

Message Passing Programming

Modes, Tags and Communicators

Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<http://creativecommons.org/licenses/by-nc-sa/4.0/>

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Acknowledge EPCC as follows: “© EPCC, The University of Edinburgh, www.epcc.ed.ac.uk”

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Overview

- Lecture will cover
 - explanation of MPI modes (**Ssend**, **Bsend** and **Send**)
 - meaning and use of message tags
 - rationale for MPI communicators
- These are all commonly misunderstood
 - essential for all programmers to understand modes
 - often useful to use tags
 - certain cases benefit from exploiting different communicators

Modes

- **MPI_Ssend** (Synchronous Send)
 - guaranteed to be synchronous
 - routine will not return until message has been delivered
- **MPI_Bsend** (Buffered Send)
 - guaranteed to be asynchronous
 - routine returns before the message is delivered
 - system copies data into a buffer and sends it later on
- **MPI_Send** (standard Send)
 - may be implemented as synchronous or asynchronous send
 - this causes a lot of confusion (see later)

MPI_Ssend

Process A



`Ssend(x, B)`

Wait in Ssend



Ssend returns

`x` can be
overwritten by A



Process B



Running other
non-MPI code

`Recv(y, A)`

Data Transfer

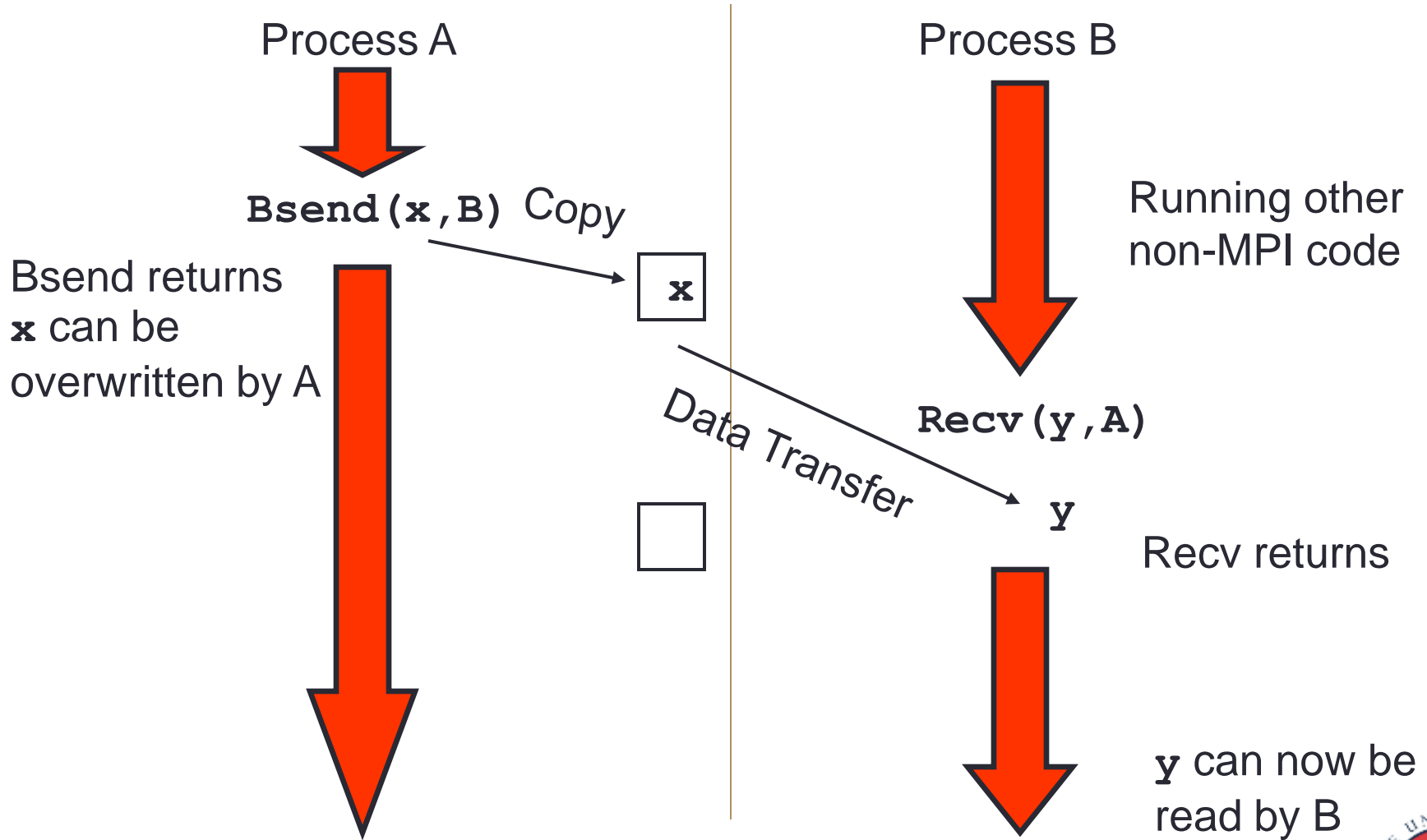
`x` → `y`

Recv returns

`y` can now be
read by B



MPI_Bsend



Notes

- **Recv** is always synchronous
 - if process B issued **Recv** before the **Bsend** from process A, then B would wait in the **Recv** until **Bsend** was issued
- Where does the buffer space come from?
 - for **Bsend**, the user provides a single large block of memory
 - make this available to MPI using **MPI_Buffer_attach**
- If A issues another **Bsend** before the **Recv**
 - system tries to store message in free space in the buffer
 - if there is not enough space then **Bsend** will FAIL!

Send

- Problems
 - **Ssend** runs the risk of deadlock
 - **Bsend** less likely to deadlock, and your code may run faster, but
 - the user must supply the buffer space
 - the routine will FAIL if this buffering is exhausted
- **MPI_Send** tries to solve these problems
 - buffer space is provided by the system
 - **Send** will normally be asynchronous (like **Bsend**)
 - if buffer is full, **Send** becomes synchronous (like **Ssend**)
- **MPI_Send** routine is unlikely to fail
 - but could cause your program to deadlock if buffering runs out

MPI_Send



- This code is NOT guaranteed to work
 - will deadlock if Send is synchronous
 - is guaranteed to deadlock if you use **Ssend**!

Solutions

- To avoid deadlock
 - either match sends and receives explicitly
 - e.g. for ping-pong
 - process A sends then receives
 - process B receives then sends
- For a more general solution use non-blocking communications (see later)
- For this course you should program with **Ssend**
 - more likely to pick up bugs such as deadlock than **Send**

Checking for Messages

- MPI allows you to check if any messages have arrived
 - you can “probe” for matching messages
 - same syntax as receive except no receive buffer specified
- e.g. in C:

```
int MPI_Probe(int source, int tag,  
              MPI_Comm comm, MPI_Status *status)
```
- Status is set as if the receive took place
 - e.g. you can find out the size of the message and allocate space prior to receive
- Be careful with wildcards
 - you can use, e.g., **MPI_ANY_SOURCE** in call to probe
 - but must use **specific** source in receive to guarantee matching same message
 - e.g. `MPI_Recv(buff, count, datatype, status.MPI_SOURCE, ...)`

Tags

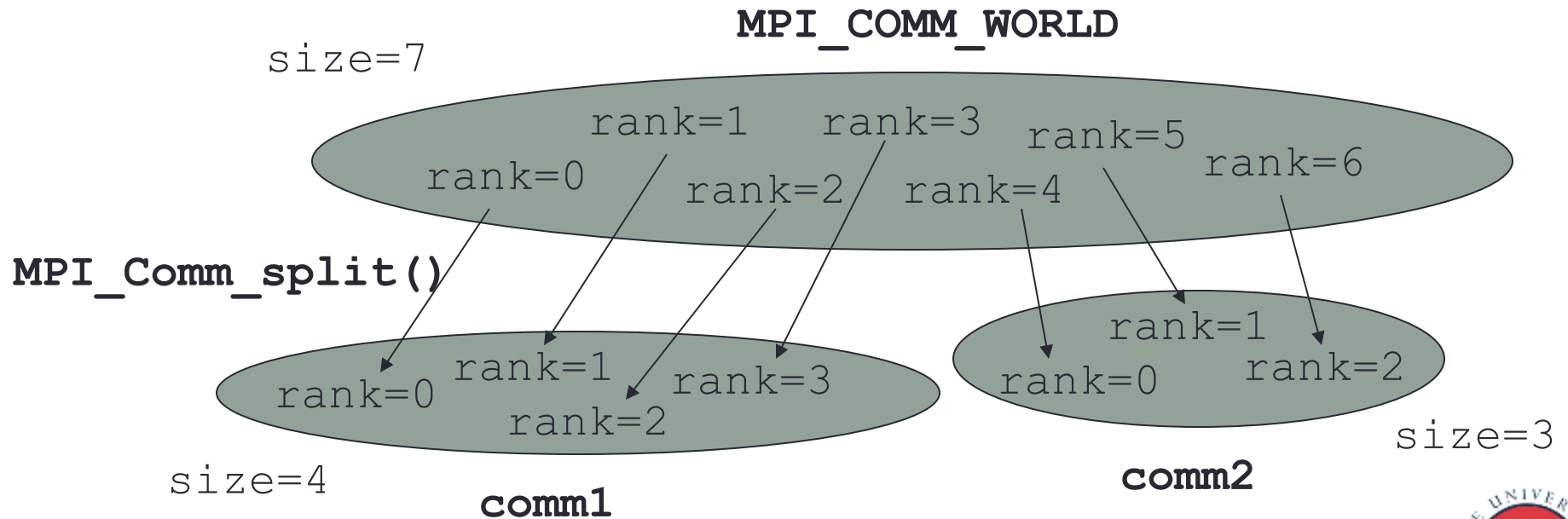
- Every message can have a tag
 - this is a non-negative integer value
 - maximum value can be queried using **MPI_TAG_UB** attribute
 - MPI guarantees to support tags of at least 32767
 - not everyone uses them; many MPI programs set all tags to zero
- Tags can be useful in some situations
 - can choose to receive messages only of a given tag
- Most commonly used with **MPI_ANY_TAG**
 - receives the most recent message regardless of the tag
 - user then finds out the actual value by looking at the **status**

Communicators

- All MPI communications take place within a communicator
 - a communicator is fundamentally a group of processes
 - there is a pre-defined communicator: **MPI_COMM_WORLD** which contains ALL the processes
 - also **MPI_COMM_SELF** which contains only one process
- A message can ONLY be received within the same communicator from which it was sent
 - unlike tags, it is not possible to wildcard on **comm**

Uses of Communicators (i)

- Can split **MPI_COMM_WORLD** into pieces
 - each process has a new rank within each sub-communicator
 - guarantees messages from the different pieces do not interact
 - can attempt to do this using tags but there are no guarantees



Uses of Communicators (ii)

- Can make a copy of `MPI_COMM_WORLD`
 - e.g. call the `MPI_Comm_dup` routine
 - containing all the same processes but in a new communicator
- Enables processes to communicate with each other safely within a piece of code
 - guaranteed that messages cannot be received by other code
 - this is **essential** for people writing parallel libraries (e.g. a Fast Fourier Transform) to stop library messages becoming mixed up with user messages
 - user cannot intercept the the library messages if the library keeps the identity of the new communicator a secret
 - not safe to simply try and reserve tag values due to wildcarding

Summary (i)

- Question: Why bother with all these send modes?
- Answer
 - it is a little complicated, but you should make sure you understand
 - **Ssend** and **Bsend** are clear
 - map directly onto synchronous and asynchronous sends
 - **Send** can be either synchronous or asynchronous
 - MPI is trying to be helpful here, giving you the benefits of **Bsend** if there is sufficient system memory available, but not failing completely if buffer space runs out
 - in practice this leads to endless confusion!
- The amount of system buffer space is variable
 - programs that run on one machine may deadlock on another
 - you should **NEVER** assume that **Send** is asynchronous!

Summary (ii)

- Question: What are the tags for?
- Answer
 - if you don't need them don't use them!
 - perfectly acceptable to set all tags to zero
 - can be useful for debugging
 - e.g. always tag messages with the rank of the sender

Summary (iii)

- Question: Can I just use `MPI_COMM_WORLD`?
- Answer
 - yes: many people never need to create new communicators in their MPI programs
 - however, it is probably bad practice to specify `MPI_COMM_WORLD` explicitly in your routines
 - using a variable will allow for greater flexibility later on, e.g.:

```
MPI_Comm comm;           /* or INTEGER for Fortran */
comm = MPI_COMM_WORLD;
...
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
....
```

Behaviour of MPI_Send

- The most misunderstood aspect of MPI
 - even among experienced MPI programmers
- **You must try and understand what is happening and why**
- Many codes mistakenly assume that MPI_Send is guaranteed to be asynchronous (i.e. buffered)
 - these programs are **incorrect** even if they happen to run correctly

Conclusion

- **MPI_Send** can be synchronous or asynchronous
 - allows MPI to choose the optimal method, but ...
 - varies depending on message size, number of MPI processes, library, ...
 - can cause incorrect codes (which deadlock if synchronous) to run OK
 - your code may run on your laptop or the login node but not on Cirrus compute nodes, or run on a single node but not multiple nodes, ...
- Solution
 - develop with synchronous send **MPI_Ssend** (or **MPI_Issend**)
 - ensures your code is correct
 - do production runs using **MPI_Send** (or **MPI_Isend**)
 - your code may be faster
 - a code that runs correctly with **MPI_Ssend** is extremely unlikely to be incorrect using **MPI_Send**