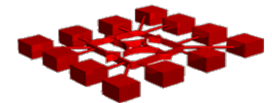
An Introduction to Message Passing and Parallel Programming with MPI

Ayesha Afzal <ayesha.afzal@fau.de>,

Georg Hager, Volker Weinberg, Markus Wittmann

Documentation:

http://www.mpi-forum.org/docs/



A. Skjellum, P. Bangalore, S. Herbert, R. Rabenseifner

Introduction to MPI: Agenda

- **Blocking point to point communication**
- **Nonblocking point to point communication**
- **Helper functions**
- **Collectives**
- **Virtual Topologies**

Point-to-Point Communication

Blocking

MPI_Send/MPI_Recv

```
// two process only example
int dst;
if (rank == 0) { dst = 1; } else { dst = 0; }

char * buffer = malloc(count * sizeof(char));

MPI_Send(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD);
MPI_Recv(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD,
                                          MPI_STATUS_IGNORE);
```

```
$ # tested on supermic
$ mpiexec -n 2 ./send 10       # OK
$ mpiexec -n 2 ./send 100      # OK
$ mpiexec -n 2 ./send 1000     # OK
$ mpiexec -n 2 ./send 10000    # OK
$ mpiexec -n 2 ./send 100000   # OK
$ mpiexec -n 2 ./send 1000000 # DEAD LOCK
```

Completion And Sending Modes

- **Completion**
  - When function call returns (for blocking p2p communciation)
  - Buffer can safely be reused

MPI function	type	completes when
MPI_Send	synchronous or buffered	depends on type
MPI_Bsend	buffered	buffer has been copied
MPI_Ssend	synchronous	remote starts receive
MPI_Recv	--	message was received

MPI_Bsend

- Always successful
- Do not care of time of delivery
- Completion does not involve action of other side

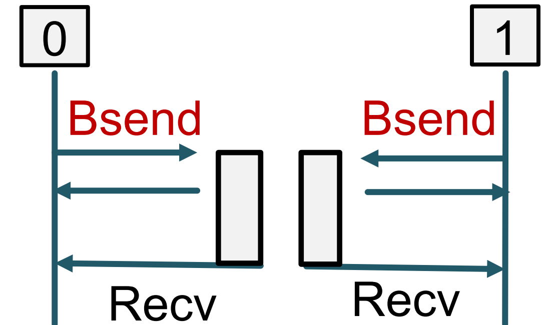
MPI_Ssend

- Completion is successful arrival of message
- Completion involves action of other side

Possible solutions for deadlock example

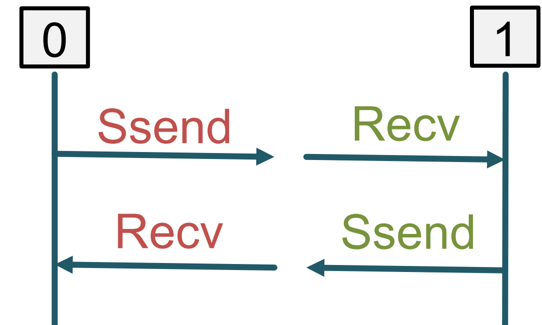
MPI_Bsend

```
// two process only example
int dst; if (rank == 0) { dst = 1; } else { dst = 0; }
char * buffer = malloc(count * sizeof(char));

// assuming buffer has been attached
MPI_Bsend(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD);
MPI_Recv(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
```



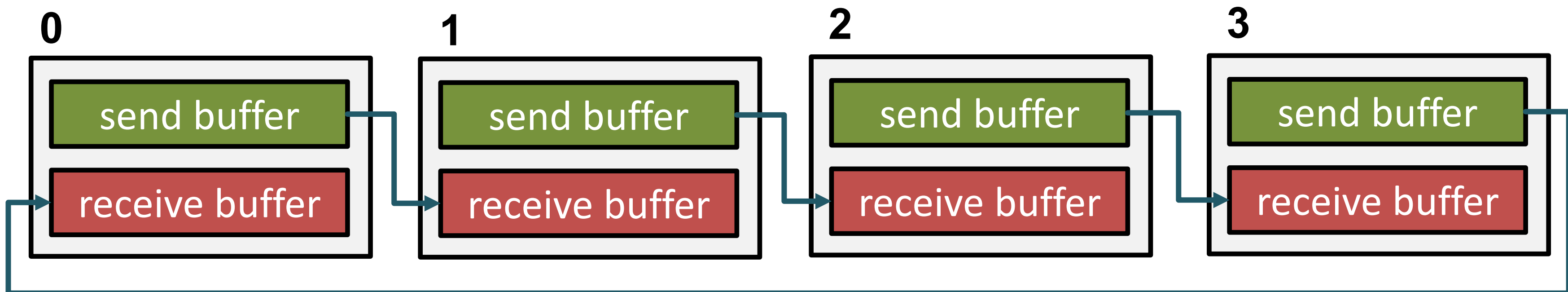
MPI_Ssend

```
// two process only example
int dst; if (rank == 0) { dst = 1; } else { dst = 0; }
char * buffer = malloc(count * sizeof(char));

if (rank == 0) {
  MPI_Ssend(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
  MPI_Recv(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
} else if (rank == 1) {
  MPI_Recv(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
  MPI_Ssend(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
}
```



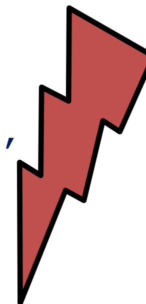
Point-to-Point Communication MPI_SENDRECV

- **Sending/Receiving at the same time is a common use case**
  - e.g.: shift messages, ring topologies, ghost cell exchange



- **MPI_Send/MPI_Recv pairs are not reliable:**

```
// Rank left from myself.
left = (rank - 1 + size) % size;
// Rank right from myself.
right = (rank + 1) % size;

MPI_Send(buffer_send, n, MPI_INT, right, 1,
   MPI_COMM_WORLD);
MPI_Recv(buffer_recv, n, MPI_INT, left, 1,
        MPI_COMM_WORLD, status);
```



How to avoid potential deadlock?

Point-to-Point Communication MPI_SENDRECV

- **Syntax: simple combination of send and receive arguments:**

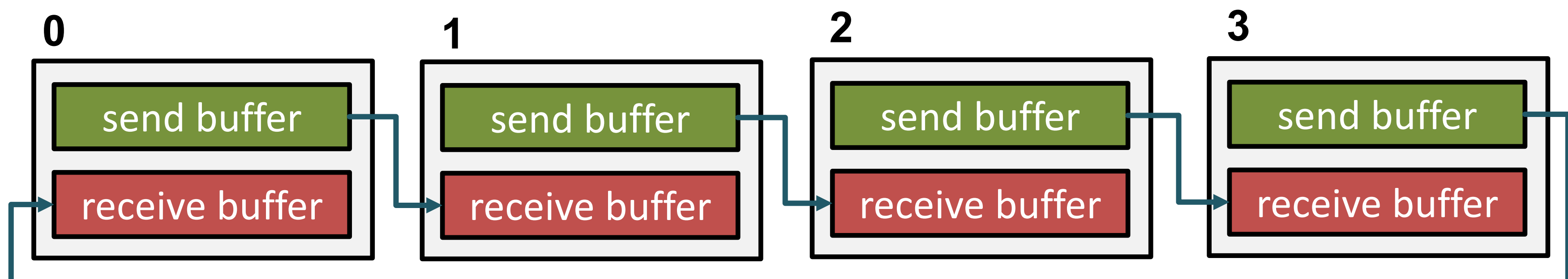
```
MPI_Sendrecv(
      buffer_send, sendcount, sendtype, dest,   sendtag,
      buffer_recv, recvcount, recvtype, source, recvtag,
      comm, MPI_Status * status)
```

- **MPI takes care no deadlocks occur**

- disjoint send/receive buffers
- can have different count & data type

```
// Rank left from myself.
left = (rank - 1 + size) % size;
// Rank right from myself.
right = (rank + 1) % size;

MPI_Sendrecv(
      buffer_send, n, MPI_INT, right, 0,
      buffer_recv, n, MPI_INT, left,  0, MPI_COMM_WORLD, status);
```

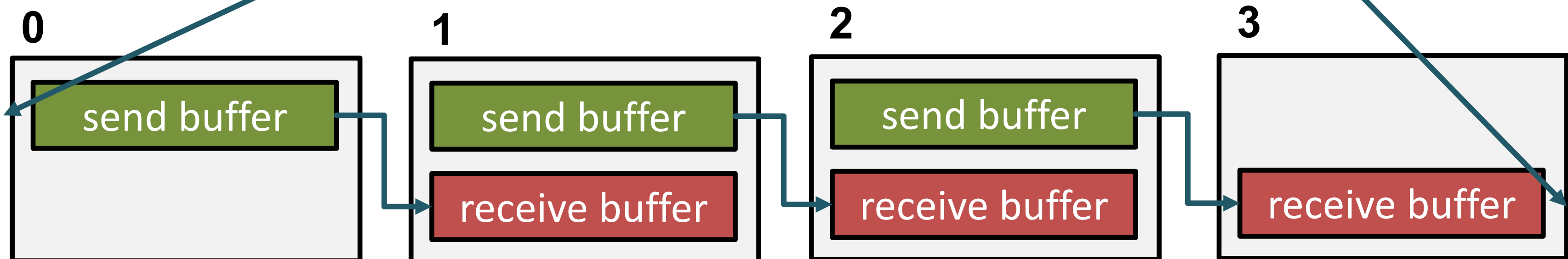
blocking call



Point-to-Point Communication MPI_SENDRECV

- **useful for open chains/non-circular shifts:**

```
// Rank left from myself.
left = rank - 1; if (left < 0) { left = MPI_PROC_NULL; }
// Rank right from myself.
right = rank + 1; if (right >= size) {right = MPI_PROC_NULL; }

MPI_Sendrecv(
        buffer_send, n, MPI_INT, right, 0,
        buffer_recv, n, MPI_INT, left,  0, MPI_COMM_WORLD, &status);
```



- **`MPI_PROC_NULL` as source/destination acts as no-op**
  - send/recv with `MPI_PROC_NULL` return as soon as possible
    buffers are not altered
- **`MPI_Sendrecv` matches with simple `*send/*recv` point-to-point calls**

Point-to-Point Communication MPI_SENDRECV_REPLACE

- **When only one single buffer is required:**

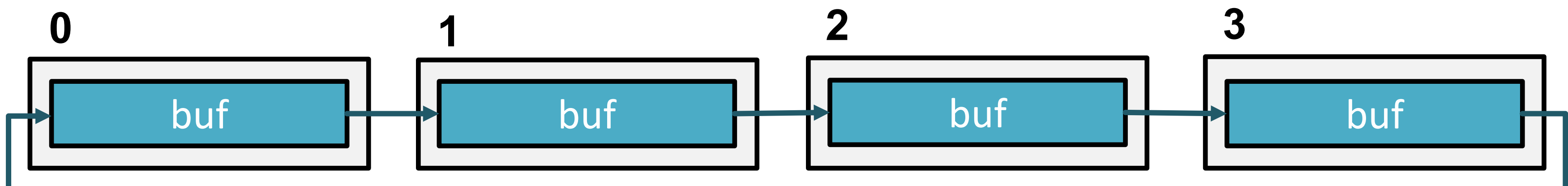
```
MPI_Sendrecv_replace(
   buf, count, datatype,
   dest, sendtag,
   source, recvtag,
   comm, MPI_Status * status)
```

same:
- buffer
- count
- data type

for send/receive

- **MPI ensures no deadlocks occur**

```
// Rank left from myself.
left = (rank - 1 + size) % size;
// Rank right from myself.
right = (rank + 1) % size;

MPI_Sendrecv_replace(
   buf, n, MPI_INT, right, 0, left,  0, MPI_COMM_WORLD, &status);
```



Blocking Point-to-Point Communication: Summary

- **Blocking MPI communication calls**
  - Operation completes when call returns
  - After completion: send/receive buffer can safely be reused

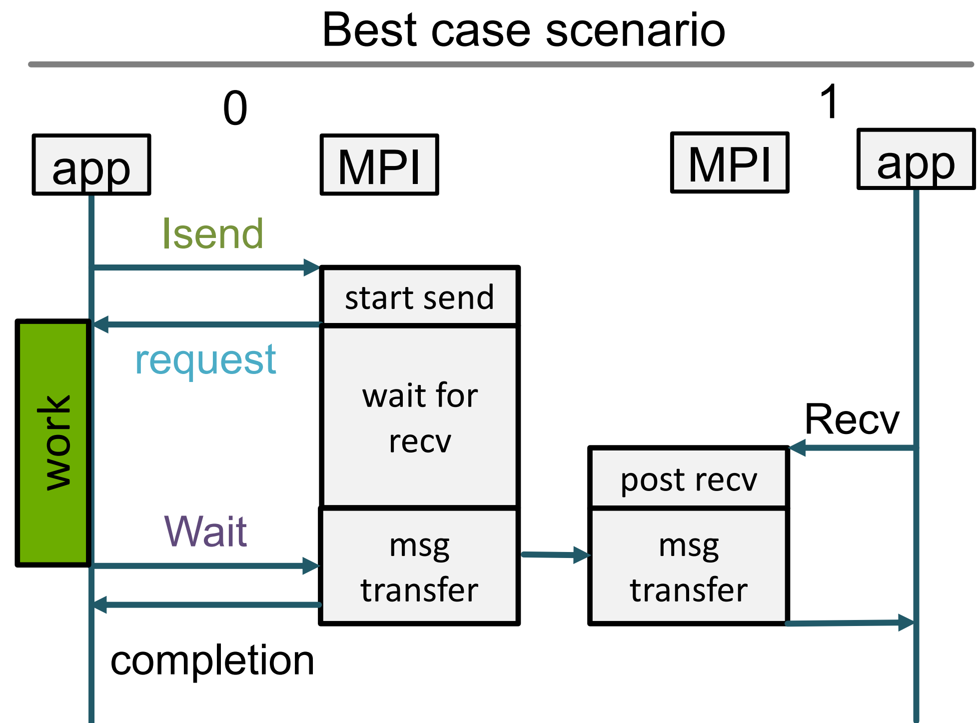
- **Available Send communication modes:**
  - Synchronous -- MPI_Ssend:
    - Guarantee receiving has started
    - Performance drawbacks, deadlock dangers
  - Buffered -- MPI_Bsend:
    - Completes after buffer is copied
    - User-provided buffer to save messages
    - Additional copy operations
  - Standard -- MPI_Send:
    - Behavior can be synchronous or buffered or depending on message length, no guarantee about that
- Other modes: MPI_Recv, MPI_Sendrecv and MPI_Sendrecv_replace

Point-to-Point Communication

Nonblocking

- **Advantages**
  - Avoid deadlocks
  - Possibility for overlapping communication with useful work
    - Best case: hide communication cost
    - Not guaranteed by the standard
  - Avoid idle time
  - Avoid synchronization

```
MPI_Request request;
MPI_Status status;

MPI_Isend(
  send_buffer, count, MPI_CHAR,
  dst, 0,
  MPI_COMM_WORLD, &request);

// do some work…
// do not use send_buffer

MPI_Wait(&request, &status)
```



Nonblocking Point-to-Point Communication: MPI_ISEND / MPI_IRECV

- **Standard nonblocking send/receive:**

```
MPI_Isend(sendbuf, count, datatype, dest,   tag,
          comm, MPI_Request * request)

MPI_Irecv(recvbuf, count, datatype, source, tag,
          comm, MPI_Request * request)
```

`request`: variable of type `MPI_Request`, will be associated with the corresponding operation

- **`MPI_Irecv` has no status argument**
  - obtained later during completion via `MPI_Wait*/MPI_Test*`

Nonblocking Send/Recv Example


```
// two process only example
int dst; if (rank == 0) { dst = 1; } else { dst = 0; }

MPI_Request requests[2];
MPI_Status  statuses[2]

MPI_Isend(send_buffer, count, MPI_CHAR, dst, 0,
        MPI_COMM_WORLD, &(requests[0]));
MPI_Irecv(recv_buffer, count, MPI_CHAR, dst, 0,
        MPI_COMM_WORLD, &(requests[1]));

// do some work…
// using send_buffer/rec_buffer is prohibited

MPI_Waitall(2, requests, statuses)
```

1. start operation

2. obtain request handle (new parameter)

buffer is only allowed to be reused after completion!

3. wait/test for completion

Nonblocking communication:

- Return from function != completion
- Each initiated operation must have a matching wait/test!

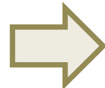
Blocking and Nonblocking Point-To-Point Communication

- **Blocking `send/recv` can be used with nonblocking ones**

- **Type synchronous/buffered affects completion**
  - Meaning: when `MPI_Wait / MPI_Test` return
  - Not when initiation, i.e. `MPI_I`…, returns

- **Nonblocking operation immediately followed by a matching wait is equivalent to the blocking operation**
  - Except for some compiler problems
  - Emulate blocking call via nonblocking operation:

```
MPI_Send(buf, …);
```



```
MPI_Request request;
MPI_Status status;

MPI_Isend(buf, …, &request);
MPI_Wait(&request, &status);
```

Nonblocking Point-to-Point Communication: Testing for Completion

- **MPI provides two test modes:**
  - `MPI_Wait`: Wait until the communication has been completed and buffer can safely be reused: Blocking
  - `MPI_Test`: Return TRUE (FALSE) if the communication has (not) completed: Nonblocking

© PPHPS 2010 - 2024, NHR@FAU, LRZ

Nonblocking Point-to-Point Communication: Test for Completion

- **Test one communication handle for completion:**

```
MPI_Wait(MPI_Request * request,
         MPI_Status * status);

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

`request:` request handle of type **`MPI_Request`**

`status`: status object of type **`MPI_Status`** (cf. **`MPI_Recv`**)

`flag`: variable of type **`int`** to test for success

Nonblocking Point-to-Point Communication: Test for Completion

MPI_Wait

```
MPI_Request request;
MPI_Status status;


MPI_Isend(
  send_buffer, count, MPI_CHAR,
  dst, 0, MPI_COMM_WORLD, &request);


// do some work…
// do not use send_buffer

MPI_Wait(&request, &status)

// use send_buffer
```

MPI_Test

```
MPI_Request request;
MPI_Status status;
int flag;

MPI_Isend(
  send_buffer, count, MPI_CHAR,
  dst, 0, MPI_COMM_WORLD, &request);

do {
    // do some work…
    // do not use send_buffer
    MPI_Test(&request, &flag, &status);
} while (!flag);

// use send_buffer
```

© PPHPS 2010 - 2024, NHR@FAU, LRZ

Nonblocking Point-to-Point Communication: Test for Completion

- MPI can handle multiple communication requests
- `Wait/Test` for completion of multiple requests:

```
MPI_Waitall(int count, MPI_Request requests[],
            MPI_Status statuses[]);

MPI_Testall(int count, MPI_Request requests[],
            int *flag, MPI_Status statuses[]);
```

- Waits for/Tests if all provided requests have been completed

```
MPI_Request requests[2];
MPI_Status  statuses[2];

MPI_Isend(send_buffer, …, &(requests[0]));
MPI_Irecv(recv_buffer, …, &(requests[1]));
// do some work…
MPI_Waitall(2, requests, statuses)
// Isend & Irecv have been completed
```

array

number of elements in the arrays

Nonblocking Point-to-Point Communication: Test for Completion

- `Wait/Test` for completion of multiple requests:

```
MPI_Waitany(int count, MPI_Request requests[],
            int * idx, MPI_Status * status);

MPI_Testany(int count, MPI_Request requests[],
            int * idx, int * flag,
            MPI_Status * status);
```

- Waits for/Tests if one request has been completed

```
MPI_Request requests[2];
MPI_Status  status;
int finished = 0;

MPI_Isend(send_buffer, …, &(requests[0]));
MPI_Irecv(recv_buffer, …, &(requests[1]));
do {
  // do some work…
  MPI_Testany(2, requests, &idx, &flag, &status);
  if (flag) { ++finished; }
} while (finished < 2)
```

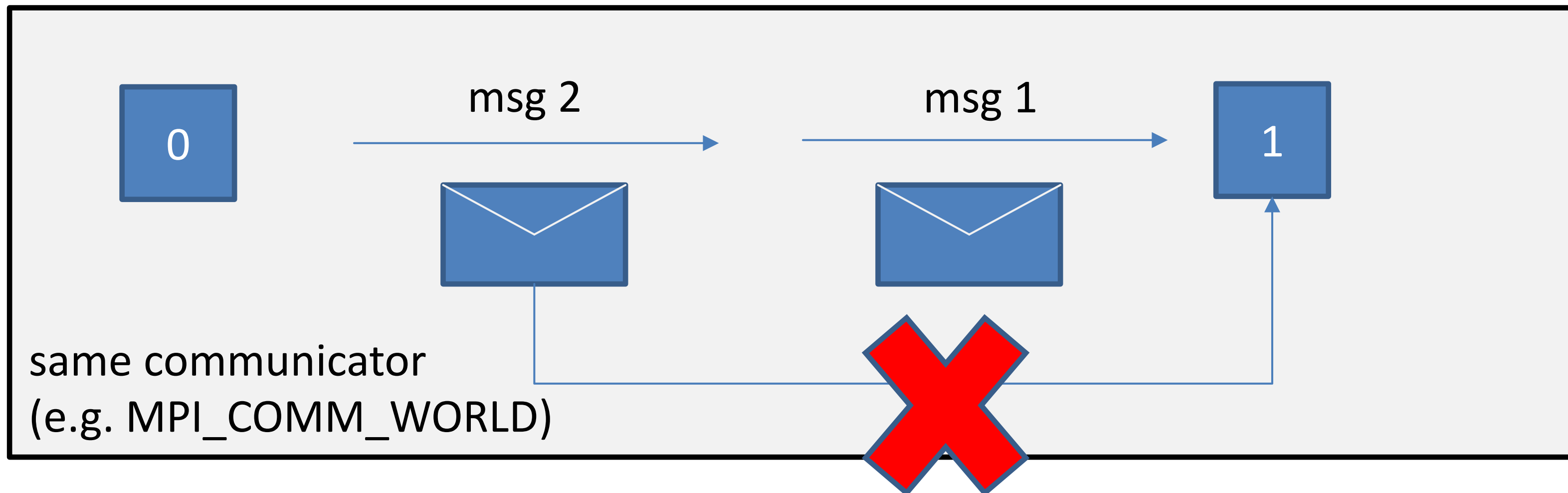
- completed requests are automatically set to `MPI_REQUEST_NULL`
- completed request `requests[idx]`

© PPHPS 2010 - 2024, NHR@FAU, LRZ

Helper functions
and
Semantics

Semantics

- **Message order preservation (guaranteed inside a communicator)**



Useful MPI Calls: MPI_GET_PROCESSOR_NAME

- Return a string to identify the hardware the process is running on

```
MPI_Get_processor_name(char * name, int * rlen);
```

- Typically the hostname of the compute node, but any arbitrary string is possible

```
char name[MPI_MAX_PROCESSOR_NAME];
int rlen;

MPI_Get_processor_name(name, &rlen);
printf("rank %d runs on %s.\n", rank, name);

# SuperMIC Output from  mpiexec -n 2./a.out
rank  0  runs on i01r13a06.
rank  1  runs on i01r13a06.
```

Useful MPI Calls: MPI_WTIME

- **Returns seconds since one point in past time**

```
double MPI_Wtime()
```

- **Use only for computation of time differences**

```
time_start = MPI_Wtime()
// …working…
duration = MPI_Wtime() – time_start
```

- **Returns time resolution in seconds,**

```
double MPI_Wtick()
```

  - e.g. if resolution is 1ms `MPI_Wtick()` returns `1e-3`
- **No `ierror` argument in Fortran**
- **Typically clocks from different ranks are not synchronized**

Useful MPI Calls: MPI_ABORT

- `MPI_ABORT` forces an MPI program to terminate:

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

- Aborts all processes in communicator
- `errorcode` will be handed as exit value to calling environment
- Safe and well-defined way of terminating an MPI program (if implemented correctly)

- In general, if something unexpected happens, try to shut down your MPI program the standard way (`MPI_Finalize()`)

Collective Communication in MPI

Operations including all ranks of a communicator

ALL RANKS MUST CALL THE FUNCTION

- **Blocking calls: buffer can be reused after return**
- **Nonblocking calls with MPI-3.0: buffer can be used after completion (`MPI_Wait*/MPI_Test*`)**
- **Cannot interfere with point-to-point communication**
  - **Completely separate modes of operation!**
- **Data type matching**
- **No tags**
- **Sent message must fill receive buffer (count is exact)**
- **Typically MPI libraries provide optimized implementations for operations**
- **May or may not synchronize the processes**
- **Types:**
  - Synchronization (barrier)
  - Data movement (broadcast, scatter, gather, all to all)
  - Collective computation (reduction)

Synchronization

- **Explicit synchronization of all ranks from specified communicator**

  ```
  MPI_Barrier(comm)
  ```

- **Ranks only return from call after every rank has called the function**

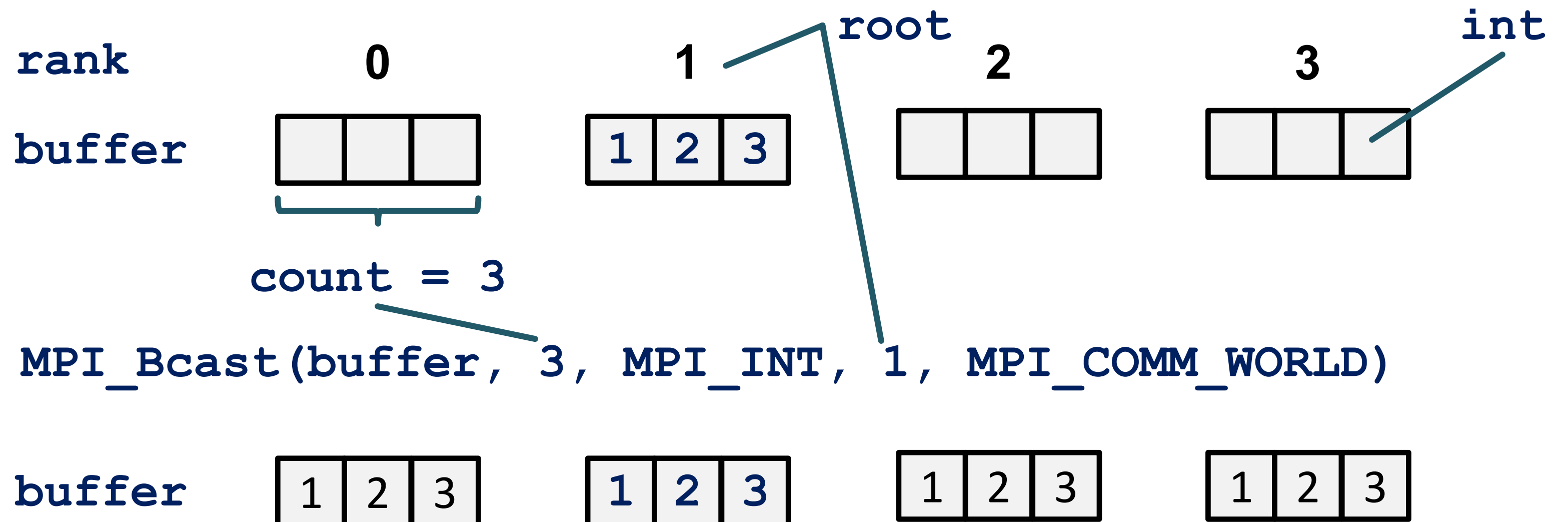
- `MPI_Barrier` **rarely needed, most of the time for debugging, e.g. to make sure every rank has reached a certain point in the application**

© PPHPS 2010 - 2024, NHR@FAU, LRZ

- **send buffer from one to all ranks**

```
MPI_Bcast(buf, count, datatype, int root, comm)
```

`root:` **rank from which data should be taken, typically 0, but everyone is allowed**

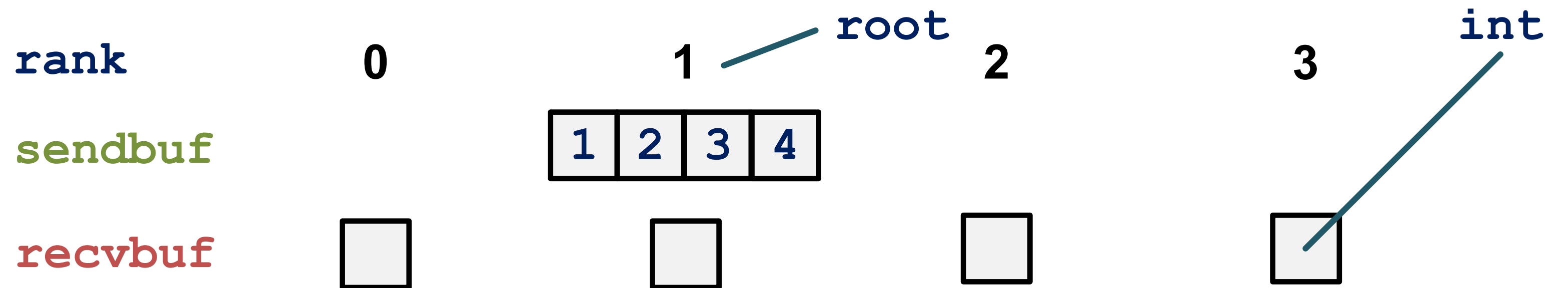


Collective Communication
Scatter

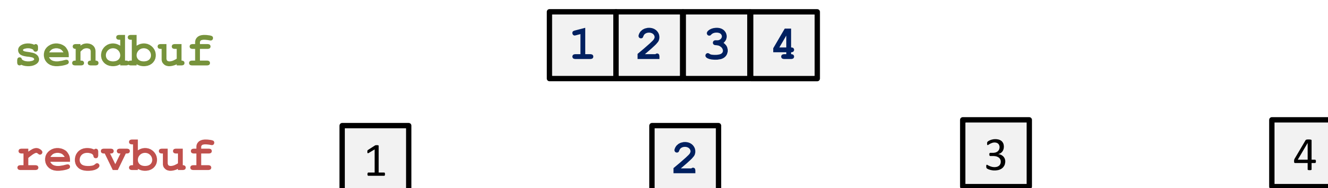
- Send the *ith* chunk to the *ith* rank

```
MPI_Scatter(sendbuf, sendcount, sendtype,
            recvbuf, recvcount, recvtype,
            root, comm)
```

- In general `sendcount = recvcount`
- `sendbuf` is ignored on non-root ranks




```
MPI_Scatter(sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT,
            root, MPI_COMM_WORLD)
```

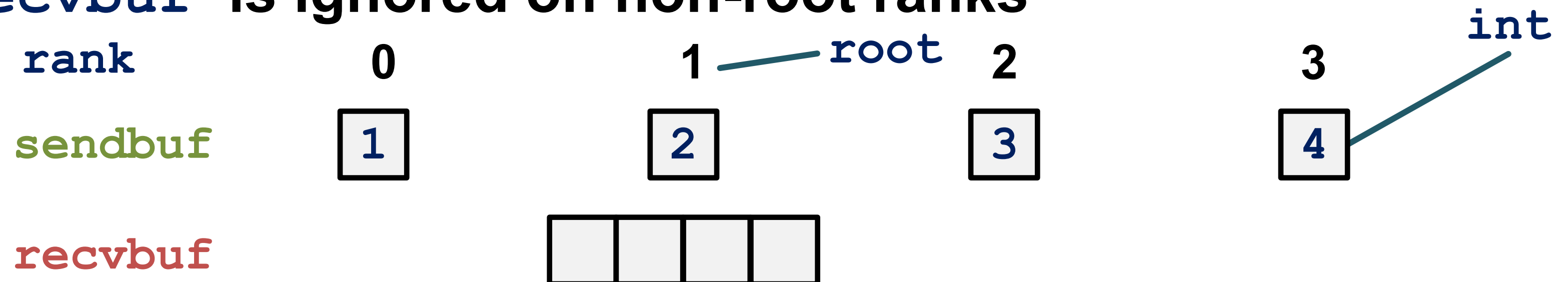


Collective Communication
Gather

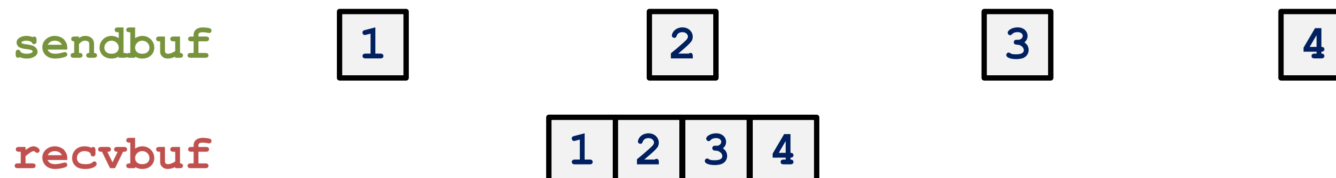
- **Receive a from each rank and place *ith* rank's msg at *ith* position in receive buffer**

```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype,
           root, comm)
```

- **In general `sendcount = recvcount`**
- **`recvbuf` is ignored on non-root ranks**




```
MPI_Gather(sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT,
           root, MPI_COMM_WORLD)
```



Collective Communication
MPI_ALLGATHER

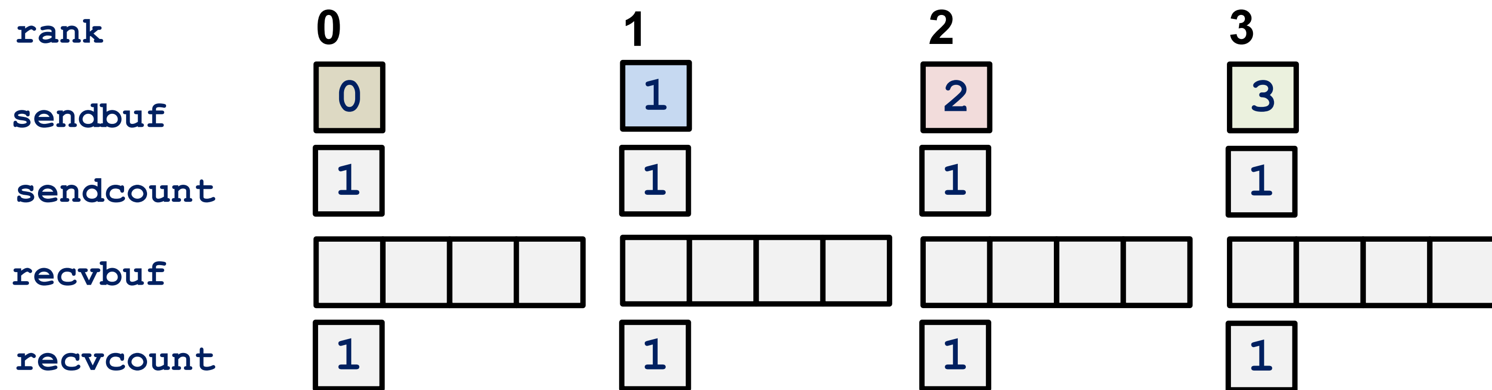
- Gather data from all ranks and broadcast it

```
MPI_Allgather(sendbuf, sendcount, sendtype,
              recvbuf, recvcount, recvtype,
              comm)
```

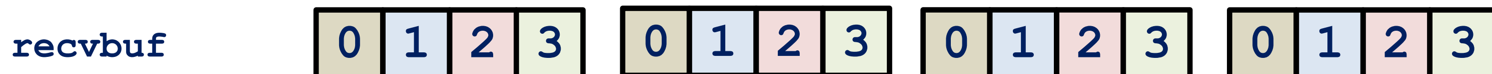
- In general `sendcount = recvcount`
- No `MPI_Allscatter`
- MPI library has more possibilities for optimization than manual `gather/bcast:`

```
MPI_Gather() with root = i
MPI_Bcast() with root = i
```

- `MPI_Allgather`: **Gather data from all ranks and broadcast it**



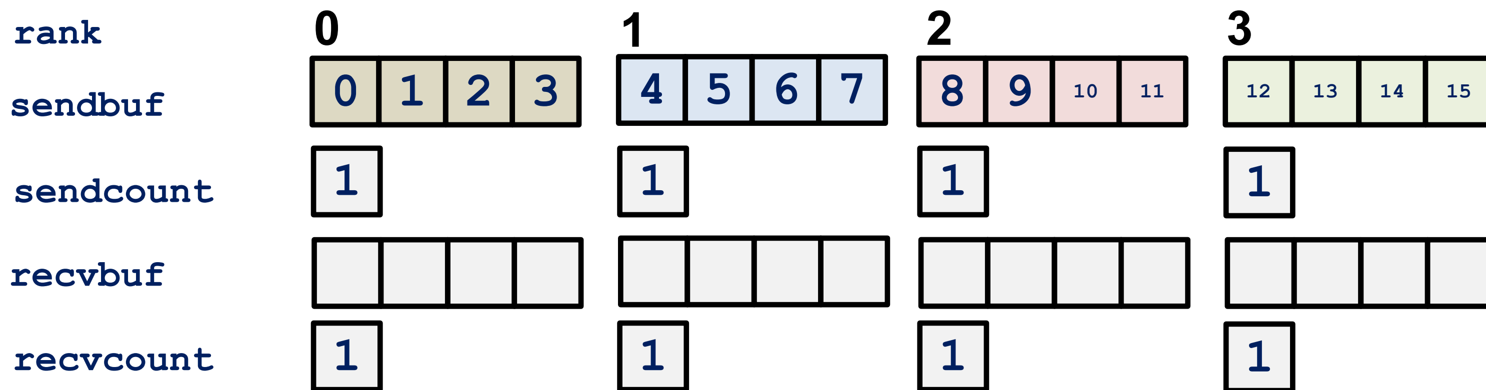
`MPI_Allgather()` **(no root required)**



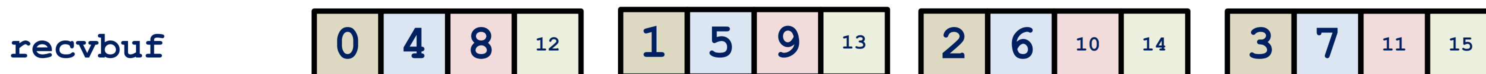
Collective Communication
MPI_ALLTOALL

- `MPI_Alltoall`: For all ranks, send ith chunk to ith rank

```
MPI_Alltoall(sendbuf, sendcount, sendtype,
             recvbuf, recvcount, recvtype,
             comm)
```



`MPI_Alltoall()` (no root required)

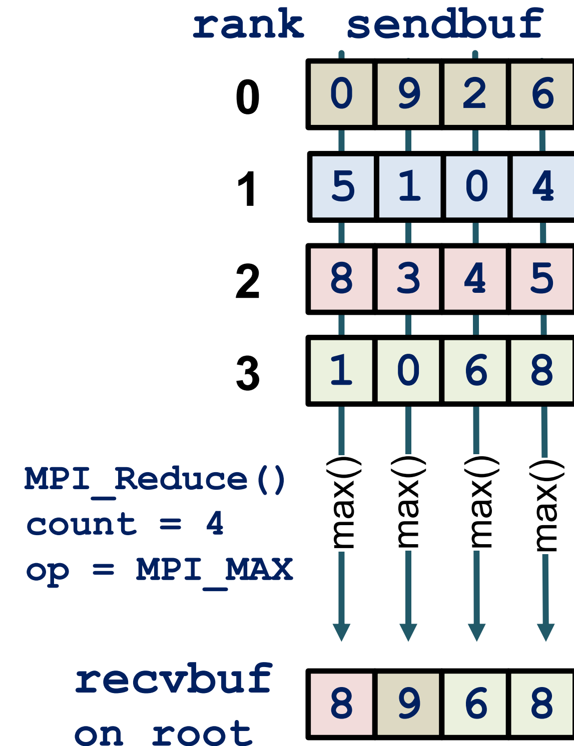


Global Operations Syntax

- Compute results over distributed data

```
MPI_Reduce(sendbuf, recvbuf, count, datatype,
           MPI_Op op, root, comm)
```

- Result in `recvbuf` only on `root` process available
- Perform operation on all `count` elements of an array with `count > 1`
- If all ranks require result, use `MPI_Allreduce`



Global Operations
Predefined Operations

Name	Operation	Name	Operation
MPI_SUM	Sum	MPI_PROD	Product
MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_LAND	Logical AND	MPI_BAND	Bit-AND
MPI_LOR	Logical OR	MPI_BOR	Bit-OR
MPI_LXOR	Logical XOR	MPI_BXOR	Bit-XOR
MPI_MAXLOC	Maximum+ Position	MPI_MINLOC	Minimum+ Position

- **If the 12 predefined ops are not enough, use `MPI_Op_create/MPI_Op_free` to create own ones**
- **MPI assumes that the operations are associative**
- **Be careful with floating point operations, as they may be not associative because of rounding errors**

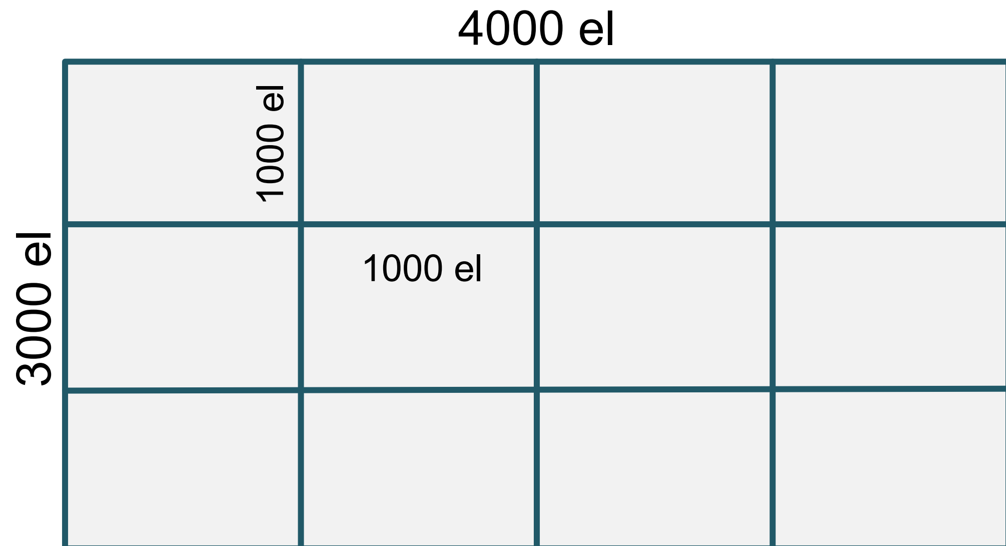
Virtual Topologies

a multi-dimensional process naming scheme

Virtual Topologies

- Convenient process naming
- Naming scheme to fit the communication pattern
- Simplifies writing of code
- Can allow MPI to optimize communications

distribute
2-D array of
4000 x 3000 el
equally on 12 ranks



- Let MPI map ranks to coordinates
- User: map array segments to ranks

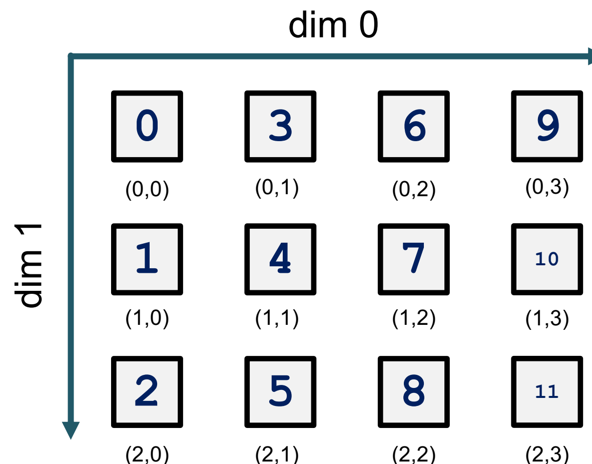
MPI_Cart_create

- **Create new communicator accompanied by Cartesian topology**

```
MPI_Cart_create(MPI_Comm oldcomm,
    ndims, int dims[], int periods[],
    int reorder, MPI_Comm * cart_comm)
```

`ndims:` number of dimensions
`dims:` array with `ndims` elements,
`dims[i]` specifies the number of ranks in dimension `i`
`periods:` array with `ndims` elements,
`periods[i]` specifies if dimension `i` is periodic
`reorder:` allow rank of `oldcomm` to have a different rank in `cart_comm`

```
ndims   = 2
dims    = {4, 3}
periods = {0, 0}
reorder = 0
```



MPI_Cart_create

- **Create new communicator accompanied by Cartesian topology**

```
MPI_Cart_create(MPI_Comm oldcomm,
    ndims, int dims[], int periods[],
    int reorder, MPI_Comm * cart_comm)
```

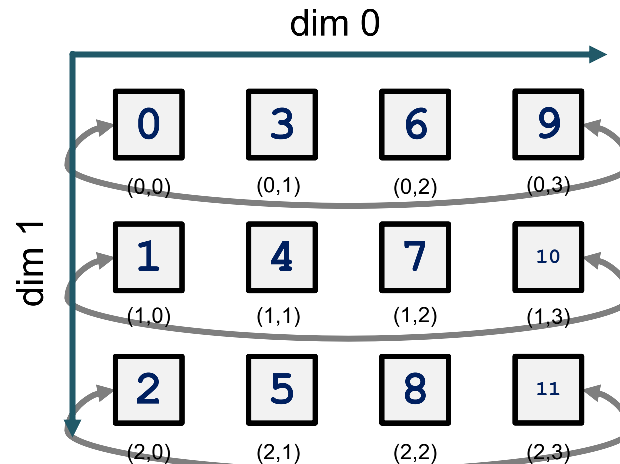
ndims: number of dimensions

dims: array with **ndims** elements,
dims[i] specifies the number of ranks in dimension **i**

periods: array with **ndims** elements,
periods[i] specifies if dimension **i** is periodic

reorder: allow rank of **oldcomm** to have a different rank in **cart_comm**

```
ndims   = 2
dims    = {4, 3}
periods = {1, 0}
reorder = 0
```



MPI_Cart_coords, MPI_Cart_rank

- **Retrieve rank in new Cartesian communicator**
  `MPI_Comm_rank(cart_comm, &cart_rank)`

- **Map rank → coordinates**
  `MPI_Cart_coords(comm, rank, int maxdims, int coords[])`

  `rank:` any rank which is part of Cartesian communicator `comm`
  `coords:` array of `maxdims` elements, receives the coordinates for `rank`

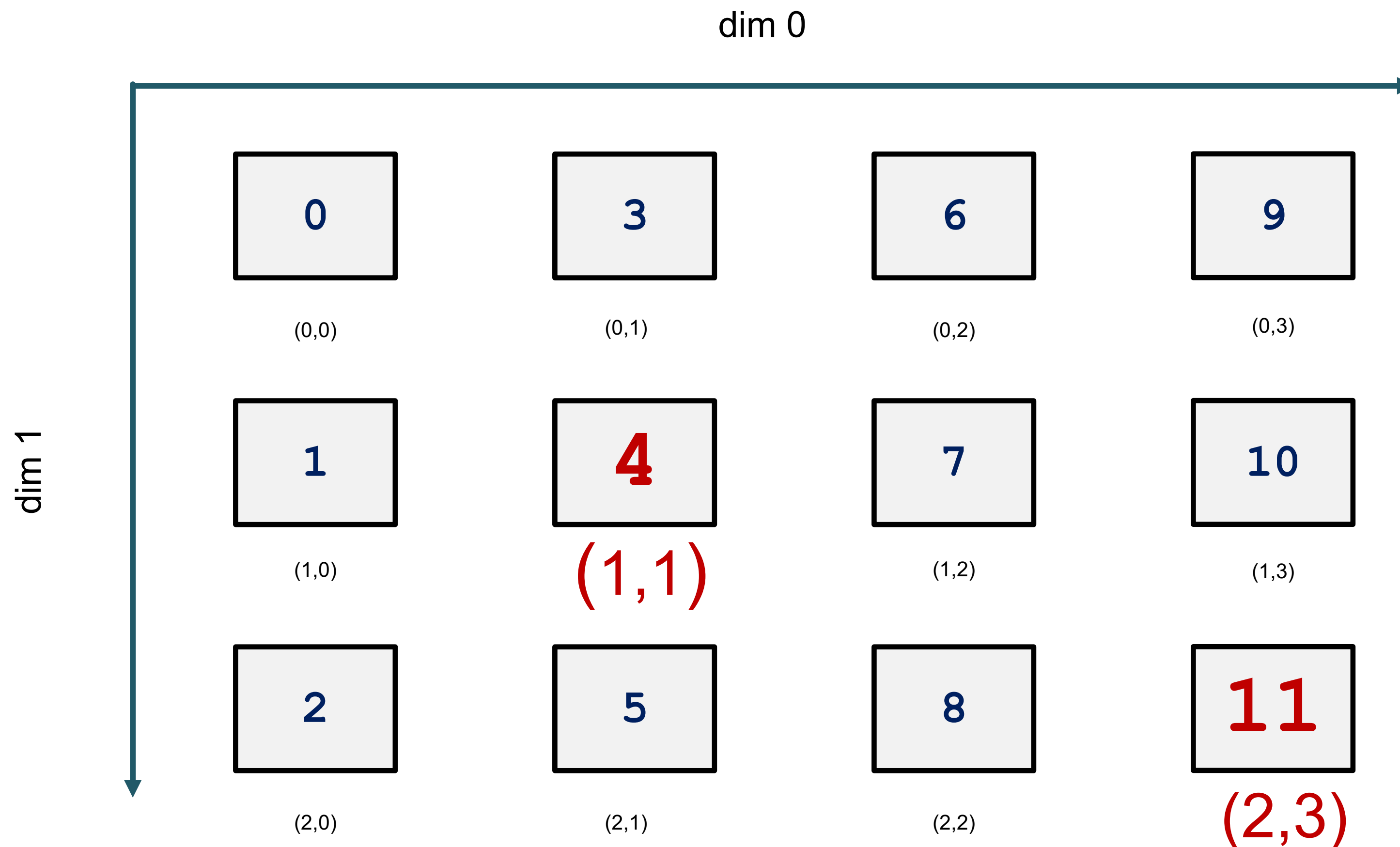
- **Map coordinates → rank**
  `MPI_Cart_rank(comm, int coords[], int * rank)`

  `coords:` coordinates; if periodic in direction `i`, `coords[i]` are automatically mapped into the valid range, else they are erroneous

- **Where am I inside the grid?**

```
int coords[ndims];
MPI_Comm_rank(cart_comm, &cart_rank);
MPI_Cart_coords(cart_comm, cart_rank, ndims, coords);
```

- Example: 12 processes arranged on a 4 x 3 grid
- Column-major numbering
- Process coordinates begin with 0



MPI_Cart_shift

- **Sending/receiving from neighbors typical task in Cartesian topologies**

```
MPI_Cart_shift(cart, direction, disp,
               int * source_rank, int * dest_rank)
```

direction: dimension to shift
disp: offset to shift: > 0 shift in positive direction,
< 0 shift in negative direction
src/dst: returned ranks as input into **MPI_Sendrecv*** calls

for non-periodic dimensions **MPI_PROC_NULL** is returned on boundaries

Exampe: 4x3 process grid, periodic in 1st dimension, each process has a int value, which gets shifted

```
MPI_Cart_shift(cart_comm, 0, 1, &src, &dst);
MPI_Sendrecvreplace(&value, 1, MPI_INT,
        dst, 0, src, 0, cart_comm, …)
```

0	3	6	9
1	4	7	10
2	5	8	11

⇨

9	0	3	6
10	1	4	7
11	2	5	8

shift in 1st dimension, which is periodic

```
MPI_Cart_shift(cart_comm, 1, 1, &src, &dst);
MPI_Sendrecvreplace(&value, 1, MPI_INT,
        dst, 0, src, 0, cart_comm, …)
```

0	3	6	9
1	4	7	10
2	5	8	11

⇨

0	3	6	9
0	3	6	9
1	4	7	10

shift in 2nd dimension, which is non-periodic