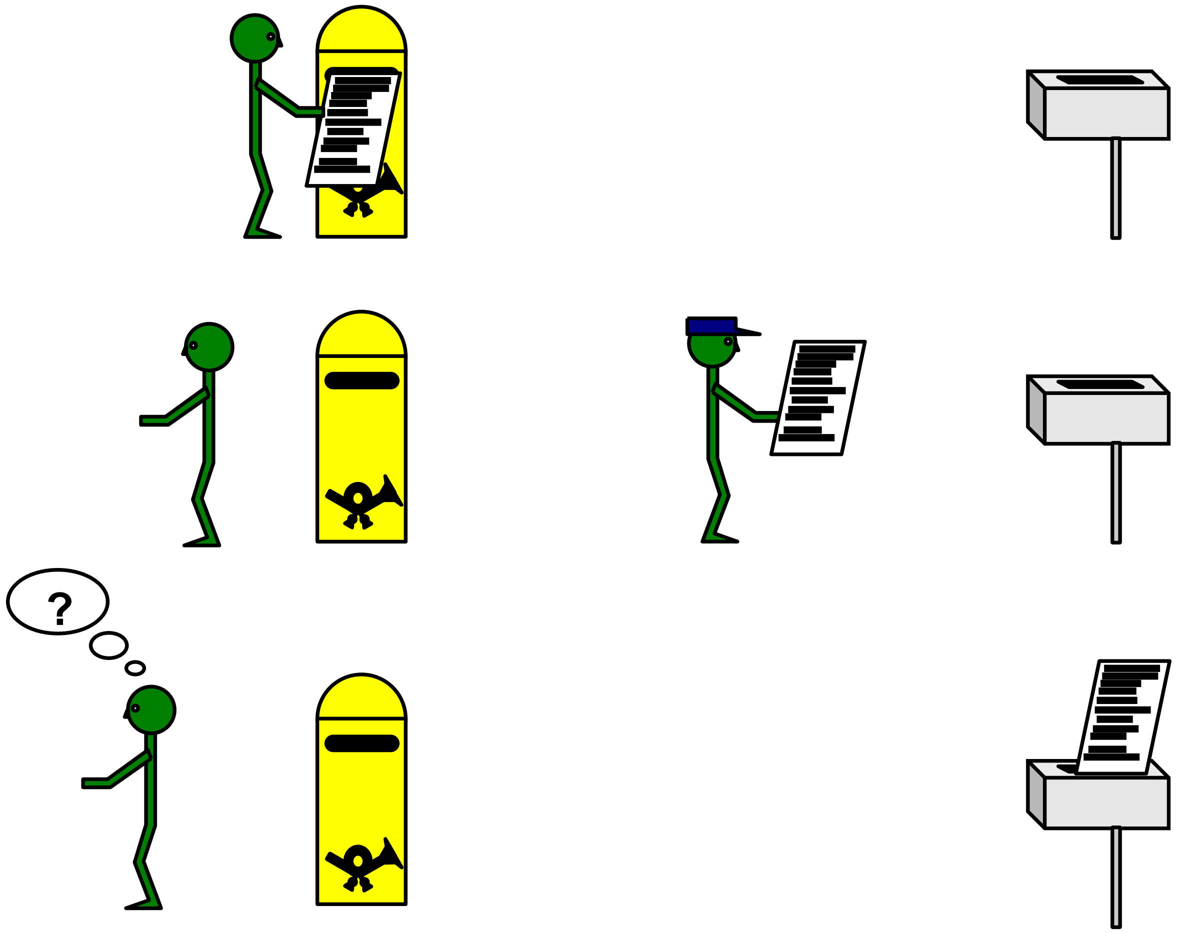
Synchronous Sends

- Provide information about the completion of the message.
  - Message has to be received!



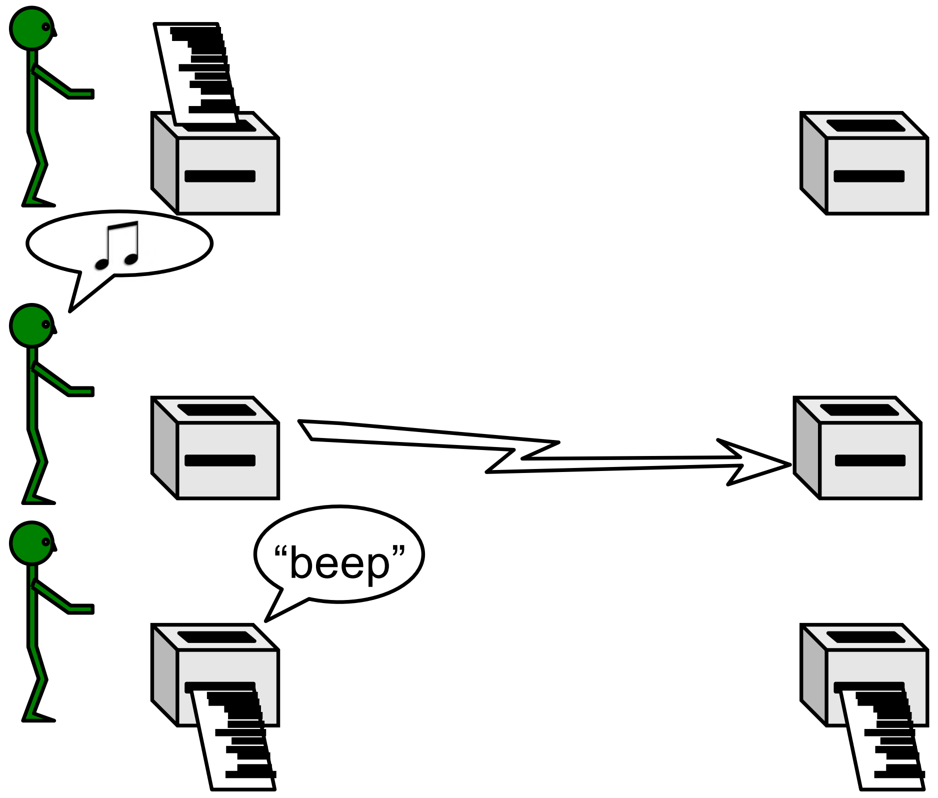
Asynchronous Sends

- Only know when the message has left.
- Don't wait for completion of message.



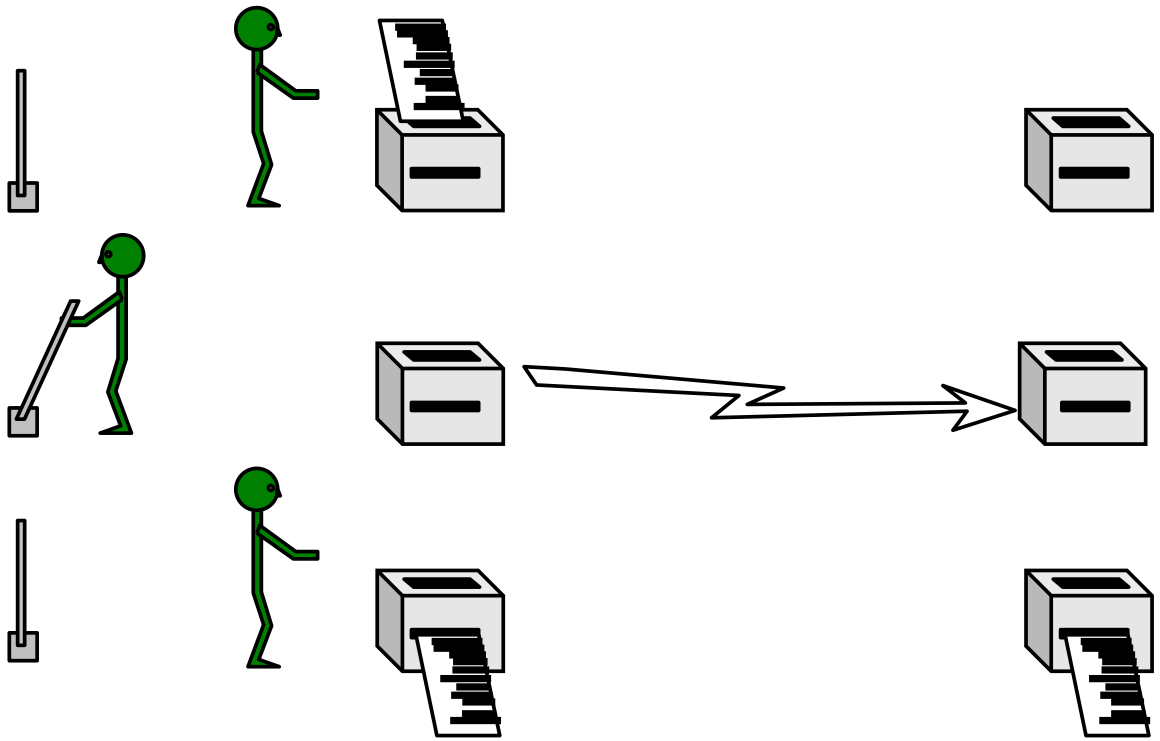
Blocking Operations

- Relate to when the operation has completed.
- Only return from the subroutine call when the operation has completed.



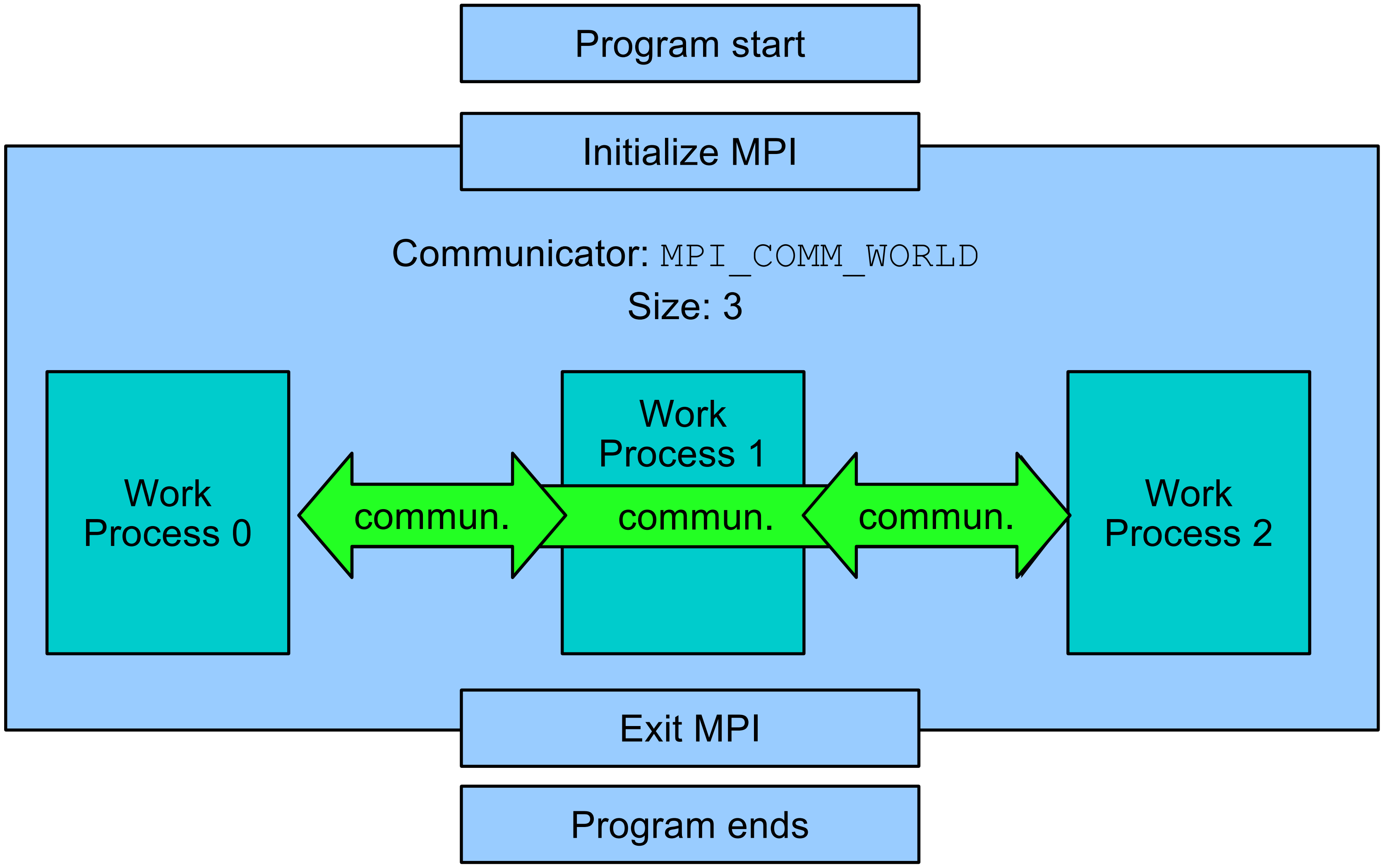
Non-Blocking Operations

- Return straight away and allow the sub-program to continue to perform other work.
- At some later time the sub-program can *test* or *wait* for the completion of the non-blocking operation.



Collective Communications

Collective Communication

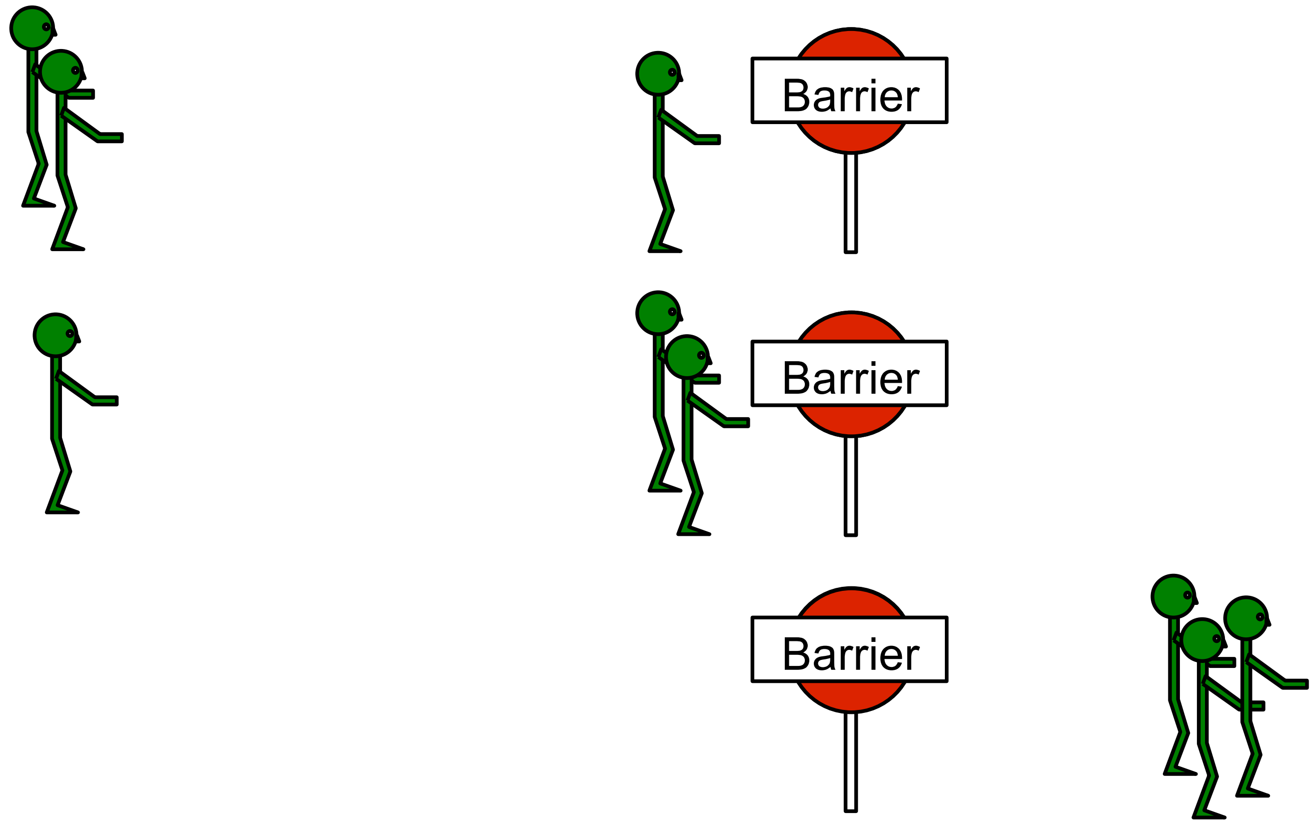


Collective Communications

- Collective communication routines are higher level routines involving several processes at a time.
- Can be built out of point-to-point communications.

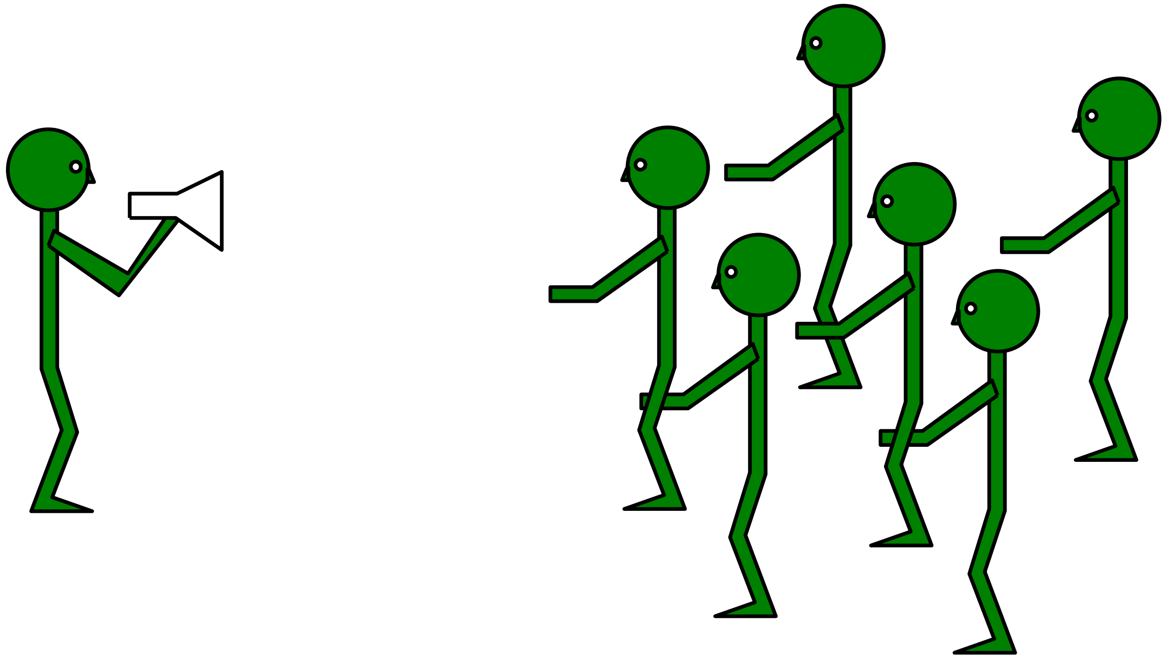
Barriers

- Synchronize processes.



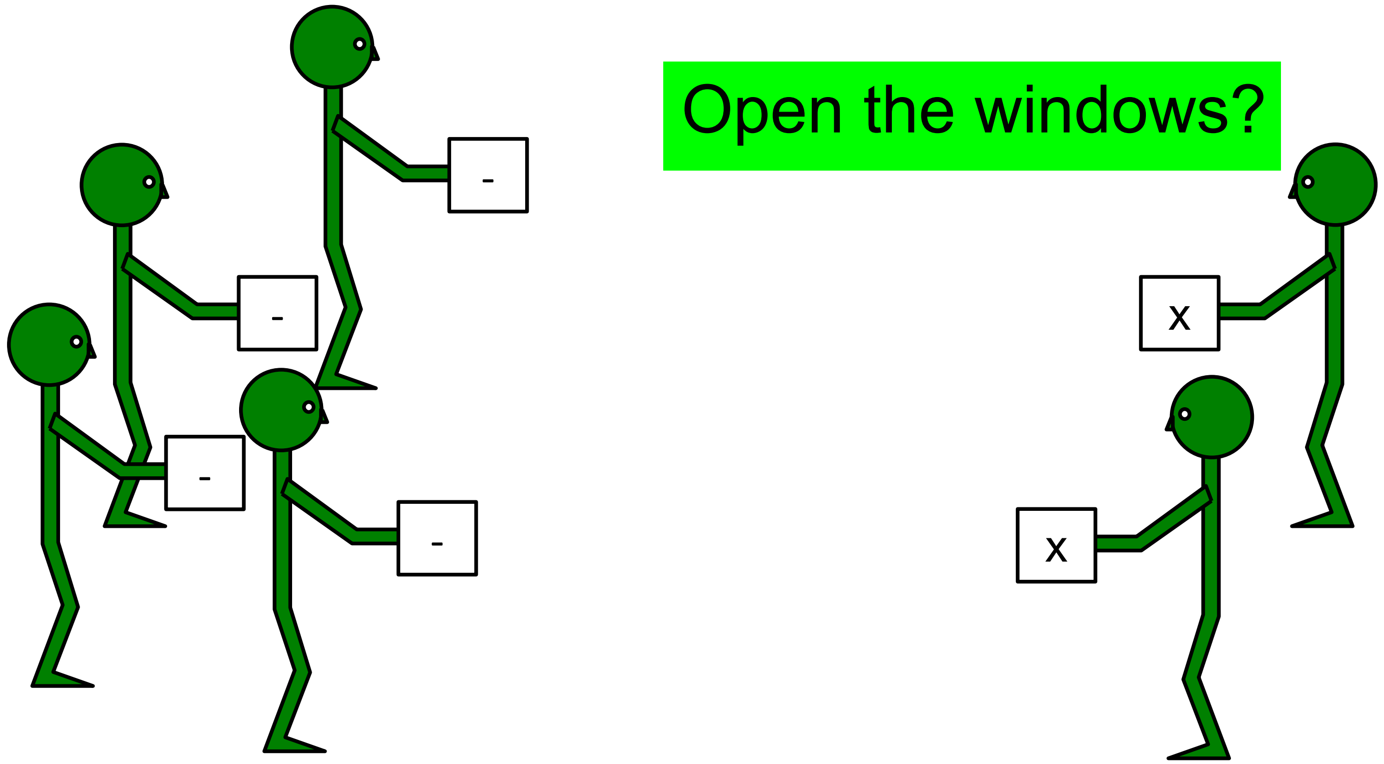
Broadcast

- A one-to-many communication.



Reduction Operations

- Combine data from several processes to produce a single result.



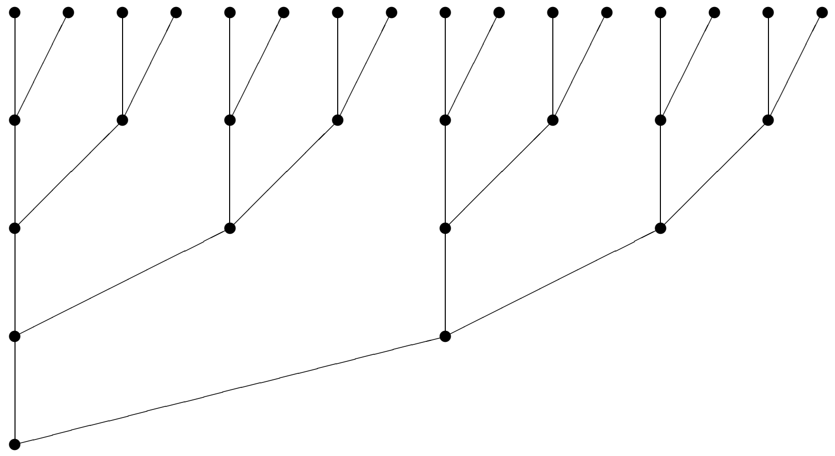
Collective Communication

- Communications involving a group of processes.
- Called by **all** processes in a communicator.
- Collective routines allow for
  - simplified code
    (one routine replacing many P2P calls)
  - optimized forms
    (implementation takes advantage of faster algorithms)

Example:
Process rank=0 sends a number to
30 processes.

Variant 1:
Use 30 calls to MPI_Send

Variant 2:
Use one collective operation



Characteristics of Collective Communication

- Collective action over a communicator
- All processes must communicate
- Synchronization may or may not occur
- No tags
- No interference of P2P and collective communication
- Receive buffers must be exactly the right size
- Collective operations are usually blocking
  (MPI 3.0 introduced non-blocking collective communication)

Barrier Synchronization

- C:

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

- Fortran:

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

Broadcast (I)

- C:

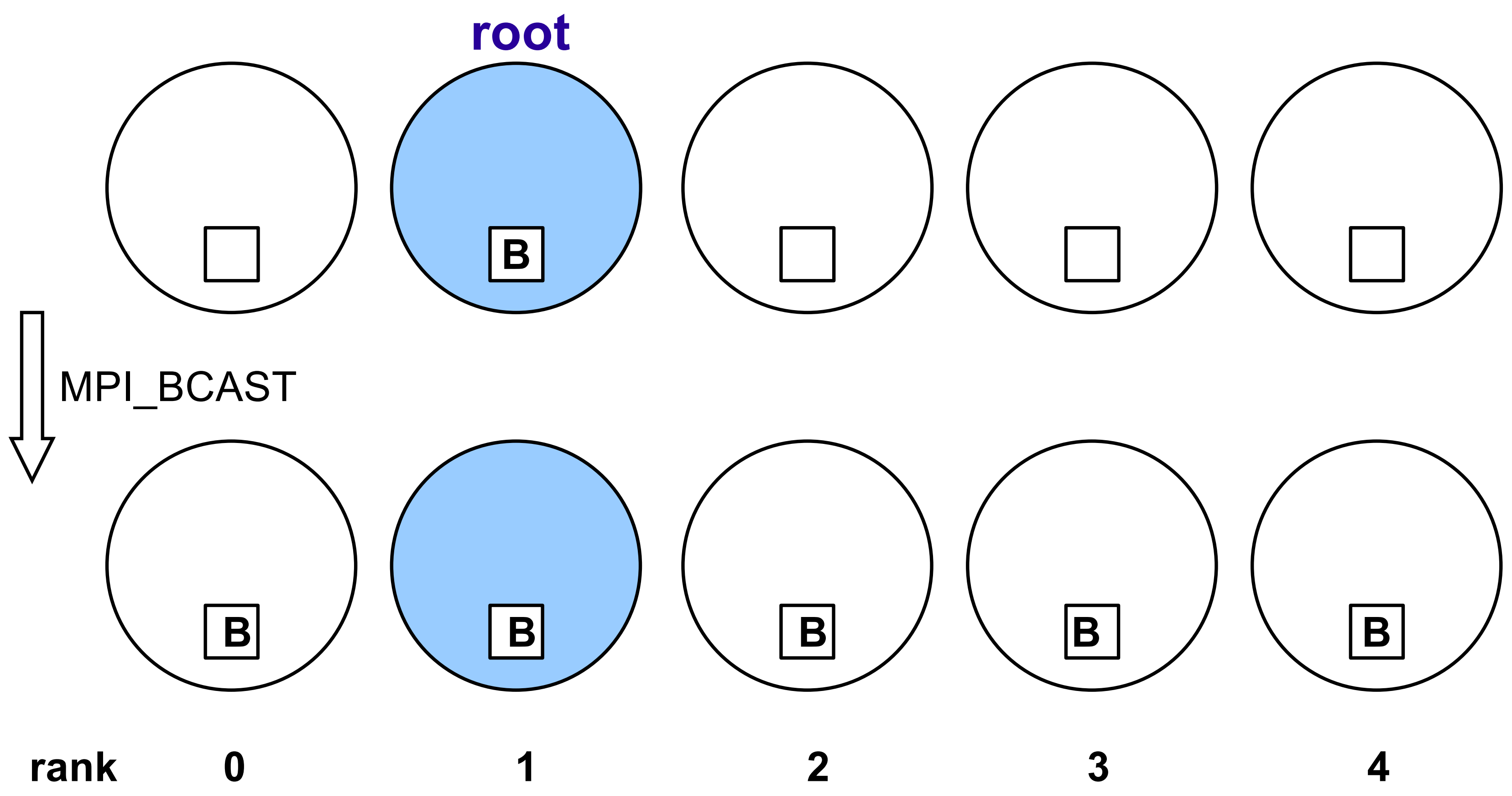
```
int MPI_Bcast (void *buffer,
        int count, MPI_Datatype datatype,
        int root, MPI_Comm comm)
```

- Fortran:

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

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

Broadcast (II)



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

Example of Global Reduction

Integer global sum

- C:

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

- Fortran:

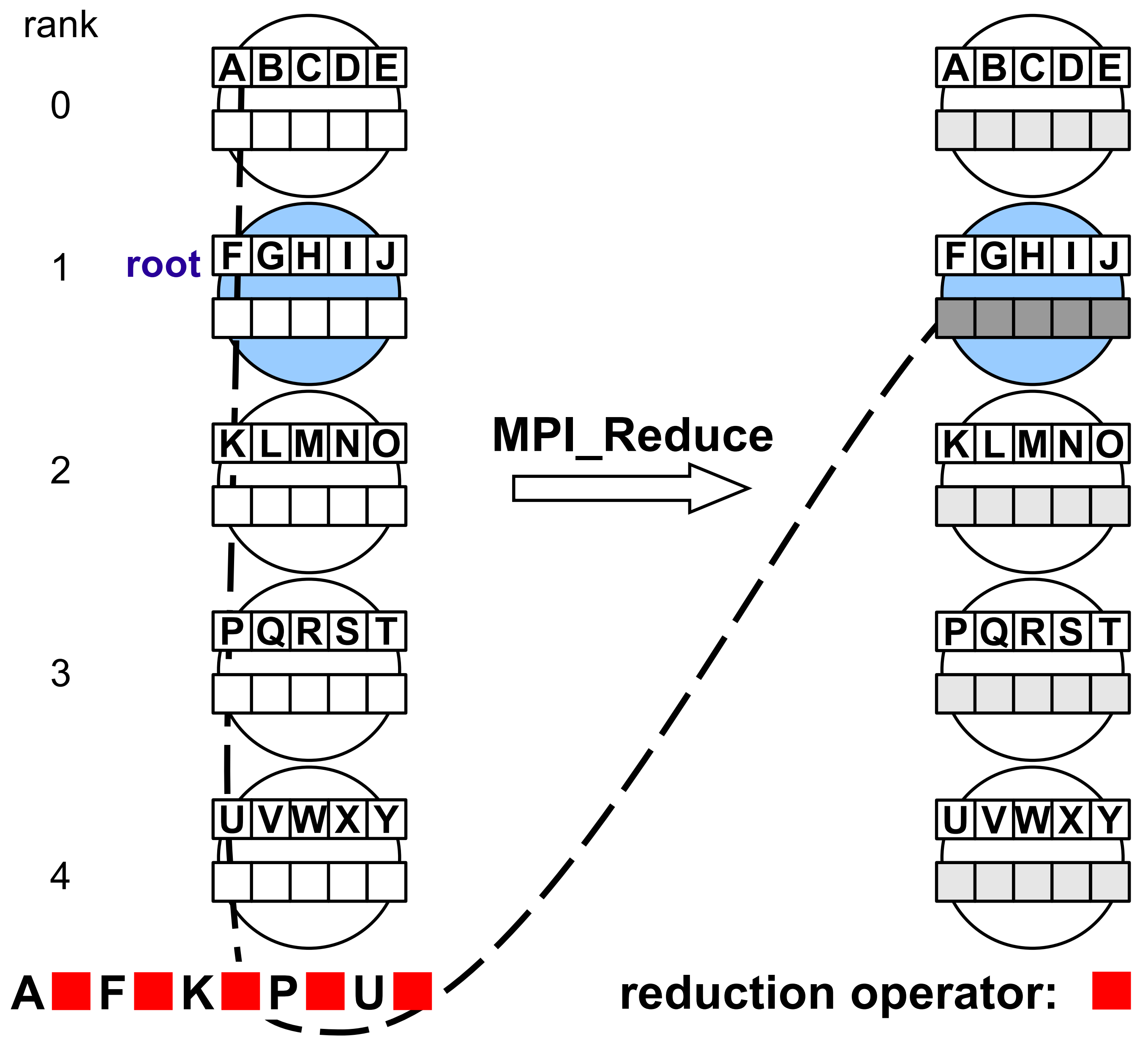
```
CALL MPI_REDUCE(x, result, 1, MPI_INTEGER,
       MPI_SUM, 0, MPI_COMM_WORLD, IERROR)
```

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

Predefined Reduction Operations

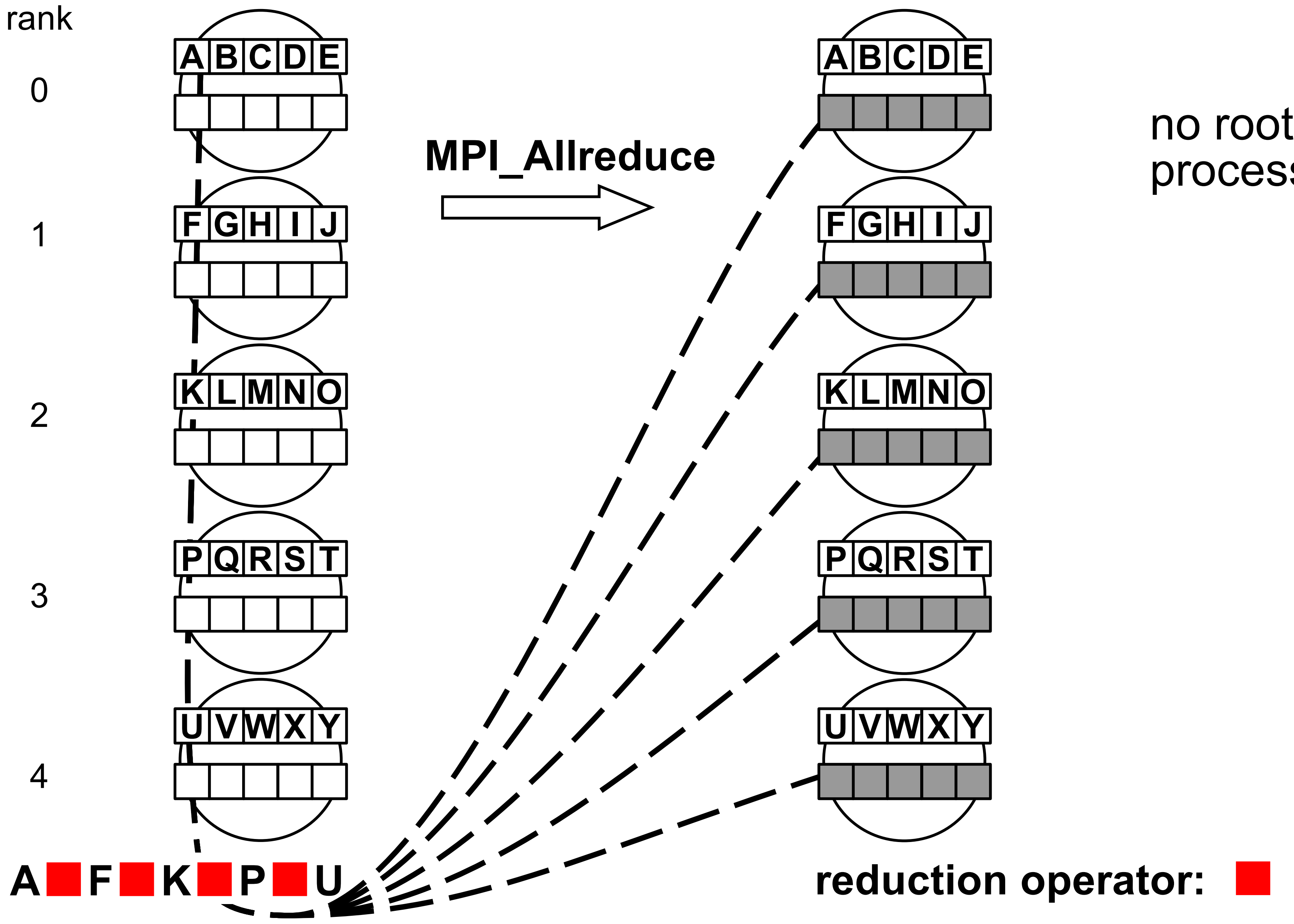
MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

MPI_Reduce



reduction operator: 🟥

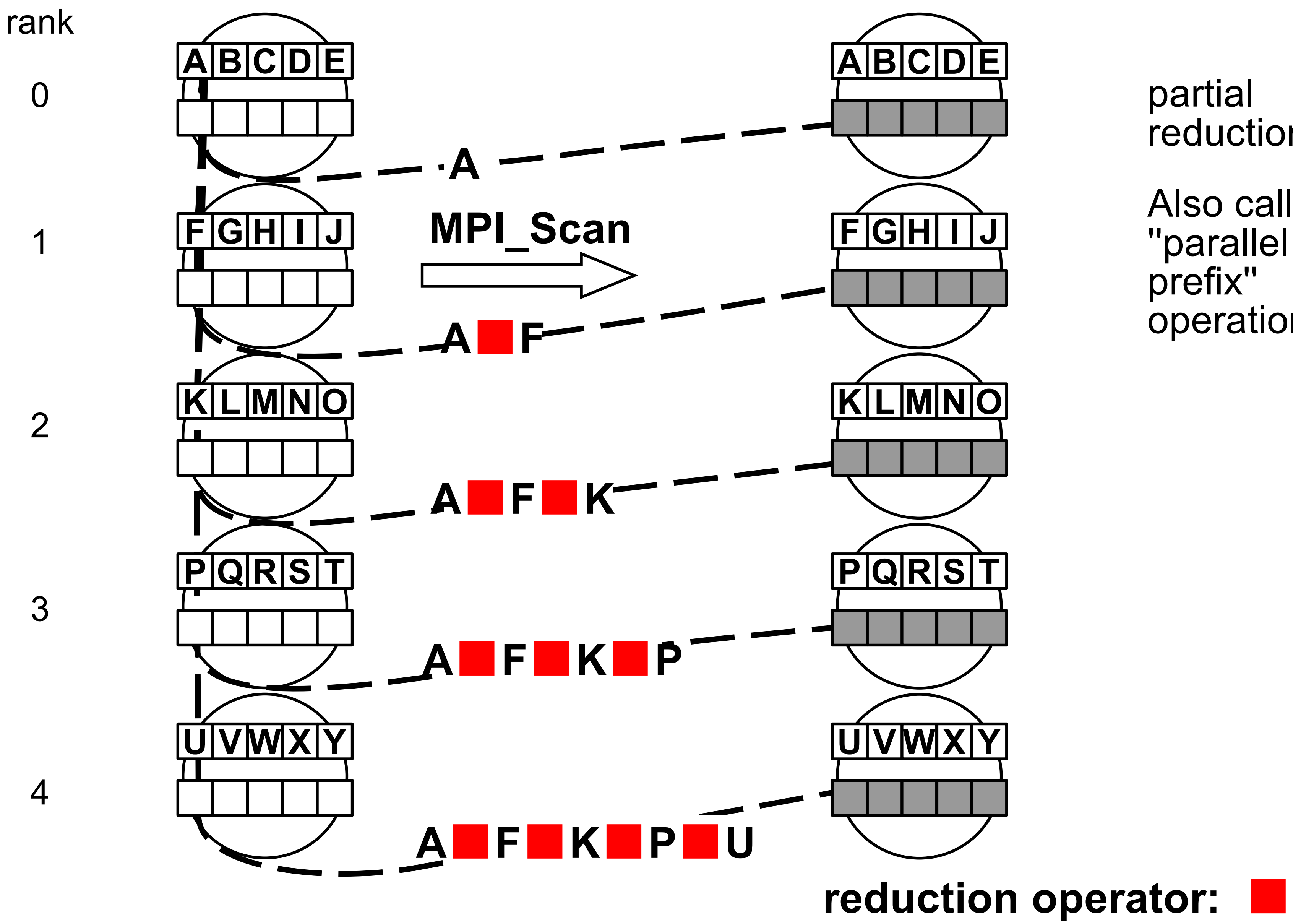
MPI_Allreduce



no root process

reduction operator: 🟥

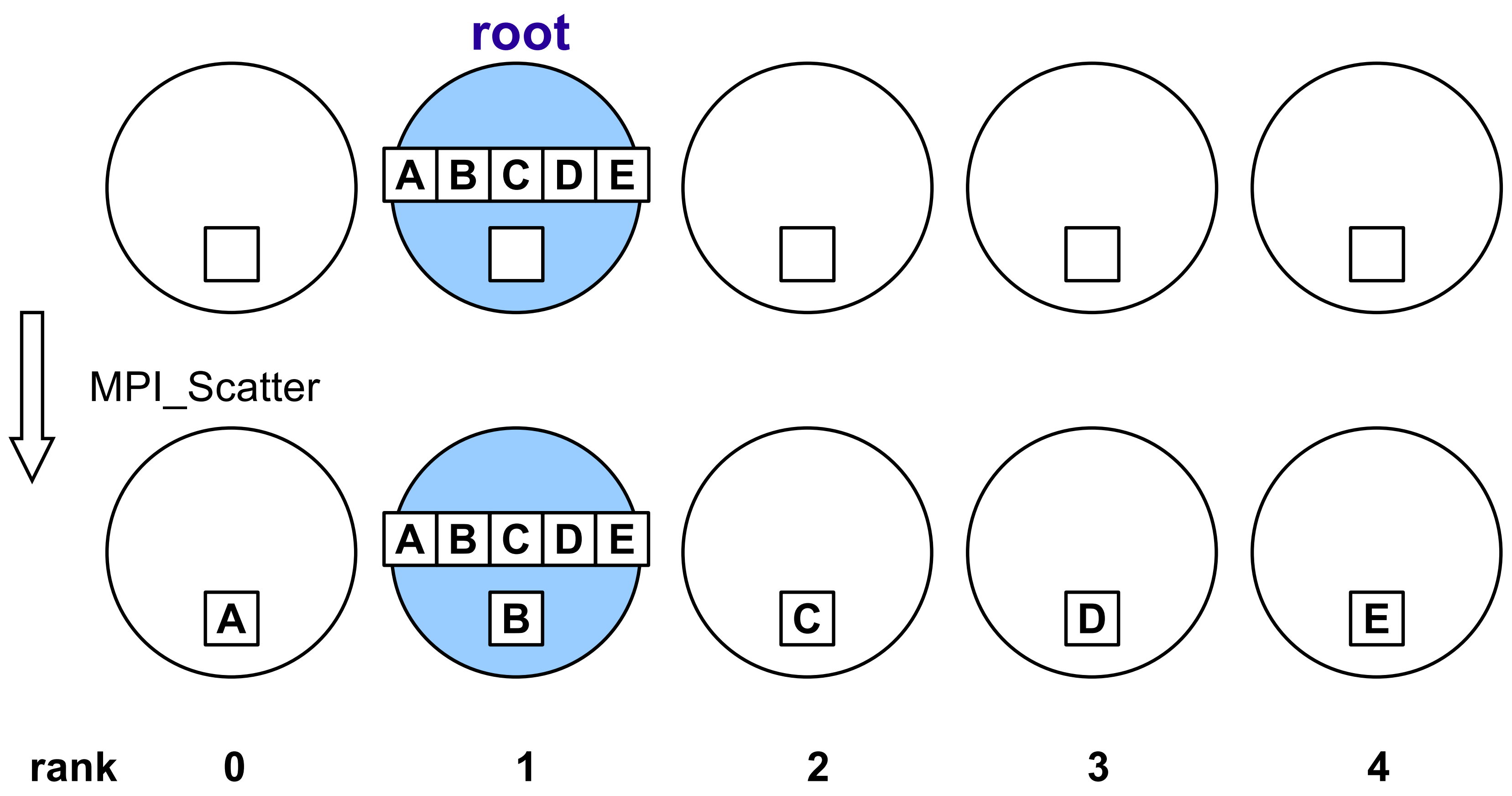
MPI_Scan



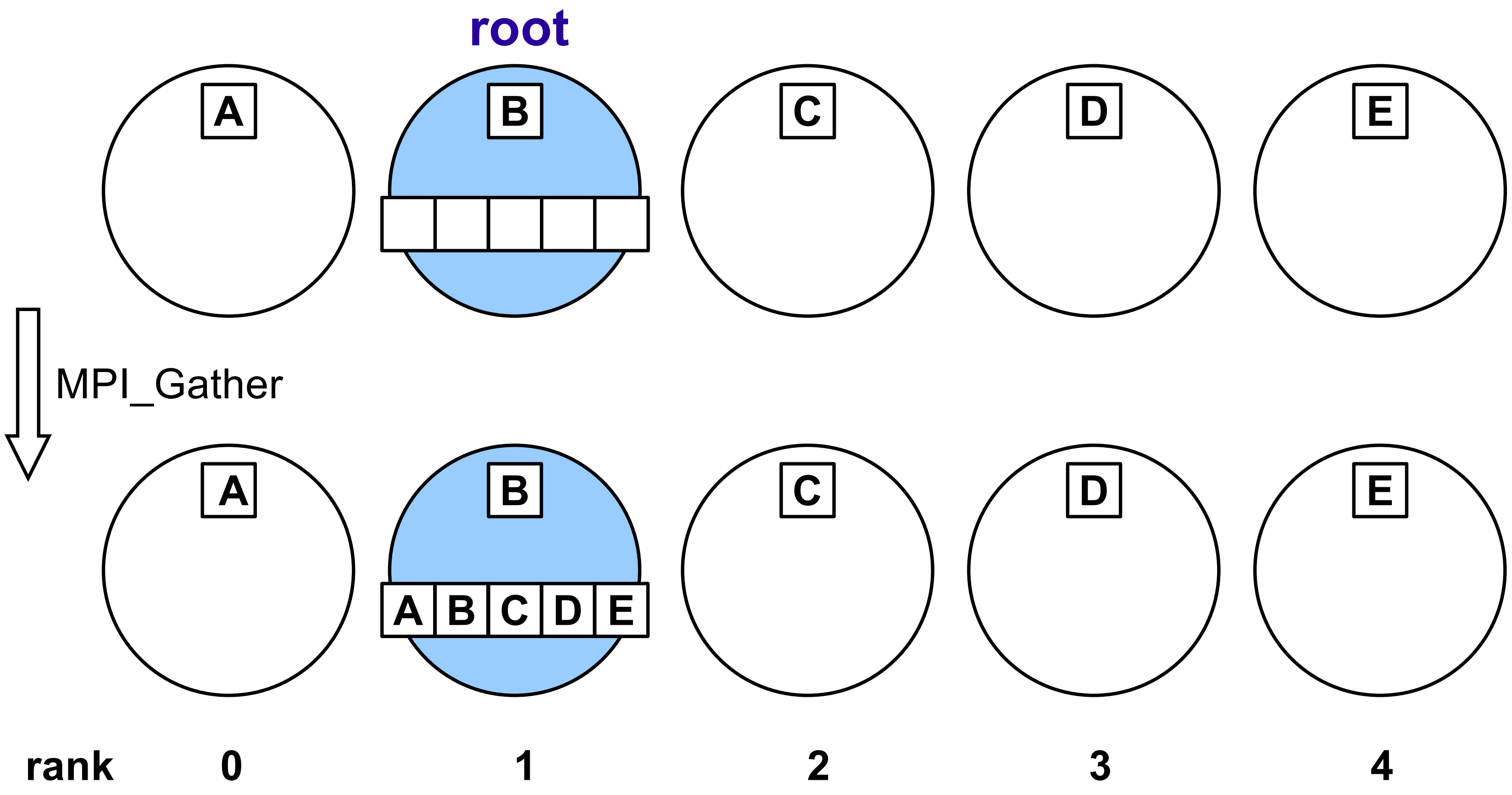
partial reductions

Also called "parallel prefix" operation

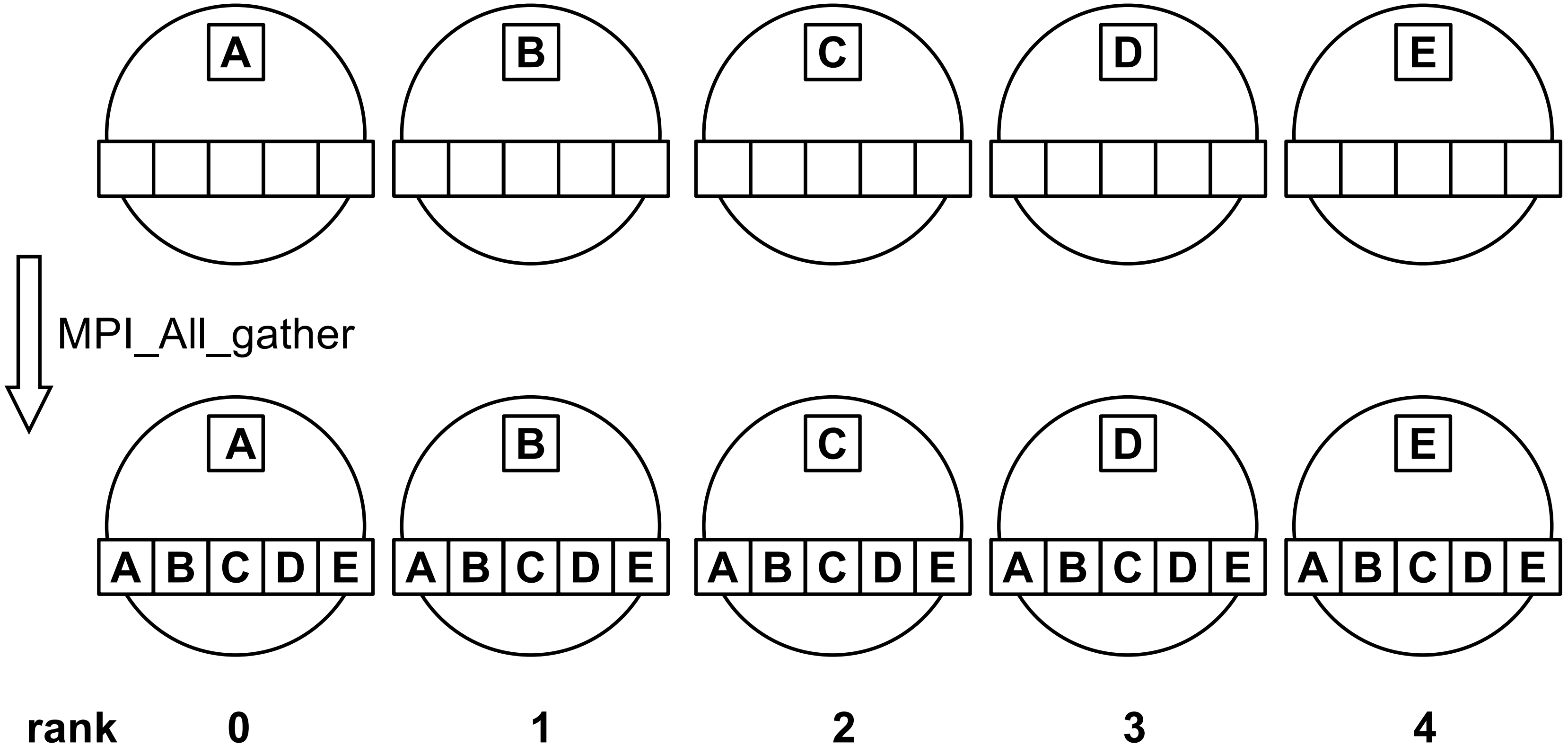
Scatter



Gather

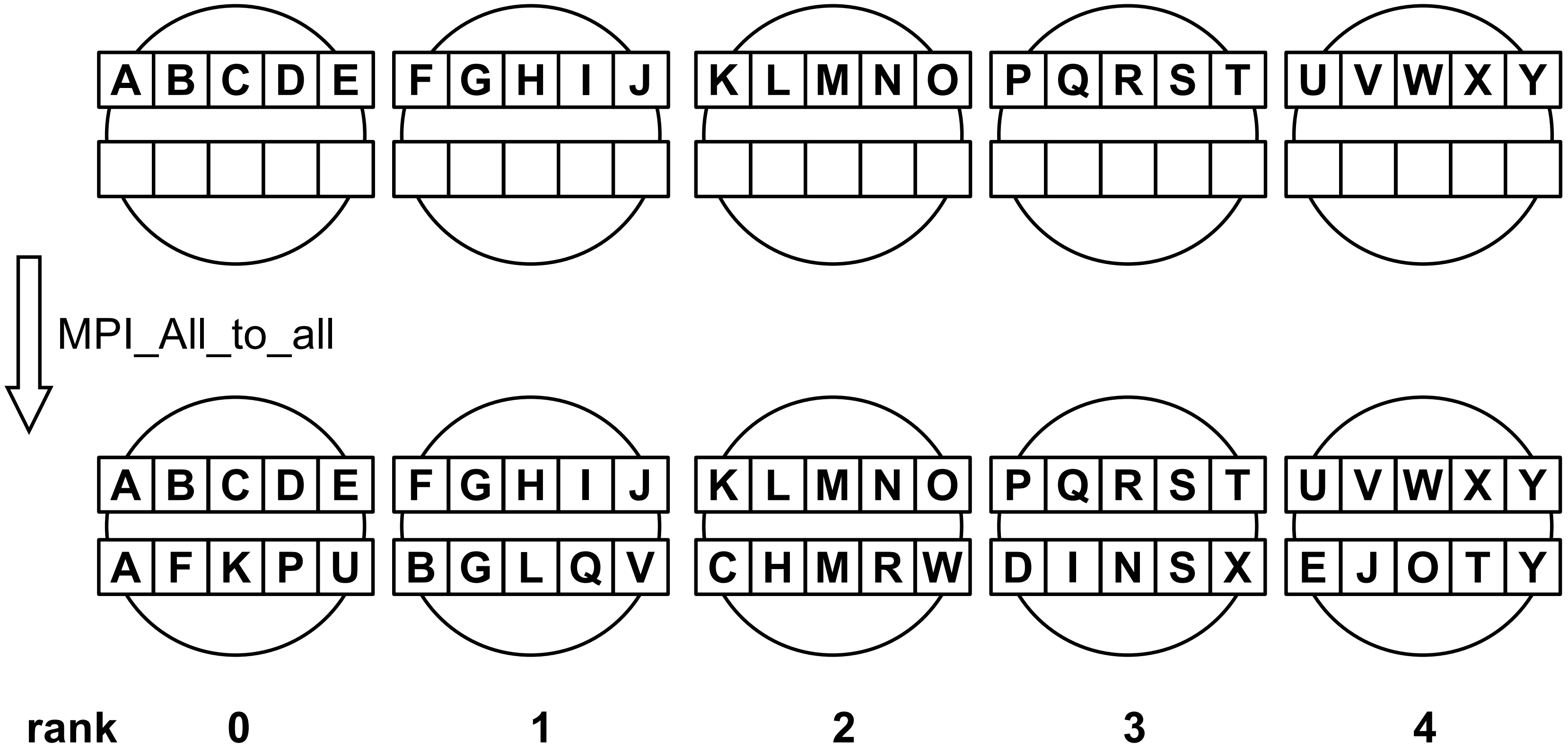


All_Gather

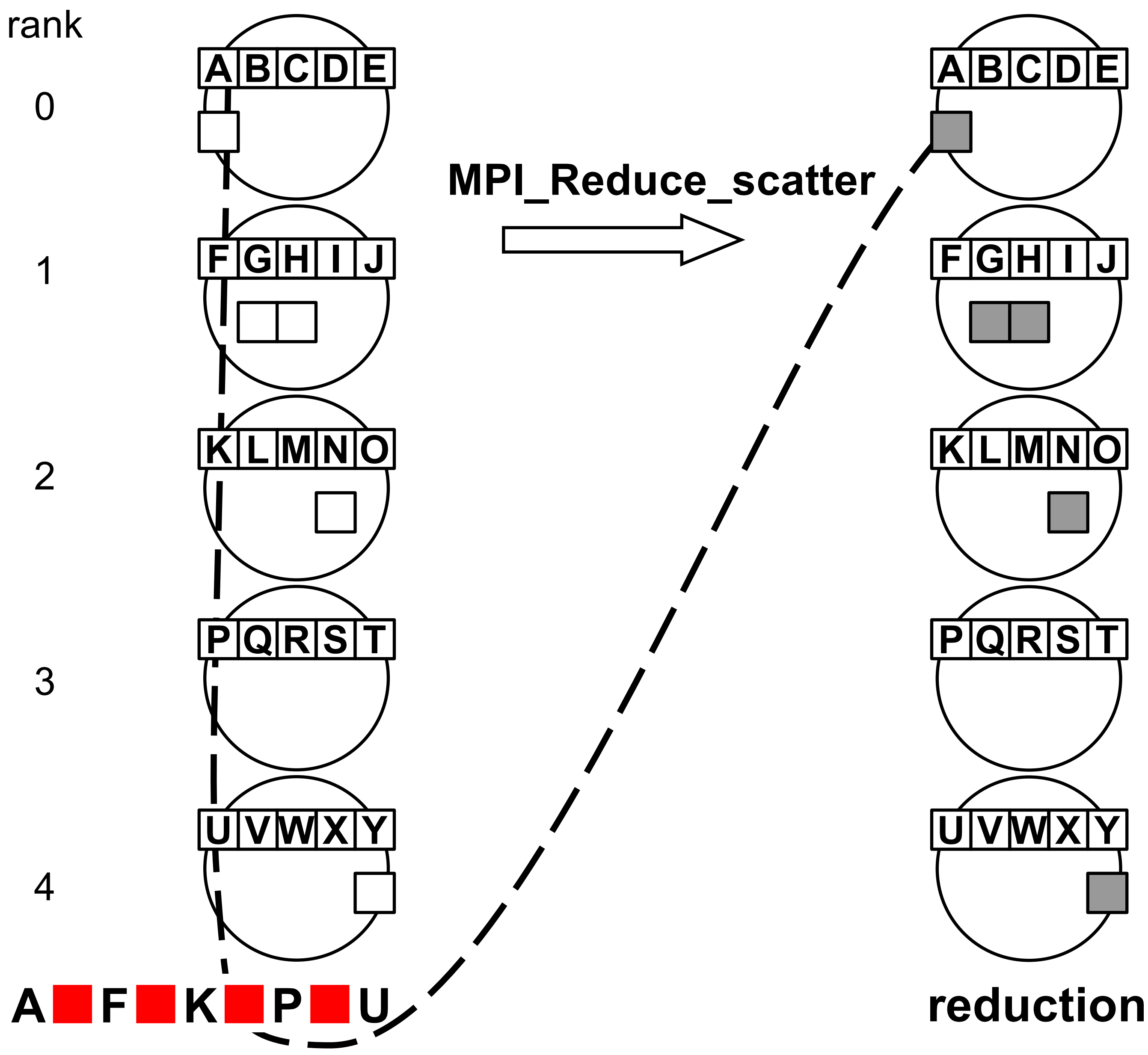


```
All_Gather = Gather + Bcast
```

All_To_All


A B C D E
F G H I J
K L M N O
P Q R S T
U V W X Y
MPI_All_to_all
A B C D E
A F K P U
F G H I J
B G L Q V
K L M N O
C H M R W
P Q R S T
D I N S X
U V W X Y
E J O T Y
rank
0
1
2
3
4

MPI_Reduce_scatter



result is scattered (combination of `MPI_Reduce` and `MPI_Scatterv`)

reduction operator: 🟥

Defining your own reduction operator

User-Defined Reduction Operators

- Reduction using an arbitrary operator, 🟥
- C - function of type `MPI_User_function`:

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

- Fortran - subroutine

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

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

Reduction Operator Functions

- Operator function for 🟥 must act as:

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

- Operator 🟥 does not need to be commutative

Registering a User-Defined Reduction Operator

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

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

- Fortran:

```
SUBROUTINE MPI_OP_CREATE (FUNC, COMMUTE, OP, IERROR)

   EXTERNAL FUNC
   LOGICAL COMMUTE
   INTEGER OP, IERROR
```

Freeing a User-Defined Reduction Operator

- C

```
int MPI_Op_free(MPI_Op *op)
```

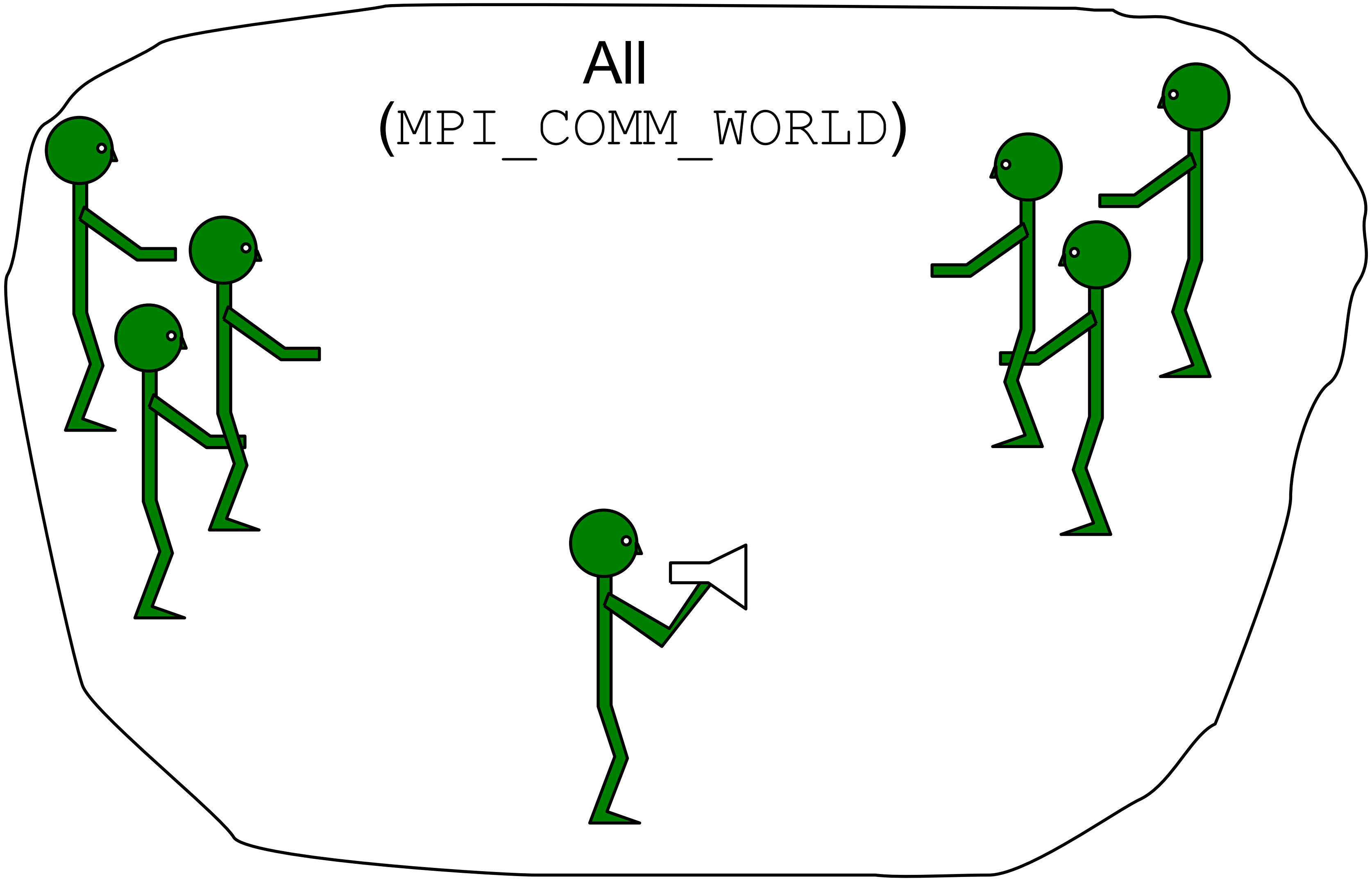
- Fortran:

```
SUBROUTINE MPI_OP_FREE (OP, IERROR)
    INTEGER OP, IERROR
```

Communicators

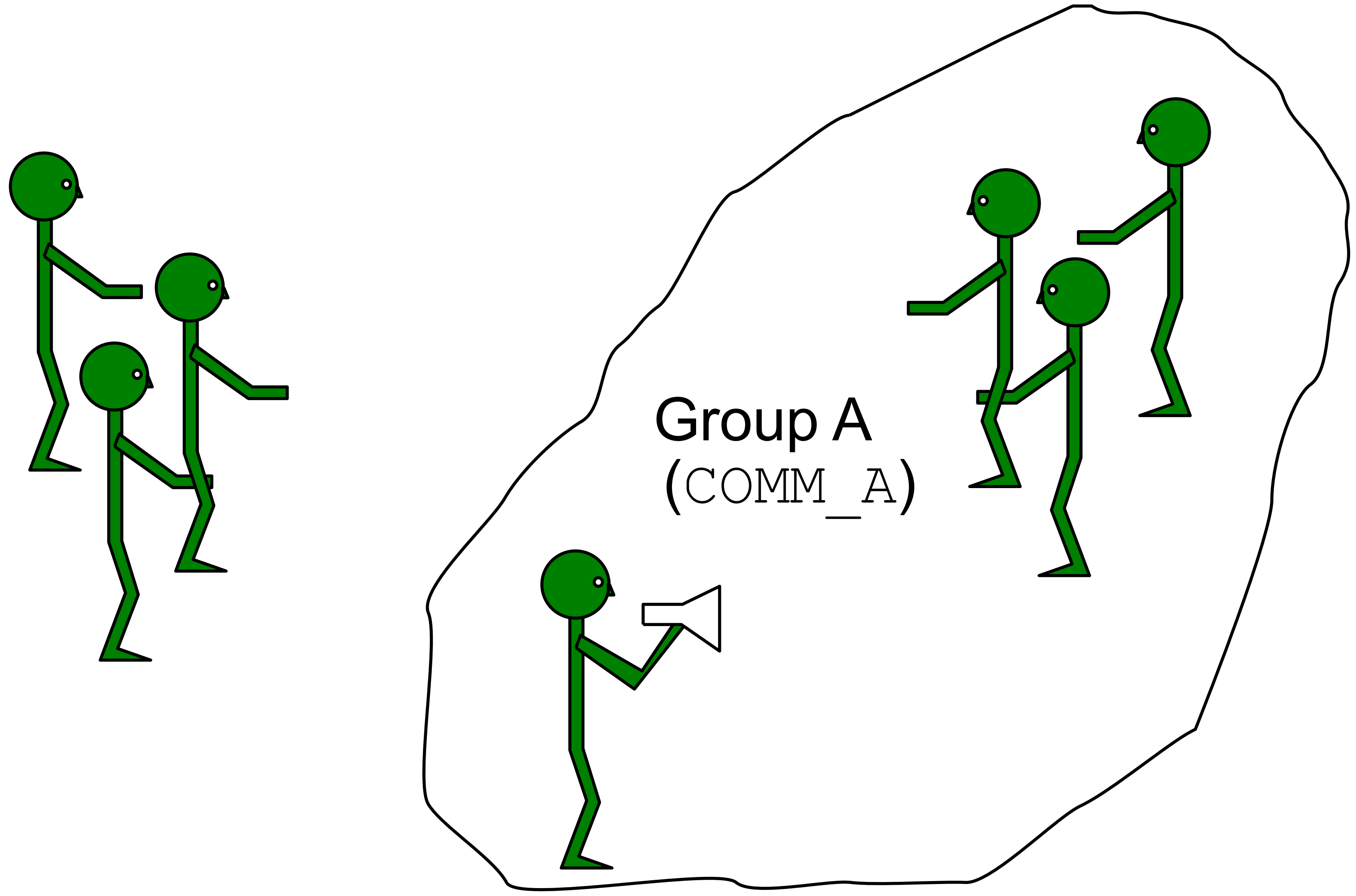
Use of communicators

- Select group of processes



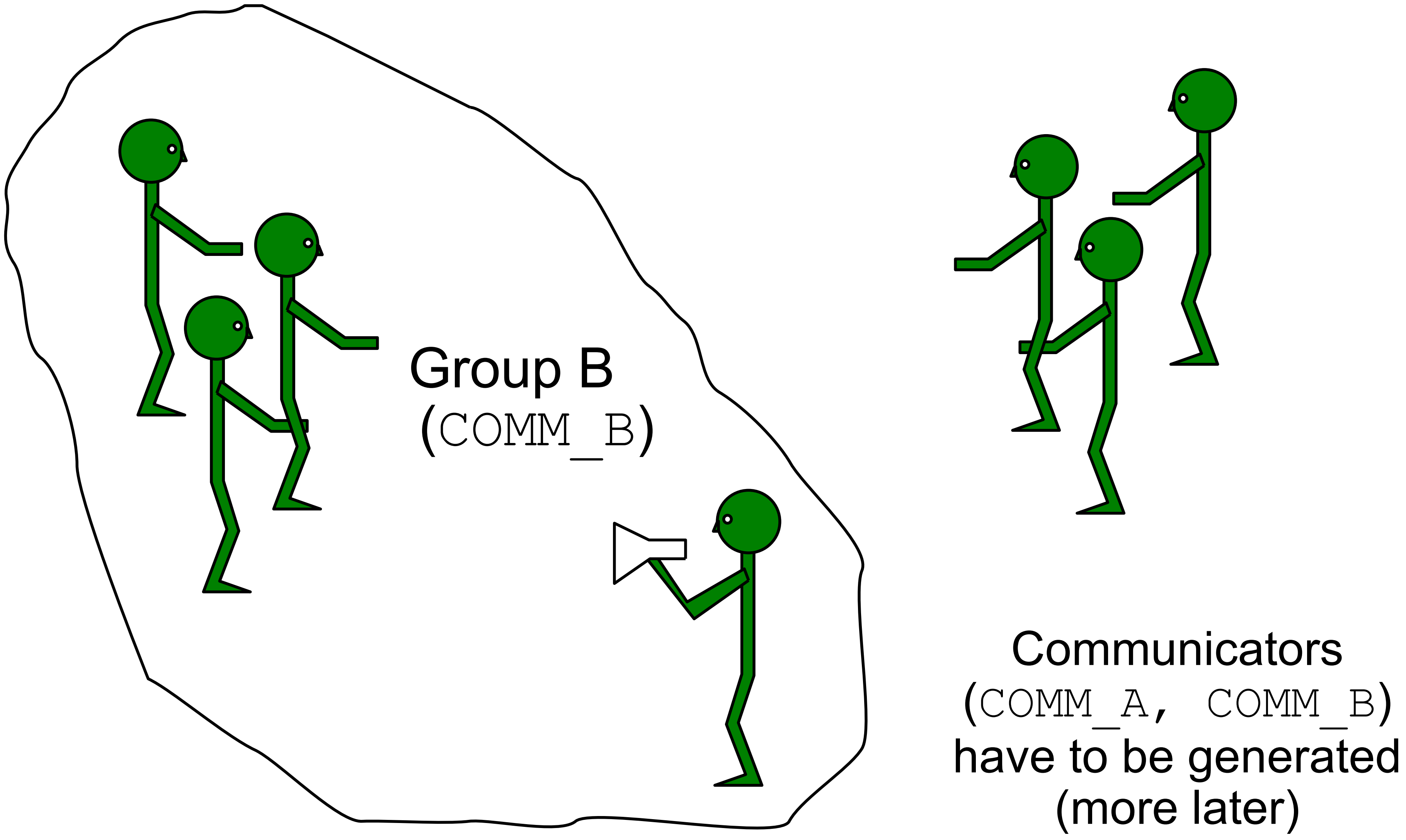
Use of communicators

- Select group of processes



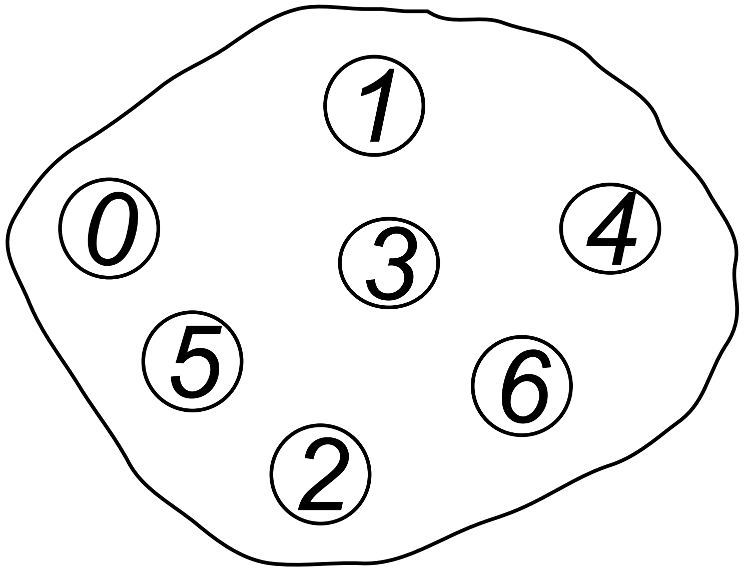
Use of communicators

- Select group of processes

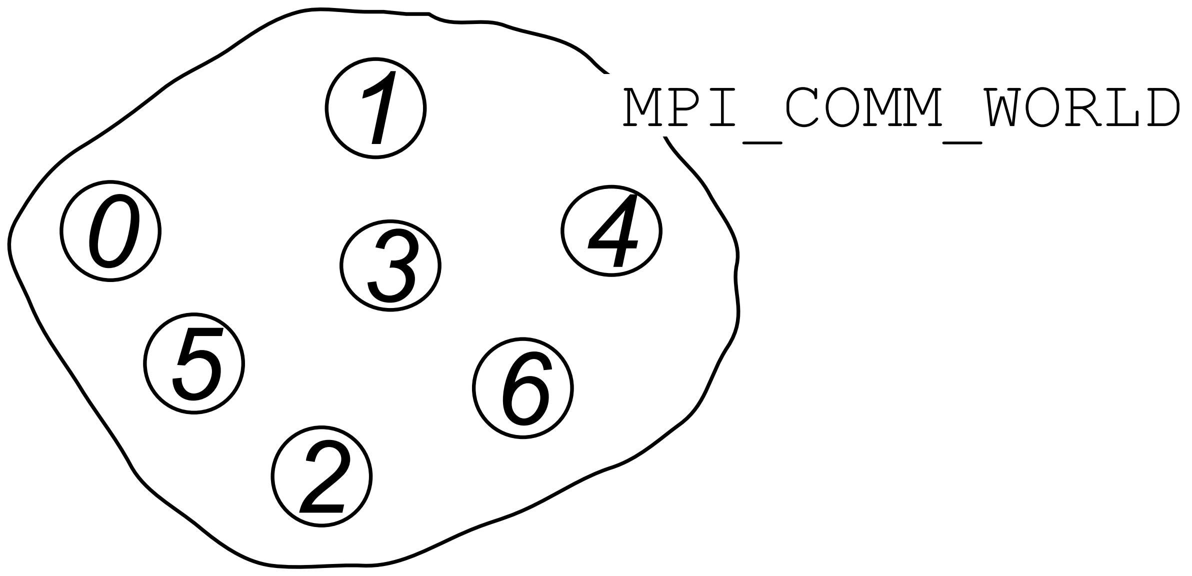


Communicators in MPI

- All MPI communication calls require communicator argument
- Communicator is a handle to a *“context”*
- Processes can have several contexts
- Communication only for sets of processes (a '*group*') that share a context
- Analogy to radio frequency
- `Tag` is not usable why?



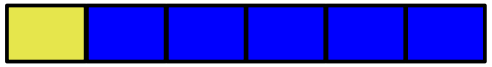
MPI_COMM_WORLD communicator



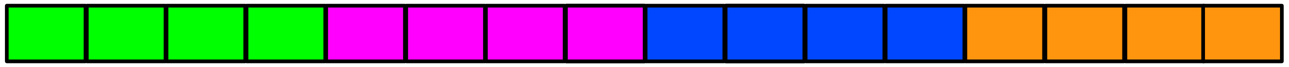
- Predefined (default) communicator; defines the ordered group of all processes and their context (a virtual network), in which they can communicate with each other.
- Additional communicators can be defined as subsets of this group.

Examples for splitting

- Master-Worker:
  - Master: *color*=1
  - Workers: color=2
  - *Key*: e.g. rank in MPI_COMM_WORLD

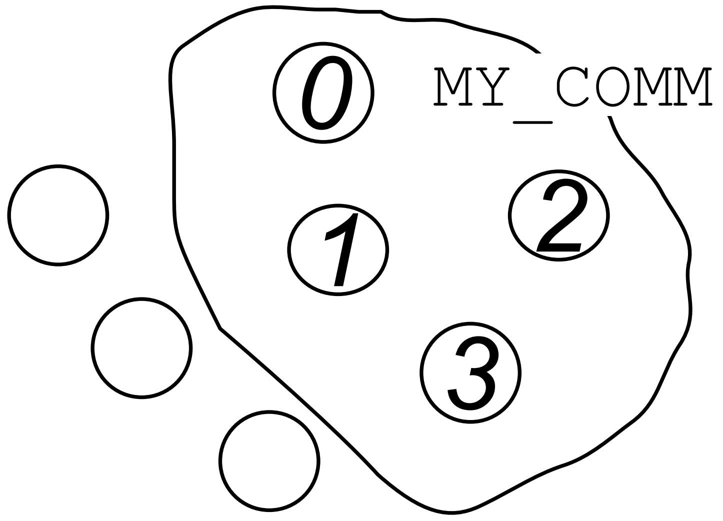


- Ensemble integrations
  (e.g. ensemble of slight different weather conditions
  in atmospheric model)
  - Distinct color for each ensemble member



Creating Communicators

- New communicators are created from existing ones
- 2 steps:
  - Select sub-group of processes for new communicator (local)
  - Global operation to create new communicator
- Simplified functions for some cases



Duplicating a Communicator

- A new context for the same set of processes
- Required, e.g. for parallel libraries (avoid interference)

- C:

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

- Fortran:

```
MPI_COMM_DUP(comm, newcomm, ierror)
    INTEGER comm, newcomm, ierror
```

Splitting a Communicator

- Generate a set of communicators
  - Subsets of processes in distinct contexts
- C:

```
int MPI_Comm_split(MPI_Comm comm, int color,
    int key, MPI_Comm *newcomm)
```

- Fortran:

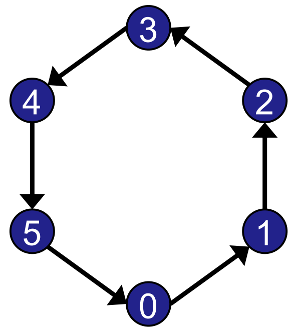
```
MPI_COMM_SPLIT(comm, color, key, newcomm, ierror)

    INTEGER comm, color, key, newcomm, ierror
```

- Processes not to be member of `newcomm` can be specified with `MPI_UNDEFINED;`
- These process get the communicator value `MPI_COMM_NULL`

Exercise 2

Exercise 2: Rotating information around a ring



- A set of processes are arranged in a ring.
- Each process stores its rank in `MPI_COMM_WORLD` in an integer.
- Each process passes this on to its neighbor on the right.
- Keep passing what is received until the own rank is back where it started.
- Each processor calculates the sum of the values.

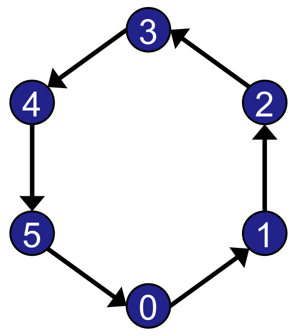
Exercise 2, Part 2: Ring with collective communication

- 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.
- Then rewrite this so that the program prints out the partial results in the correct order (process 0, then process 1, etc.).

Note on Exercise 2: Rotating information around a ring

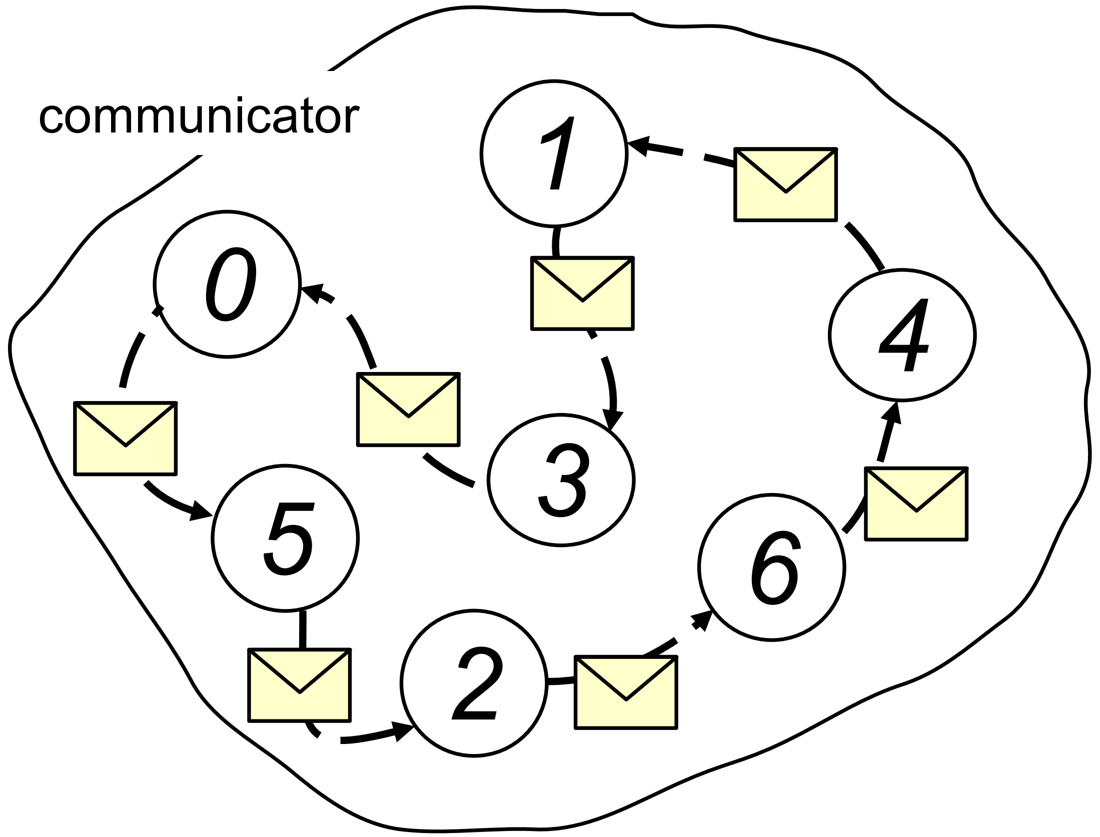
- Different from Ping-Pong
  - There is no start point – all processes do the same work
  - SIMD (single instruction - multiple data)

    → Aim for a code that does the same for all processes



Non-Blocking Communications

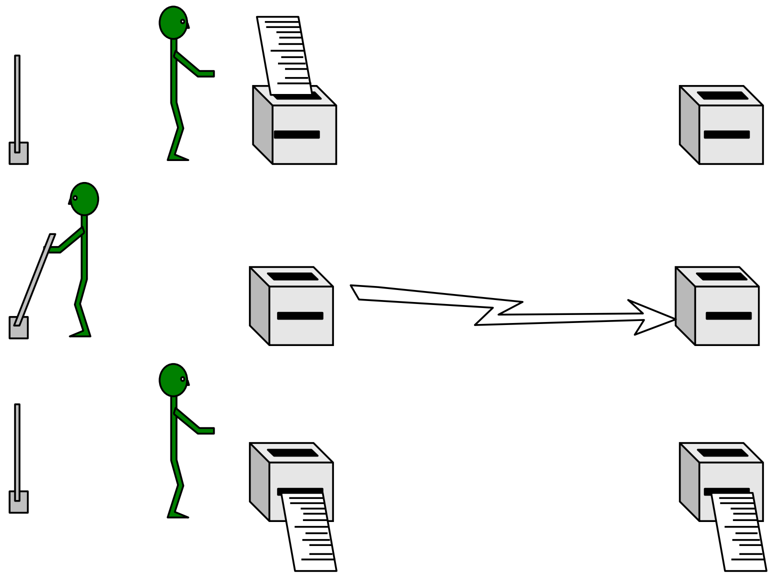
Deadlock



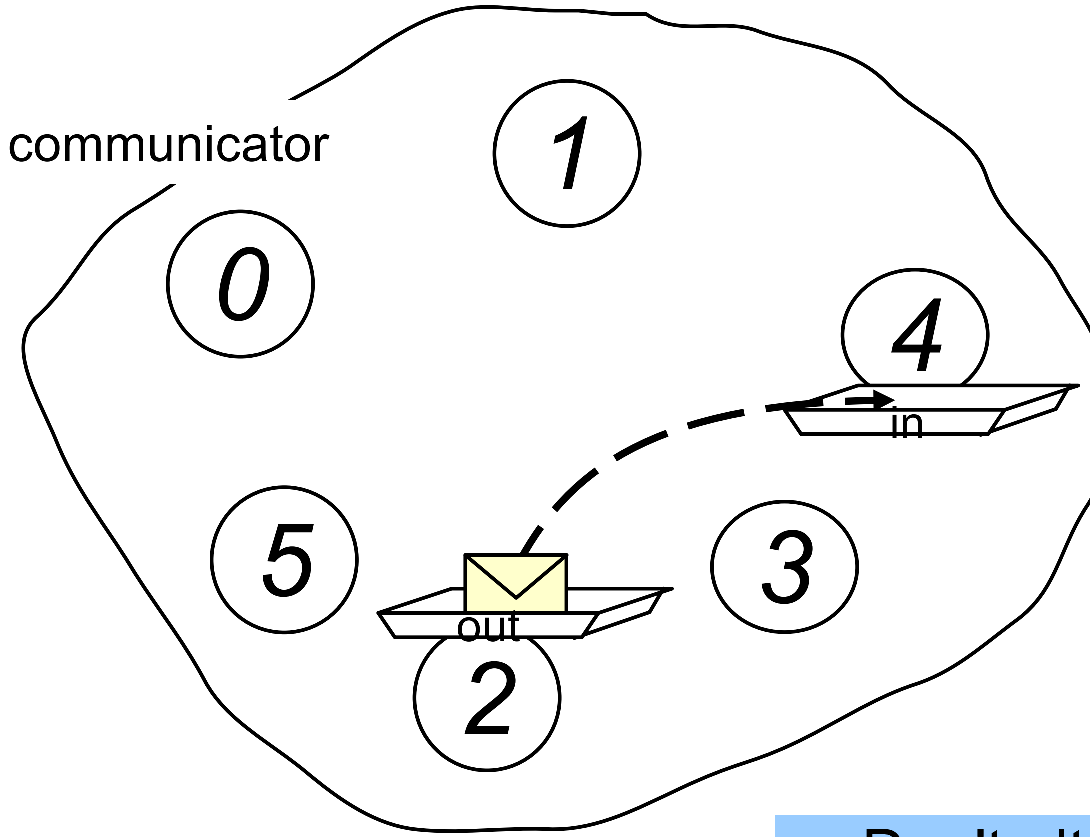
- Deadlock possible
- Avoid by
  - red-black numbering
  - Non-blocking communication
- Non-blocking communication allows for latency-hiding

Non-Blocking Communications

- Separate communication into three phases:
  1. Initiate non-blocking communication.
  2. Do some work (perhaps involving other communications?)
  3. Wait for non-blocking communication to complete.

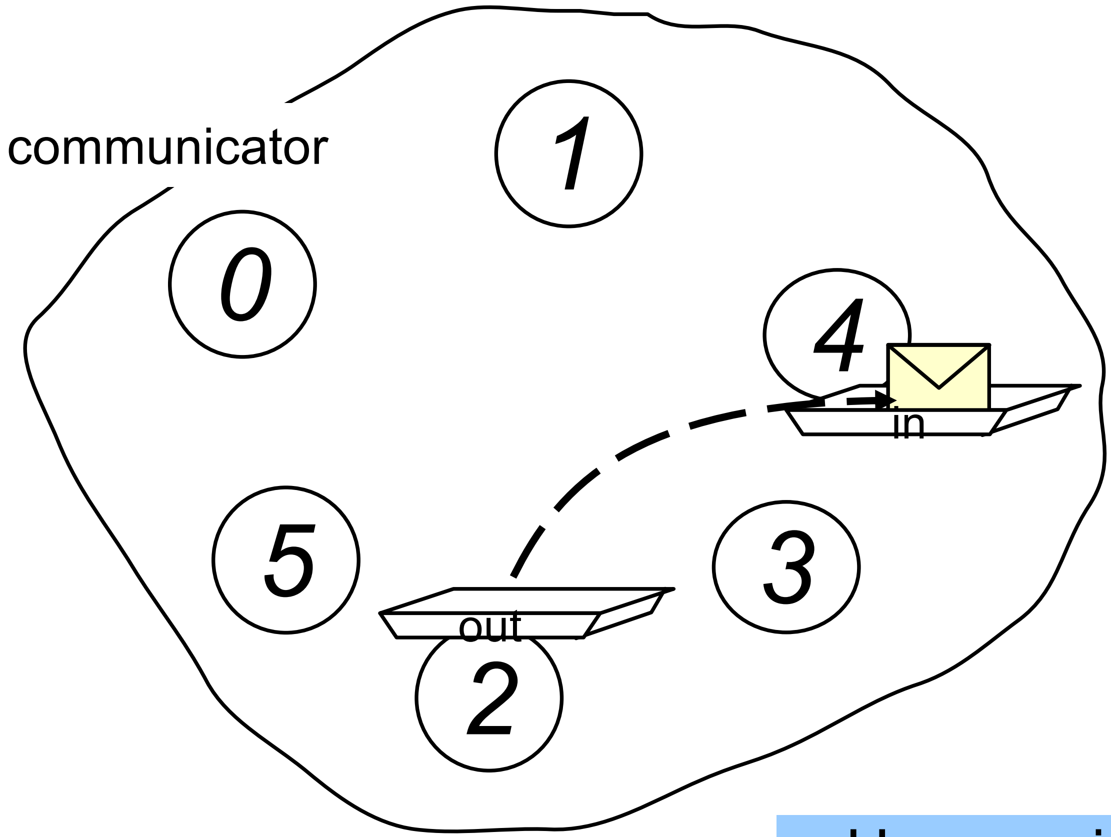


Non-Blocking Send



Don't alter send buffer until send is complete!

Non-Blocking Receive



Use receive buffer after receive is complete!

Handles used for Non-blocking Communication

- datatype – same as for blocking (`MPI_Datatype` or `INTEGER`)
- communicator – same as for blocking (`MPI_Comm` or `INTEGER`)
- request – `MPI_Request` or `INTEGER`
- A *request handle* is allocated when a communication is initiated.

Non-blocking Synchronous Send

- C:

```
MPI_Issend(buf, count, datatype, dest,
              tag, comm, request)
MPI_Wait(request, status)
```

- Fortran:

```
MPI_ISSEND(buf, count, datatype, dest,
              tag, comm, request, ierror)
MPI_WAIT(request, status, ierror)
```

Non-blocking Receive

- C:

```
MPI_Irecv(buf, count, datatype,
          src, tag, comm, request)
MPI_Wait(request, status)
```

- Fortran:

```
MPI_IRECV(buf, count, datatype,
          src, tag, comm, request, ierror)
MPI_WAIT(request, status, ierror)
```

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive, and vice-versa.
- Non-blocking sends can use any mode – synchronous, buffered, standard, or ready.
- Synchronous mode affects completion, not initiation.

Communication Types

Non-blocking Operation	MPI call
Standard send	MPI_Isend
Synchronous send	MPI_Issend
Buffered send	MPI_Ibsend
Ready send	MPI_Irsend
Receive	MPI_Irecv

- mnemonic: “`I`”mmediate

Completion Checking

- Waiting versus Testing.
- C:

```
MPI_Wait(request, status)

MPI_Test(request, flag, status)
```

- Fortran:

```
MPI_WAIT(request, status, ierror)

MPI_TEST(request, flag, status, ierror)
```

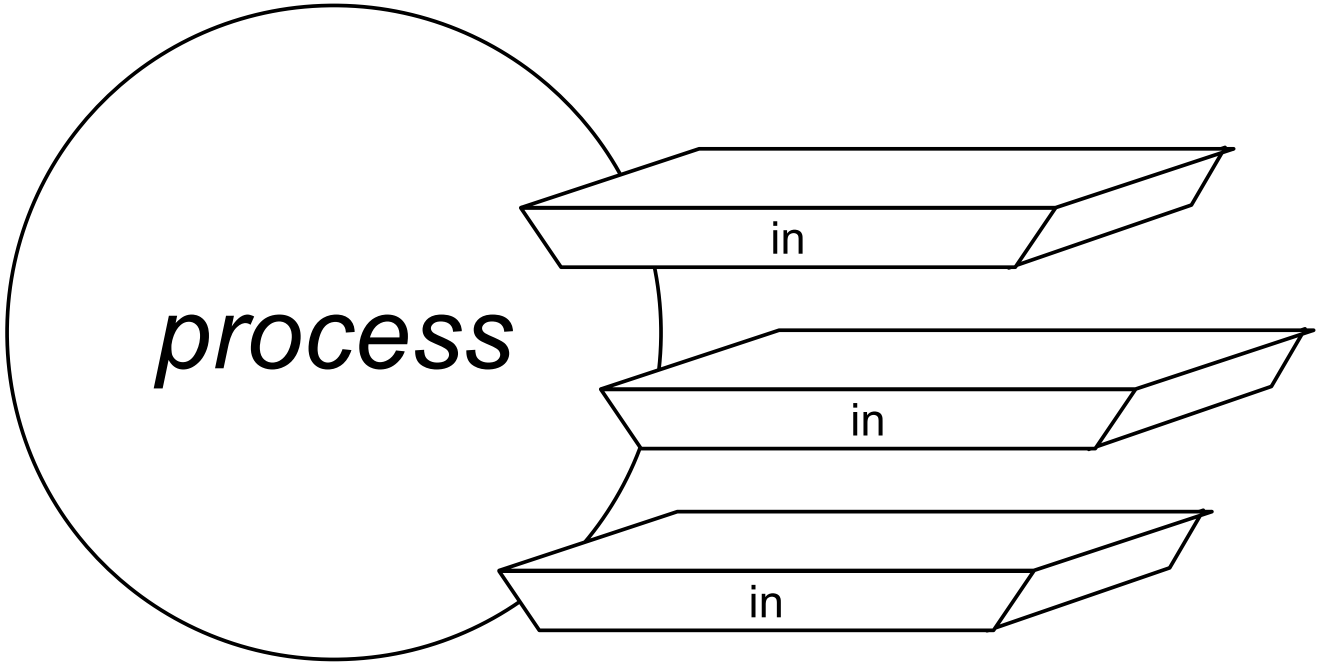
- **Do not reuse buffer** before wait returns or test is successful.
- wait/test is mandatory!

Checking Multiple Communications

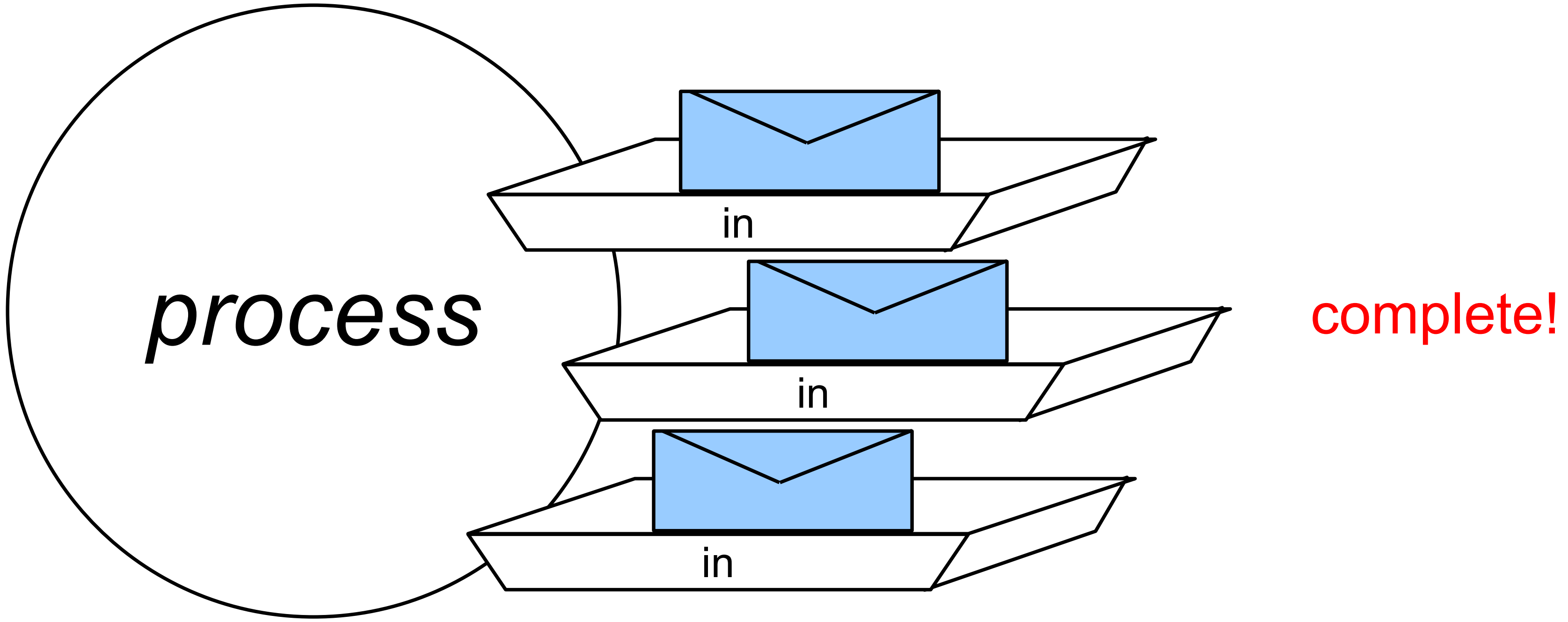
Possibilities:

- Test or wait for completion of **one** message
  ```
  MPI_Waitany (count, array_of_requests, index,
   status)
  ```
- Test or wait for completion of **all** messages
  ```
  MPI_Waitall (count, array_of_requests,
   array_of_statuses)
  ```
- Test or wait for completion of **as many** messages **as possible**
  ```
  MPI_Waitsome(count, array_of_requests, outcount,
   array_of_indices, array_of_statuses)
  ```
- With (in C)
  ```
  MPI_Request request[LENGTH];
  MPI_Status status[LENGTH];
  ```

Testing Multiple Non-Blocking Communications



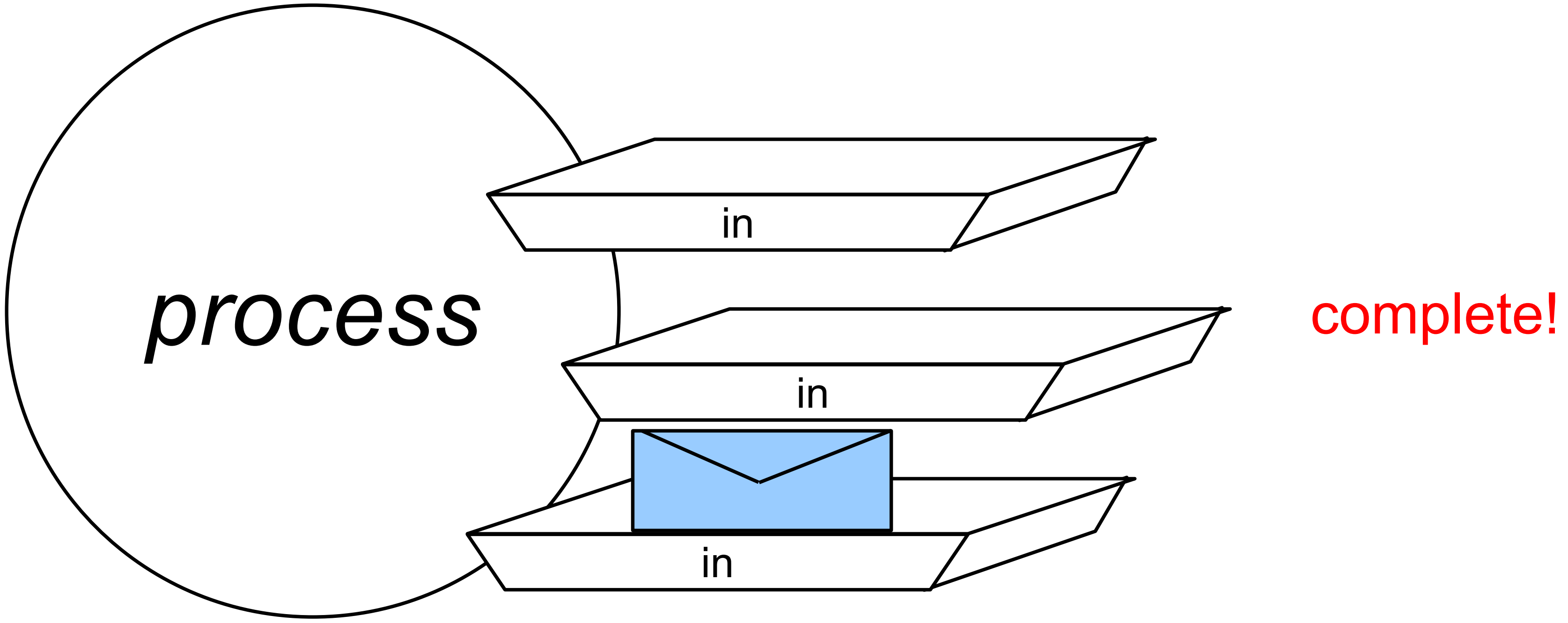
MPI_Waitall



```
MPI_Waitall (count, array_of_requests, array_of_statuses)
```

- Test or wait for completion of **all** messages

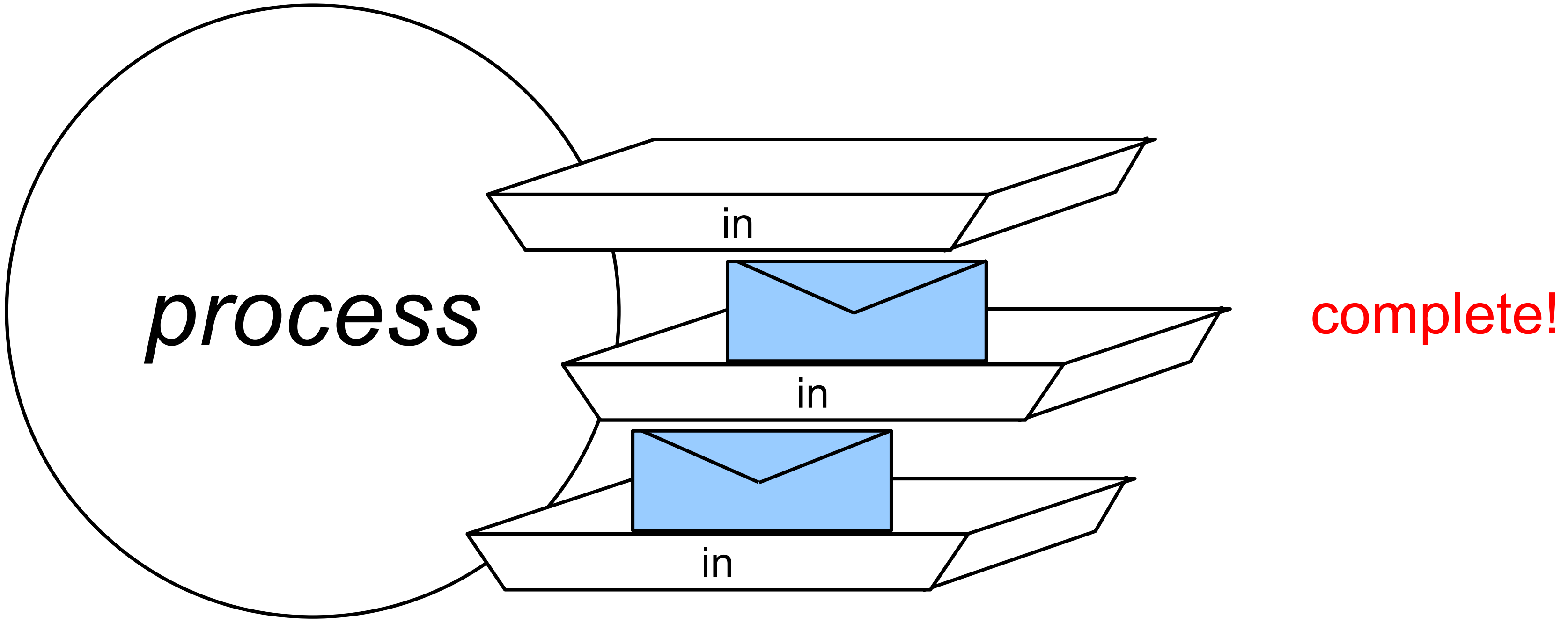
MPI_Waitany



```
MPI_Waitany (count, array_of_requests, index, status)
```

- Test or wait for completion of **one** message

MPI_Waitsome



```
MPI_Waitsome(count, array_of_requests, outcount,
                  array_of_indices, array_of_statuses)
```

- Test or wait for completion of **as many** messages **as possible**

Non-blocking Operations

- Split communication operations into two parts.
  - First part initiates the operation. It does not block.
  - Second part waits for the operation to complete.

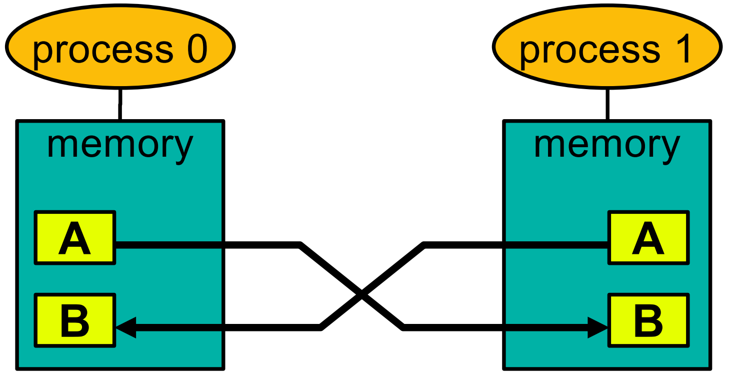
```
MPI_Request request;

MPI_Recv(buf, count, type, dest, tag, comm, status)
  =
MPI_Irecv(buf, count, type, dest, tag, comm, &request)
  +
MPI_Wait(&request, &status)

MPI_Send(buf, count, type, dest, tag, comm)
  =
MPI_Isend(buf, count, type, dest, tag, comm, &request)
  +
MPI_Wait(&request, &status)
```

Using non-blocking Operations (I)

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
```



Process 0:

```
MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request)
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
MPI_Wait(&request, &status)
```

Process 1:

```
MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request)
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
MPI_Wait(&request, &status)
```

- No deadlock
- Data may be transferred concurrently
- “status” argument provides information on received data

Using non-blocking Operations (II)

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
p=1-me; /* calculates partner in 2 process exchange */
```

Process 0 and 1:

```
MPI_Isend(A, 100, MPI_DOUBLE, p, MYTAG, WORLD, &request)
MPI_Recv(B, 100, MPI_DOUBLE, p, MYTAG, WORLD, &status)
MPI_Wait(&request, &status)
```

- No deadlock
- "`status`" argument to `MPI_Wait` doesn't return useful info here!
- Better to use `Irecv` instead of `Isend` if only using one.
- Note: The calls on processes 0 and 1 are identical!

Overlapping communication and computation

On some computers it may be possible to do useful work while data is being transferred.

```
MPI_Request requests[2];
MPI_Status statuses[2];

MPI_Irecv(B, 100, MPI_DOUBLE, p, MYTAG, WORLD, &requests[1])
MPI_Isend(A, 100, MPI_DOUBLE, p, MYTAG, WORLD, &requests[0])
... do some useful work here ...
MPI_Waitall(2, requests, statuses)
```

- `Irecv/Isend` initiate communication
- Communication proceeds “behind the scenes” while processor is doing useful work
- Need both `Isend` and `Irecv` for real overlap (not just one)
- Hardware support necessary for true overlap

Non-Blocking Operations (cont'd)

- All non-blocking operations need a matching wait operation. Some systems cannot free internal resources until wait has been called.
- Buffers must not be reused before test/wait completes.
- A non-blocking operation immediately followed by a matching (blocking) wait is equivalent to the corresponding blocking operation.
- Note: Non-blocking operations are not the same as sequential subroutine calls as the operation continues after the call has returned.
- Can we have a non-blocking synchronous send?

  → Example: Registered mail with receipt