Corso di Laurea
Magistrale in Informatica

UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dipartimento di Scienze
Fisiche, Informatiche e Matematiche

Titolare del corso: prof. Marko Bertogna (marko.bertogna@unimore.it)

AA 2016/2017

MPI: Advanced uses

The MPI runtime

- Provides for process placement, execution & handling
- Handles signals (SIGKILL, SIGSUSP, SIGUSR1/2)
- Usually collects stdout and stderr, may propagate stdin
- May propagate environment variables
- May provide support for debugging, profiling, tracing
- May interface with a queuing system for better process placement
- MPI-2 specifies (but doesn't require) standardized mpirun clone: mpiexec. Others: poe, mpprun, prun...
- Command line arguments and/or environment variables allow for different behavior/performance

Point to Point (pt2pt) communications

- Blocking comms: Block until completed (send stuff on your own)
- Non-blocking comms: Return without waiting for completion (give them to someone else)
- Forms of Sends:
  - Synchronous: message gets sent only when it is known that someone is already waiting at the other end (think fax)
  - Buffered: message gets sent and if someone is waiting for it so be it; otherwise it gets saved in a temporary buffer until someone retrieves it. (think mail)
  - Ready: Like synchronous, only there is no ack that there is a matching receive at the other end, just a programmer's assumption! (Use it with extreme care)

Blocking pt2pt communication

- MPI_Ssend: Synchronous send
  - The sender notifies the receiver; after the matching receive is posted the receiver acks back and the sender sends the message.
- MPI_Bsend: Buffered (asynchronous) send
  - The sender notifies the receiver and the message is either buffered on the sender side or the receiver side according to size until a matching receive forces a network transfer or a local copy respectively.
- MPI_Rsend: Ready send
  - The receiver is notified and the data starts getting sent immediately following that. Use at your own peril!

Things to consider

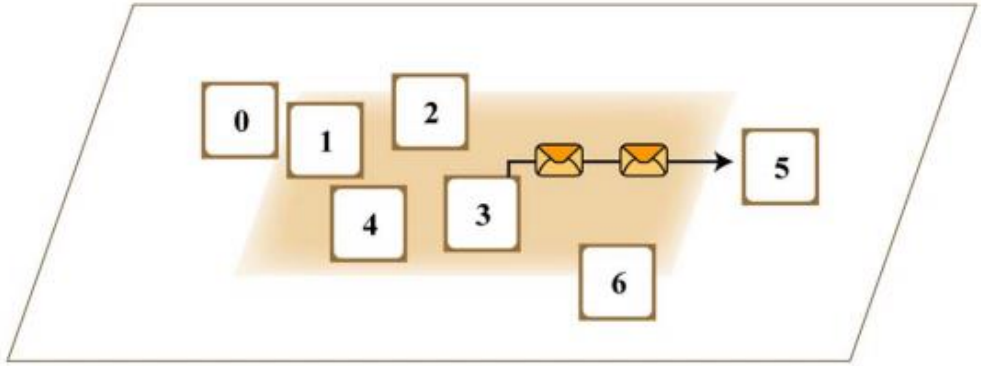
- Depending on the MPI implementation, MPI_Send may behave as either MPI_Ssend or MPI_Bsend.
  - Usually buffered for small messages, synchronous for larger ones. Very small messages may piggypack on the initial notification sent to the receiver. The switchover point can be tunable (see P4_SOCKBUFSIZE for MPICH)
  - For buffered sends it uses an internal system buffer
- MPI_Bsend may use a system buffer but cannot be guaranteed to work without a user-specified buffer setup using
  - MPI_Buffer_attach(void *buffer, int size)
- MPI_Ssend can lead to deadlocks

Blocking send performance

- Synchronous sends offer the highest asymptotic data rate (AKA bandwidth) but the startup cost (latency) is very high, and they run the risk of deadlock.
- Buffered sends offer the lowest latency but:
  - suffer from buffer management complications
  - have bandwidth problems because of the extra copies and system calls
- Ready sends should offer the best of both worlds but are so prone to cause trouble they are to be avoided!
- Standard sends are usually the ones that are most carefully optimized by the implementors. For large message sizes they can always deadlock.

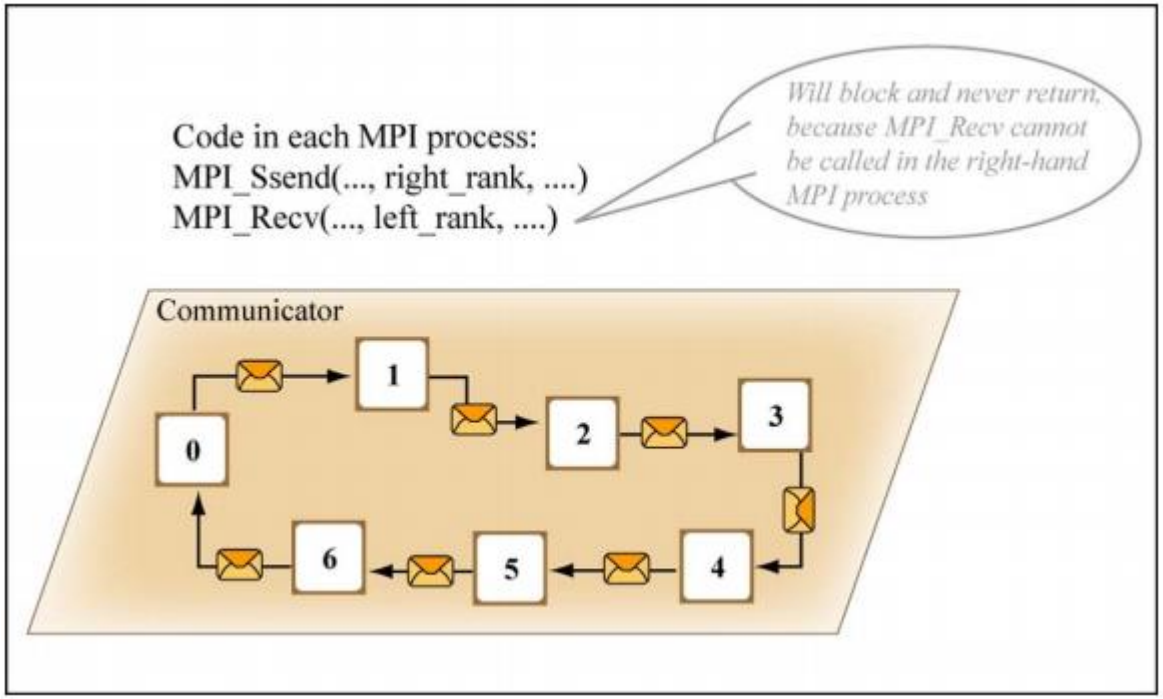
Message passing restrictions

- Order is preserved. For a given channel (sender, receiver, communicator) message order is enforced:
  - If P sends to Q, messages sa and sb in that order, that is the order they will be received at B, even if sa is a medium message sent with MPI_Bsend and sb is a small message sent with MPI_Send. Messages do not overtake each other.
  - If the corresponding receives ra and rb match both messages (same tag) again the receives are done in order of arrival.
  - This is actually a performance drawback for MPI but helps avoid major programming errors.



Deadlock around the ring

Consider a ring communication scenario where everyone talks to one's neighbour to the right around a circle. If synchronous blocking communicatios are used (MPI_Ssend or MPI_Send for large messages) the messages never get delivered as everybody needs to send before receiving!



Bidirectional Exchange

- Consider the cases where two processes are exchanging messages concurrently or a process is sending a message while receiving another.
- There are two routines that provide an optimized deadlock free macro for this operation:
- MPI_Sendrecv(int *sendbuf, int sendcnt, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int src, int recvtag, MPI_Comm comm, MPI_Status *stat)
- MPI_Sendrecv_replace does not use the recvbuf, recvcnt and recvtype arguments as the source array gets overwritten (like a swap)

Nonblocking communications

- The situation can be rectified by using nonblocking communication routines that return immediately, without making sure that the data has been safely taken care of.
  - That way after the send the receive can be posted and the deadlock is avoided
  - But **beware**: Until such a time that the communication is successfully completed, no pointer input arguments to the routines may be modified as they wrong data/parameters will be used when the communication does take place.
  - This is unlike the situation with the blocking comms where upon return from the call, one is free to reuse the args.
  - MPI_Ixxxx instead of MPI_xxxx for the names

MPI nonblocking standard send

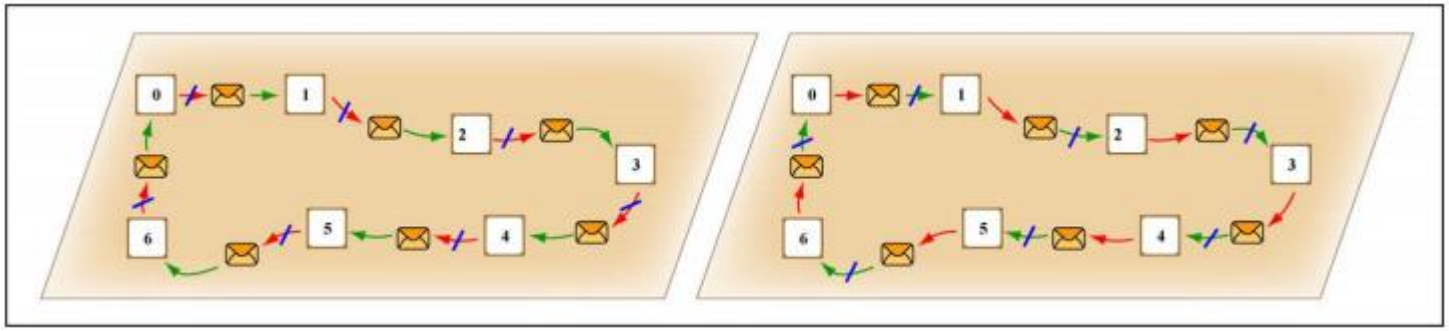
- MPI_Isend(void *buf, int cnt, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)
- MPI_Wait(MPI_Request *req, MPI_Status *stat)
- Call MPI_Isend, store the request handle, do some work to keep busy and then call MPI_Wait with the handle to complete the send.
- MPI_Isend produces the request handle, MPI_Wait consumes it.
- The status handle is not actually used

MPI nonblocking receive

- MPI_Irecv(void *buf, int cnt, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Request *req)
- MPI_Wait(MPI_Request *req, MPI_Status *stat)
- Call MPI_Irecv, store the request handle, do some work to keep busy and then call MPI_Wait with the handle to complete the receive.
- MPI_Irecv produces the request handle, MPI_Wait consumes it.
- In this case the status handle is actually used.

Deadlock avoidance

- If the nonblocking sends or receives are called backto-back with MPI_Wait we basically retrieve the blocking behavior as MPI_Wait is a blocking call.
  - To avoid deadlock we need to interlace nonblocking sends with blocking receives, or nonblocking receives with blocking sends; the nonblocking calls always precede the blocking ones. Using both nonblocking calls may land us in trouble again unless we reverse the order of Wait calls, or interlace the order of send and receive calls (even #P).

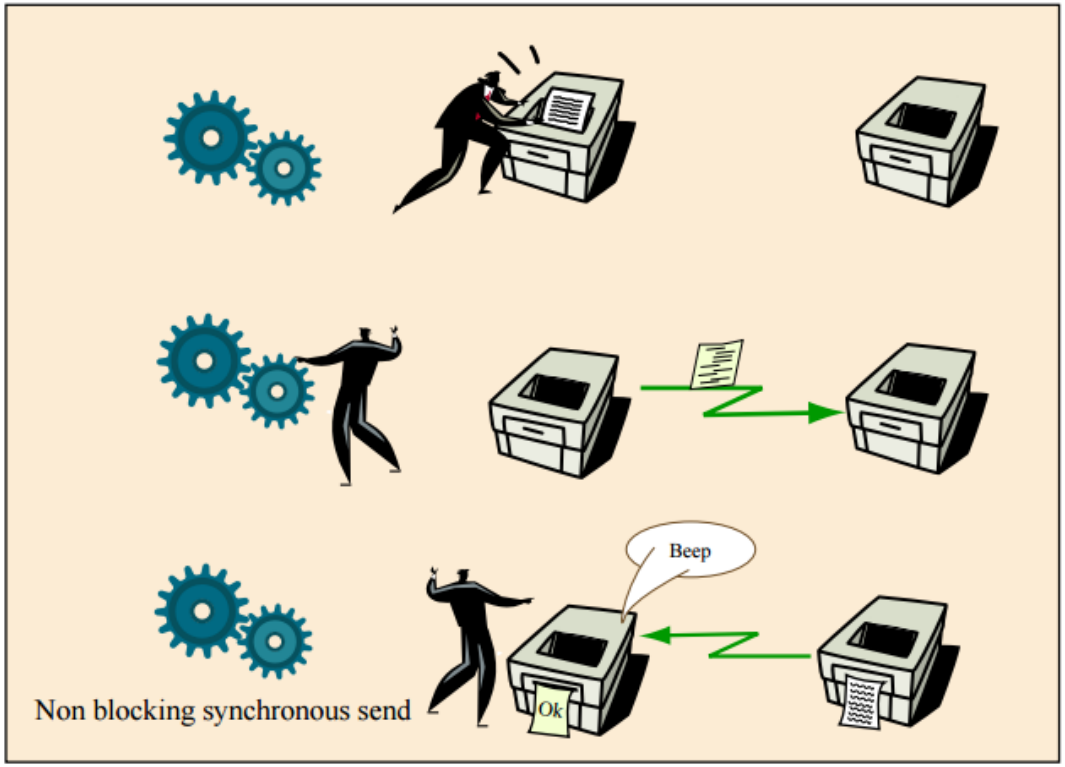


Other nonblocking sends

- For each blocking send, a nonblocking equivalent:
  - MPI_Issend: nonblocking synchronous send
  - MPI_Ibsend: nonblocking asynchronous send
  - MPI_Irsend: nonblocking ready send
- Take care not to confuse nonblocking send with asynchronous send although the terminology has been used interchangeably in the past!
  - A successful blocking asynchronous send returns very quickly and the send buffer can be reused.
  - **Any nonblocking call** returns immediately and the buffer cannot be tampered with until the corresponding blocking MPI_Wait call has returned!

Nonblocking Synchronous Send

For example, an MPI_Issend() works like using an unattended fax machine. You set up the fax to be sent, go away but need to come and check if all's been sent.



Implementing nonblocking communications

- The actual communication in the case of the nonblocking calls can take place at any given time between the call to the MPI_Isend/Irecv operation and the corresponding MPI_Wait.
- The moment it actually happens is implementation dependent and in many cases it is coded to take place mostly within MPI_Wait.
- On systems with "intelligent" network interfaces it is possible for communications to be truly taking place concurrently with the computational work the sending process is performing, thus allowing for computation to "hide" communication; otherwise nonblocking calls just help avoid deadlock without helping performance.

Testing instead of Waiting

- MPI_Test(MPI_Request *req, int *flag, MPI_Status *stat)
- Instead of calling the blocking MPI_Wait call MPI_Test; if flag is true, the message has been received with more information in stat and ier. Otherwise the routine returns and can be called after a while to test for message reception again.
- On systems where one can have real overlap of communication with computation MPI_Test allows finer control over communication completion times.

All, some and any

- Suppose one has a large number of outstanding nonblocking calls to wait or test for. MPI provides us with special shorthand routines for this case:
  - MPI_Waitall/MPI_Testall deal with arrays of requests and statuses as their arguments. They test for all pending communication requests.
  - Ditto for MPI_Waitsome/MPI_Testsome but they also mark the locations of successful operations in another index array. They return after some time at least one completion on pending requests (usually more).
  - Finally MPI_Waitany/MPI_Testany will test for all the pending requests and return if they come across one that is completed or if none are completed.

Other comms routines & handles

- MPI_Probe/MPI_IProbe will check for a message awaiting to be received but will not actually receive it - one needs to call MPI_Recv/MPI_Irecv for that.
- MPI_Cancel(MPI_Request *req) will mark a pending send or receive for cancellation. One still needs to call MPI_Wait or MPI_Test or MPI_Request_free to free the request handle. The message may still be delivered at that time! *MPICH based implementations beware!*
- MPI_Test_cancelled
- MPI_Send_init, MPI_Recv_init and MPI_Start, MPI_Startall: Persistent comms
- MPI_PROC_NULL, MPI_REQUEST_NULL

Persistent Communications

- In the case of very regular communications (say inside a loop), some communication overhead can be avoided by setting up a persistent communication request ("half" channel or port). **There is no binding of receiver to sender!**
- MPI_Send_init(buf, count, datatype, dest, tag, comm, request) sets up persistent sends. The request is inactive and corresponds to an Isend(). Corresponding initialization calls for Bsend, Ssend and Rsend exist.
- MPI_Recv_init(buf, count, datatype, source, tag, comm, request) sets up persistent receives. The request is inactive and corresponds to an Irecv().

Persistent Communications

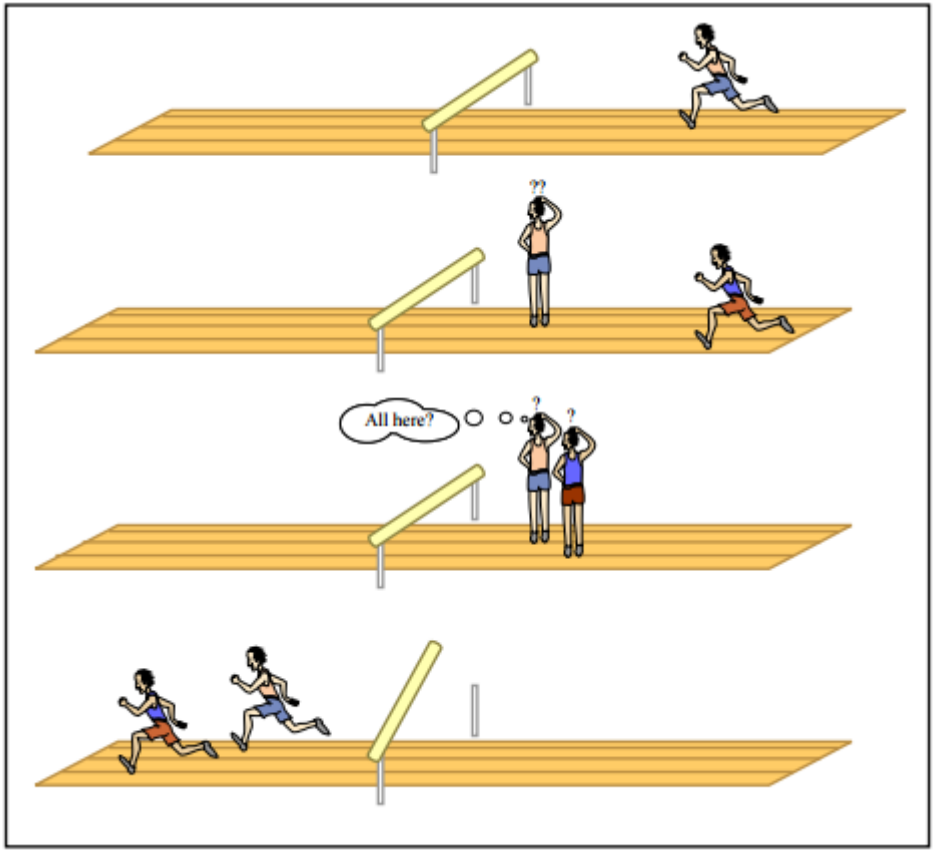
- To activate a persistent send or receive pass the request handle to MPI_Start(request).
- For multiple persistent communications employ MPI_Startall (count, array_of_requests). This processes request handles in some arbitrary order.
- To complete the communication, MPI_Wait()/Test() and friends are needed. Once they return, the request handle is once again inactive but allocated. To deallocate it MPI_Request_free() is needed. Make sure it operates on an inactive request handle.
- Persistent sends can be matched with blocking or nonblocking receives and vice-versa for the receives.

Wildcards & Constants

- MPI_PROC_NULL: operations specifying this do not actually execute. Useful for not treating boundary cases separately to keep code cleaner.
- MPI_REQUEST_NULL: The value of a null handle, after it is released by the MPI_Wait()/Test() family of calls or by MPI_Request_free()
- MPI_ANY_SOURCE: Wild card for source
- MPI_ANY_TAG: Wildcard for tag
- MPI_UNDEFINED: Any undefined return value

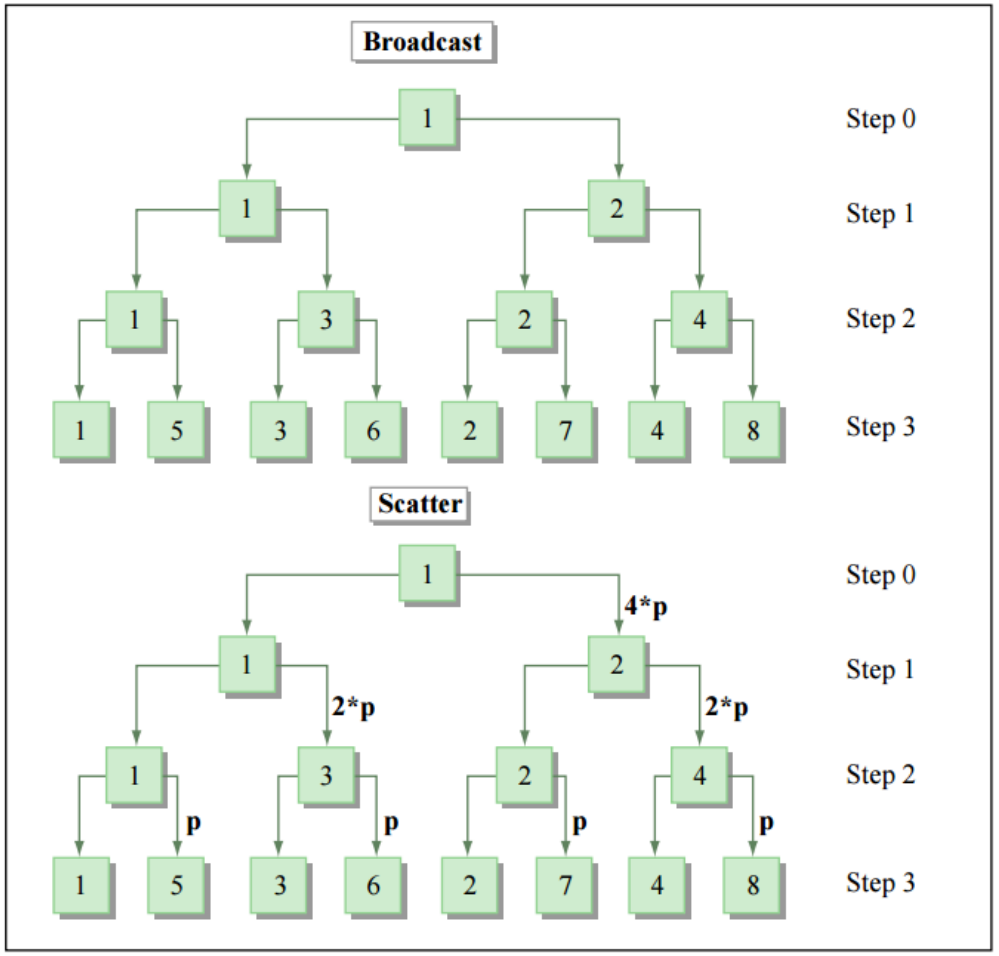
Synchronization

- MPI_Barrier(MPI_Comm comm)
- Forces synchronization for:
  - timing purposes
  - non-parallel I/O purposes
  - debugging
- Costly for large numbers of processes, try to avoid.



Collectives implementation

Binary trees



All to All Personalized Comm

- MPI_Alltoall(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)
- Everybody sends something different to everyone else, like a scatter/gather for all. If what was getting sent was the same it would be like a bcast/gather for all.
- This is the most stressful communication pattern for a communication network as it floods it with messages: P*(P-1) for P procs for certain direct implementations.
- MPI_Alltoallv has additional arguments for variable sendcnt and recvcnt, and input and output strides

Groups, Contexts, Communicators

- Group: An ordered set of processes, each associated with a rank (within a continuous range). Part of a communicator.
  - Predefined: MPI_GROUP_EMPTY,MPI_GROUP_NULL
- Context: A property of a communicator that partitions the communication space. Not externally visible.
  - Contexts allow Pt2Pt and collective calls not to interfere with each other; same with calls belonging to different communicators.
- Communicator: Group+Context+cached info
  - Predefined: MPI_COMM_WORLD,MPI_COMM_SELF
- Intra- and Inter-communicators

Group Constructors

- MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
- MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
- MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
- MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
- MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
- MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
- MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)
- MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)

More group functions

- MPI_Group_free(MPI_Group *group)
- MPI_Group_size(MPI_Group group, int *size)
- MPI_Group_rank(MPI_Group group, int *rank)
- MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)
- MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)
- MPI_IDENT results if the group members and group order is exactly the same in both groups. This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if the group members are the same but the order is different. MPI_UNEQUAL results otherwise.

Communicator Functions

- MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
- MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
- MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
- MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)
- MPI_Comm_free(MPI_Comm *comm)
- And the MPI_Comm_size,MPI_Comm_rank we have already met.

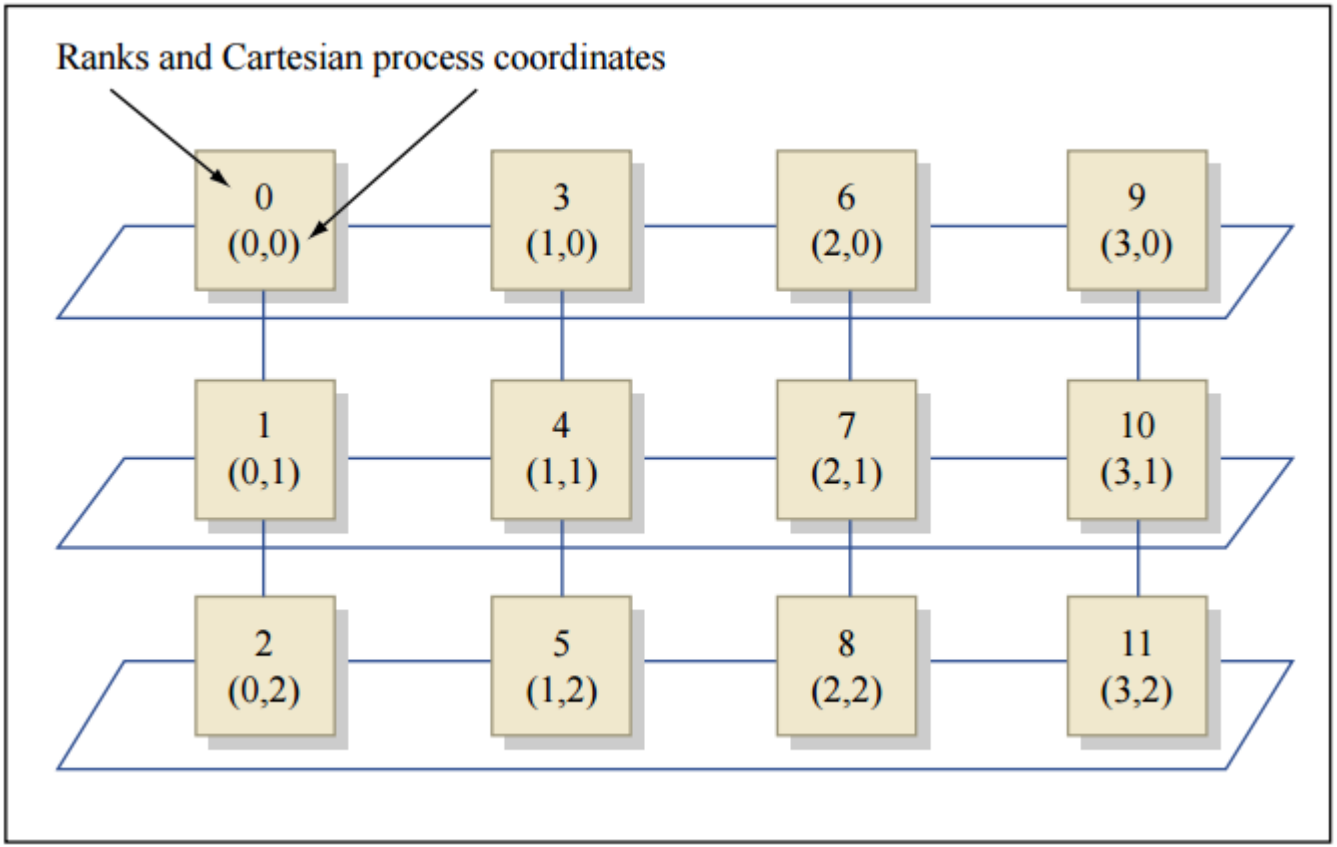
Inter-Communicators

- So far all communications have been between processes belonging to the same communicator.
- MPI allows for communications between different communicators.
  - They can only be Pt2Pt and not collective
  - They require the generation of inter-communicator objects.
  - For more look at the MPI standard definitions.

Virtual Topologies

- Employing the information cached in communicators we can map an (intra-)communicator's processes to an underlying topology (cartesian or graph) that better reflects the communication requirements of our algorithm.
- This has possible performance advantages: The process to hardware mapping could be thus more optimal. *In practice this is rare*.
- The notational power of this approach however allows code to be far more readable and maintainable.

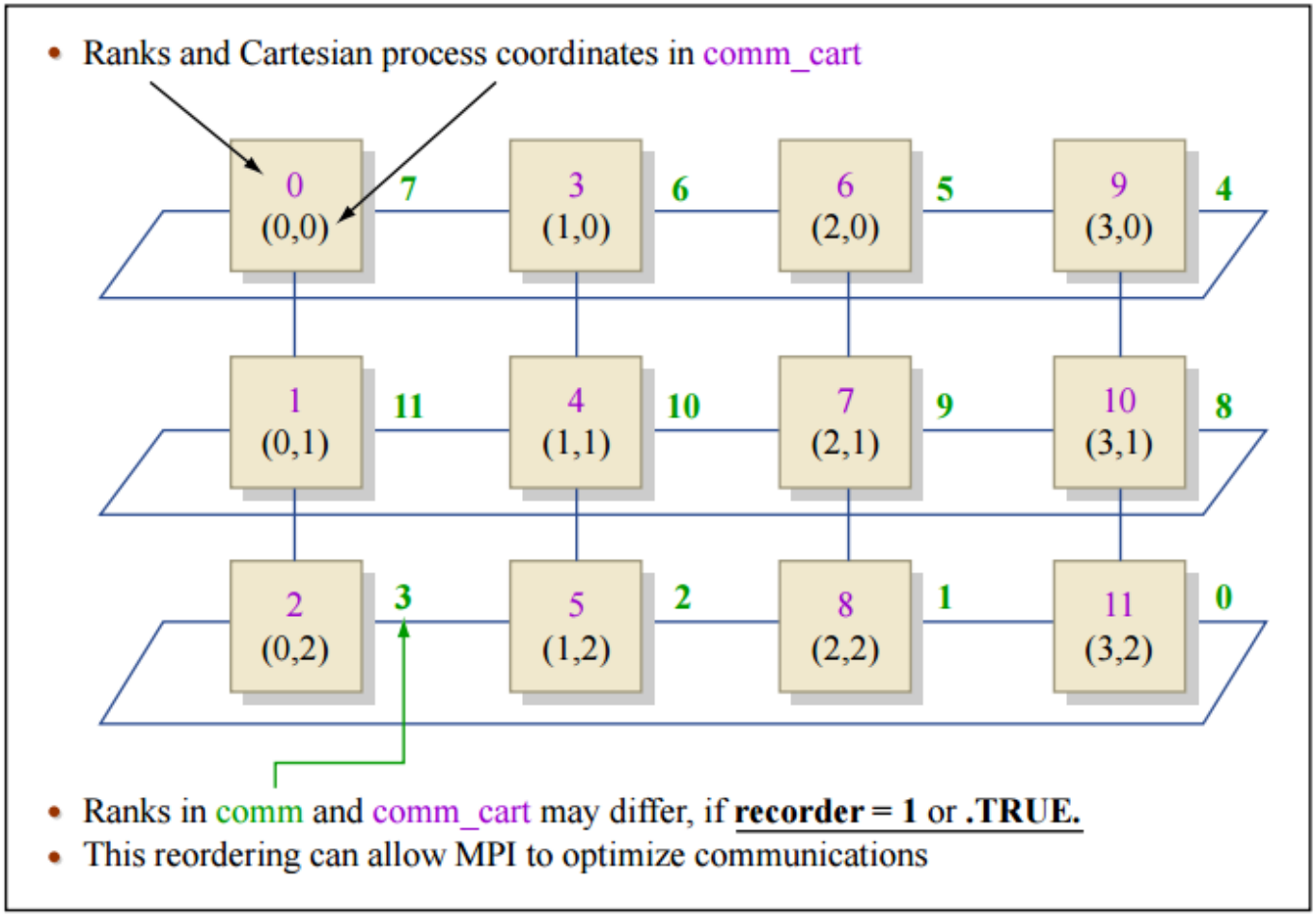
A cartesian topology



Cartesian topology calls

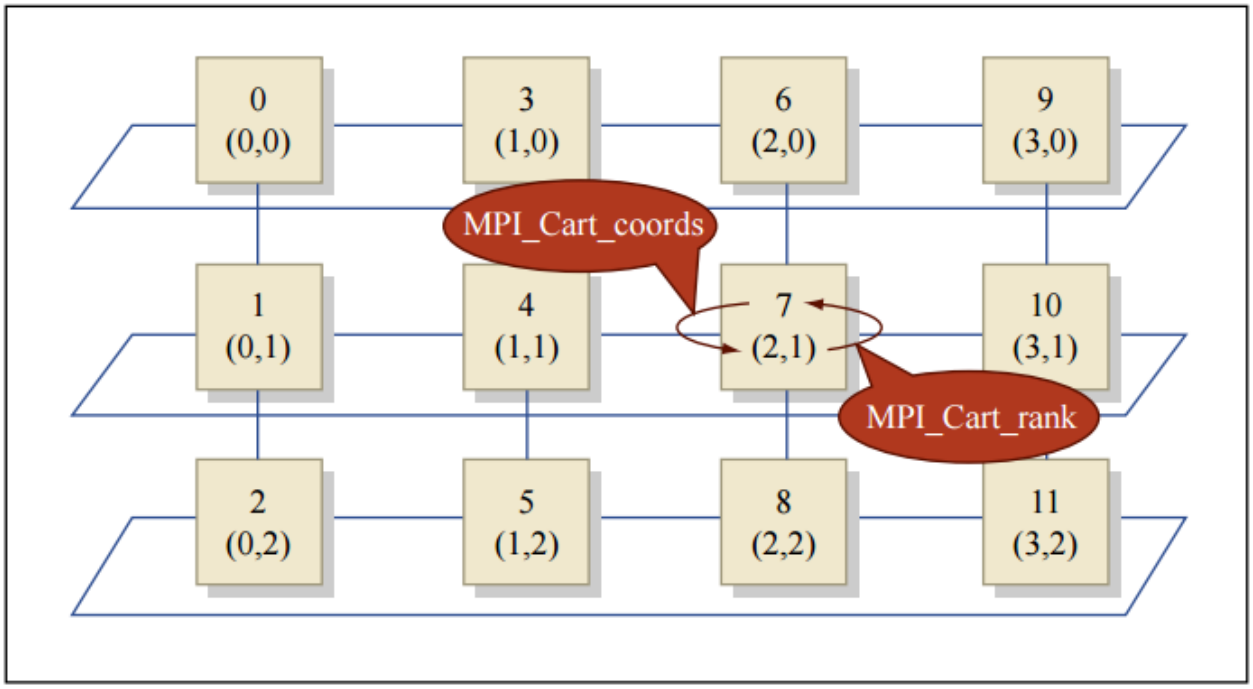
- MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)
  - Extra processes get MPI_COMM_NULL for comm_cart
- MPI_Dims_create(int nnodes, int ndims, int *dims)
  - If ndims(k) is set, this is a constraint
- For graphs, MPI_Graph_create(), same rules
- MPI_Topo_test(MPI_Comm comm, int *status)
  - Returns MPI_CART, MPI_GRAPH, MPI_UNDEFINED
- MPI_Cartdim_get, MPI_Cart_get etc. for cartesian topologies
- MPI_Graphdim_get, MPI_Graph_get etc. for graphs

Ranks in cartesian communicators



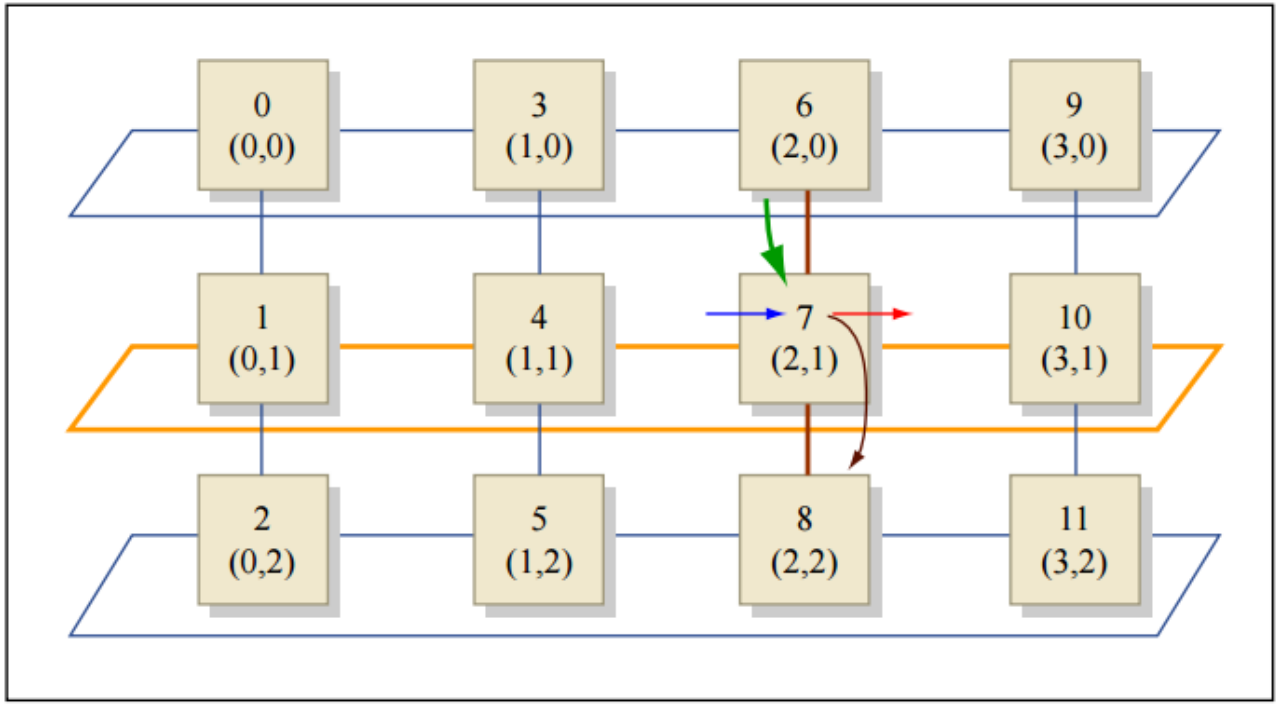
Cartesian rank/coordinate functions

- MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank); out of range values get shifted (periodic topos)
- MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

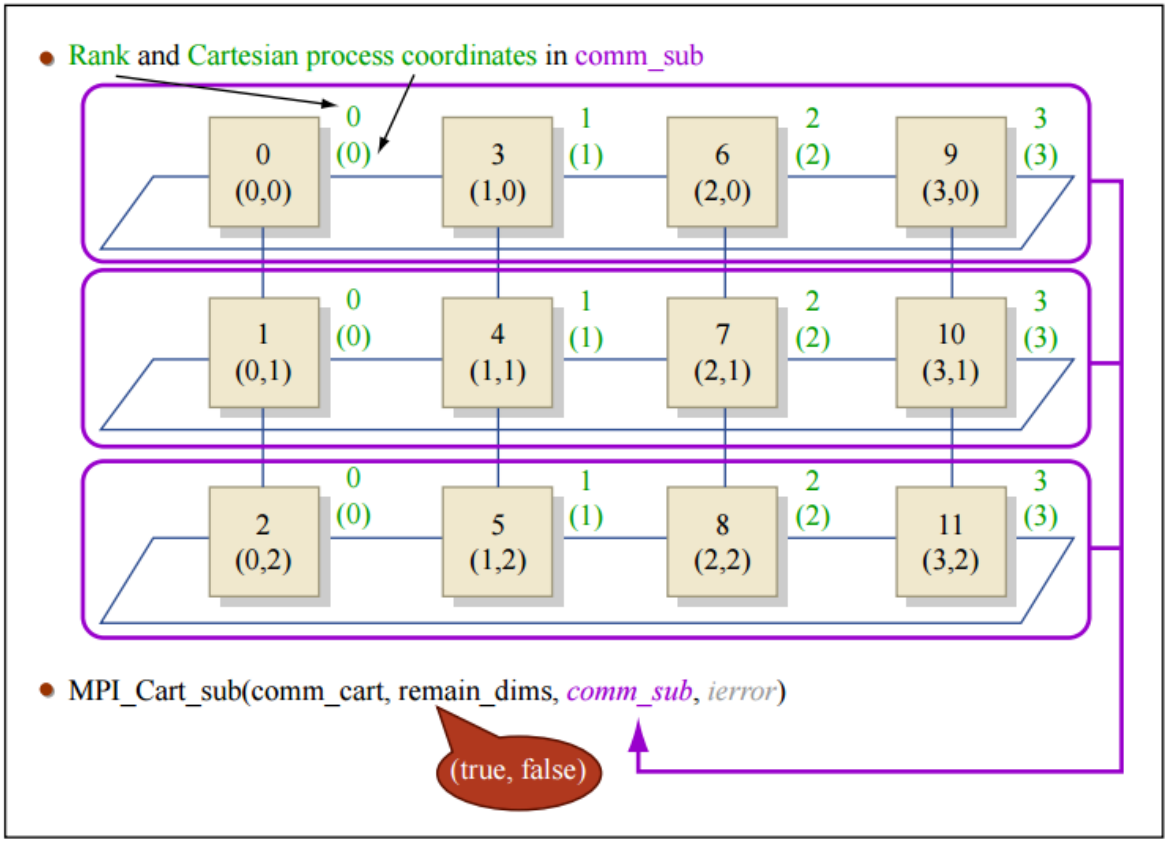


Cartesian shift

- MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
  - MPI_PROC_NULL for shifts at non-periodic boundaries



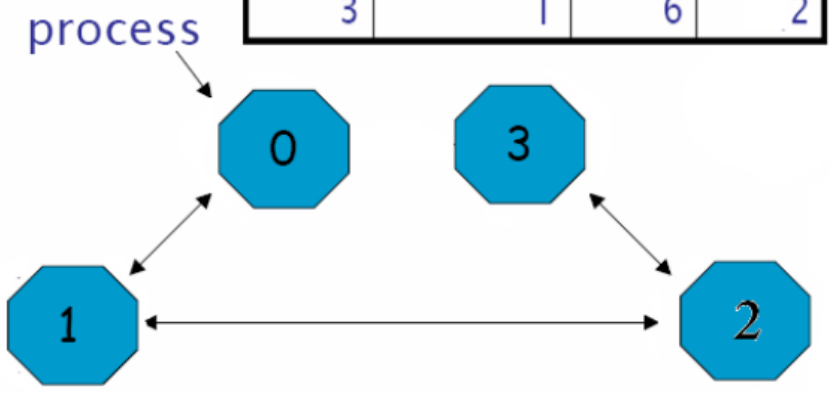
Cartesian subspaces



Graph Topology

- Elements of Graph Topology
  - Nodes:Processors
  - Lines: Communicators between nodes
  - Arrows:Show origins and destinations of links
  - Index: array of integers describing node degrees

Node	Nneighbors	index	edges
0	1	1	1
1	2	3	0,2
2	2	5	1,3
3	1	6	2



Graph functions

- MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
- MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)
- Used in that order to get the neighbors of a process in a graph.