Message Passing

Slide source: Ananth Grama, Purdue; Jim Demmel, Berkeley; Bill Gropp, ANL

Message Passing Paradigm

Oldest and most widely used →
Minimal requirements on underlying hardware

Principles of Message-Passing Programming

1. Logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
2. Has two constraints, while onerous, make underlying costs very explicit to the programmer.
   - Every data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
   - All interactions (Rd or Rd/Wr) require cooperation of two processes - the process that has the data and the process that wants to access the data.

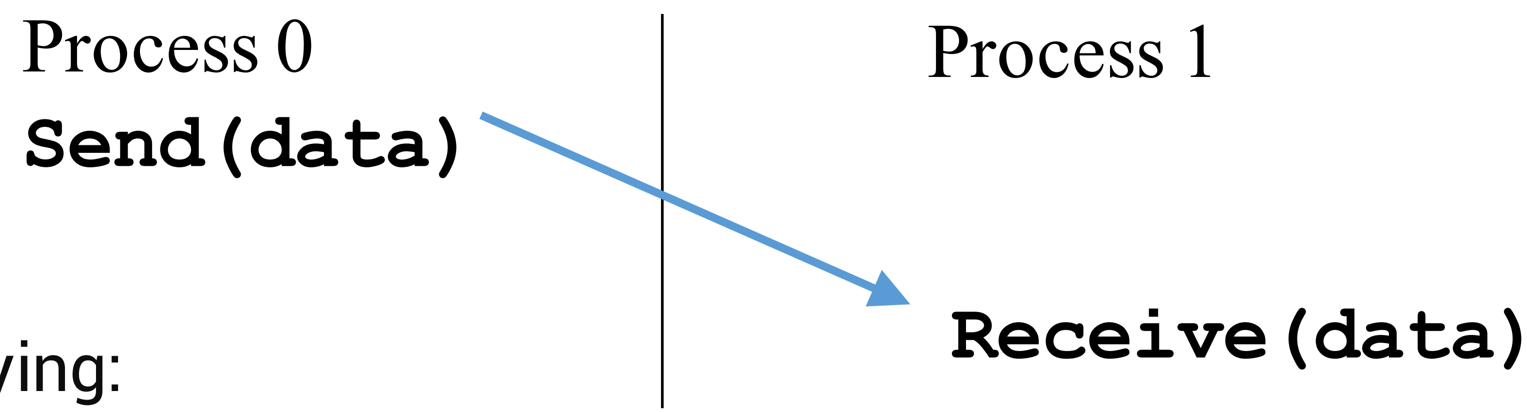
Principles of Message-Passing Programming

Asynchronous Or Loosely Synchronous?

1. ASYNC paradigm: All concurrent tasks execute asynchronously.
2. Vs. Loosely Sync.: Tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.
3. Most message-passing programs are written using the *single program multiple data* (SPMD) model.

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?

The Building Blocks: Send and Receive Operations

- The prototypes of these operations are as follows:

```
send    (void *sendbuf, int nelems, int dest)
receive (void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

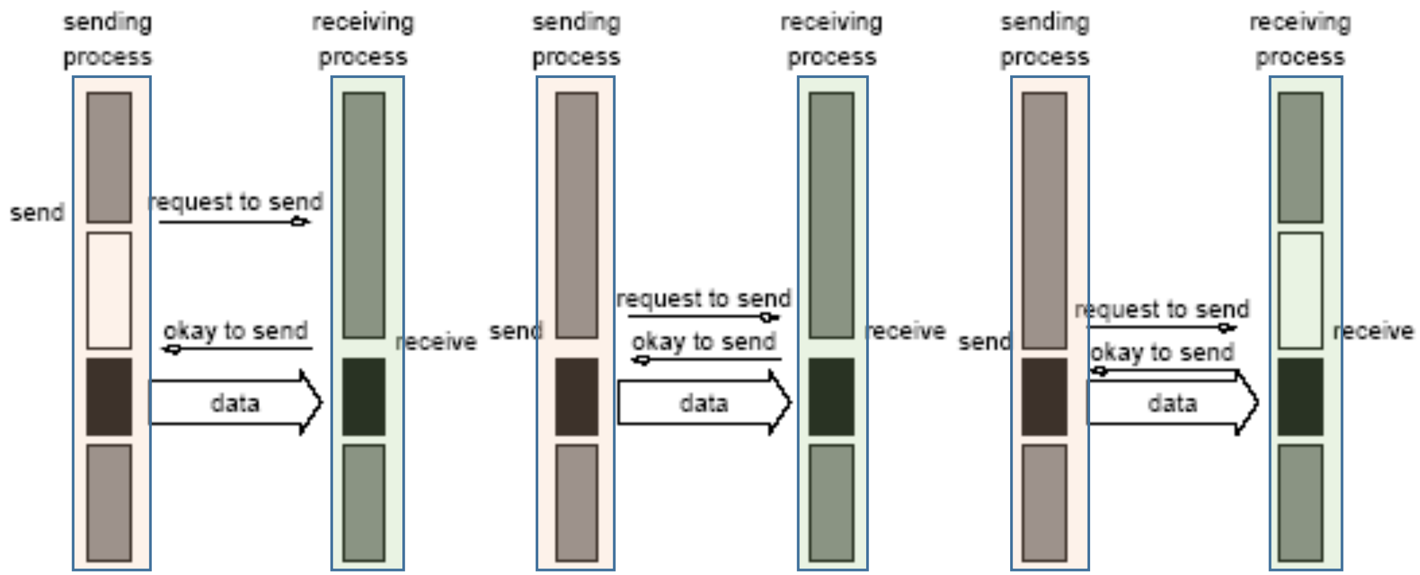
```
P0
a = 100;
send(&a, 1, 1);
a = 0;
```

```
P1
receive(&a, 1, 0)
printf("%d\n", a);
```

- What is the value of a in P0?
- What is the value of a in P1?

Non-Buffered **Blocking** Message Passing Operations

- Handshake for a blocking non-buffered send/receive operation.
- It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.



(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

Deadlocks

Send and Receive Refresher

```
send     (void *sendbuf, int nelems, int dest)
receive  (void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

```
P0
send(&a, 1, 1);
receive(&a, 1, 1)
```

```
P1
send(&a, 1, 0);
receive(&a, 1, 0);
```

Buffered Blocking Message Passing Operations

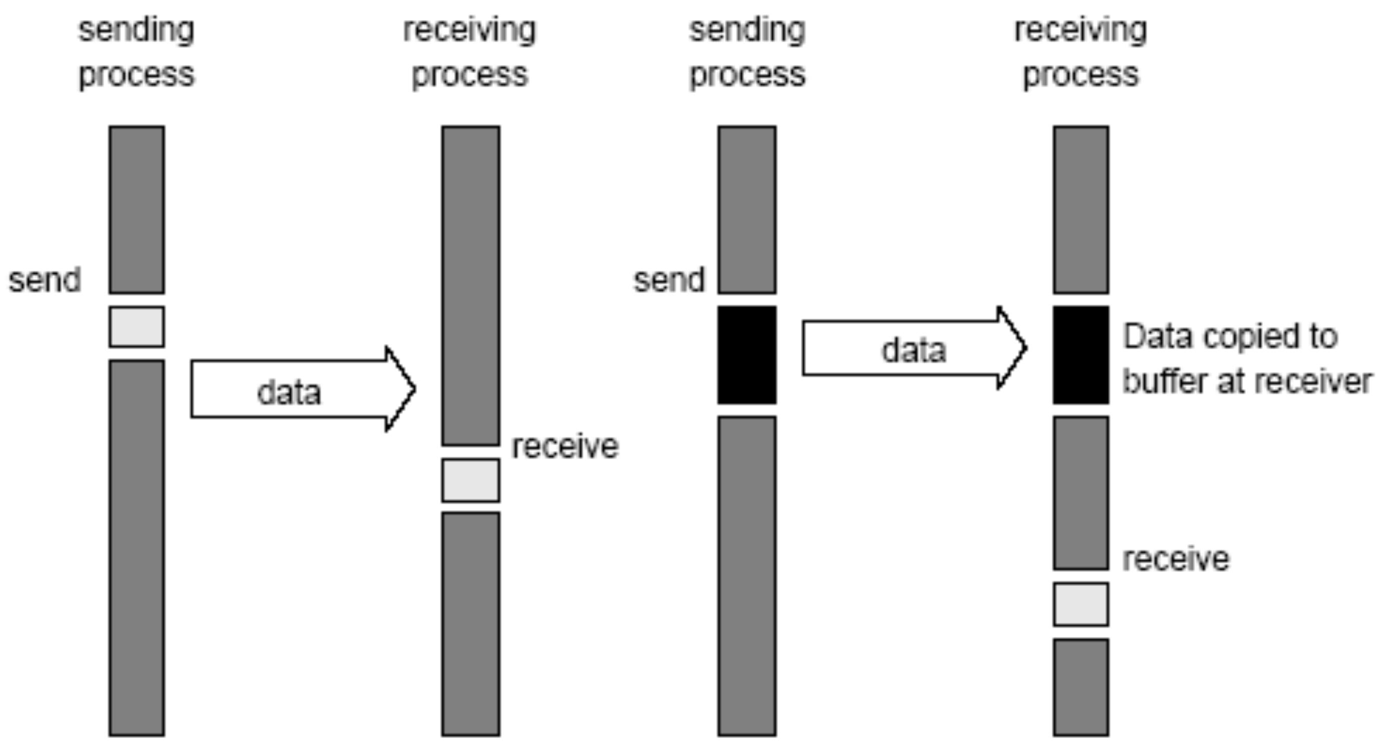
- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.

TRADEOFF: Idling overhead vs. buffer copying overhead.

Buffered Blocking Message Passing

Blocking buffered transfer:

- If DMA (like) is present;
- If no DMA → sender interrupts receiver and deposits data in buffer at receiver end.



Buffered Blocking Message Passing Operations

What if consumer was much slower than producer?

```
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);
}
```

Use Bounded Buffers

Bounded buffer sizes can have significant impact on performance.

Buffered Blocking Message Passing Operations

Can you hit a deadlock in a buffered environment.

```
P0

receive(&a, 1, 1);

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

```
P1

receive(&a, 1, 0);

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

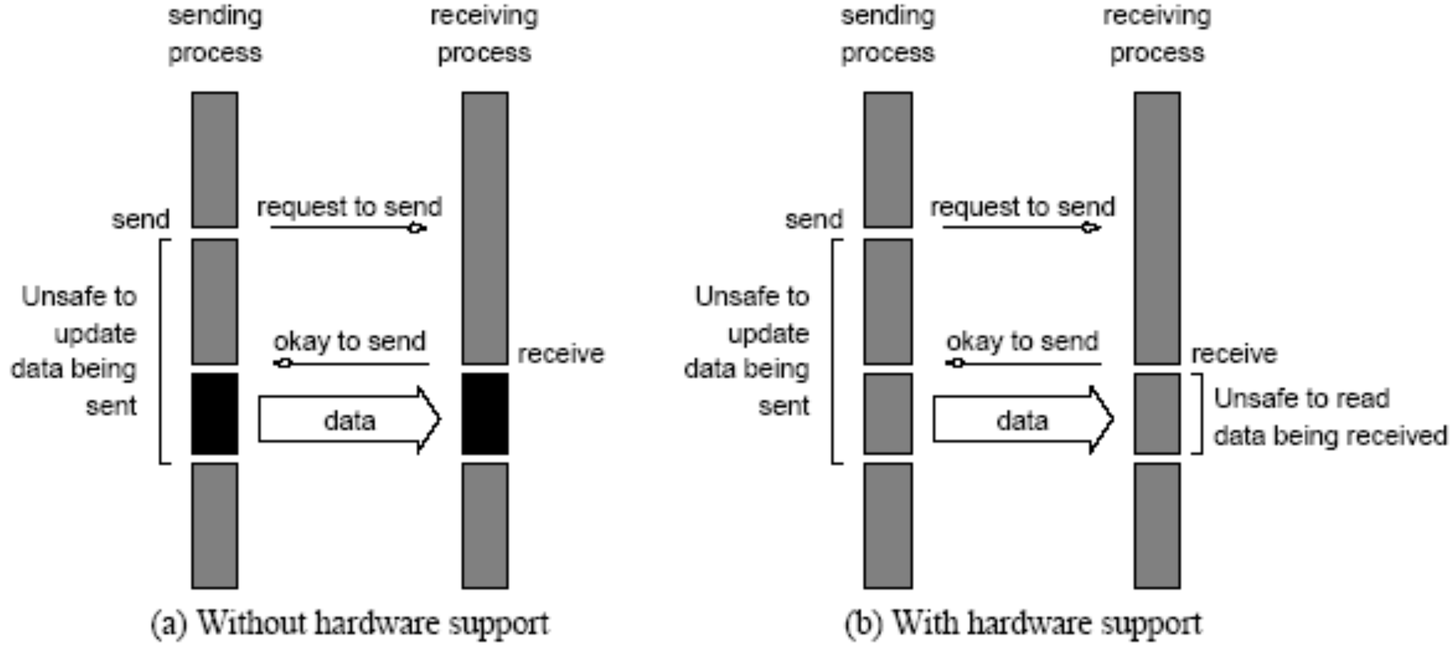
Non-Blocking
Message Passing

- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a check-status operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

Non-Blocking Message Passing Operations

Non-blocking non-buffered send and receive operations:

- Without hardware support
- With hardware support



Recap of Send and Receive Protocols

Implementations support both:

	Blocking	Non-Blocking
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non-Buffered	Sending process blocks until matching receive operation has been encountered	

Message Passing Libraries (1)

- Many “message passing libraries” were once available
  - Chameleon, from ANL.
  - CMMD, from Thinking Machines.
  - Express, commercial.
  - MPL, native library on IBM SP-2.
  - NX, native library on Intel Paragon.
  - Zipcode, from LLL.
  - PVM, Parallel Virtual Machine, public, from ORNL/UTK.
  - Others...
  - MPI, Message Passing Interface, now the industry standard.
- Need standards to write portable code.

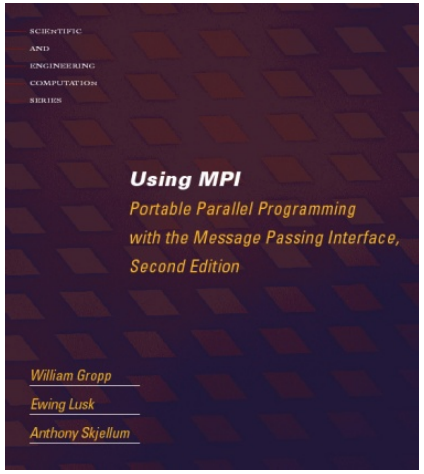
MPI: the Message Passing Interface

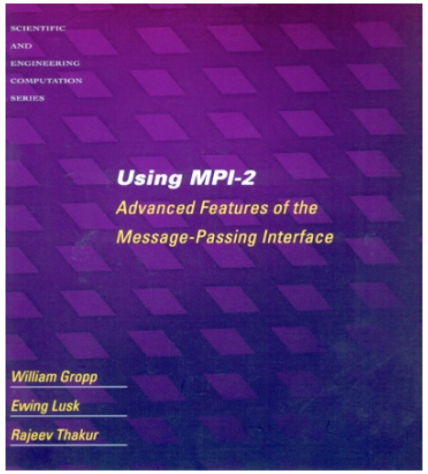
The Six MPI Subroutines to get you running

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

MPI References (in other words...*go beyond the Six*)

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
  - Latest version MPI 3.1, released June 2015
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages





Slide source: Bill Gropp, ANL

Message Passing Libraries (2)

- All communication, synchronization require subroutine calls
  - No shared variables
  - Program run on a single processor just like any uniprocessor program, except for calls to message passing library
- Subroutines for
  1. Communication
     - Pairwise or point-to-point: Send and Receive
     - Collectives all processor get together to
       - Move data: Broadcast, Scatter/gather
       - Reduction — Compute and move: sum, product, max, … of data on many processors
  2. Enquiries
     - How many processes? Which one am I? Any messages waiting?
  3. Synchronization
     - Barrier (No locks because there are no shared variables to protect)

Novel Features of MPI

- Datatypes reduce copying costs and permit heterogeneity
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication
- Topology conscious → permit efficient process placement, user views of process layout
- Communicators encapsulate communication spaces for library safety

Communicators

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.

Who am I?

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.

- The calling sequences of these routines are as follows:

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

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

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;
}
```

Hello (C++)


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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "I am " << rank << " of " << size << "\n";
    MPI::Finalize();
    return 0;
}
```

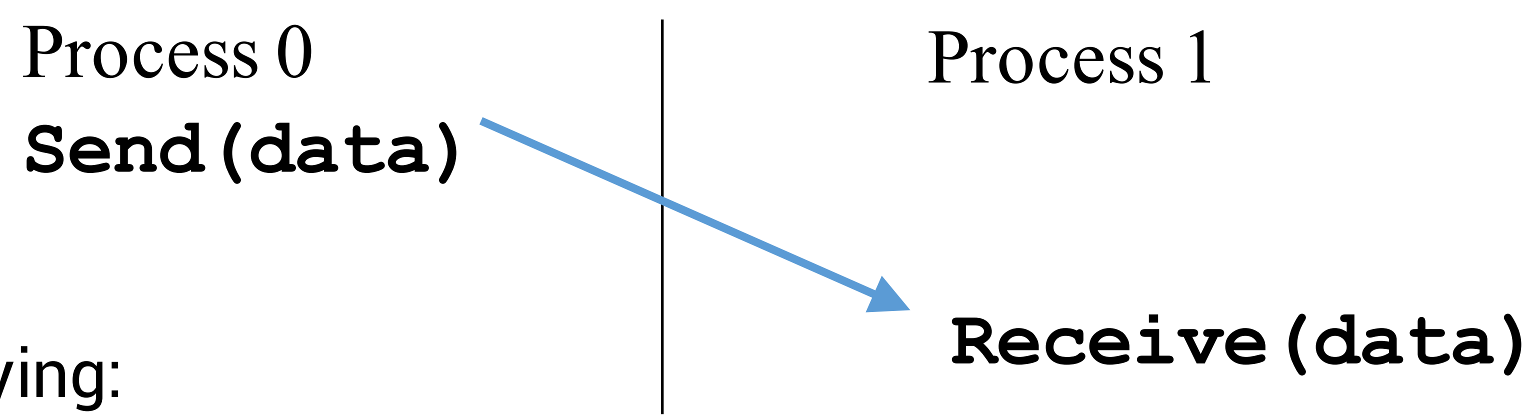
Notes on Hello World

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C/C++) or and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the printf/print statements
- The MPI-1 Standard does not specify how to run an MPI program, but many implementations provide

mpirun –np 4 a.out

MPI Basic Send/Receive

- We need to fill in the details in



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

MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays
- May hurt performance if datatypes are complex

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	

Sending and Receiving Messages

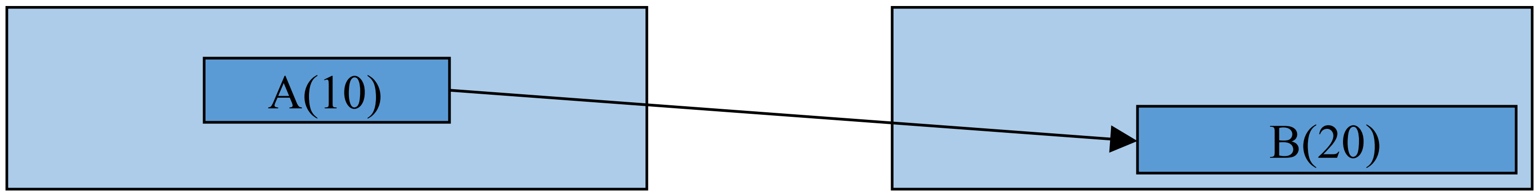
- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.
- The calling sequences of these routines are as follows:

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

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

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.
- The datatype `MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data.
- The message-tag can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`.

MPI Basic (Blocking) Send



MPI_Send(A, 10, MPI_DOUBLE, 1, …)

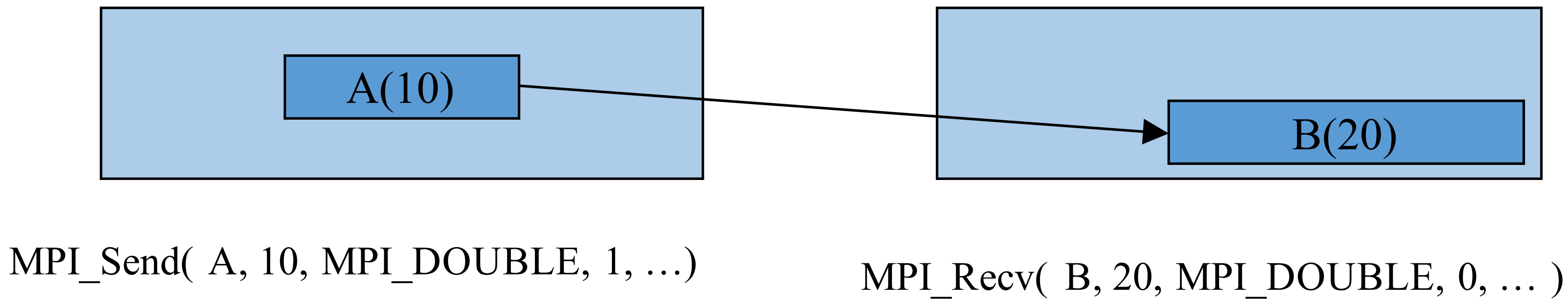
MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

- 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, 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.

Various MPI Sends (FYI...)

- MPI_Send:
- MPI_Bsend: May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- MPI_Ssend: will not return until matching receive posted
- MPI_Rsend: May be used ONLY if matching receive already posted. User responsible for writing a correct program.
- MPI_Isend: Nonblocking send.
- MPI_Ibsend: buffered nonblocking
- MPI_Issend: Synchronous nonblocking. Note that a Wait/Test will complete only when the matching receive is posted.
- MPI_Irsend: As with MPI_Rsend, but nonblocking.

MPI Basic (Blocking) 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 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)

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size, tag = 99;
  MPI_Status status;

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

  if (rank == 0)
  {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

  printf( "Message from process %d : %.13s\n", rank, message);

  MPI_Finalize();
}
```


Hello World in a little more detail

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size, tag = 99;
  MPI_Status status;

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

  if (rank == 0)
  {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

  printf( "Message from process %d : %.13s\n", rank, message);

  MPI_Finalize();
}
```


MPI_Init

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size, tag = 99;
  MPI_Status status;

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

  if (rank == 0)
  {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

  printf( "Message from process %d : %.13s\n", rank, message);

  MPI_Finalize();
}
```

MPI_Comm_size

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size, tag = 99;
  MPI_Status status;

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

  if (rank == 0)
  {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

  printf( "Message from process %d : %.13s\n", rank, message);

  MPI_Finalize();
}
```


MPI_Comm_rank

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size, tag = 99;
  MPI_Status status;

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

  if (rank == 0)
  {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

  printf( "Message from process %d : %.13s\n", rank, message);

  MPI_Finalize();
}
```

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

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size, tag = 99;
  MPI_Status status;

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

  if (rank == 0)
  {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

  printf( "Message from process %d : %.13s\n", rank, message);

  MPI_Finalize();
}
```


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

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size, tag = 99;
  MPI_Status status;

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

  if (rank == 0)
  {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

  printf( "Message from process %d : %.13s\n", rank, message);

  MPI_Finalize();
}
```

MPI_Finalize

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive
- Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with datatypes

Status - a data structure allocated in the user's program

- In C:
  - int recvd_tag, recvd_from, recvd_count;
  - MPI_Status status;
  - MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status)
  - recvd_tag = status.MPI_TAG;
  - recvd_from = status.MPI_SOURCE;
  - MPI_Get_count(&status, datatype, &recvd_count);

Status

- On the receiving end, the status variable can be used to get information about the `MPI_Recv` operation.
- The corresponding data structure contains:

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

- The MPI_Get_count function returns the precise count of data items received.

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

Another Approach to Parallelism

- Collective routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations…

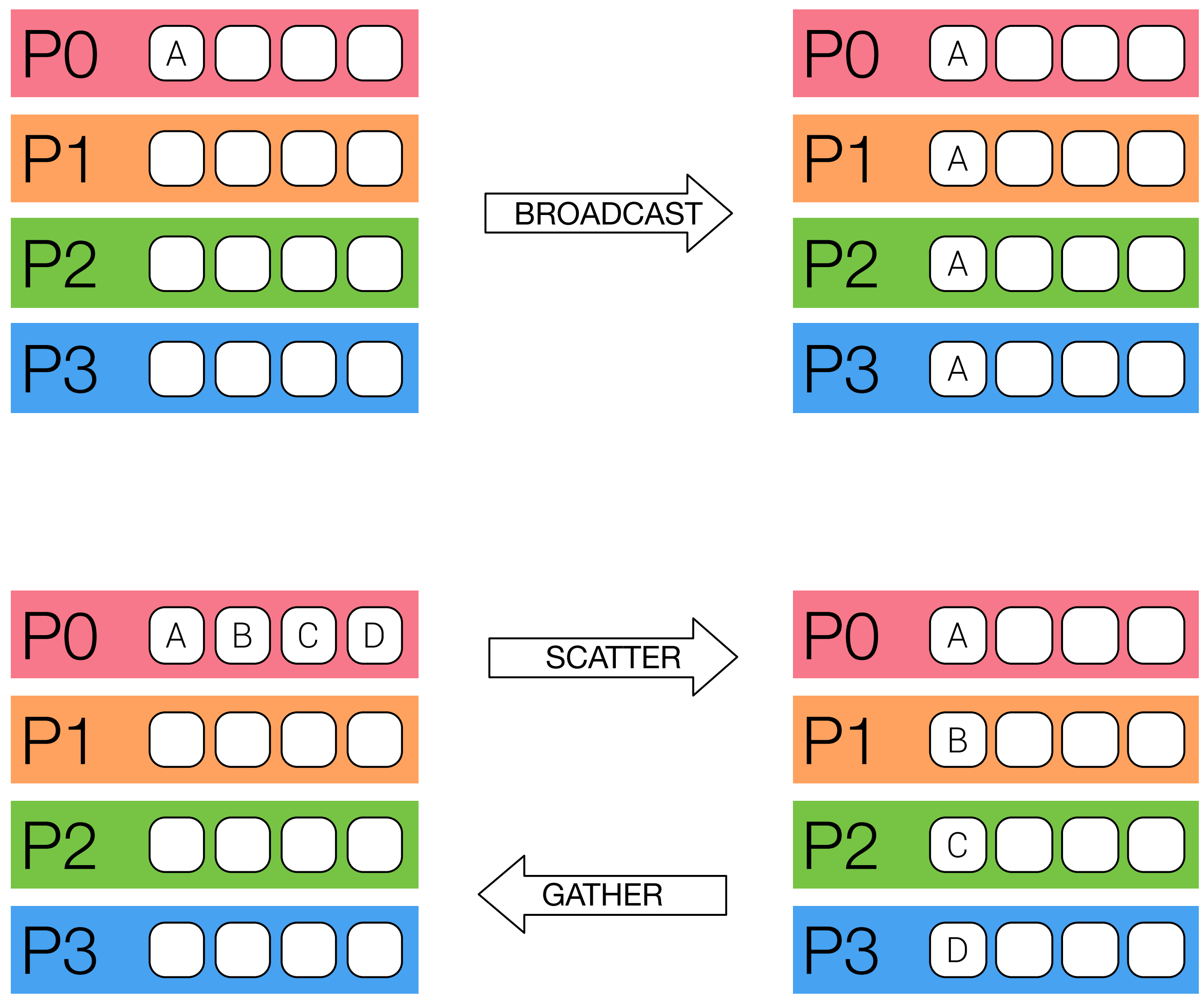
Examples of Collective Ops. in MPI

- Collective operations are called by all processes in a communicator
- MPI_BCAST distributes data from one process (the root) to all others in a communicator
- MPI_REDUCE combines data from all processes in communicator and returns it to one process
- In many numerical algorithms, SEND/RECEIVE can be replaced by BCAST/REDUCE, improving both simplicity and efficiency

Synchronization

- MPI_Barrier(comm)
- Blocks until all processes in the group of the communicator comm call it.
- Almost never required in a parallel program
    - Occasionally useful in measuring performance and load balancing

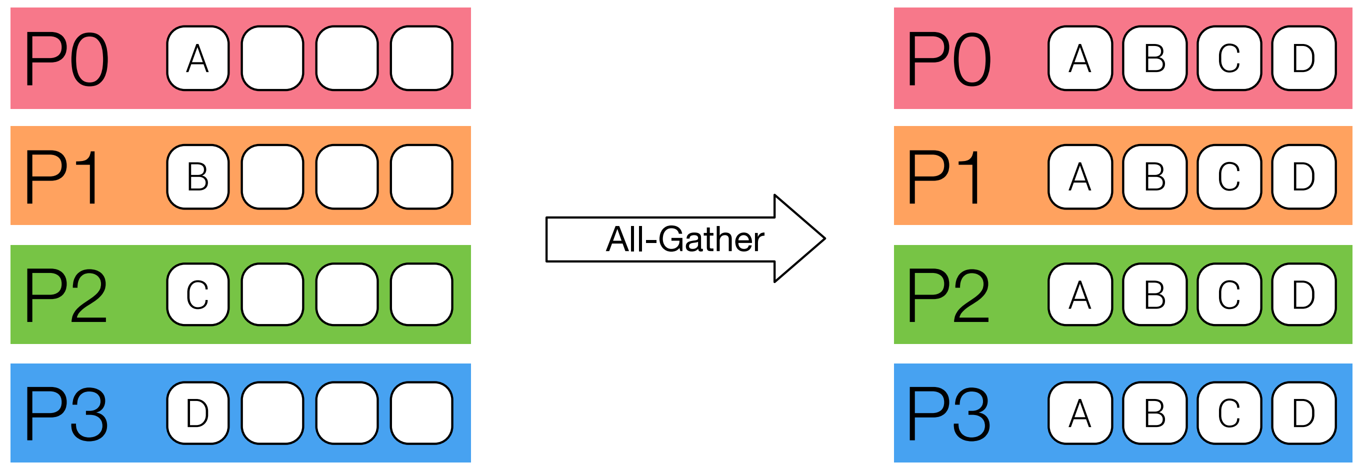
Collective Data Movement

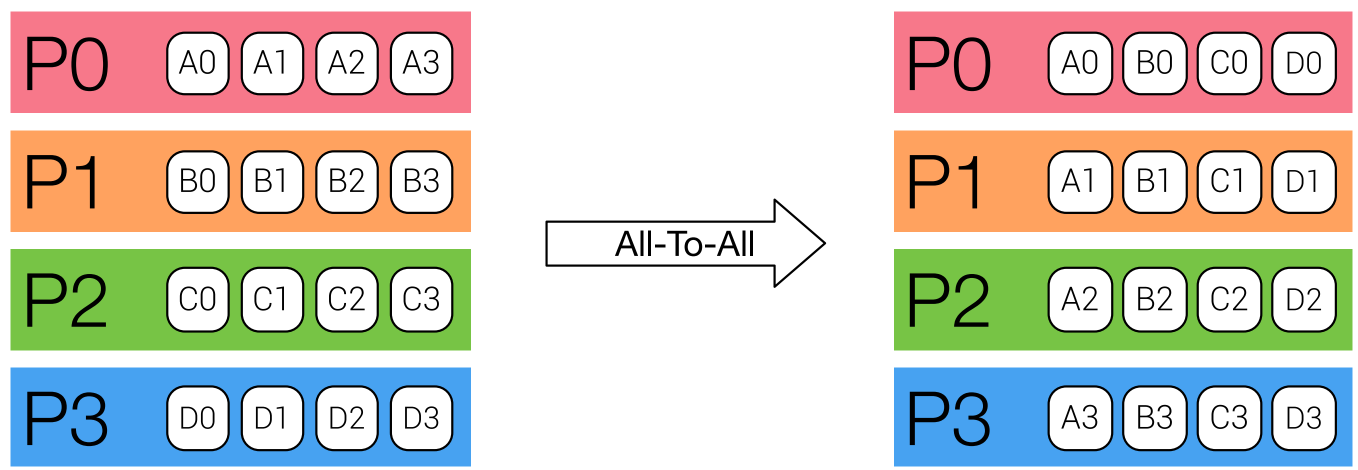


Comments on Broadcast, other Collectives

- 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
  - “root” argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive

More Collective Data Movement





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, not just root.
- V versions allow the chunks to have variable sizes.
- Allreduce, Reduce, Reduce_scatter, and Scan take both built-in and user-defined combiner functions.
- MPI-2 adds Alltoallw, Exscan, intercommunicator versions of most routines

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