Large-Scale Parallel Computing

Prof. Dr. Felix Wolf

MESSAGE PASSING INTERFACE PART 2

Outline

- Virtual topologies
- Point-to-point communication

2D Poisson problem

Simple PDE at the core of many applications

$$\nabla^2 u = f(x,y) \quad \text{in the interior} \qquad (1)$$

$$u(x,y) = g(x,y) \quad \text{on the boundary} \qquad (2)$$

Simplification

- Domain is unit square
- Discretization via square mesh
    - n+2 points along each edge

$$x_i = \frac{i}{n+1}, i = 0,\ldots,n+1$$

$$y_j = \frac{j}{n+1}, j = 0,\ldots,n+1$$

Jacobi iteration

We can approximate Equation (1) at each of these points using the formula:

$$\frac{u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - 4u_{i,j}}{h^2} = f_{i,j} \quad \text{with} \quad h = \frac{1}{n+1}$$

Can be rewritten as:

$$u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - h^2 f_{i,j})$$

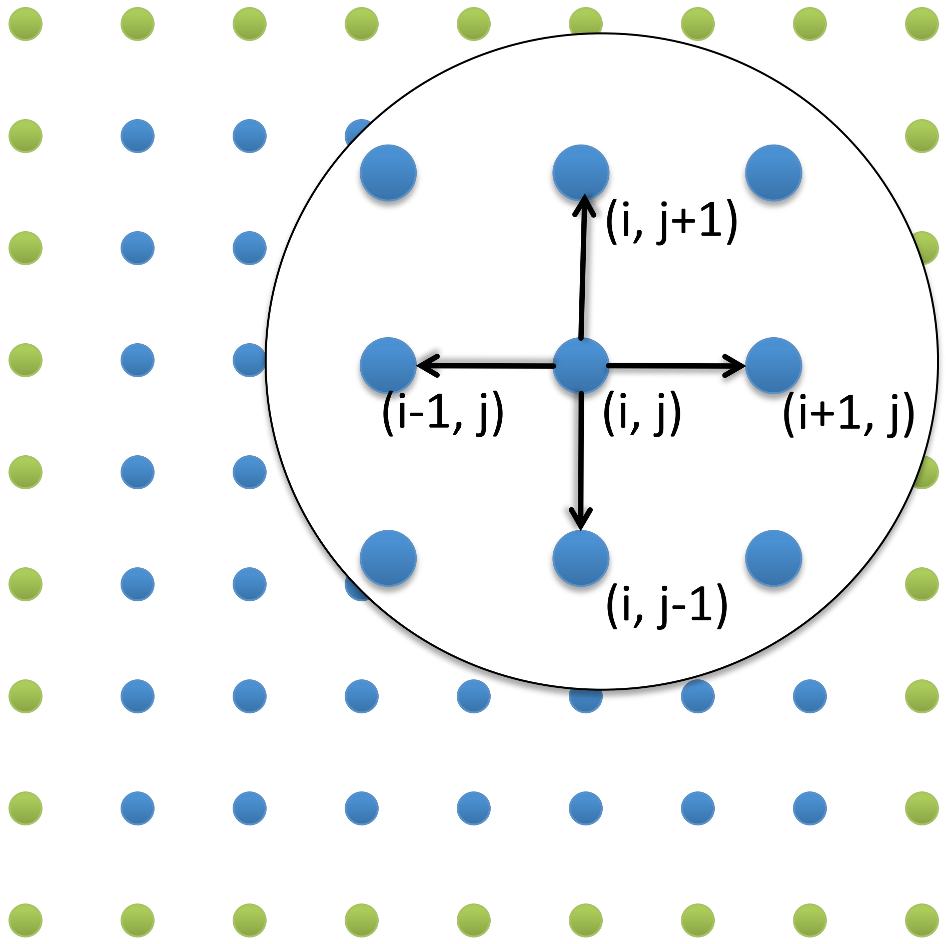
We iterate by choosing values for all $u_{i,j}$ and replace them using:

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} + u_{i+1,j}^{k} - h^2 f_{i,j})$$

Stencil approximation

TECHNISCHE UNIVERSITÄT DARMSTADT

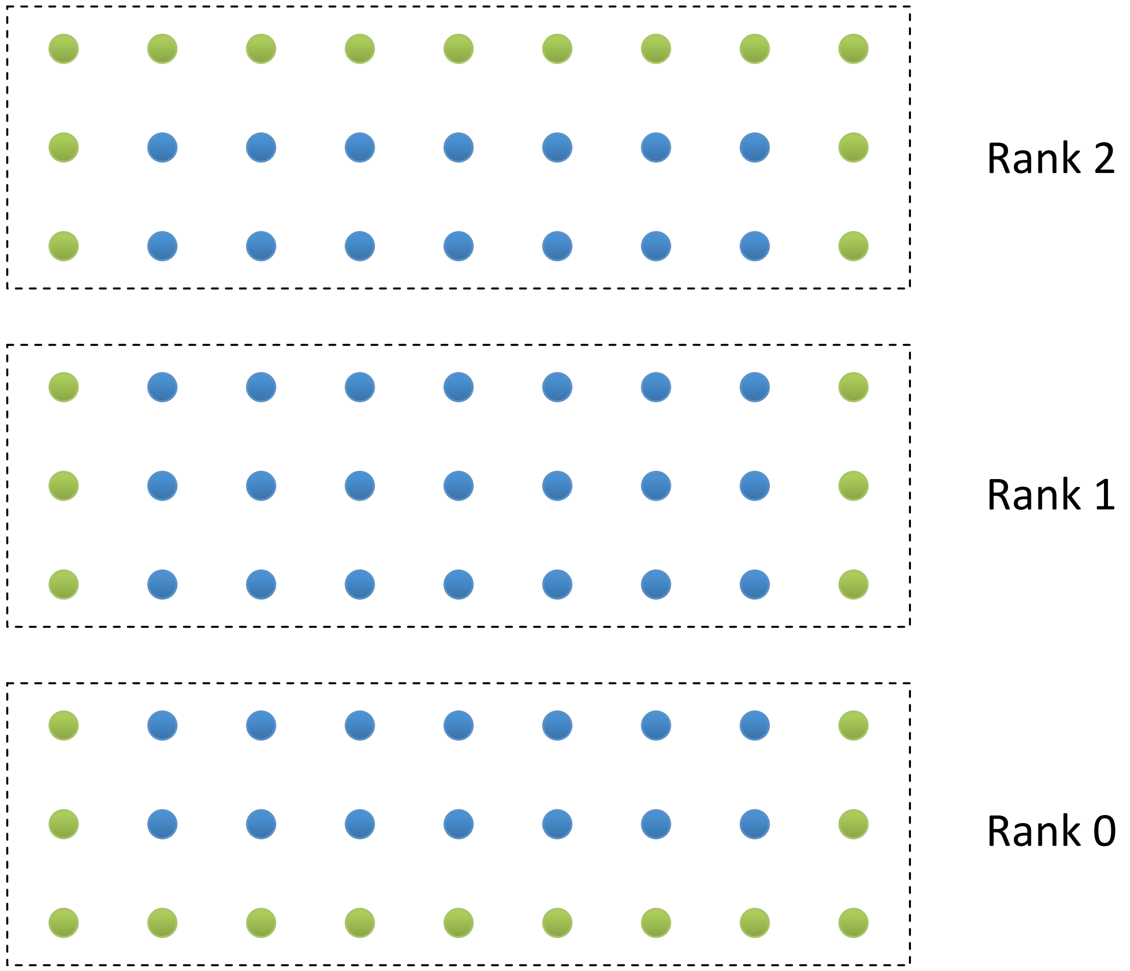
n = 7



Jacobi iteration

```
    integer i, j, n
    double precision u(0:n+1,0:n+1), unew(0:n+1,0:n+1)
    do 10 j=1, n
       do 10 i=1, n
          unew(i,j) = &
              0.25 * (u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - &
              h * h * f(i,j)
10: continue
```

1D decomposition



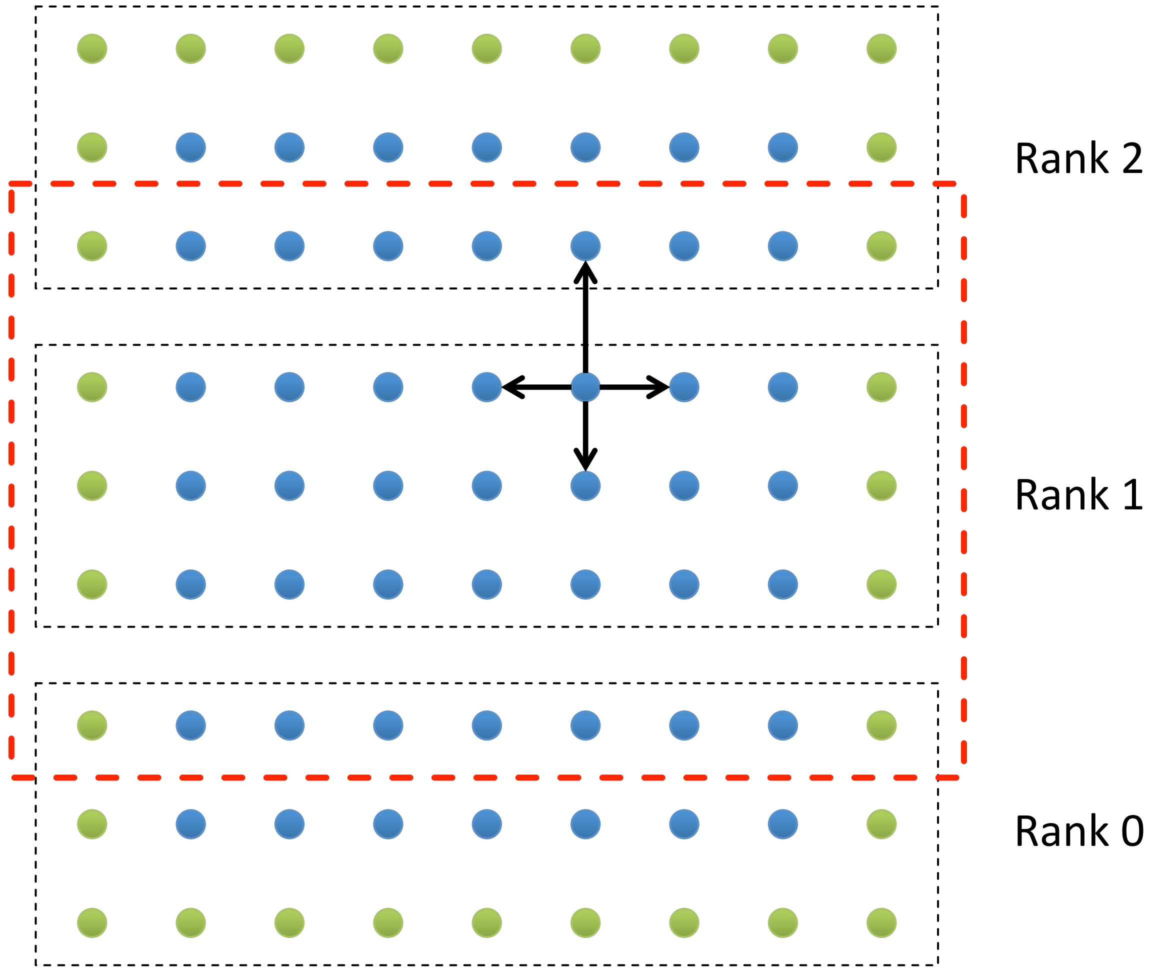
Jacobi iteration for a slice of the domain

```
    integer i, j, n
    double precision u(0:n+1,s:e), unew(0:n+1,s:e)
    do 10 j=s, e
       do 10 i=1, n
          unew(i,j) = &
              0.25 * (u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - &
              h * h * f(i,j)
10: continue
```

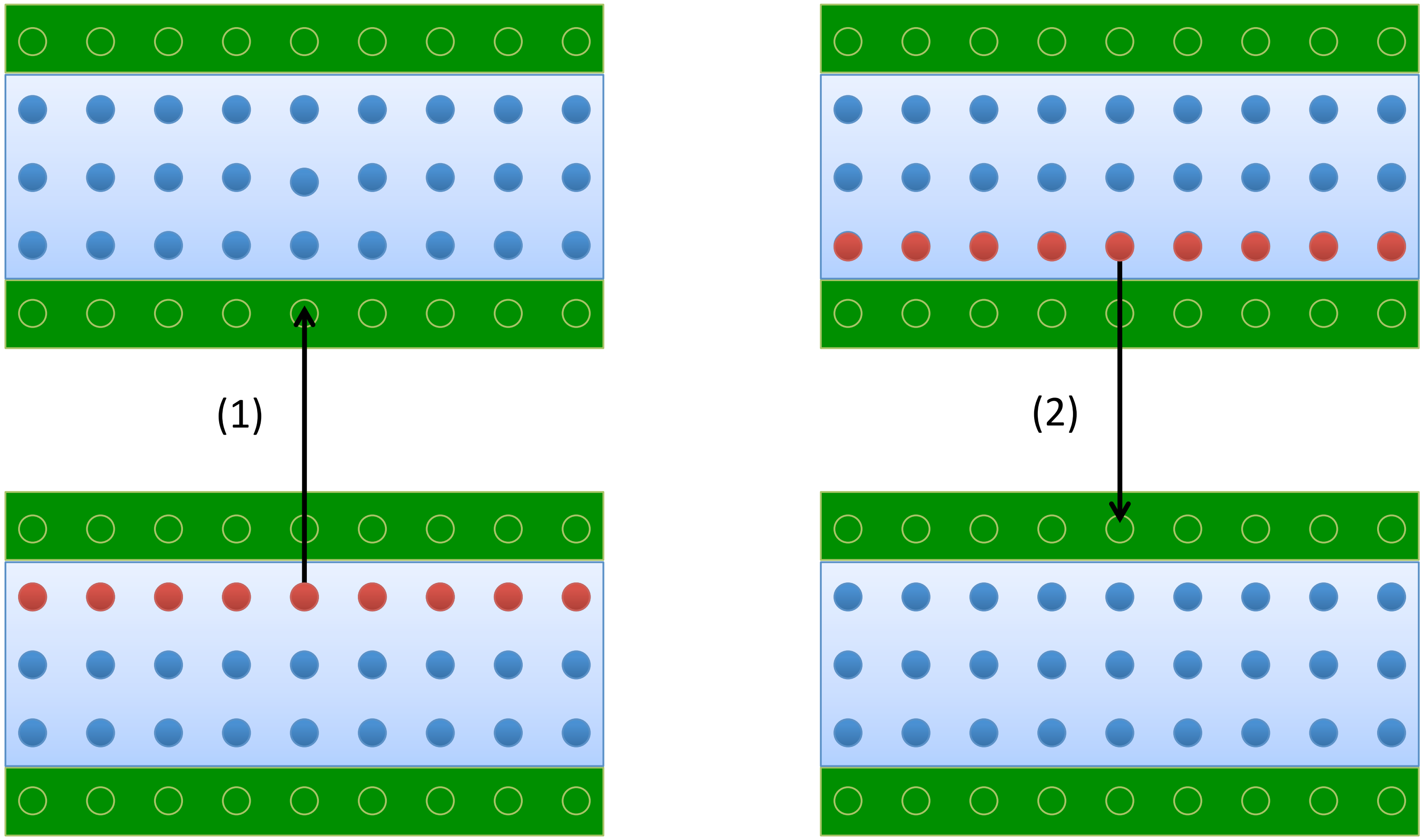
- Problem – the loop will require elements such as u(i, s-1) that belong to another process
- Necessitates array expansion to hold ghost points

```
!   center slice
    double precision u(0:n+1,s-1:e+1), unew(0:n+1,s:e)
```

Computational domain with ghost points



Two-step data transfer



Topologies

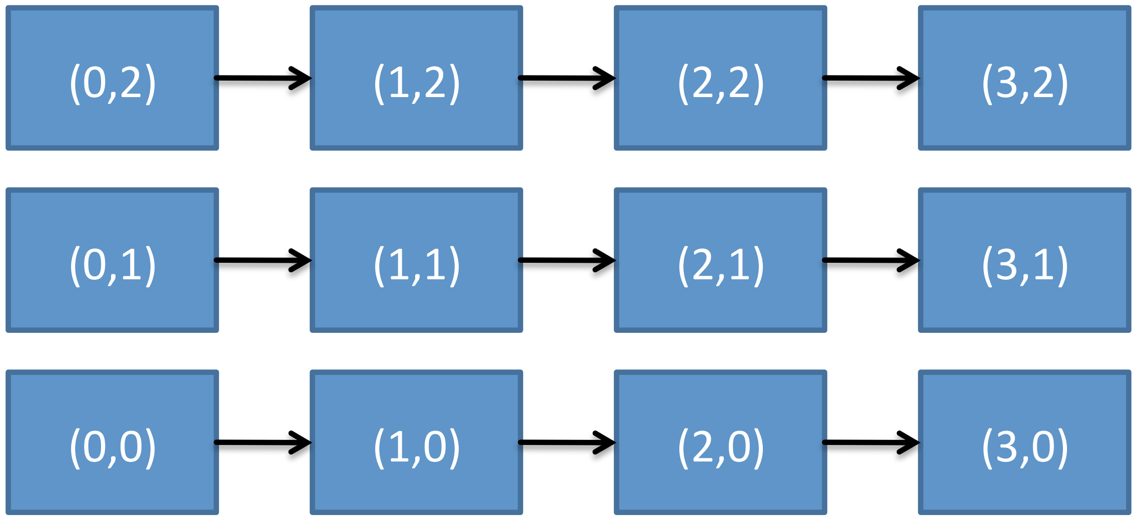
- The description of how processors in a parallel computer are connected to each other is called network topology or physical topology
- The description of which processes in a parallel program communicate with each other is called application topology or virtual topology

Topologies (2)

- Topology functions allow convenient process naming
- The way the virtual topology is mapped onto the physical topology can influence performance
  - Some mappings are better than others
- MPI allows the implementation to help optimize this aspect through topology functions

Cartesian topologies

- Decomposition in the natural coordinate directions
  - Arrows show shift in the first dimension



- MPI provides collection of routines for defining, examining, and manipulating Cartesian topologies

Creating a Cartesian topology

```
integer dims(2), ndim
logical isperiodic(2), reorder

dims(1)     = 4
dims(2)     = 3
isperiodic(1) = .false.
isperiodic(2) = .false.
reorder       = .true.
ndim          = 2
call MPI_CART_CREATE( MPI_COMM_WORLD, ndim, dims, isperiodic, &
                      reorder, comm2d, ierr )
```

Creates Cartesian topology from previous slide

- isperiodic indicates whether processes at the “end” are connected
- reorder allows function to reorder process ranking for better performance

How to access coordinates?

- In one dimension we can simply use the rank in the communicator unless processes have been reordered
- More complicated for more than one dimension

```
call MPI_CART_GET( comm2d, 2, dims, isperiodic, coords, ierr )
print *, '(', coords(1), ',', coords(2),')'
```

- Returns coordinates of the calling process plus dimensions sizes and periodicity

```
call MPI_COMM_RANK( comm2d, myrank, ierr )
call MPI_CART_COORDS( comm2d, myrank, 2, coords, ierr )
```

- Returns coordinates of a given rank

How to find neighbors?

```
[…]
call MPI_CART_CREATE( MPI_COMM_WORLD, ndim, dims, isperiodic, &
                      reorder, comm2d, ierr )
call MPI_CART_SHIFT( comm2d, 0, 1, nbrleft, nbrright, ierr )
call MPI_CART_SHIFT( comm2d, 1, 1, nbrbottom, nbrtop, ierr )
```

- 2nd argument indicates the direction
  - The coordinate dimension (0,..., ndim-1) to be traversed by the shift.
- 3rd argument indicates displacement (integer)
  - > 0: upwards shift
  - < 0: downwards shift
- Depending on the periodicity, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift
  - In the case of an end-off shift, the value MPI_PROC_NULL is returned

C-binding of topology functions

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims,
                    int *dims, int *isperiodic,
                    int reorder, MPI_comm *newcomm);

int MPI_Cart_shift(MPI_Comm comm, int direction,
                   int displacement, int *src, int *dest);

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,
                 int *isperiodic, int *coords);

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank);

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
                    int *coords);
```

Domain decomposition

- How to divide the domain among the processes?
- Trivial if number of processes evenly divides n

```
s        = 1 + myrank * (n / procs)
e        = s + (n / procs) - 1
```

- Otherwise

```
nlocal   = n / numprocs
s        = myid * nlocal + 1
deficit = mod(n,numprocs)
s        = s + min(myid,deficit)
if (myid .lt. deficit) nlocal = nlocal + 1
e        = s + nlocal - 1
if (e .gt. n .or. myid .eq. numprocs-1) e = n
```

Exchange of ghost points

```
      subroutine exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
      include 'mpif.h'
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1)
      integer comm1d, nbrbottom, nbrtop
      integer status(MPI_STATUS_SIZE), ierr
c
      call MPI_SEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, &
                     comm1d, ierr )
      call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, &
                     comm1d, status, ierr )
      call MPI_SEND( a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, &
                     comm1d, ierr )
      call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, &
                     comm1d, status, ierr )
      return
      end
```

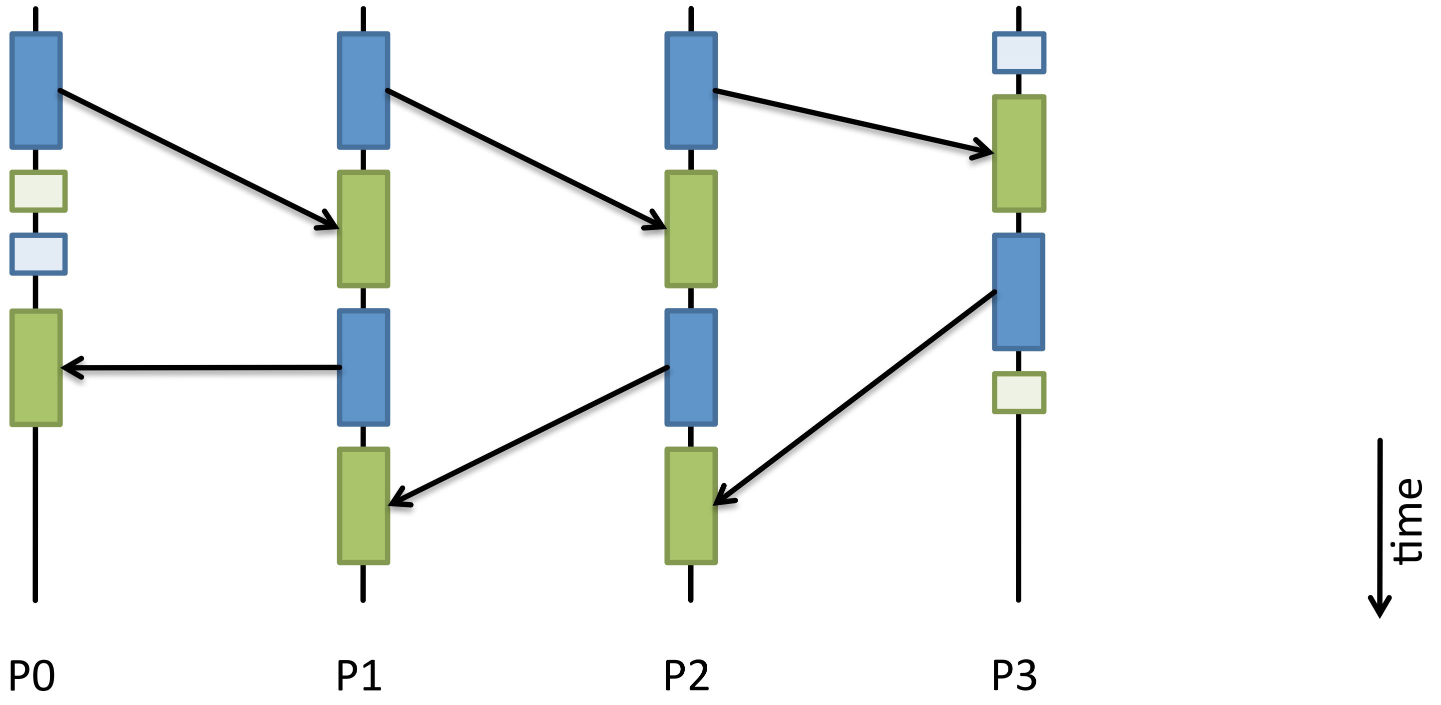
Sweep

```
      subroutine sweep1d( a, f, nx, s, e, b )
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1), f(0:nx+1,s-1:e+1)
     +                 b(0:nx+1,s-1:e+1)
c
      integer i, j
      double precision h
c
      h = 1.0d0 / dble(nx+1)
      do 10 j=s, e
         do 10 i=1, nx
            b(i,j) = 0.25 * (a(i-1,j)+a(i,j+1)+a(i,j-1)+a(i+1,j)) -
     +                 h * h * f(i,j)
10    continue
      return
      end
```

Jacobi iteration

```
      call MPI_CART_CREATE( MPI_COMM_WORLD, 1, numprocs, .false., &
                            .true., comm1d, ierr )
      call MPI_COMM_RANK( comm1d, myid, ierr )
      call MPI_Cart_shift( comm1d, 0,  1, nbrbottom, nbrtop, ierr   )
c
c Compute the decomposition and initialize a, b, and f
      […]
c
      do 10 it=1, maxit
         call exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
         call sweep1d( a, f, nx, s, e, b )
         call exchng1( b, nx, s, e, comm1d, nbrbottom, nbrtop )
         call sweep1d( b, f, nx, s, e, a )
         dwork = diff( a, b, nx, s, e )
         call MPI_Allreduce( dwork, diffnorm, 1, MPI_DOUBLE_PRECISION, &
                             MPI_SUM, comm1d, ierr )
        if (diffnorm .lt. 1.0e-5) goto 20
10     continue
      if (myid .eq. 0) print *, 'Failed to converge'
20    continue
      if (myid .eq. 0) print *, 'Converged after ', 2*it, ' Iteratios'
```

Intended exchange pattern

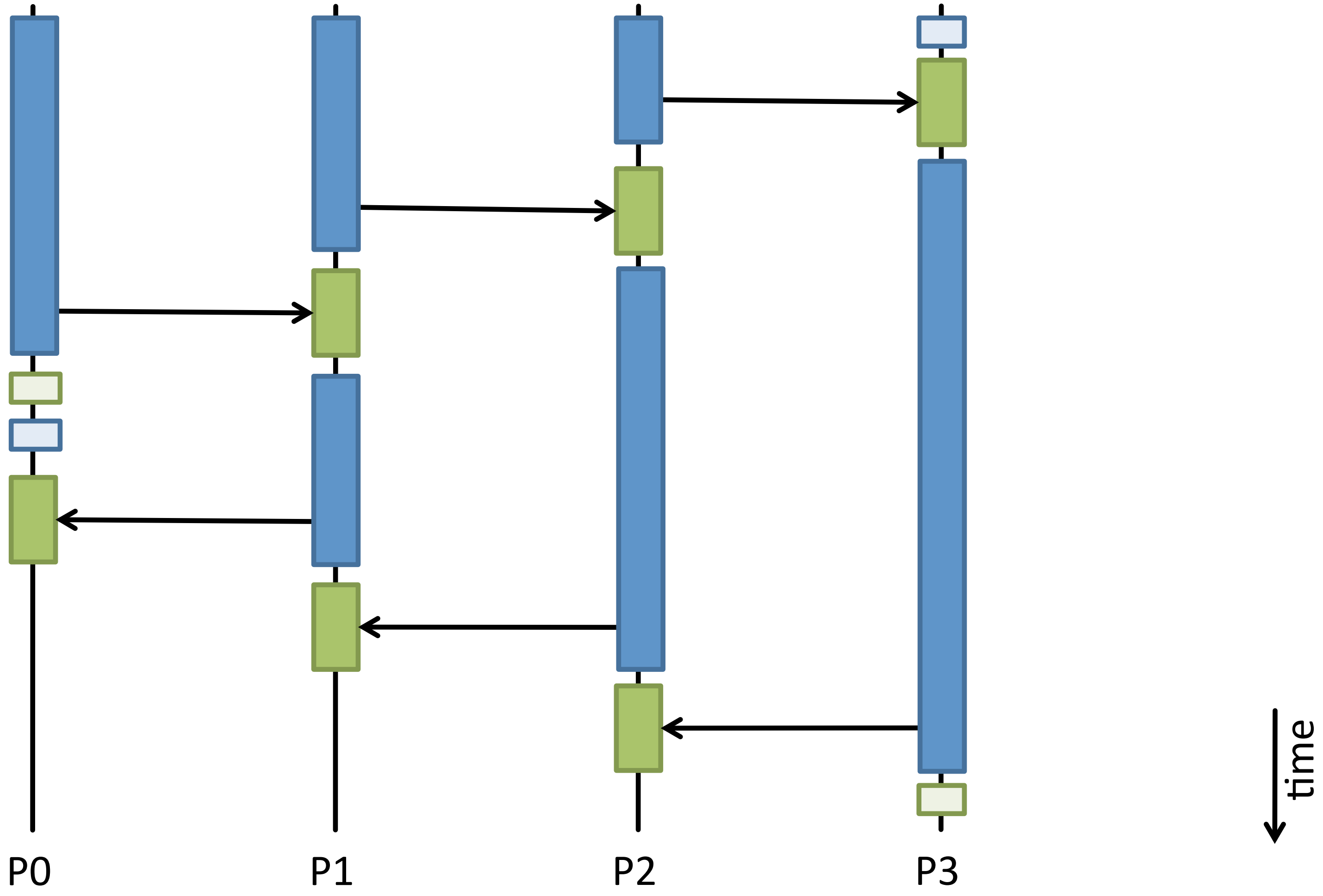


Assumes that messages can be buffered

Buffering semantics of MPI_Send

- An MPI implementation is permitted to copy the message to be sent into internal storage to allow the MPI_Send to return
  - This is called buffering the message
- But it is not required to do so because there simply might not be enough buffer space

Potential serialization in the case of large nx

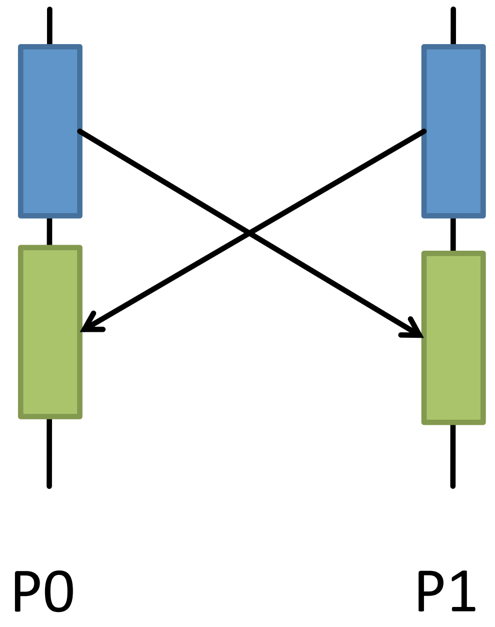


Solution alternatives

- Pairing sends and receives
- Combined send and receive
- Buffered sends
- Non-blocking communication

Excursion: deadlock

- What happens if the two sends do not return until the receivers have posted their receive?
- Definition
  - Two processes are deadlocked if each process is waiting for an event that only the other process can cause
  - If more than two processes are involved in a deadlock then they are waiting in a circular chain
- Analogy: chicken and egg problem



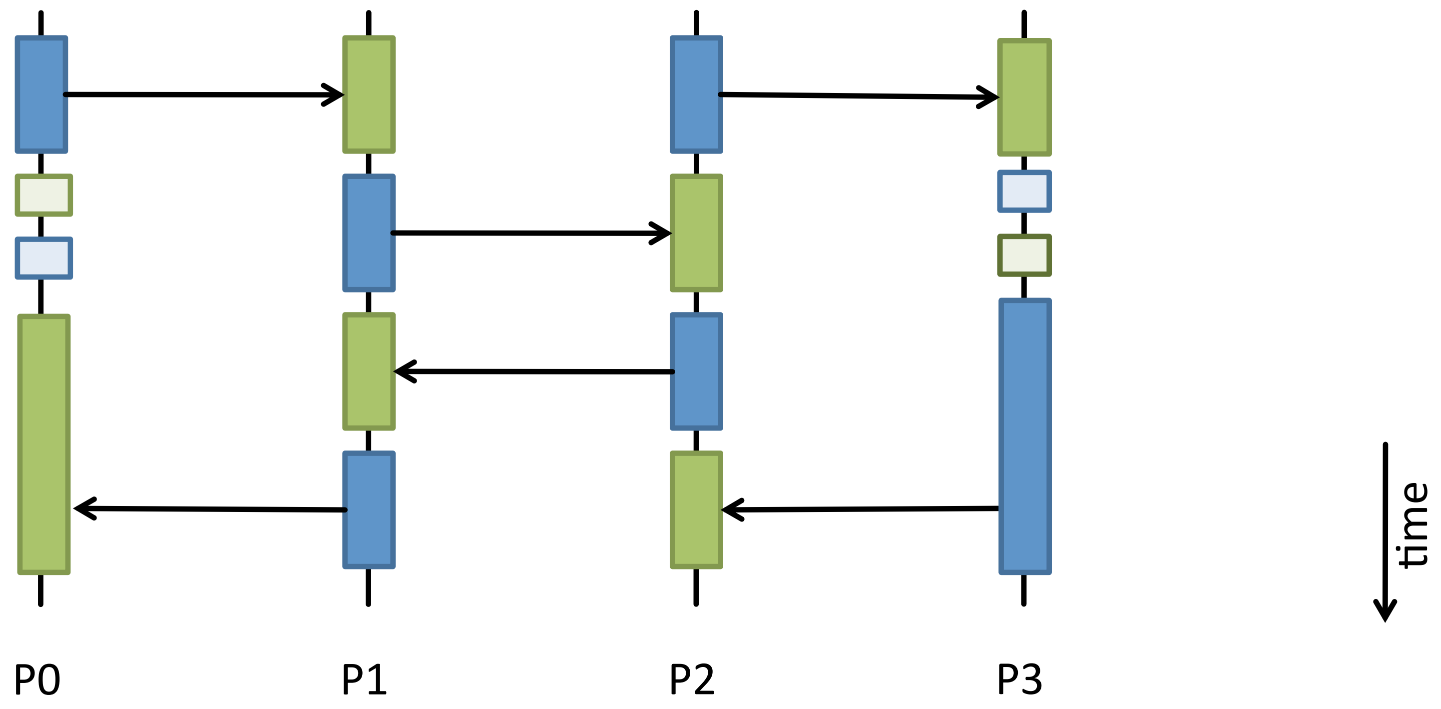
Pairing sends and receives

Order sends and receives so that they are paired

- If one process is sending to another, the destination will do a receive that matches that send before doing a send of its own

```
if (mod( coord, 2 ) .eq. 0) then
  call MPI_SEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, ...)
  call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, ...)
  call MPI_SEND( a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, ...)
  call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, ...)
else
  call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, ...)
  call MPI_SEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, ...)
  call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, ...)
  call MPI_SEND( a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, ...)
endif
```

Pairing sends and receives (2)



Combined send and receive

- Pairing sends and receives can be difficult when the arrangement of processes is complex
  - Example: irregular grids
- Alternative: MPI_Sendrecv
  - Allows process to send and receive without worrying about deadlock from lack of buffering

```
      call MPI_SENDRECV(
&              a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0,
&              a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0,
&              comm1d, status, ierr )
      call MPI_SENDRECV(
&              a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1,
&              a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1,
&              comm1d, status, ierr )
```

MPI_Sendrecv

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
                 MPI_Datatype sendtype, int dest,
                 int sendtag,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype, int source,
                 int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

- Executes a blocking send and receive operation
- Both send and receive use the same communicator, but possibly different tags
- The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes
- The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads

Buffered sends

- MPI allows the programmer to provide a buffer into which data can be placed until it is delivered (or at least until it leaves the buffer)
- Buffer must be large enough to hold all messages that must be sent before the matching receives are called

```
double precision buffer(2*MAXNX+2*MPI_BSEND_OVERHEAD)
integer size
[...]
size = 2*MAXNX*8 + 2*MPI_BSEND_OVERHEAD*8
call MPI_BUFFER_ATTACH( buffer, size, ierr )
```

- Once the buffer is no longer needed, it can be detached

```
call MPI_BUFFER_DETACH( buffer, size, ierr )
```

Buffered sends (2)

- To use buffer, replace MPI_Send with MPI_Bsend

```
call MPI_BSEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, ...)
call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, ...)
call MPI_BSEND( a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, ...)
call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, ...)
```

- Remember that buffering may incur a performance penalty

C-binding of buffered sends

```
int MPI_Buffer_attach(void *buffer, int size)
```

- Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages. The size is given in bytes
- The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time

```
int MPI_Buffer_detach(void *buffer_addr, int *size)
```

- Detach the buffer currently associated with MPI
- Will block until all messages currently in the buffer have been transmitted
- Upon return, the user may reuse or deallocate the space taken by the buffer

```
int MPI_Bsend(void *buf, int count, MPI_Datatype
              datatype, int dest, int tag, MPI_Comm comm)
```

- Send in buffered mode

Further send modes

- Synchronous send (MPI_Ssend)
  - Will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send
  - Can be used to check a program's dependence on buffering
  - Caveat: MPI_Send may be implemented using synchronous mode
- Ready send (MPI_Rsend)
  - Same semantics as a standard send operation
  - The sender only provides additional information to the system –namely that a matching receive is already posted
  - Can save some overhead
- Same signatures as MPI_Send

Summary send modes

- Standard – Up to MPI to decide whether outgoing messages will be buffered
- Buffered – Send can be started whether or not a matching receive has been posted. May complete before a matching receive is posted
- Synchronous – Send can be started whether or not a matching receive was posted. Will complete successfully only if a matching receive is posted and the receive operation has started to receive the message
- Ready – Send may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined

There is only one receive mode, but it matches any of the send modes

Possible send protocols

- Ready send – The message is sent as soon as possible
- Synchronous send – The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message and the sender now sends the message
- Standard send – First protocol may be used for short messages (“eager” protocol). Second protocol for long messages
- Buffered send – The sender copies the message into a buffer and then sends it with a non-blocking send (using the same protocol as for standard send)

Non-blocking communication

- Motivation 1 – improve performance by overlapping communication and computation
  - Especially useful on systems where communication can be executed autonomously by intelligent communication controller
- Motivation 2 – avoid buffering by deferring completion of communication until receive operation specifies destination
  - Also allows the destination to be specified early in the program
- Semantic advantage – avoids deadlock problem

Sender
- Start send
- Do something else
- Complete send

Receiver
- Start receive
- Do something else
- Complete receive

Non-blocking send

- Request object used to determine whether an operation has completed
- Test returns immediately

```
     call MPI_ISEND( buffer, count, datatype, dest, tag, &
                     comm, request , ierr )
     <do something else>
10  call MPI_TEST( request, flag, status, ierr )
     if (.not. flag) goto 10
```

- Wait blocks until completion

```
     call MPI_ISEND( buffer, count, datatype, dest, tag, &
                     comm, request , ierr )
     <do something else>
     call MPI_WAIT ( request, status, ierr )
```

- Receive works analogously

Multiple completions

- MPI provides a way to wait for a collection of non-blocking operations to complete

```
integer statuses(MPI_STATUS_SIZE, 2), requests(2)

call MPI_IRECV( ..., requests(1) , ierr )
call MPI_IRECV( ..., requests(2) , ierr )
[...]
call MPI_WAITAll ( 2, requests, statuses, ierr)
```

- Also supported
  - Waiting for any of a collection (MPI_Waitany)
  - Waiting for some of a collection (MPI_Waitsome)
  - Testing for all, any, or some of a collection (MPI_Testall, MPI_Testany, MPI_Testsome)

Non-blocking exchange of ghost points

```
         integer status_array(MPI_STATUS_SIZE,4), ierr, req(4)
c
         call MPI_IRECV ( &
              a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, &
              comm1d, req(1), ierr )
         call MPI_IRECV ( &
              a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, &
              comm1d, req(2), ierr )
         call MPI_ISEND ( &
              a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, &
              comm1d, req(3), ierr )
         call MPI_ISEND ( &
              a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, &
              comm1d, req(4), ierr )
c
         call MPI_WAITALL ( 4, req, status_array, ierr )
```

C-binding of non-blocking operations

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag,
              MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag,
              MPI_Comm comm, MPI_Request *request)

int MPI_Wait(MPI_Request *request, MPI_Status *status)

int MPI_Waitall(int count, MPI_Request *array_of_requests,
                 MPI_Status *array_of_statuses)

int MPI_Waitany(int count, MPI_Request *array_of_requests,
                 int *index, MPI_Status *status)

int MPI_Waitsome(int count, MPI_Request *array_of_requests,
          int *numcompl, int *indices, MPI_Status *statuses)
```

C-binding of non-blocking operations (2)

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)

int MPI_Testall(int count, MPI_Request *array_of_requests,
                int *flag, MPI_Status *array_of_statuses)

/* index identifies the one operation that completed */
int MPI_Testany(int count, MPI_Request *array_of_requests,
                int *index, int *flag,
                MPI_Status *status)

/* the first numcompl entries in indices are completed */
int MPI_Testsome(int count, MPI_Request *array_of_requests,
                 int *numcompl, int *indices,
                 MPI_Status *statuses)
```

Persistent requests

TECHNISCHE UNIVERSITÄT DARMSTADT

- Enable reuse of all communication parameters
  - Sending or receiving rank
  - Buffer address of payload
  - Size of message
  - Datatype of message
- Persistent requests are either
  - Inactive: no transfer is ongoing
  - Active: transfer is ongoing
- They are created and started by special calls, but completed by usual non-blocking completion calls

Initialization of persistent requests

With the initialization, an implementation can optimize some of the things it would otherwise have to decide on ‘on-the-fly’

```
int MPI_Send_init(void *buf, int count,
                  MPI_Datatype datatype, int dest, int tag,
                  MPI_Comm comm, MPI_Request *request)

int MPI_Recv_init(void *buf, int count,
                  MPI_Datatype datatype, int src, int tag,
                  MPI_Comm comm, MPI_Request *request)
```

Activation of persistent requests

- The user can activate single or multiple persistent requests at the same time
- Data is only transferred when requests are activated
- Buffers can be accessed while requests are inactive
  - After the init call, until the first start call
  - After a wait call, until the next start call

```
int MPI_Start(MPI_Request *request)

int MPI_Startall(int count, MPI_Request *request)
```

Summary

- Virtual topology
  - Description of which processes in a parallel program communicate with each other
  - Reasons to use MPI topology interface
  - Convenient process naming
  - Potentially more efficient
- Domain decomposition is common parallelization approach
- Different flavors of point-to-point communication
  - Blocking vs. non-blocking
  - 4 different send modes: standard, buffered, synchronous, ready
  - Persistent requests