Lecture 27: Halo Exchange and Contention

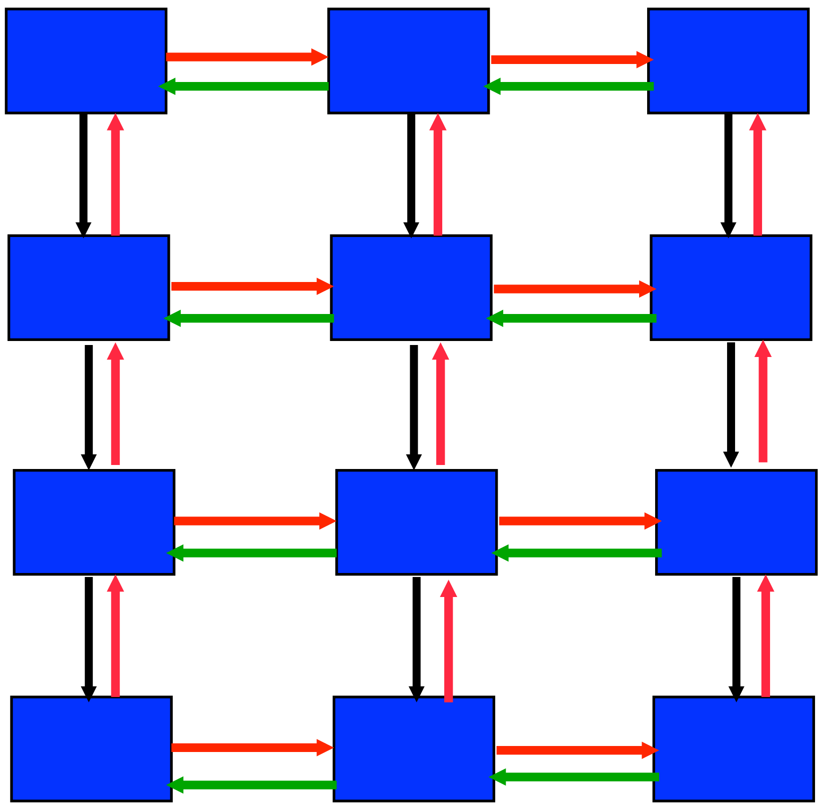
William Gropp
www.cs.illinois.edu/~wgropp

Unexpected Hot Spots

- Even simple operations can give surprising performance behavior.
- Examples arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - ♦ Blocking operations may cause unavoidable stalls

Mesh Exchange

- Exchange data on a mesh



Sample Code

- Do i=1,n_neighbors

```
  Call MPI_Send(edge(1,i), len, MPI_REAL, &
                    nbr(i), tag,comm, ierr)
Enddo
Do i=1,n_neighbors
  Call MPI_Recv(edge(1,i), len, MPI_REAL, &
                 nbr(i), tag, comm, status, ierr)
Enddo
```

Deadlocks!

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

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

  sequentializes (all except the bottom process blocks)

PARALLEL@ILLINOIS

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

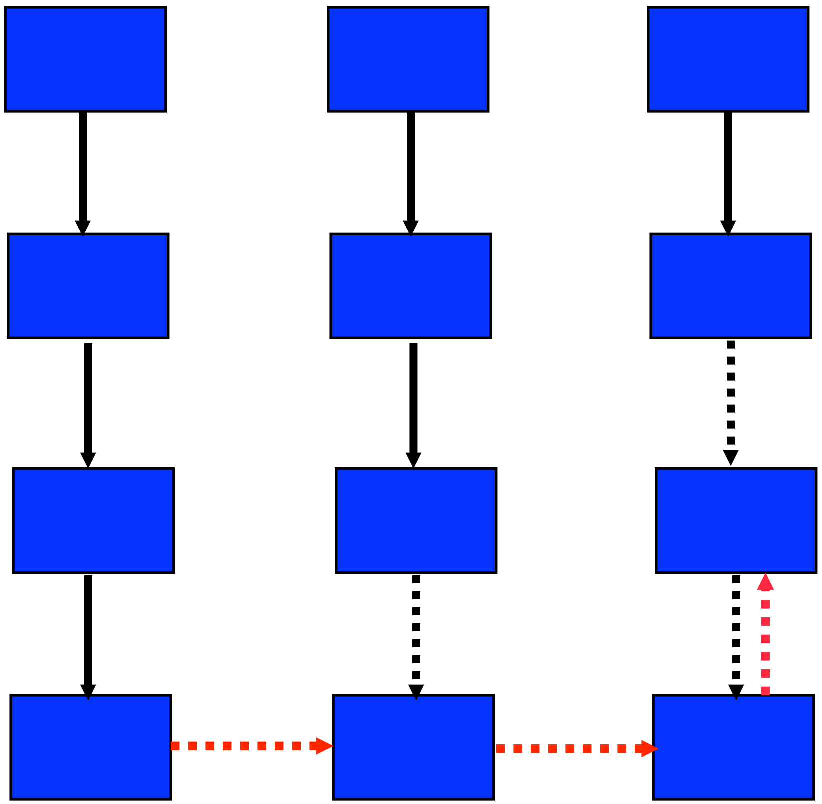
- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(i), ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
- Does not perform well in practice.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

Understanding the Behavior: 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)

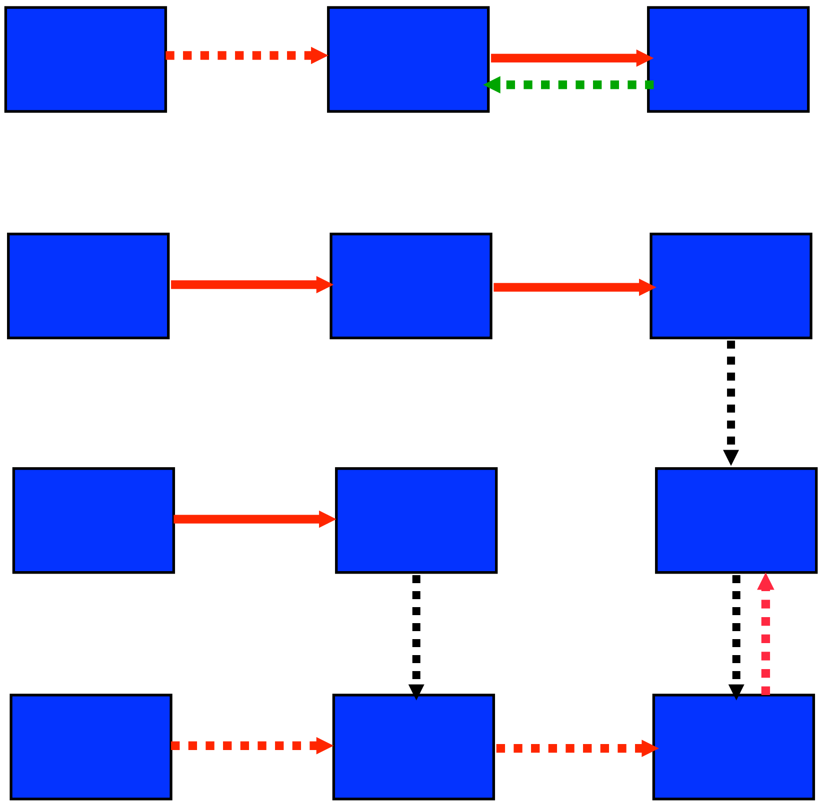
Mesh Exchange - Step 1

- Exchange data on a mesh



Mesh Exchange - Step 2

- Exchange data on a mesh

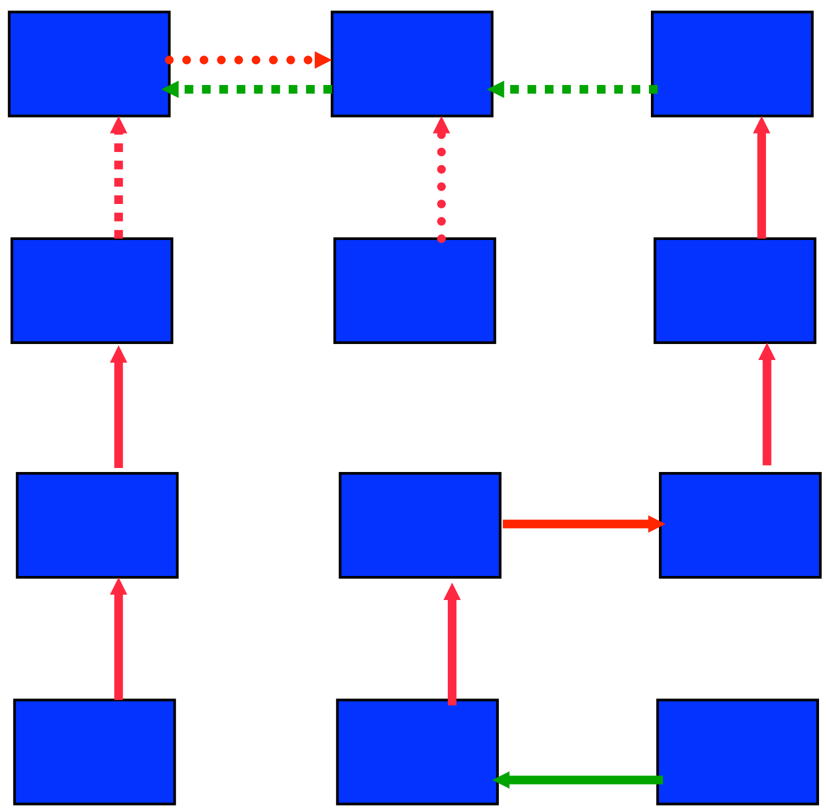


PARALLEL@ILLINOIS

1867

Mesh Exchange - Step 3

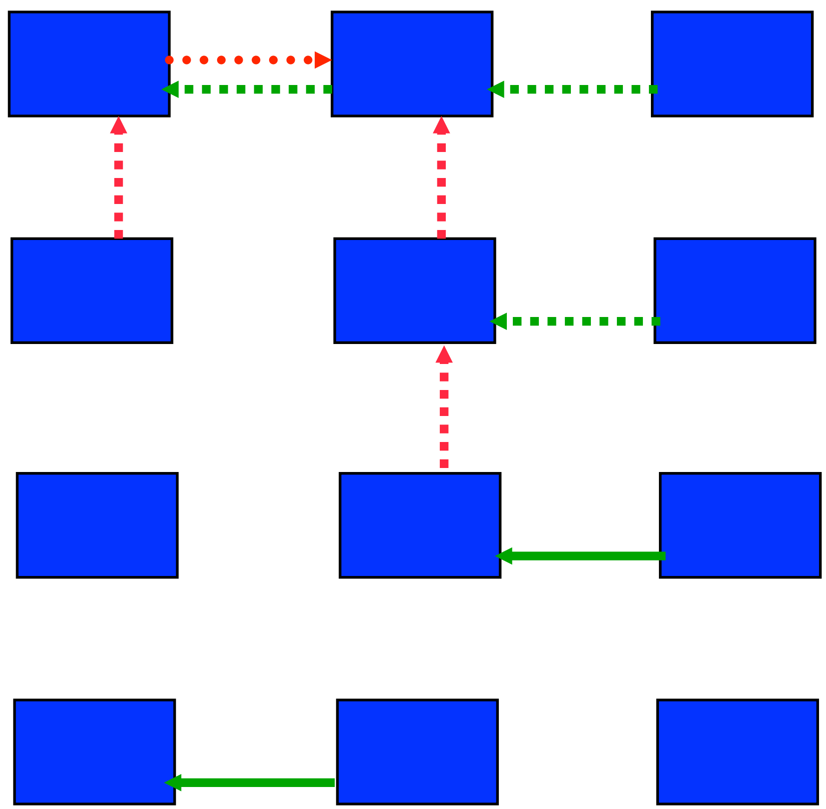
- Exchange data on a mesh



PARALLEL@ILLINOIS

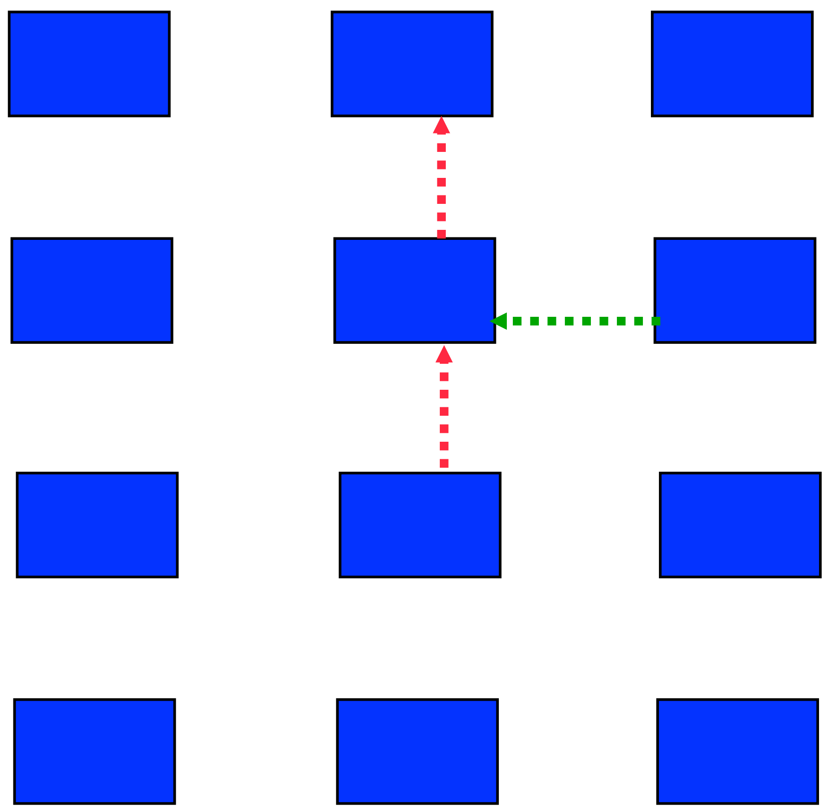
Mesh Exchange - Step 4

- Exchange data on a mesh



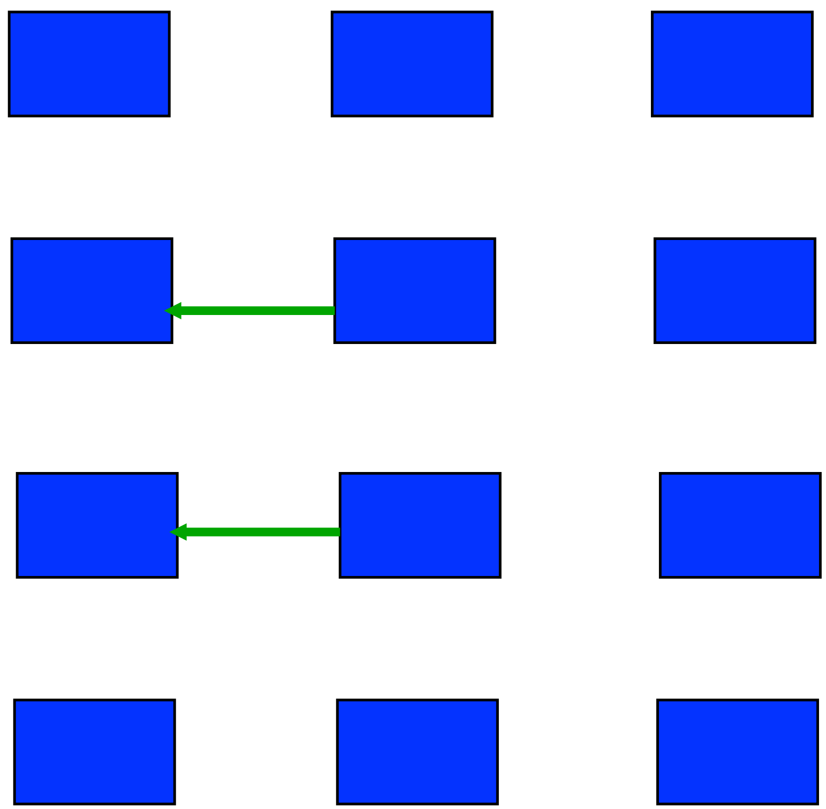
Mesh Exchange - Step 5

- Exchange data on a mesh



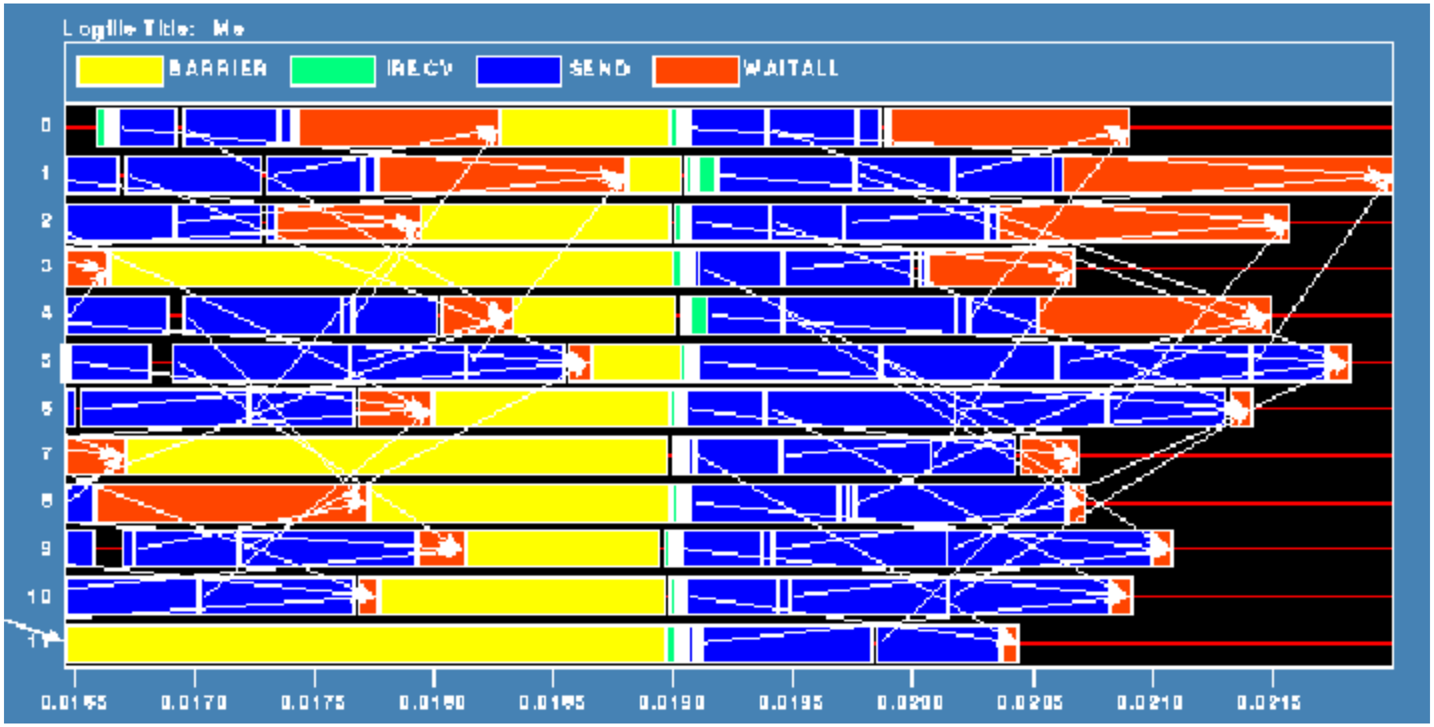
Mesh Exchange - Step 6

- Exchange data on a mesh



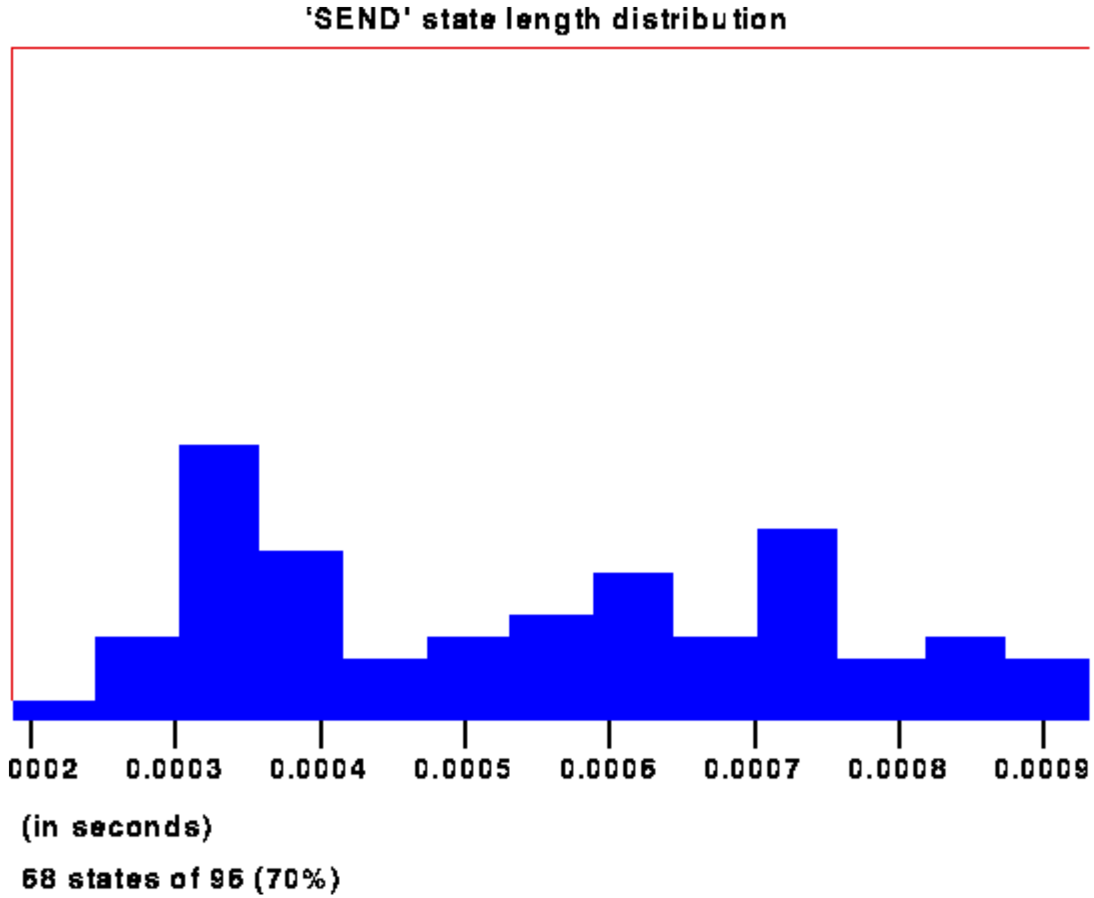
PARALLEL@ILLINOIS

Timeline from IBM SP



- Note that process 1 finishes last, as predicted

Distribution of Sends



PARALLEL@ILLINOIS

Why Six Steps?

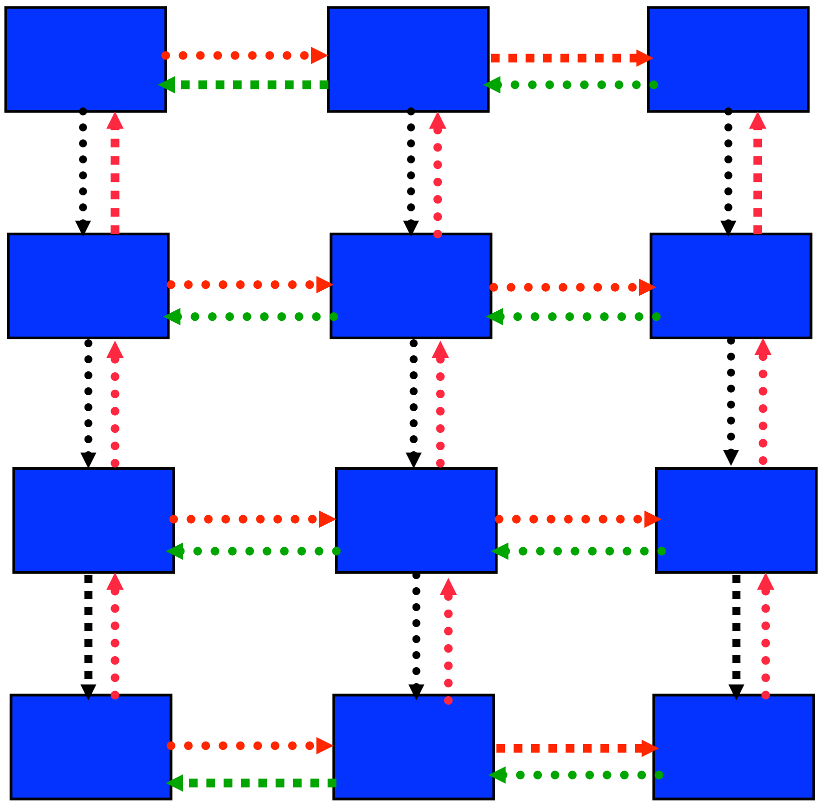
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

Fix 2: Use Isend and Irecv

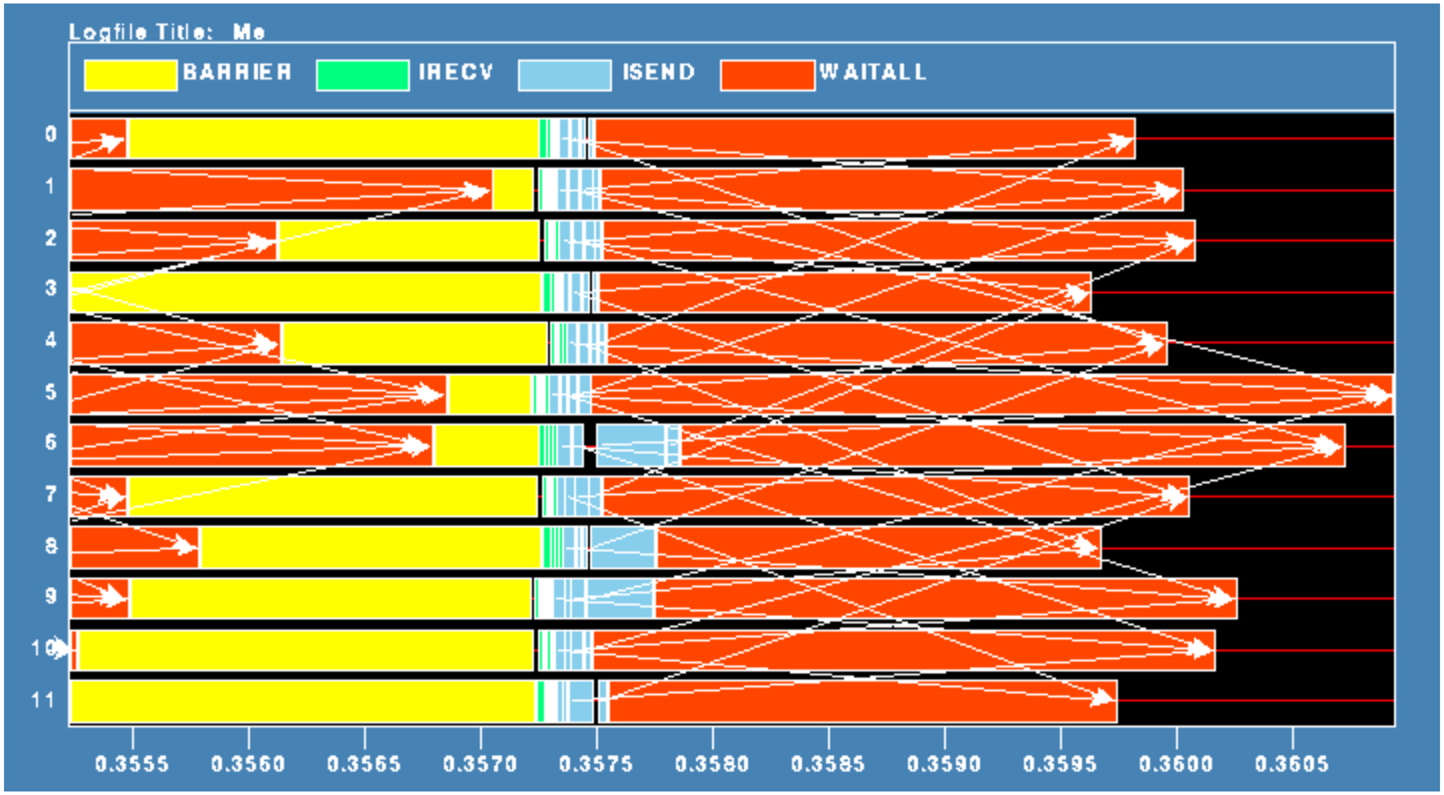
- ```
  Do i=1,n_neighbors
     Call MPI_Irecv(inedge(1,i),len,MPI_REAL,nbr(i),tag,&
                    comm, requests(i),ierr)
  Enddo
  Do i=1,n_neighbors
     Call MPI_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag,&
                    comm, requests(n_neighbors+i), ierr)
  Enddo
  Call MPI_Waitall(2*n_neighbors,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

PARALLEL@ILLINOIS

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
- Effect still common; recently observed on Blue Waters

More Flexibility

- MPI_Waitall forces the process (strictly thread) to wait until *all* requests have completed
- At the cost ot extra code complexity, can use
  - MPI_Waitany – return when any one of the requests complete
  - MPI_Waitsome – return all complete request once at least one is complete
- Now available data can be processed while the rest arrives
  - Works best when there is asynchronous progress by the MPI implementation