

Parallel Computing for Science & Engineering Spring 2013: MPI point-to-point 1

Instructors:

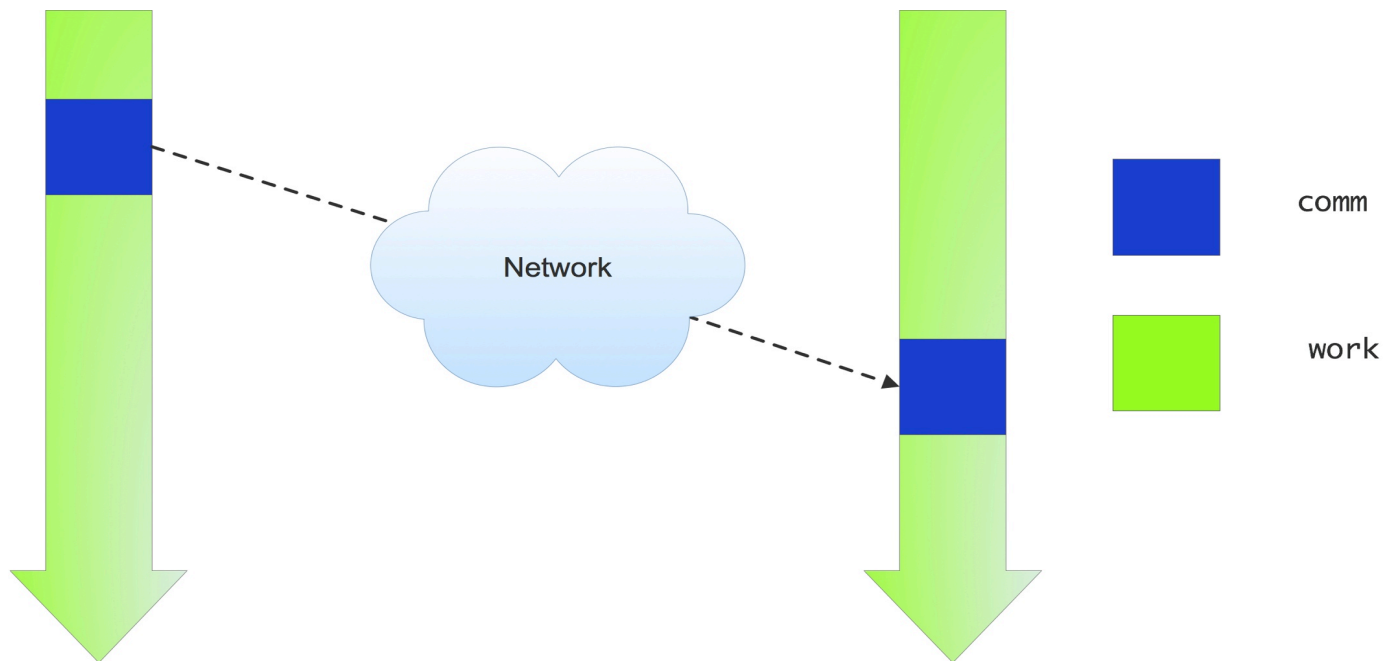
Victor Eijkhout, Research Scientist, TACC

Kent Milfeld, Research Associate, TACC



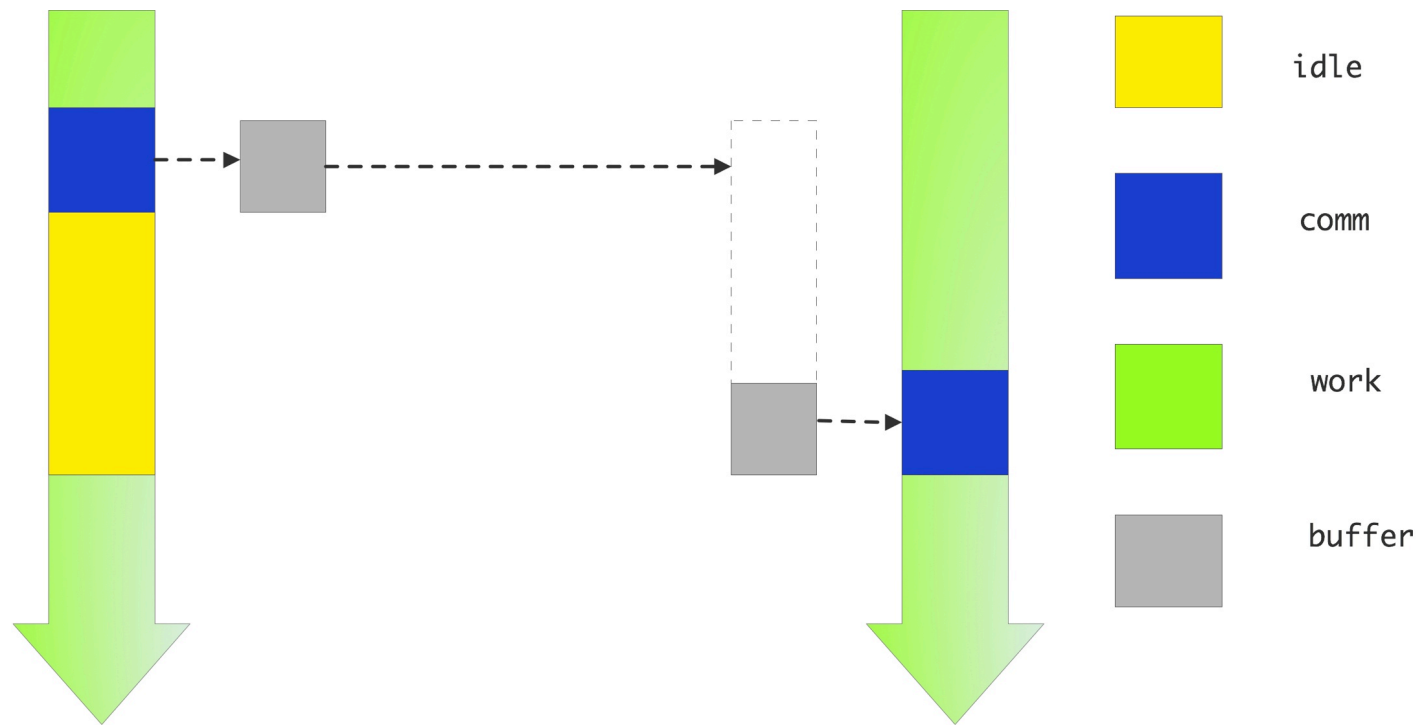
Life would be simple if....

- Processors would just send and receive, and the network would DWIM



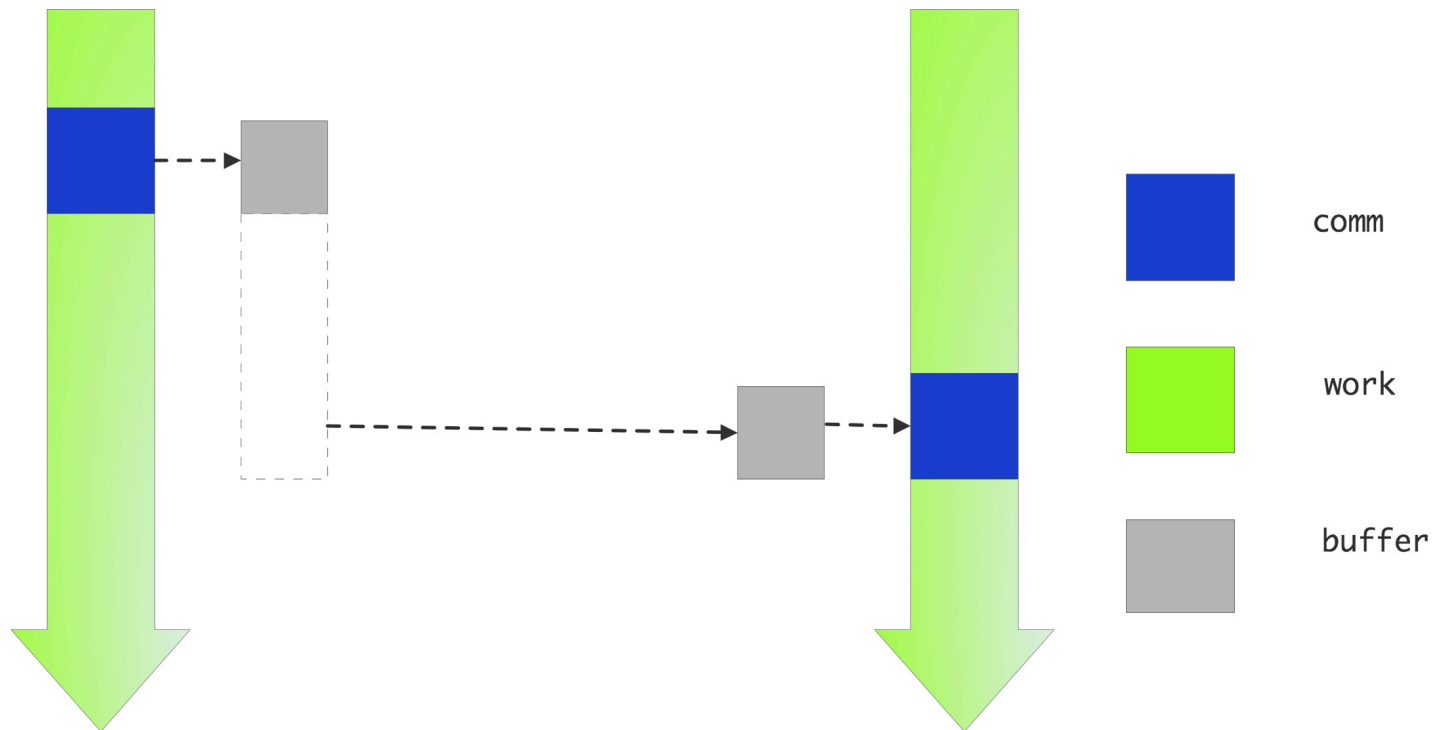
Unfortunately

- Data has to be somewhere: on one process or the other



Non-blocking solution

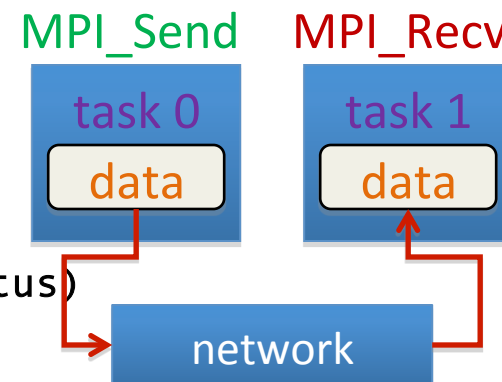
- Create a buffer and let the send data sit there until someone picks it up



Blocking Send/Receive

Generic Syntax

- `MPI_Send(buf, count, datatype, dest, tag, comm)`
- `MPI_Recv(buf, count, datatype, source, tag, comm, status)`
- When MPI sends a message, it doesn't just send the contents; it also sends an *envelope* describing the contents:



Argument	Description
<code>buf</code>	initial address of send/receive buffer (reference)
<code>count</code>	number of items to send (integer)
<code>datatype</code>	MPI data type of items to send/receive
<code>dest</code>	MPI rank of task receiving the data (integer)
<code>source</code>	MPI rank of task sending the data (integer)
<code>tag</code>	message ID (integer)
<code>comm</code>	MPI communicator (set of exchange processors)
<code>status</code>	returns information on the message received

Parts of a P-2-P Communication:

- Data
- Send to/Recv from
- Message ID

Details

buffer	data (address in C, name of array/value in Fortran)
count	Length of source array (in elements, 1 for scalars)
datatype	Data Type: e.g. MPI_INT (C), MPI_INTEGER (F90), MPI_DOUBLE_PRECISION (F90), MPI_DOUBLE (C), etc.
source	Rank (proc #) of source in communicator group
tag	Message identifier (arbitrary integer)
communicator	Group of processors
status	Information about message
ierr	Error (argument in Fortran, returned in C)

	C	Fortran
status	<code>MPI_Status mystat;</code>	<code>integer mystat(MPI_STATUS_SIZE)</code>
datatype	<code>MPI_Datatype mytype;</code>	<code>integer mytype</code>
comm.	<code>MPI_Comm mycomm;</code>	<code>integer mycomm</code>

Language Example

- C

```
ierr=MPI_Send(&a[0], cnt, type, dest, tag, com);
```

- F

```
call MPI_Send( a, cnt, type, dest, tag, com, ierr)
```

- C

```
ierr=MPI_Recv(&b[0], cnt, type, src, tag, com, &status);
```

- F

```
call MPI_Recv( b, cnt, type, src, tag, com, status, ierr)
```

- Call blocks until send data of a has been sent or copied to a buffer. Recv's block until data is in b .

P-2-P Example

```
#include <mpi.h>
int main(int argc, char* argv[]){
MPI_Comm Comm=MPI_COMM_WORLD;
int npes, iam=-1, ierr;

ierr=MPI_Init(&argc, &argv);
ierr=MPI_Comm_size(Comm, &npes);
ierr=MPI_Comm_rank(Comm, &iam);

ierr=MPI_Finalize();

printf("iam=%d\n", iam);
}
```


P-2-P Example

```
#include <mpi.h>
int main(int argc, char* argv[]){
MPI_Comm  Comm=MPI_COMM_WORLD;
MPI_Status status;
int npes, iam=-1, ierr, irec=-1;
ierr=MPI_Init(&argc, &argv);
ierr=MPI_Comm_size(Comm, &npes);
ierr=MPI_Comm_rank(Comm, &iam);

if(iam==0)
    ierr=MPI_Send(&iam, 1, MPI_INT, 1, 9, Comm);
if(iam==1)
    ierr=MPI_Recv(&irec, 1, MPI_INT, 0, 9, Comm, &status);
ierr=MPI_Finalize();

printf("iam=%d, received=%d\n", iam, irec);
}
```

The 6 Basic MPI Call Summary

- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- The 6 basic calls in MPI are

```
MPI_Init(&argc, &argv);  
MPI_Comm_Rank(MPI_COMM_WORLD, &myid);  
MPI_Comm_Size(MPI_COMM_WORLD, &numprocs);  
MPI_Send(buffer, count, MPI_TYPE, dest, tag, MPI_COMM_WORLD);  
MPI_Recv(buffer, count, MPI_TYPE, src, tag, MPI_COMM_WORLD, &stat);  
MPI_Finalize();
```

MPI_TYPE is an MPI Parameter or User Data Type
buffer is passed by reference

MPI_SendRecv

```
MPI_SendRecv (senddat, sendcount, sendtype, dest, sendtag,  
              recvdat, recvcount, recvtype, src, recvtag,  
              comm, status)
```

- Initiates send and receive at the same time.
- Completes when both send and receive buffers are safe to use
- Useful for communications patterns where each node sends and receives messages (two-way communication). Good for avoiding deadlock, implementing shifts/rings.
- Executes a **standard mode** send & receive operation for dest and src, respectively.
- The send and receive operations use the same communicator, but have distinct tags.

Bidirectional Communication with MPI_Sendrecv

- C

```
ierr=MPI_Sendrecv(&sb[0],scnt,stype,dest,stag,  
                  &rb[0],rcnt,rtype, src,rtag,  
                  MPI_COMM_WORLD,&status);
```

- Fortran

```
call MPI_Sendrecv( sb,  scnt,stype,dest,stag,  
                  rb,  rcnt,rtype, src,rtag,  
                  MPI_COMM_WORLD, status,ierr)
```

Blocking vs Non-blocking

Blocking

- A blocking send routine will only return after it is *safe* to modify the **data area**.
- *Safe* means that modifications in the data area will not affect the data to be sent.
- A *Safe send* does not imply that the data was actually received.
- A blocking send can be either synchronous or asynchronous.

Non-blocking

- Send/receive routines return immediately.
- Non-blocking operations request the MPI library to perform the operation when possible.
- It is **unsafe to modify the data area** until the requested operation has been performed. There are *wait* routines used to do this (MPI_Wait)
- Primarily used to overlap computation with communication

Blocking vs non-Blocking Routines

Description	Syntax for C bindings
Blocking send	<code>MPI_Send(buf, count, datatype, dest, tag, comm)</code>
Non-blocking send	<code>MPI_Isend(buf, count, datatype, dest, tag, comm, request)</code>
Blocking receive	<code>MPI_Recv(buf, count, datatype, source, tag, comm, status)</code>
Non-blocking receive	<code>MPI_Irecv(buf, count, datatype, source, tag, comm, request)</code>
Wait for completion	<code>MPI_Wait(request, status)</code>

request: used by non-blocking send and receive operation.

Non-blocking Communication

- Non-blocking send
 - send call returns immediately
 - send actually occurs later
- Non-blocking receive
 - receive call returns immediately
 - when received data is needed, call a wait subroutine
- Non-blocking communication used to overlap communication with computation (and communication with communication!).
- Can be used to prevent deadlock.

Non-blocking Send with MPI_Isend

- C

```
MPI_Request request;  
ierr = MPI_Isend(&data, count, datatype,  
                dest, tag, comm, &request);
```

- Fortran

```
integer request  
call MPI_Isend( data, count, datatype,  
                dest, tag, comm, request, ierr)
```

- **request** is the id for the message call
- Don't use **data** area until communication is complete

Non-blocking Receive with MPI_Irecv

- C

```
MPI_Request request;  
ierr = MPI_Irecv(&data, count, datatype,  
                source, tag, comm, &request);
```

- Fortran

```
integer request  
call MPI_Irecv( data, count, datatype,  
               source, tag, comm, request, ierr)
```

- **request** is an id for communication
- Note: There is **no status parameter**.
- Don't use **data** area until communication is complete

MPI_Wait Used to Complete Communication

- **request** from **MPI_Isend** or **MPI_Irecv**
 - the completion of a send operation indicates that the sender is now free to update the data in the send buffer
 - the completion of a receive operation indicates that the receive buffer contains the received message
- **MPI_Wait** blocks until message specified by **request** completes

MPI_Wait Usage

- C

```
MPI_Request request;  
MPI_Status status;  
...  
ierr = MPI_Wait(&request, &status)
```

- Fortran

```
integer request  
integer status(MPI_STATUS_SIZE)  
...  
call MPI_Wait(    request,    status, ierr)
```

Nonblocking Examples

One-way Send/Recv Communication

- **Blocking Send & Non-blocking Recv**

```
if (rank==0) then
    call MPI_send(    sendbuf, count, MPI_REAL, 1, tag, MPI_COMM_WORLD,
                     ierr)
elseif (rank==1) then
    call MPI_Irecv(    recvbuf,  count, MPI_REAL, 0, tag, MPI_COMM_WORLD,
                     req, ierr)
    call MPI_Wait(    req,status,ierr)
endif
```

- **Non-blocking Send & Non-blocking Recv**

```
if (rank==0) then
    call MPI_Isend(    sendbuf,count,MPI_REAL,
                     1,tag,MPI_COMM_WORLD,req,ierr)
elseif (rank==1) then
    call MPI_Irecv(    recvbuf,count,MPI_REAL,
                     0,tag,MPI_COMM_WORLD,req,ierr)
endif
    call MPI_Wait(    req,status,ierr)
```

Two-way Communication: Deadlock

Deadlock 1 (always deadlocks)

```
if (rank==0) then
    call MPI_Recv(    recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
    call MPI_Send(    sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
elseif (rank==1) then
    call MPI_Recv(    recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
    call MPI_Send(    sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
endif
```

Deadlock 2 (deadlocks when system buffer is too small)

```
if (rank==0) then
    call MPI_Send(    sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
    call MPI_Recv(    recvbuf,count,MPI_REAL,
    1,tag,MPI_COMM_WORLD,status,ierr)
elseif (rank==1) then
    call MPI_Send(    sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
    call MPI_Recv(    recvbuf,count,MPI_REAL,
    0,tag,MPI_COMM_WORLD,status,ierr)
endif
```

Two-way Communication: Solutions

Solution 1 (but this doesn't allow bidirectional communication)

```
if (rank==0) then
    call MPI_Send( sendbuf,count,MPI_REAL, 1,tag,MPI_COMM_WORLD,ierr)
    call MPI_Recv( recvbuf,count,MPI_REAL, 1,tag,MPI_COMM_WORLD,status,ierr)
elseif (rank==1) then
    call MPI_Recv( recvbuf,count,MPI_REAL, 0,tag,MPI_COMM_WORLD,status,ierr)
    call MPI_Send( sendbuf,count,MPI_REAL, 0,tag,MPI_COMM_WORLD,ierr)
endif
```

Solution 2

```
if (rank==0) then
    call MPI_SendRecv( sendbuf,sendcount,sendtype,dest,sendtag,
                      recvbuf,recvcount,
                      recvtype,source,recvtag,MPI_COMM_WORLD,status,ierr)
elseif (rank==1) then
    call MPI_SendRecv( sendbuf,sendcount,sendtype,dest,sendtag,
                      recvbuf,recvcount,
                      recvtype,source,recvtag,MPI_COMM_WORLD,status,ierr)
endif
```

Two-way Communication: Solutions

Solution 3

```
if (rank==0) then
    call MPI_Isend(    sendbuf,count,MPI_REAL,
    1,tag,MPI_COMM_WORLD,req1,ierr)
    call MPI_Irecv(    recvbuf,count,MPI_REAL,
    0,tag,MPI_COMM_WORLD,req2,ierr)
elseif (rank==1) then
    call MPI_Isend(    sendbuf,count,MPI_REAL,
    0,tag,MPI_COMM_WORLD,req1,ierr)
    call MPI_Irecv(    recvbuf,count,MPI_REAL,
    1,tag,MPI_COMM_WORLD,req2,ierr)
endif
    call MPI_Wait(    req1,status,ierr)
    call MPI_Wait(    req2,status,ierr)
```

Solution 4

```
if (rank==0) then
    call MPI_Bsend(    sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
    call MPI_Recv(    recvbuf,count,MPI_REAL,
    0,tag,MPI_COMM_WORLD,status,ierr)
elseif (rank==1) then
    call MPI_Bsend(    sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
    call MPI_Recv(    recvbuf,count,MPI_REAL,
    0,tag,MPI_COMM_WORLD,status,ierr)
endif
```


Two-way Communications Summary

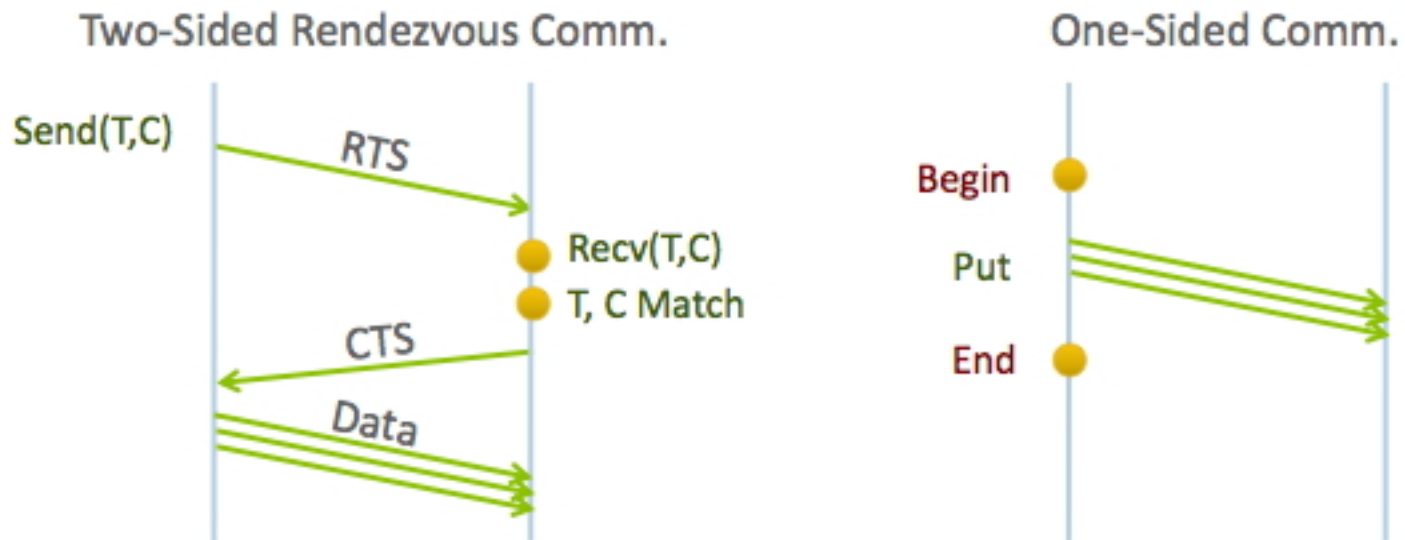
	CPU 1	CPU 2
Deadlock 1	Recv/Send	Recv/Send
Deadlock 2	Send/Recv	Send/Recv
Solution 1	Send/Recv	Recv/Send
Solution 2	SendRecv	SendRecv
Solution 3	Isend/Irecv/Wait	Isend/Irecv/Wait
Solution 4	Bsend/Recv	Bsend/Recv

Wait types

- `MPI_Wait` : wait for one request
- `MPI_Waitall` : wait for an array of requests, good for load balanced tasks, or when all needed
- `MPI_Waitany` : wait for one in an array of requests, good for unbalanced tasks, or if they can be processed individually
- `MPI_Waitsome` : wait for any number in an array, much like `MPI_Waitany`

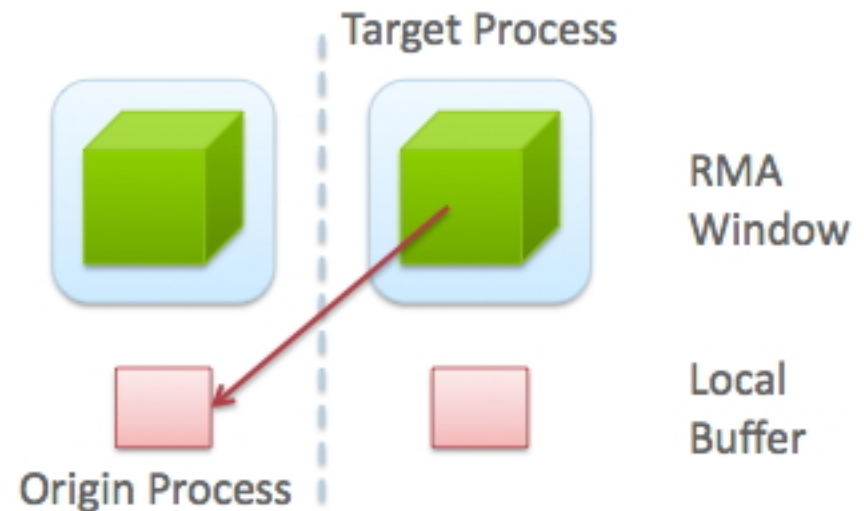
One-Sided or RMA

- It would be nice to avoid that two-way orchestration: just write into another process' memory or read from it
- Less overhead, easier to code



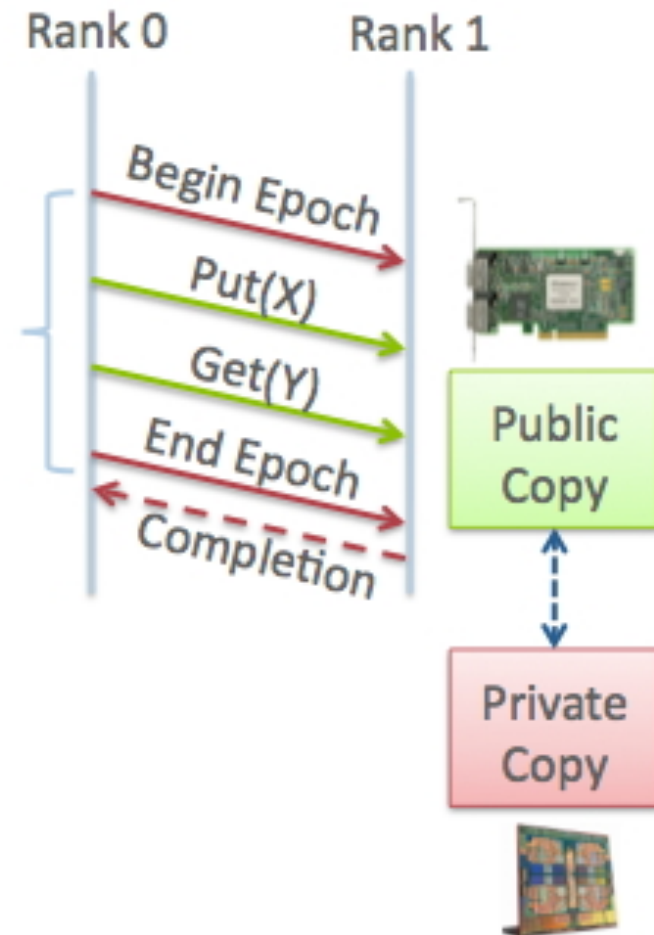
One-Sided concepts

- Target & origin processes: origin issues the call, target does nothing explicit
- Window & local memory: window is accessible to others



More RMA concepts

- Origin vs Target, window vs local mem
- Actions: Put, Get, Accumulate
- Epoch: just like MPI_Wait: you have to make sure data has arrived



RMA routines

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,  
    MPI_Comm comm, MPI_Win *win)
```

```
MPI_Get( origin_addr, origin_count, origin_datatype,  
    target_rank, target_disp, target_count, target_datatype,  
    win)
```

```
MPI_Win_fence(assert,win)
```

Use of fences is one way to synchronize. There are more.

Fences

- `MPI_Win_fence(assert,win)`
- Assertions:
 - `MPI_MODE_NOSTORE`
 - `MPI_MODE_NOPUT`
 - `MPI_MODE_NOPRECEDE`
 - `MPI_MODE_NOSUCCEED`
- Example:
`MPI_Win_fence(
 (MPI_MODE_NOSTORE|MPI_MODE_NOPRECEDE), win);`

RMA limitations

