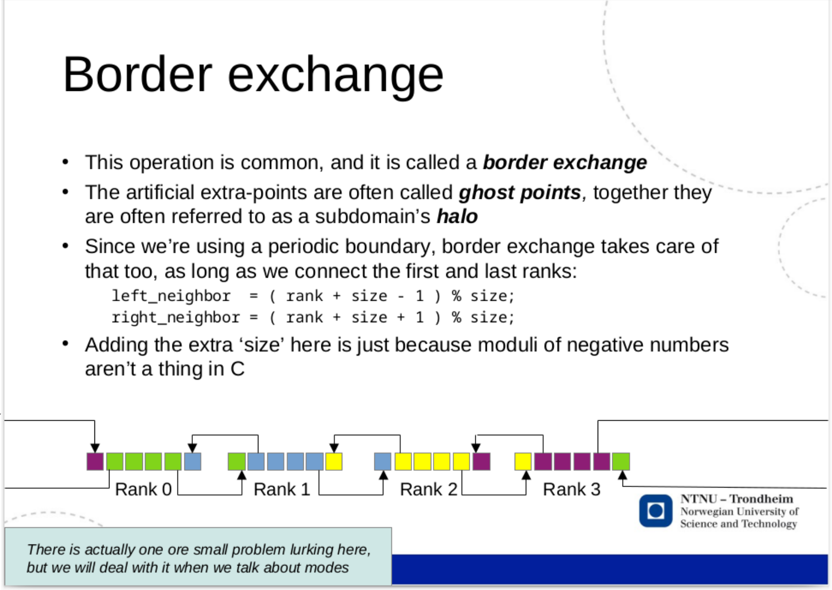
NTNU – Trondheim
Norwegian University of
Science and Technology

Point-to-point communication modes

(and other variants)

Jan.Christian.Meyer@ntnu.no

From last time



- As we noted previously
  - This send/recv pattern has an issue
  - The example code works anyway
- The issue is that the pattern is prone to *deadlock*
- Our program says
  - Send left
  - Receive from right
  - Send right
  - Receive from left
- If the first Send cannot return until anyone has received it, the program will stop at step 1
  - Everyone in a circle has a left neighbor

NTNU – Trondheim
Norwegian University of Science and Technology

The manual resolution

- We know how to fix this!
- If half the participants send first and the other half receive first, the cycle will be broken:

```
if ( rank % 2 == 0 )
    Send
    Recv
else
    Recv
    Send
```

- Mutual exchanges are common, though
  - It would be a hassle to pay attention to deadlocks every time

NTNU – Trondheim
Norwegian University of
Science and Technology

One MPI resolution

- Because mutual exchanges are so common, MPI has a very own function dedicated to combining one Send and one Recv:

```
int MPI_Sendrecv (
    const void *sbuf, int scount, MPI_Datatype stype, int dest,
    void *rbuf, int rcount, MPI_Datatype rtype, int source,
    MPI_Comm communicator, MPI_Status *status
);
```

- It is literally just the argument lists of the Send and Recv rolled into one
- Its entire purpose is that it comes with a guarantee that the pair will not deadlock

...but why does the example work?

- The ‘default’ MPI_Send is the one everyone is expected to use by default
  - It should be optimized for the common case
- Waiting for messages to be acknowledged takes time
- It is usually faster to
  - Copy the message into an MPI-internal buffer
  - Promise to send it as soon as possible, and return to caller
- Network interfaces usually feature little processors that can keep this promise without disturbing the CPU
  - Thus, everyone buffers their Sends, and start their Recvs
  - The example only sends 8 bytes in each direction

Internal buffering is not *always* faster

- When making the internal message copy takes longer than just sending the message right away, it's not efficient anymore
- The message size that crosses this limit is usually impossible to determine *exactly*, but
  - It's possible to estimate more-or-less reliably, and
  - MPI implementors often have a better opinion about it than MPI user programs
- Thus, beyond some "big enough" message size, MPI_Send switches to behaving like a blocking function
  - The exact size is implementation-dependent
  - It's just practically always much bigger than 8 bytes

Communication modes

- As we can see, the behaviour of MPI_Send can vary with what kind of protocol is guaranteed by the implementation
  - Even when the argument list is exactly the same
- MPI gives us four different variants, these *modes* come with different behaviours and assumptions:
  - *Standard* mode
  - *Synchronized* mode
  - *Buffered* mode
  - *Ready* mode

Standard mode

- This is the MPI_Send we already know
- Its arguments are the buffer-pointer, count, type, destination, tag, and communicator, as discussed previously
- It's at liberty to do whatever is "best on this machine", and return control as soon as it can
  - Provided that it's able to send the message correctly no matter what else the calling program gets up to
  - That is, the program must be able to modify the buffer contents without corrupting the copy that is being sent

Synchronized mode

- This one's called MPI_Ssend
  - The argument list is the same as for standard mode
- A synchronized send call will not return to the caller before the receiving process starts receiving
- It may be a little slower, but gives you a different kind of consistency between how far the communicating processes must have come when it returns
  - It synchronizes their progress
- If you try our mildly broken border exchange with MPI_Ssend instead of MPI_Send, it will always deadlock
  - Regardless of message size

Buffered mode

- If you want to send myriads of tiny messages at a time (without warning MPI first), it will cause myriads of tiny buffer allocations and deallocations
  - With enough messages, this takes time (and fragments heap memory)
- MPI_Bsend lets you allocate the buffer yourself, so that you can make it one long, contiguous memory range
  - As long as you make sure it has space for all the upcoming messages together

Allocating the buffer

- MPI_Bsend still has the same argument list as the other sends
- Since it expects us to have made a buffer for it, we have to register that before using Bsend:

```
int buffer_size = n*sizeof(msgsize) + MPI_BSEND_OVERHEAD;
int *my_buffer = malloc ( buffer_size );
MPI_Buffer_attach ( my_buffer, buffer_size );
```

- When we're finished with our Bsending, the registration can be released again:

```
MPI_Buffer_detach ( &my_buffer, &buffer_size );
```

Ready mode

- MPI_Rsend has the liberty to bypass protocols that establish whether or not the recipient is ready
- Its use indicates that the programmer is absolutely, 100% confident that the matching Recv call has already been made
- If the matching Recv call has *not* yet been made, it is an error to use Rsend, and its result is arbitrary

Receiving all this stuff

- There are no modal variants of MPI_Recv, it takes in messages from all of the above
  - As long as the arguments match
- The sender decides how to handle the transmission, because it’s the one who can do things differently depending on the acknowledgment from the other side

Non-blocking communication

- All the Send variants we've covered are *blocking*
  - Control doesn't return to the caller until transmission has been guaranteed
- There are also *non-blocking* variants of these calls
- This is not another mode
  - It doesn't affect the mechanics of data movement
- It's a natural thing to discuss along with the modes anyway

Non-blocking send

```
int MPI_Isend (
    const void *buffer,
    int count,
    MPI_Datatype type,
    int destination,
    int tag,
    MPI_Comm communicator,
    MPI_Request *request
);
```

- It's mostly the same as before, but it has an additional argument
  - It's an output, you hand some memory over to MPI, and it writes there

NTNU – Trondheim
Norwegian University of
Science and Technology

This returns immediately

- The idea of Isend is that it can whisk the message away into the background, and send it later at MPIs own convenience
- This leaves your program free to do something else in the meantime
- When you finally need to make sure that the transfer has completed, you wait for the request-thing to say that it's finished

  ```
  int MPI_Wait ( MPI_Request *req, MPI_Status *stat );
  ```

  – We can use MPI_STATUS_IGNORE here as well, if the status object is not needed for anything

We can have many of these

- If we make an entire array of MPI_Requests

```
MPI_Request my_reqs[42];
```

and attach them to different non-blocking sends,

```
for ( int m=0; m<42; m++ )
    MPI_Isend (
        &msgs[m], 1, MPI_INT, dst, 0, MPI_COMM_WORLD, &my_reqs[m]
    );
```

we can wait for them all at once:

```
MPI_Waitall ( 42, my_reqs, MPI_STATUSES_IGNORE );
```

  - MPI_STATUSES_IGNORE is like its singular counterpart, but it type-checks as an array of ignores instead of just 1

Communicate vs. compute

- Relatively speaking, communication calls are much more expensive than local operations
- A rule of thumb for performance programming goes:

    “Send early, receive late”
- The idea is that if you can compute a nice result while your messages are underway, you won’t waste CPU cycles while sitting around in the meantime
- Overlapping communication and computation is a popular application of MPI_Isend

Beware of false assumptions

- MPI_Isend does give your MPI implementation the opportunity to run communication in the background
- If you measure it, you will also find that most of them actually do, at least up until some critical message size
- However, **they are not obliged to**
  - The MPI standard doesn't actually require anything at all to happen until you issue the wait call on the request
- Non-blocking sends were originally introduced as yet another way to prevent deadlock in mutual exchanges
  - You can use them for overlapping, but take care to measure that it works on the machine you are using

Non-blocking comms and modes

- We have a full range of non-blocking counterparts to everything we've talked about:
  - MPI_Isend
  - MPI_Issend
  - MPI_Ibsend
  - MPI_Irsend
  - MPI_Irecv
- There are even non-blocking collectives
  - They were only introduced in MPI 3.0, though, so you won't see them in a lot of production code yet

Persistent communication

- The MPI_Request-objects of Isend also have another application
- If you're going to use the same communication pattern over and over

  (*e.g.* running neighbor exchanges every iteration)

  you can let MPI prepare them once and for all, and just call on them every time you want to activate them
  - It saves a little bit of time with setting up the transmission
  - It saves a bit of code complexity in the middle of a loop that you're probably filling up with other complicated expressions

Persistent sending and receiving

All our sending and receiving calls can be initialized like this:

int MPI_Send_init (<all the usual stuff>, MPI_Request *req);

int MPI_Recv_init (<all the usual stuff>, MPI_Request *req);

triggered like this:

int MPI_Start (MPI_request *req);

(there is also an MPI_Startall that takes a count and an array of requests)

and waited for if they're non-blocking, as before.