THE UNIVERSITY of York

High Performance Computing - MPP Programming with MPI - part II

Prof Matt Probert

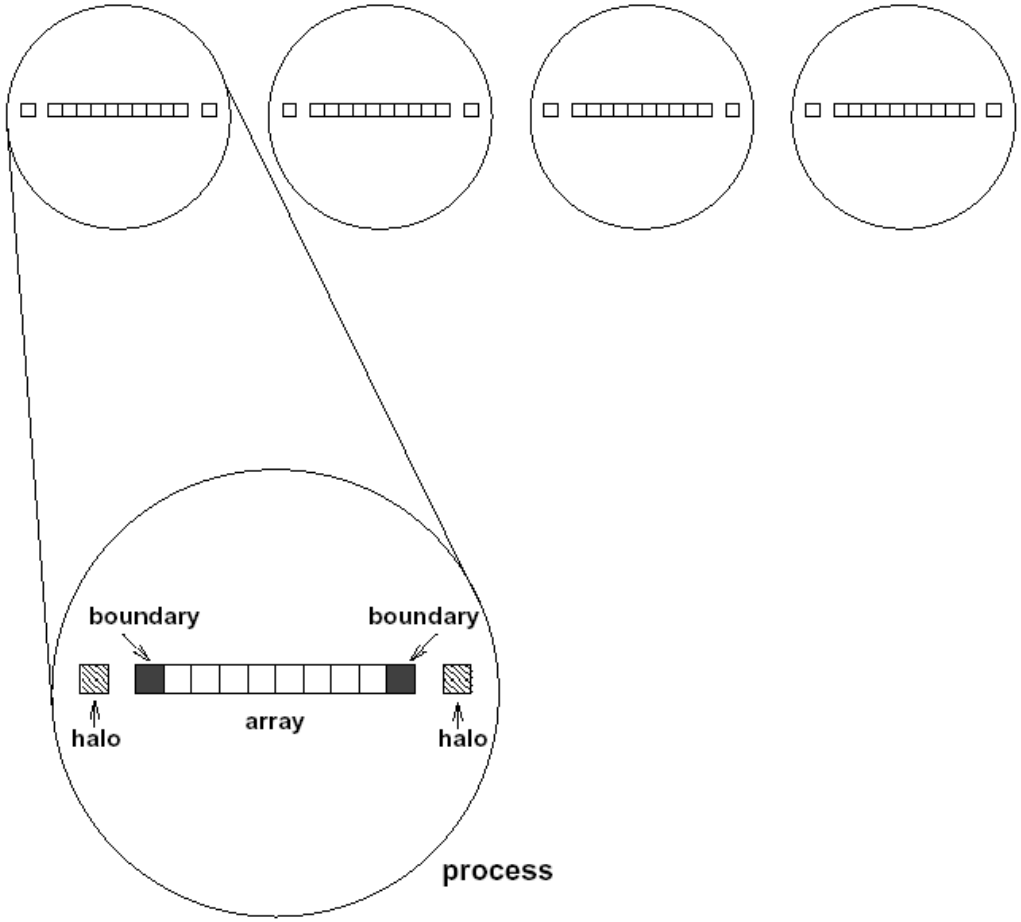
http://www-users.york.ac.uk/~mijp1

Overview

- More Point-to-Point Communication
  - Non-blocking version
- Advanced Collective Communication
- Manipulating Communicators
- Miscellaneous MPI Features

Simple Example

- Consider a regular domain decomposition with periodic boundary conditions :



Want to replace each element by the average of its two neighbours.

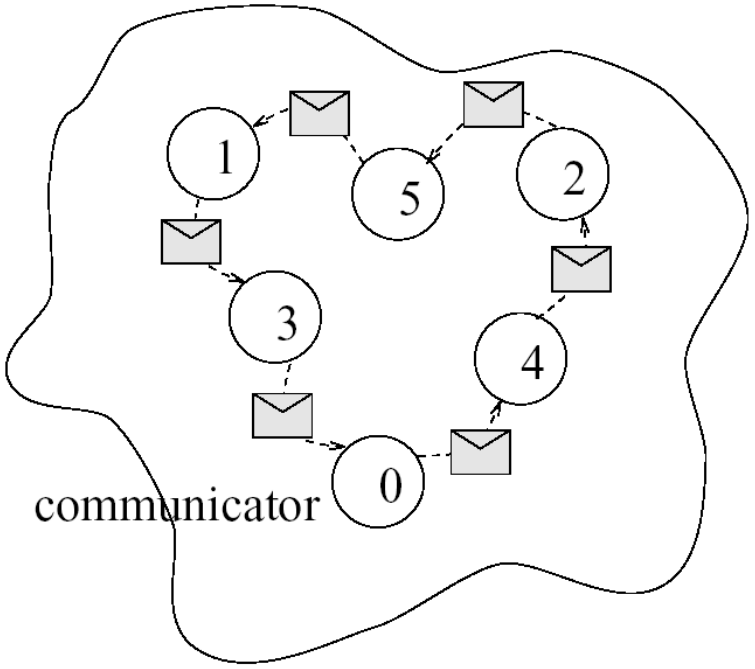
Data is distributed as shown so need halo data to perform smoothing at edges of local data on each node.

Interior data requires no comms for update.

Deadlock Potential

- Simple-minded implementation of 1D parallel data smoothing might be:

```
For each iteration do:
   update all cells
   send boundary data to neighbours
   receive halo data from neighbours
```



But this has potential for deadlock: Using a standard send, it *may* be that the send cannot complete until the receive has started – yet as all nodes are sending, none can start receiving, and hence get deadlock.

Possible Solutions

- There are various possible solutions to the previous problem:
  - Use buffered send
  - Use “red-black” pattern, i.e. every “red” node sends whilst every “black” node receives and then switch over.
  - But both of these solutions have a drawback – system has to idle whilst comms take place.
  - A better solution would allow *latency hiding*, where calculations can proceed whilst comms are in progress.
- Hence need for non-blocking (asynchronous) comms, where a send can complete regardless of state of receive (c.f. sending a letter by mail).

Non-Blocking Solution

- With non-blocking comms, need to use additional MPI commands to test for state of communication:

```
For each iteration do (on all nodes):
   update boundary cells
   initiate sending of boundary values to neighbours
   initiate receipt of halo data from neighbours
   update non-boundary cells
   wait for completion of sending of boundary values
   wait for completion of receipt of halo values
```

NB Cannot get deadlock with this solution and comms can be fully bi-directional.

NB Completion tests only posted when data is needed, hence can hide comms costs behind other updates.

Non-Blocking MPI Communications

Mode	Non-Blocking
Standard	`MPI_Isend`
Buffered	`MPI_Ibsend`
Synchronous	`MPI_Issend`
Receive	`MPI_Irecv`

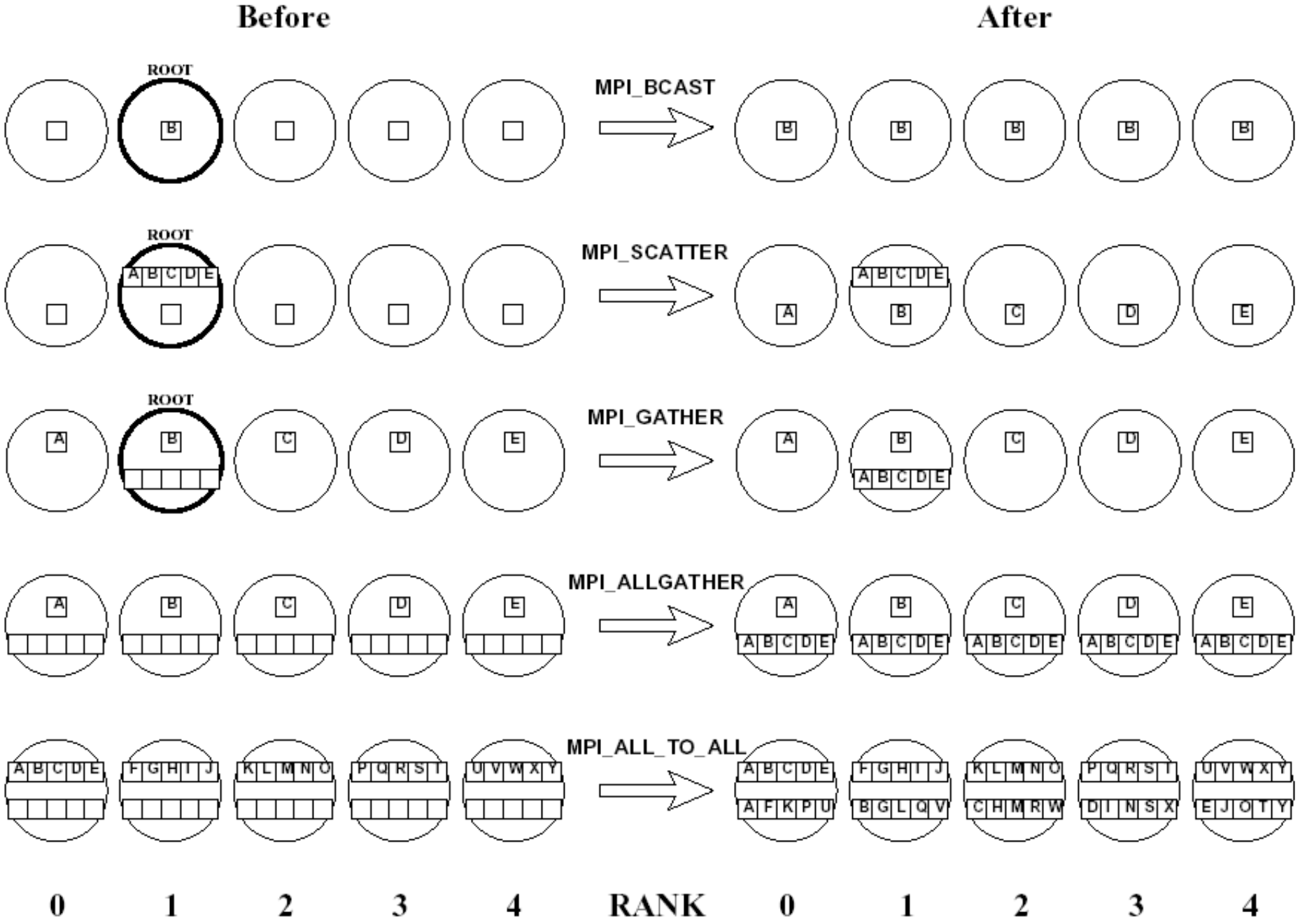
Non-blocking forms have similar args to blocking forms but with an additional unique **`request`** handle which use to test completion state.

- `MPI_Issend(data, count, datatype, destination, tag, comm, request, ierror)`
  - Can then proceed with calculations that do not change the send `data`, until `Issend` is complete.
- `MPI_Irecv(data, count, datatype, source, tag, comm, request, ierror)`
  - This posts the receive and then need to explicitly check for completion before can use the received `data`.

Testing for Completion

- Every non-blocking comm *must* have a matching test
  - Must not modify (send) or use (recv) data until test OK
  - Can choose to delay the test until data needed
  - Missing out a test will lead to a resource leak …
- Simplest to use a blocking test on a single communication
  - `MPI_Wait(request, status, ierror)`
  - where `request` is the integer handle to the required non-blocking communication and `status` is a user-defined integer array (of `MPI_STATUS_SIZE`) which holds information about message, as discussed for `MPI_Recv` in earlier lecture.
- Can also do non-blocking `MPI_Test` and also multi-message test routines available – complex!
- NB Syntax given for Fortran – all C/C++ versions the same except case-sensitive names, use `&data`, `&status` etc. and no final `ierror` as called as function not subroutine

Advanced Collective Communications



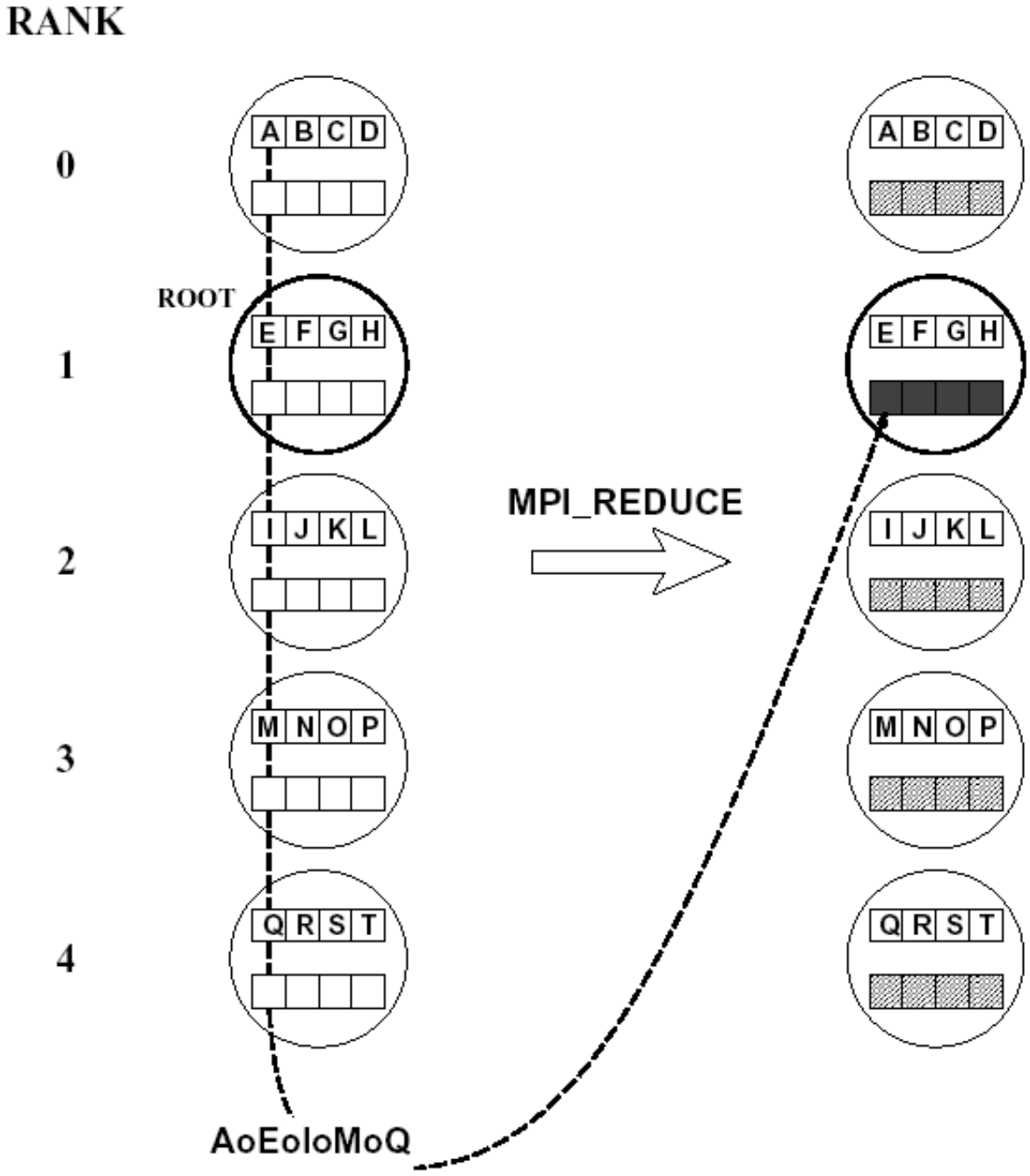
More Collective Communications

- `MPI_Bcast(data, count, datatype, root, comm, ierror)`
  - As last lecture - broadcasts `count` items of `data` from `root` process to all process in specified `comm`unicator.
- `MPI_Scatter(send_data, send_count, send_type, recv_data, recv_count, recv_type, root, comm, ierror)`
  - NB `send_count` is number of items sent to each processor not total. `Send_*` items only relevant on `root` process. All processes in `comm` must participate.
  - `Send_type` *may* be different to `recv_type` but if it is the same then `send_count` must equal `recv_count`
- `MPI_Gather(…)` same syntax as `MPI_Scatter` but `recv_*` items only relevant on `root`.

Non-Root Collectives

- `MPI_Allgather (send_data, send_count, send_type, recv_data, recv_count, recv_type, comm, ierror)`
  - Like `MPI_Gather` but without a `root` process. `Send_*` and `recv_*` items relevant on all processes. All processes in `comm` must participate.
- `Ditto for MPI_Alltoall(…)`
- Also `MPI_Scatterv, MPI_Gatherv, MPI_Allgatherv, MPI_Alltoallv`
  - Augmented versions of `MPI_Scatter` etc.
  - `send_count` becomes an array `send_counts` (i.e. can send different number of elements to each process)
  - Extra integer array `displs` is added ("displacements" so the data to be scattered need not be contiguous, i.e. can send sub-blocks of arrays).

Global Reduction

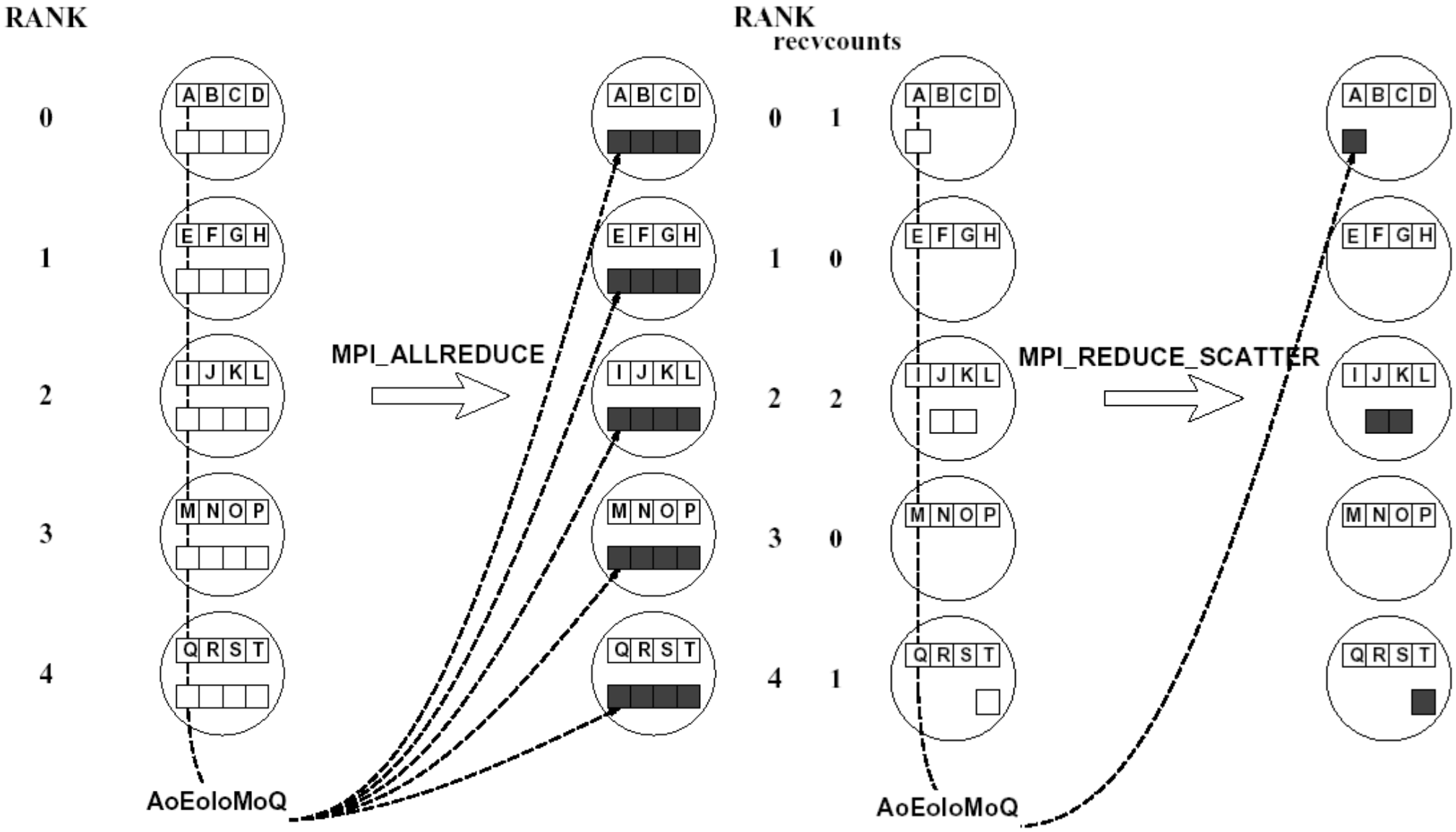


- Use to compute a global result from distributed data.
- Result is an array on `root` process only, undefined on others.
- All processes call with identical arguments except for `send_data` and `recv_data`
  ...

Global Reductions

- `MPI_Reduce(send_data, recv_data, count, type, op, root, comm, ierror)`
  - Where `op` is either one of the predefined MPI reduction operators for MPI standard datatypes:
    - `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC` (b=bitwise)
  - Or a user-defined operation which must then register with MPI library via `MPI_Op_create` – see literature for details

Non-Root Global Reductions



`MPI_ALLREDUCE` is like `MPI_REDUCE` but no `root` so all processes receive same result. `MPI_REDUCE_SCATTER` differs in that processes elect to receive a certain sized segment of the result.

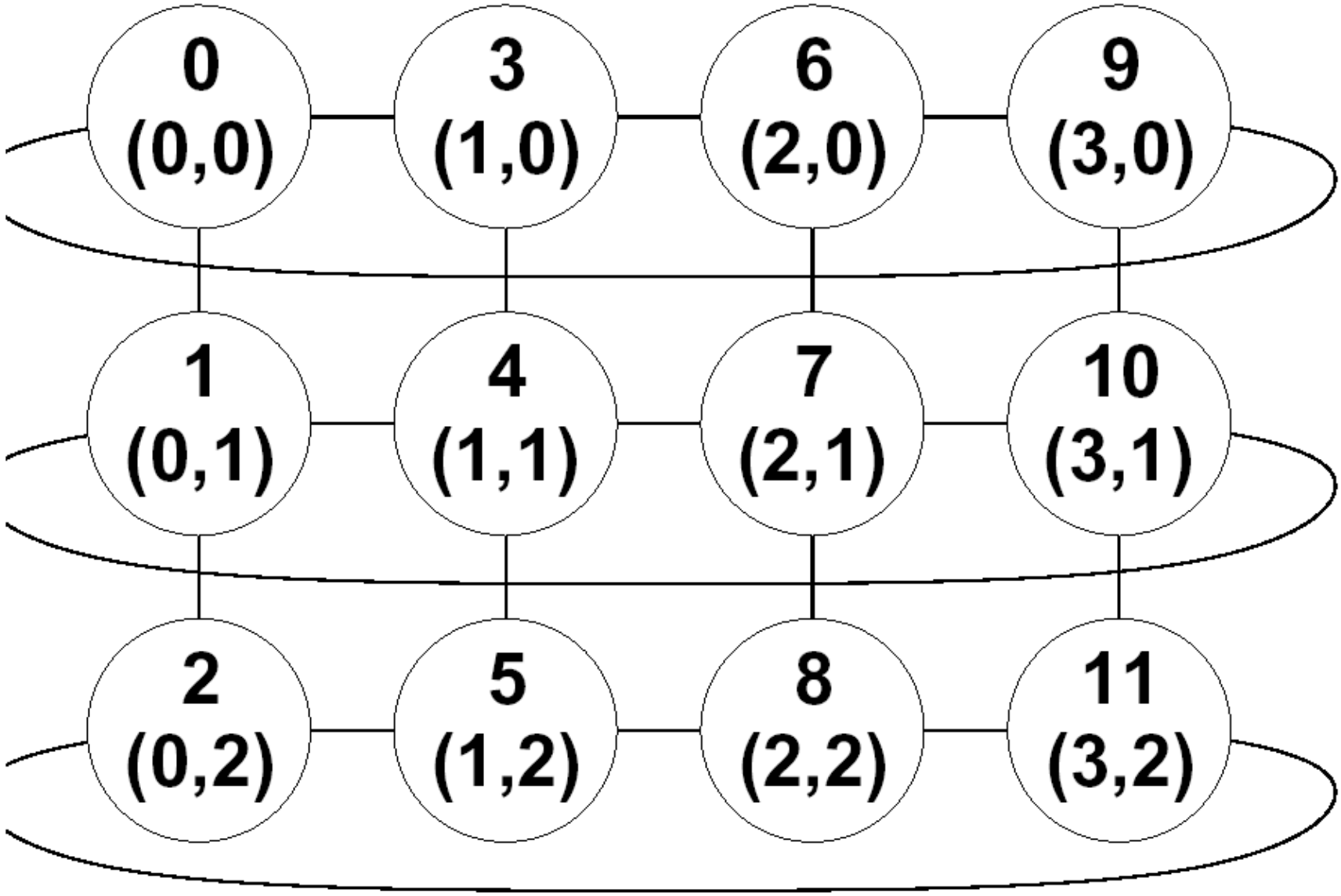
Manipulating Communicators

- By default, all comms is within `MPI_COMM_WORLD` but sometimes useful to split into smaller groups
- Can create a (set of) new communicators by splitting existing communicator using
  - `MPI_Comm_split(old_comm, split_key, split_rank, new_comm, ierror)`
  - Where all processors with same value of `split_key` will be in same `new_comm`, and rank in `new_comm` will be given by `split_rank`.
- Can also create arbitary sub-groupings using `MPI_Comm_create` but a bit more complex.

Virtual Topologies

- Sometimes useful to simplify coding and/or communications by defining a *virtual topology*.
  - Especially when mapping grid data onto processors with appropriate boundary conditions
  - Provides a way of mapping virtual ranks to actual ranks of processes
  - `MPI_Cart_create(old_comm, ndims, dims, periods, reorder, comm_cart, ierror)`
  - Where `ndims` is number of dimensions in `comm_cart`, `dims` is number of processes in each dimension, `periods` is a logical array for PBCs and `reorder` is `.FALSE.` if data already on processors (so ranks remain as in `old_comm`) , otherwise `.TRUE.` may reassign ranks if better for faster communications.
  - NB MPI numbers dimensions from `0` to `ndim-1` …

Example Virtual Topology



Virtual topology with PBCs in only 1 direction. Must use `comm_cart` in other MPI calls to benefit from new mappings

Cartesian Mapping

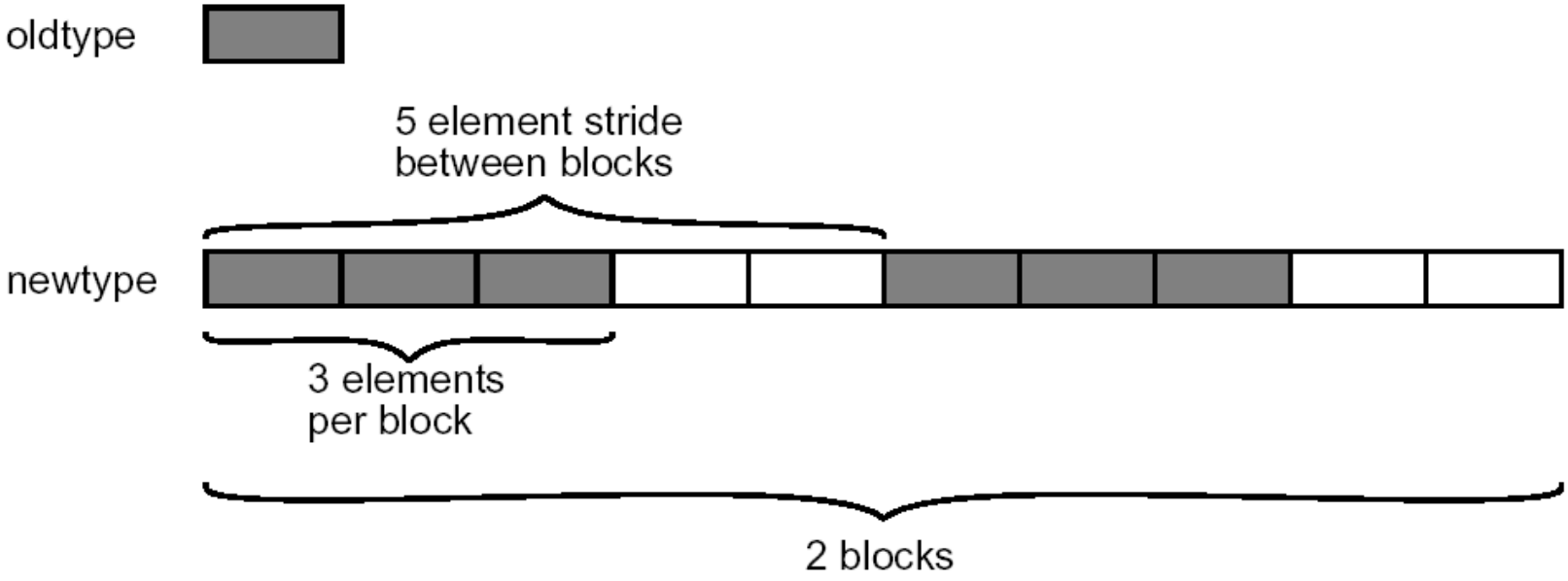
- To benefit from virtual topology can use Cartesian mapping functions to convert grid coordinates into processor ranks :
  - `MPI_Cart_rank(cart_comm, coords, rank, ierror)`
- And v.v. :
  - `MPI_Cart_coords(cart_comm, rank, maxdims, coords, ierror)`
    - where `maxdims` is length of `coords` array that is returned
  - `MPI_Cart_shift(cart_comm, direction, disp, rank_source, rank_dest, ierror)`
    - Returns correct ranks corresponding to virtual shift in `direction` (`0` to `ndims-1`) of `disp` process coordinates. Returns `rank_source` as where the calling process should receive a message from, and `rank_dest` as to where the message should be sent.

MPI Derived Types

- Might want to send several items data of same type but non-contiguous in memory (e.g. array slice) or contiguous data of mixed type (e.g. C `struct` or F90 `type`).
  - Can either use lots of small messages or create new MPI derived type and save on latency and number of MPI calls.
- Need to construct the new MPI type using combination of existing types with calls to `MPI_Type_vector(...)` and/or `MPI_Type_struct(...)`, then register it with the system using `MPI_Type_commit(new_type)` after which it can be used multiple times before being released with `MPI_Type_free(new_type)`

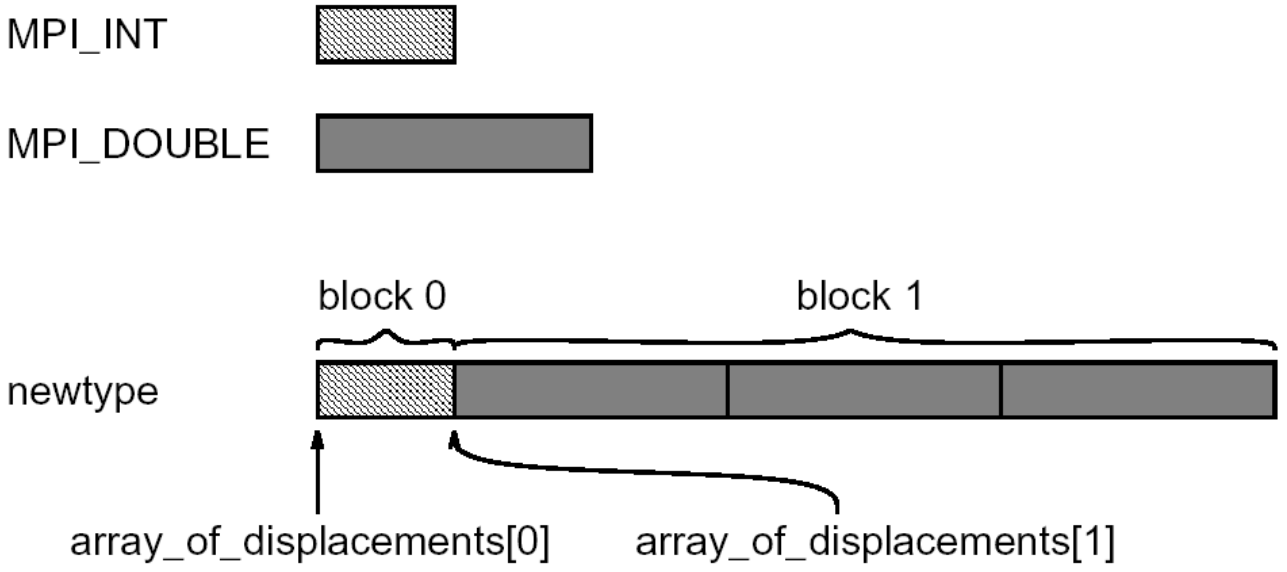
Creating a Vector Type

- `MPI_Type_vector (count, block_length, stride, old_type, new_type, ierror)`
- E.g. `count=2, stride=5, block_length=3`:



Creating a Structure Type

- `MPI_Type_struct(count, blocklengths, displacements, types, new_type, ierror)`
- e.g. `struct{int a; double b[2]} foo` has `count=2`
- `blocklengths[0]=1; displacements[0]=0; types[0]=MPI_INT`
- `blocklengths[1]=3; displacements[1]=&foo.b-&foo; types[1]=MPI_DOUBLE`
- Issues with padding and alignment of mixed types in MPIv1. Fixed with `MPI_Type_create_struct` in MPIv2



MPI debugging

- Can use gdb on each MPI instance …
  - Launch code using `mpiexec` NOT `mpirun`
  - Then logon to relevant node and launch gdb with `--pid` option or the `attach` command
  - See http://www.sci.utah.edu/~tfogal/academic/Fogal-ParallelDebugging.pdf for more GNU details
- Also similar with Intel idb …
  - http://www.jaist.ac.jp/iscenter-new/mpc/altix/altixdata/opt/intel/idb/10.0.026/doc/idb_manual/lin/idb_starting_parallel_debugging_session.htm
  - (and/or Google for the Intel idb manual)

MPI profiling

- MPI profiling using `gprof` …
  - Need to tell gprof to add pid to each process gmon.out file so all are unique:

```
export GMON_OUT_PREFIX=gmon.out-
```

  - Then compile MPI code & run as usual
  - Then combine the different gmon.out files

```
gprof -s a.out gmon.out-*
```

  - Then use gmon.sum to generate an overall profile

```
gprof a.out gmon.sum
```

- Use `mpiP` to profile the MPI comms so combined with gprof can do comms:calc ratio

MPI v2

- 1st v2 in 1996 but final MPI 1.3 in 2008!
- Added features for parallel I/O
  - Requires specialized hardware support
  - Can have multiple processes write to different parts of same file at same time
- Dynamic process management
  - MPI can now spawn new MPI processes
- And remote memory access (RMA)
  - Can now do 1-sided “get” and “put” data
  - Faster but requires hardware support

What happened to MPI v2?

- MPI v2 was not a great success as many of its features required specialized hardware support
  - Hence not widely available
  - Hence not very portable
  - Hence not very popular
- And not many programmers needed the extra features so stayed with MPI v1

MPI v3

- Launched in 2012 (v3.1 in 2015)
  - Extends the 1-sided RMA functions
  - Adds support for shared memory programming (e.g. OpenMP within a node) so can now do *hybrid parallelism*
  - Adds support for non-blocking collective comms
  - Adds F2008 bindings
- MUCH more useful and portable …
  - See Advanced HPC lectures for more!

Further Reading

- Chapter 9 of “Introduction to High Performance Computing for Scientists and Engineers”, Georg Hager and Gerhard Wellein, CRC Press (2011).
- EPCC course notes at http://www.epcc.ed.ac.uk/education-training/
- MPI forum https://www.mpi-forum.org
- MPI homepage http://www.mcs.anl.gov/research/projects/mpi including MPI standards, examples and more.
- mpiP from LLNL at http://mpip.sourceforge.net/