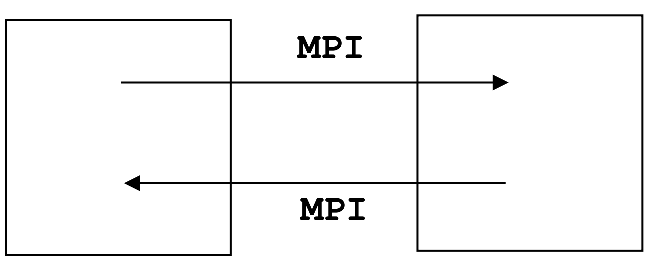
Introduction to MPI

Rajeev Thakur
Argonne National Laboratory

(excerpted and condensed by Brad Chamberlain for CSEP524, Winter 2013)

The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - synchronization
  - movement of data from one process's address space to another's.



What is MPI?

- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

Where Did MPI Come From?

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable).
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts.
  - Did not address the full spectrum of message-passing issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum organized in 1992 with broad participation by:
  - vendors: IBM, Intel, TMC, SGI, Convex, Meiko
  - portability library writers: PVM, p4
  - users: application scientists and library writers
  - MPI-1 finished in 18 months

MPI Implementations

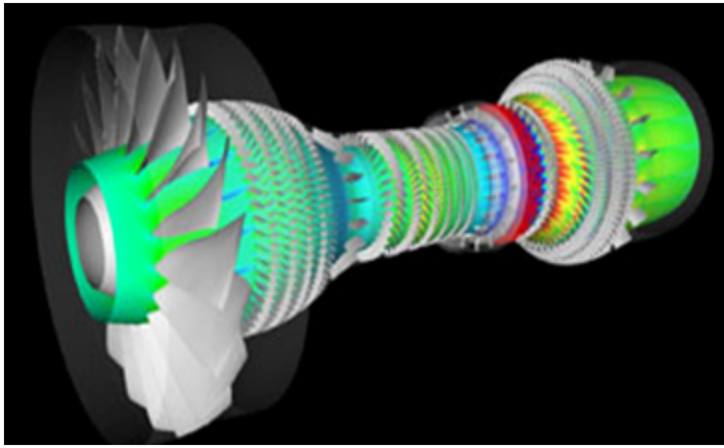
- MPI is available on all platforms – from laptops to clusters to the largest supercomputers in the world
- Currently, two prominent open-source implementations
  - MPICH2 from Argonne
    - www.mcs.anl.gov/mpich2
  - Open MPI
    - www.open-mpi.org
- Many vendor implementations (many derived from MPICH2)
  - IBM, Cray, Intel, Microsoft, Myricom, SGI, HP, etc
- MVAPICH2 from Ohio State Univ. for InfiniBand
  - http://mvapich.cse.ohio-state.edu/

MPI Resources

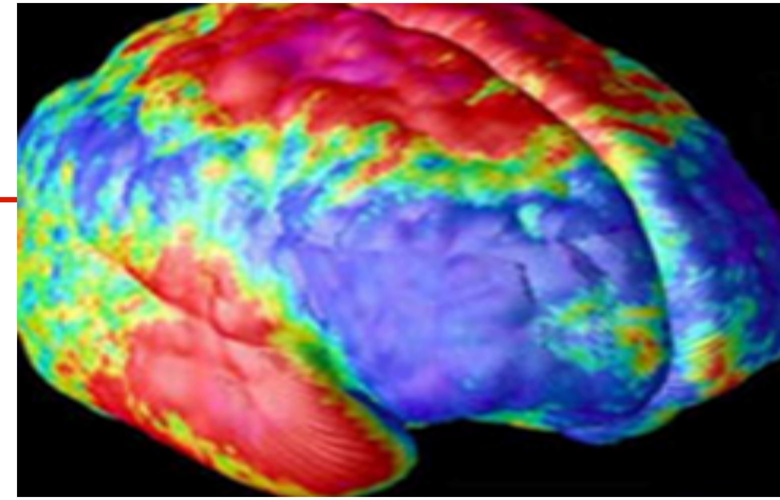
- The Standard itself:
  - At http://www.mpi-forum.org
    - All MPI official releases. Latest version is MPI 3.0
    - Download pdf versions
- Online Resources
  - http://www.mcs.anl.gov/mpi
    - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages
  - Tutorials: http://www.mcs.anl.gov/mpi/learning.html
  - Google search will give you many more leads

Applications (Science and Engineering)

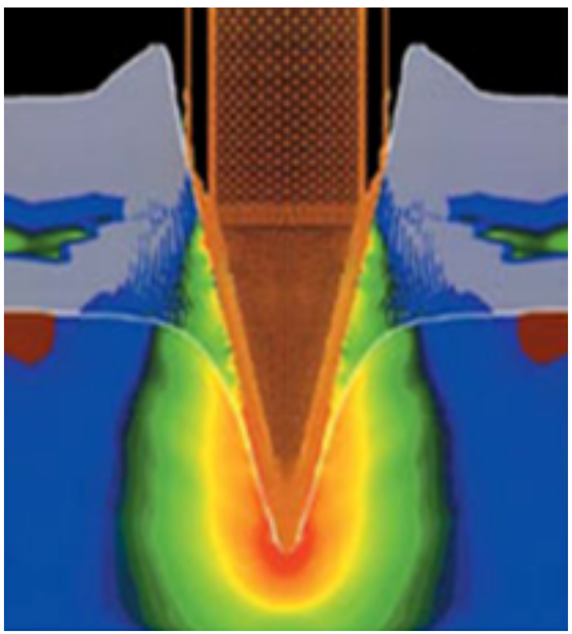
- MPI is widely used used in large scale parallel applications in science and engineering
- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical Engineering - from prosthetics to spacecraft
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics



Turbo machinery (Gas turbine/compressor)

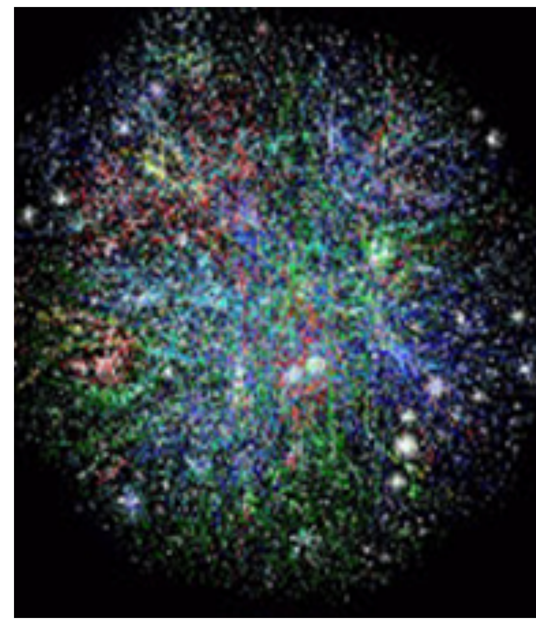


Biology application

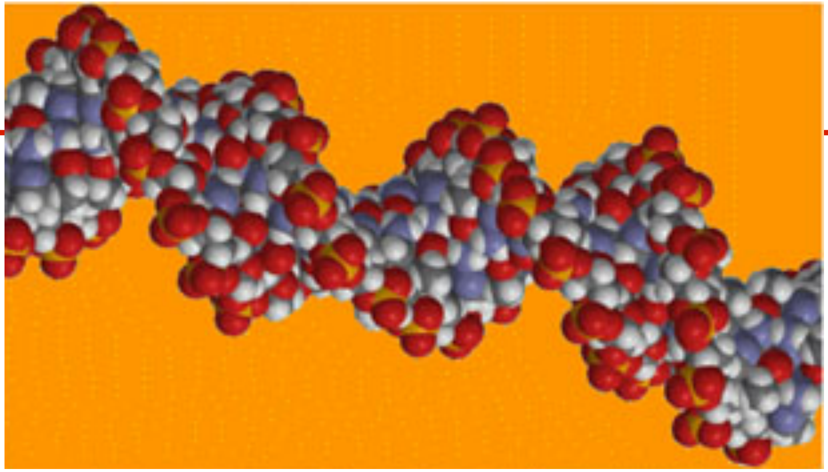


Drilling application

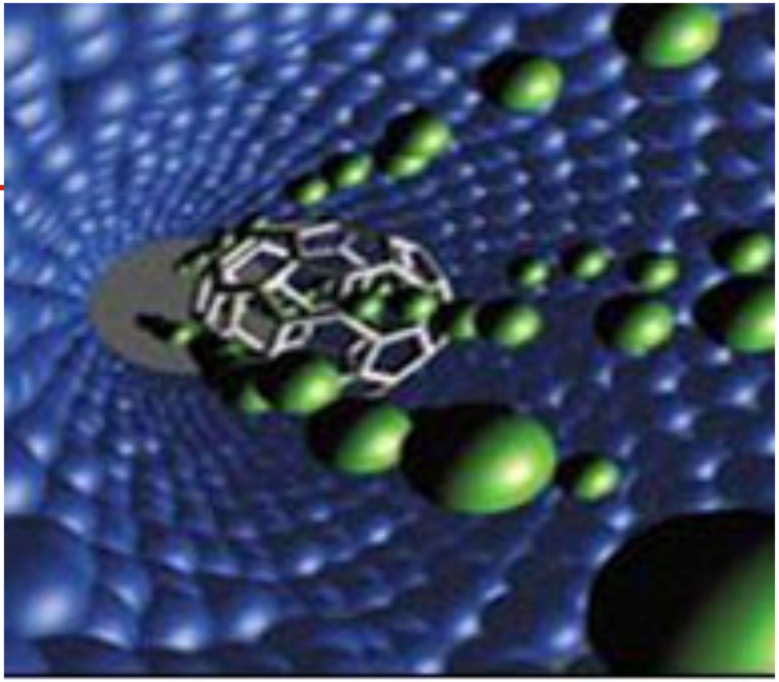
Transportation & traffic application



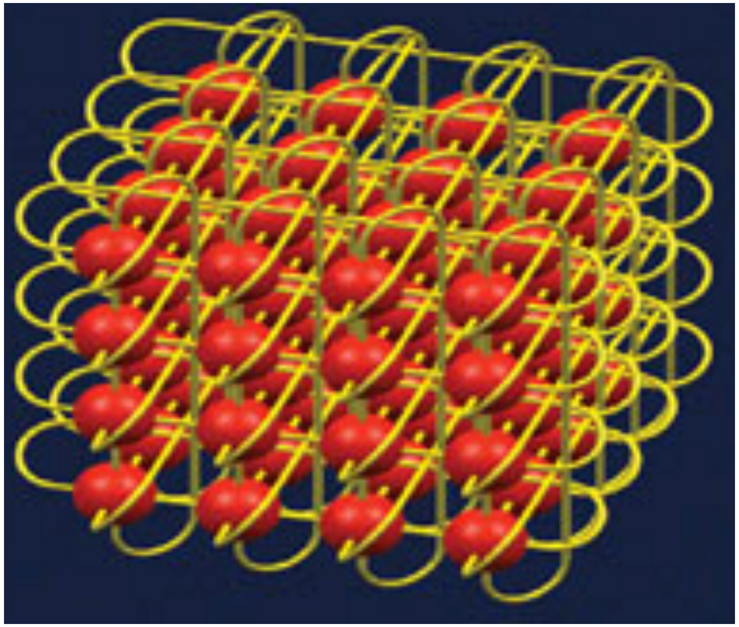
Astrophysics application



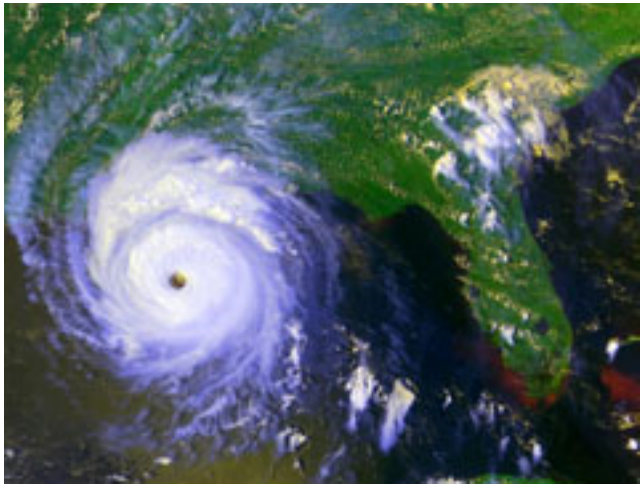
Drug discovery



Advanced Graphics



New materials



Weather modeling

Reasons for Using MPI

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.
- **Functionality** – Rich set of features
- **Availability** - A variety of implementations are available, both vendor and public domain.

Hello World (C)

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

int main( argc, argv )
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;
}
```

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- 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`**.

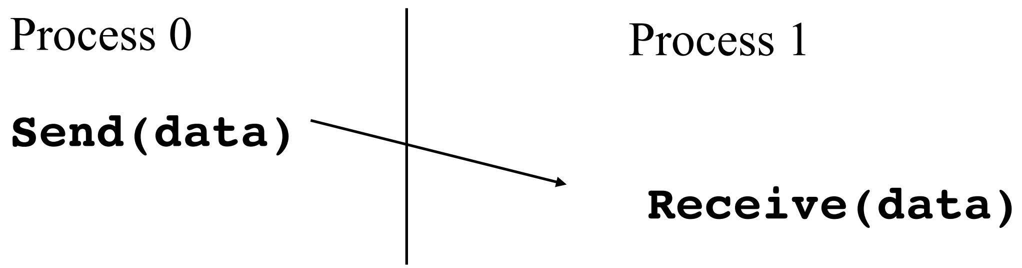
Compiling and Running

- mpicc -o hello hello.c
  - (or mpif77 for Fortran 77, mpif90 for Fortran 90, mpicxx for C++)
  - mpicc etc are scripts provided by the MPI implementation that call the local compiler (e.g., gcc) with the right include paths and link with the right libraries

- mpirun –np 8 hello (or: mpiexec –n 8 hello)
  - Will run 8 processes with the hello executable
  - Further control available to specify location of these processes via a “hosts” file

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?

MPI Datatypes

- The data in a message to be sent or received 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_PRECISION)
  - 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, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

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

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

(Let’s jump back to 3-pt stencil)

Status Object

- The status object is used after completion of a receive to find the actual length, source, and tag of a message
- Status object is MPI-defined type and provides information about:
  - The source process for the message (status.source)
  - The message tag (status.tag)
- The number of elements received is given by:

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

status return status of receive operation (Status)
datatype datatype of each receive buffer element (handle)
count number of received elements (integer)(OUT)

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **MPI_INIT** - initialize the MPI library (must be the first routine called)
  - **MPI_COMM_SIZE** - get the size of a communicator
  - **MPI_COMM_RANK** - get the rank of the calling process in the communicator
  - **MPI_SEND** - send a message to another process
  - **MPI_RECV** - send a message to another process
  - **MPI_FINALIZE** - clean up all MPI state (must be the last MPI function called by a process)
- For performance, however, you need to use other MPI features

Introduction to 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.

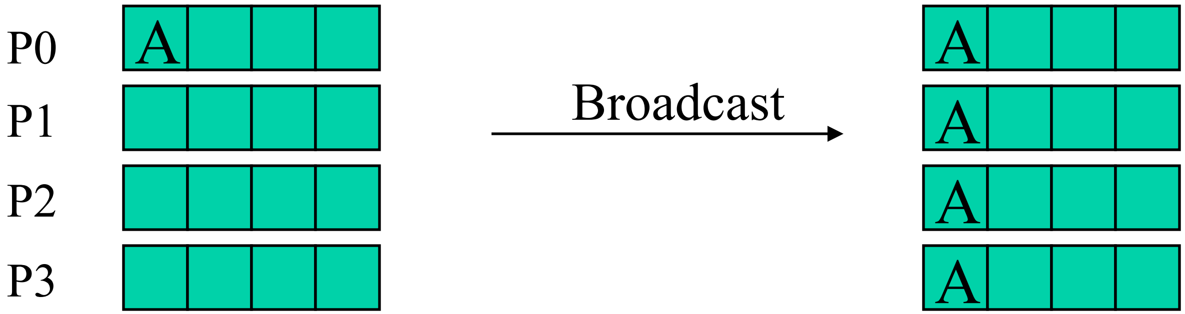
MPI Collective Communication

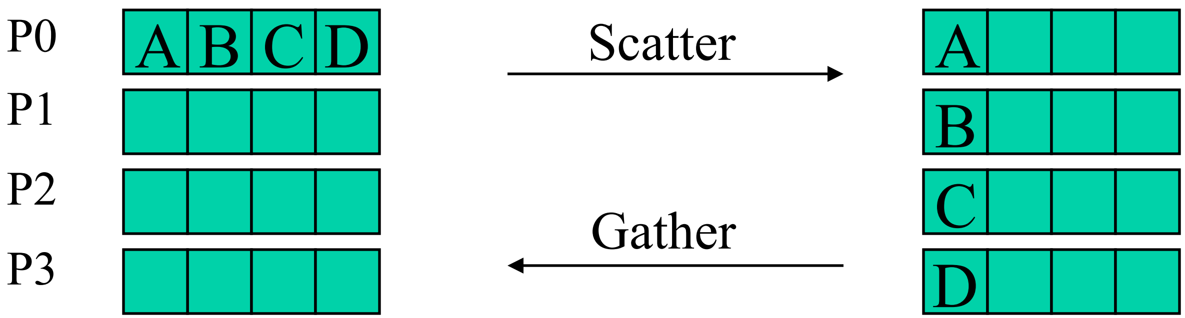
- Communication and computation is coordinated among a group of processes in a communicator.
- Groups and communicators can be constructed "by hand" or using MPI's topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations
  - (they are being added in MPI-3)
- Three classes of operations: synchronization, data movement, collective computation.

Synchronization

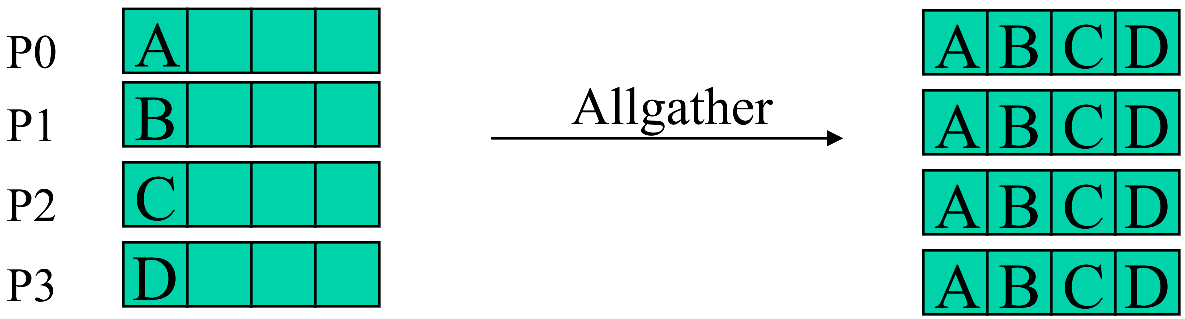
- **`MPI_Barrier( comm )`**
- Blocks until all processes in the group of the communicator **`comm`** call it.
- A process cannot get out of the barrier until all other processes have reached barrier.

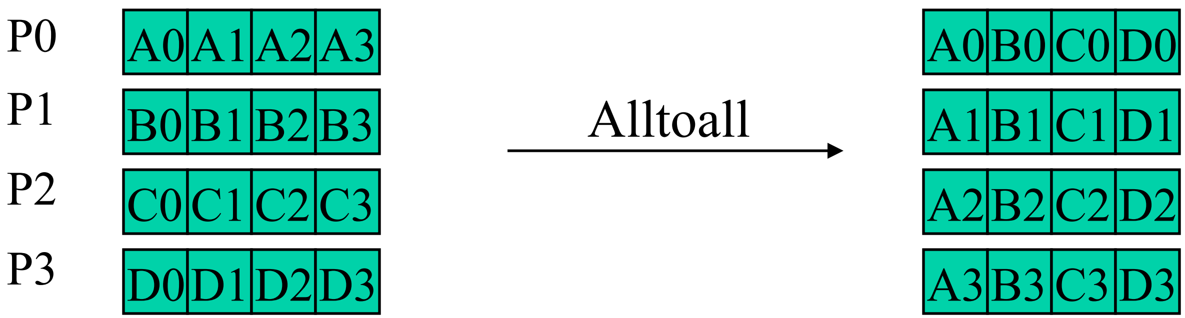
Collective Data Movement



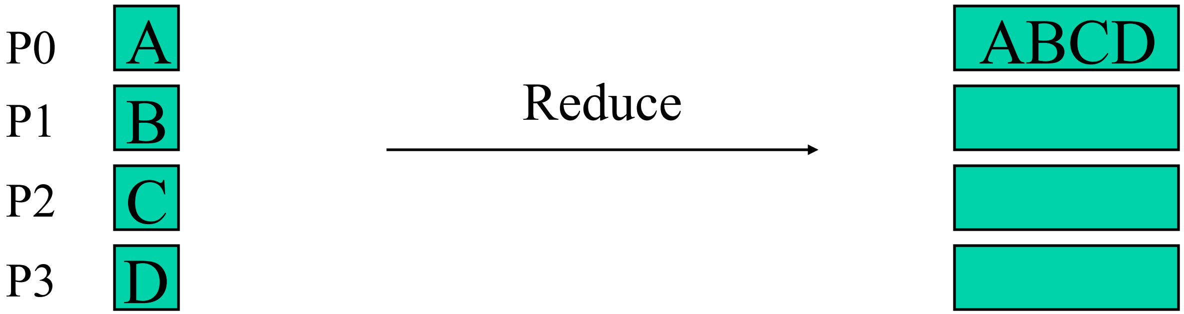


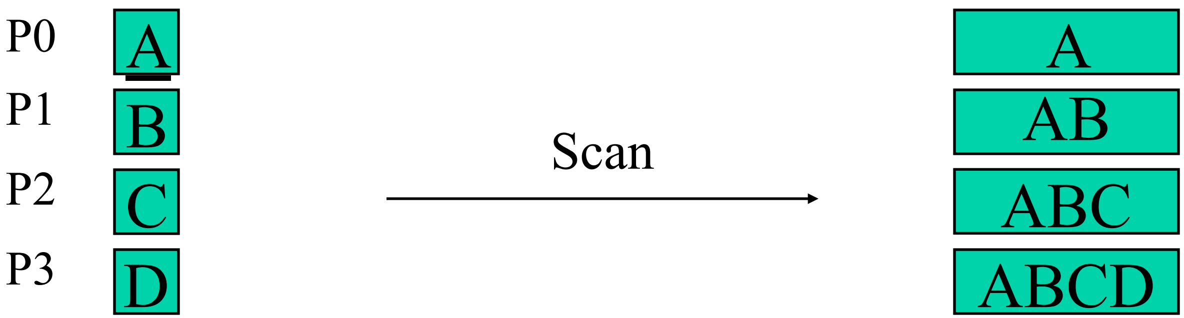
More Collective Data Movement





Collective Computation





MPI Collective Routines

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

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

Defining your own Reduction Operations

- Create your own collective computations with:
```
MPI_Op_create( user_fcn, commutes, &op );
MPI_Op_free( &op );

user_fcn( invec, inoutvec, len, datatype );
```
- The user function should perform:
```
inoutvec[i]  =  invec[i]  op  inoutvec[i];
```
  for i from 0 to len-1.
- The user function can be non-commutative, but must be associative.

Example of Collectives: PI in C (1/2)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, width, 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;
```

input/output data

root process

Example of Collectives: PI in C (2/2)

```
   width = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = width * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = width * 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;
}
```

input location

output data

operation

root process

Blocking v/s Non-blocking modes

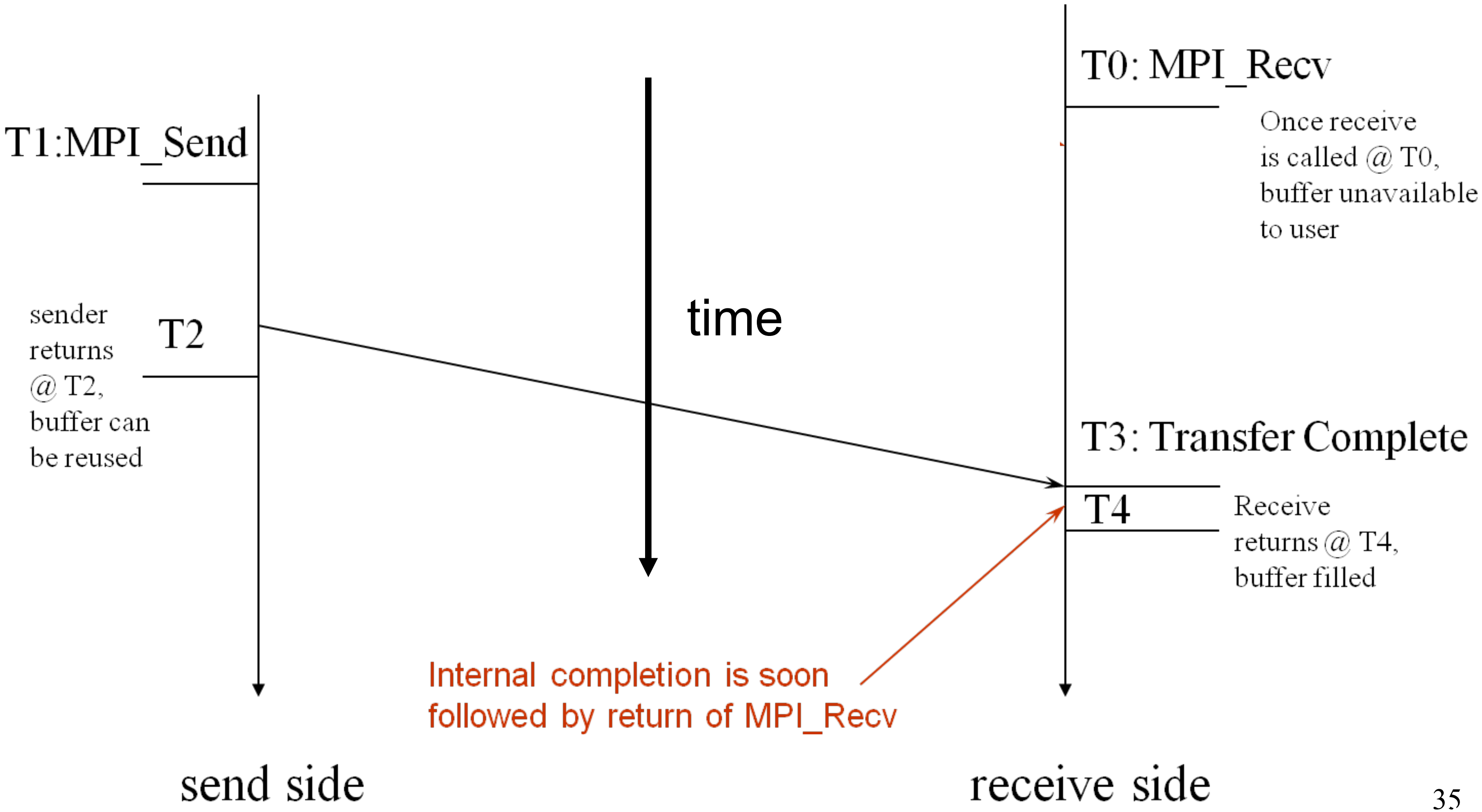
- "Completion" means that memory locations used in the message transfer can be safely accessed for reuse.
  - Safe means that modifications will not affect the data intended for the receive task.
  - For "send" completion implies variable sent can be reused/modified
  - For "receive" variable received can be read.
- Blocking mode:
  - Return from routine implies completion.
- Non-Blocking mode:
  - Routine returns immediately, completion is tested for.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains

Blocking Communication

- In Blocking communication.
  - **MPI_SEND** does not complete until buffer is empty (available for reuse).
  - **MPI_RECV** does not complete until buffer is full (available for use)
- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Completion of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks

```
                  If (my_proc.eq.0) Then
                         Call mpi_send(..)
                         Call mpi_recv(…)
Usually deadlocks →     Else
                         Call mpi_send(…) ← UNLESS you reverse send/recv
                         Call mpi_recv(….)
                        Endif
```

Blocking Send-Receive Diagram



Non-Blocking Communication

- Non-blocking (asynchronous) operations return (immediately) ‘‘request handles” that can be waited on and queried
  - MPI_ISEND(start, count, datatype, dest, tag, comm, request)
  - MPI_IRECV(start, count, datatype, src, tag, comm, request)
  - MPI_WAIT(request, status)

- Non-blocking operations allow overlapping computation and communication.
- One can also test without waiting using **MPI_TEST**
  - **MPI_TEST(request, flag, status)**
- Anywhere you use **MPI_Send** or **MPI_Recv**, you can use the pair of **MPI_Isend/MPI_Wait** or **MPI_Irecv/MPI_Wait**
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead of barriers

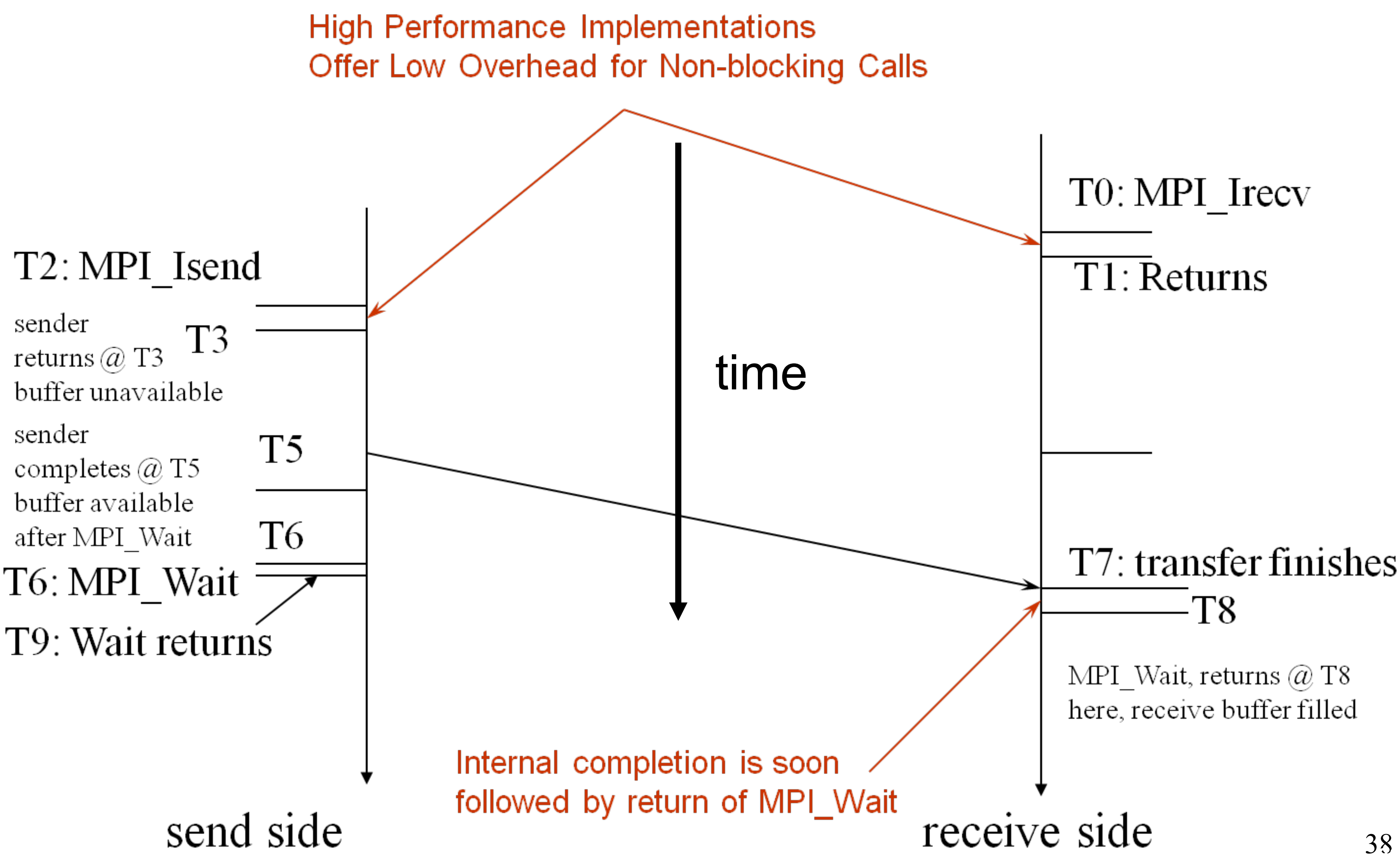
Multiple Completions

- 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.

Non-Blocking Send-Receive Diagram



Message Completion and Buffering

- For a communication to succeed:
  - Sender must specify a valid destination rank
  - Receiver must specify a valid source rank
  - The communicator must be the same
  - Tags must match
  - Receiver's buffer must be large enough
- A send has completed when the user supplied buffer can be reused

```
*buf =3;
MPI_Send (buf, 1, MPI_INT …)
*buf = 4; /*OK, receiver will always receive 3
```

```
*buf =3;
MPI_Isend (buf, 1, MPI_INT …)
*buf = 4; /*Not certain if receiver gets 3 or 4
MPI_Wait(…);
```

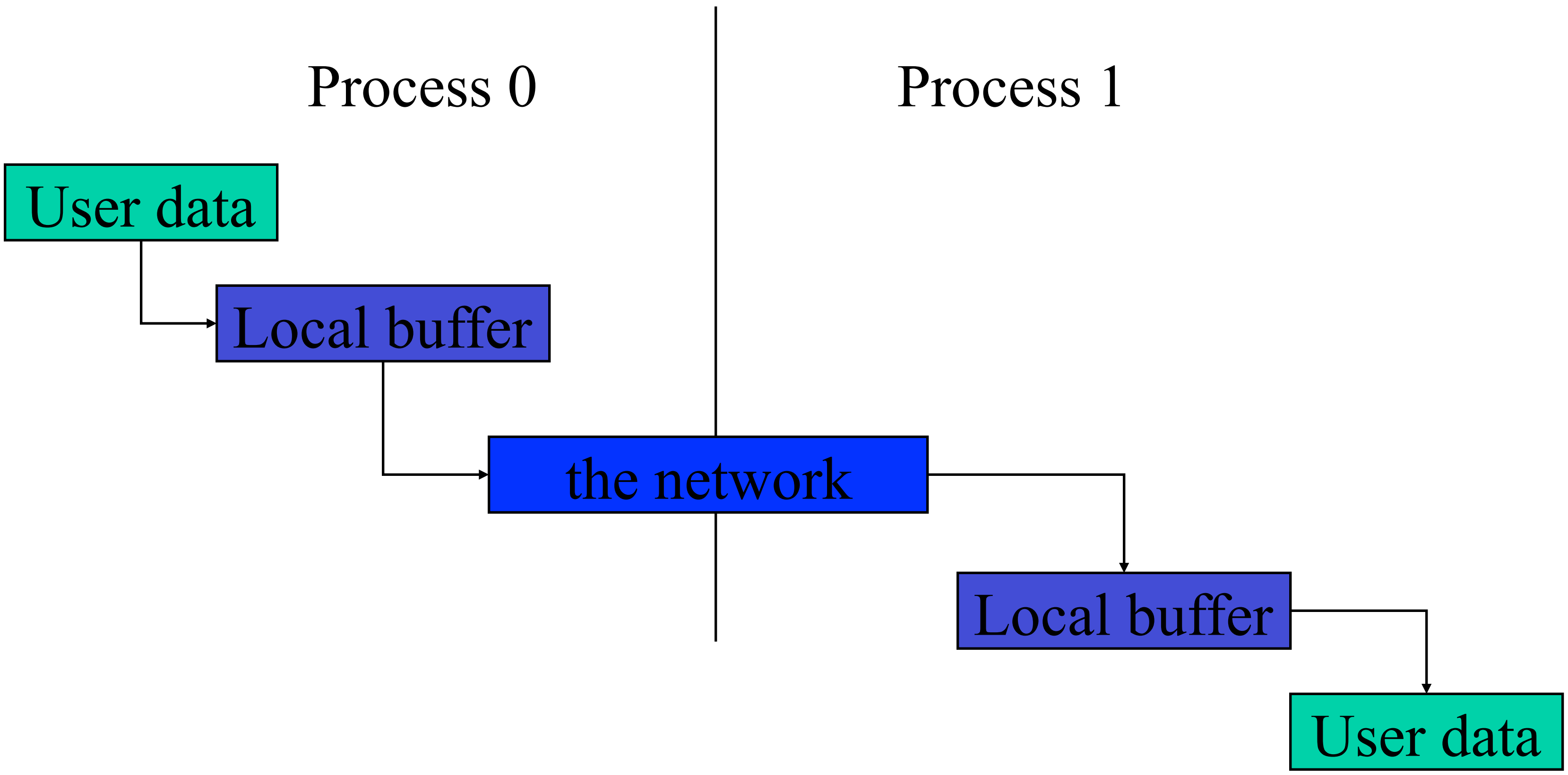
- Just because the send completes does not mean that the receive has completed
  - Message may be buffered by the system
  - Message may still be in transit

More on Message Passing

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

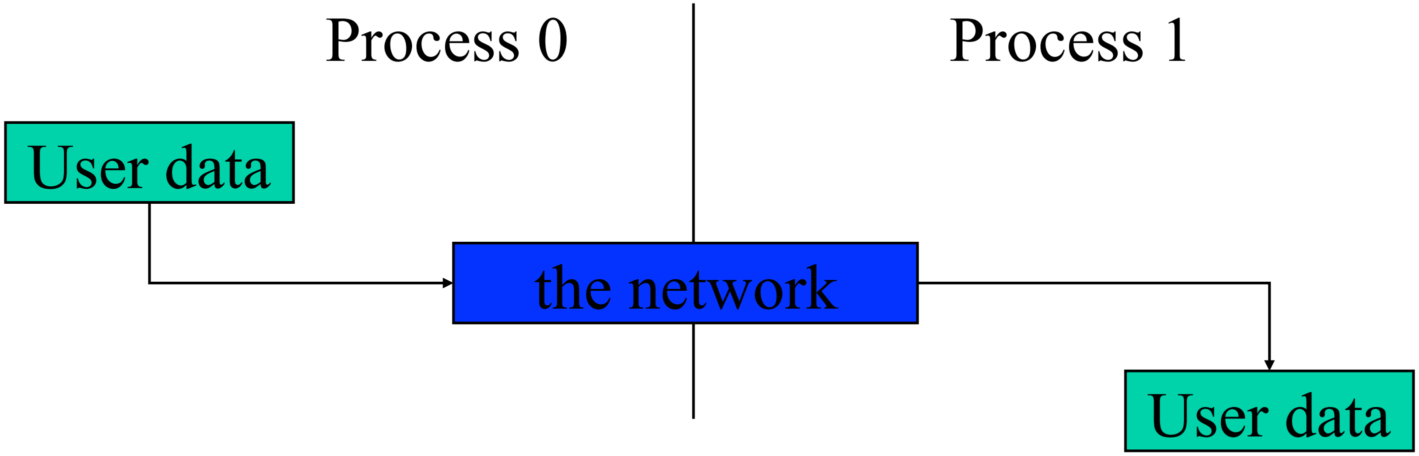
Buffers

- 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.

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`**

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.

Buffered Mode

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

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

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

Other Point-to Point Features

- **`MPI_Sendrecv`**
- **`MPI_Sendrecv_replace`**
- **`MPI_Cancel(request)`**
  - Cancel posted Isend or Irecv
- Persistent requests
  - Useful for repeated communication patterns
  - Some systems can exploit to reduce latency and increase performance
  - MPI_Send_init(…., &request)
  - MPI_Start(request)

MPI_Sendrecv

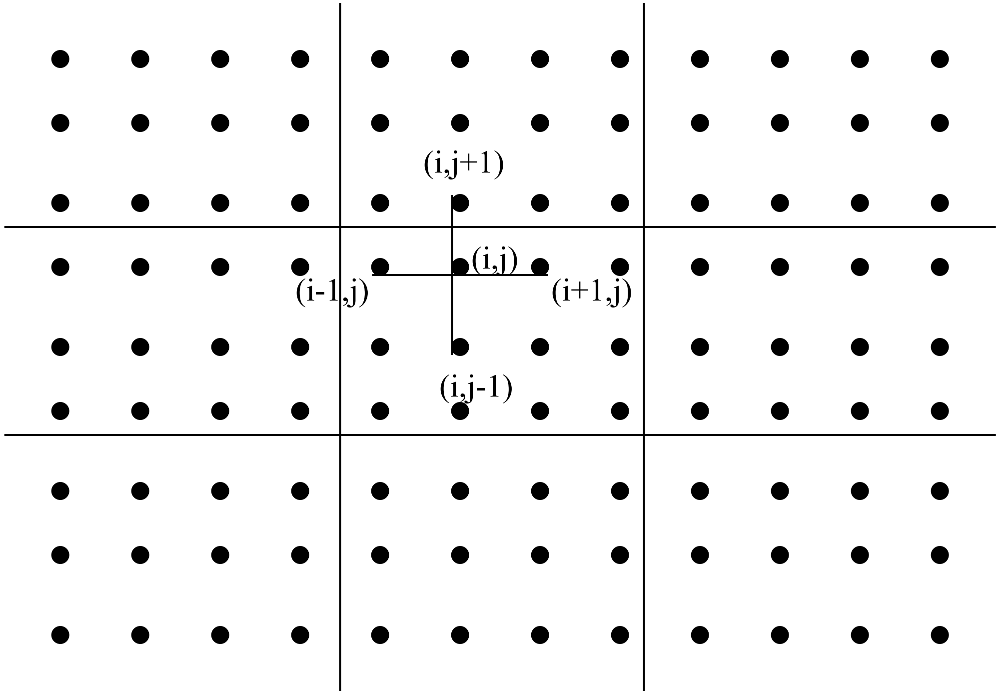
- Allows simultaneous send and receive
- Everything else is general.
  - Send and receive datatypes (even type signatures) may be different
  - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, …)
  - More general than "send left"

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

Understanding Performance: Unexpected Hot Spots

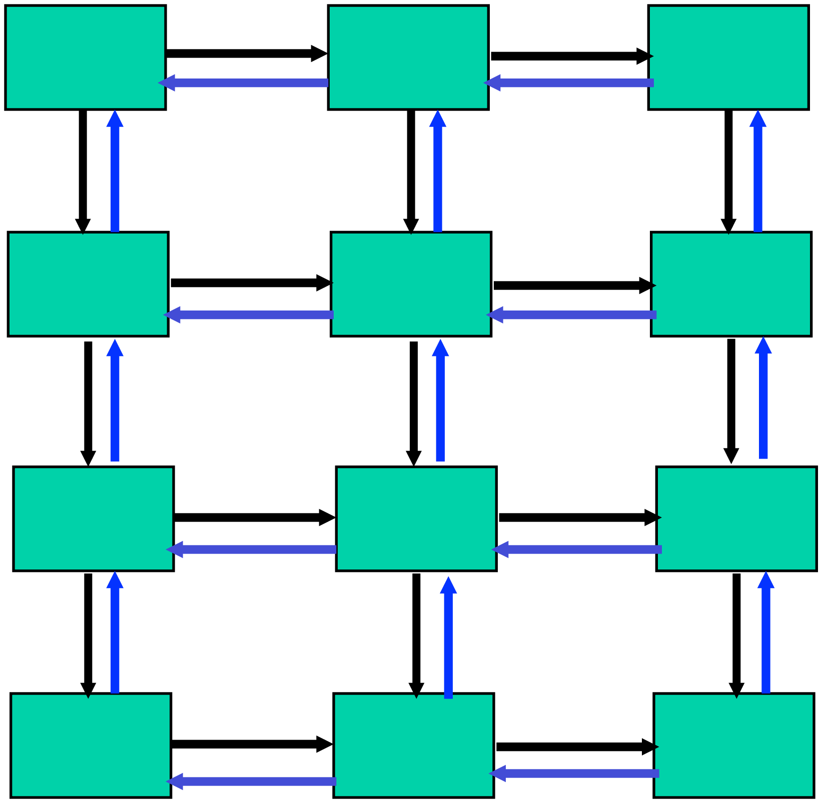
- Basic performance analysis looks at two-party exchanges
- Real applications involve many simultaneous communications
- Performance problems can arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable memory stalls

2D Poisson Problem



Mesh Exchange

- Exchange data on a mesh



Sample Code

- Do i=1,n_neighbors
  Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
  comm, ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Recv(edge,len,MPI_REAL,nbr(i),tag,
  comm,status,ierr)
  Enddo
- What is wrong with this code?

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of

```
if (has down nbr)
    Call MPI_Send( … down … )
if (has up nbr)
    Call MPI_Recv( … up … )
…
```

  sequentializes (all except the bottom process blocks)

Sequentialization

Start	Start	Start	Start	Start	Start	Send	Recv
Send	Send	Send	Send	Send	Send		
					Send	Recv	
				Send	Recv		
			Send	Recv			
		Send	Recv				
	Send	Recv					
Send	Recv						

Fix 1: Use Irecv

- Do i=1,n_neighbors
  Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
  comm,requests(i),ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
  comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
- Does not perform well in practice. Why?

Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)

Fix 2: Use Isend and Irecv

```
Do i=1,n_neighbors
  Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
                 comm,request(i),ierr)
Enddo
Do i=1,n_neighbors
  Call MPI_Isend(edge, len, MPI_REAL, nbr(i), tag,
                 comm, request(n_neighbors+i), ierr)
Enddo
Call MPI_Waitall(2*n_neighbors, request, statuses,
                 ierr)
```

Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Use nonblocking operations and MPI_Waitall to defer synchronization