MPI

Message Passing Programming

- Model
  - Set of processes that each have local data and are able to communicate with each other by sending and receiving messages
- Advantages
  - Useful and complete model to express parallel algorithms
  - Potentially fast
  - What is used in practice

What is MPI ?

- A coherent effort to produce a standard for message passing
  - Before MPI, proprietary (Cray shmem, IBM MPL), and research community (PVM, p4) libraries
- A message-passing library specification
  - For Fortran, C, and C++

MPI History

- MPI forum: government, academia, and industry
  - November 1992 committee formed
  - May 1994 MPI 1.0 published
  - June 1995 MPI 1.1 published (clarifications)
  - April 1995 MPI 2.0 committee formed
  - July 1997 MPI 2.0 published
  - July 1997 MPI 1.2 published (clarifications)
  - November 2007, work on MPI 3 started

Current Status

- MPI 1.2
  - MPICH from ANL/MSU
  - LAM from Indiana University (Bloomington)
  - IBM, Cray, HP, SGI, NEC, Fujitsu
- MPI 2.0
  - Fujitsu (all), IBM, Cray, NEC (most), MPICH, LAM, HP (some)

Parallel Programming With MPI

- Communication
  - Basic send/receive (blocking)
  - Collective
  - Non-blocking
  - One-sided (MPI 2)
- Synchronization
  - Implicit in point-to-point communication
  - Global synchronization via collective communication
- Parallel I/O (MPI 2)

Creating Parallelism

- Single Program Multiple Data (SPMD)
  - Each MPI process runs a copy of the same program on different data
  - Each copy runs at own rate and is not explicitly synchronized
  - May take different paths through the program
    - Control through rank and number of tasks

Creating Parallelism

- Multiple Program Multiple Data
  - Each MPI process can be a separate program
- With OpenMP, pthreads
  - Each MPI process can be explicitly multi-threaded, or threaded via some directive set such as OpenMP

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_Init`**
  - **`MPI_Finalize`**
  - **`MPI_Comm_size`**
  - **`MPI_Comm_rank`**
  - **`MPI_Send`**
  - **`MPI_Recv`**

Gropp, Lusk

Simple Hello (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Gropp, Lusk

Notes on C, Fortran, C++

- In C:
  - `#include mpi.h`
  - MPI functions return error code or `MPI_SUCCESS`
- In Fortran
  - `include mpif.h`
  - `use mpi` (MPI 2)
  - All MPI calls are subroutines, return code is final argument
- In C++
  - `Size = MPI::COMM_WORLD.Get_size();` (MPI 2)

Timing MPI Programs

- **MPI_WTIME** returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past

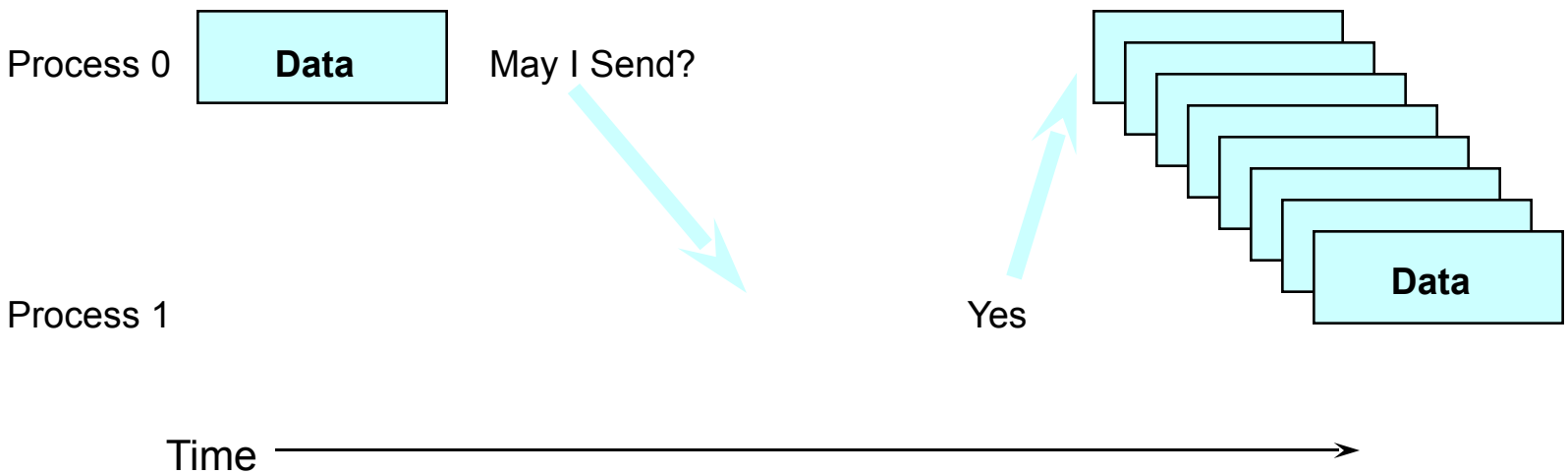
```
double MPI_Wtime( void )
DOUBLE PRECISION MPI_WTIME( )
```

- **MPI_WTICK** returns the resolution of **MPI_WTIME** in seconds. It returns, as a double precision value, the number of seconds between successive clock ticks.

```
double MPI_Wtick( void )
DOUBLE PRECISION MPI_WTICK( )
```

What is message passing?

- Data transfer plus synchronization

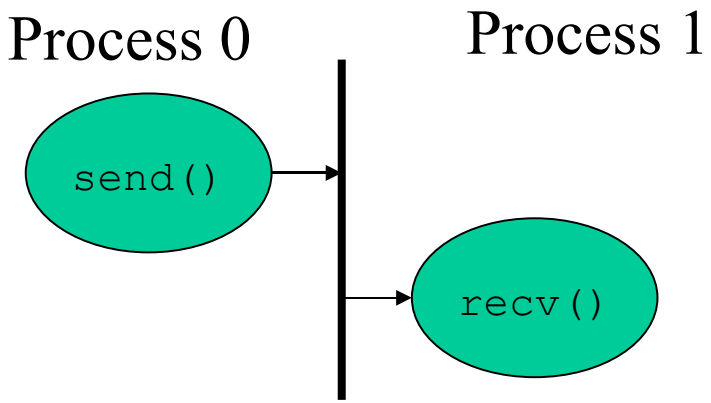


- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

Gropp, Lusk

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Identifying Processes

- MPI Communicator
  - Defines a group (set of ordered processes) and a context (a virtual network)
- Rank
  - Process number within the group
  - `MPI_ANY_SOURCE` will receive from any process
- Default communicator
  - `MPI_COMM_WORLD` the whole group

Identifying Messages

- An MPI Communicator defines a virtual network, `send/recv` pairs must use the same communicator
- `send/recv` routines have a tag (integer variable) argument that can be used to identify a message, or screen for a particular message.
  - `MPI_ANY_TAG` will receive a message with any tag

Identifying Data

- Data is described by a triple (address, type, count)
  - For `send`, this defines the message
  - For `recv`, this defines the size of the receive buffer
- Amount of data received, source, and tag available via status data structure
  - Useful if using `MPI_ANY_SOURCE`, `MPI_ANY_TAG`, or unsure of message size (must be smaller than buffer)

MPI Types

- Type may be recursively defined as:
  - An MPI predefined type
  - A contiguous array of types
  - An array of equally spaced blocks
  - An array of arbitrary spaced blocks
  - Arbitrary structure
- Each user-defined type constructed via an MPI routine, e.g. `MPI_TYPE_VECTOR`

MPI Predefined Types

C:	Fortran:
MPI_INT	MPI_INTEGER
MPI_FLOAT	MPI_REAL
MPI_DOUBLE MPI_DOUBLE_PRECISION	
MPI_CHAR	MPI_CHARACTER
MPI_UNSIGNED	MPI_LOGICAL
MPI_LONG	MPI_COMPLEX

Language Independent:

MPI_BYTE

MPI Types

- Explicit data description is useful:
  - Simplifies programming, e.g. send row/column of a matrix with a single call
  - Heterogeneous machines
  - May improve performance
    - Reduce memory-to-memory copies
    - Allow use of scatter/gather hardware
  - May hurt performance
    - User packing of data likely faster

MPI Standard Send

MPI_SEND(start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns (completes), the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process. The semantics of this call is up to the MPI middleware.

MPI Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (both **source** and **tag**) message is received from the system, and the buffer can be used
- **source** is rank in communicator specified by **comm**, or **MPI_ANY_SOURCE**
- **tag** is a tag to be matched on or **MPI_ANY_TAG**
- receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error
- **status** contains further information (e.g. size of message)

MPI Status Data Structure

- In C

```
MPI_Status status;
int recvd_tag, recvd_from, recvd_count;

// information from message
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, MPI_INT, &recvd_count);
```

Point-to-point Example

Process 0

```
#define TAG 999
float a[10];
int dest=1;
MPI_Send(a, 10,
MPI_FLOAT, dest, TAG,
MPI_COMM_WORLD);
```

Process 1

```
#define TAG 999
MPI_Status status;
int count;
float b[20];
int sender=0;
MPI_Recv(b, 20,
MPI_FLOAT, sender, TAG,
MPI_COMM_WORLD,
&status);
MPI_Get_count(&status,
MPI_FLOAT, &count);
```

Message Delivery

- Non-overtaking messages
  - Message sent from the same process will arrive in the order sent
- No fairness
  - On a wildcard receive, possible to receive from only one source despite other messages being sent
- Progress
  - For a pair of matched send and receives, at least one will complete independent of other messages.

Data Exchange

Process 0	Process 1
`MPI_Recv(…,1,…)`	`MPI_Recv(…,0,…)`
`MPI_Send(…,1,…)`	`MPI_Send(…,0,…)`

Deadlock. `MPI_Recv` will not return until send is posted.

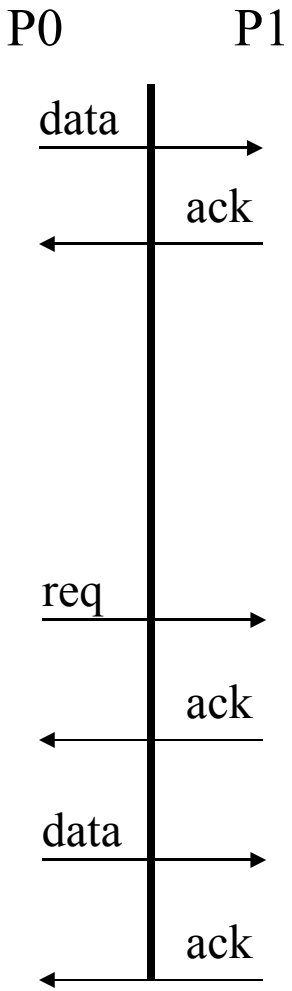
Data Exchange

Process 0	Process 1
`MPI_Send(…,1,…)`	`MPI_Send(…,0,…)`
`MPI_Recv(…,1,…)`	`MPI_Recv(…,0,…)`

May deadlock, depending on the implementation. If the messages can be buffered, program will run. Called 'unsafe' in the MPI standard.

Message Delivery

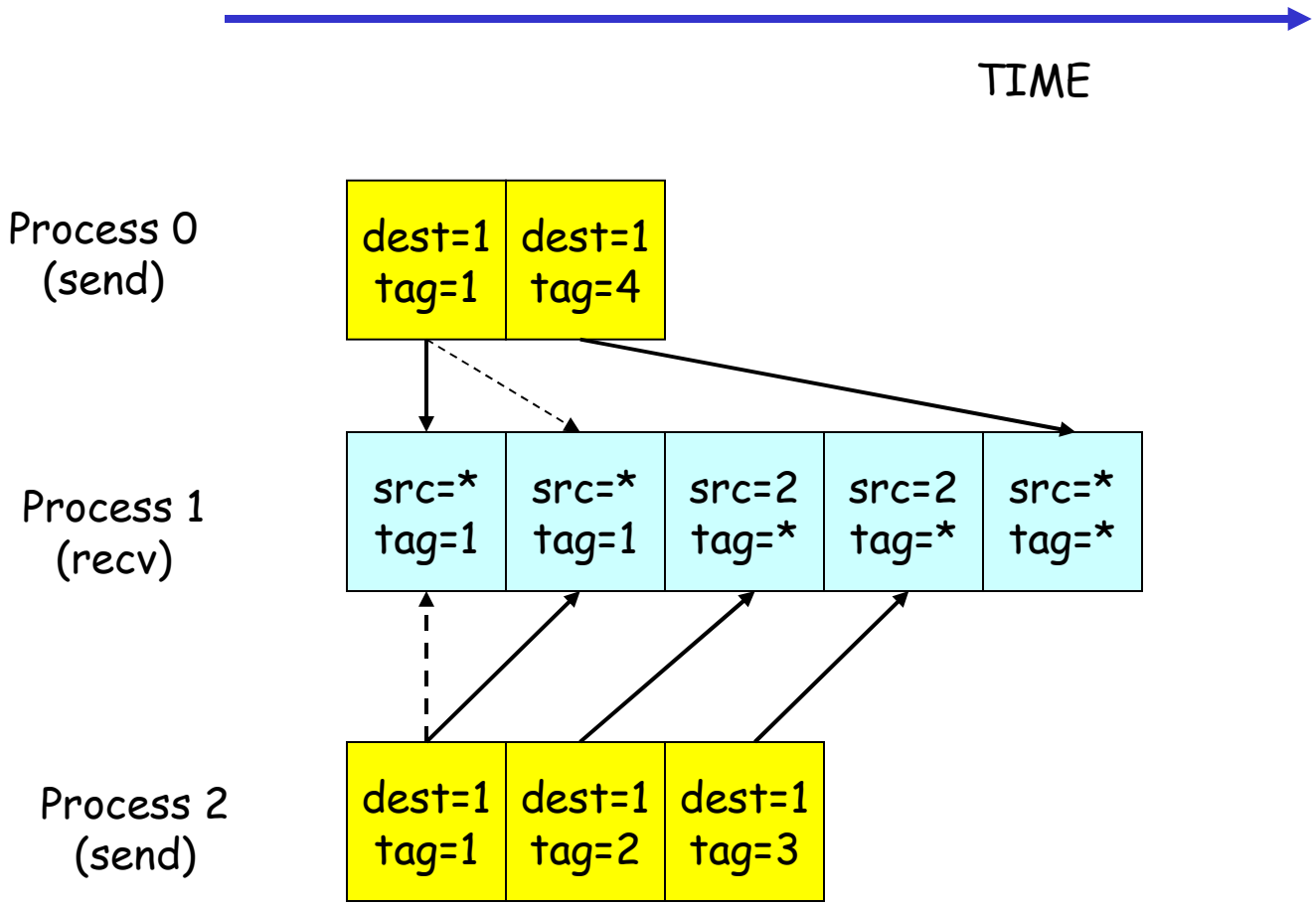
- Eager: send data immediately; store in remote buffer
  - No synchronization
  - Only one message sent
  - Data is copied
  - Uses memory for buffering (less for application)
- Rendezvous: send message header; wait for recv to be posted; send data
  - No data copy
  - More memory for application
  - More messages required
  - Synchronization (send blocks until recv posted)



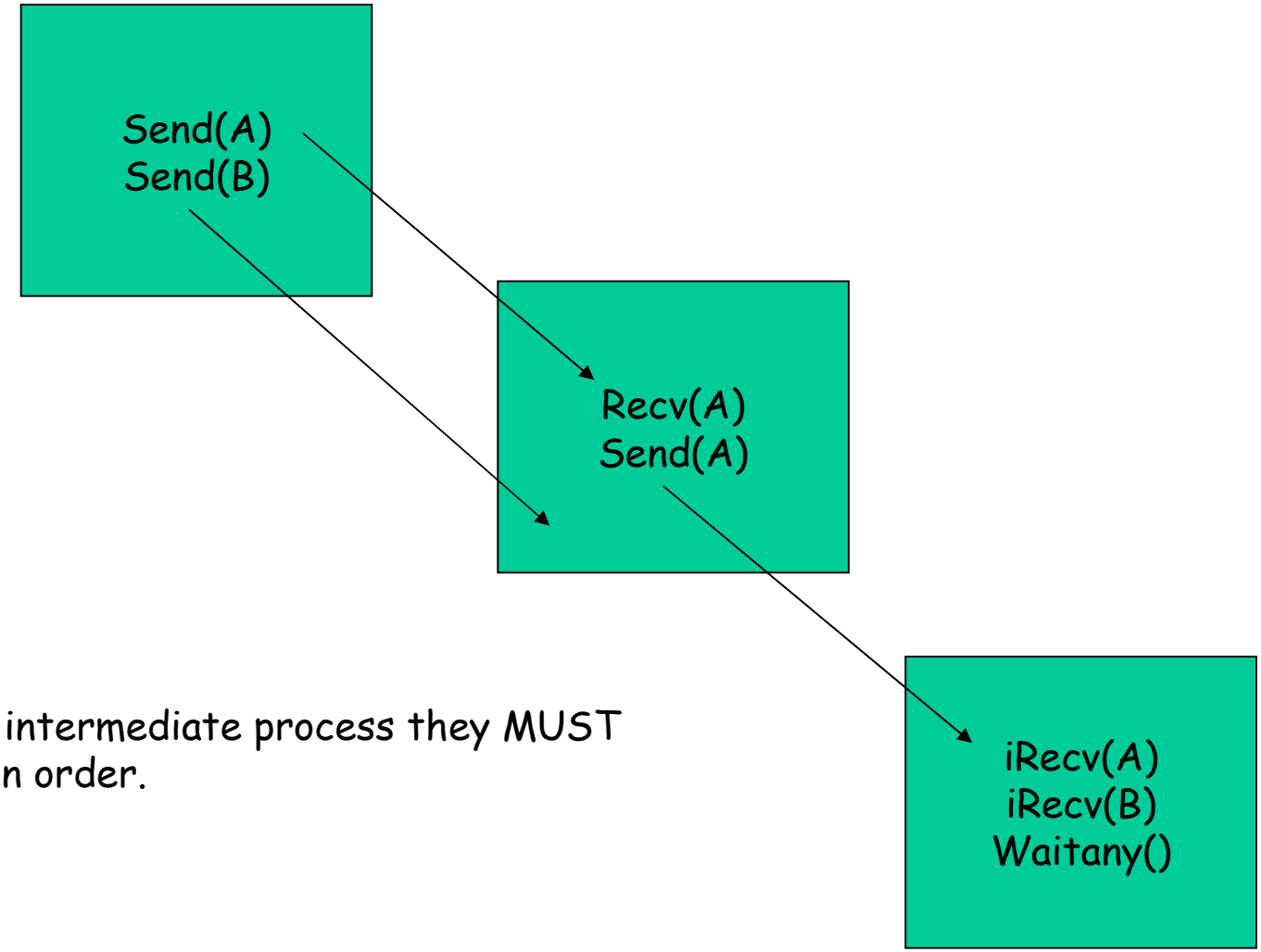
Message Delivery

- Many MPI implementations use both the eager and rendezvous methods of message delivery
- Switch between the two methods according to message size
- Often the cutover point is controllable via an environment variable, e.g. `MP_EAGER_LIMIT` and `MP_USE_FLOW_CONTROL` on the IBM SP

Messages matched in order



Message ordering



Without the intermediate process they MUST be received in order.

MPI point to point routines

MPI_Send	Standard send
MPI_Recv	Standard receive
MPI_Bsend	Buffered send
MPI_Rsend	Ready send
MPI_Ssend	Synchronous send
MPI_Ibsend	Nonblocking, buffered send
MPI_Irecv	Nonblocking receive
MPI_Irsend	Nonblocking, ready send
MPI_Isend	Nonblocking send
MPI_Issend	Nonblocking synchronous send
MPI_Sendrecv	Exchange
MPI_Sendrecv_replace	Exchange, same buffer
MPI_Start	Persistent communication

Communication Modes

- Standard
  - Usual case (system decides)
  - **MPI_Sen**d, **MPI_Isend**
- Synchronous
  - The operation does not complete until a matching receive has started copying data into its receive buffer. (no buffers)
  - **MPI_Ssen**d, **MPI_Issend**
- Ready
  - Matching receive already posted. (0-copy)
  - **MPI_Rsen**d, **MPI_Irsend**
- Buffered
  - Completes after being copied into user provided buffers (Buffer_attach, Buffer_detach calls)
  - **MPI_Bsen**d, **MPI_Ibsend**

Point to point with modes

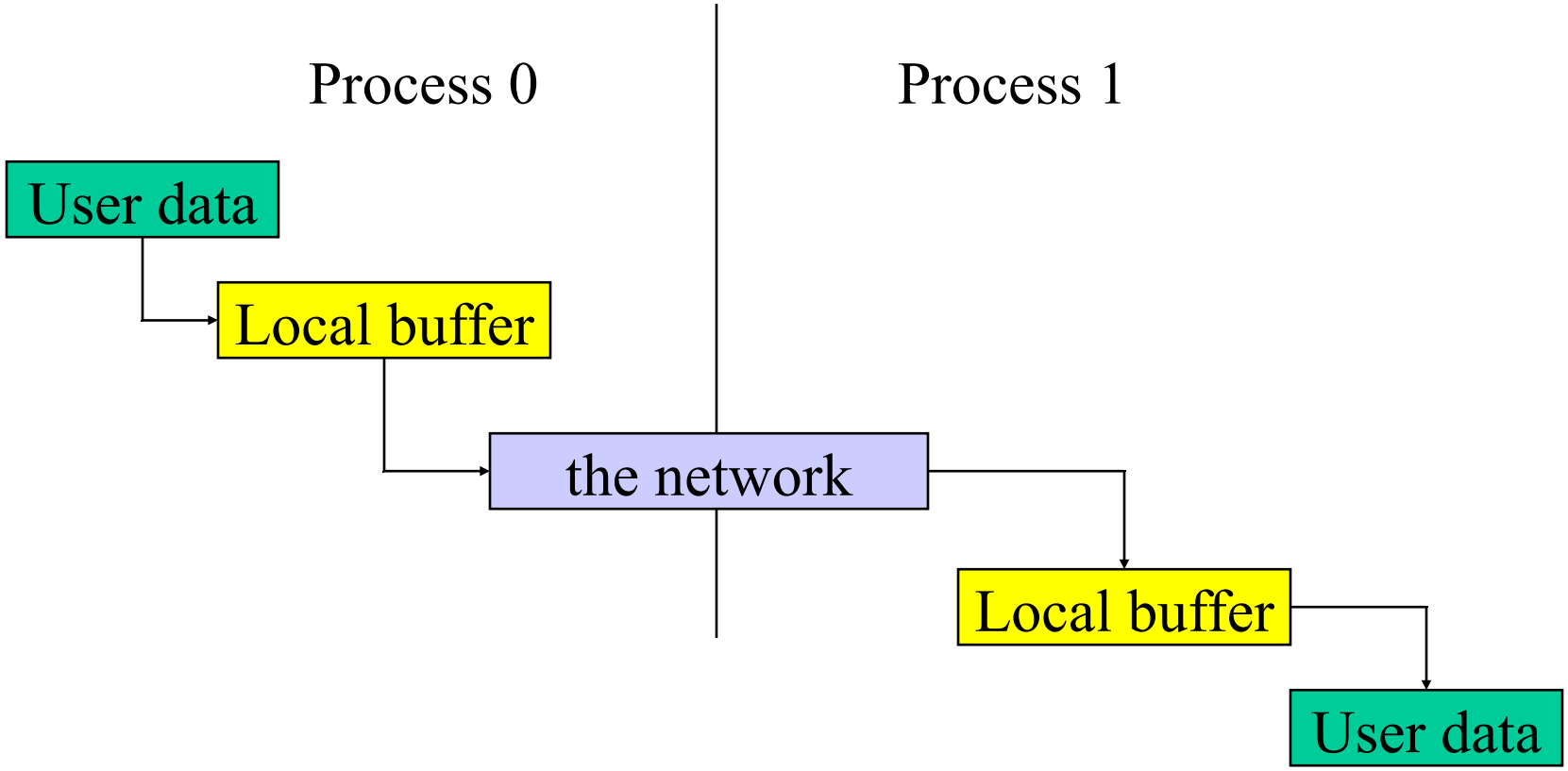
MPI_[SBR]send(start, count, datatype, dest, tag, comm)

There is only one mode for receive!

Buffering

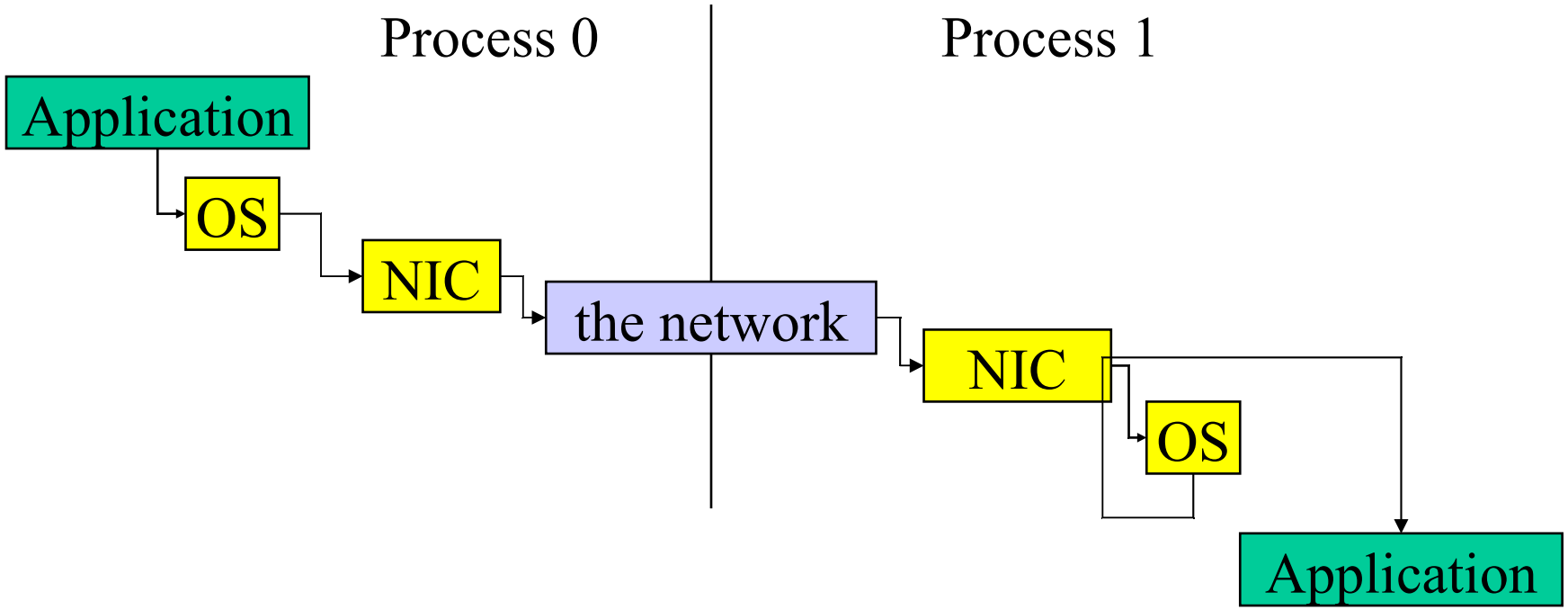
Usual type of scenario

- User level buffering in the application and buffering in the middleware or system



System buffers

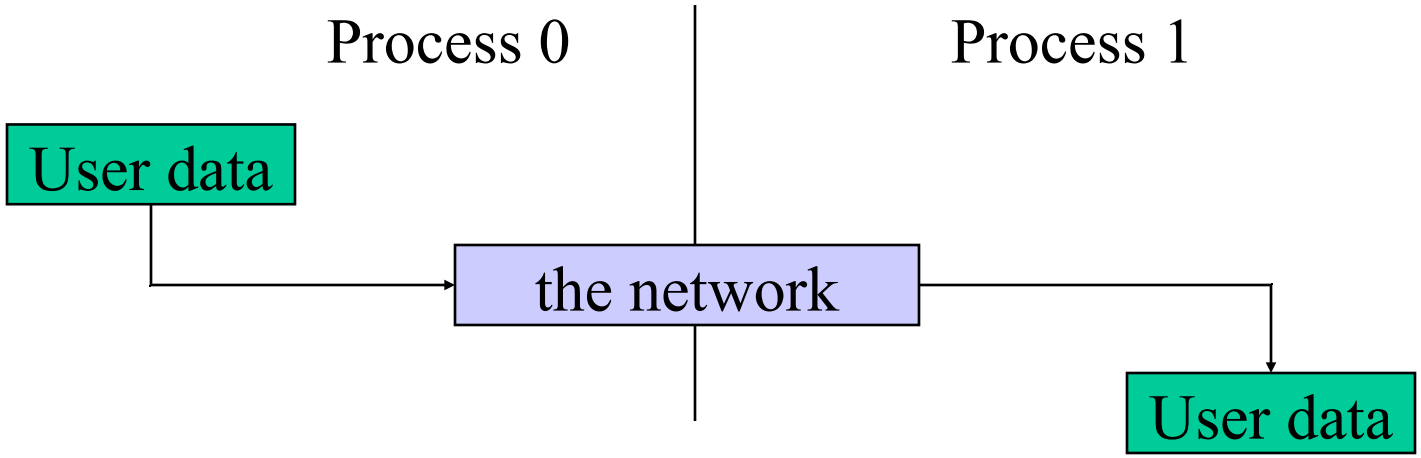
- System buffering depends on OS and NIC card



May provide varying amount of buffering depending on system. MPI tries to be independent of buffering.

Some machines by-pass the system

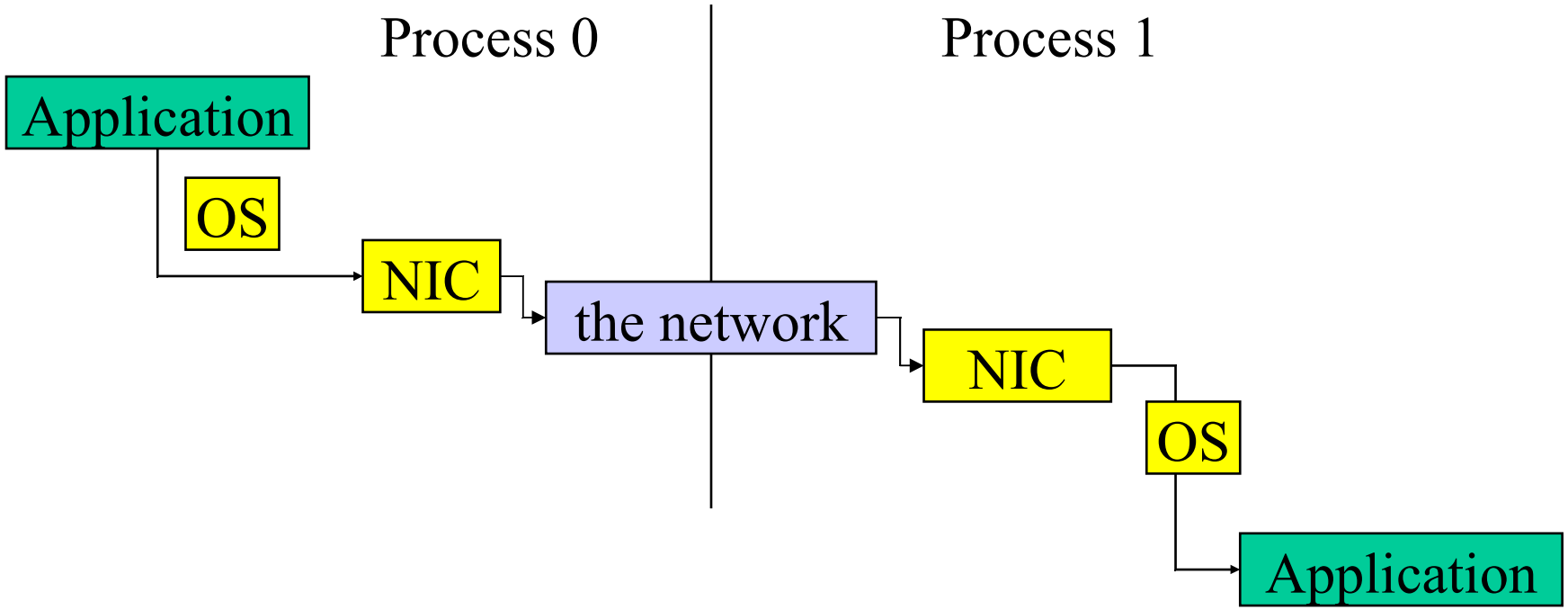
- Avoids the OS, no buffering except in network



This requires that **`MPI_Send`** wait on delivery, or that **`MPI_Send`** return before transfer is complete, and we wait later.

Some machines by-pass the OS

- Avoids the OS, zero copy
  - Zero copy may be either on the send and/or receive



Send side easy, but the receive side can only work if the receive buffer is known

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on. (Posts a send/receive)

```
MPI_Request request;
MPI_Isend(start, count, datatype,
    dest, tag, comm, &request)
MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request)
MPI_Wait(&request, &status)
```

- One can also test without waiting:

```
MPI_Test(request, &flag, status)
```

Example

```
#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)
```

Using Non-Blocking Send

Also possible to use non-blocking send:

- "status" argument to **MPI_Wait** doesn't return useful info here.

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

Non-Blocking Gotchas

- Obvious caveats:
  - 1. You may not modify the buffer between Isend() and the corresponding Wait(). Results are undefined.
  - 2. You may not look at or modify the buffer between Irecv() and the corresponding Wait(). Results are undefined.
  - 3. You may not have two pending Irecv()s for the same buffer.
- Less obvious:
  - 4. You may not look at the buffer between Isend() and the corresponding Wait().
  - 5. You may not have two pending Isend()s for the same buffer.
- **Why the isend() restrictions?**
  - Restrictions give implementations more freedom, e.g.,
    - Heterogeneous computer with differing byte orders
    - Implementation swap bytes in the original buffer

Multiple Completions

- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_of_requests,
   array_of_statuses)
MPI_Waitany(count, array_of_requests,
   &index, &status)
MPI_Waitsome(count, array_of_requests,
   array_of indices, array_of_statuses)
```

- There are corresponding versions of `test` for each of these.

Multiple completion

- Source of non-determinism (new issues fairness?), process what is ready first
- Latency hiding, parallel slack
- Still need to poll for completion, {do some work; check for comm}*
- Alternative: multiple threads or co-routine like support

Buffered mode

Buffered Mode

- When MPI_Isend is awkward to use (e.g. lots of small messages), the user can provide a buffer for the system to store messages that cannot immediately be sent.

```
int bufsize;
char *buf = malloc( bufsize );
MPI_Buffer_attach( buf, bufsize );
...
MPI_Bsend( ... same as MPI_Send ... )
...
MPI_Buffer_detach( &buf, &bufsize );
```

- **MPI_Buffer_detach** waits for completion.
- Performance depends on MPI implementation and size of message.

Careful using buffers

- What is wrong with this code?

```
MPI_Buffer_attach(buf,bufsize+MPI_BSEND_OVERHEAD)

for (i=1,i<n, i++) {

    ...
    MPI_Bsend( bufsize bytes ... )
    ...
    Enough MPI_Recvs( )
}
MPI_Buffer_detach(buff_addr, bufsize)
```

Buffering is limited

- Processor 0
  i=1
  MPI_Bsend
  MPI_Recv
  i=2
  MPI_Bsend

- i=2 Bsend fails because first Bsend has not been able to deliver the data

- Processor 1
  i=1
  MPI_Bsend
  … delay due to computing, process scheduling,…
  MPI_Recv

Correct Use of MPI_Bsend

- Fix: Attach and detach buffer in loop

- ```
  MPI_Buffer_attach( buf, bufsize+MPI_BSEND_OVERHEAD)
  for (i=1,i<n, i++) {
      ...
      MPI_Bsend( bufsize bytes ... )
      ...
      Enough MPI_Recvs( )
      MPI_Buffer_detach(buf_addr, bufsize)
  }
  ```

- Buffer detach will wait until messages have been delivered
  ```

Ready send

- Receive side zero copy
- May avoid an extra copy that can happen on unexpected messages
- Sometimes know this because of protocol

```
P0:
iRecv( 0 )
Ssend(1)

P1:
Recv(1)
Rsend(0)
```

Other Point-to Point Features

- **`MPI_Sendrecv`**
- **`MPI_Sendrecv_replace`**
- **`MPI_Cancel`**
  - Useful for multi-buffering, multiple outstanding sends/receives

MPI_Sendrecv

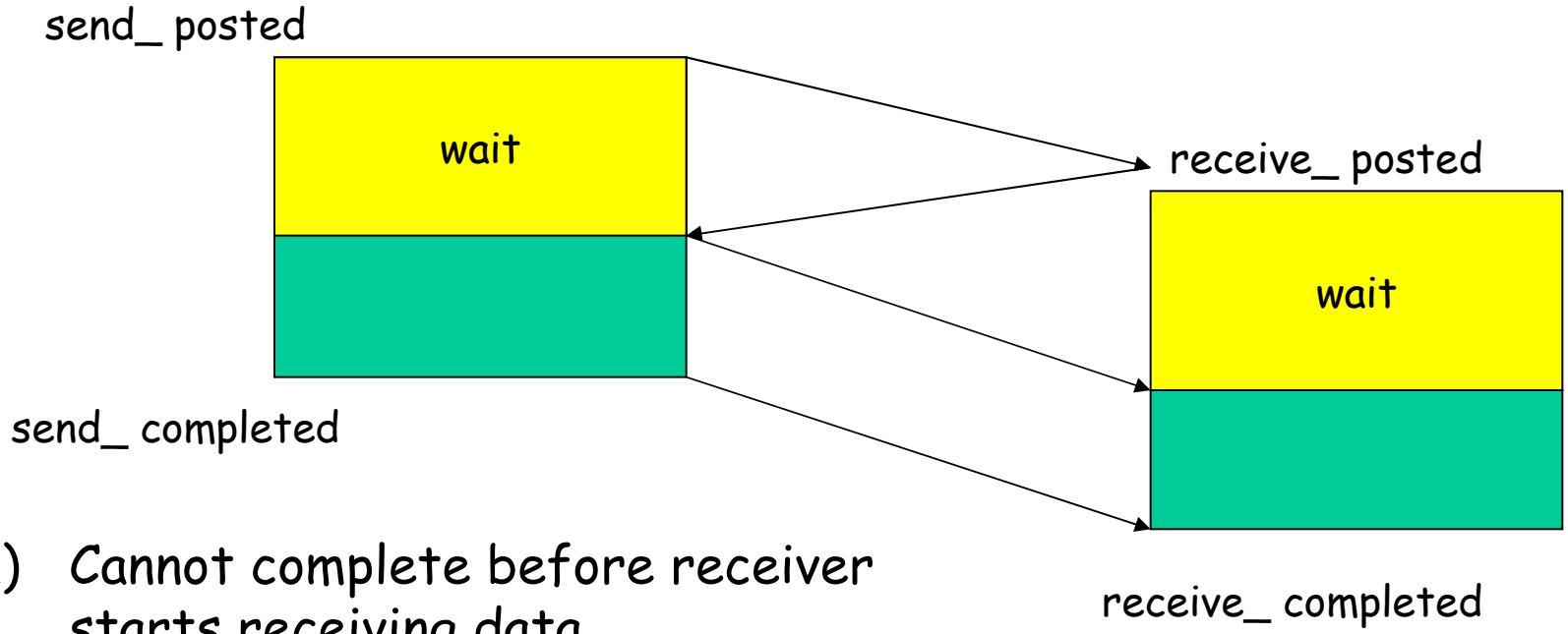
- Allows simultaneous send and receive
- Everything else is general.
  - Send and receive datatypes (even type signatures) may be different
  - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, …)
  - More general than "send left"

Process 0	Process 1
`SendRecv(1)`	**`SendRecv(0)`**

Safety property

- An MPI program is considered safe, if the program executed correctly when all point to point communications are replaced by synchronous communication

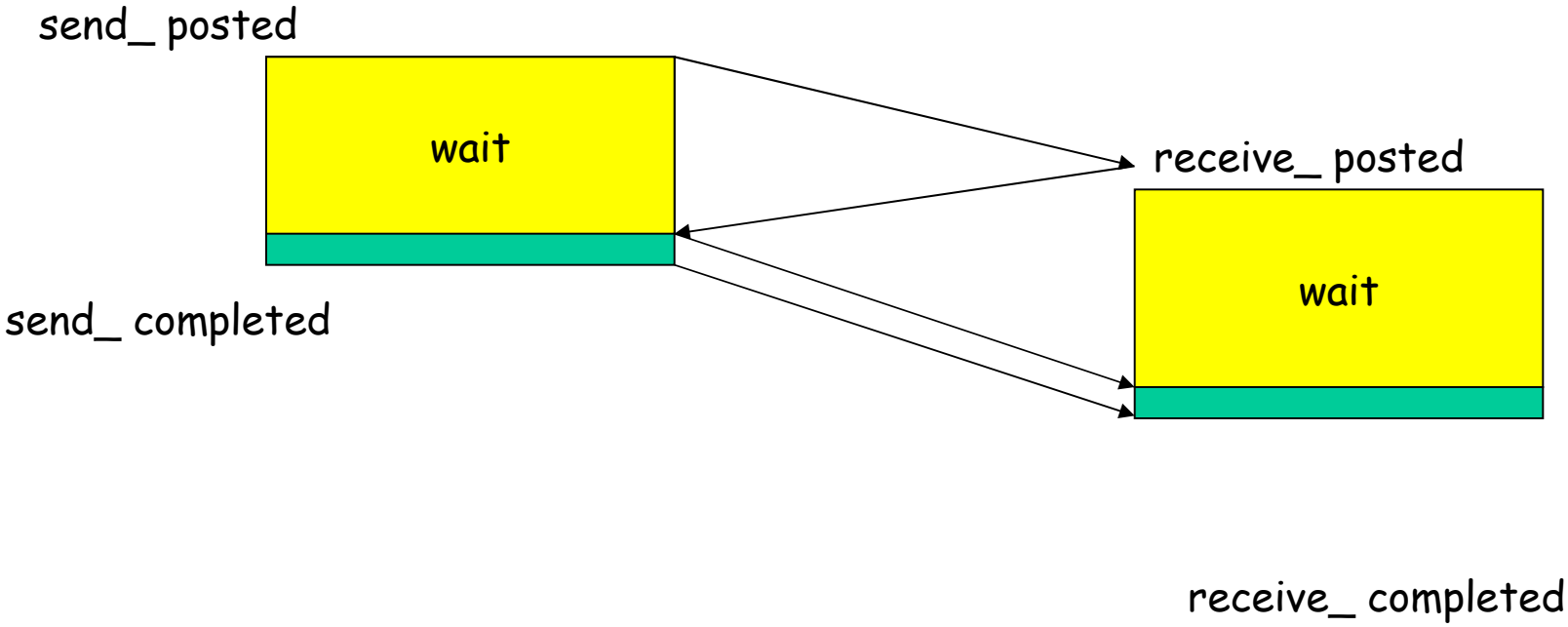
Synchronous send-receive



a) Cannot complete before receiver starts receiving data,
b) Cannot complete until buffer is emptied

Advantage: one can reason about the state of the receiver

Synchronous send-receive



Is this correct?

Deadlock

- Consequence of insufficient buffering
- Affects the portability of code

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
`Send(1)`	**`Send(0)`**
`Recv(1)`	**`Recv(0)`**

- This is called "unsafe" because it depends on the availability of system buffers

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	**`Recv(0)`**
`Recv(1)`	**`Send(0)`**

Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	**`Sendrecv(0)`**

More Solutions to the "unsafe" Problem

- Supply own space as buffer for send

Process 0	Process 1
`Bsend(1)`	**`Bsend(0)`**
`Recv(1)`	**`Recv(0)`**

Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	**`Isend(0)`**
`Irecv(1)`	**`Irecv(0)`**
`Waitall`	**`Waitall`**

Persistent Communication

Persistent Operations

- Many applications use the same communications operations over and over
- Same parameters used many time

```
for( i=1,i<n, i++){
    MPI_Isend(…)
    MPI_Irecv(…)
    MPI_Waitall(…)
}
```

- MPI provides persistent operations to make this more efficient
  - Reduce error checking of args (needed only once)
  - Implementation may be able to make special provision for repetitive operation (though none do to date)
  - All persistent operations are nonblocking

Persistent Operations and Networks

- Zero-copy and "OS bypass"
  - Provides direct communication between designated user-buffers without OS intervention
- Requires registration of memory with OS; may be a limited resource (pinning pages)
  - Examples are UNET, VIA, LAPI
- persistent communication is a good match to this capability

Using Persistent Operations

- Replace

```
MPI_Isend( buf, count, datatype, tag, dest, comm,
                &request )
```

  with

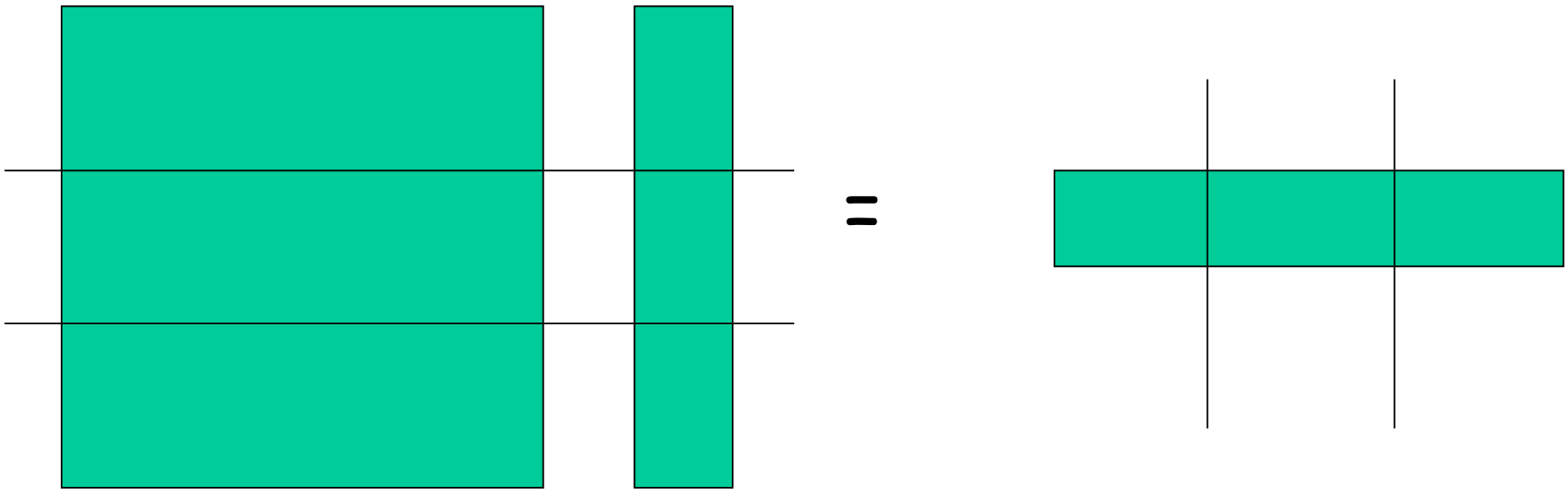
```
MPI_Send_init( buf, count, datatype, tag, dest, comm,
                    &request )
MPI_Start(request)
```

- MPI_Irecv with MPI_Recv_init, MPI_Irsend with MPI_Rsend_init, etc.
- Wait/test requests just like other nonblocking requests, once completed you call start again.
- Free requests when done with MPI_Request_free

Example: Sparse Matrix-Vector Product

- Many iterative methods require matrix-vector products
- Same operation (with same arguments) performed many times (vector updated in place)
- Divide sparse matrix into groups of rows by process: e.g., rows 1-10 on process 0, 11-20 on process 1. Use same division for vector.
- To perform matrix-vector product, get elements of vector on different processes with Irecv/Isend/Waitall

Matrix Vector Multiply



Changing MPI Nonblocking to MPI Persistent

- For i=1 to N
  ! Exchange vector information
  MPI_Isend(…)
  MPI_Irecv(…)
  MPI_Waitall(…)
- Replace with
  MPI_Send_init(…)
  MPI_Recv_init(…)

```
for i=1 to N
{
   MPI_Startall( 2, requests)
   MPI_Waitall( 2, requests, statuses)
}
MPI_Request_free( request(1))
MPI_Request_free( request(2))
```

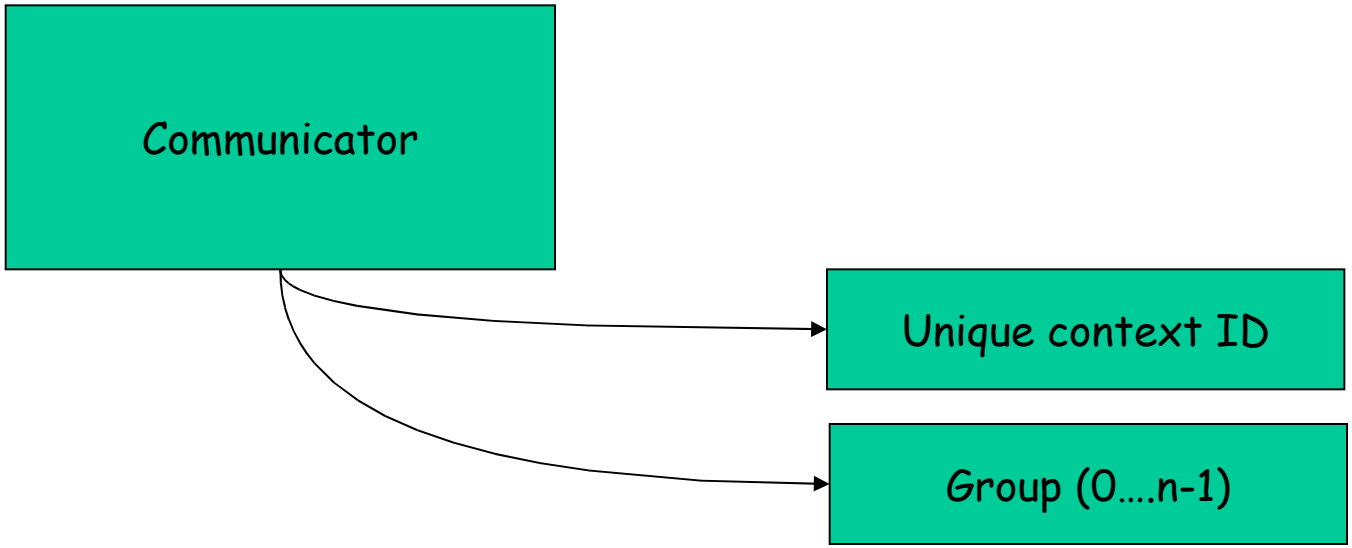
Identical arguments

Context and communicators

Communicators

http://www.linux-mag.com/id/1412

Communicator



Communicators

- All MPI communication is based on a communicator which contains a context and a group
- Contexts define a safe communication space for message-passing
- Contexts can be viewed as system-managed tags
- Contexts allow different libraries to co-exist
- The group is just a set of processes
- Processes are always referred to by unique rank in group

Pre-Defined Communicators

- MPI-1 supports three pre-defined communicators:

  MPI_COMM_WORLD

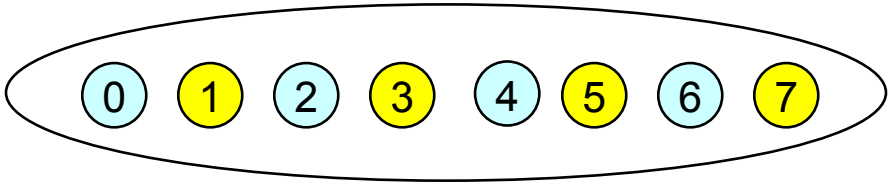
  MPI_COMM_NULL

  MPI_COMM_SELF (only returned by some functions, or in initialization. NOT used in normal communications)

- Only **MPI_COMM_WORLD** is used for communication
- Predefined communicators are needed to "get things going" in MPI

Uses of MPI_COMM_WORLD

- Contains all processes available at the time the program was started
- Provides initial safe communication space
- Simple programs communicate with MPI_COMM_WORLD
  - Even complex programs will use MPI_COMM_WORLD for most communications
- Complex programs duplicate and subdivide copies of MPI_COMM_WORLD
  - Provides a global communicator for forming smaller groups or subsets of processors for specific tasks



MPI_COMM_WORLD

Subdividing a Communicator with MPI_Comm_split

- **MPI_COMM_SPLIT** partitions the group associated with the given communicator into disjoint subgroups
- Each subgroup contains all processes having the same value for the argument color

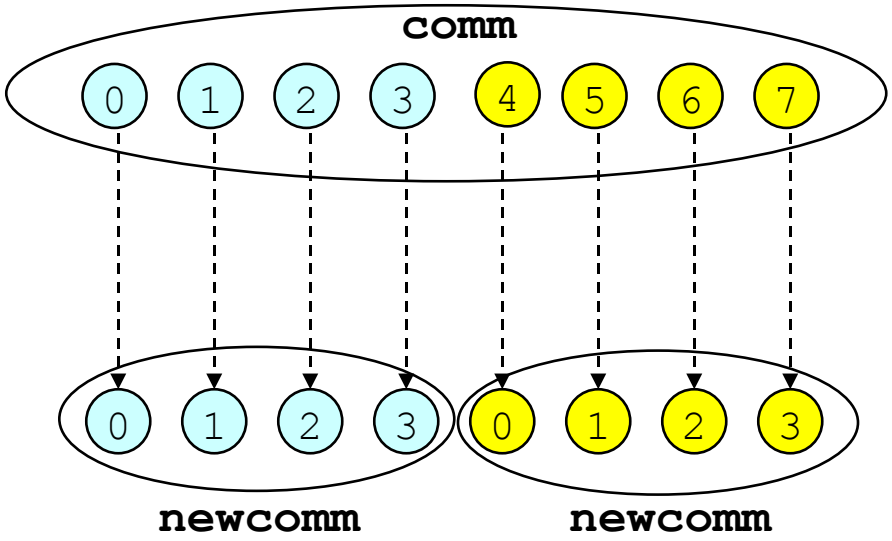
```
int MPI_Comm_split( MPI_Comm comm, int color,
                    int key, MPI_Comm *newcomm)
```

- Within each subgroup, processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in old communicator

Subdividing a Communicator

- To divide a communicator into two non-overlapping groups

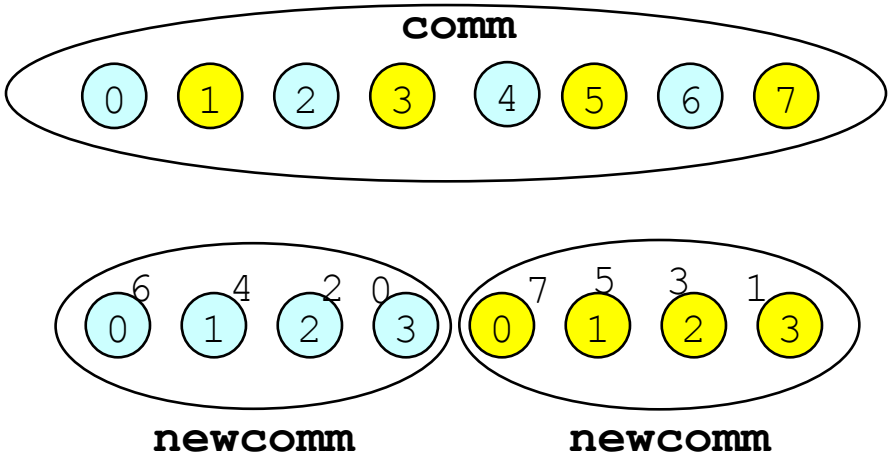
```
color = (rank < size/2) ? 0 : 1 ;
MPI_Comm_split(comm, color, 0, &newcomm) ;
```



Subdividing a Communicator

- To divide a communicator such that
  - all processes with even ranks are in one group
  - all processes with odd ranks are in the other group
  - maintain the reverse order by rank

```
color = (rank % 2 == 0) ? 0 : 1 ;
key = size - rank ;
MPI_Comm_split(comm, color, key, &newcomm) ;
```



Example of MPI_Comm_split

```
int row_comm, col_comm;
int myrank, size, P, Q, myrow, mycol;
P = 4;
Q = 3;
MPI_InitT(ierr);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

/* Determine row and column position */
myrow = myrank/Q;
mycol = myrank % Q;
/* Split comm into row and column comms */
MPI_Comm_split(MPI_COMM_WORLD, myrow, mycol, row_comm);
MPI_Comm_split(MPI_COMM_WORLD, mycol, myrow, col_comm);
```

MPI_Comm_Split

- Collective call for the old communicator
- Nodes that don't wish to participate can call the routine with MPI_UNDEFINED as the colour argument (it will return MPI_COMM_NULL)

Groups

- Group operations are all local operations. Basically, operations on maps (sequences with unique values).
- Like communicators, work with handles to the group
- Group underlying a communicator

Group Manipulation Routines

- To obtain an existing group, use

```
MPI_group group;
MPI_Comm_group ( comm, &group )
```

- To free a group, use

```
MPI_Group_free (&group)
```

- A new group can be created by specifying the members to be included/excluded from an existing group using the following routines
  - **MPI_Group_incl**: specified members are included
  - **MPI_Group_excl**: specified members are excluded
  - **MPI_Group_range_incl** and **MPI_Group_range_excl**: a range of members are included or excluded
  - **MPI_Group_union** and **MPI_Group_intersection**: a new group is created from two existing groups
- Other routines

  MPI_Group_compare, MPI_Group_translate_ranks

Subdividing a Communicator with MPI_Comm_create

- Creates new communicators having all the processes in the specified group with a new context
- The call is erroneous if all the processes do not provide the same handle
- **MPI_COMM_NULL** is returned to processes not in the group
- **MPI_COMM_CREATE** is useful if we already have a group, otherwise a group must be built using the group manipulation routines

```
int MPI_Comm_create( MPI_Comm comm, MPI_Group group,
MPI_Comm *newcomm )
```

Context

Contexts (hidden in communicators)

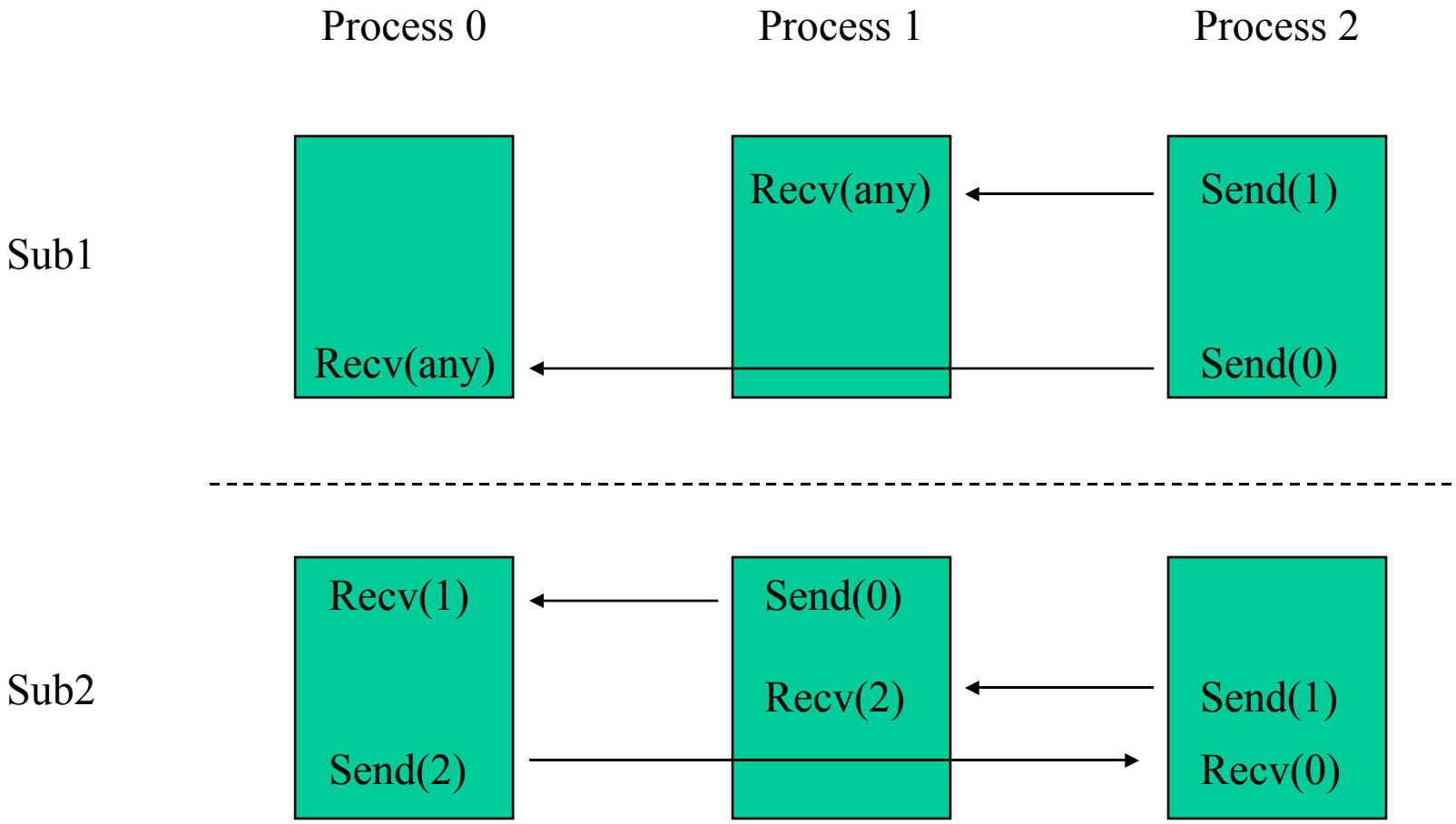
- Parallel libraries require isolation of messages from one another and from the user that cannot be adequately handled by tags.
- The context hidden in a communicator provides this isolation
- The following examples are due to Marc Snir.
  - Sub1 and Sub2 are from different libraries

```
Sub1();
Sub2();
```

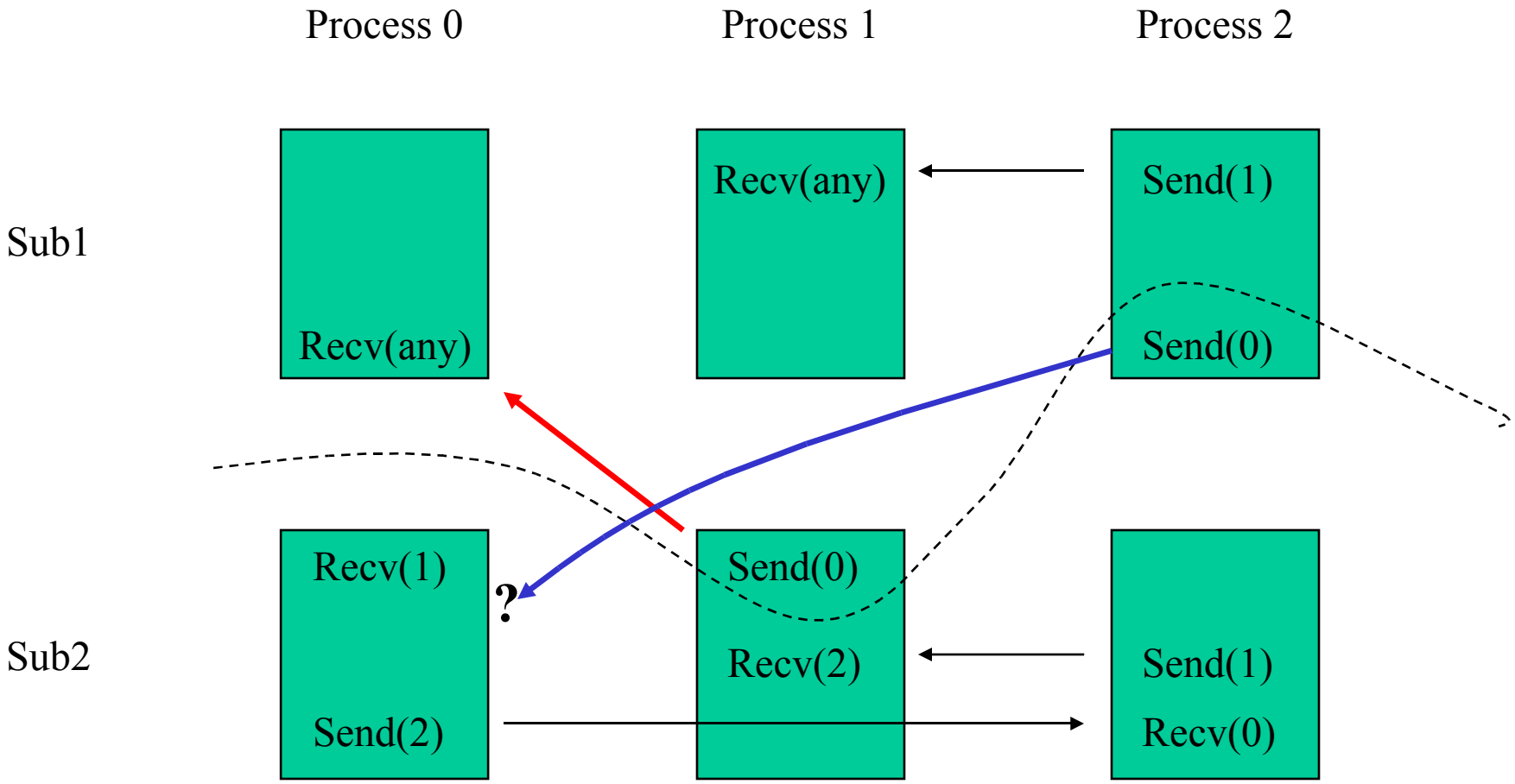
  - Sub1a and Sub1b are from the same library

```
Sub1a();
Sub2();
Sub1b();
```

Correct Execution of Library Calls

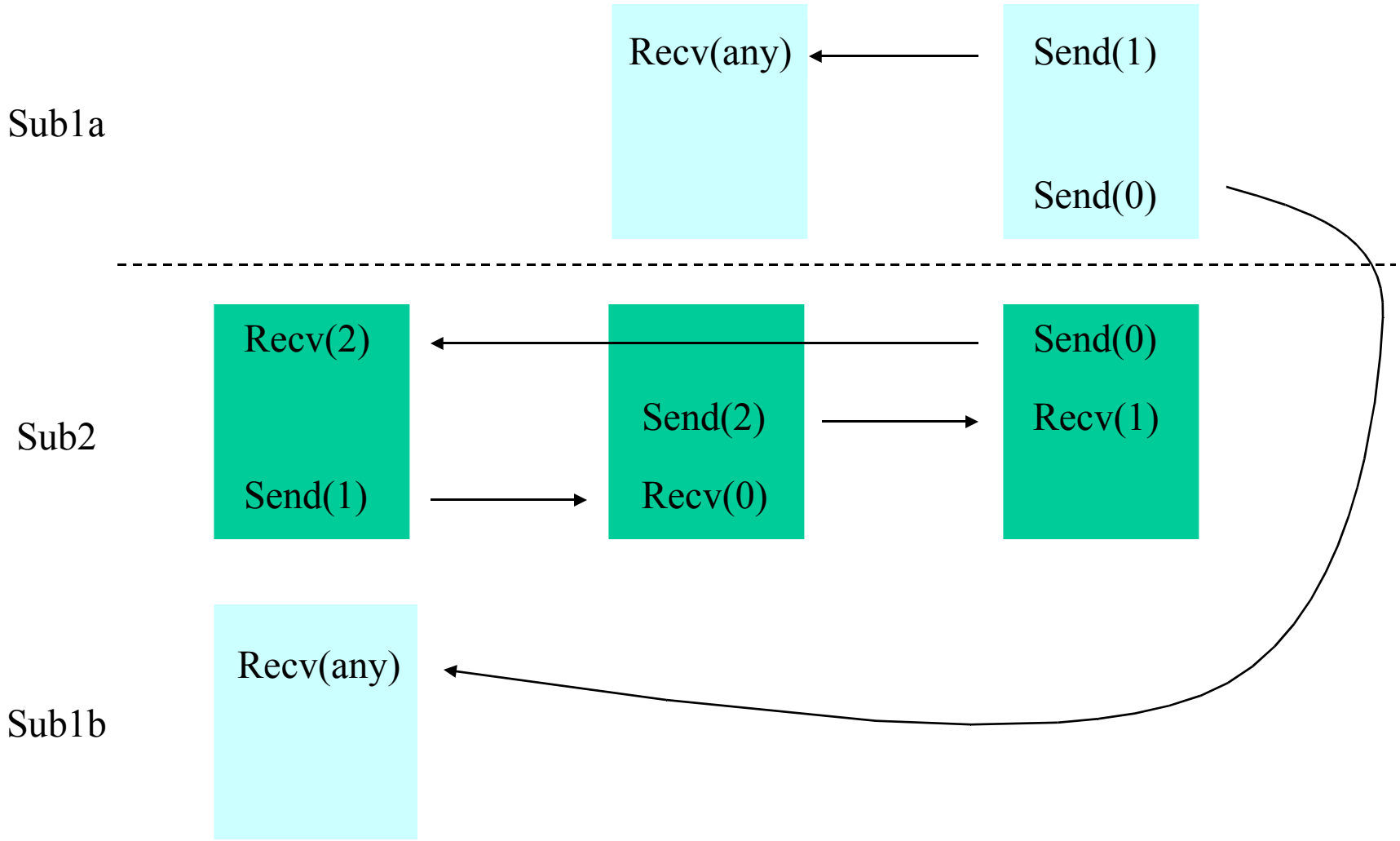


Incorrect Execution of Library Calls

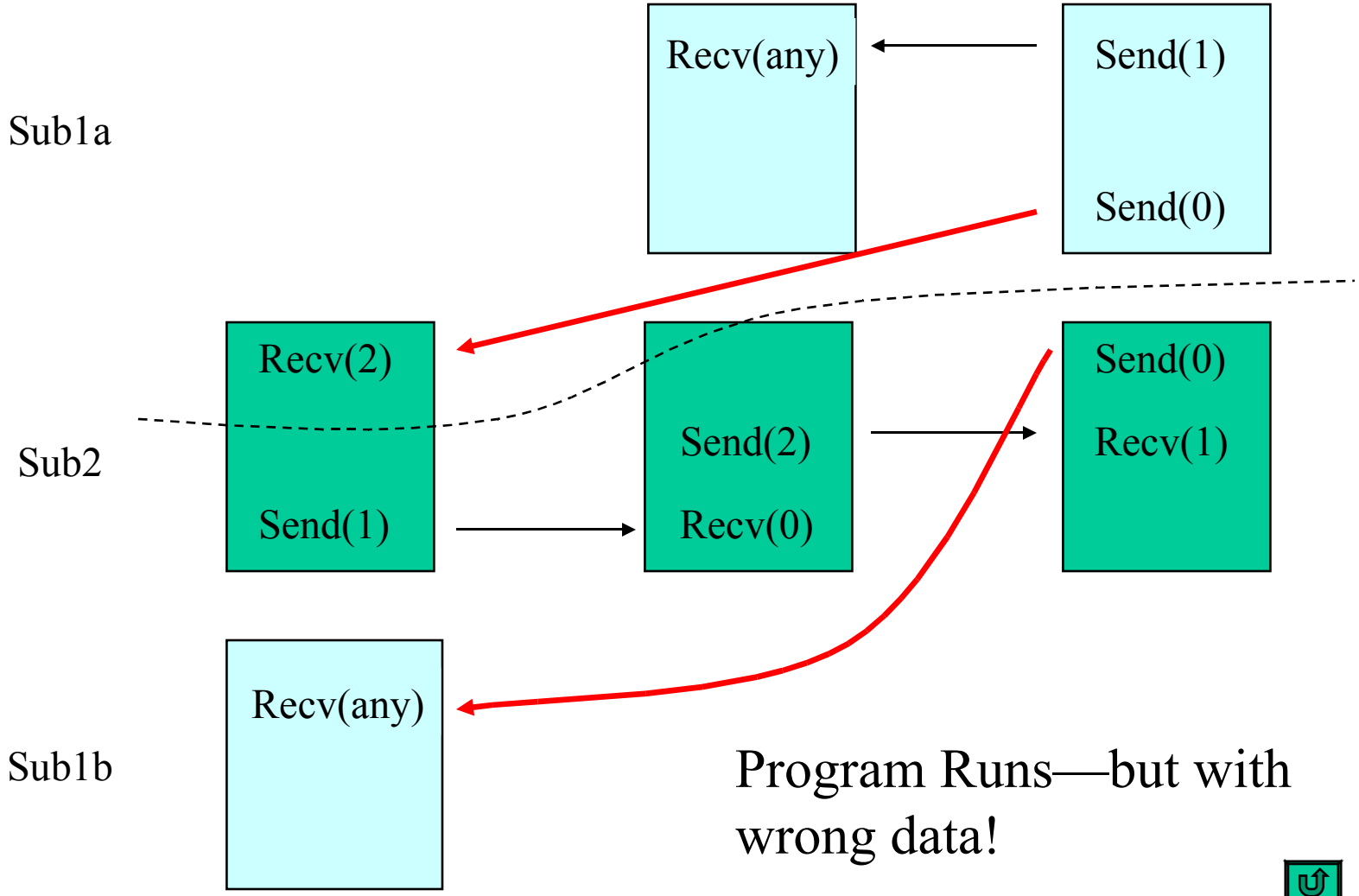


Program hangs (Recv(1) never satisfied)

Correct Execution of Library Calls with Pending Communication



Incorrect Execution of Library Calls with Pending Communication



Program Runs—but with wrong data!

Inter-communicators

Inter-communicators (MPI-1)

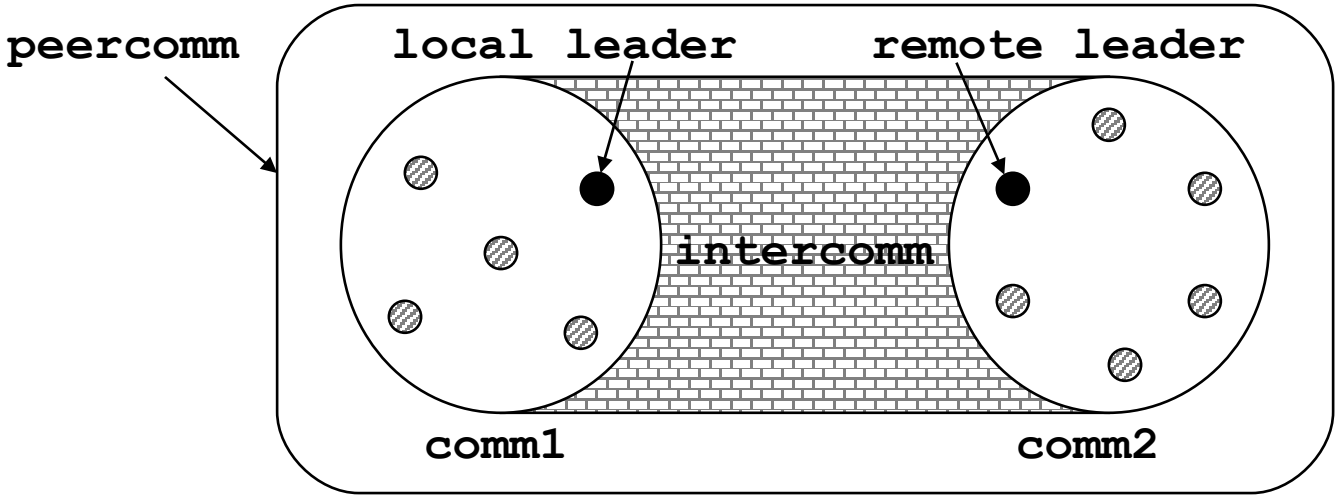
- Intra-communication: communication between processes that are members of the same group
- Inter-communication: communication between processes in different groups (say, local group and remote group)
- Both inter- and intra-communication have the same syntax for point-to-point communication
- Inter-communicators can be used only for point-to-point communication (no collective and topology operations with inter-communicators)
- A target process is specified using its rank in the remote group
- Inter-communication is guaranteed not to conflict with any other communication that uses a different communicator

Inter-communicator Accessor Routines

- To determine whether a communicator is an intra-communicator or an inter-communicator
  - **MPI_Comm_test_inter(comm, &flag)**
    flag = true, if **comm** is an inter-communicator
    flag = false, otherwise
- Routines that provide the local group information when the communicator used is an inter-communicator
  - **MPI_COMM_SIZE, MPI_COMM_GROUP, MPI_COMM_RANK**
- Routines that provide the remote group information for inter-communicators
  - **MPI_COMM_REMOTE_SIZE, MPI_COMM_REMOTE_GROUP**

Inter-communicator Create

- **MPI_INTERCOMM_CREATE** creates an inter-communicator by binding two intra-communicators
  - **MPI_INTERCOMM_CREATE(local_comm,local_leader, peer_comm, remote_leader,tag, intercomm)**



Inter-communicator Create (cont)

- Both the local and remote leaders should
  - belong to a peer communicator
  - know the rank of the other leader in the peer communicator
- Members of each group should know the rank of their leader
- An inter-communicator create operation involves
  - collective communication among processes in local group
  - collective communication among processes in remote group
  - point-to-point communication between local and remote leaders

```
MPI_SEND(..., 0, intercomm)
MPI_RECV(buf, ..., 0, intercomm);
MPI_BCAST(buf, ..., localcomm);
```

Note that the source and destination ranks are specified w.r.t the other communicator

MPI Collectives

MPI Collective Communication

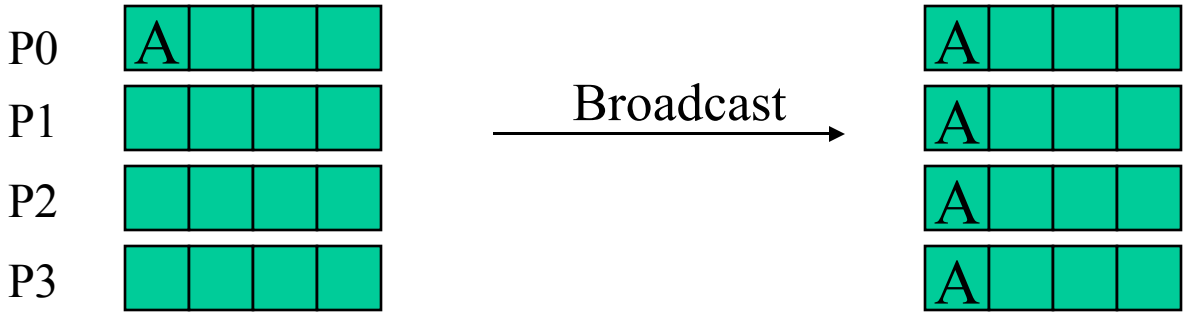
- Communication and computation is coordinated among a group of processes in a communicator.
- Groups and communicators can be constructed "by hand" or using topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations.
- Three classes of operations: synchronization, data movement, collective computation.

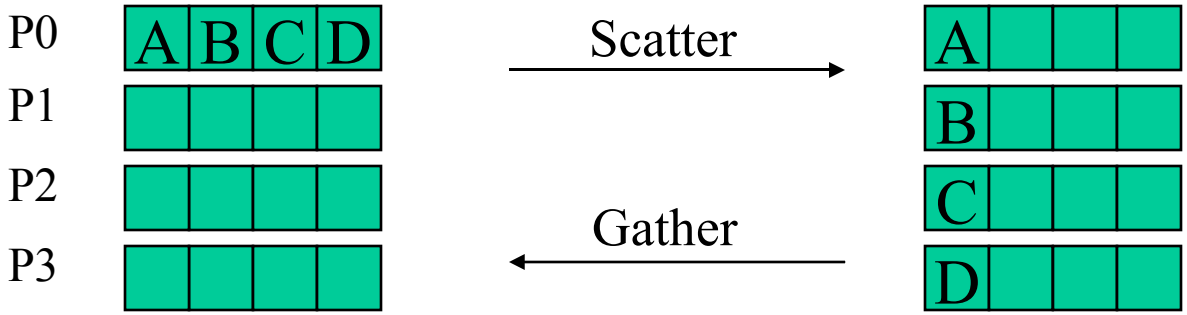
Synchronization

- **`MPI_Barrier( comm )`**

- Blocks until all processes in the group of the communicator **`comm`** call it.

Collective Data Movement

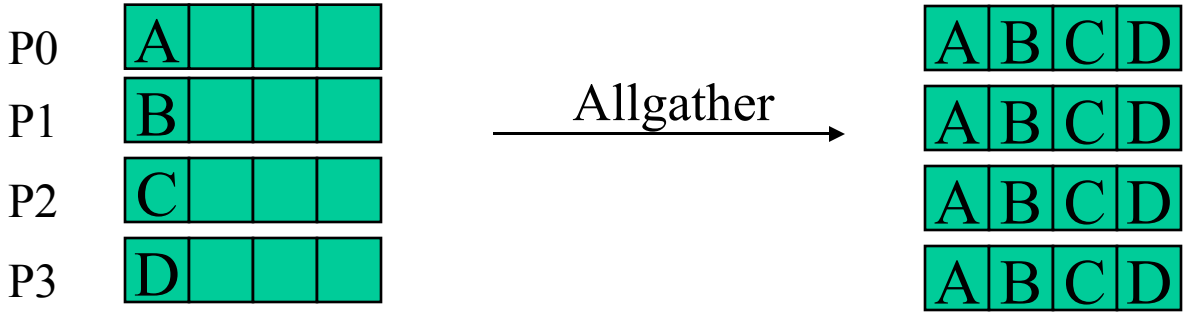


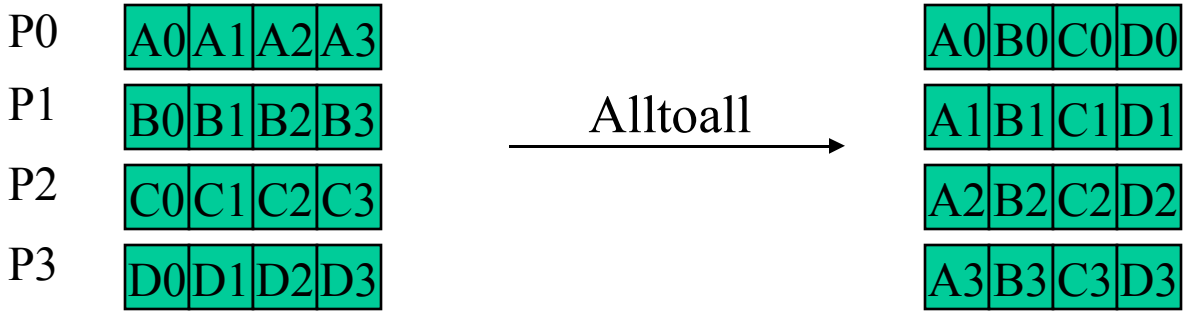


Comments on Broadcast

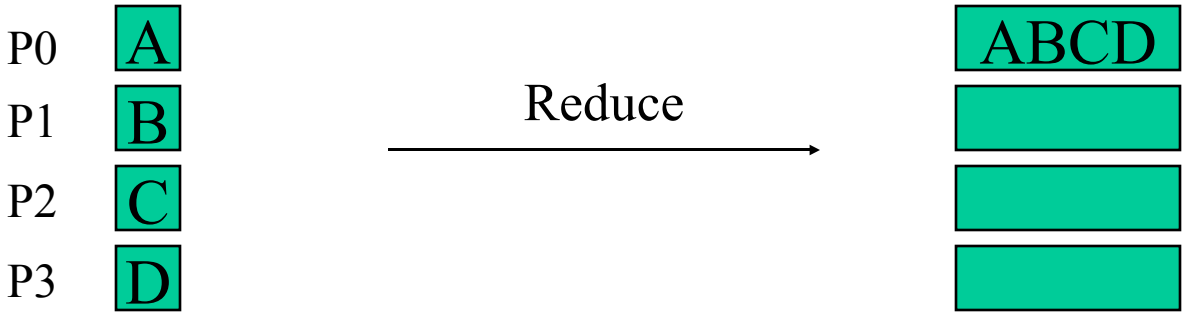
- All collective operations must be called by all processes in the communicator
- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
  - MPI_Bcast is not a "multi-send"
  - "root" argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive
- Example of orthogonally of the MPI design: MPI_Recv need not test for "multi-send"

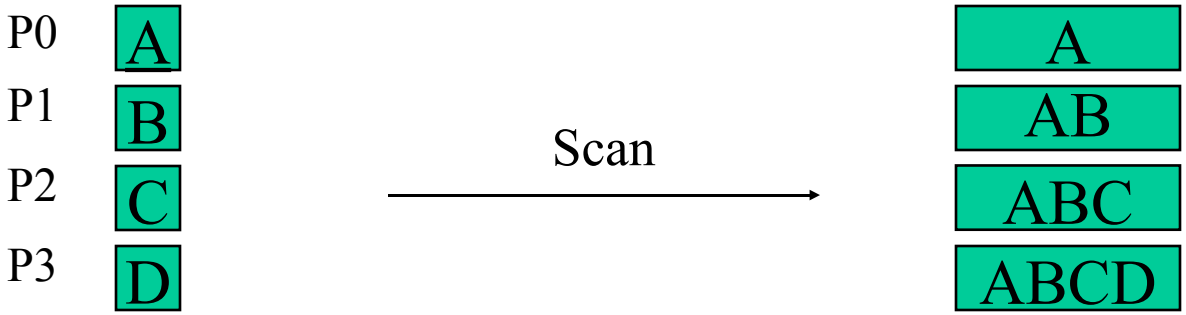
More Collective Data Movement





Collective Computation





MPI Collective Routines

- Many Routines: **`Allgather`**, **`Allgatherv`**, **`Allreduce`**, **`Alltoall`**, **`Alltoallv`**, **`Bcast`**, **`Gather`**, **`Gatherv`**, **`Reduce`**, **`Reduce_scatter`**, **`Scan`**, **`Scatter`**, **`Scatterv`**
- **`All`** versions deliver results to all participating processes.
- V versions allow the chunks to have different sizes.
- **`Allreduce`**, **`Reduce`**, **`Reduce_scatter`**, and **`Scan`** take both built-in and user-defined combiner functions.

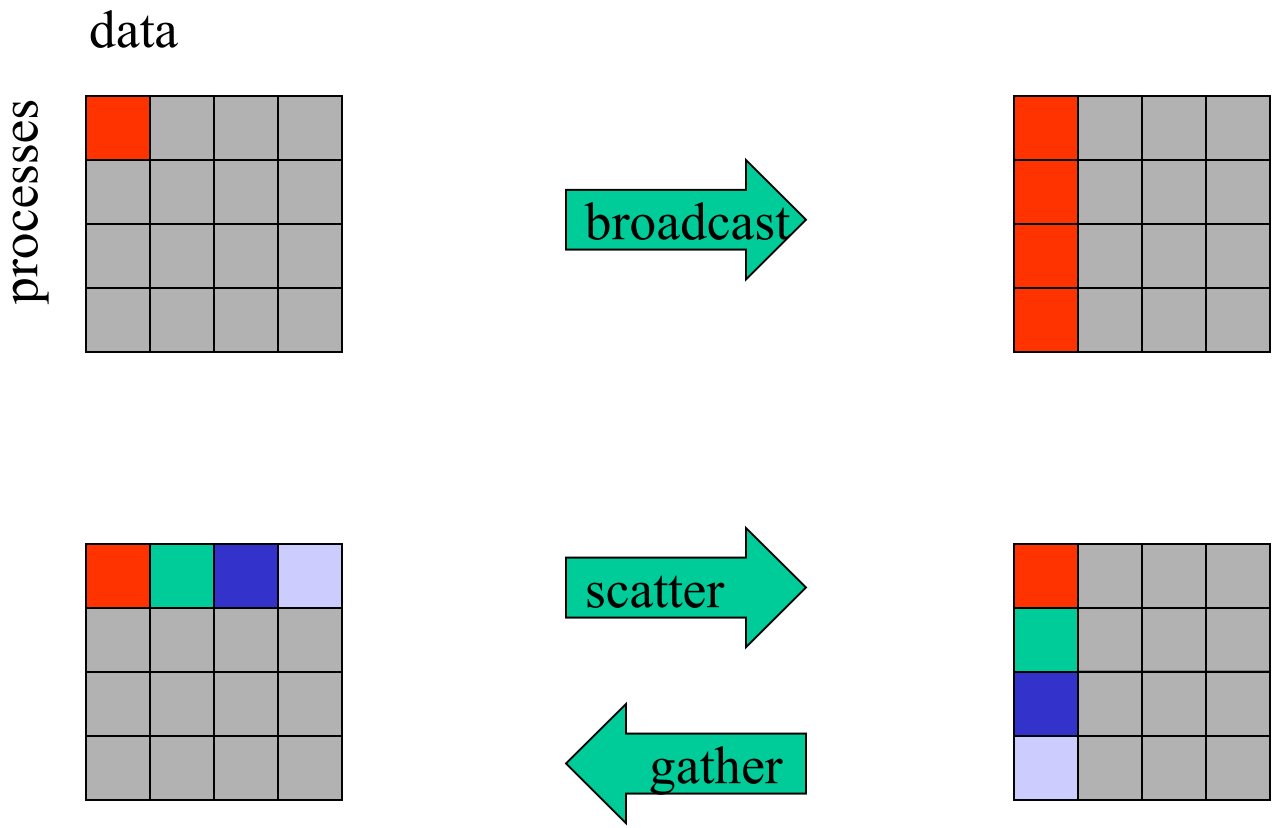
Collective Communication

- Optimized algorithms, scaling as log(n)
- Differences from point-to-point
  - Amount of data sent must match amount of data specified by receivers
  - No tags
  - Blocking only
- `MPI_barrier(comm)`
  - All processes in the communicator are synchronized. The only collective call where synchronization is guaranteed.

Collective Move Functions

- `MPI_Bcast(data, count, type, src, comm)`
  - Broadcast data from `src` to all processes in the communicator.
- `MPI_Gather(in, count, type, out, count, type, dest, comm)`
  - Gathers data from all nodes to dest node
- `MPI_Scatter(in, count, type, out, count, type, src, comm)`
  - Scatters data from src node to all nodes

Collective Move Functions



Collective Move Functions

- Additional functions
  - `MPI_Allgather, MPI_Gatherv, MPI_Scatterv, MPI_Allgatherv, MPI_Alltoall`

Collective Reduce Functions

- `MPI_Reduce(send, recv, count, type, op, root, comm)`
  - Global reduction operation, op, on send buffer. Result is at process root in recv buffer. op may be user defined, MPI predefined operation.
- `MPI_Allreduce(send, recv, count, type, op, comm)`
  - As above, except result broadcast to all processes.

Collective Reduce Functions

data

processes

A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

reduce

A0+A2+ A3+A4	B0+B1+ B2+B3	C0+C1+ C2+C3	D0+D1+ D2+D3

A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

allreduce

A0+A2+ A3+A4	B0+B1+ B2+B3	C0+C1+ C2+C3	D0+D1+ D2+D3
A0+A2+ A3+A4	B0+B1+ B2+B3	C0+C1+ C2+C3	D0+D1+ D2+D3
A0+A2+ A3+A4	B0+B1+ B2+B3	C0+C1+ C2+C3	D0+D1+ D2+D3
A0+A2+ A3+A4	B0+B1+ B2+B3	C0+C1+ C2+C3	D0+D1+ D2+D3

Collective Reduce Functions

- Additional functions
  - `MPI_Reduce_scatter, MPI_Scan`
- Predefined operations
  - Sum, product, min, max, …
- User-defined operations
  - `MPI_Op_create`

MPI Built-in Collective Computation Operations

`MPI_Max`	Maximum
`MPI_Min`	Minimum
`MPI_Prod`	Product
`MPI_Sum`	Sum
`MPI_Land`	Logical and
`MPI_Lor`	Logical or
`MPI_Lxor`	Logical exclusive or
`MPI_Band`	Binary and
`MPI_Bor`	Binary or
`MPI_Bxor`	Binary exclusive or
`MPI_Maxloc`	Maximum and location
`MPI_Minloc`	Minimum and location