Message Passing and MPI

CS 406/531 Week 10

slides are adapted from John Mellor-Crummey's slides from Rice University

Message Passing Overview

- The logical view of a message-passing platform
  - —*p* processes
  - —each with its own exclusive address space
- All data must be explicitly partitioned and placed
- All interactions (read-only or read/write) are two-sided
  - —process that has the data
  - —process that wants the data
- Typically use single program multiple data (SPMD) model
- The bottom line …
  - —strengths
    - – simple performance model: underlying costs are explicit
    - – portable high performance
  - —weakness: two-sided model can be awkward to program

Send and Receive

- Prototype operations

```
send(void *sendbuf, int nelems, int dest_rank)

receive(void *recvbuf, int nelems, int source_rank)
```

- Consider the following code fragments:

```
Processor 0

a = 100;

send(&a, 1, 1);

a = 0;
```

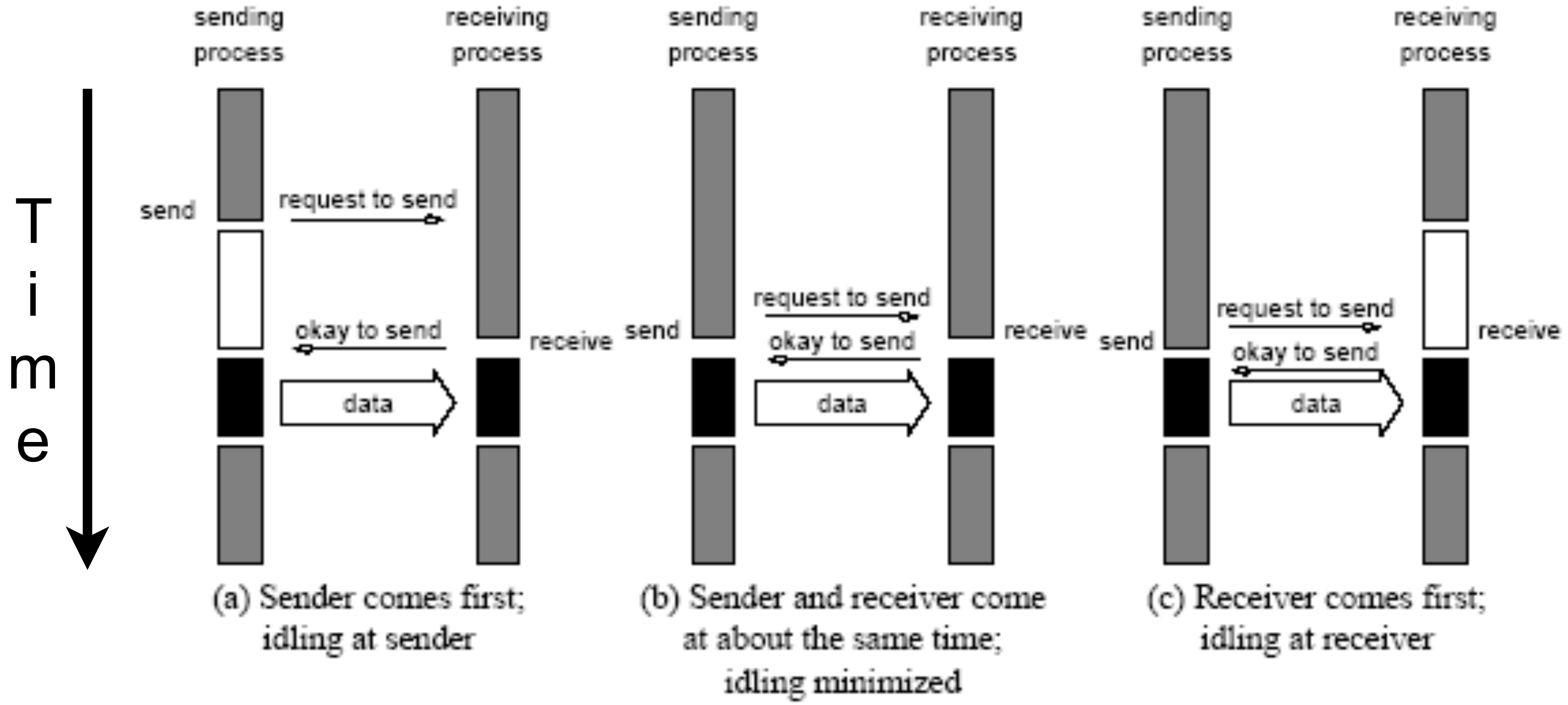
```
Processor 1

receive(&a, 1, 0)

printf("%d\n", a);
```

- The semantics of send
  - —value received by process P1 must be 100, not 0
  - —motivates the design of send and receive protocols

Blocking Message Passing

- Non-buffered, blocking sends
  - —send does not return until the matching receive executes
- Concerns
  - —idling
  - —deadlock

Non-Buffered, Blocking Message Passing



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Handshaking for blocking non-buffered send/receive

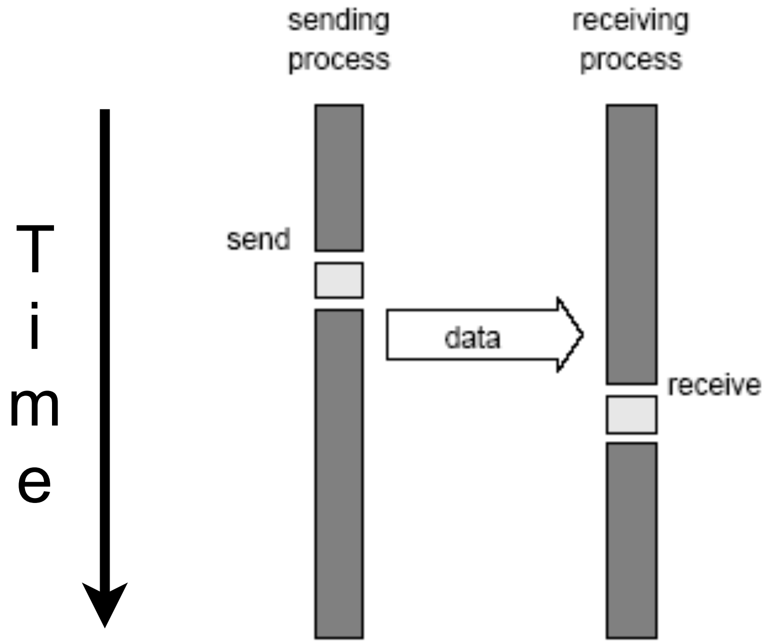
Idling occurs when operations are not simultaneous

(Case shown: no NIC support for communication)

Buffered, Blocking Message Passing

- **Buffered, blocking sends**
  - —sender copies the data into a buffer
  - —send returns after the copy completes
  - —data may be delivered into a buffer at the receiver as well
- **Tradeoff**
  - —buffering trades idling overhead for data copying overhead

Buffered, Blocking Message Passing



NIC moves the data behind the scenes

(illustrations show case when sender comes first)

Buffered Blocking Message Passing

Bounded buffer sizes can have significant impact on performance

```
Processor 0
for (i = 0; i < 1000; i++){
    produce_data(&a);
    send(&a, 1, 1);
  }
```

```
Processor 1
for (i = 0; i < 1000; i++){
   receive(&a, 1, 0);
   consume_data(&a);
}
```

Larger buffers enable the computation to tolerate asynchrony better

Buffered, Blocking Message Passing

Deadlocks are possible with buffering

since receive operations block

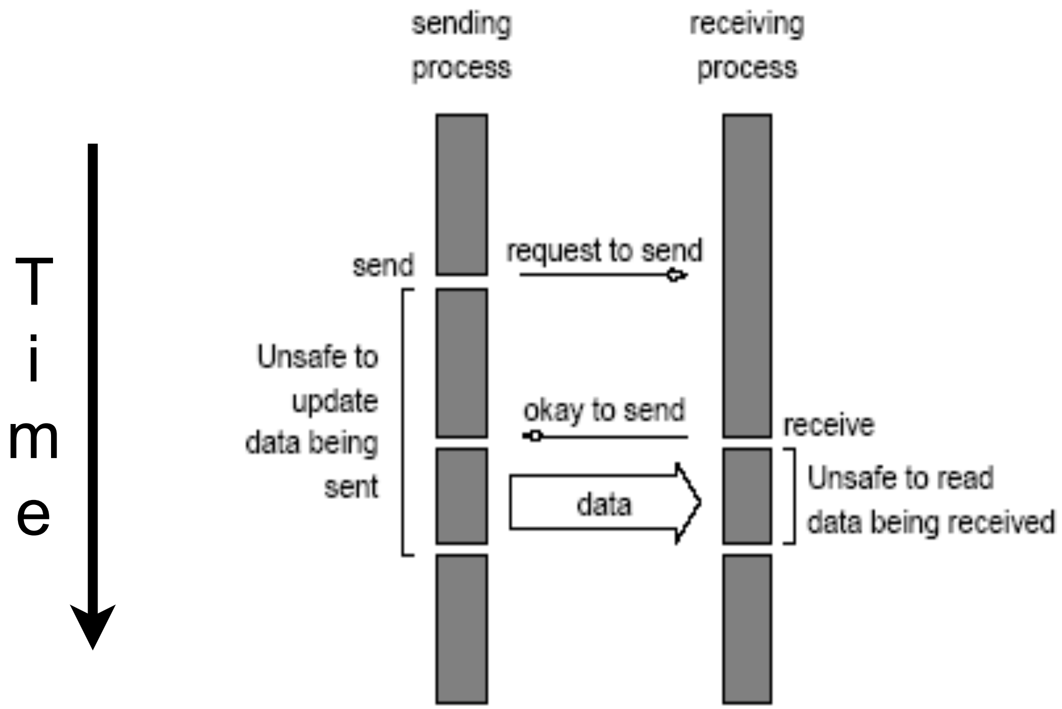
```
Processor 0

receive(&a, 1, 1);

send(&b, 1, 1);
```

```
Processor 1

receive(&a, 1, 0);

send(&b, 1, 0);
```

Non-Blocking Message Passing

- **Non-blocking protocols**
  - send and receive return before it is safe
    - sender: data can be overwritten before it is sent
    - receiver: can read data out of buffer before it is received
  - ensuring proper usage is the programmer’s responsibility
  - status check operation to ascertain completion
- **Benefit**
  - capable of overlapping communication with useful computation

Non-Blocking Message Passing

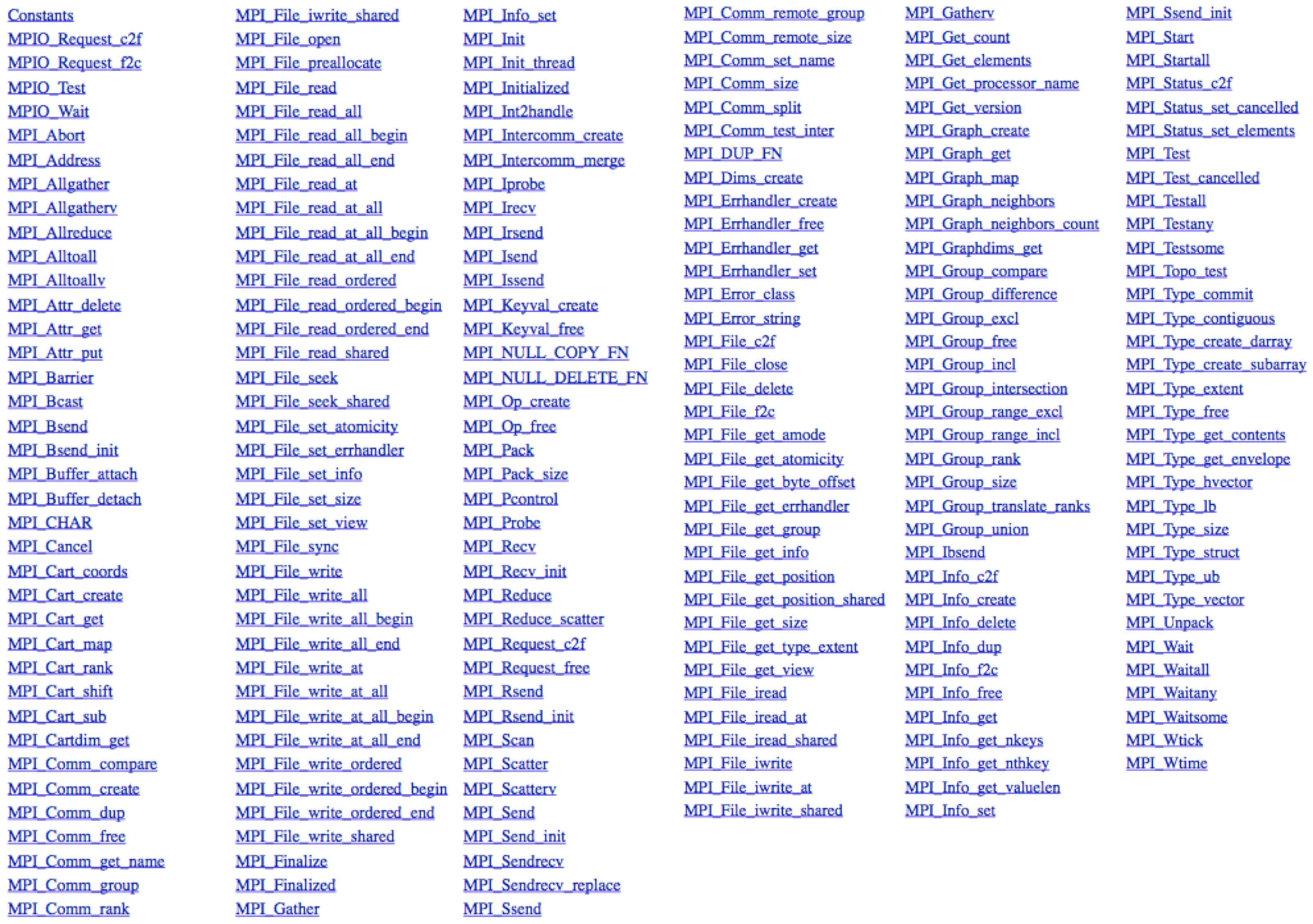


NIC moves the data behind the scenes

MPI: the Message Passing Interface

- Standard library for message-passing
  - —portable
  - —almost ubiquitously available
  - —high performance
  - —C and Fortran APIs
- MPI standard defines
  - —syntax of library routines
  - —semantics of library routines
- Details
  - —MPI routines, data-types, and constants are prefixed by “MPI_”
- Simple to get started
  - —fully-functional programs using only six library routines

MPI Primitives at a Glance



Constants
MPIO_Request_c2f
MPIO_Request_f2c
MPIO_Test
MPIO_Wait
MPI_Abort
MPI_Address
MPI_Allgather
MPI_Allgatherv
MPI_Allreduce
MPI_Alltoall
MPI_Alltoallv
MPI_Attr_delete
MPI_Attr_get
MPI_Attr_put
MPI_Barrier
MPI_Bcast
MPI_Bsend
MPI_Bsend_init
MPI_Buffer_attach
MPI_Buffer_detach
MPI_CHAR
MPI_Cancel
MPI_Cart_coords
MPI_Cart_create
MPI_Cart_get
MPI_Cart_map
MPI_Cart_rank
MPI_Cart_shift
MPI_Cart_sub
MPI_Cartdim_get
MPI_Comm_compare
MPI_Comm_create
MPI_Comm_dup
MPI_Comm_free
MPI_Comm_get_name
MPI_Comm_group
MPI_Comm_rank

MPI_File_iwrite_shared
MPI_File_open
MPI_File_preallocate
MPI_File_read
MPI_File_read_all
MPI_File_read_all_begin
MPI_File_read_all_end
MPI_File_read_at
MPI_File_read_at_all
MPI_File_read_at_all_begin
MPI_File_read_at_all_end
MPI_File_read_ordered
MPI_File_read_ordered_begin
MPI_File_read_ordered_end
MPI_File_read_shared
MPI_File_seek
MPI_File_seek_shared
MPI_File_set_atomicity
MPI_File_set_errhandler
MPI_File_set_info
MPI_File_set_size
MPI_File_set_view
MPI_File_sync
MPI_File_write
MPI_File_write_all
MPI_File_write_all_begin
MPI_File_write_all_end
MPI_File_write_at
MPI_File_write_at_all
MPI_File_write_at_all_begin
MPI_File_write_at_all_end
MPI_File_write_ordered
MPI_File_write_ordered_begin
MPI_File_write_ordered_end
MPI_File_write_shared
MPI_Finalize
MPI_Finalized
MPI_Gather

MPI_Info_set
MPI_Init
MPI_Init_thread
MPI_Initialized
MPI_Int2handle
MPI_Intercomm_create
MPI_Intercomm_merge
MPI_Iprobe
MPI_Irecv
MPI_Irsend
MPI_Isend
MPI_Issend
MPI_Keyval_create
MPI_Keyval_free
MPI_NULL_COPY_FN
MPI_NULL_DELETE_FN
MPI_Op_create
MPI_Op_free
MPI_Pack
MPI_Pack_size
MPI_Pcontrol
MPI_Probe
MPI_Recv
MPI_Recv_init
MPI_Reduce
MPI_Reduce_scatter
MPI_Request_c2f
MPI_Request_free
MPI_Rsend
MPI_Rsend_init
MPI_Scan
MPI_Scatter
MPI_Scatterv
MPI_Send
MPI_Send_init
MPI_Sendrecv
MPI_Sendrecv_replace
MPI_Ssend

MPI_Comm_remote_group
MPI_Comm_remote_size
MPI_Comm_set_name
MPI_Comm_size
MPI_Comm_split
MPI_Comm_test_inter
MPI_DUP_FN
MPI_Dims_create
MPI_Errhandler_create
MPI_Errhandler_free
MPI_Errhandler_get
MPI_Errhandler_set
MPI_Error_class
MPI_Error_string
MPI_File_c2f
MPI_File_close
MPI_File_delete
MPI_File_f2c
MPI_File_get_amode
MPI_File_get_atomicity
MPI_File_get_byte_offset
MPI_File_get_errhandler
MPI_File_get_group
MPI_File_get_info
MPI_File_get_position
MPI_File_get_position_shared
MPI_File_get_size
MPI_File_get_type_extent
MPI_File_get_view
MPI_File_iread
MPI_File_iread_at
MPI_File_iread_shared
MPI_File_iwrite
MPI_File_iwrite_at
MPI_File_iwrite_shared

MPI_Gatherv
MPI_Get_count
MPI_Get_elements
MPI_Get_processor_name
MPI_Get_version
MPI_Graph_create
MPI_Graph_get
MPI_Graph_map
MPI_Graph_neighbors
MPI_Graph_neighbors_count
MPI_Graphdims_get
MPI_Group_compare
MPI_Group_difference
MPI_Group_excl
MPI_Group_free
MPI_Group_incl
MPI_Group_intersection
MPI_Group_range_excl
MPI_Group_range_incl
MPI_Group_rank
MPI_Group_size
MPI_Group_translate_ranks
MPI_Group_union
MPI_Ibsend
MPI_Info_c2f
MPI_Info_create
MPI_Info_delete
MPI_Info_dup
MPI_Info_f2c
MPI_Info_free
MPI_Info_get
MPI_Info_get_nkeys
MPI_Info_get_nthkey
MPI_Info_get_valuelen
MPI_Info_set

MPI_Ssend_init
MPI_Start
MPI_Startall
MPI_Status_c2f
MPI_Status_set_cancelled
MPI_Status_set_elements
MPI_Test
MPI_Test_cancelled
MPI_Testall
MPI_Testany
MPI_Testsome
MPI_Topo_test
MPI_Type_commit
MPI_Type_contiguous
MPI_Type_create_darray
MPI_Type_create_subarray
MPI_Type_extent
MPI_Type_free
MPI_Type_get_contents
MPI_Type_get_envelope
MPI_Type_hvector
MPI_Type_lb
MPI_Type_size
MPI_Type_struct
MPI_Type_ub
MPI_Type_vector
MPI_Unpack
MPI_Wait
MPI_Waitall
MPI_Waitany
MPI_Waitsome
MPI_Wtick
MPI_Wtime

http://www.mcs.anl.gov/research/projects/mpi/www/www3

MPI: the Message Passing Interface

Minimal set of MPI routines

`MPI_Init`	initialize MPI
`MPI_Finalize`	terminate MPI
`MPI_Comm_size`	determine number of processes in group
`MPI_Comm_rank`	determine id of calling process in group
`MPI_Send`	send message
`MPI_Recv`	receive message

Starting and Terminating the MPI Programs

- `int MPI_Init(int *argc, char ***argv)`
  - —initialization: must call this prior to other MPI routines
  - —effects
    - strips off and processes any MPI command-line arguments
    - initializes MPI environment
- `int MPI_Finalize()`
  - —must call at the end of the computation
  - —effect
    - performs various clean-up tasks to terminate MPI environment
- **Return codes**
  - —MPI_SUCCESS
  - —MPI_ERROR

Communicators

- `MPI_Comm`: communicator = communication domain
  - —group of processes that can communicate with one another
- Supplied as an argument to all MPI message transfer routines
- Process can belong to multiple communication domains
  - —domains may overlap
- `MPI_COMM_WORLD`: root communicator
  - — includes all the processes

Communicator Inquiry Functions

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - —determine the number of processes
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - —index of the calling process
  - —$0 \leq$ rank < communicator size

“Hello World” Using MPI

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

int main(int argc, char *argv[])
{
      int npes, myrank;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &npes);
      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
      printf("From process %d out of %d, Hello World!\n",
             myrank, npes);
      MPI_Finalize();
      return 0;
}
```

Sending and Receiving Messages

-
  ```
  int MPI_Send(void *buf, int count, MPI_Datatype datatype,
               int dest_pe, int tag, MPI_Comm comm)
  ```

-
  ```
  int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
              int source_pe, int tag, MPI_Comm comm,
              MPI_Status *status)
  ```

- **Message source or destination PE**
  - **—index of process in the communicator `comm`**
  - **—receiver wildcard: `MPI_ANY_SOURCE`**
    - **any process in the communicator can be source**
- **Message-tag: integer values, 0 ≤ tag < MPI_TAG_UB**
  - **—receiver tag wildcard: `MPI_ANY_TAG`**
    - **messages with any tag are accepted**
- **Receiver constraint**
  - **— message size ≤ buffer length specified**

MPI Primitive Data Types

MPI data type	C data type
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	8 bits
`MPI_PACKED`	packed sequence of bytes

Receiver Status Inquiry

- `Mpi_Status`
  - —stores information about an `MPI_Recv` operation
  - —data structure

```
typedef struct MPI_Status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR; };
```

- `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`
  - —returns the count of data items received
    - – not directly accessible from status variable

Deadlock with MPI_Send/Recv?

destination

tag

```
int a[10], b[10], myrank;
MPI_Status s1, s2;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, &s1);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &s2);
}
...
```

Definition of MPI_Send says: “This routine may block until the message is received by the destination process”

Deadlock if MPI_Send is blocking

Another Deadlock Pitfall?

Send data to neighbor to your right on a ring ...

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
             MPI_COMM_WORLD);

MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
              MPI_COMM_WORLD, &status);
...
```

Deadlock if MPI_Send is blocking

Avoiding Deadlock with Blocking Sends

Send data to neighbor to your right on a ring ...

Break the circular wait

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank%2 == 1) {    // odd processes send first, receive second
       MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
              MPI_COMM_WORLD);
       MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
              MPI_COMM_WORLD, &status);
}
else {                  // even processes receive first, send second
       MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
              MPI_COMM_WORLD, &status);
       MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
              MPI_COMM_WORLD);
}
...
```

Primitives for Non-blocking Communication

- Non-blocking send and receive return before they complete

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

- `MPI_Test:` has a particular non-blocking request finished?

```
int MPI_Test(MPI_Request *request, int *flag,
        MPI_Status *status)
```

- `MPI_Waitany:` block until some request in a set completes

```
int MPI_Wait_any(int req_cnt, MPI_Request *req_array,
                 int *req_index, MPI_Status *status)
```

- `MPI_Wait:` block until a particular request completes

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Avoiding Deadlocks with NB Primitives

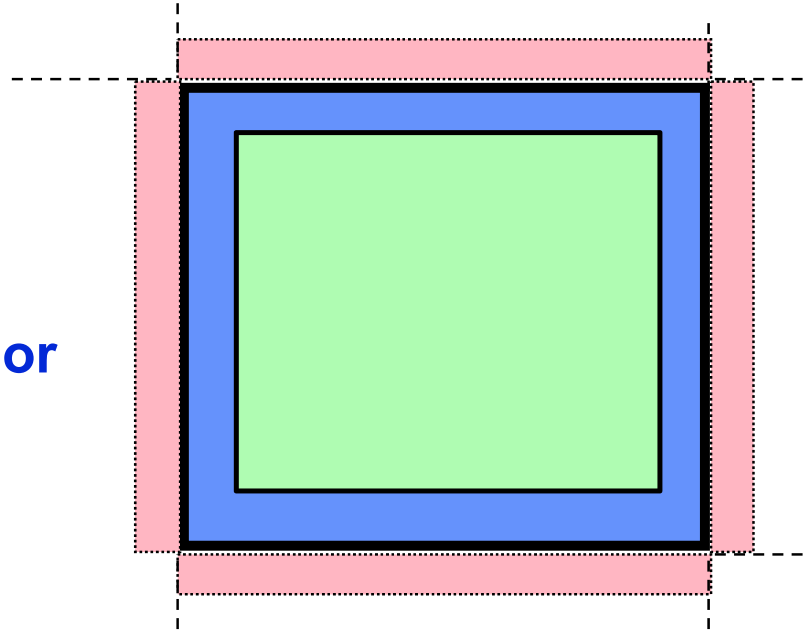
Using non-blocking operations avoids most deadlocks

tag

```
int a[10], b[10], myrank;
MPI_Request r1, r2;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_ISend(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD, &r1);
   MPI_ISend(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD, &r2);
}
else if (myrank == 1) {
   MPI_IRecv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD, &r1);
   MPI_IRecv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD, &r2);
}
...
```

Overlapping Communication Example

- **Original**
  - —send boundary layer (blue) to neighbors with blocking send
  - —receive boundary layer (pink) from neighbors
  - —compute data volume (green + blue)
- **Overlapped**
  - —send boundary layer (blue) to neighbor with non-blocking send
  - —compute interior region (green) from
  - —receive boundary layer (pink)
  - —wait for non-blocking sends to complete (blue)
  - —compute boundary layer (blue)



Message Exchange

To exchange messages in a single call (both send and receive)

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
  MPI_Datatype senddatatype, int dest, int sendtag,
  void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
  int source, int recvtag, MPI_Comm comm,
  MPI_Status *status)
```

Requires both send and receive arguments

Why Sendrecv?

Sendrecv is useful for executing a shift operation along a chain of processes. If blocking send and recv are used for such a shift, then one needs to avoid deadlock with an odd/even scheme. When Sendrecv is used, MPI handles these issues.

To use same buffer for both send and receive

```
int MPI_Sendrecv_replace(void *buf, int count,
  MPI_Datatype datatype, int dest, int sendtag,
  int source, int recvtag, MPI_Comm comm,
  MPI_Status *status)
```

Collective Communication in MPI

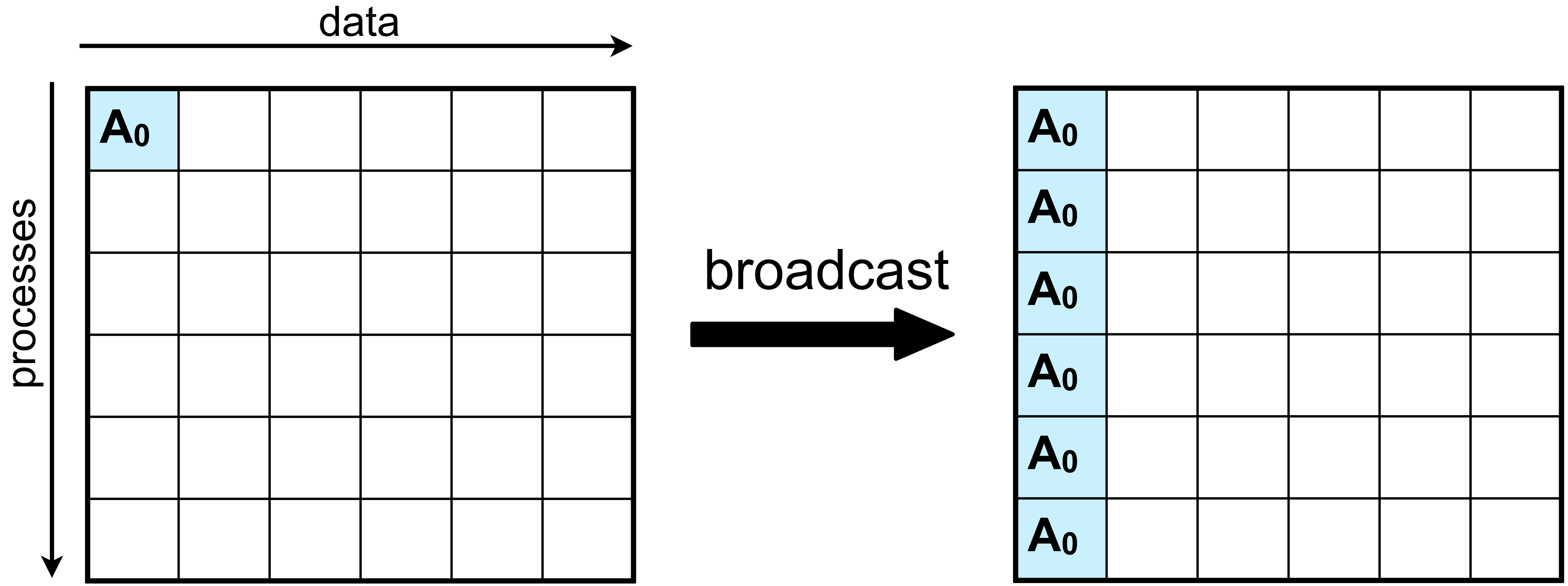
- MPI provides an extensive set of collective operations
- Operations defined over a communicator's processes
- All processes in a communicator must call the same collective operation
  - —e.g. all participants in a one-to-all broadcast call the broadcast primitive, even though all but the root are conceptually just "receivers"
- Simplest collective: barrier synchronization

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

  - wait until all processes arrive

One-to-all Broadcast

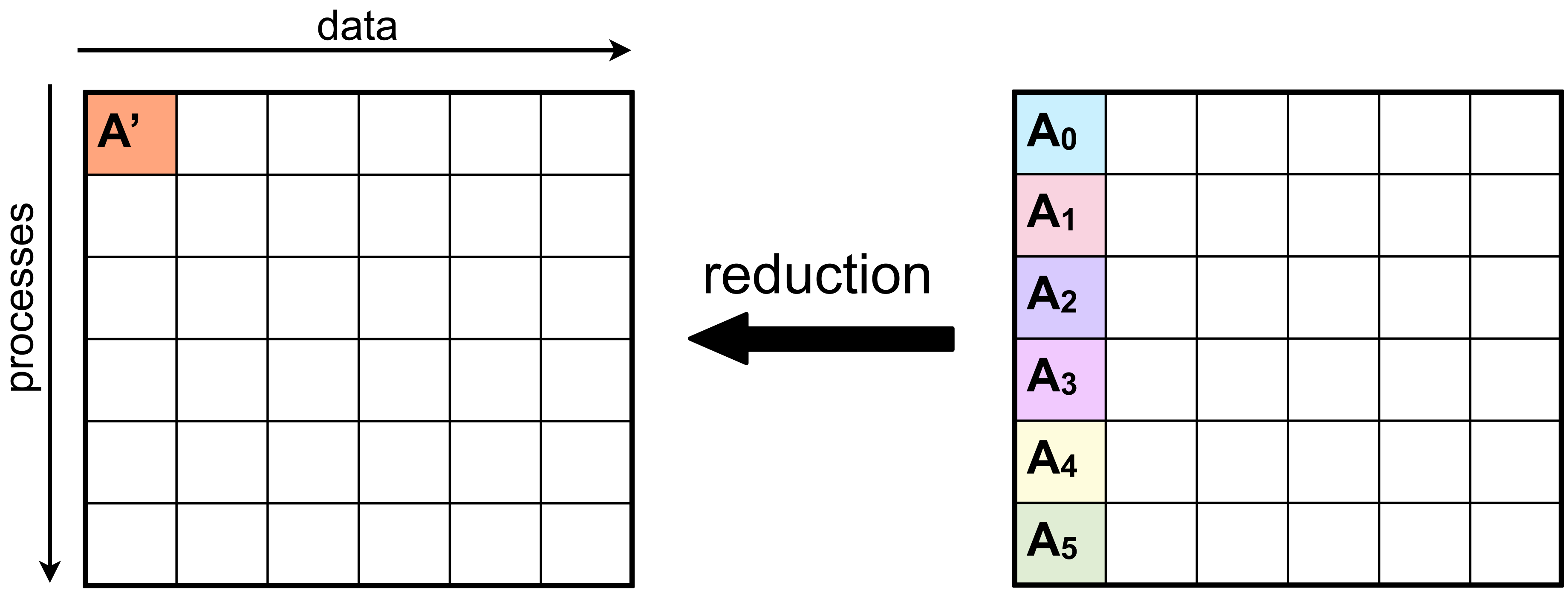
```
int MPI_Bcast(void *buf, int count,
              MPI_Datatype datatype, int source,
              MPI_Comm comm)
```



All-to-one Reduction

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

MPI_Op examples: sum, product, min, max, ... (see next page)



$A' = op(A_0, \ldots A_{p-1})$

MPI_Op Predefined Reduction Operations

Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	integers and floating point
`MPI_MIN`	Minimum	integers and floating point
`MPI_SUM`	Sum	integers and floating point
`MPI_PROD`	Product	integers and floating point
`MPI_LAND`	Logical AND	integers
`MPI_BAND`	Bit-wise AND	integers and byte
`MPI_LOR`	Logical OR	integers
`MPI_BOR`	Bit-wise OR	integers and byte
`MPI_LXOR`	Logical XOR	integers
`MPI_BXOR`	Bit-wise XOR	integers and byte
`MPI_MAXLOC`	Max value-location	Data-pairs
`MPI_MINLOC`	Min value-location	Data-pairs

MPI_MAXLOC and MPI_MINLOC

- `MPI_MAXLOC`
  - combines pairs of values (v_i, l_i)
  - returns the pair (v, l) such that
    - v is the maximum among all v_i 's
    - l is the corresponding l_i

      if non-unique, it is the smallest among l_i 's
- `MPI_MINLOC` analogous

Value	15	17	11	12	17	11
Process	0	1	2	3	4	5

```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

Data Types for MINLOC and MAXLOC Reductions

`MPI_MAXLOC` and `MPI_MINLOC` reductions operate on data pairs

MPI Datatype	C Datatype
`MPI_2INT`	**pair of** ints
`MPI_SHORT_INT`	short **and** int
`MPI_LONG_INT`	long **and** int
`MPI_LONG_DOUBLE_INT`	long double **and** int
`MPI_FLOAT_INT`	float **and** int
`MPI_DOUBLE_INT`	double **and** int

All-to-All Reduction and Prefix Sum

- **All-to-all reduction - every process gets a copy of the result**

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

  —semantically equivalent to MPI_Reduce + MPI_Bcast

- **Parallel prefix operations**

  —inclusive scan: processor i result = op(v_0, ... v_i)

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

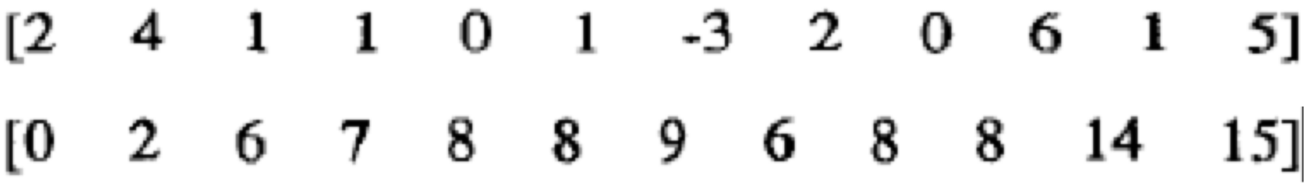
  —exclusive scan: processor i result = op(v_0, ... v_{i-1})

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

Exscan
example
MPI_SUM

input

output



```
[2  4  1  1  0  1  -3  2  0  6  1   5]
[0  2  6  7  8  8   9  6  8  8  14  15]
```

Scatter/Gather

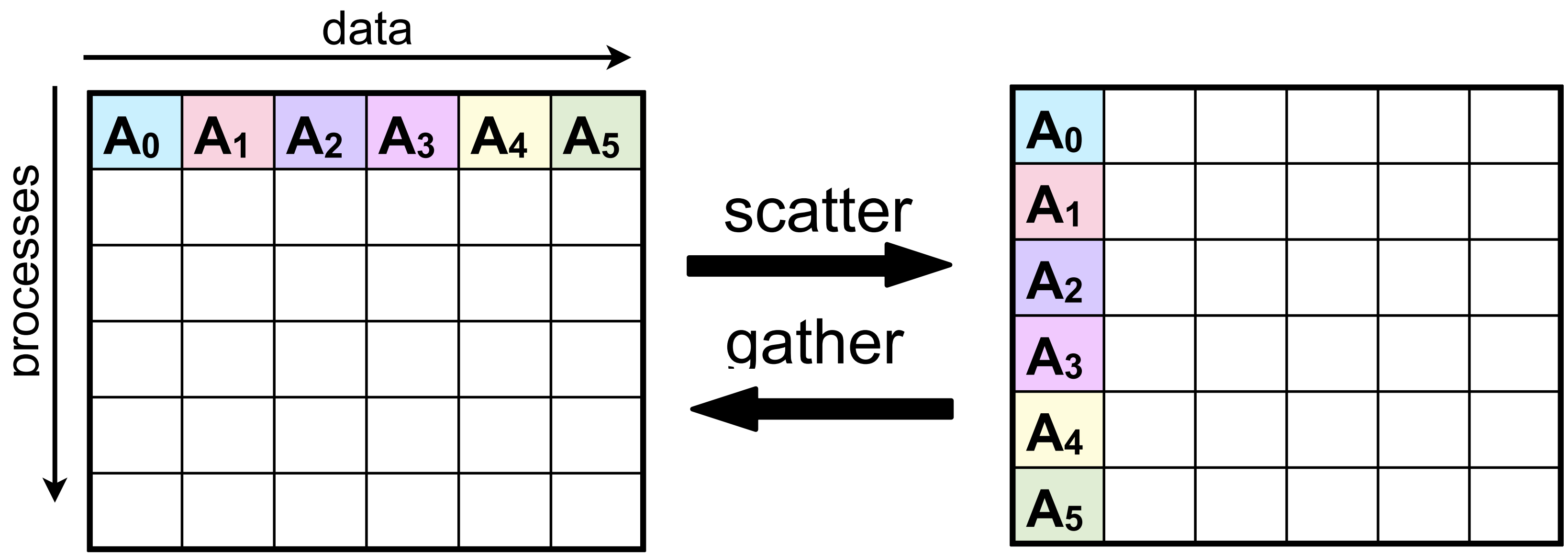
- **Scatter data p-1 blocks from root process delivering one to each other**

```
int MPI_Scatter(void *sendbuf, int sendcount,
         MPI_Datatype senddatatype, void *recvbuf,
         int recvcount, MPI_Datatype recvdatatype,
         int source, MPI_Comm comm)
```

sendcount = number sent to each

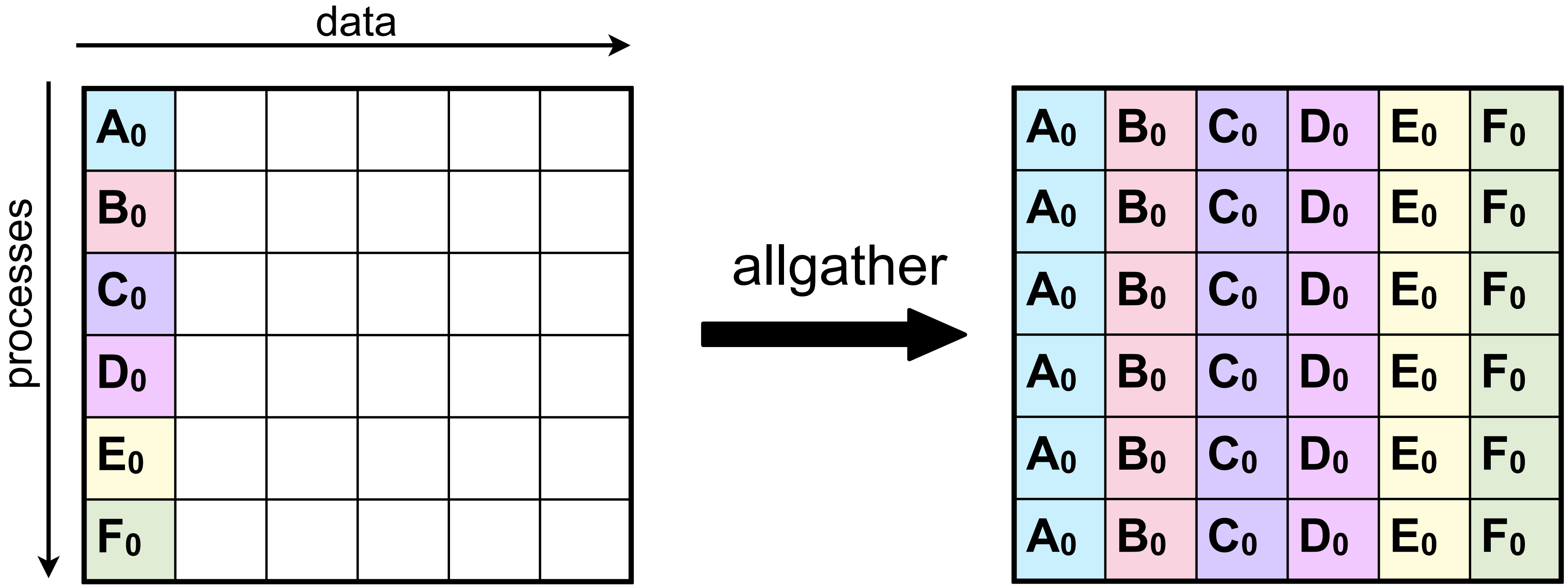
- **Gather data at one process**

```
int MPI_Gather(void *sendbuf, int sendcount,
         MPI_Datatype senddatatype, void *recvbuf,
         int recvcount, MPI_Datatype recvdatatype,
         int target, MPI_Comm comm)
```



Allgather

```
int MPI_AllGather(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype, void *recvbuf,
                  int recvcount, MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```



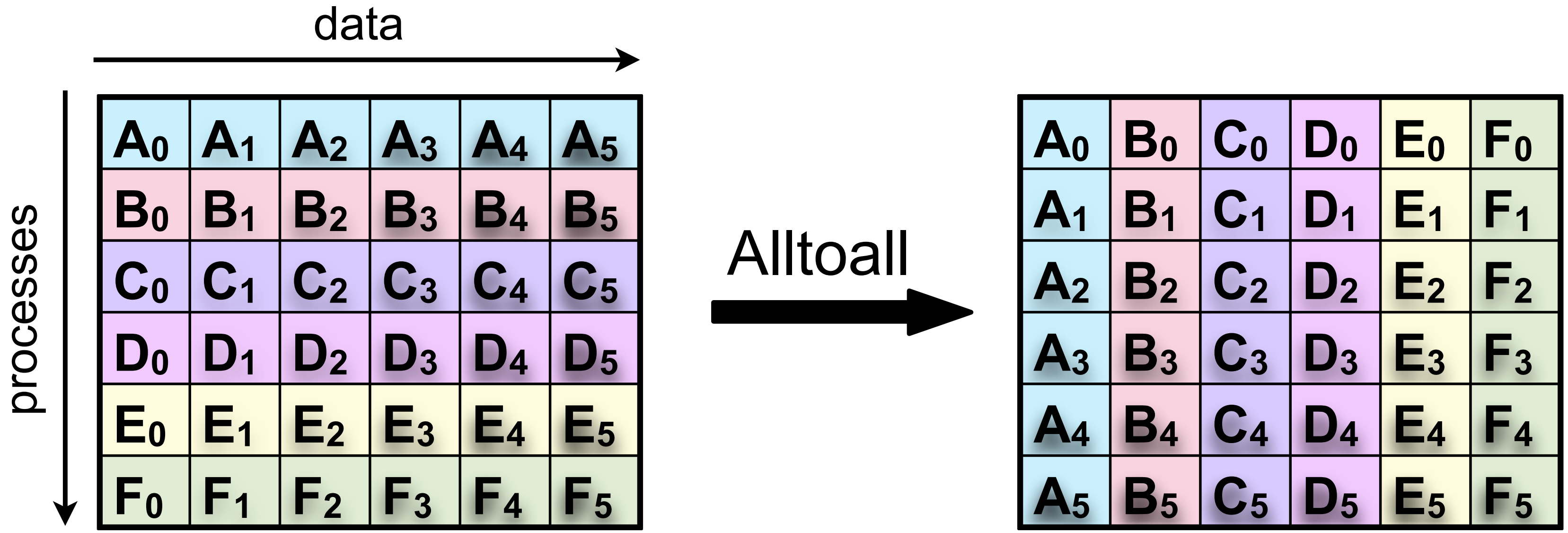
A_0					
B_0					
C_0					
D_0					
E_0					
F_0					

A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0

All-to-All Personalized Communication

- Each process starts with its own set of blocks, one destined for each process
- Each process finishes with all blocks destined for itself
- Analogous to a matrix transpose

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```



A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

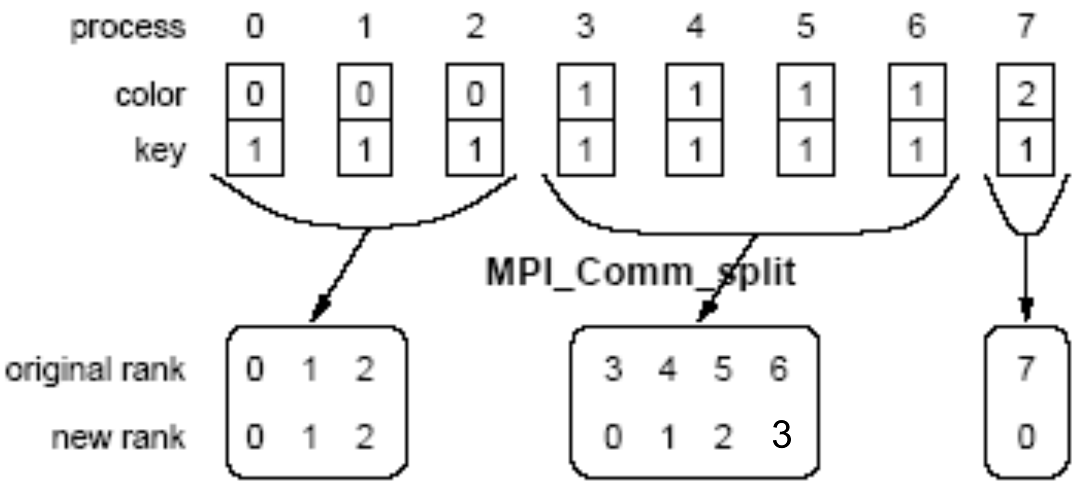
Splitting Communicators

- Useful to partition communication among process subsets
- MPI provides mechanism for partitioning a process group
  - —splitting communicators
- Simplest such mechanism

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

  - —effect
    - – group processes by color
    - – sort resulting groups by key

Splitting Communicators



Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups