7. Features of MPI

Parallel Programming

Dr Hamidreza Khaleghzadeh
School of Computing
University of Portsmouth

Goals

- This lecture fills in many of the details of the MPI programming interface.
- Concentrates on what is called *point-to-point communication*, which is the foundation of MPI.
- Will touch on the details of the standard mode send and receive operations, other modes of send, immediate communications and their completions, and process groups and communicators.

MORE ABOUT RECV

Recv Reprise

- As we saw in an earlier lecture, the basic method for *receiving* a message in MPJ is:

  Recv(buffer, offset, count, type, src, tag)

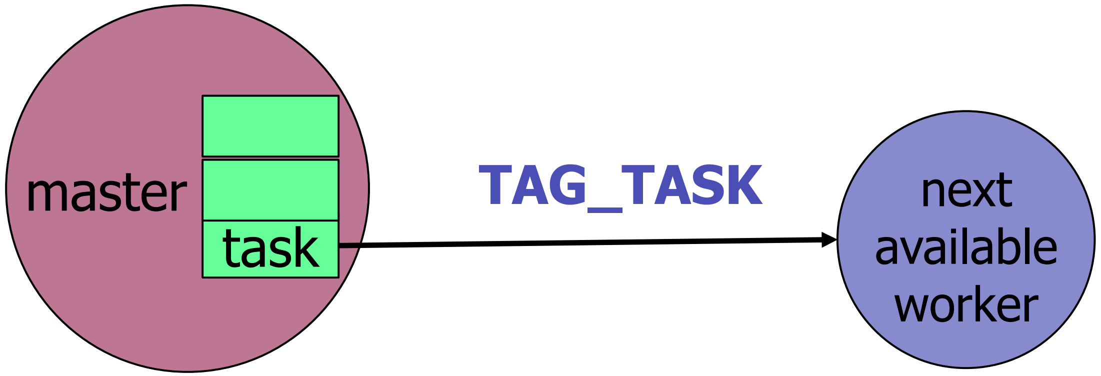
- Here the arguments buffer, offset, count and type describes the data elements of the message, and where it is to be stored.
- src is the rank of the process we expect the message to come *from*.
- tag is a user-defined integer value sent in the message to describe its "*purpose*".

Example Scenario for Tags

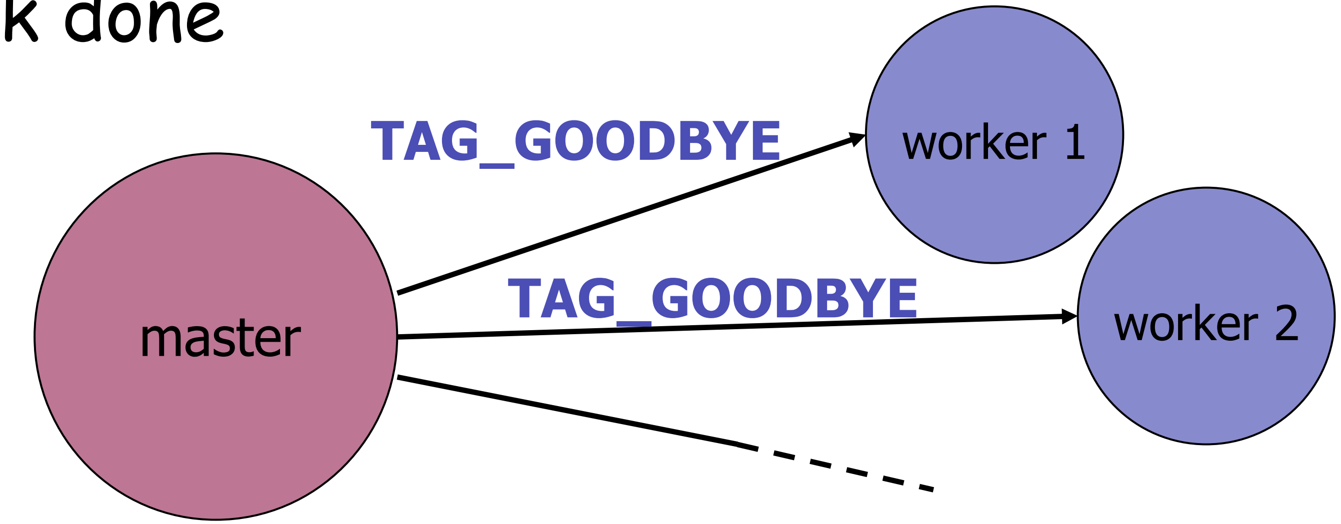
- One process, a *master*, is sending tasks to other *worker* processes.
- A message containing a task has a tag value TAG_TASK, identifying it as containing work.
- When all tasks are done, master sends a last message with tag value TAG_GOODBYE to all workers, telling them they don't need to wait for more work and can now shut down.

Example Scenario for Tags

- Work remains



- All work done



Wildcards

- The src argument can take the special value MPI.ANY_SRC, in which case the Recv accepts a message with matching tag from *any source*.
- Similarly, the tag argument can take the special value MPI.ANY_TAG, in which case the Recv accepts a message with specified source from *any tag*.
- Can combine both these wildcards to accept *any* incoming message (on the current *communicator* – see later).

The Status object

- The Recv call actually returns an object of type Status, which previously we have ignored, e.g.:

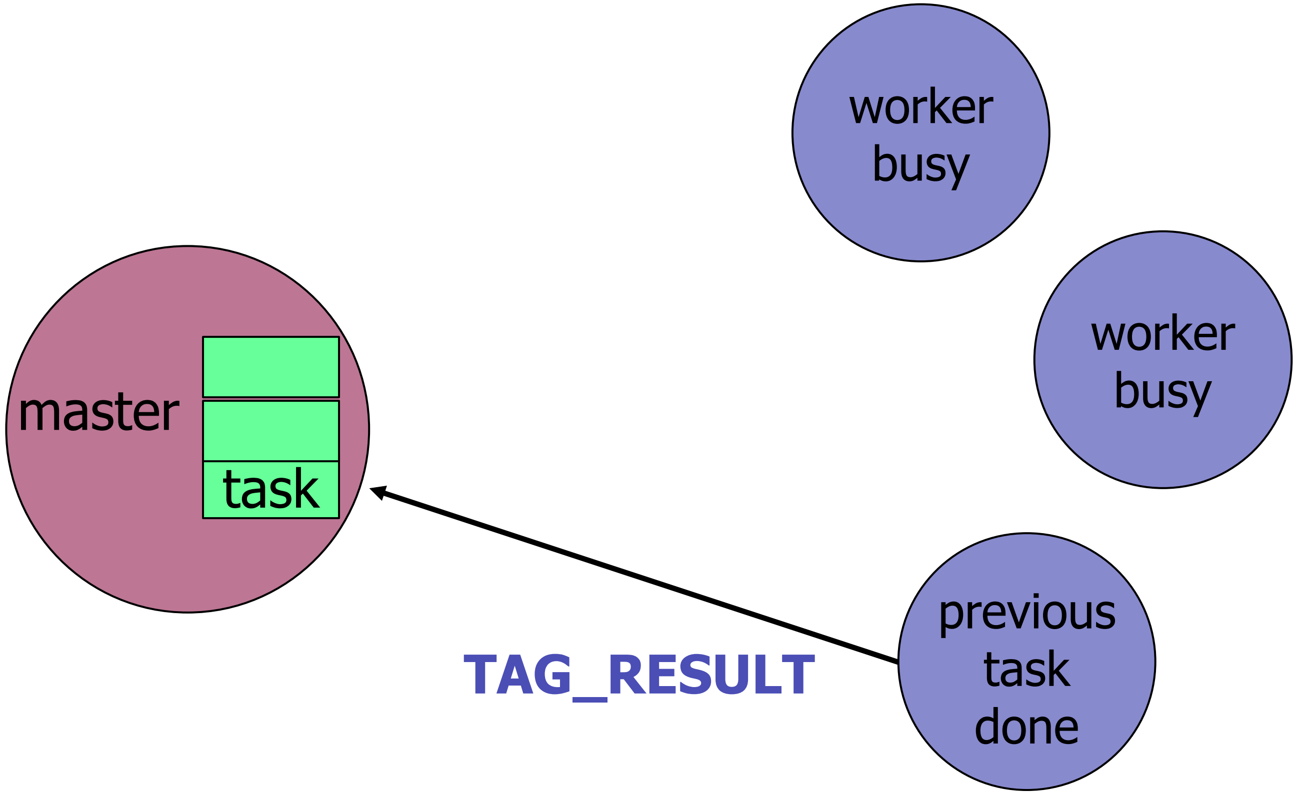
  Status stat = **Recv**(buffer, offset, count, type, src, tag) ;

  - The stat object here contains fields source and tag which specify the source and tag of the message (useful where wildcards are used).
  - It also has methods which can be used to determine how much data was received (useful if the actual message contained fewer than count elements – see later).

Scenario for MPI.ANY_SOURCE

- In earlier scenario, master knows a worker is available do *more* work when worker sends a message containing result of its previous task.
  - But master doesn't know in advance *which* worker will become available next.
- It receives requests for work using MPI.ANY_SOURCE, then determines which worker they came from using status.source.
  - It may then send a new task to identified worker.

Scenario for MPI.ANY_SOURCE



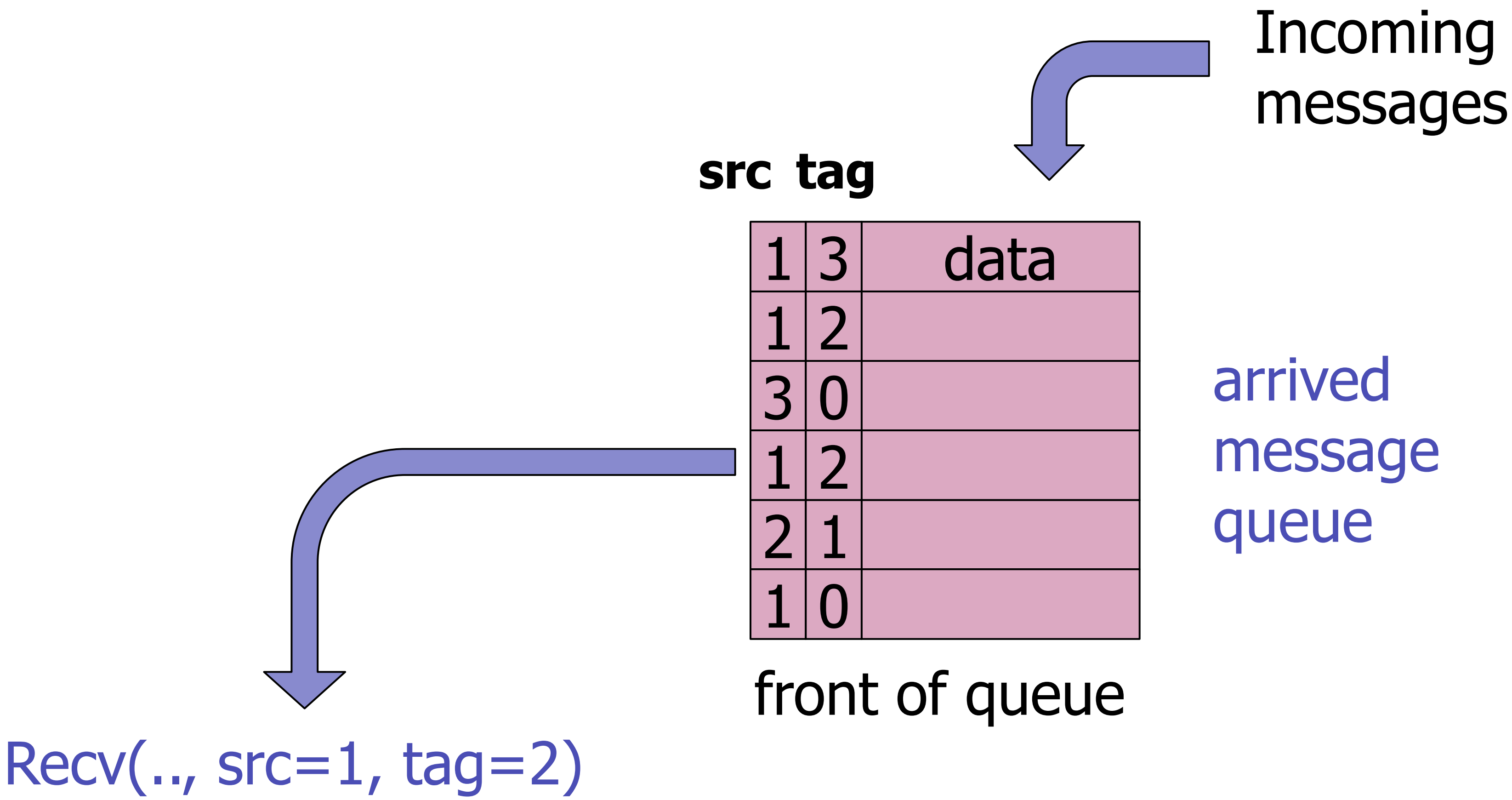
Scenario for MPI.ANY_TAG

- Worker process in earlier scenario receives a series of messages from master with MPI.ANY_TAG.
  - If status.tag from Recv is TAG_TASK, do work on contents of message.
  - If status.tag is TAG_GOODBYE, end our main loop, thus stop waiting for any more tasks, and shut down.

Behaviour of **Recv**

- When we call Recv, the MPI system first looks for any message with matching the src and tag values that have *already arrived* at this node.
- If there are any, it immediately copies the data from the matching message that arrived *first* to buffer, then Recv completes.
  - Errors may occur if the amount or type of data in the that message disagrees with the Recv arguments.
- Otherwise the Recv call *blocks* until a matching message arrives from src.
  - Arriving messages with different tags left in queue.

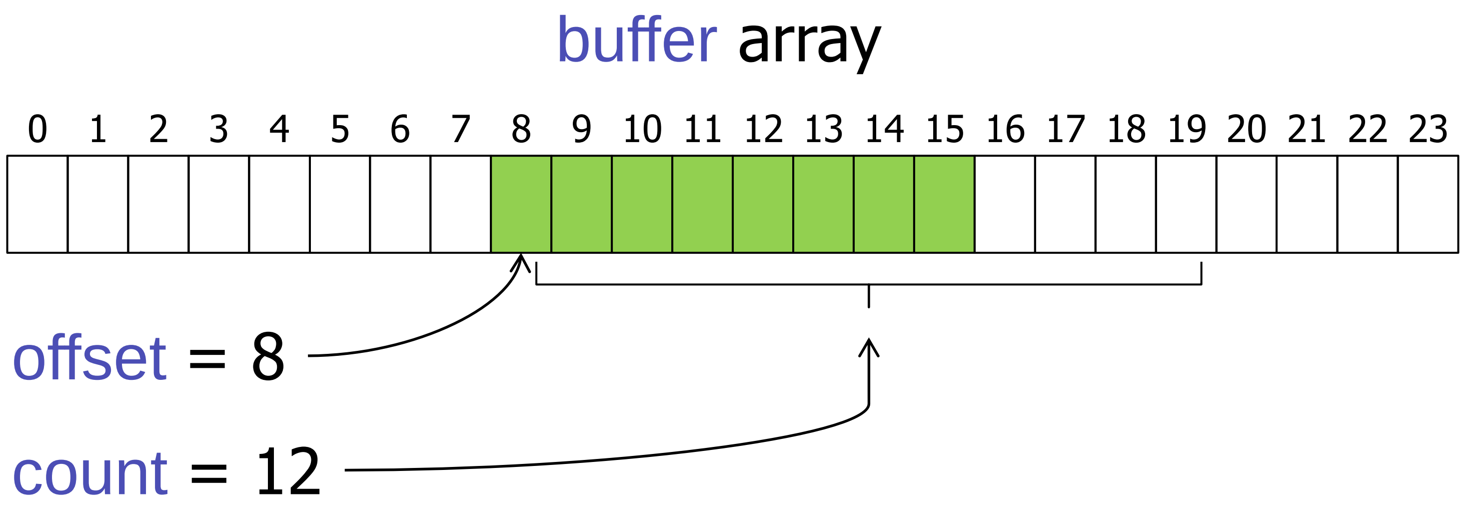
Destination Processor



Ordering of **Recv** completions

- MPI *delivers* messages from a given source to a given destination in the same order that Send was called at the source processes.
- But if Recv is posted with a different tag, messages may effectively “overtake” – an earlier message may be ignored until another Recv is posted with a matching tag.

Buffer Example



- Elements of buffer actually written are in green
  - Example assumes *actual message sent* contained *fewer elements* than the *maximum* specified by count in the Recv call.
  - Again, query status to find actual message size.

Synopsis

- The Recv operation generally blocks until a matching send has been posted.
- Messages are received in the same order they are sent – except where the requirements of matching source and tag cause messages to be queued.
- The Recv method returns a Status object that can be queried to find out where the message came from, the tag it was sent with, and how much data it contained.

Illustrating Features of Recv

- Several of the new features of Recv described above will be exercised in the *task farm* example in this week lab script.

MORE ABOUT SEND

Send Reprise

- The basic method for *sending* a message in MPJ is:

  Send(buffer, offset, count, type, dest, tag)

- This is what is called a *standard mode send* to the process with rank dest.

Blocking of Sends

- As we saw earlier, a Recv call will *block* if the matching message has not yet been sent.
- This seems natural for receive operations, but in general *send operations may also block* if the corresponding receive operation has not yet been initiated ("posted") by calling Recv at the destination process.
  - Local MPI system may not have enough memory available to *buffer* the sent message internally.

Standard Mode Send

- The *MPI standard* says that the basic "standard mode Send" *may or may not block*:
- It is up to the implementer of MPI.
  - In many implementations of MPI, *short* messages will be sent immediately, while *longer* messages will wait until Recv is posted at destination, leading to blocking.
  - In other implementations, *all* calls to Send block until Recv is posted.
- To avoid unexpected deadlocks, *all MPI programs should be written with the "pessimistic" assumption that calls to* Send *block*.
  - It is essentially unpredictable whether they will or not.

Buffered Mode Send

- Another method in MPJ for sending a message is:

  Bsend(buffer, offset, count, type, dest, tag)

- This is what is called a *buffered mode send*.
- The idea is that send operations using this mode should *not* block.
- *But* it depends on the programmer explicitly allocating enough memory to implement local buffering using MPI.Buffer_attach.
  - In my opinion this is *not* an attractive option.

Other Modes

- *Synchronous mode send*, Ssend(), *always* blocks until the matching receive has been posted
  - May be useful because potential deadlocks will show up early in development. But likely to have a performance overhead for short messages.
- *Ready mode send*, Rsend(), may only be used if the program logic *guarantees* that the matching receive has been posted *ahead of time*
  - Potential performance gain when such a guarantee can be established, but strictly for advanced MPI programmers!

Synopsis

- MPI defines several *modes* of message send.
- General advice is to stick to standard mode send but be aware of its *blocking* behaviour, which may lead to deadlocks and loss of potential concurrency.
- To resolve these issues, the best solution is often to use *immediate* communications.

IMMEDIATE COMMUNICATIONS

“Non-blocking” Communication

- MPI has a mechanism for separating the *initiation* of communications from the stage of *waiting for their completion*.
- This is often referred to as *non-blocking communication*, because the *initiation* operation never blocks a process.
  - Slightly misleading, because all communications *must* also be *completed*, and that stage *may or may not block* (depending amongst other things on the mode).
- Alternate name: *immediate communication* – initiation completes “immediately”.

Non-Blocking Send Example

```
Request req = MPI.COMM_WORLD.Isend(buffer, offset, count,
                                   type, dest, tag) ;
... do other business

req.Wait() ;
```

- Immediate communication methods like Isend() return *immediately* with a Request object.
- To wait for completion, execute the Wait() method on that object.
- Effect of Isend/Wait above identical Send, but can do other things (...) in between initiation and waiting.

Example - implementing Sendrecv

- In the last lecture saw this useful but unwieldy method:

```
MPI.COMM_WORLD.Sendrecv(sbuffer, soffset, scount, stype, dest, stag,
                        rbuffer, roffset, rcount, rtype, src, rtag) ;
```

- Equivalent thing using immediate methods:

```
Request sreq = MPI.COMM_WORLD.Isend(sbuffer, soffset, scount, stype, dest, stag) ;
Request rreq = MPI.COMM_WORLD.Irecv(rbuffer, roffset, rcount, rtype, src, rtag) ;
rreq.Wait();
sreq.Wait();
```

The Wait() method

- *Must* call Wait() (or equivalent - see later) on any request to guarantee completion of communication.
  - After Wait() completes on a request object, object is *inactive* and cannot be used again.
- Wait() has a return value which is a Status object.
  - If the communication was an Irecv(), contains same information as Status returned by blocking Recv().
  - Can usually ignore return value if Request was created by e.g. an Isend().
- Don't confuse it with the very different wait() method (lower case) from java.lang.Object!

Waitany()

- Most useful of several other methods for waiting for completion of immediate communications.
- This one is a *static* method in the Request class, and takes as argument an *array* of Request objects.
- Waits for completion of *one* communication in the request array – so for example you can wait on several possible receives until the *first* arrives, then act on that before dealing with other communications, later.

Synopsis

- *Immediate* communication methods separate the *initiation* of a communication from the stage of *waiting for completion* of the communication.
- This can be used to avoid over-synchronization problems like *deadlock*, and also allows useful work to be performed while communications are in progress.
- In general it allows communications to be handled in a more asynchronous and responsive manner, should this be required.

COMMUNICATORS AND PROCESS GROUPS

Communicators

- All the basic communication operations like Send() and Recv() are methods of the *communicator* class, Comm.
- In all the examples so far we used one particular communicator – the *default* or *world* communicator:

  MPI.COMM_WORLD

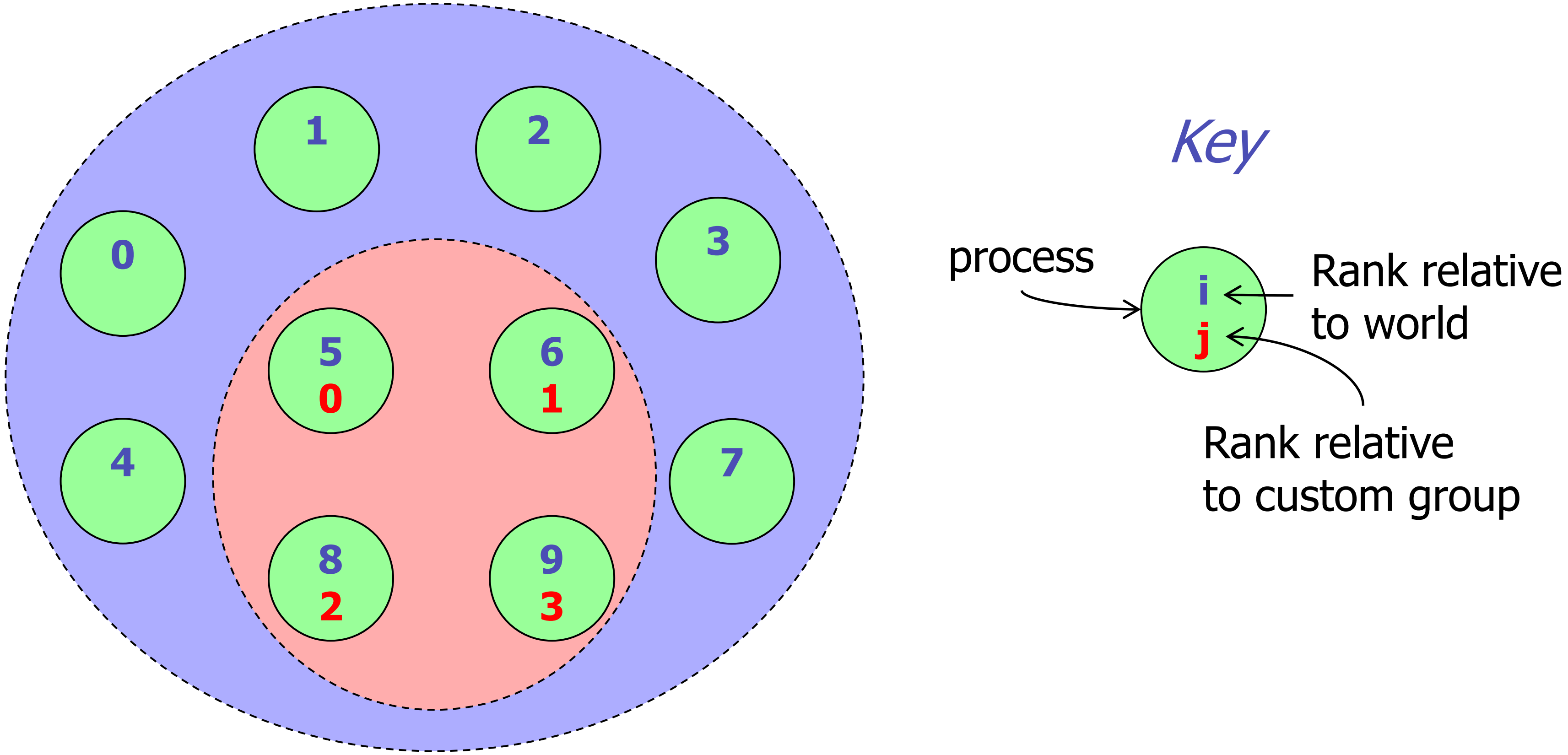
  - Note COMM_WORLD is a constant of type Comm in the MPI class.

Other Communicators

- Although most programs in practise seem to use MPI.COMM_WORLD, there are good reasons why other communicators may be required, and MPI provides a fairly elaborate API for creating new communicators.
- One reason for creating a new communicator is if you want to partition or otherwise divide the complete set of available processors into smaller groups working independently.

Groups Example

- Custom group of 4 processes in MPI world of 10 processes



Group Class

- A separate Group class describes groups of processes, and can be set up to describe a quite general subset of the "world".
- Other examples of subgroups might be if the world was viewed as a *2 dimensional grid* of processes – subgroups might be 1d rows or columns of this grid.

Comm Class

- Every *communicator*, class Comm, spans a particular Group of processes, and provides the resources needed to implement communication between processes within that group.
- Can have multiple communicators spanning the same group.
- Supports independent libraries of *collective operations*, where each library can have an independent *communication context*, reducing danger of interference between 3rd party libraries.

A Role for Groups

- One role for groups - and communicators spanning them - is in *collective communications where only a subset of processes need to involved*.
  - See main week 6 lecture
- e.g. if we want to broadcast to just a subset of processes (sometimes called a *multicast*), may create a Group and Intracomm spanning that subset, and call Bcast on that.

Synopsis

- It is relatively unlikely you will need to use communicators other than MPI.COMM_WORLD.
- But communicators do seem like a natural idea in distributed memory SPMD programming.
  - Collective generalization of the *channel* idea.
- MPI system of groups and communicators is perhaps slightly heavyweight, but does attempt to provide a basis for communication abstractions that can be embodied in reusable libraries of parallel code.

Summary

- Although we have been selective, we have at least touched on many of the important parts of the MPI interface.

Further Reading

- MPJ Express API:

  http://mpj-express.org/docs/javadocs/index.html

- William Gropp, Ewing Lusk and Anthony Skjellum, *Using MPI*, 2nd Edition MIT Press, 1999.
  - Standard text on MPI, but examples are in C and Fortran.
  - Available as an electronic book through the library.