COMP 364 / 464
High Performance Computing

Distributed Memory Parallelism:

Point-to-Point (P2P) Communication

Minimal MPI program

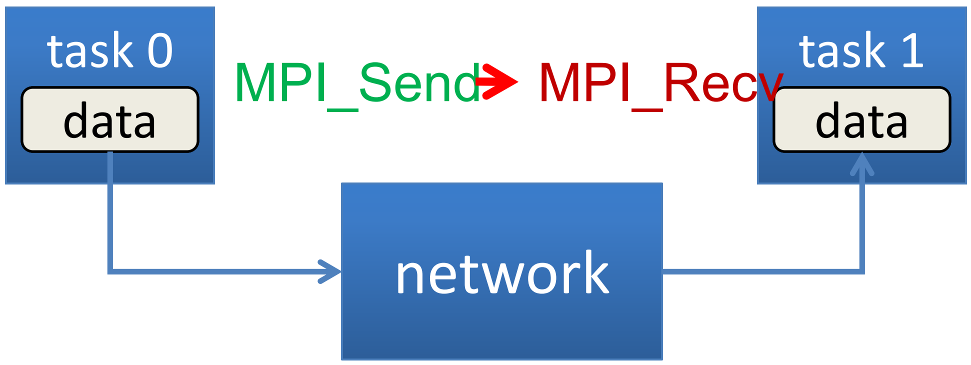
- Every MPI program needs these…
  – C version

```
#include <mpi.h>
 ...
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &numNodes);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
 ...
ierr = MPI_Finalize();
```

- In C MPI routines are functions which return the error value

Point-to-Point Communication

- Sending data from one point (process/task) to another point (process/task)
- One task sends while another receives



Basic Communications in MPI

- Standard **`MPI_Send/MPI_Recv`** routines
  - Blocking calls used for basic messaging

Point-to-Point Modes of Operation

- Blocking
  - Call does not return until the sent and received data is safe to use
- Non-blocking
  - Initiates send or receive operation, returns immediately
  - Can check or wait for completion of the operation
  - Data area is not safe for use until completion.
- Synchronous and Buffered (later)

Data Types (basics)

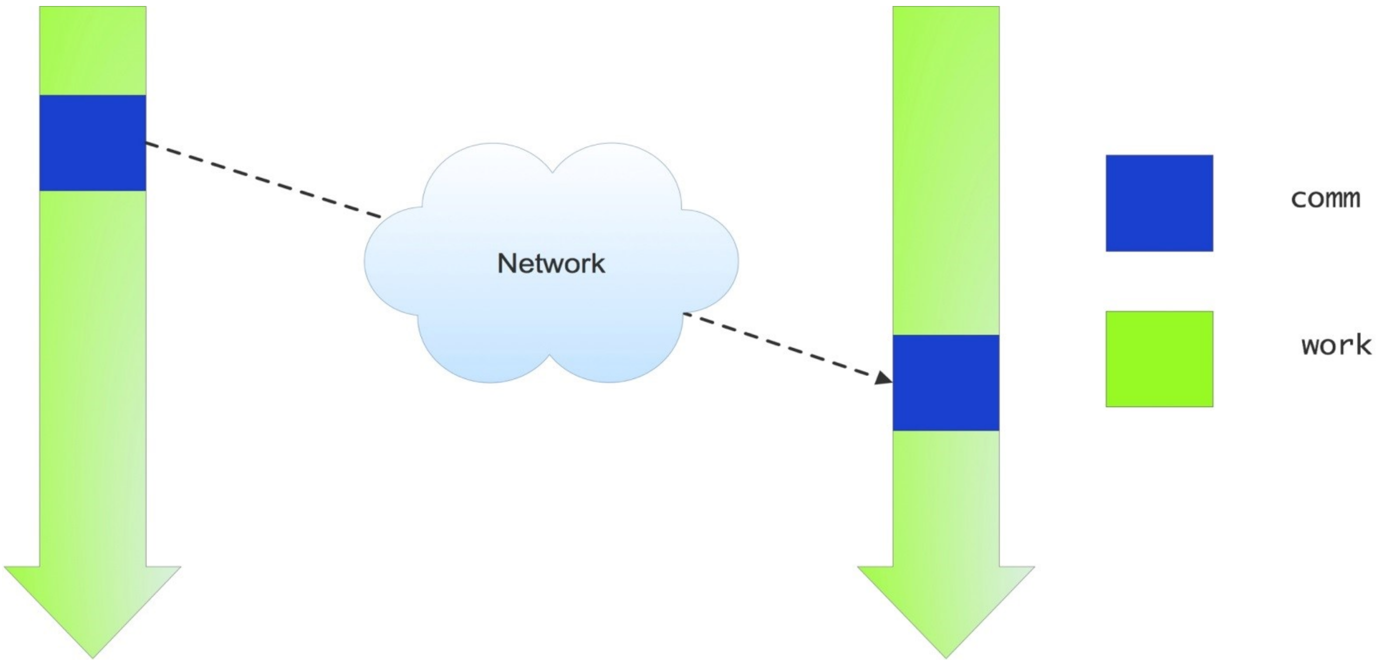
- Data types (more of a mapping than a declaration)
  - Specifies the data type and size in MPI routines
  - Predefined MPI types correspond to language types

Representation	MPI Type C	C
32-bit floating point	`MPI_FLOAT`	`float`
64-bit floating point	`MPI_DOUBLE`	`double`
32-bit integer	`MPI_INT`	`int`

- Methods exists for creating user-defined types
  - Simple (just combinations of normal data types)
  - Advanced (a map of data to be send)

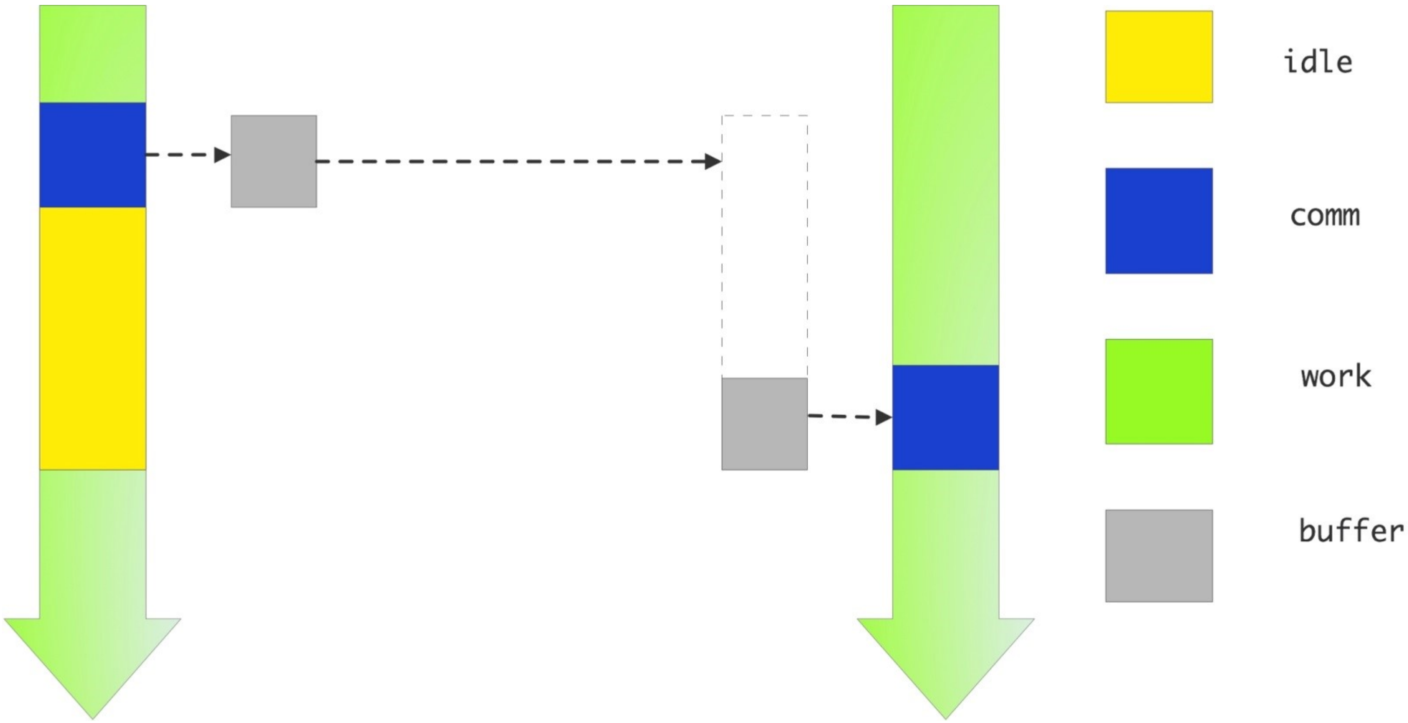
Life would be simple if….

- Processors would just send and receive, and the network would DWIM (do what I meant)



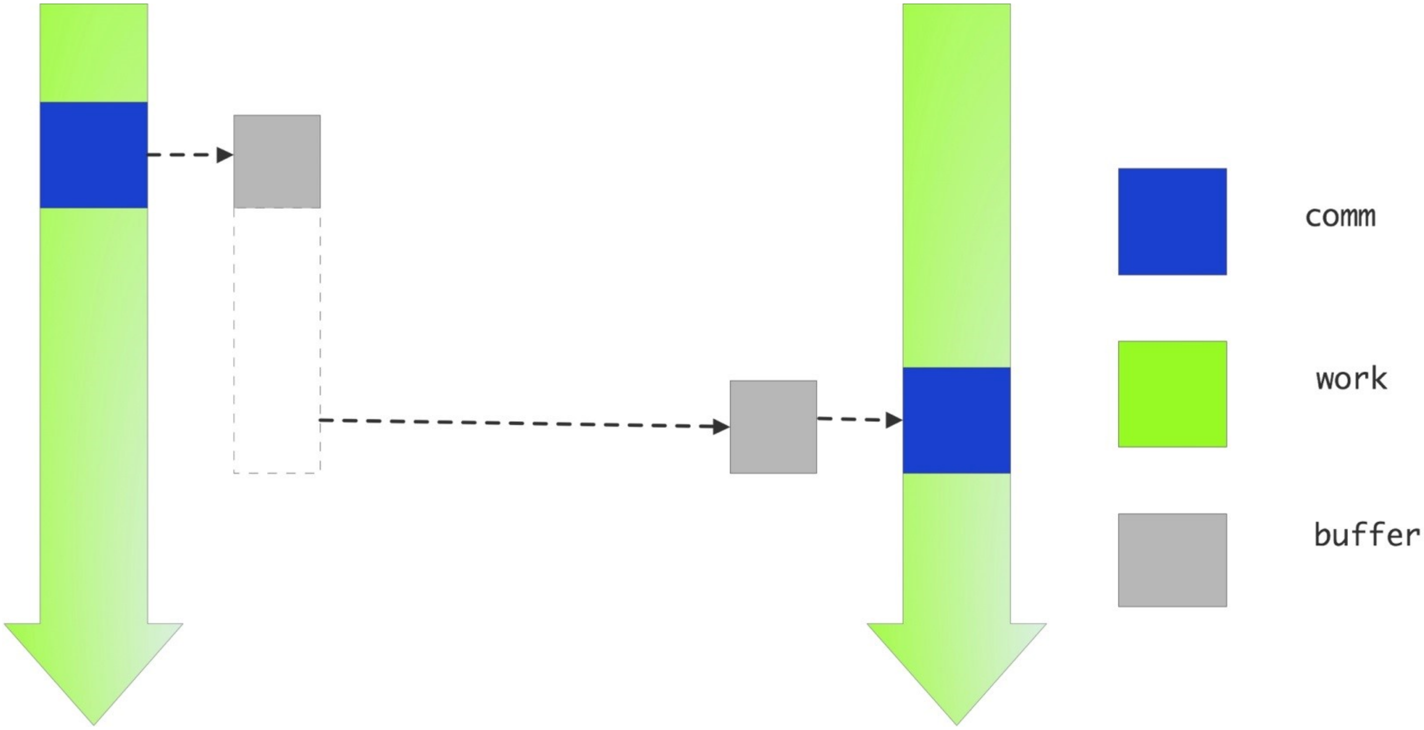
Unfortunately...

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

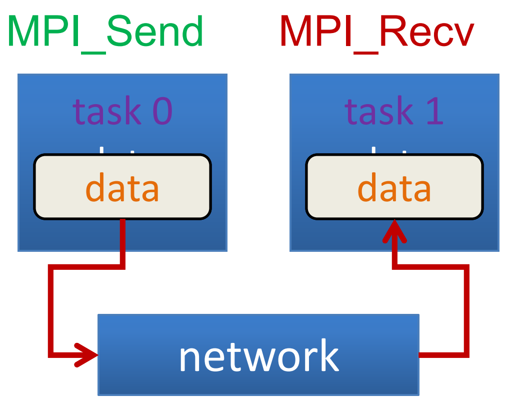


Non-Blocking Solution

- 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
buf	initial address of send/receive buffer (reference): void*
count	number of items to send (integer): int
datatype	MPI data type of items to send/receive
dest	MPI rank of task receiving the data (integer): int
source	MPI rank of task sending the data (integer): int
tag	message ID (integer): int
comm	MPI communicator (set of exchange processors): MPI_Comm
status	returns information on the message received: MPI_Status

Parts of a P-2-P Communication:
- Data
- Send to/Recv from
- Message ID

Language Example

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

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

- 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 numRanks,myRank=-1,ierr;

   ierr=MPI_Init(&argc, &argv);
   ierr=MPI_Comm_size(Comm,&numRanks);
   ierr=MPI_Comm_rank(Comm, &myRank);





   ierr=MPI_Finalize();

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

P-2-P Example

```
#include <mpi.h>
int main(int argc, char* argv[])
{
   MPI_Comm Comm=MPI_COMM_WORLD; // Don't do this!
   MPI_Status status;
   int numRanks = 1, myRank = -1;
   int ierr=MPI_Init(&argc, &argv);
   ierr=MPI_Comm_size(Comm, &numRanks);
   ierr=MPI_Comm_rank(Comm, &myRank);

   int irecv = -1;
   if(myRank==0)
        ierr=MPI_Send(&myRank, 1,MPI_INT, 1,9, Comm);
   if(myRank==1)
        ierr=MPI_Recv(&irecv,1,MPI_INT, 0,9, Comm,&status);
   ierr=MPI_Finalize();

   printf("myRank=%d, received=%d\n",myRank,irecv);
}
```

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(Comm,&myid);
MPI_Comm_Size(Comm,&numprocs);
MPI_Send(s_pointer,count,MPI_TYPE,dest,tag,Comm);
MPI_Recv(r_pointer,count,MPI_TYPE,src,tag,Comm,&stat);
MPI_Finalize();
```

`MPI_TYPE` is an MPI Parameter

MPI_SendRecv

`MPI_SendRecv (sdat, scount, stype, dest, stag, rdat, rcount, rtype, src, rtag, 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.

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	`MPI_Send(buf, count, datatype, dest, tag, comm)`
Non-blocking send	`MPI_Isend(buf, count, datatype, dest, tag, comm, request)`
Blocking receive	`MPI_Recv(buf, count, datatype, source, tag, comm, status)`
Non-blocking receive	`MPI_Irecv(buf, count, datatype, source, tag, comm, request)`
Wait for completion	`MPI_Wait(request, status)`

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

Non-Blocking Communication

- Non-blocking send
  - send call returns immediately
  - send actually occurs later, call mpi_wait() to before destroying / modifying send data.
- Non-blocking receive
  - receive call returns immediately
  - when received data is needed, call a wait() to verify receipt
- Non-blocking communication used to overlap communication with computation (and communication with communication!).
- And … used to prevent deadlock.

Non-Blocking Send With **MPI_Isend**

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

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

Non-Blocking Send With `MPI_Irecv`

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

- `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

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

Non-Blocking Examples

Two-way Communication: Deadlock

- Deadlock 1 (always deadlocks)

```
int other = 1 - myRank;   //! Assume a total of 2 MPI tasks
MPI_Recv(recvbuf, count, MPI_FLOAT, other, tag, MPI_COMM_WORLD, status);
MPI_Send(sendbuf, count, MPI_FLOAT, other, tag, MPI_COMM_WORLD);
```

- Deadlock 2 (deadlocks when system buffer is too small)

```
other = 1-mytid    // Assume a total of 2 MPI tasks
MPI_Send(sendbuf, count, MPI_FLOAT, other, tag, MPI_COMM_WORLD);
MPI_Recv(recvbuf, count, MPI_FLOAT, other, tag, MPI_COMM_WORLD, status);
```

Two-way Communication: Solutions

- Solution 1: specify sends & receives in an order that is guaranteed not to deadlock. This is cumbersome for more than two processors

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

- Solution 2: use a sendrecv instruction. This works for communication where every processors does one send and one receive.

```
other = 1-myRank;      //! Assume there are exactly 2 total tasks
MPI_Sendrecv(   sendbuf,sendcount,sendtype,other,sendtag,
   recvbuf,recvcount, recvtype,other,recvtag,MPI_COMM_WORLD,&status)
```

Two-way Communication: Solutions

- Solution 3: use isend and irecv.
  - This is easier to write and probably more efficient than solution 1.
  - It can deal with more general communication patterns than solution 2.

```
MPI_Isend(sendbuf, count, MPI_FLOAT, other, tag, MPI_COMM_WORLD, &req1);
MPI_Irecv(recvbuf, count, MPI_FLOAT, other, tag, MPI_COMM_WORLD, &req2);
MPI_Wait(req1, &status);
MPI_Wait(req2, &status);
```

Two-way Communications Summary

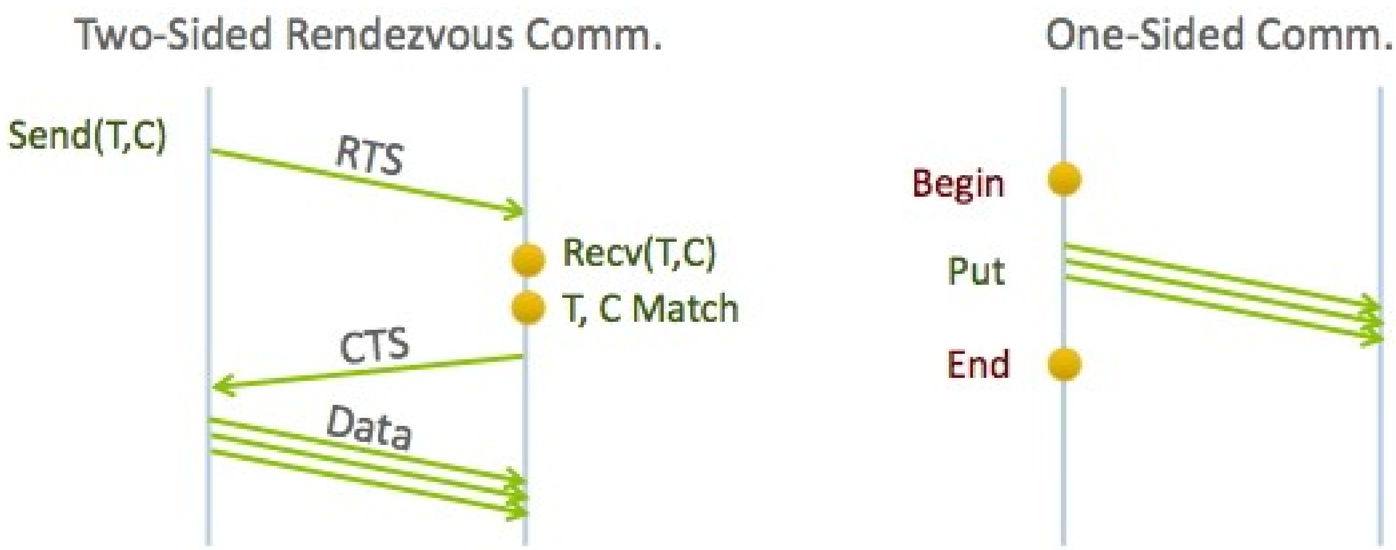
	CPU 1	CPU 2
Deadlock 1	Recv / Send	Recv / Send
Deadlock 2	Send / Recv	Send / Recv
Solution 1	Send / Recv	Recv / Send
Solution 2	SendRecv	SendRecv
Solution 3	Isend / Irecv/ Wait	Isend/ Irecv/ Wait

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 Communications

- 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



Wildcards (C)

- Enables programmer to avoid having to specify a tag and/or source.
- Example:

```
MPI_Status status;
int        data[5];
int        ierr;
ierr = MPI_Recv(&data[0], 5, MPI_INT,
                MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD,&status);
```

- `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are wild cards
- `status` structure is used to get wildcard values

More on Status

- **status** (type **`MPI_Status`**) is a structure which contains three fields **`MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`**
- **`status.MPI_SOURCE`**, **`status.MPI_TAG`**, and **`status.MPI_ERROR`** contain the source, tag, and error code respectively of the received message

Order Semantics

- Messages with the same tag are ordered; for the rest, make no assumptions on message ordering!
  - the first receive always matches the first send in the following

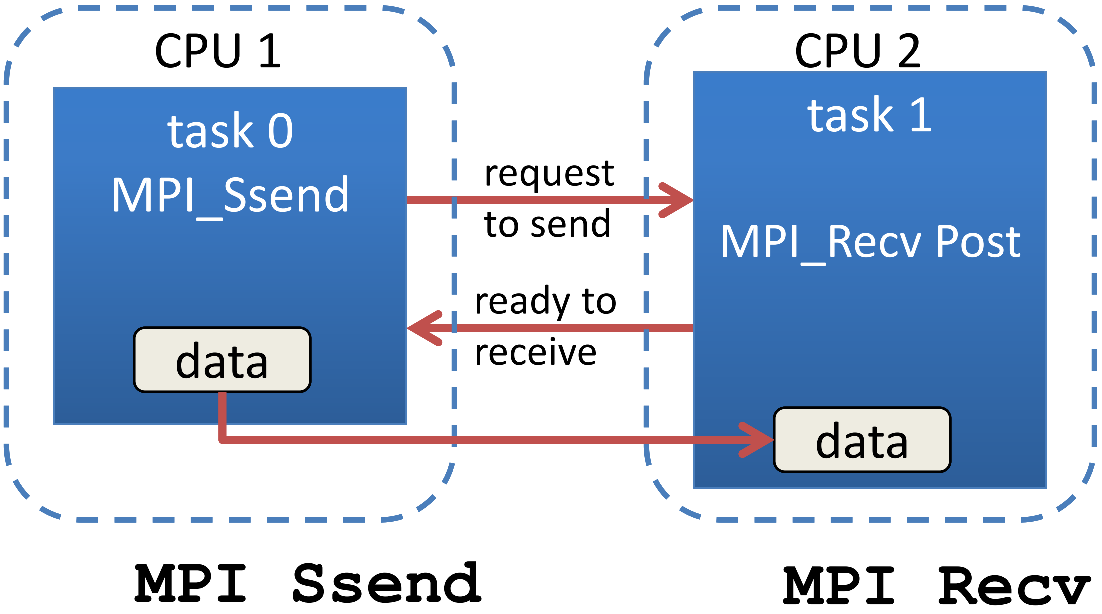
```
tag=123456
if (rank == 0) then
  call MPI_Send(b1,cnt,MPI_REAL,1,tag,comm,err)
  call MPI_Send(b2,cnt,MPI_REAL,1,tag,comm,err)
ELSE ! rank.EQ.1
  call MPI_Recv(b1,cnt,MPI_REAL,0,tag,comm,
                status,ierr)
  call MPI_Recv(b2,cnt,MPI_REAL,0,tag,comm,
                 status, ierr)
END if
```

MPI Receive Modes

IMPORTANT: From the MPI-2.2 Standard:

"There is only one receive operation, but it matches any of the send modes [emphasis mine]. The receive operation described in the last section is blocking: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started)."

Synchronous Communication



- Data isn't sent until Receive has been posted.
- Synchronous send returns when data area is safe for re-use.
- There is no **`MPI_Srecv`**

Synchronous Communication

Ssend

```
    ...
i=1;
if(irank == 0){
   MPI_Ssend(&i, 1, MPI_INT,      1, 9, MPI_COMM_WORLD);
}else {
   MPI_Recv( &j, 1, MPI_INT,      0, 9, MPI_COMM_WORLD, &status);
}
```

`MPI_Test`

- Value of flags signifies whether a message has been delivered
- Similar to **`MPI_Wait`**, but does not block

```
int flag;
ierr= MPI_Test(&request, &flag, &status);
```

MPI_Cancel

- Cancel a pending non-blocking send or receive

```
MPI_Request request;
ierr= MPI_Cancel(&request);
```

MPI_Probe

- **MPI_Probe** allows incoming messages to be checked without actually receiving them
  - the user can then decide how to receive the data
  - Used when different actions need to be taken, depending on the "who, what, and how much" information of the message.

MPI_Probe

```
ierr = MPI_Probe(source, tag, comm, &status);
```

- Parameters
  - source: source rank or `MPI_ANY_SOURCE`
  - tag: tag value or `MPI_ANY_TAG`
  - comm: communicator
  - status: status object

We gratefully acknowledge the sponsorship of Chevron Corporation, whose generous support of TACC has made possible this Scientific Computing Curriculum and other student-focused initiatives.