CS 267: Distributed Memory Machines and Programming

Jonathan Carter

jtcarter@lbl.gov

www.cs.berkeley.edu/~skamil/cs267

Programming Distributed Memory Machines with Message Passing

Most slides from Kathy Yelick's 2007 lecture

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.

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
  - Communication
    - Pairwise or point-to-point: Send and Receive
    - Collectives all processor get together to
      - Move data: Broadcast, Scatter/gather
      - Compute and move: sum, product, max, … of data on many processors
  - Synchronization
    - Barrier
    - No locks because there are no shared variables to protect
  - Enquiries
    - How many processes? Which one am I? Any messages waiting?

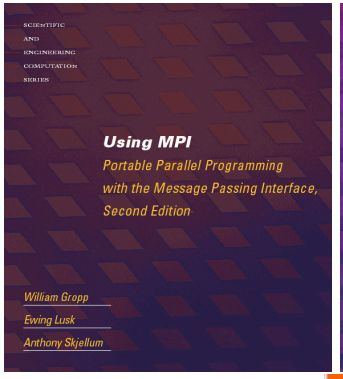
Novel Features of MPI

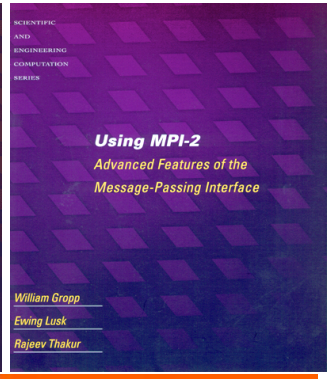
- Communicators encapsulate communication spaces for library safety
- Datatypes reduce copying costs and permit heterogeneity
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication
- Process topologies permit efficient process placement, user views of process layout
- Profiling interface encourages portable tools

MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- 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

Books on MPI





- *Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition)*, by Gropp, Lusk, and Skjellum, MIT Press, 1999.
- *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- *MPI: The Complete Reference - Vol 1 The MPI Core,* by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- *MPI: The Complete Reference - Vol 2 The MPI Extensions*, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
- *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.

Programming With MPI

- MPI is a library
  - All operations are performed with routine calls
  - Basic definitions in
    - mpi.h for C
    - mpif.h for Fortran 77 and 90
    - MPI module for Fortran 90 (optional)
- First Program:
  - Write out process number
  - Write out some variables (illustrate separate name space)

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

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 (Fortran)

```
program main
include 'mpif.h'
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

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;
}
```

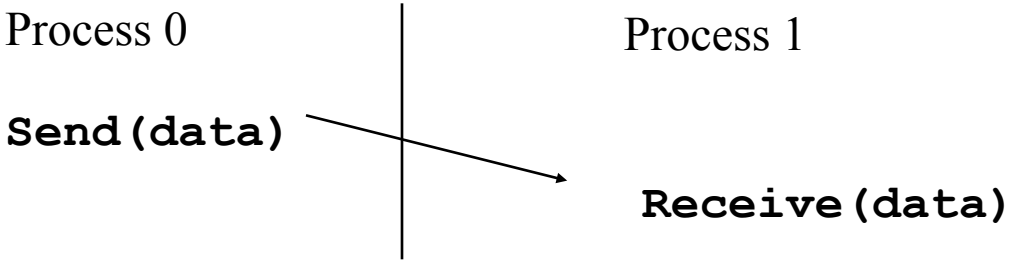
Notes on Hello World

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `printf/print` statements
- I/O not part of MPI-1but is in MPI-2
  - print and write to standard output or error not part of either MPI-1 or MPI-2
  - output order is undefined (may be interleaved by character, line, or blocks of characters),
- 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:
  - 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?

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**

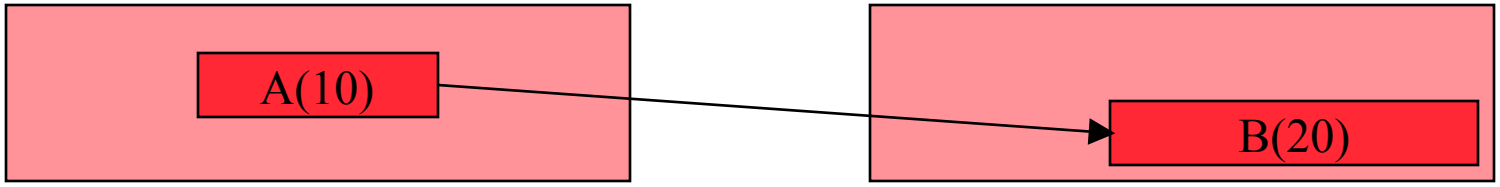
MPI Datatypes

- 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 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

MPI Basic (Blocking) Send



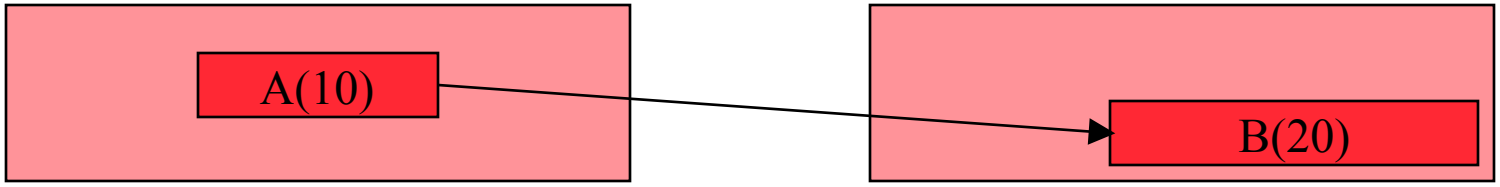
MPI_Send(A, 10, MPI_DOUBLE, 1, …)

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.

MPI Basic (Blocking) Receive



MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

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)

A Simple MPI Program

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              &status );
    printf( "Received %d\n", buf );
  }

  MPI_Finalize();
  return 0;
}
```

A Simple MPI Program (Fortran)

```
      program main
      include 'mpif.h'
      integer rank, buf, ierr, status(MPI_STATUS_SIZE)

      call MPI_Init(ierr)
      call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
C Process 0 sends and Process 1 receives
      if (rank .eq. 0) then
         buf = 123456
         call MPI_Send( buf, 1, MPI_INTEGER, 1, 0,
     *                  MPI_COMM_WORLD, ierr )
      else if (rank .eq. 1) then
         call MPI_Recv( buf, 1, MPI_INTEGER, 0, 0,
     *                  MPI_COMM_WORLD, status, ierr )
         print *, "Received ", buf
      endif
      call MPI_Finalize(ierr)
      end
```

A Simple MPI Program (C++)

```
#include "mpi.h"
#include <iostream>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI::Init(argv, argc);
  rank = MPI::COMM_WORLD.Get_rank();

  // Process 0 sends and Process 1 receives
  if (rank == 0) {
    buf = 123456;
    MPI::COMM_WORLD.Send( &buf, 1, MPI::INT, 1, 0 );
  }
  else if (rank == 1) {
    MPI::COMM_WORLD.Recv( &buf, 1, MPI::INT, 0, 0 );
    std::cout << "Received " << buf << "\n";
  }

  MPI::Finalize();
  return 0;
}
```

Retrieving Further Information

- `Status` is 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 );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..
  status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C++:

```
int recvd_tag, recvd_from, recvd_count;
MPI::Status status;
Comm.Recv(..., MPI::ANY_SOURCE, MPI::ANY_TAG, ...,
          status )
recvd_tag   = status.Get_tag();
recvd_from  = status.Get_source();
recvd_count = status.Get_count( datatype );
```

Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of “wild card” tags.
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application

Running MPI Programs

- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- **`mpiexec <args>`** is part of MPI-2, as a recommendation, but not a requirement, for implementors.
- Use

  mpirun –np # -nolocal a.out

  for your clusters, e.g.

  mpirun –np 3 –nolocal cpi

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`**

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…

Collective Operations 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

Example: PI in C - 1

```
#include "mpi.h"
#include <math.h>
   #include <stdio.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

Example: PI in C - 2

```
     h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is .16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
   return 0;
}
```

Example: PI in Fortran - 1

```
      program main
      include 'mpif.h'
      integer done, n, myid, numprocs, i, rc
      double pi25dt, mypi, pi, h, sum, x, z
      data done/.false./
      data PI25DT/3.14159265358979323846264 3/
      call MPI_Init(ierr)
      call MPI_Comm_size(MPI_COMM_WORLD,numprocs, ierr )
      call MPI_Comm_rank(MPI_COMM_WORLD,myid, ierr)
      do while (.not. done)
        if (myid .eq. 0) then
         print *,"Enter the number of intervals: (0 quits)"
         read *, n
        endif
        call MPI_Bcast(n, 1, MPI_INTEGER, 0,
     *                 MPI_COMM_WORLD, ierr )
        if (n .eq. 0) goto 10
```

Example: PI in Fortran - 2

```
      h   = 1.0 / n
      sum = 0.0
      do i=myid+1,n,numprocs
        x = h * (i - 0.5)
        sum += 4.0 / (1.0 + x*x)
      enddo
      mypi = h * sum
      call MPI_Reduce(mypi, pi, 1, MPI_DOUBLE_PRECISION,
     *                MPI_SUM, 0, MPI_COMM_WORLD, ierr )
      if (myid .eq. 0) then
          print *, "pi is approximately ", pi,
     *       ", Error is ", abs(pi - PI25DT)
     enddo
14   continue
      call MPI_Finalize( ierr )
     end
```

Example: PI in C++ - 1

```
#include "mpi.h"
#include <math.h>
#include <iostream>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  MPI::Init(argc, argv);
  numprocs = MPI::COMM_WORLD.Get_size();
  myid     = MPI::COMM_WORLD.Get_rank();
  while (!done)  {
    if (myid == 0) {
      std::cout << "Enter the number of intervals: (0
quits) ";
      std::cin >> n;;
    }
    MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0 );
    if (n == 0) break;
```

Example: PI in C++ - 2

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE,
                           MPI::SUM, 0);
    if (myid == 0)
      std::cout << "pi is approximately " << pi <<
            ", Error is " << fabs(pi - PI25DT) << "\n";
  }
  MPI::Finalize();
   return 0;
}
```

Notes on C and Fortran

- C and Fortran bindings correspond closely
- In C:
  - mpi.h must be #included
  - MPI functions return error codes or **`MPI_SUCCESS`**
- In Fortran:
  - mpif.h must be included, or use MPI module
  - All MPI calls are to subroutines, with a place for the return code in the last argument.
- C++ bindings, and Fortran-90 issues, are part of MPI-2.

Alternative Set of 6 Functions

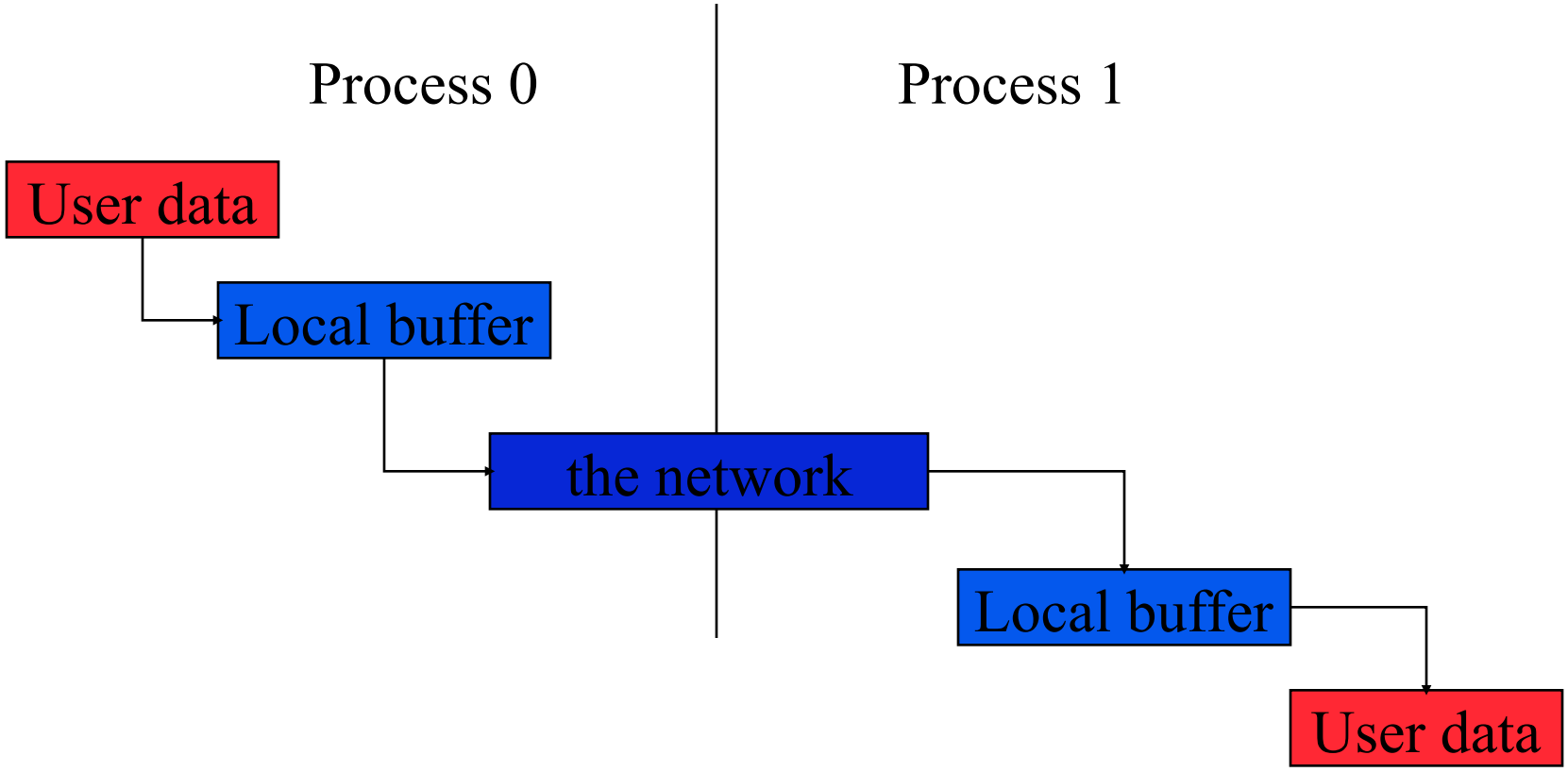
- Using collectives:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_BCAST`**
  - **`MPI_REDUCE`**

More on Message Passing

- Message passing is a simple programming model, but there are some special issues
  - Buffering and deadlock
  - Deterministic execution
  - Performance

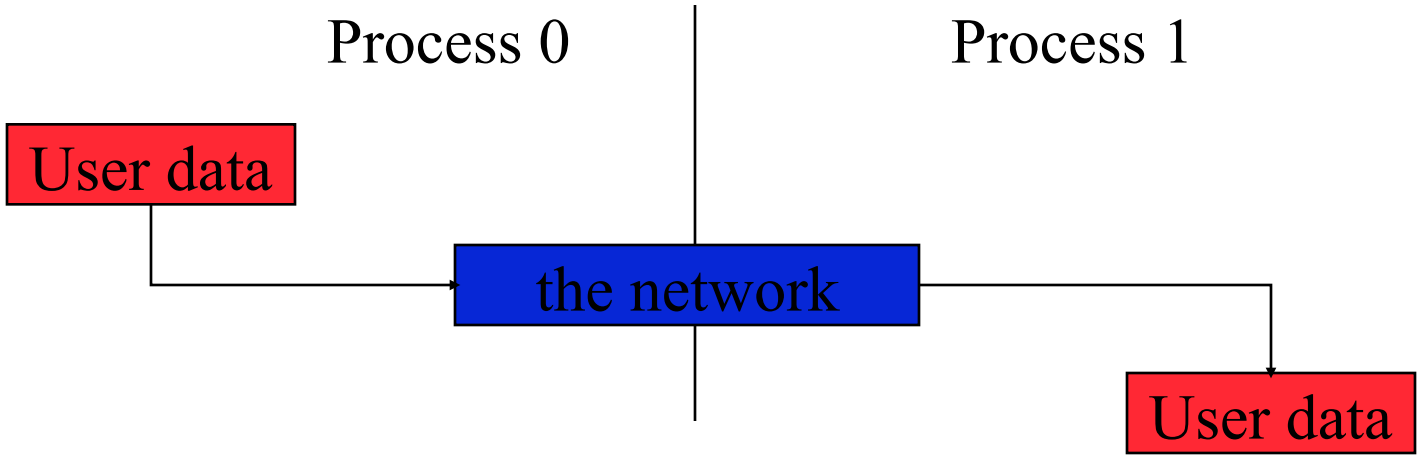
Buffers

- When you send data, where does it go? One possibility is:



Avoiding Buffering

- It is better to avoid copies:



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

Blocking and Non-blocking Communication

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

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 in which to store the data sent until it can be received

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`

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

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

(each request must be Waited on)

- One can also test without waiting:

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

MPI's Non-blocking Operations (Fortran)

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
integer request
integer status(MPI_STATUS_SIZE)
call MPI_Isend(start, count, datatype,
    dest, tag, comm, request,ierr)
call MPI_Irecv(start, count, datatype,
    dest, tag, comm, request, ierr)
call MPI_Wait(request, status, ierr)
```

(Each request must be waited on)

- One can also test without waiting:

```
call MPI_Test(request, flag, status, ierr)
```

MPI's Non-blocking Operations (C++)

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
MPI::Request request;
MPI::Status  status;
request = comm.Isend(start, count,
                     datatype, dest, tag);
request = comm.Irecv(start, count,
                     datatype, dest, tag);
request.Wait(status);
```

  (each request must be Waited on)

- One can also test without waiting:

```
flag = request.Test( status );
```

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 Completions (Fortran)

- It is sometimes desirable to wait on multiple requests:

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

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

Communication Modes

- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**`MPI_Ssend`**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (**`MPI_Bsend`**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
  - Ready mode (**`MPI_Rsend`**): user guarantees that a matching receive has been posted.
    - Allows access to fast protocols
    - undefined behavior if matching receive not posted
- Non-blocking versions (**`MPI_Issend`**, etc.)
- **`MPI_Recv`** receives messages sent in any mode.

Other Point-to Point Features

- **`MPI_Sendrecv`**
- **`MPI_Sendrecv_replace`**
- **`MPI_Cancel`**
  - Useful for multibuffering
- Persistent requests
  - Useful for repeated communication patterns
  - Some systems can exploit to reduce latency and increase performance

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)`

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.
- Almost never required in a parallel program
  - Occasionally useful in measuring performance and load balancing

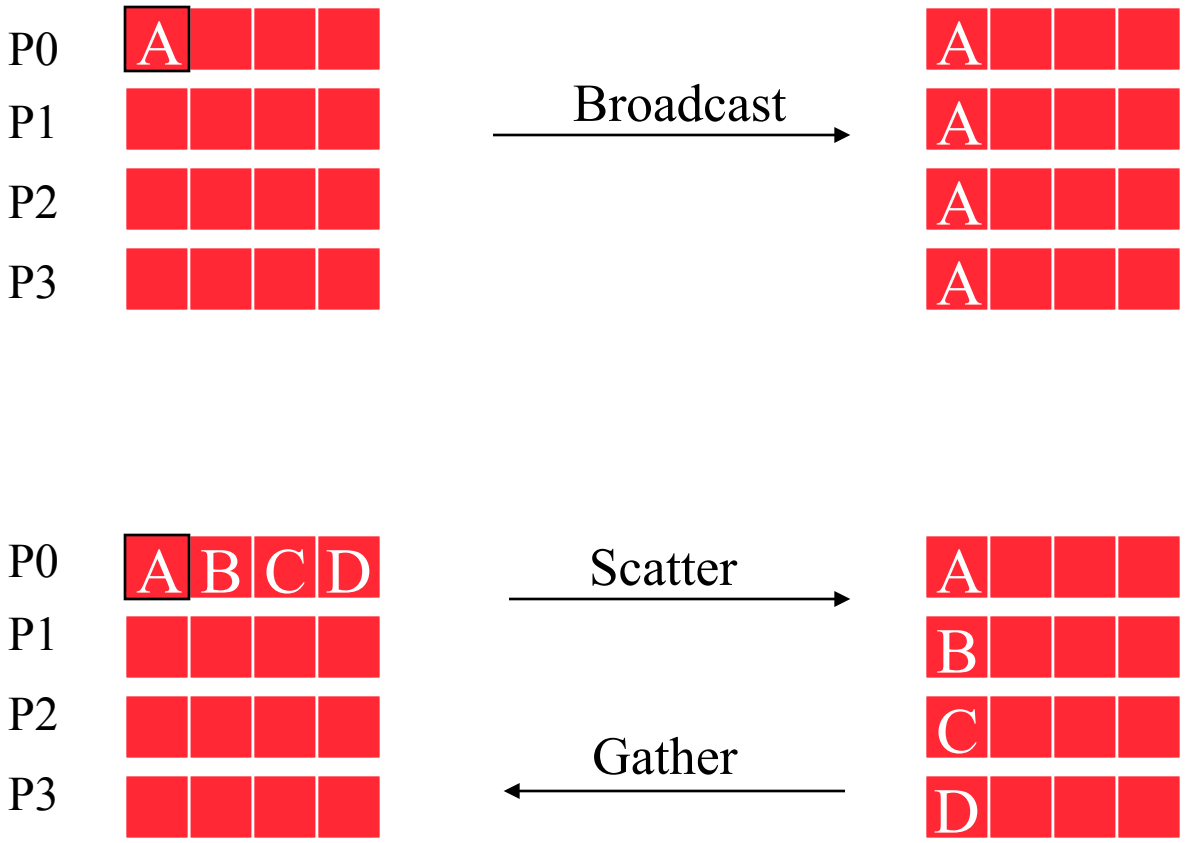
Synchronization (Fortran)

- `MPI_Barrier( comm, ierr )`
- Blocks until all processes in the group of the communicator `comm` call it.

Synchronization (C++)

- `comm.Barrier();`
- Blocks until all processes in the group of the communicator `comm` call it.

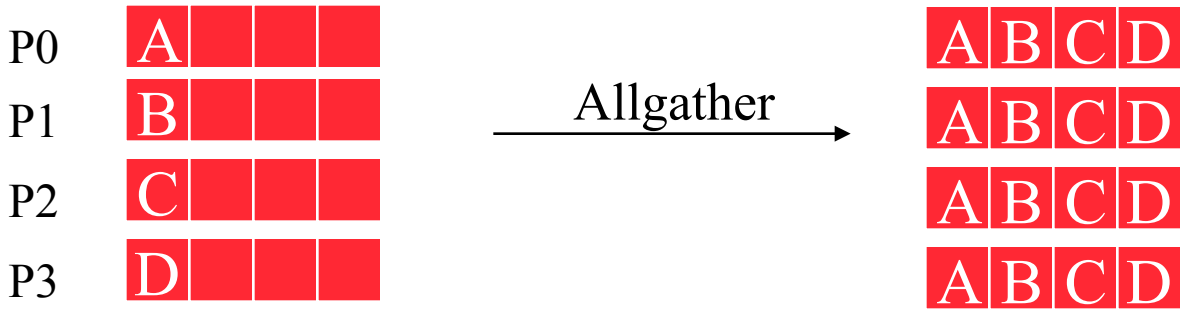
Collective Data Movement

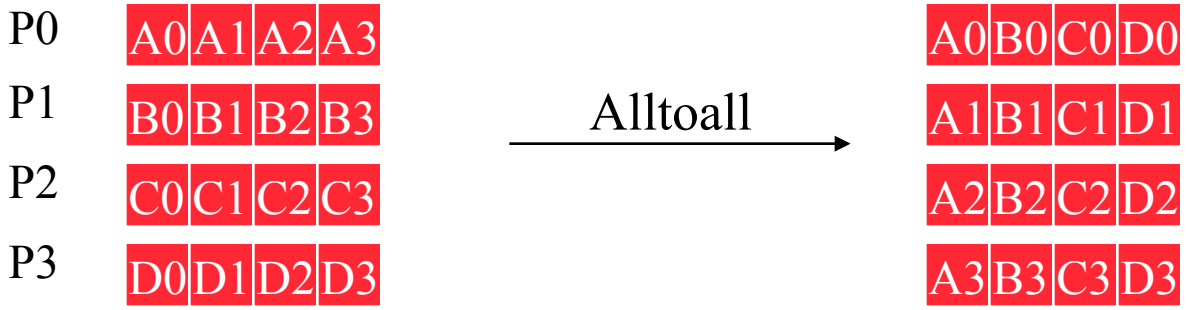


Comments on Broadcast

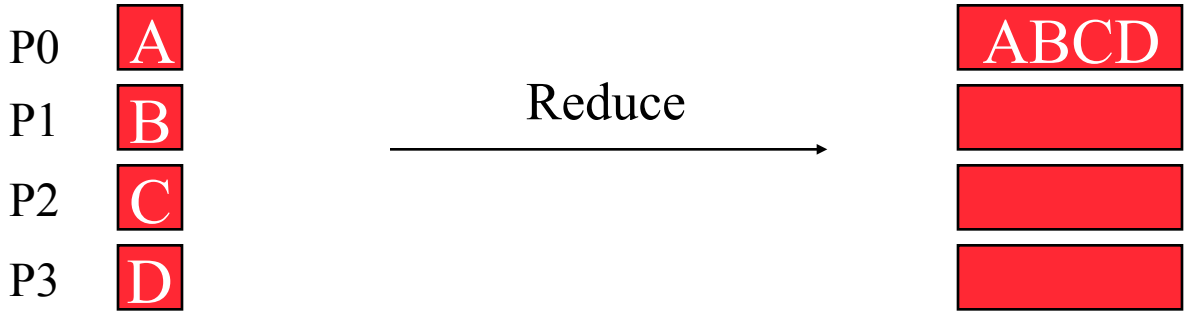
- All collective operations must be called by *all* processes in the communicator
- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
  - MPI_Bcast is not a “multi-send”
  - “root” argument is the rank of the sender; this tells MPI which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

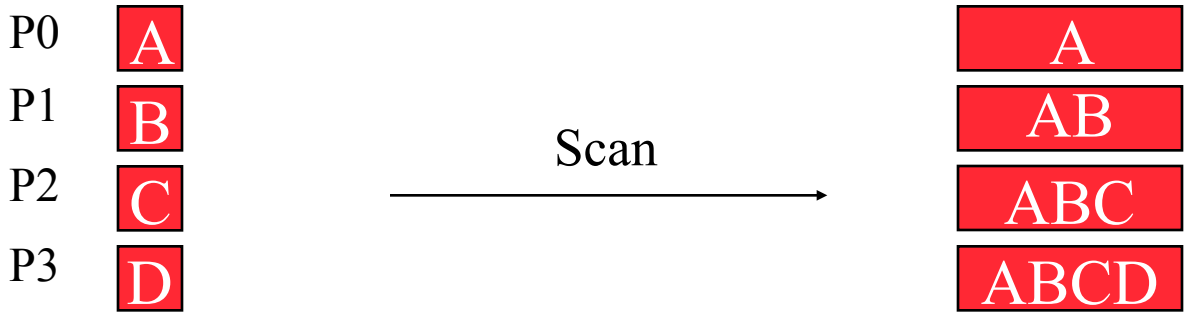
More Collective Data Movement





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 hunks to have different 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

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

The Collective Programming Model

- One style of higher level programming is to use *only* collective routines
- Provides a “data parallel” style of programming
  - Easy to follow program flow

What MPI Functions are in Use?

- For simple applications, these are common:
  - Point-to-point communication
    - MPI_Irecv, MPI_Isend, MPI_Wait, MPI_Send, MPI_Recv
  - Startup
    - MPI_Init, MPI_Finalize
  - Information on the processes
    - MPI_Comm_rank, MPI_Comm_size, MPI_Get_processor_name
  - Collective communication
    - MPI_Allreduce, MPI_Bcast, MPI_Allgather

Not Covered

- Topologies: map a communicator onto, say, a 3D Cartesian processor grid
  - Implementation can provide ideal logical to physical mapping
- Rich set of I/O functions: individual, collective, blocking and non-blocking
  - Collective I/O can lead to many small requests being merged for more efficient I/O
- One-sided communication: puts and gets with various synchronization schemes
- Task creation and destruction: change number of tasks during a run
  - Few implementations available

Backup Slides

Implementing Synchronous Message Passing

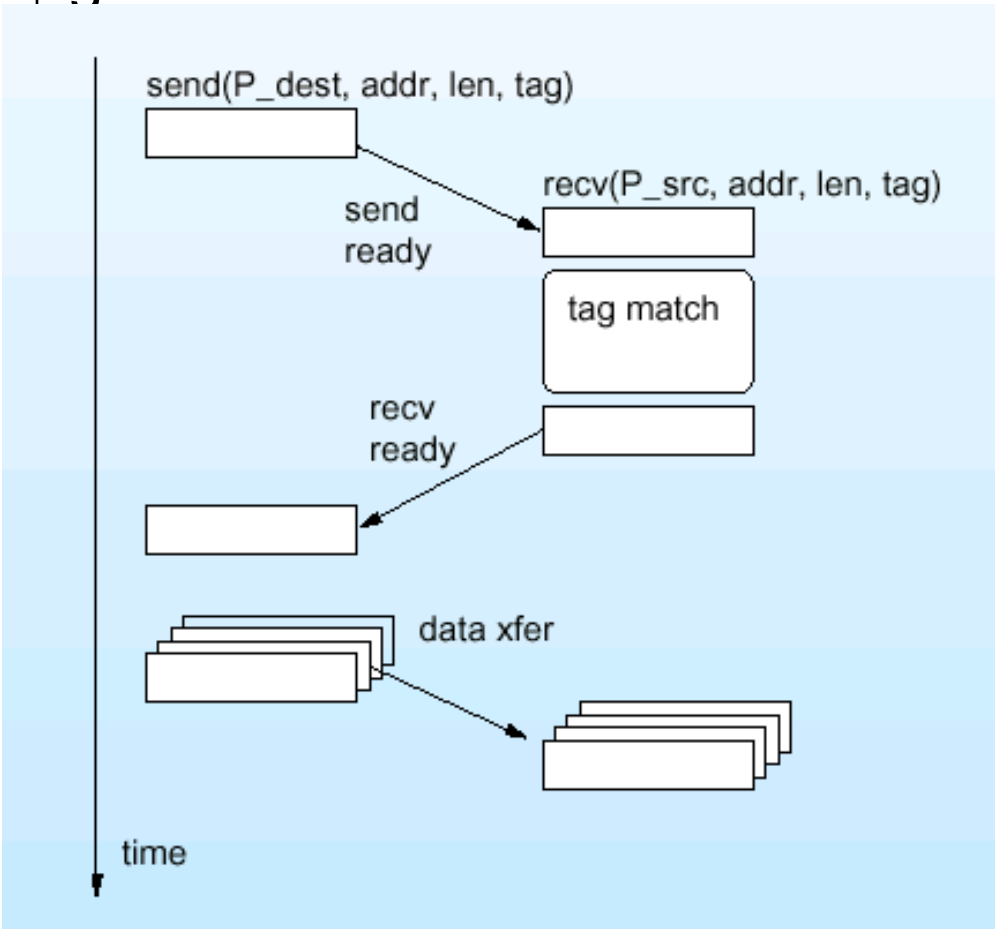
- Send operations complete after matching receive and source data has been sent.
- Receive operations complete after data transfer is complete from matching send.

1) Initiate send
2) Address translation on P_{dest}
3) Send-Ready Request

4) Remote check for posted receive

5) Reply transaction

6) Bulk data transfer

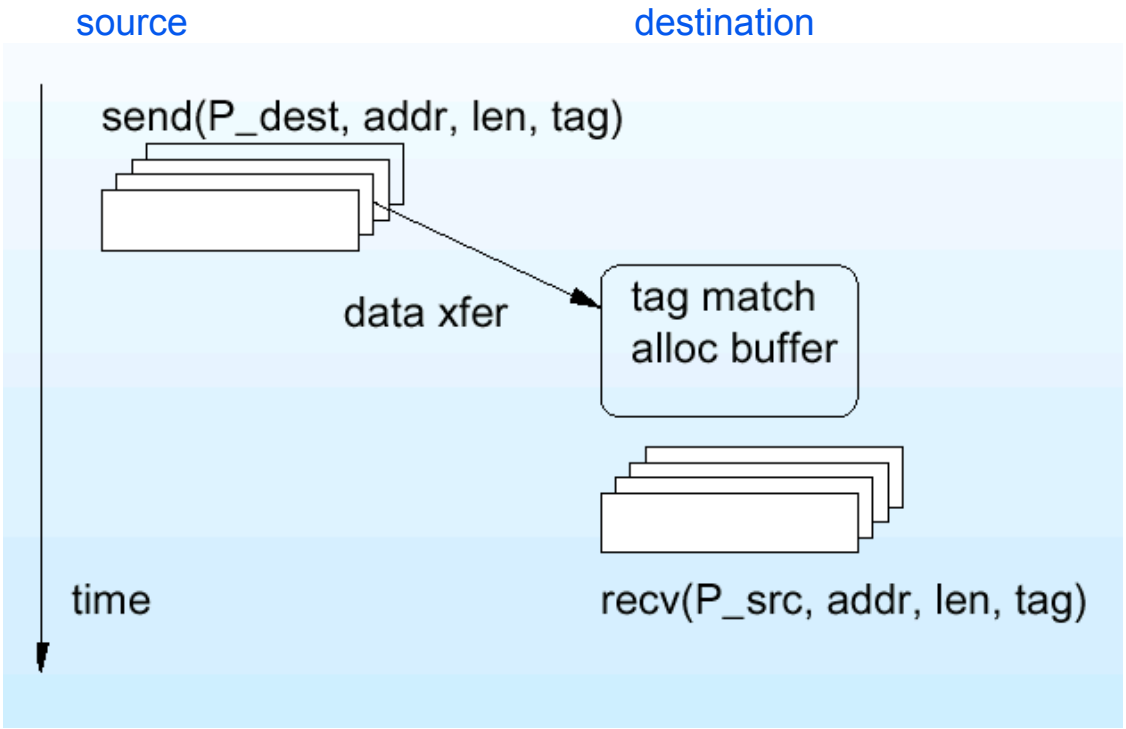


Implementing Asynchronous Message Passing

- Optimistic single-phase protocol assumes the destination can buffer data on demand.

1) Initiate send
2) Address translation on P_{dest}
3) Send Data Request

4) Remote check for posted receive
5) Allocate buffer (if check failed)
6) Bulk data transfer



Safe Asynchronous Message Passing

- Use 3-phase protocol
- Buffer on sending side
- Variations on send completion
  - wait until data copied from user to system buffer
  - don't wait -- let the user beware of modifying data

1) Initiate send
2) Address translation on P_{dest}
3) Send-Ready Request

4) Remote check for posted receive
   record send-rdy
5) Reply transaction

6) Bulk data transfer