More on MPI: Buffering, Deadlock, Collectives

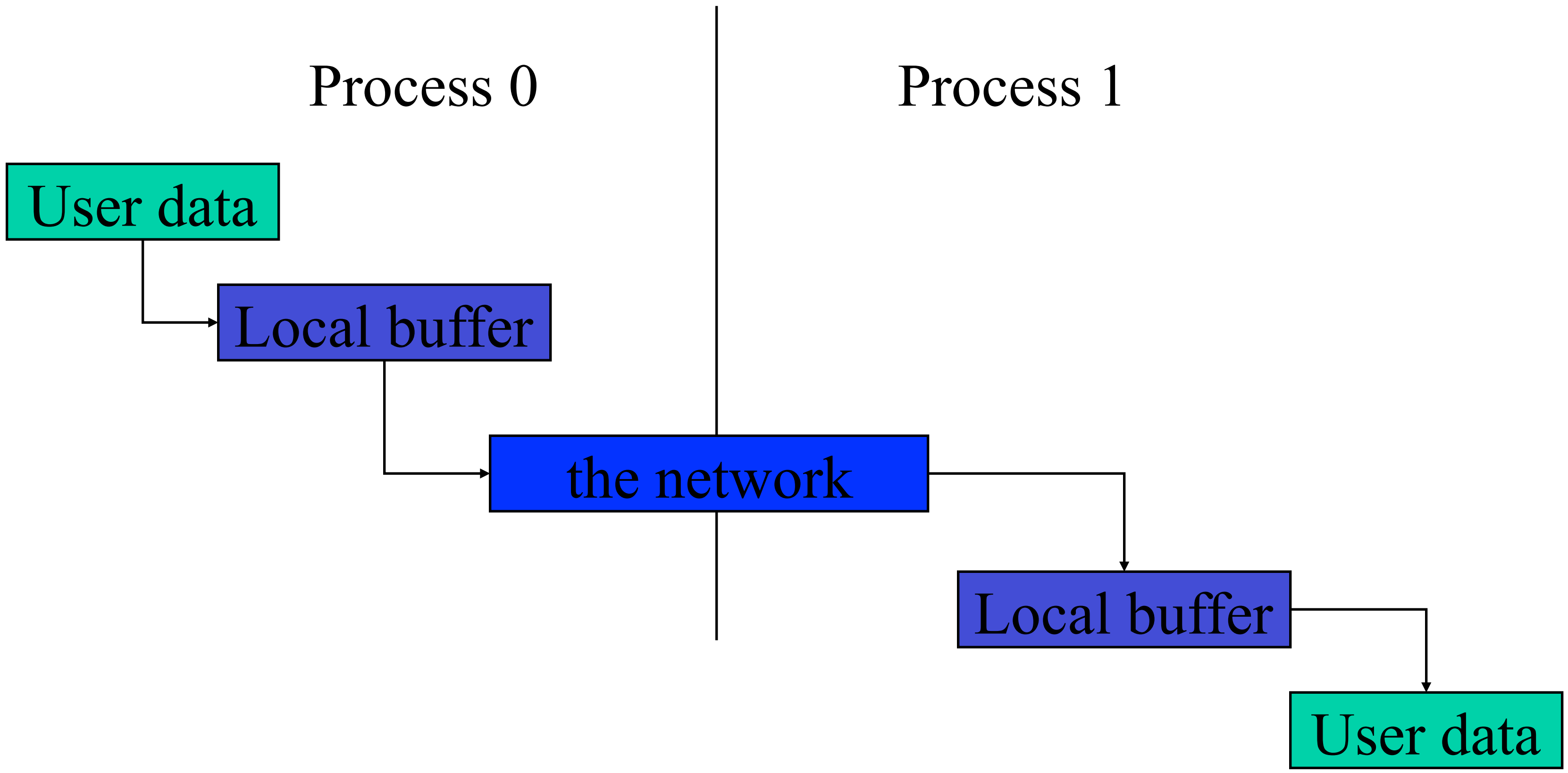
Rajeev Thakur
Argonne National Laboratory

More on Message Passing

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

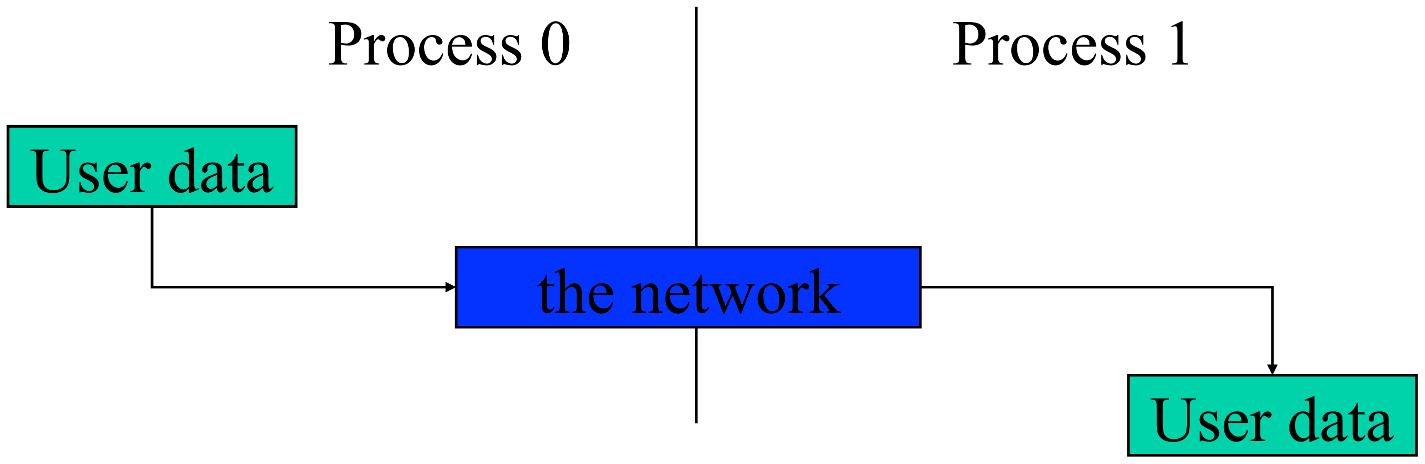
Buffers

- 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

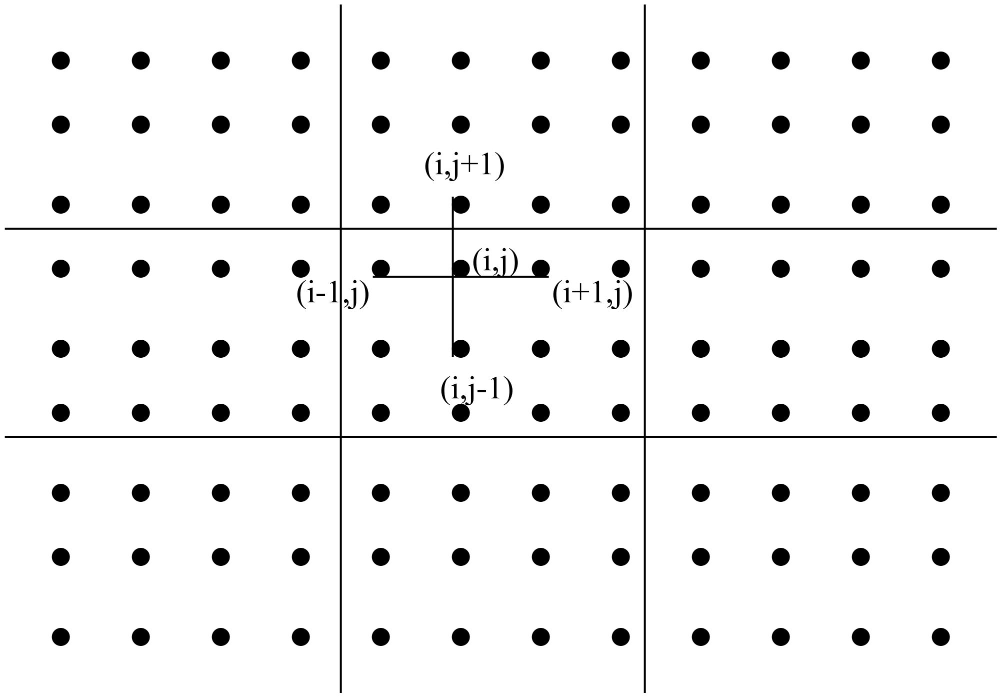
- Allows simultaneous send and receive
- Everything else is general.
  - Send and receive datatypes (even type signatures) may be different
  - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, …)
  - More general than “send left”

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

Understanding Performance: Unexpected Hot Spots

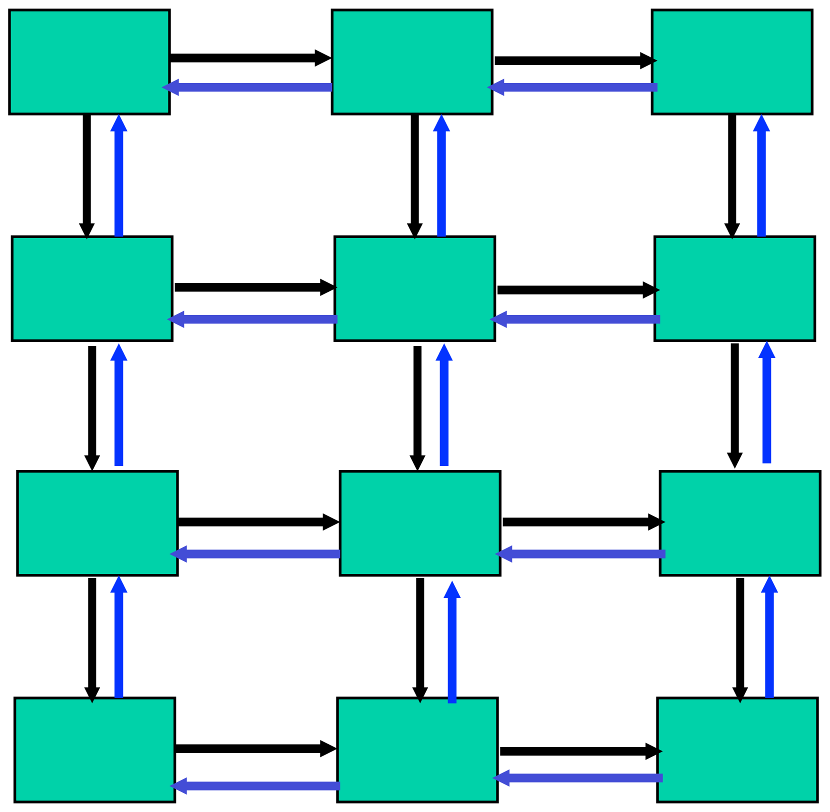
- Basic performance analysis looks at two-party exchanges
- Real applications involve many simultaneous communications
- Performance problems can arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable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

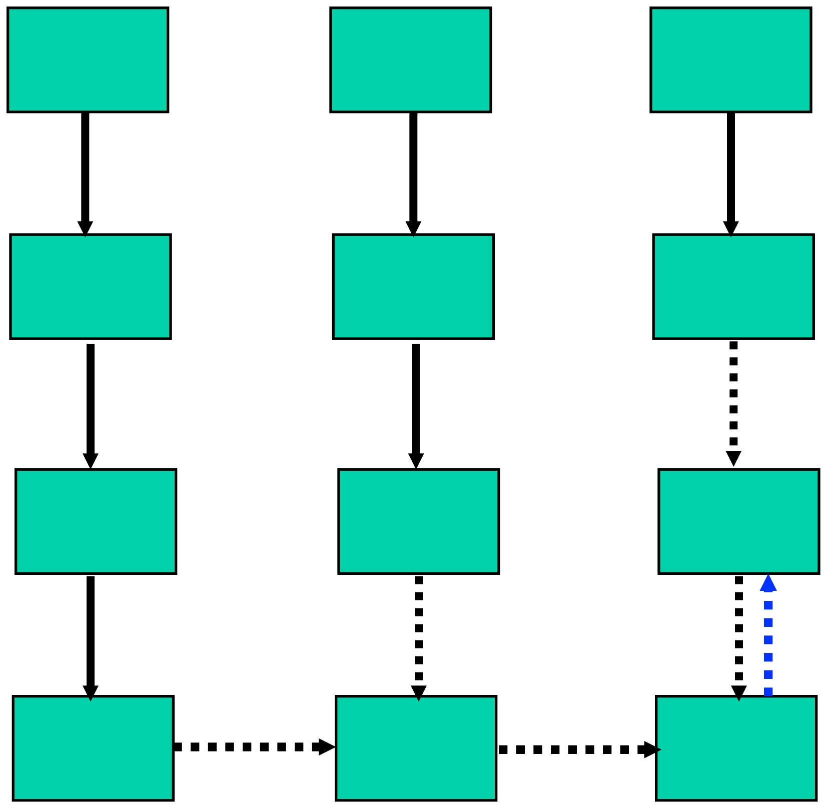
-
  ```
  Do i=1,n_neighbors
     Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
                          comm,requests(i),ierr)
  Enddo
  Do i=1,n_neighbors
     Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
                          comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, 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)

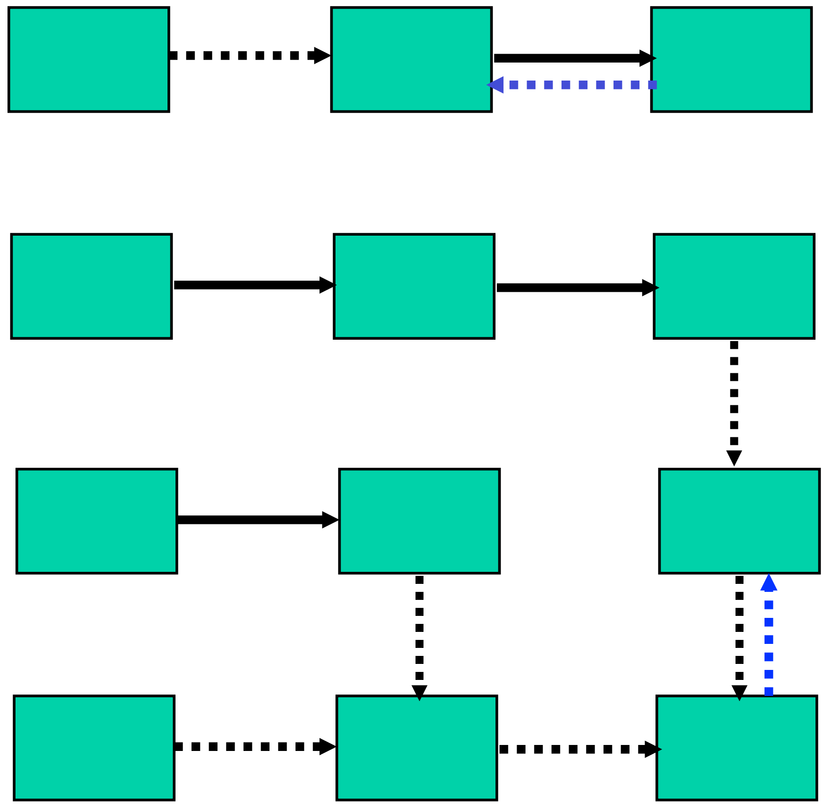
Mesh Exchange - Step 1

- Exchange data on a mesh



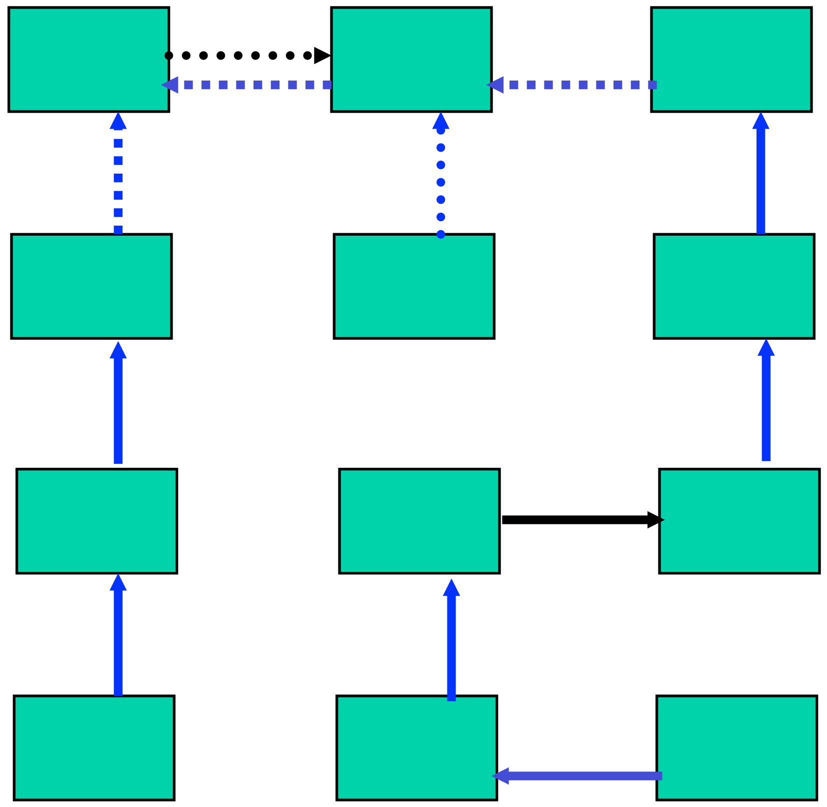
Mesh Exchange - Step 2

- Exchange data on a mesh



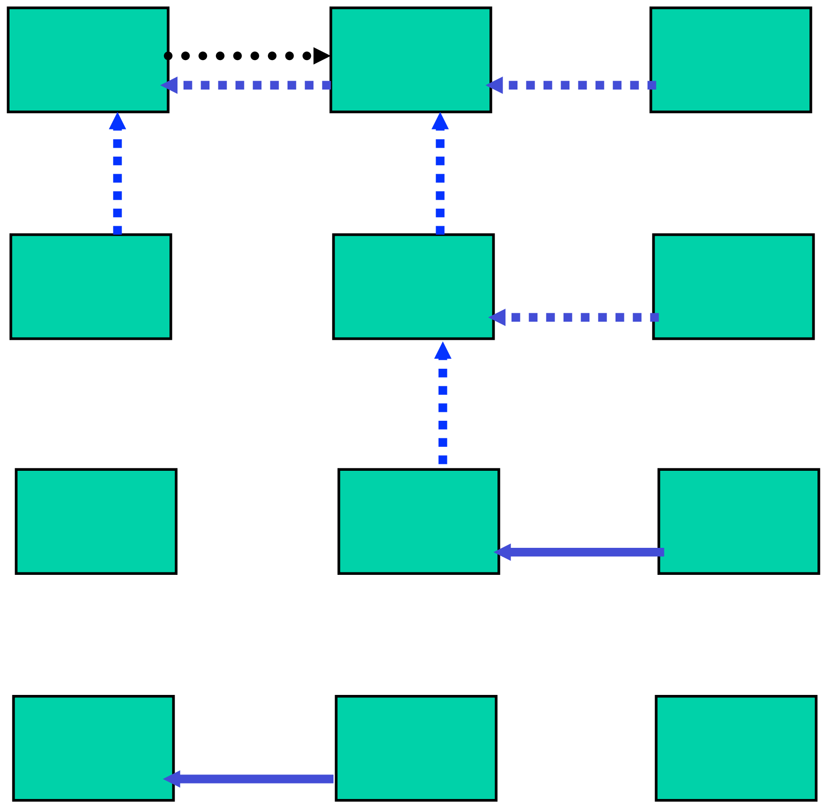
Mesh Exchange - Step 3

- Exchange data on a mesh



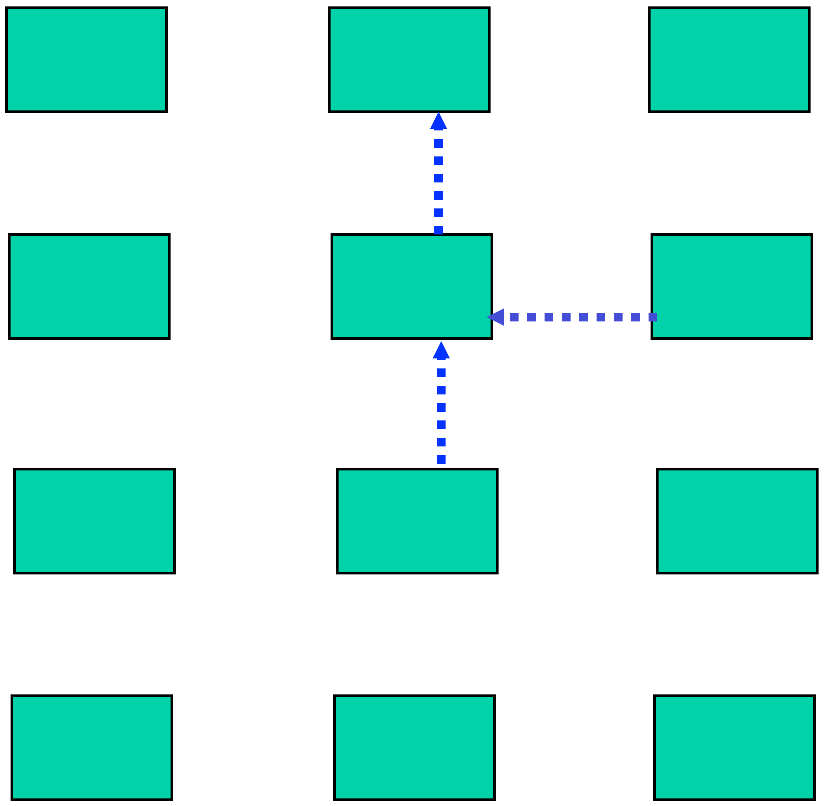
Mesh Exchange - Step 4

- Exchange data on a mesh



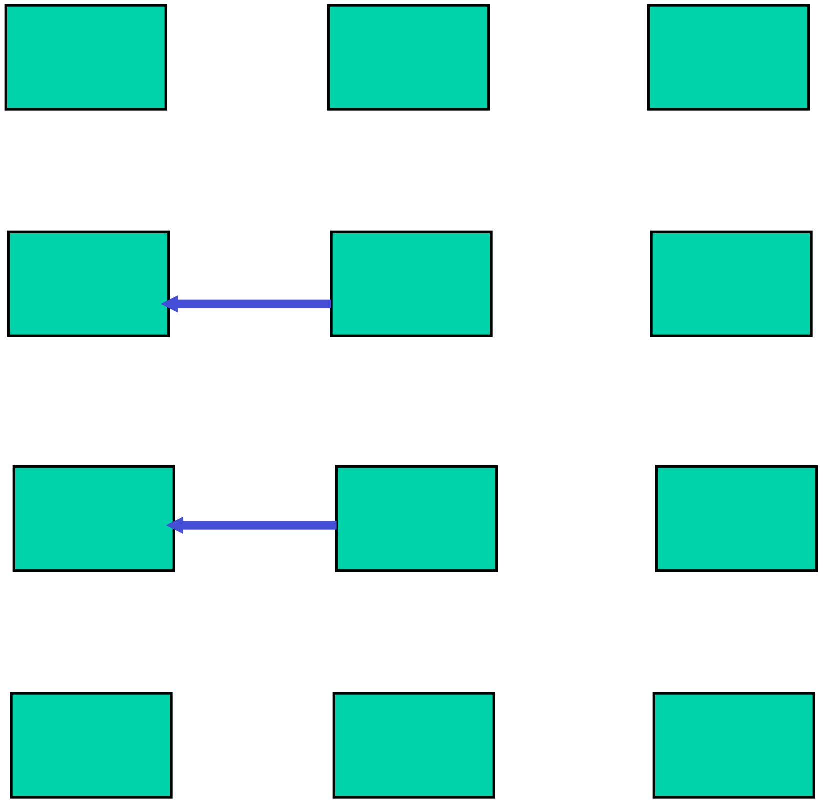
Mesh Exchange - Step 5

- Exchange data on a mesh

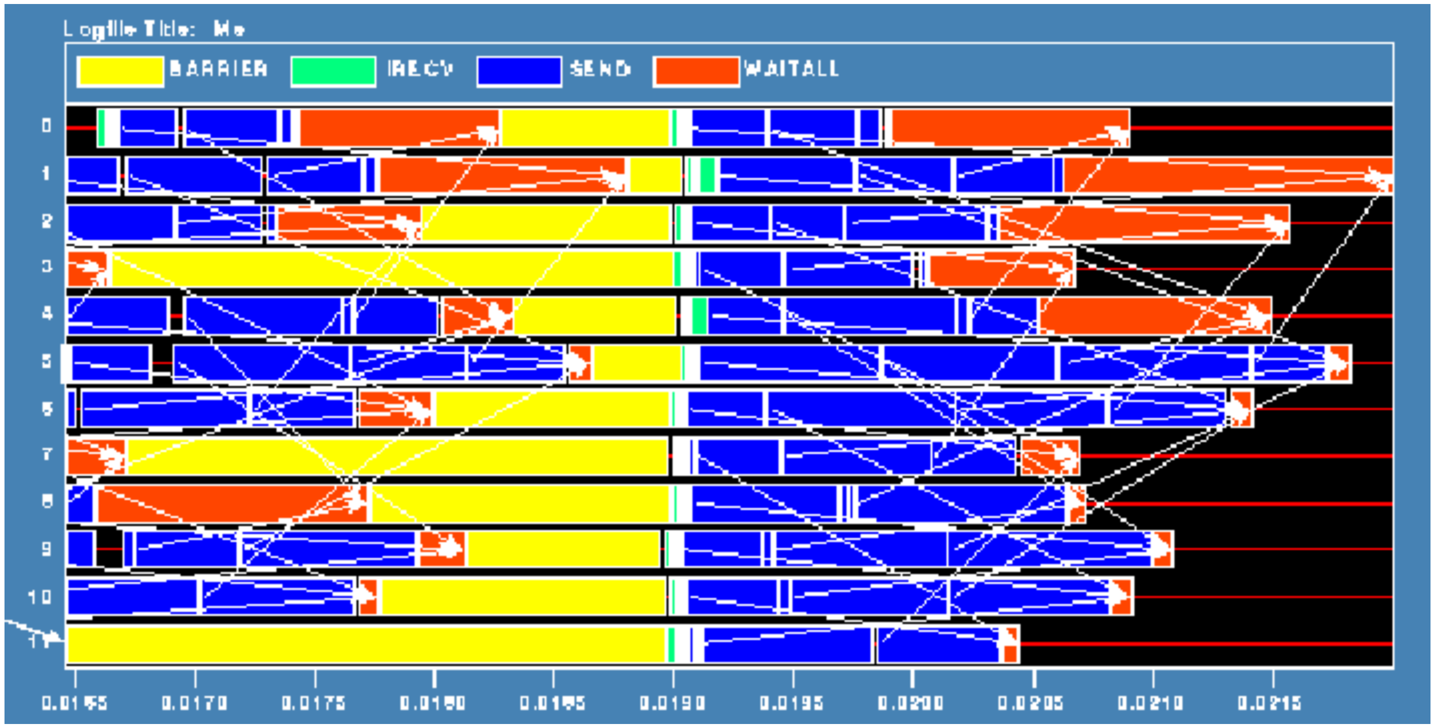


Mesh Exchange - Step 6

- Exchange data on a mesh

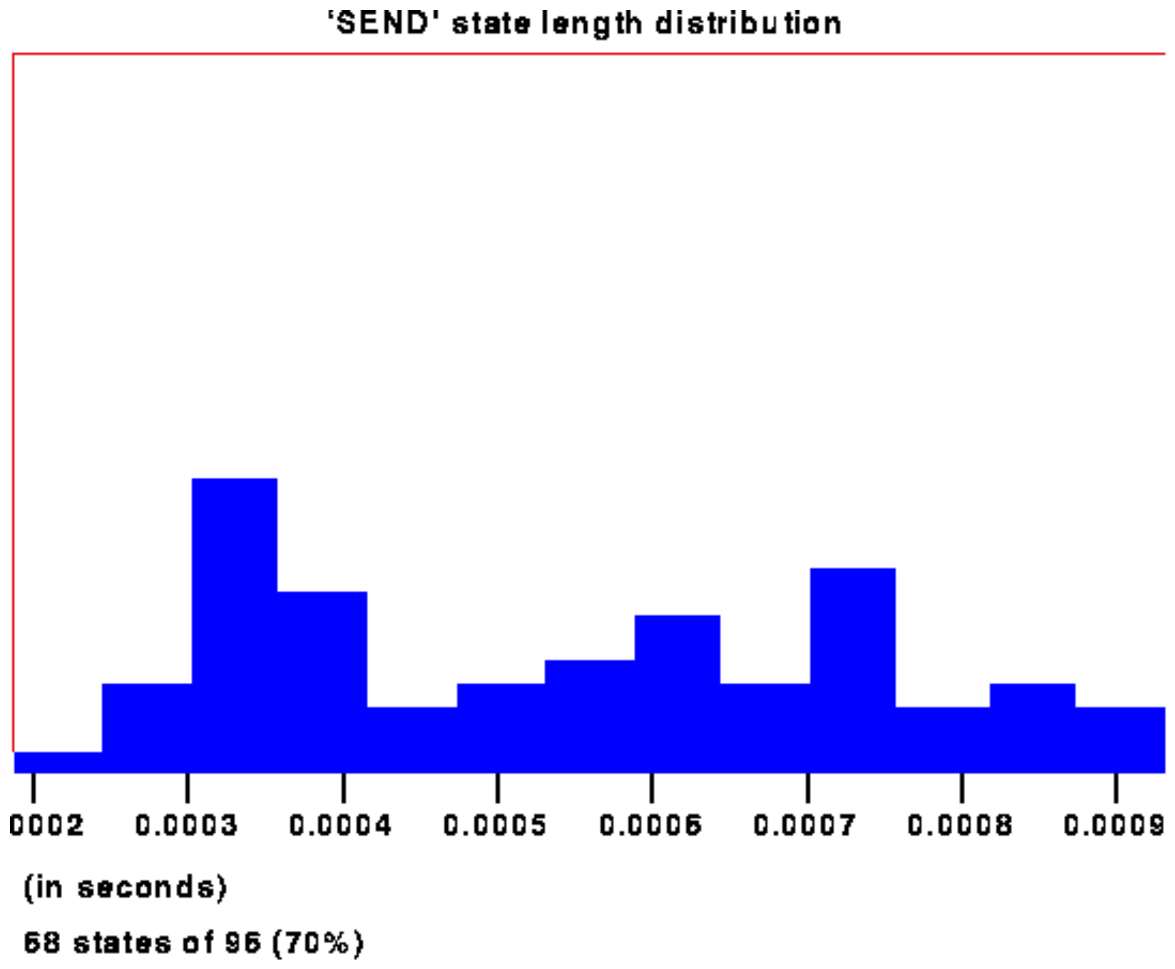


Timeline from IBM SP



• Note that process 1 finishes last, as predicted

Distribution of Sends



Why Six Steps?

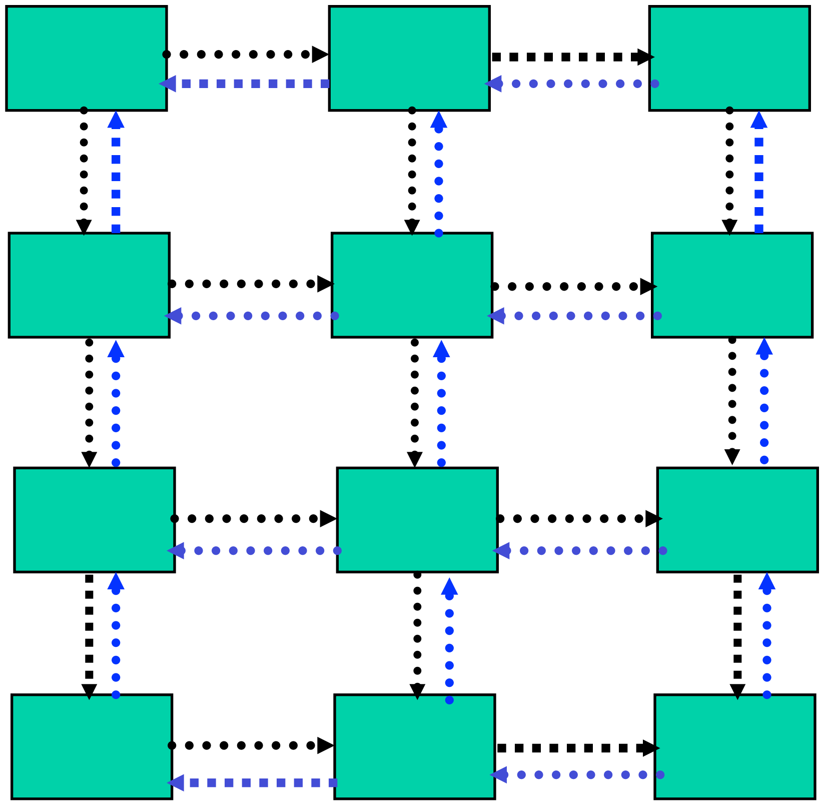
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

Fix 2: Use Isend and Irecv

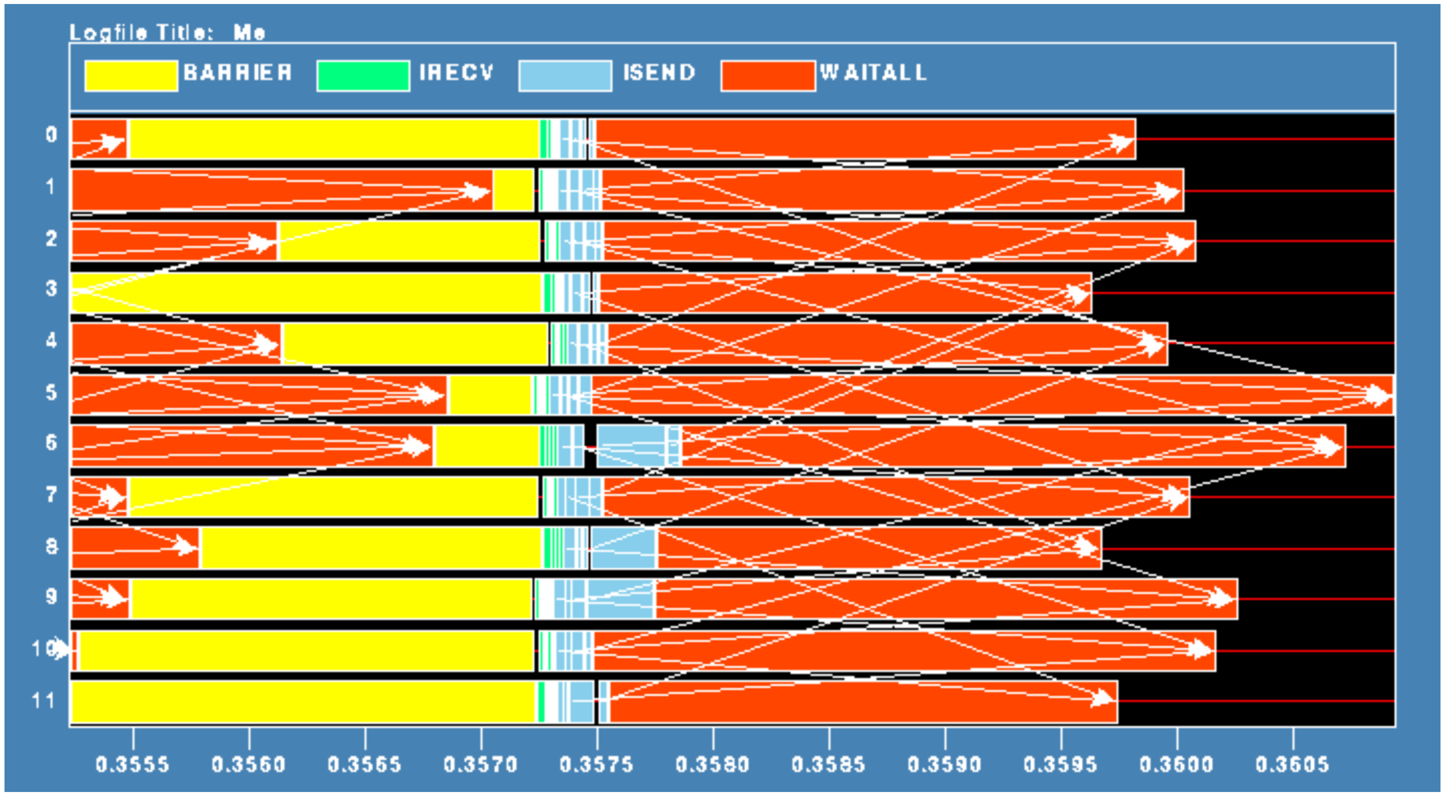
-
  ```
  Do i=1,n_neighbors
     Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
                          comm,request(i),ierr)
  Enddo
  Do i=1,n_neighbors
     Call MPI_Isend(edge, len, MPI_REAL, nbr(i), tag,
                          comm, request(n_neighbors+i), ierr)
  Enddo
  Call MPI_Waitall(2*n_neighbors,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

Mesh Exchange - Steps 1-4

- Four interleaved steps



Timeline from IBM SP



Note processes 5 and 6 are the only interior processors; these perform more communication than the other processors

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

MPI Message Ordering

- Multiple messages from one process to another will be ***matched*** in order, not necessarily *completed* in order

Rank 0	Rank 1	Rank 2
MPI_Isend(dest=1)	MPI_Irecv(any_src, any_tag)	MPI_Isend(dest=1)
MPI_Isend(dest=1)	MPI_Irecv(any_src, any_tag)	MPI_Isend(dest=1)
	MPI_Irecv(any_src, any_tag)	
	MPI_Irecv(any_src, any_tag)	

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.
  - But not always…

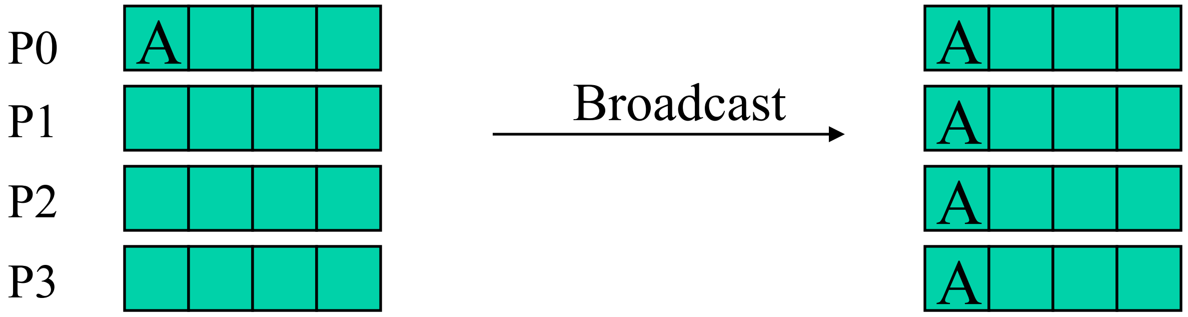
MPI Collective Communication

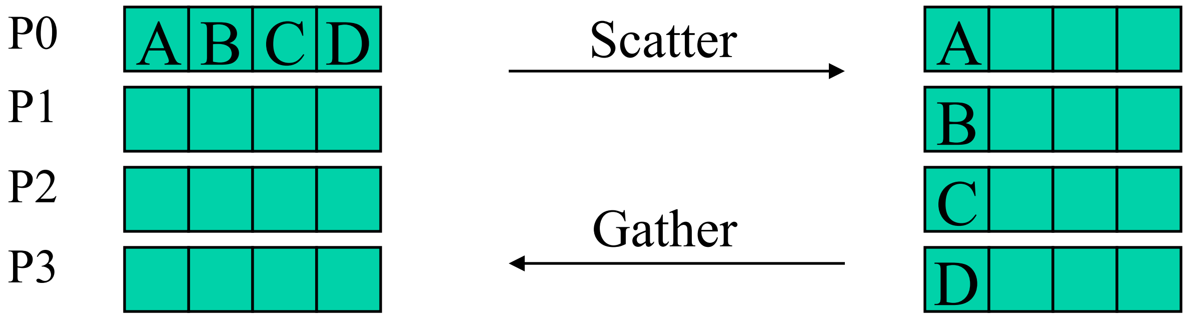
- Communication and computation is coordinated among a group of processes in a communicator.
- Groups and communicators can be constructed "by hand" or using MPI's topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations
 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

Synchronization

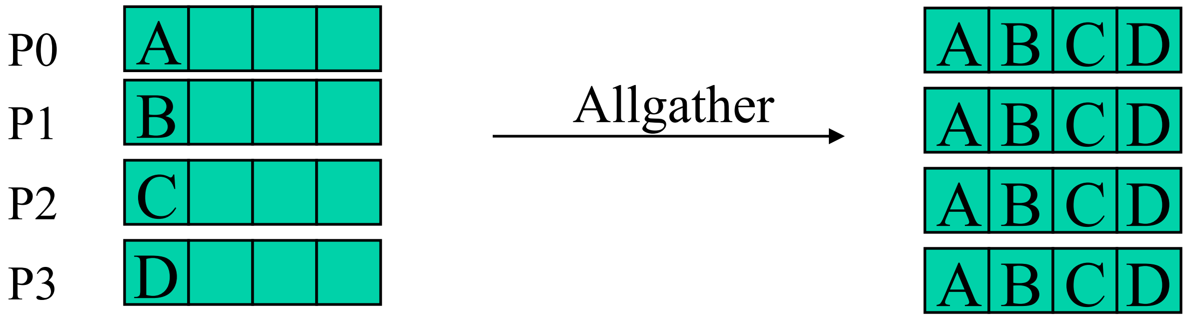
- `MPI_Barrier( comm )`
- Blocks until all processes in the group of the communicator `comm` call it.
- A process cannot get out of the barrier until all other processes have reached barrier.

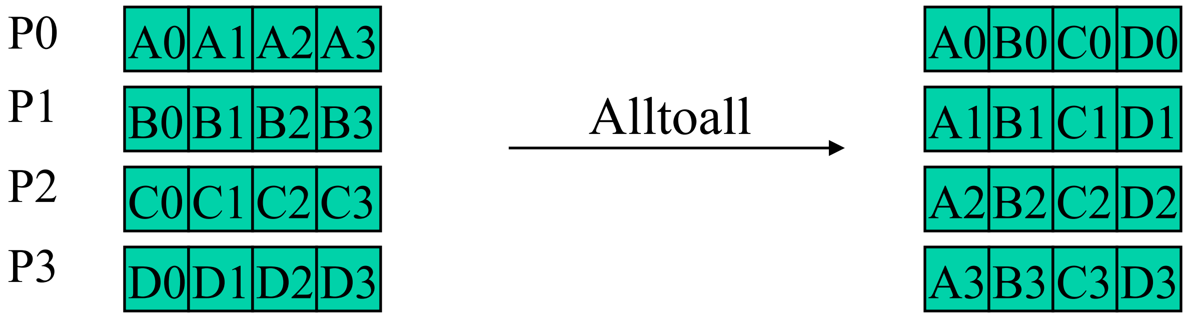
Collective Data Movement



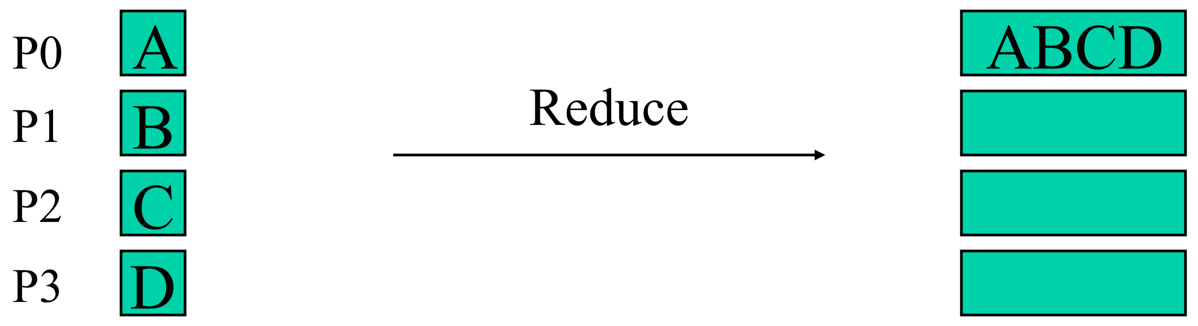


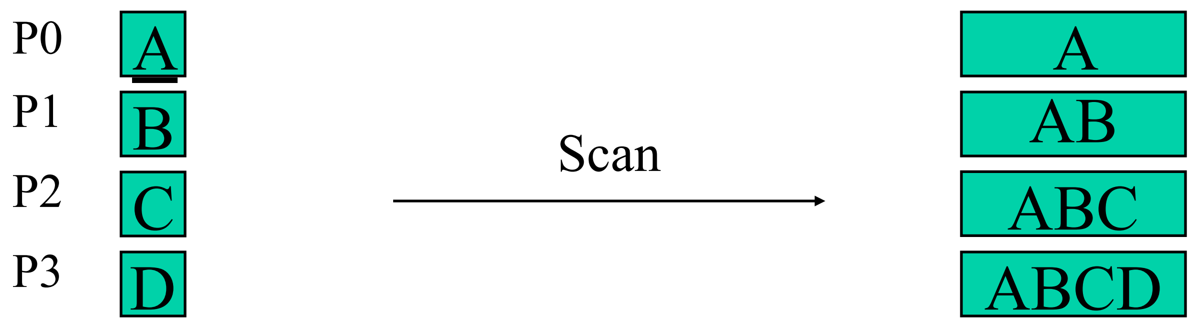
More Collective Data Movement





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

How Deterministic are Reduction Operations?

- In exact arithmetic, you always get the same results
  - but roundoff error, truncation can happen
- MPI does *not* require that the same input give the same output
  - Implementations are encouraged but not required to provide *exactly* the same output given the same input
  - Round-off error may cause slight differences
- Allreduce does guarantee that the same value is received by all processes for each call
- Why didn’t MPI mandate determinism?
  - Not all applications need it
  - Implementations can use “deferred synchronization” ideas to provide better performance

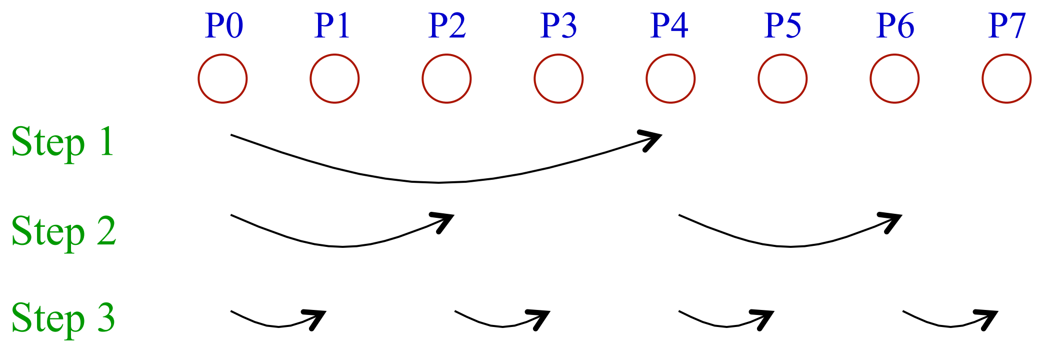
Defining your own Reduction Operations

- 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.

Hands on exercise

- Implement MPI_Bcast using
  - A linear algorithm: root sends data one by one to other ranks
  - A ring algorithm: store and forward
  - A binomial tree algorithm: assume power of two number of processes for simplicity
  - Assume basic datatypes



Binomial Tree broadcast