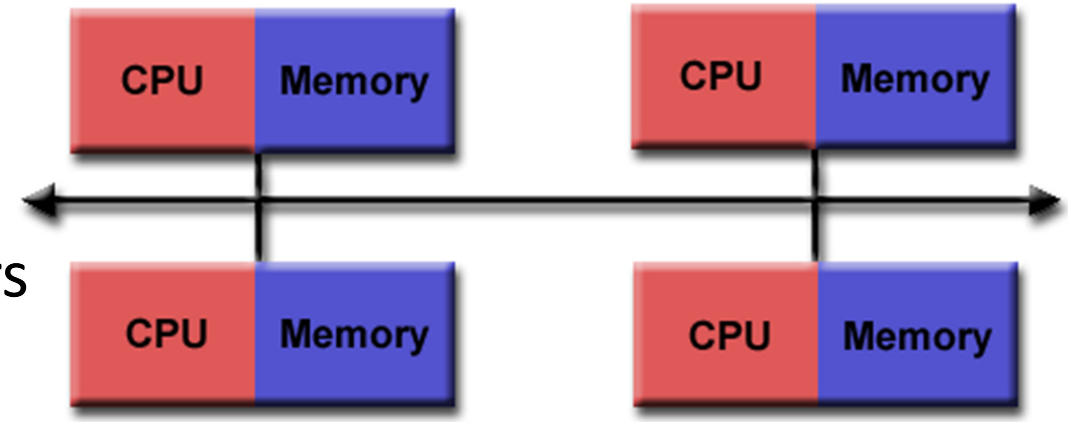
CSE 613: Parallel Programming

Lecture 27
(The Message Passing Interface)

Jesmin Jahan Tithi
(Slides: Rezaul A. Chowdhury)
Department of Computer Science
SUNY Stony Brook
Spring 2015

Principles of Message-Passing Programming



Source: Blaise Barney, LLNL

- One of the oldest and most widely used approaches for programming parallel computers
- Two key attributes
  - Assumes a partitioned address space
  - Supports only explicit parallelism
- Two immediate implications of partitioned address space
  - Data must be explicitly partitioned and placed to appropriate partitions
  - Each interaction (read-only and read/write) requires cooperation between two processes: process that has the data, and the one that wants to access the data

Structure of Message-Passing Programs

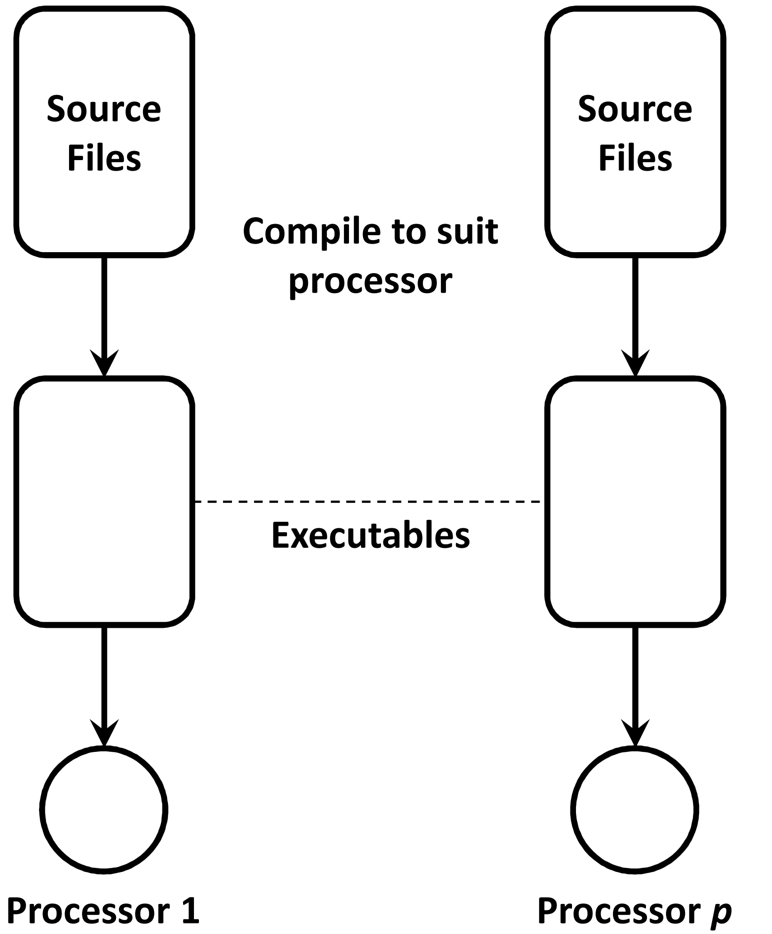
Asynchronous

- All concurrent tasks execute asynchronously
- Most general (can implement any parallel algorithm)
- Can be difficult to reason about
- Can have non-deterministic behavior due to races

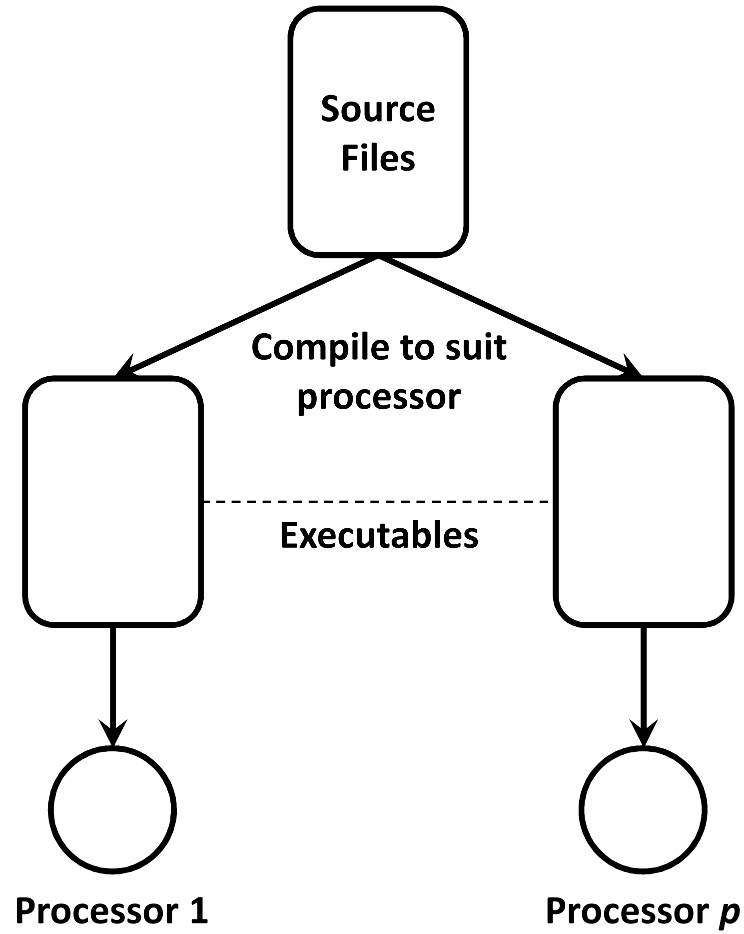
Loosely Synchronous

- A good compromise between synchronous and asynchronous
- Tasks or subset of tasks synchronize to interact
- Between the interactions tasks execute asynchronously
- Easy to reason about these programs

Structure of Message-Passing Programs



Multiple Program Multiple Data (MPMD)



Single Program Multiple Data (SPMD)

– Ultimate flexibility in parallel programming
– Unscalable

– Most message-passing programs
– Loosely synchronous or completely asynchronous

The Building Blocks: Send & Receive Operations

send(&*data*, *n*, *dest* **):**

Send *n* items pointed to by &*data* to a processor with id *dest*

receive(&*data*, *n*, *src* **):**

Receive *n* items from a processor with id *src* to location pointed to by &*data*

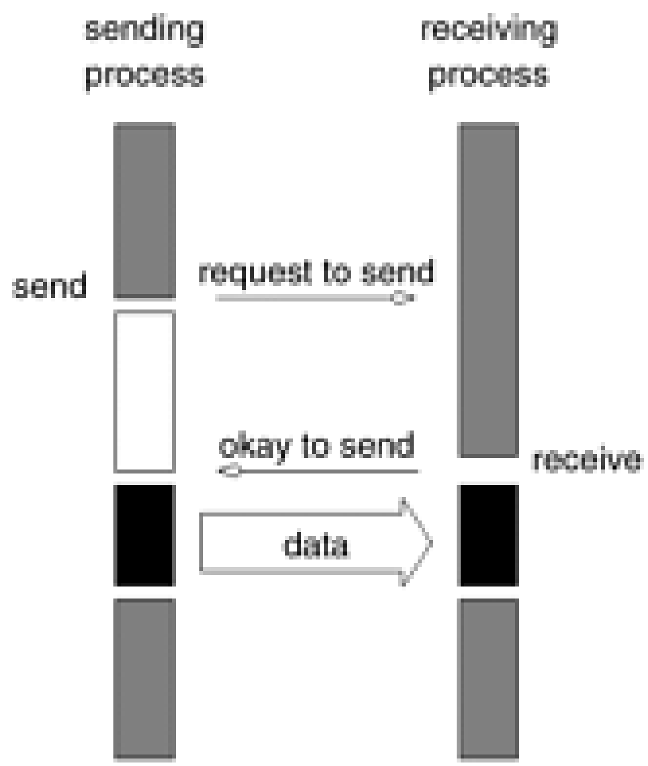
But wait! What P1 prints when P0 and P1 execute the following code?

```
P0                          P1

a = 100;                    receive(&a, 1, 0)
send(&a, 1, 1);             printf("%d\n", a);
a=0;
```

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

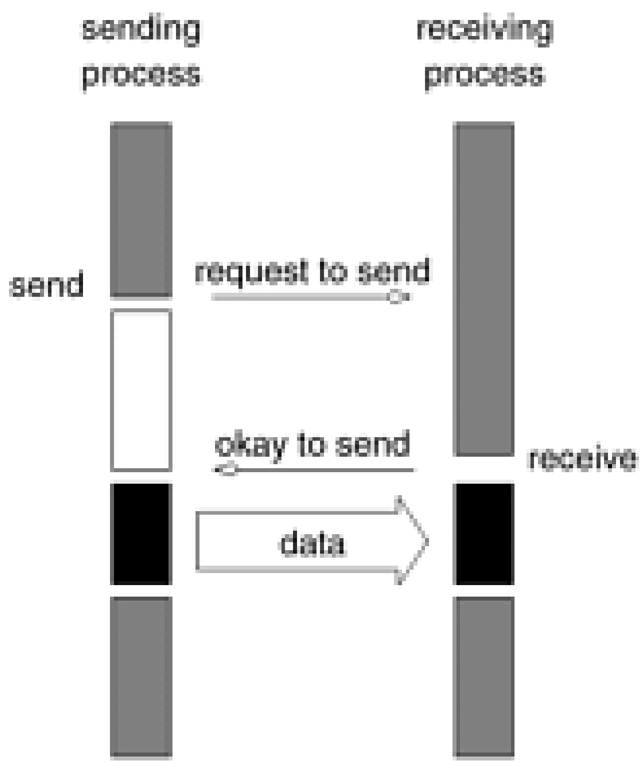
Blocking Non-Buffered Send / Receive

Sending operation waits until the matching receive operation is encountered at the receiving process, and data transfer is complete.



Blocking Non-Buffered Send / Receive

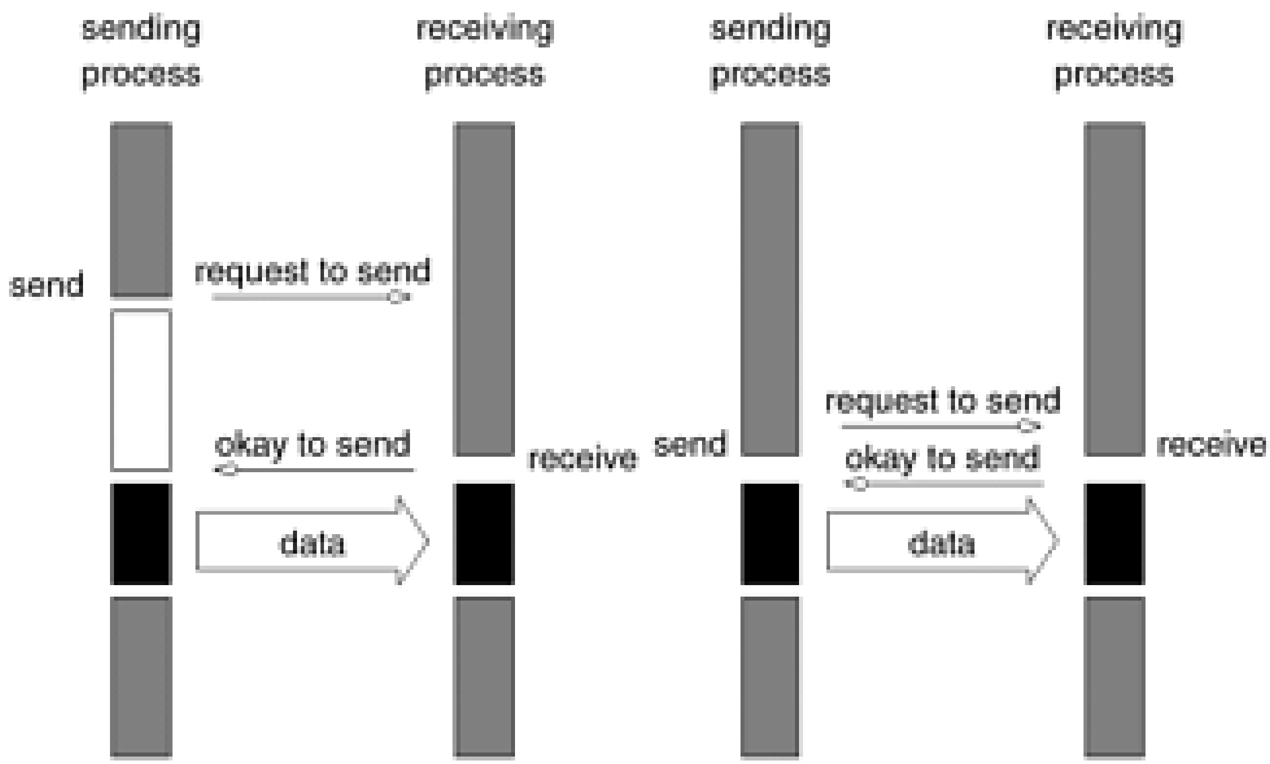
May lead to idling:



(a) Sender comes first;
idling at sender

Blocking Non-Buffered Send / Receive

May lead to idling:

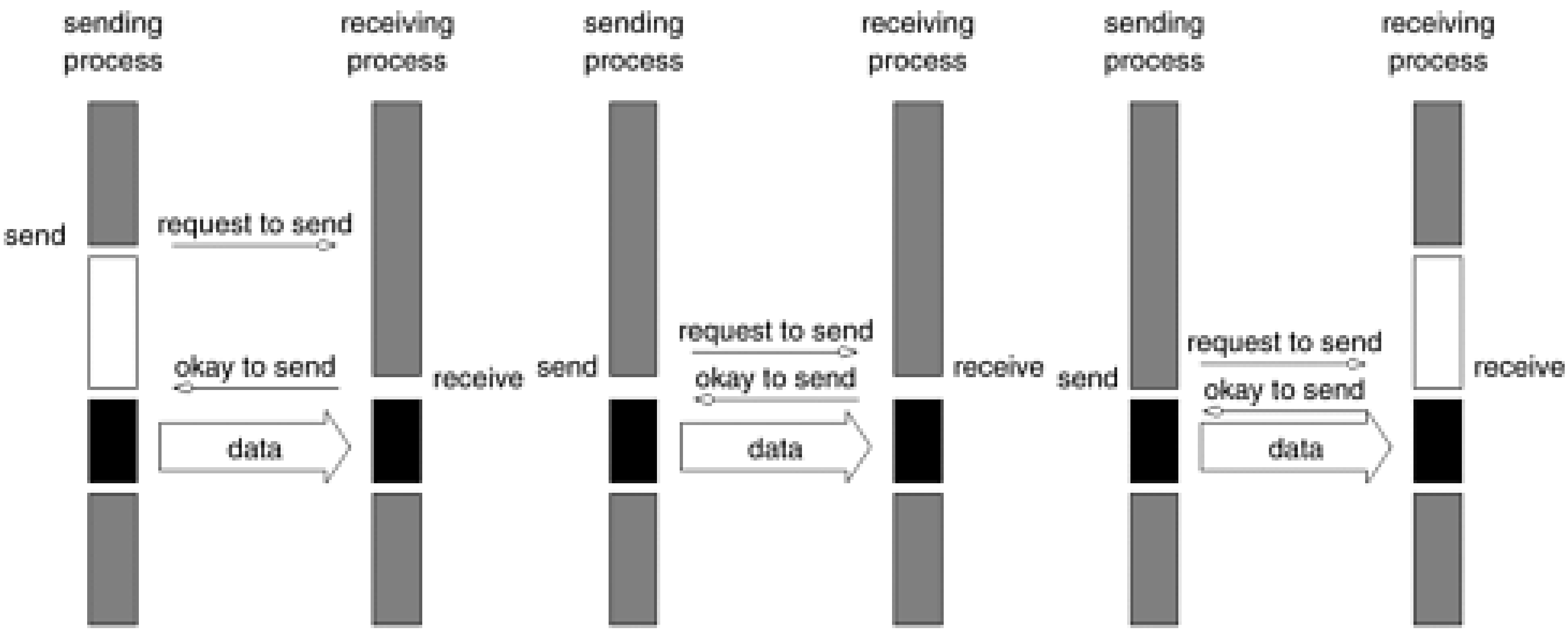


(a) Sender comes first; idling at sender

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

Blocking Non-Buffered Send / Receive

May lead to idling:



(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

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Blocking Non-Buffered Send / Receive

May lead to deadlocks:

```
        P0                                  P1

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

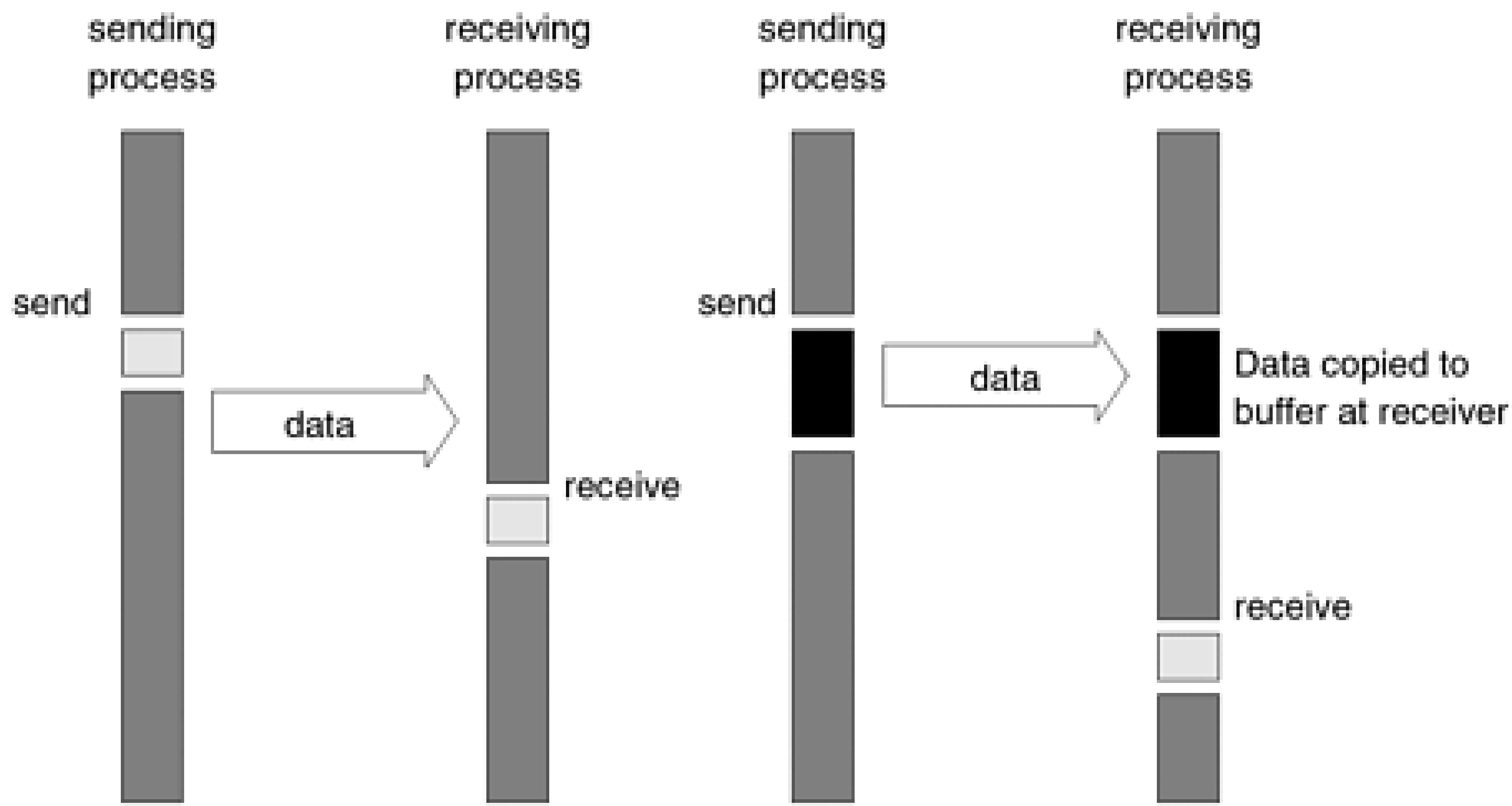
Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

- The send at P0 waits for the matching receive at P1
- The send at P1 waits for the matching receive at P0

Blocking Buffered Send / Receive

– Sending operation waits until data is copied into a pre-allocated communication buffer at the sending process

– Data is first copied into a buffer at the receiving process as well, from where data is copied to the target location by the receiver



(a) With hardware support (b) Without hardware support

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Blocking Buffered Send / Receive

Finite buffers lead to delays:

```
P0

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

```
P1

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

Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition

– What happens if the sender’s buffer can only hold 10 items?

Blocking Buffered Send / Receive

May still lead to deadlocks:

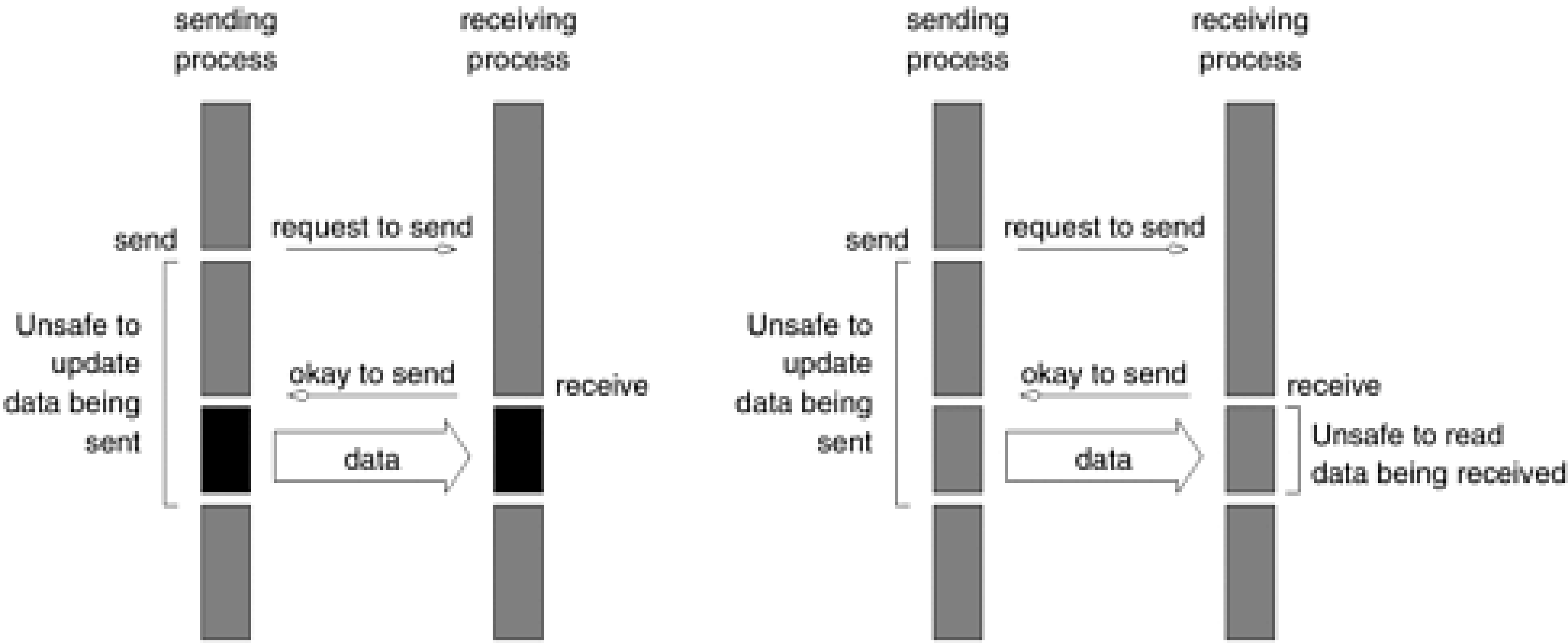
```
P0                                    P1

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

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

– Blocks because the receive calls are always blocking in order to ensure consistency

Non-Blocking Non-Buffered Send / Receive

- Sending operation posts a pending message and returns
- When the corresponding receive is posted data transfer starts
- When data transfer is complete the *check-status* operation indicates that it is safe to touch the data



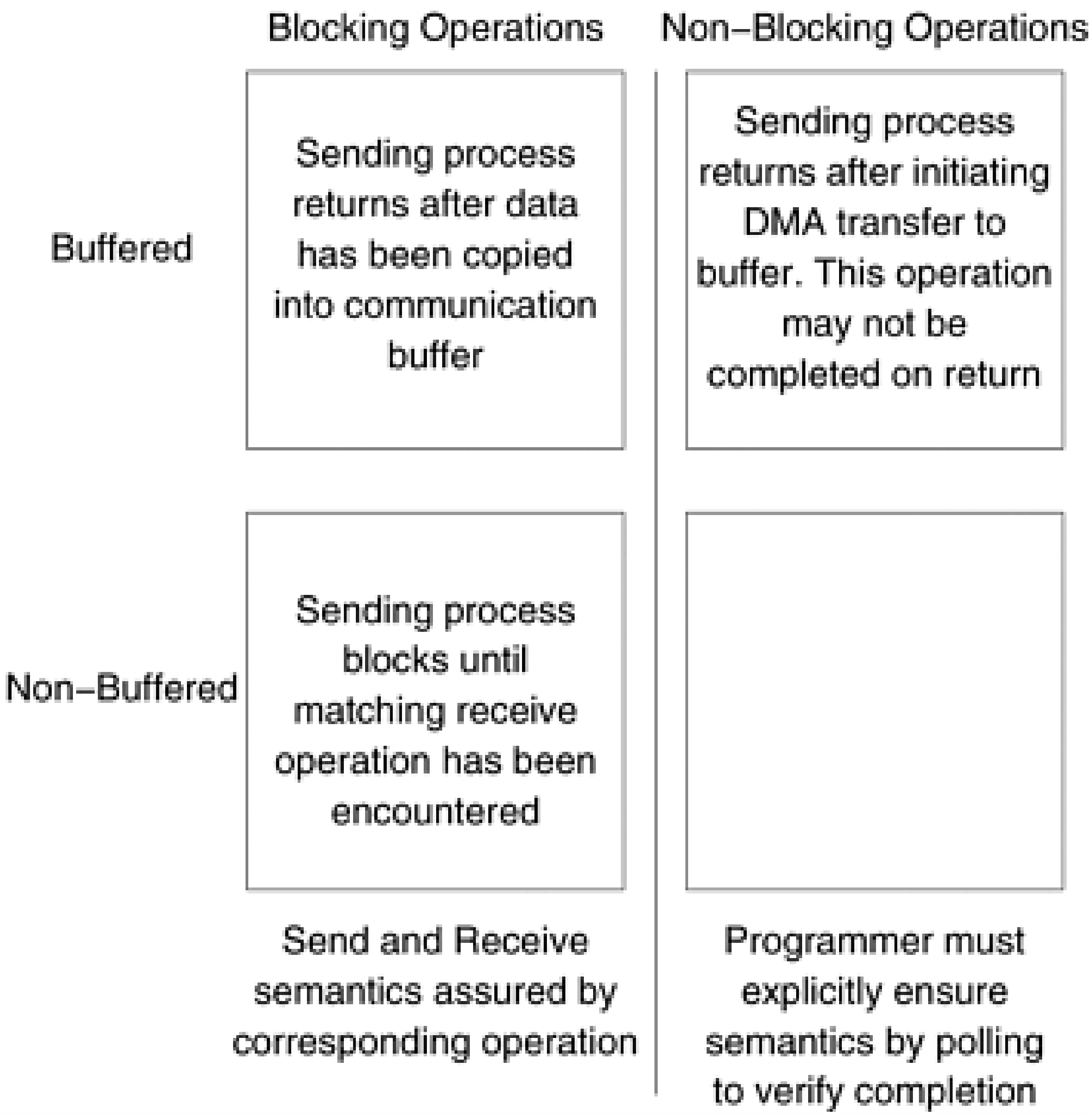
(a) Without hardware support (b) With hardware support

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Non-Blocking Buffered Send / Receive

- Sending operation initiates a DMA (Direct Memory Access) operation and returns immediately
- Data becomes safe as soon as the DMA operation completes
- The receiver initiates a transfer from sender's buffer to receiver's target location
- Reduces the time during which the data is unsafe to touch

Possible Protocols for Send & Receive Operations



Source: Grama et al.,
"Introduction to Parallel Computing",
2nd Edition

The Minimal Set of MPI Routines

- The MPI library contains over 125 routines
- But fully functional message-passing programs can be written using only the following 6 MPI routines

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of the calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

- All 6 functions return *MPI_SUCCESS* upon successful completion, otherwise return an implementation-defined error code
- All MPI routines, data-types and constants are prefixed by *MPI_*
- All of them are defined in *mpi.h* (for C/C++)

Starting and Terminating the MPI Library

```
#include < mpi.h >

main( int argc, char *argv[ ] )
{
    MPI_Init( &argc, &argv );
    … … …                          // do some work
    MPI_Finalize( );
}
```

- Both *MPI_Init* and *MPI_Finalize* must be called by all processes
- Command line should be processed only after *MPI_Init*
- No MPI function may be called after *MPI_Finalize*

Communicators

- A *communicator* defines the scope of a communication operation
- Each process included in the communicator has a rank associated with the communicator
- By default, all processes are included in a communicator called *MPI_COMM_WORLD*, and each process is given a unique rank between 0 and $p - 1$, where p is the number of processes
- Additional communicator can be created for groups of processes
- To get the size of a communicator:

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

- To get the rank of a process associated with a communicator:

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

Communicators

```
#include < mpi.h >

main( int argc, char *argv[ ] )
{
    int p, myrank;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &p );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    printf( “This is process %d out of %d!\n”, p, myrank );
    MPI_Finalize( );
}
```

MPI Standard Blocking Send Format

data parameters

address of send buffer

number of items to send

datatype of each item

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

rank of destination process

message tag

communicator

envelope parameters

MPI Standard Blocking Receive Format

data parameters

address of receive buffer

number of items to receive

datatype of each item

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

rank of source process

message tag

communicator

status after operation

envelope parameters

MPI Datatypes

MPI Datatype	C Datatype
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	
MPI_PACKED	

Blocking Send/Receive between Two Processes

```
#include < mpi.h >

main( int argc, char *argv[ ] )
{
    int myrank, v = 121;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if ( myrank == 0 ) {
        MPI_Send( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD );
        printf( “Process %d sent %d!\n”, p, myrank, v );
    }  else if ( myrank == 1 ) {
        MPI_Recv( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD , &status );
        printf( “Process %d received %d!\n”, p, myrank, v );
    }
    MPI_Finalize( );
}
```

Non-Blocking Send / Receive

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

int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,
               int src, int tag, MPI_Comm comm, MPI_Request *req )
```

The MPI_Request object is used as an argument to the following two functions to identify the operation whose status we want to query or to wait for its completion.

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

- Returns *flag = 1, if the operation associated with *req has completed, otherwise returns *flag = 0

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

- Waits until the operation associated with *req completes

Non-Blocking Send and Blocking Receive

```
#include < mpi.h >

main( int argc, char *argv[ ] )
{
    int myrank, v = 121;
    MPI_Status status;
    MPI_Request req;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if ( myrank == 0 ) {
        MPI_Isend( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
        compute( );            /* but do not modify v */
        MPI_Wait( &req, &status );
    }  else if ( myrank == 1 ) MPI_Recv( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD , &status );
    MPI_Finalize( );
}
```

Non-Blocking Send/Receive

```
#include < mpi.h >
main( int argc, char *argv[ ] )
{
    int myrank, v = 121;
    MPI_Status status;
    MPI_Request req;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if ( myrank == 0 ) {
        MPI_Isend( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
        compute( );            /* but do not modify v */
        MPI_Wait( &req, &status );
    } else if ( myrank == 1 ) {
        MPI_Irecv( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
        compute( );            /* but do not read or modify v */
        MPI_Wait( &req, &status );
    }
    MPI_Finalize( );
}
```

MPI Collective Communication & Computation Operations

Synchronization

- Barrier

Data Movement

- Broadcast
- Scatter
- Gather
- All-to-all

Global Computation

- Reduce
- Scan

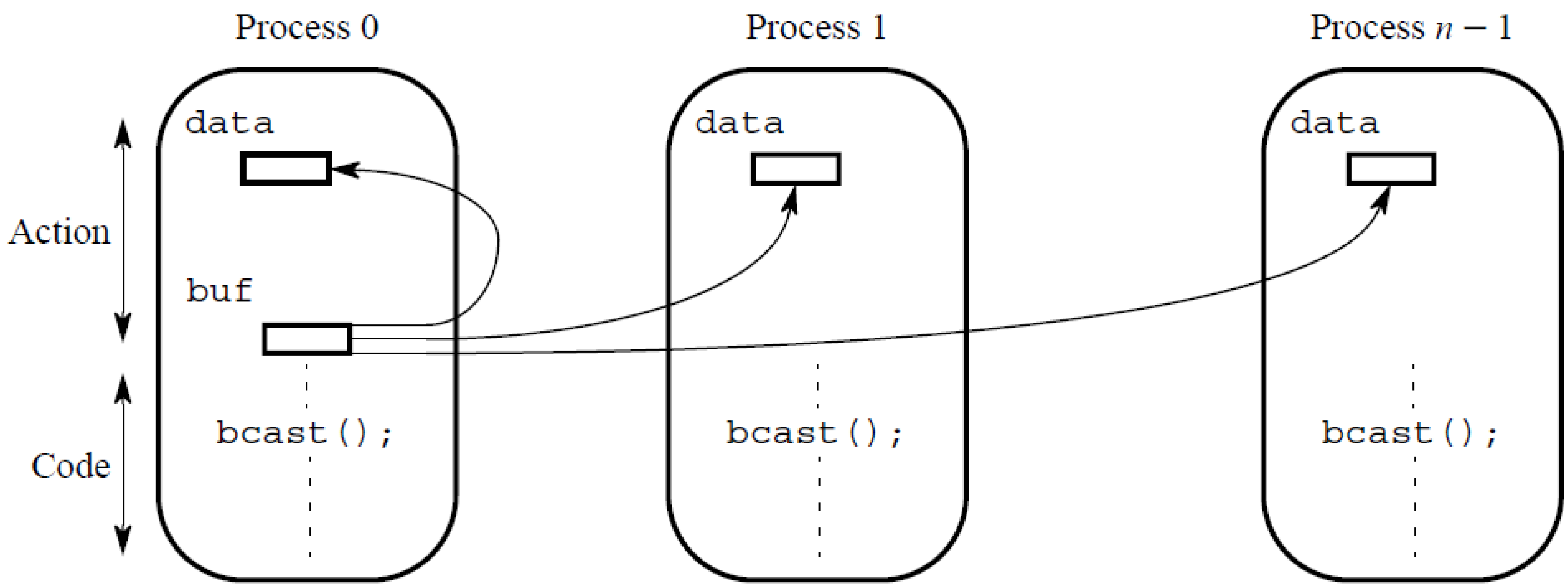
These routines must be called by all processes in the communication group

Barrier Synchronization

int MPI_Barrier(MPI_Comm comm)

Returns only after all processes in the communication group have called this function

Broadcast

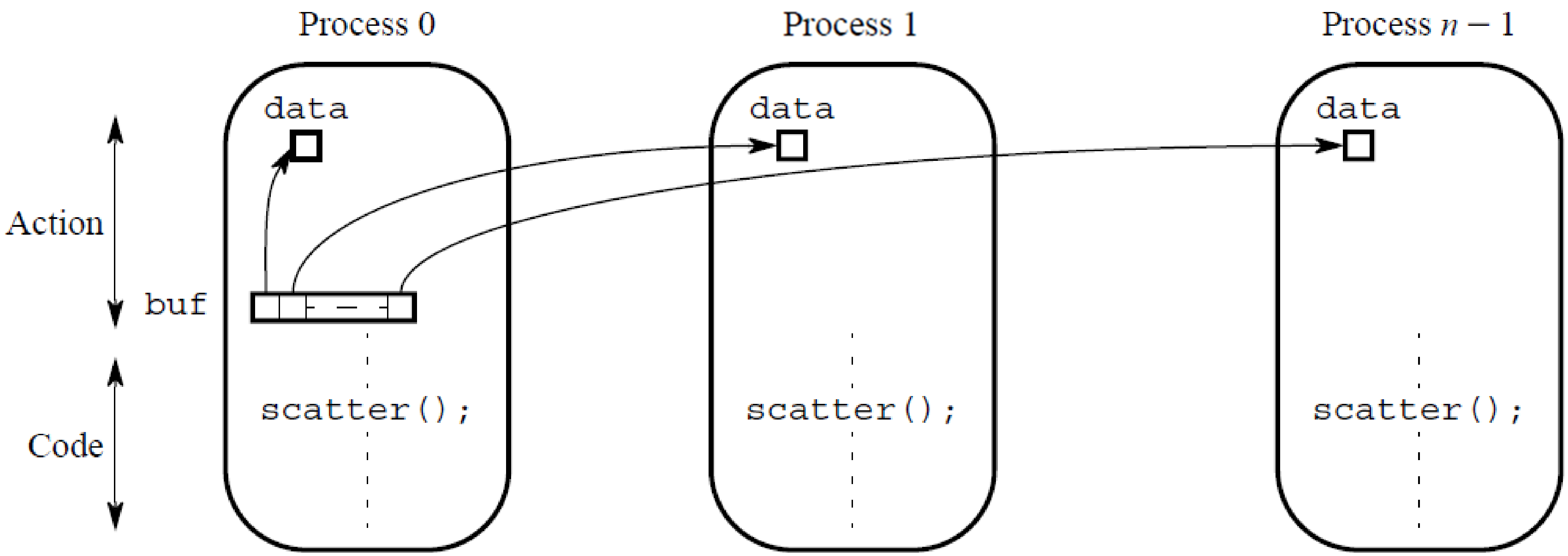


Source: Wilkinson & Allen., "Parallel Programming", 2nd Edition

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

Sends the data stored in the buffer *buf* of process *src* to all the other processes in the group

Scatter



Source: Wilkinson & Allen., "Parallel Programming", 2nd Edition

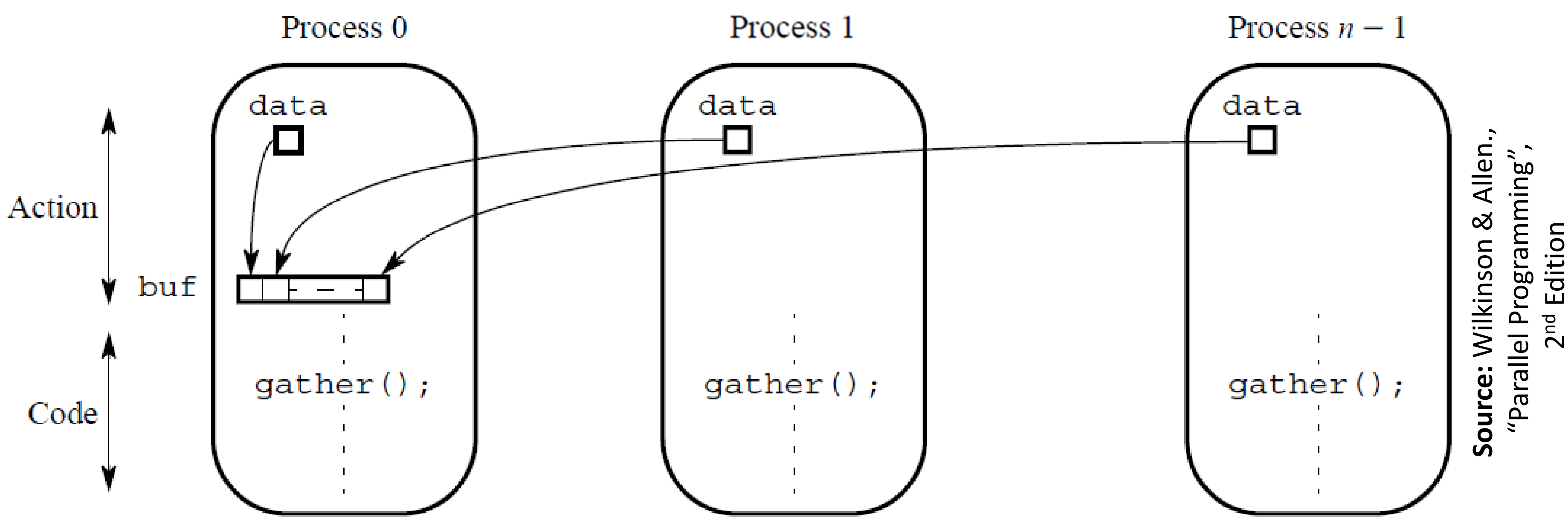
```
int MPI_Scatter( void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int src,
                 MPI_Comm comm )
```

The *src* process sends a different part of *sendbuf* to each process, including itself.

Process *i* receives *sendcount* contiguous elements starting from *i* × *sendcount.*

The received data are stored in *recvbuf*.

Gather



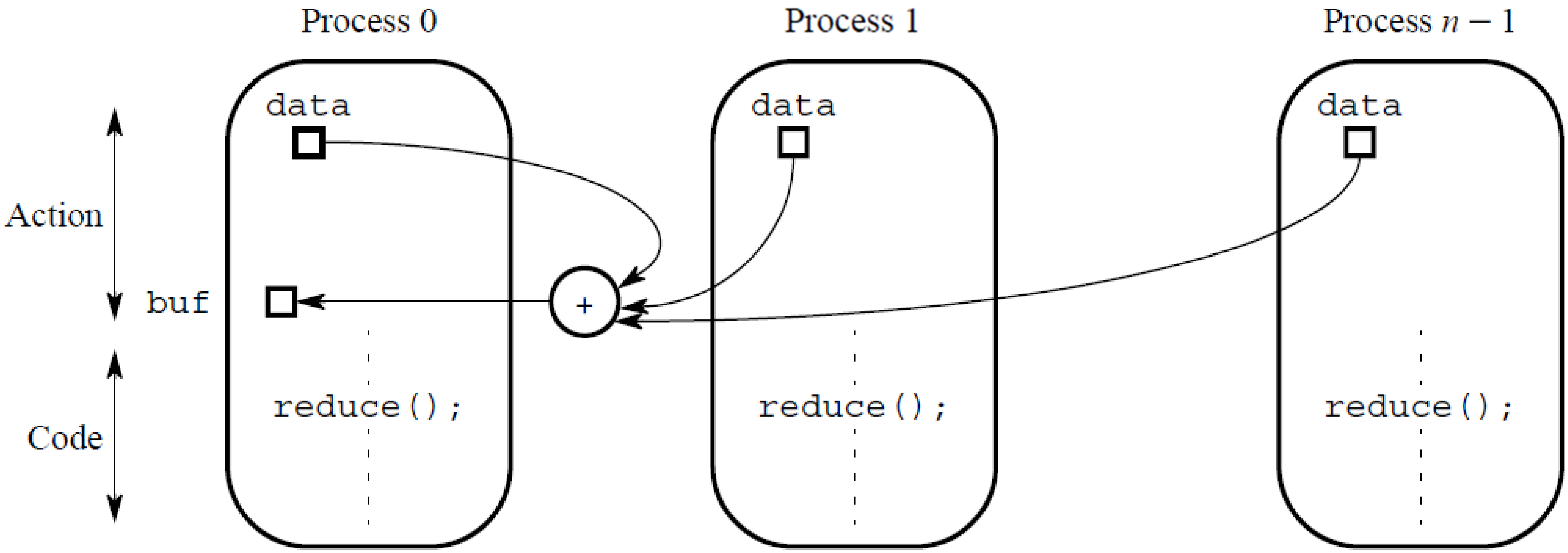

```
int MPI_Gather( void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int dest,
                MPI_Comm comm )
```

The opposite of scatter.

Every process, including *dest* sends data stored in *sendbuf* to *dest*.

Data from process *i* occupy *sendcount* contiguous locations of *recvbuf* starting from $i \times$ *sendcount.*

Reduce



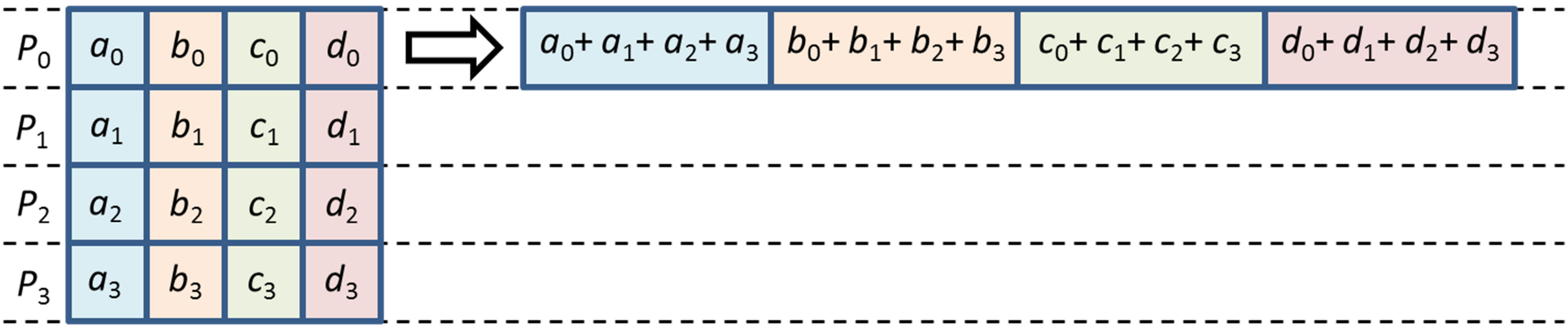
Source: Wilkinson & Allen., "Parallel Programming", 2nd Edition

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

Combines the elements stored in *sendbuf* of each process using the operation *op*, and stores the combined values in *recvbuf* of the process with rank *dest*.

Reduce

MPI_Reduce(vals, sums, 4, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)



Predefined Reduction Operations

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

Scan / Prefix

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

Performs a prefix reduction of the data stored in *sendbuf* at each process and returns the results in *recvbuf* of the process with rank *dest*.

P_0	a_0	b_0	c_0	d_0
P_1	a_1	b_1	c_1	d_1
P_2	a_2	b_2	c_2	d_2
P_3	a_3	b_3	c_3	d_3

⇒

a_0	b_0	c_0	d_0
$a_0 + a_1$	$b_0 + b_1$	$c_0 + c_1$	$d_0 + d_1$
$a_0 + a_1 + a_2$	$b_0 + b_1 + b_2$	$c_0 + c_1 + c_2$	$d_0 + d_1 + d_2$
$a_0 + a_1 + a_2 + a_3$	$b_0 + b_1 + b_2 + b_3$	$c_0 + c_1 + c_2 + c_3$	$d_0 + d_1 + d_2 + d_3$

```
MPI_Scan( vals, sums, 4, MPI_INT, MPI_SUM, MPI_COMM_WORLD )
```