



# Lecture 12: MPI Point-to-Point 2

**CMSE 822: Parallel Computing**  
**Prof. Sean M. Couch**



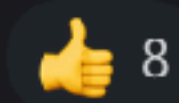
# PCA Questions

## Bucket Brigade



**Stephen White** 3:42 PM

PCA10: For serial operations such as the bucket brigade problem described, what advantage is there in "parallelizing" this code, doesn't this just take an already  $O(n)$  runtime problem and then add  $O(n)$  communication on top? (edited)



8



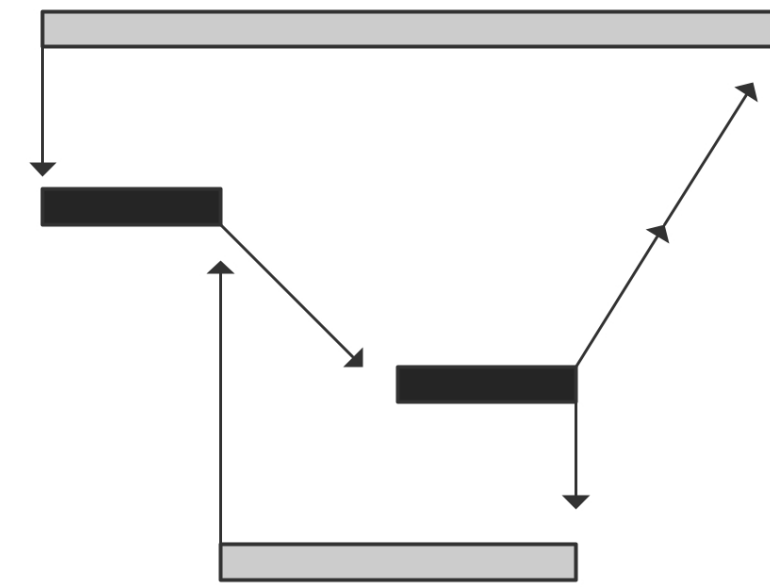
1 reply 5 days ago

- Yes! Essentially this is a serial operation.
- MPI used because overall calculation is parallel.
- Likely, there are better ways to do this with other MPI routines...

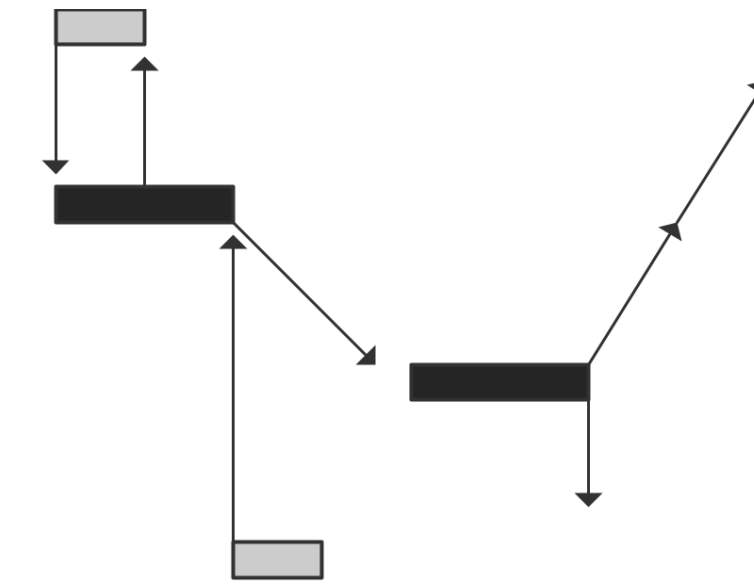


# PCA Questions

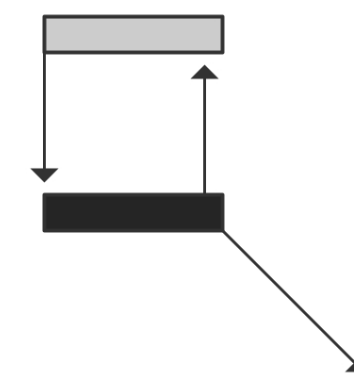
- “synchronous”: processes are coordinated
- “blocking”: no use of system buffer



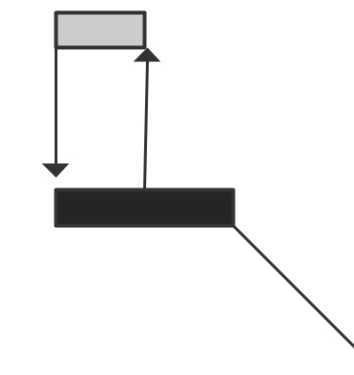
blocking synchronous send,  
blocking receive



nonblocking synchronous send,  
nonblocking receive

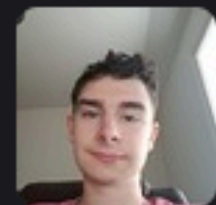


blocking asynchronous send



nonblocking asynchronous  
send

Figure 4.14: Blocking and synchronicity



**Luke Wiseman** 3:51 PM

PCA11: Could you go over the 4 cases shown in figure 4.14 where it is talking about the differences between blocking and nonblocking combined with synchronous and asynchronous communication? (edited)



4





# Blocking vs. Non-blocking

- **Blocking:**

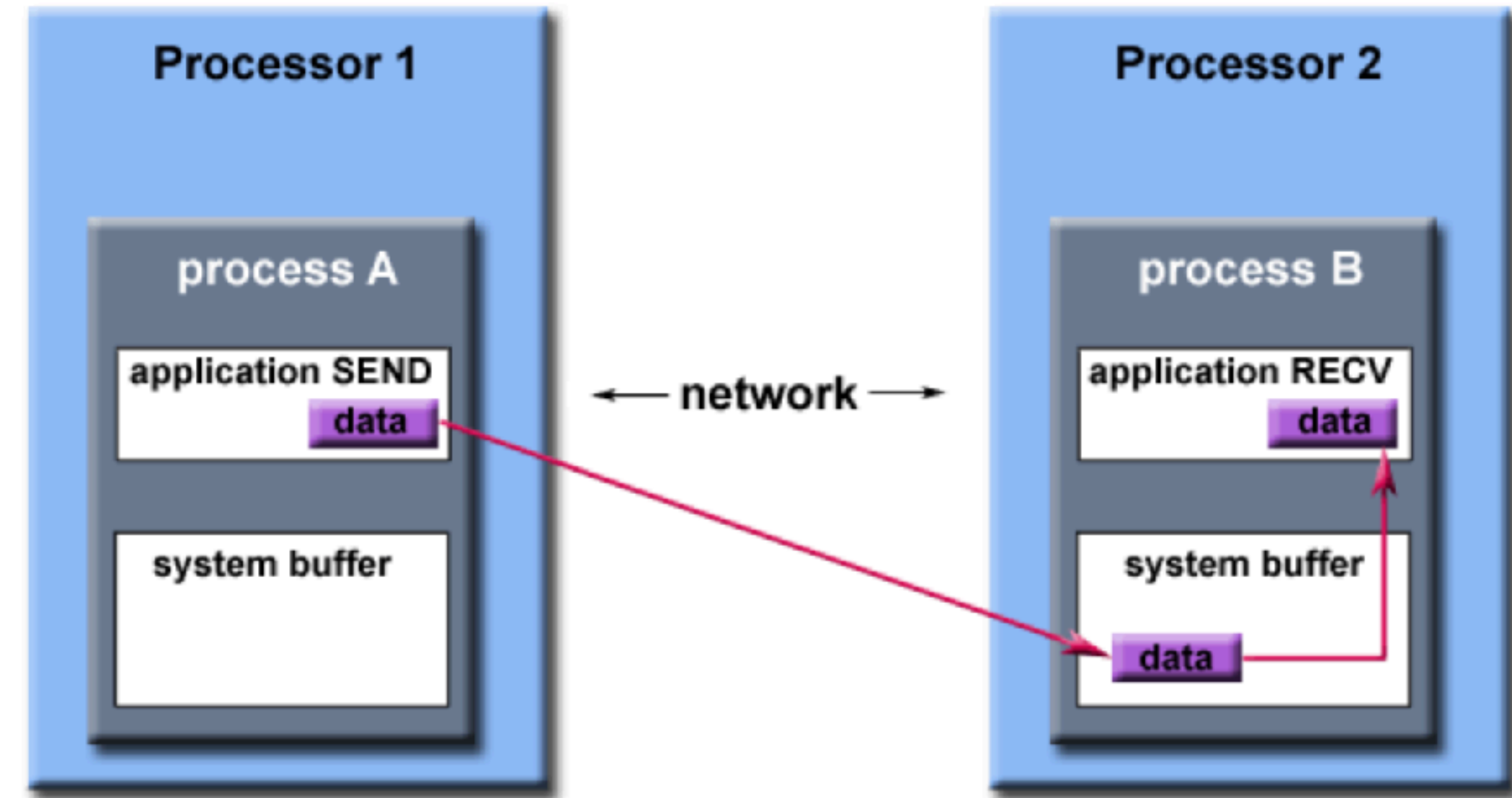
- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
- A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
- A blocking receive only "returns" after the data has arrived and is ready for use by the program.

- **Non-blocking:**

- Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
- Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
- It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.



# Blocking vs. Non-blocking



Path of a message buffered at the receiving process

- System buffer space is:
  - Opaque to the programmer and managed entirely by the MPI library
  - A finite resource that can be easy to exhaust
  - Often mysterious and not well documented
  - Able to exist on the sending side, the receiving side, or both
  - Something that may improve program performance because it allows send - receive operations to be asynchronous.



# Asynchronous vs. Synchronous

## MPI\_Send

Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. Note that this routine may be implemented differently on different systems. The MPI standard permits the use of a system buffer but does not require it. Some implementations may actually use a synchronous send (discussed below) to implement the basic blocking send.

```
MPI_Send (&buf, count, datatype, dest, tag, comm)  
MPI_SEND (buf, count, datatype, dest, tag, comm, ierr)
```

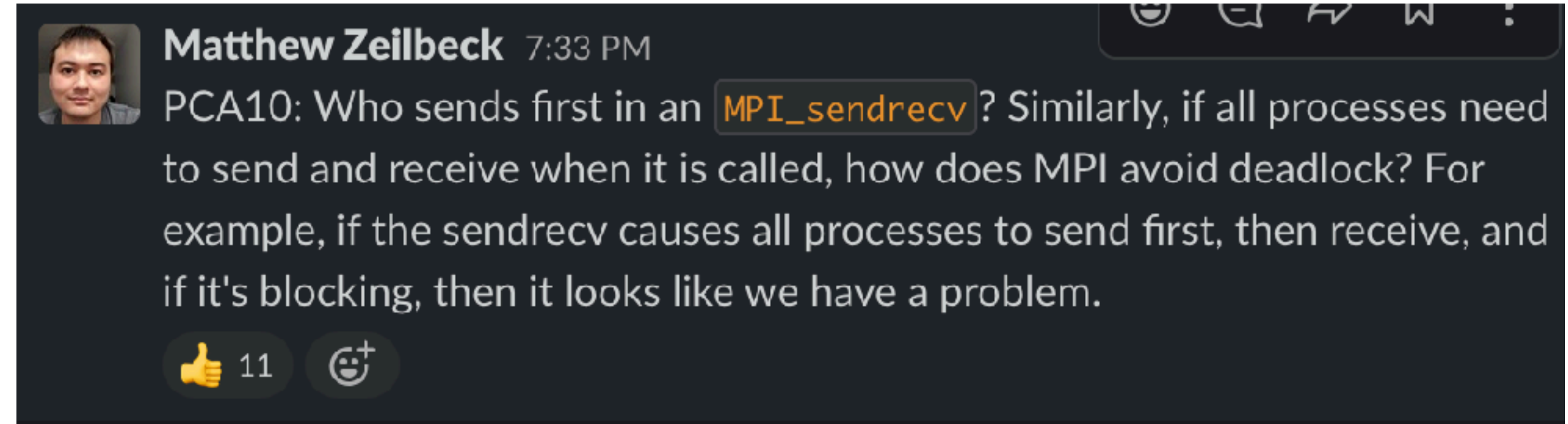
## MPI\_Ssend

Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

```
MPI_Ssend (&buf, count, datatype, dest, tag, comm)  
MPI_SSEND (buf, count, datatype, dest, tag, comm, ierr)
```



# PCA Questions



## MPI\_Sendrecv


Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.

```
MPI_Sendrecv (&sendbuf, sendcount, sendtype, dest, sendtag,  
             &recvbuf, recvcount, recvtype, source, recvtag,  
             comm, &status)  
MPI_SENDRECV (sendbuf, sendcount, sendtype, dest, sendtag,  
             recvbuf, recvcount, recvtype, source, recvtag,  
             comm, status, ierr)
```

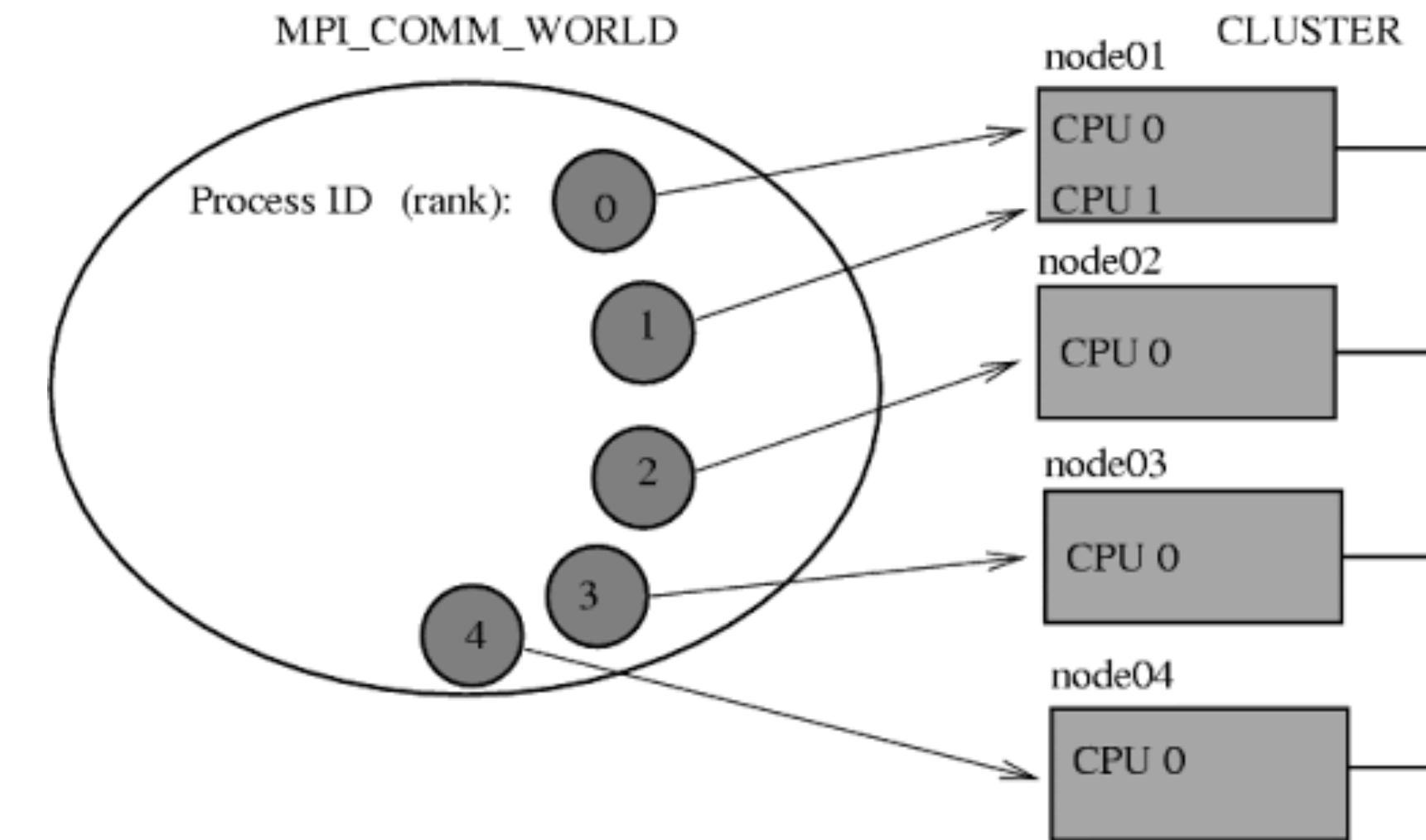
- send and recv posted at same time and either can be completed first.
- blocks until both finished



# PCA Questions

 **Brian Nevins** 6:30 PM  
PCA11: Why don't you want to use processes 0 and 1 for a ping-pong timing? I'm not seeing anything in the text about it, and I'm having trouble seeing why those would be problematic if the timing is the sole purpose of the program

👍 8 🗨️ 1 😊



- Affinity: rank 0 and 1 may be on the same socket!





# PCA Questions

## Load balancing



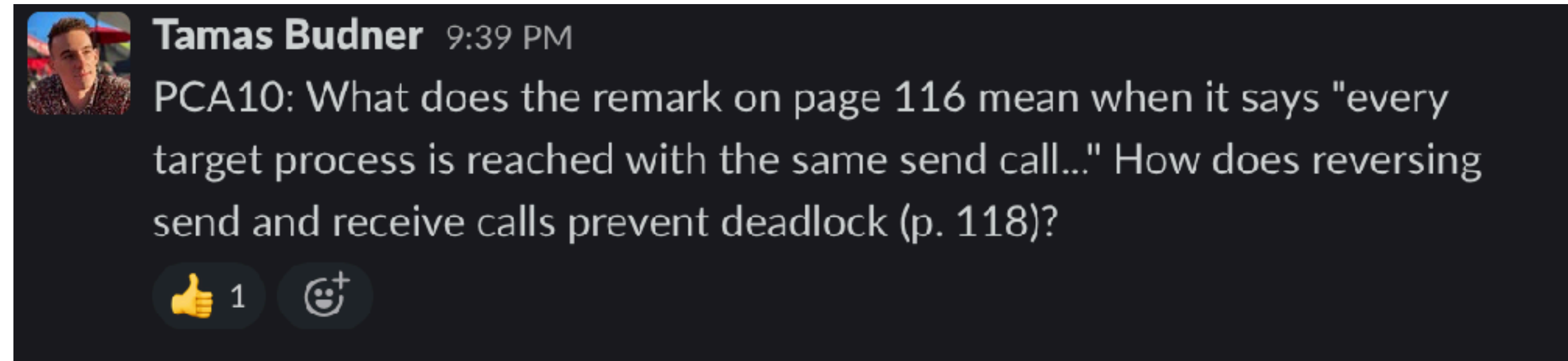
**Nathan Haut** 1:01 PM

PCA10: On page 112 in the book it describes a situation where the number of items can't be evenly divided among the number of processors so it adds the excess to the last processor. Wouldn't this be a really poor way of dealing with the excess since in the worst case scenario it could nearly double the amount of data added to the last processor. Wouldn't it be better to go through and add one additional item to each processor until all items are given to processors, so rather than the worst case being nearly doubling the data on a single processor the worst case would be one extra item on almost all processors?

- Yes! Load balancing is always tricky. Complex cases might involve cases where “work-per-data” is not uniform, too...



# PCA Questions



**Remark 5** *The structure of the send call shows the symmetric nature of MPI: every target process is reached with the same send call, no matter whether it's running on the same multicore chip as the sender, or on a computational node halfway across the machine room, taking several network hops to reach. Of course, any self-respecting MPI implementation optimizes for the case where sender and receiver have access to the same shared memory. However, even then, there will be a copy operation from the sender buffer to the receiver buffer, so there is no actual memory sharing going on.*

The following code is guaranteed to block, since a **MPI\_Recv** always blocks:

```
// recvblock.c
other = 1-procno;
MPI_Recv(&recvbuf, 1, MPI_INT, other, 0, comm, &status);
MPI_Send(&sendbuf, 1, MPI_INT, other, 0, comm);
printf("This statement will not be reached on %d\n", procno);
```

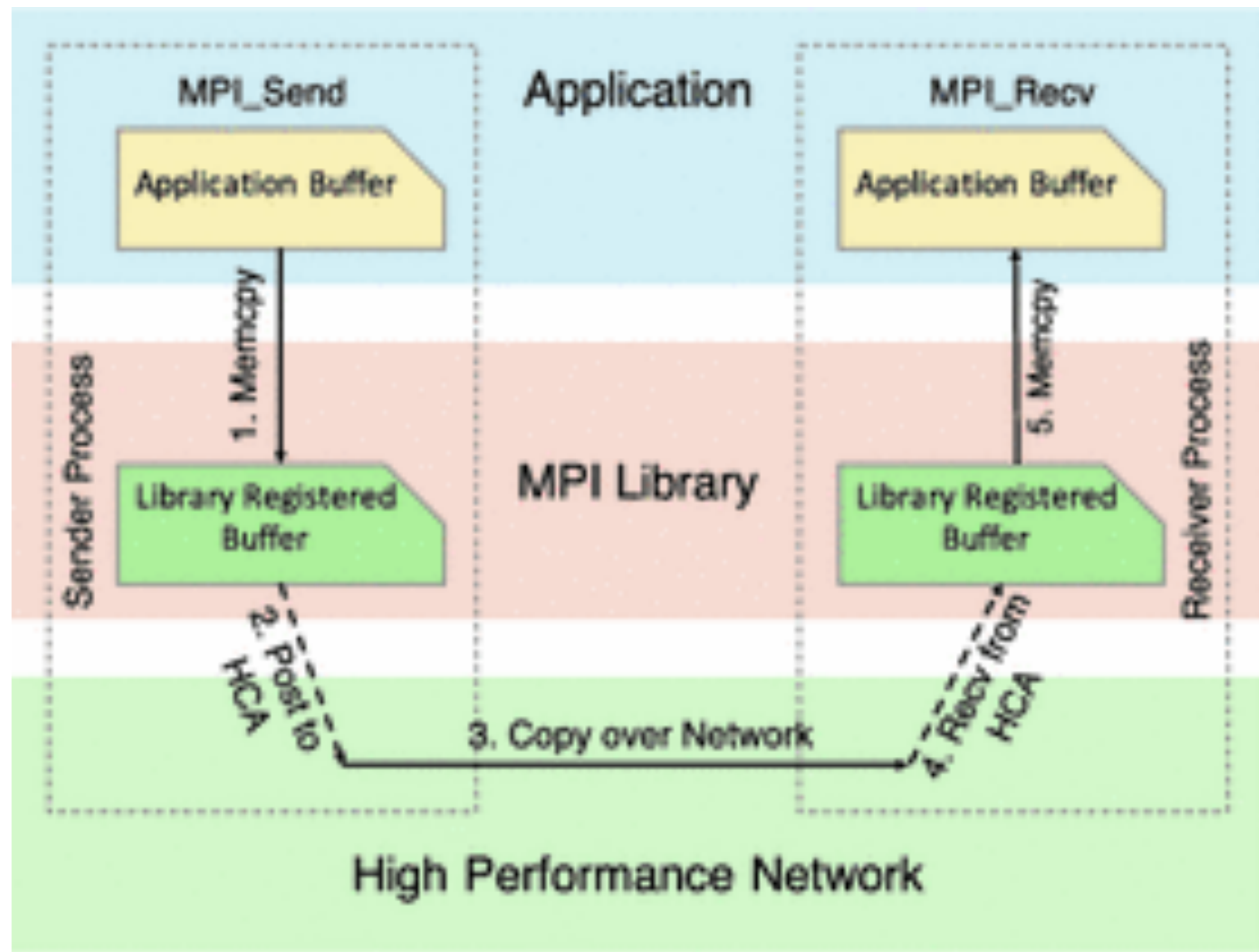
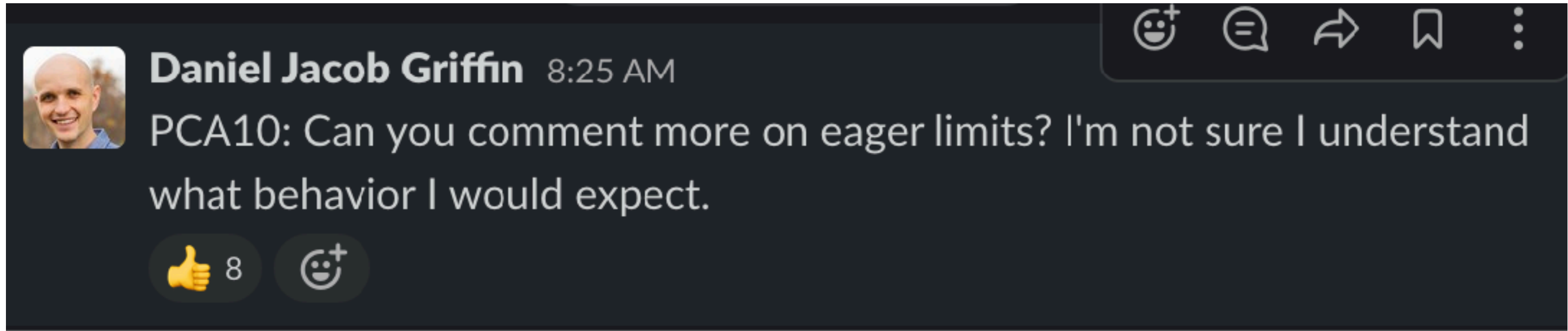
*For the source of this example, see section 4.6.5*

On the other hand, if we put the send call before the receive, code may not block for small messages that fall under the eager limit.

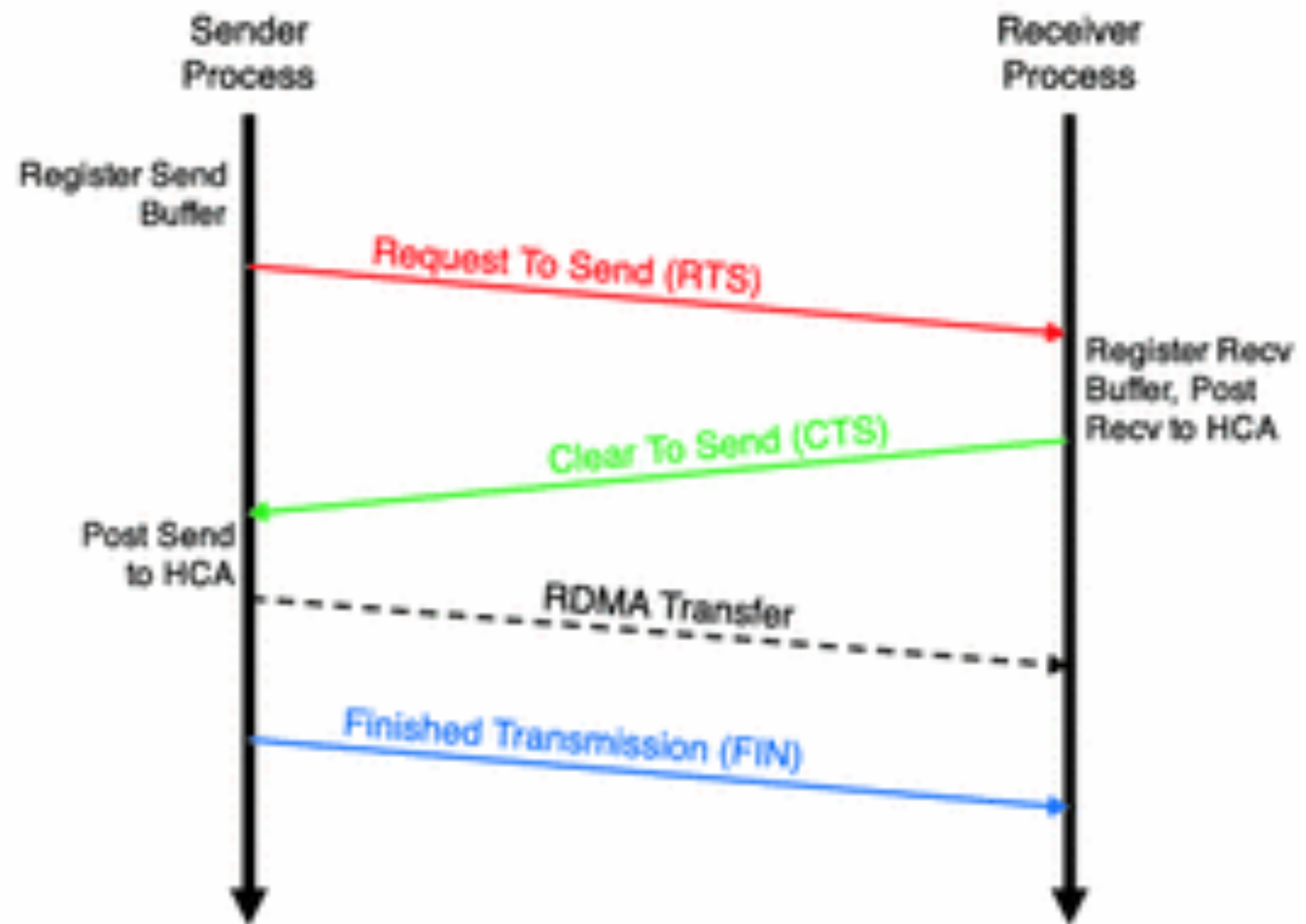


# PCA Questions

## Eager limit



(a) Eager Protocol



(b) Rendezvous Protocol



# PCA Questions

## MPI Persistent

- Better performance? Not necessarily...
- More advantage for collectives
- Less overheads
- Repeated communication in inner loop

```
// persist.c
if (procno==src) {
    MPI_Send_init(send, s, MPI_DOUBLE, tgt, 0, comm, requests+0);
    MPI_Recv_init(recv, s, MPI_DOUBLE, tgt, 0, comm, requests+1);
    printf("Size %d\n", s);
    t[cnt] = MPI_Wtime();
    for (int n=0; n<NEXPERIMENTS; n++) {
        fill_buffer(send, s, n);
        MPI_Startall(2, requests);
        MPI_Waitall(2, requests, MPI_STATUSES_IGNORE);
        int r = chck_buffer(send, s, n);
        if (!r) printf("buffer problem %d\n", s);
    }
    t[cnt] = MPI_Wtime() - t[cnt];
    MPI_Request_free(requests+0); MPI_Request_free(requests+1);
} else if (procno==tgt) {
    for (int n=0; n<NEXPERIMENTS; n++) {
        MPI_Recv(recv, s, MPI_DOUBLE, src, 0, comm, MPI_STATUS_IGNORE);
        MPI_Send(recv, s, MPI_DOUBLE, src, 0, comm);
    }
}
```