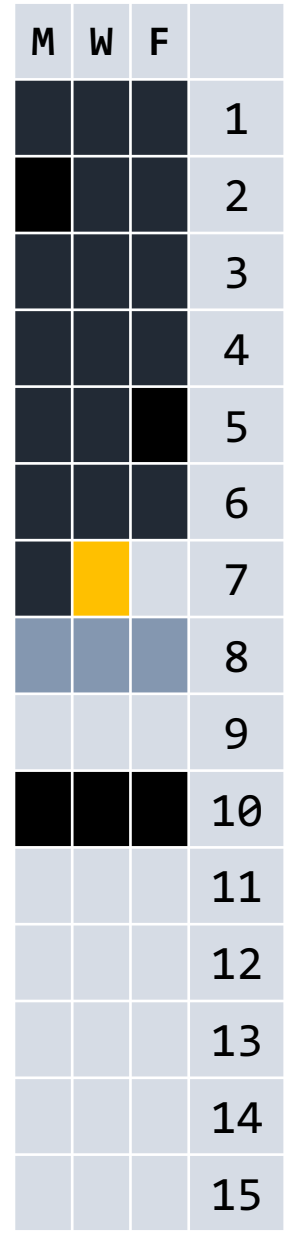
CMOR 421/521:

MPI: Communication and Synchronization



Topics

- Communication in MPI
  - Send-receives
- Synchronization
- Deadlocks

MPI Send

```
int MPI_Send(
    const void* buffer,
    int count,
    MPI_Datatype data_type,
    int dest_rank,
    int tag,
    MPI_Comm comm
);
```

Argument	Description
buffer	Pointer to the data to be sent
count	How many entries to send
data_type	Data type of the data to be sent
dest_rank	The rank to send to
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

MPI: more C-style than C++

- MPI has a stronger C “flavor” than OpenMP
- A lot of pass by pointer

OpenMP	MPI
Thread ID	Rank
Thread	Process
Thread Team	Groups/Communicators

MPI Data Types

Primitive Type	MPI Data Type
int	MPI_INT
long long int	MPI_LONG_LONG
float	MPI_FLOAT
double	MPI_DOUBLE
char	MPI_BYTE

Note on Message Tags:

tag	MPI_ANY_TAG

MPI Send Example

```
int data[] = {0, 1, 2, 3, 4, 5};
int dest_rank = …;

MPI_Send(data, 3, MPI_INT, dest_rank, 0, MPI_COMM_WORLD);
```

MPI Recv (Receive)

```
int MPI_Recv(
    void* buffer,
    int count,
    MPI_Datatype data_type,
    int src_rank,
    int tag,
    MPI_Comm comm,
    MPI_Status* status
);
```

Argument	Description
buffer	Pointer to where to copy the data
count	**AT MOST** how many entries to read
data_type	Data type of the data to be sent
src_rank	The rank to receive from
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)
status	Information about the message

MPI_Status Structure

Fields	Description
int count	Number of *received* entries
int cancelled	Was the request cancelled?
int MPI_SOURCE	Source rank
int MPI_TAG	Tag value
int MPI_ERROR	Any errors associated with the message

If we don't care about the status:

```
MPI_STATUS_IGNORE
```

Directing Sends and Receives

- Send/recv doesn't specify which rank to send/recv
- We need to specify that ourselves

```
int data* = NULL; // Rank 2: {0, 1, 2, 3, 4, 5}; Rank 3: {0…}

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if( rank == 2 )
    MPI_Send(data, 3, MPI_INT, 3, 0, MPI_COMM_WORLD);
if( rank == 3 )
    MPI_Recv(data, 3, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
```

Synchronization in MPI

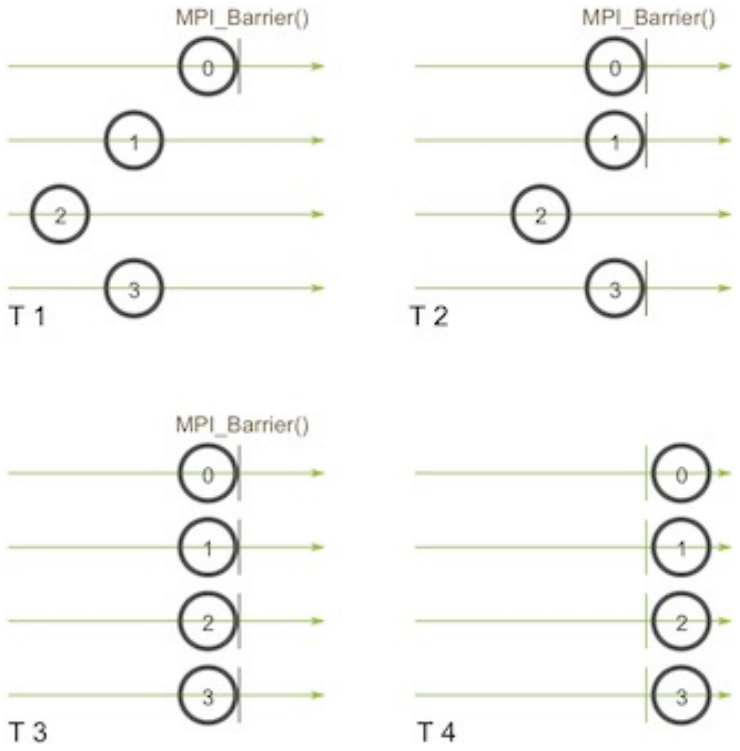
- Like OpenMP, MPI allows for both explicit and implicit synchronization among parallel workers

- Unlike OpenMP, there are also synchronization considerations in communication steps

Synchronization: MPI Barrier

- Synchronizes an entire communicator (group of threads/ranks/processes) like MPI_COMM_WORLD
- Timing, testing, and load balancing: parallel programs are more efficient without barriers.

```
MPI_Barrier(MPI_Comm comm);

// For us:
MPI_Barrier(MPI_COMM_WORLD);
```



Implicit Synchronization

- MPI_Send and MPI_Recv are what is called “blocking” communication
- They cause the affected processes to wait for the communication to complete
  - What we want in a lot of cases
  - But not what we want in every case

Parallel Safety

Things to consider in parallel programs:

- **Race conditions**
  - Do multiple threads try to read/write to the same variable in parallel?
- **Thread safety**
  - Will the result be valid when called in parallel?
- **Deadlocks**
  - Can the synchronization get stuck?

Synchronization Issues

- Synchronization can lead to a ***deadlock***
- A deadlock is when some processes cannot move forward in the program, causing the program to hang

```
// OpenMP Example
if ( thread_ID == 0 ) {
    #pragma omp barrier
}
// Not all threads can see the
// barrier!
```

```
// MPI Example
if ( rank == 0 ) {
    MPI_Barrier(MPI_COMM_WORLD);
}
// Not all threads can see the
// barrier!
```

MPI_Send/Recv and Synchronization

- **MPI_Send:** *Blocks* until the send buffer is safe to use again
  - Might mean it has been received by the destination, just that it is safe to use (it has been copied somewhere)
- **MPI_Recv:** *Blocks* until the receiving buffer is safe to use
  - i.e., it has received its message

Deadlocks and Communication

- Blocking communication can cause deadlocks!
- Both ranks are waiting for the other to receive

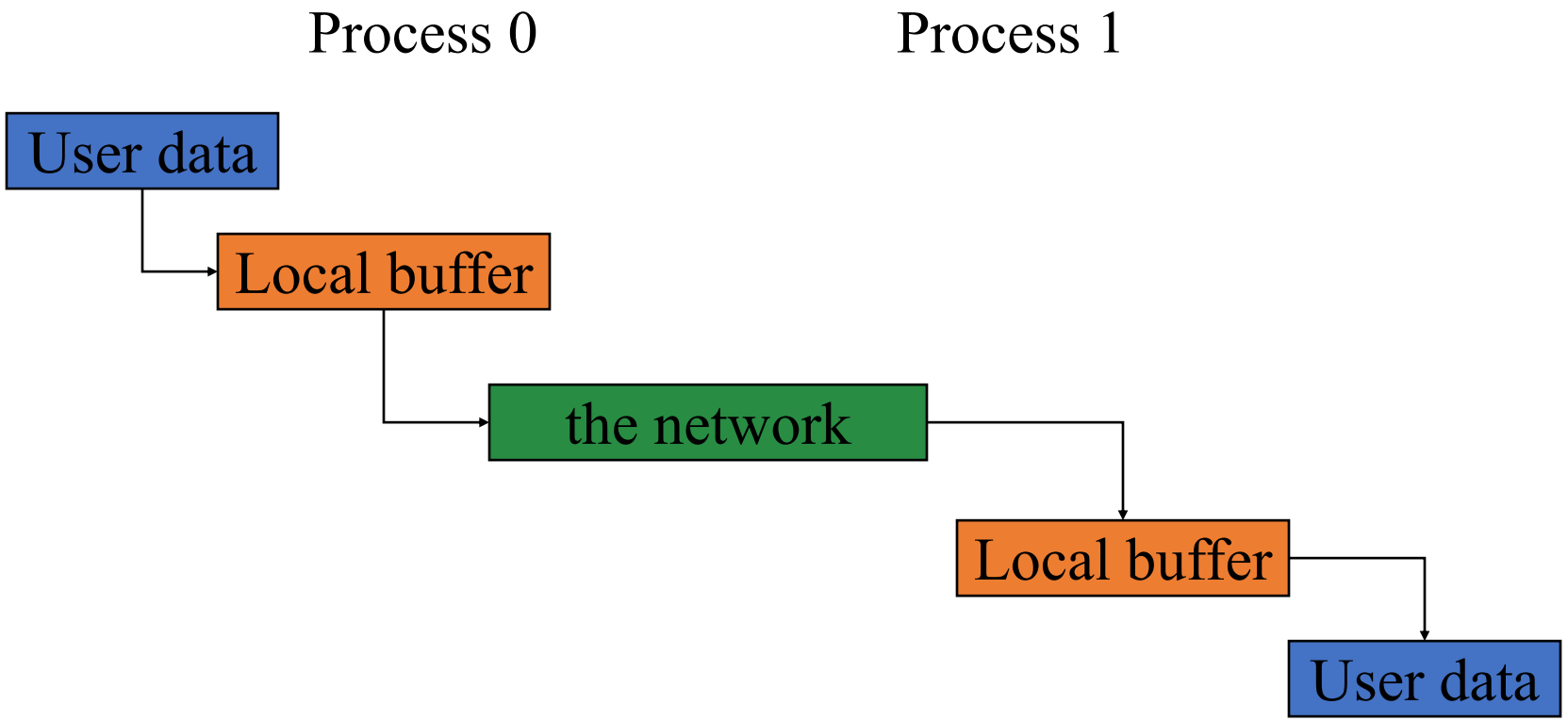
```
if (rank == 0) {
    MPI_Send(send_data, n, MPI_INT, 1, tag, MPI_COMM_WORLD);
    MPI_Recv(recv_data, n, MPI_INT, 1, tag, MPI_COMM_WORLD,
             MPI_IGNORE_STATUS);
}

if (rank == 1) {
    MPI_Send(send_data, n, MPI_INT, 0, tag, MPI_COMM_WORLD);
    MPI_Recv(recv_data, n, MPI_INT, 0, tag, MPI_COMM_WORLD,
             MPI_IGNORE_STATUS);
}
```

Deadlocks and Communication

- Example: mpi_deadlock.cpp

Deadlocks and Communication

- Why does the code not deadlock for small arrays?



Deadlocks and Communication

- This fix works!

```
if (rank == 0) {
    MPI_Send(send_data, n, MPI_INT, 1, tag, MPI_COMM_WORLD);
    MPI_Recv(recv_data, n, MPI_INT, 1, tag, MPI_COMM_WORLD,
             MPI_IGNORE_STATUS);
}
// Swap the order of the send and recv on this rank
if (rank == 1) {
    MPI_Recv(recv_data, n, MPI_INT, 0, tag, MPI_COMM_WORLD,
             MPI_IGNORE_STATUS);
    MPI_Send(send_data, n, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
```

Asynchronous Communication

- What if MPI_Send and MPI_Recv didn’t block?
- Can then overlap communication and computation
  - Better parallelism, less overhead and synchronization
  - Presumes that computation does not involve/change the data being communicated
- If we’re communicating data, we will probably do some computation with it.
  - How do we know when the communication finishes?

MPI (Asynchronous) Send

```
int MPI_Isend(
    const void* buffer,
    int count,
    MPI_Datatype data_type,
    int dest_rank,
    int tag,
    MPI_Comm comm,
    MPI_Request* request
);
```

Argument	Description
buffer	Pointer to the data to be sent
count	How many entries to send
data_type	Data type of the data to be sent
dest_rank	The rank to send to
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)
request	**MPI data structure for monitoring the send's status**

MPI (Asynchronous) Receive

```
int MPI_Irecv(
    void* buffer,
    int count,
    MPI_Datatype data_type,
    int src_rank,
    int tag,
    MPI_Comm comm,
    MPI_Status* status,
    MPI_Request* request
);
```

Argument	Description
buffer	Pointer to where to copy the data
count	**AT MOST** how many entries to read
data_type	Data type of the data to be sent
src_rank	The rank to receive from
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)
status	Information about the message
request	**MPI data structure for monitoring the send's status**

MPI_Isend/Irecv and Synchronization

- **MPI_Isend, MPI_Irecv:** these do not block
  - You can continue on; but watch out for the send buffer!
- When do we know when the recv buffer is filled?
- When can we use our buffers? How can we tell the communication finished? **The request object!**

Ensuring Completion

- The function MPI_Wait lets us wait for completion

```
MPI_Wait(MPI_Request* request, MPI_Status* status);
```

```
MPI_Request request = MPI_REQUEST_NULL;
MPI_Isend(send_data, n, MPI_INT, 1, tag, MPI_COMM_WORLD, &request);

do_work(other_data);

MPI_Wait(&request, MPI_IGNORE_STATUS);
refill_send_data(other_data, send_data);
```

What if we just want to check?

- The function MPI_Test lets us check for completion

```
MPI_Test(MPI_Request* request, int* flag, MPI_Status* status);
```

```
MPI_Request* request = new MPI_Request;
MPI_Isend(send_data, n, MPI_INT, 1, tag, MPI_COMM_WORLD, request);

int is_not_complete = 0;
MPI_Test(request, is_not_complete, MPI_IGNORE_STATUS);
while( is_not_complete ) {
    do_work(…);
    MPI_Wait(request, is_not_complete, MPI_IGNORE_STATUS);
}
do_stuff(send_data); // now that Isend has completed
```

What if we just want to check?

- The function MPI_Probe queries for MPI_Send info

```
MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status);
```

```
MPI_Status status;

if (rank==0){
      MPI_Send(xsend, xcount, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else {
      MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

      // determine “count” dynamically
      MPI_Get_count(&status, MPI_INT, &xcount);
}
```

MPI_Wait and MPI_Test variants

- Less commonly used, but can be useful

```
MPI_Waitall(count, array_of_requests,
  array_of_statuses)
MPI_Waitany(count, array_of_requests,
  &index, &status)
MPI_Waitsome(count, array_of_requests,
  array_of indices, array_of_statuses)
```

- Analogous commands for MPI_Test

Point-to-point vs collective MPI

- **All this so far is point-to-point communication**
  - i.e. single rank to single rank

- What about broadcasting?
  - One rank to all others (or vice versa)
  - “One-to-many” (or “many-to-one”)
- What about all-to-all situations?
  - Collectives and broadcasting
  - “Many-to-many”