Erlangen Regional Computing Center

UNIVERSITÄT GREIFSWALD
Wissen lockt. Seit 1456

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Winter term 2020/2021

Parallel Programming with OpenMP and MPI

Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald

Lecture 11: Advanced MPI: derived data types and virtual topologies, MPI performance pitfalls

High Performance Computing

Outline of course

- Basics of parallel computer architecture
- Basics of parallel computing
- Introduction to shared-memory programming with OpenMP
- OpenMP performance issues
- Introduction to the Message Passing Interface (MPI)
- Advanced MPI
- MPI performance issues
- Hybrid MPI+OpenMP programming

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

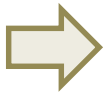
Derived data types in MPI

High Performance Computing

MPI data types: why?

Example: Root reads configuration and broadcasts it to all others

```
// root: read configuration from
// file into struct config
MPI_Bcast(&cfg.nx, 1, MPI_INT, …);
MPI_Bcast(&cfg.ny, 1, MPI_INT, …);
MPI_Bcast(&cfg.du, 1, MPI_DOUBLE,…);
MPI_Bcast(&cfg.it, 1, MPI_INT, …);
```



Want to do something like:

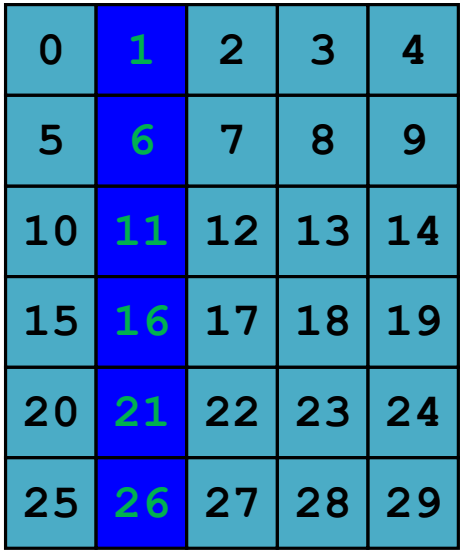
```
MPI_Bcast(
   &cfg, 1, <type cfg>,…);
```

```
MPI_Bcast(&cfg, sizeof(cfg),
          MPI_BYTE, ..)
```

is **not** a solution. Its not portable as no data conversion can take place

MPI data types: why?

- Example: Send column of matrix (noncontiguous in C):
  - Send each element alone?
  - Manually copy elements out into a contiguous buffer and send it?



Making an MPI data type

Three steps:

1. Construct with

```
MPI_Type_*(...);
```

2. Commit new data type with

```
MPI_Type_commit(MPI_Datatype * nt);
```

3. After use, deallocate the data type with

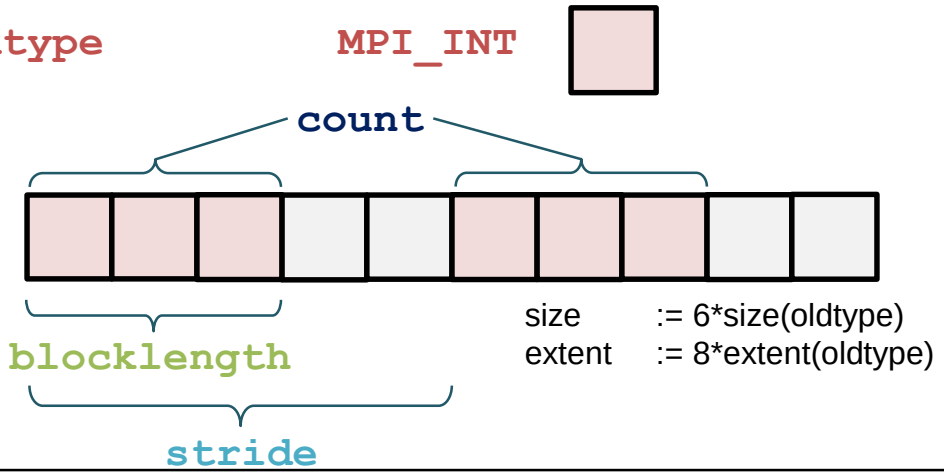
```
MPI_Type_free(MPI_Datatype * nt);
```

All local, non-collective calls

A flexible, vector-like type: MPI_Type_vector

```
MPI_Type_vector(int count, int blocklength, int stride,
                MPI_Datatype oldtype,
                MPI_Datatype * newtype);
```

count	2 (no. of blocks)
blocklength	3 (no. of elements in each block)
stride	5 (no. of elements b/w start of each block)
oldtype	MPI_INT



size := 6*size(oldtype)
extent := 8*extent(oldtype)

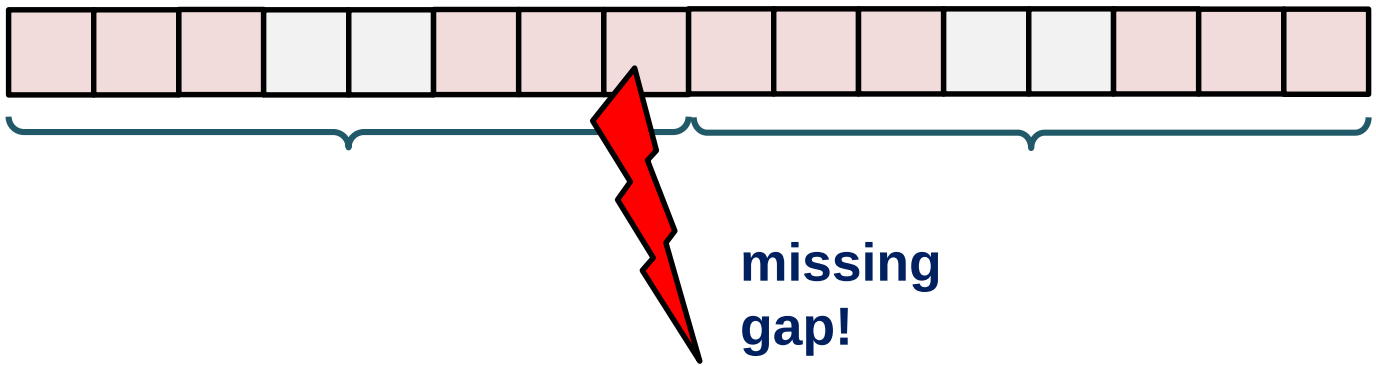
```
MPI_Datatype nt;
MPI_Type_vector(
2, 3, 5, MPI_INT, &nt);

MPI_Type_commit(&nt);
// use nt…
MPI_Type_free(&nt);
```

Caveat when using a type

- Caution: Concatenating such types in a send operation can lead to unexpected results!
- **`count`** argument to **`send`** and others must be handled with care:

`MPI_Send(buf, 2, nt,...)` with **`nt`** (newtype from prev. slide)



Derives type size and extent

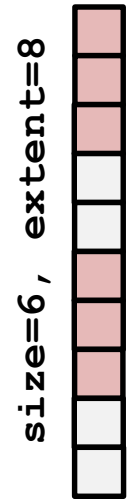
- Get the total **`size`** (in bytes) of datatype in a message

```
int MPI_type_size(MPI_Datatype newtype, int *size);
```

- Get the lower bound and the **`extent`** (span from the first byte to the last byte) of datatype

```
int MPI_type_get_extent(MPI_Datatype newtype,
                        MPI_Aint *lb,
                        MPI_Aint *extent);
```

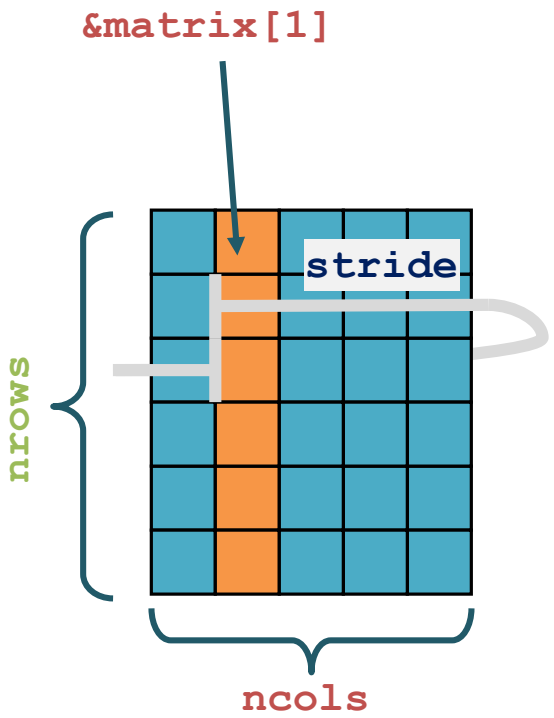
MPI type for memory addresses or offsets



- MPI allows to change the extent of a datatype
  - using **`lb_marker`** and **`ub_marker`**
  - does not affect the size or count of a datatype, and the message content
  - does affect the outcome of a replication of this datatype

Sending a column of a matrix in C

Row-major data layout in C → cannot use plain array




```
double matrix[30];
MPI_Datatype nt;

// count = nrows, blocklength = 1,
// stride = ncols
MPI_Type_vector(nrows, 1, ncols,
                MPI_DOUBLE, &nt);
MPI_Type_commit(&nt);

// send column
MPI_Send(&matrix[1], 1, nt, …);

MPI_Type_free(&nt);
```

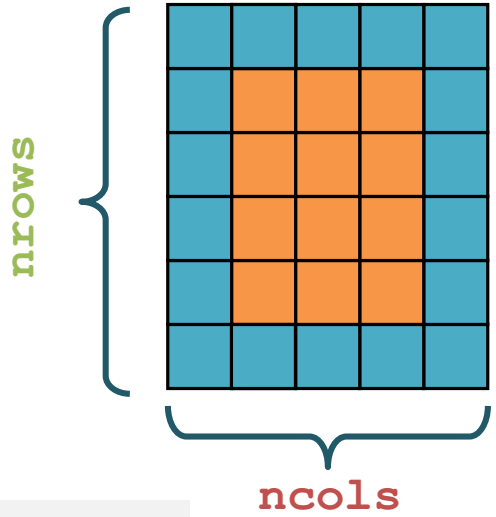
A sub-array type: MPI_Type_create_subarray

```
MPI_Type_create_subarray(int dims,
  int ar_sizes[], int ar_subsizes[], int ar_starts[],
  int order, MPI_Datatype oldtype, MPI_Datatype * newtype);
```

- **dims:** dimension of the array
- **ar_sizes:** array with sizes of array (dims entries)
- **ar_subsizes:** array with sizes of subarray (dims entries)
- **ar_starts:** start indices of the subarray inside array (dims entries), start at 0 (also in Fortran)
- **order**
  - row-major: **MPI_ORDER_C**
  - column-major: **MPI_ORDER_FORTRAN**

Example for a sub-array type: “bulk” of a matrix

dims	2
ar_sizes	{ncols, nrows}
ar_subsizes	{ncols-2, nrows-2}
ar_starts	{1, 1}
order	MPI_ORDER_C
oldtype	MPI_INT

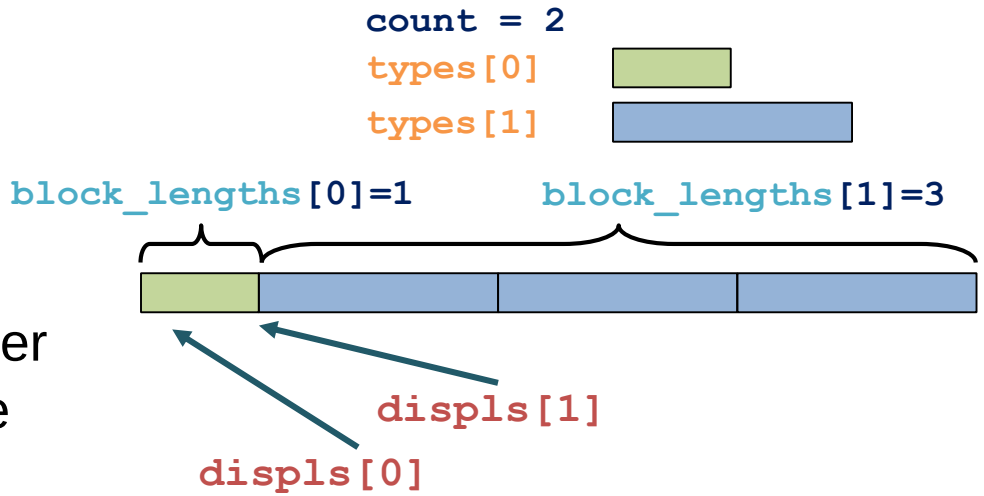



```
MPI_Type_create_subarray(dims, ar_sizes, ar_subsizes,
                         ar_starts, order, oldtype, &nt);
MPI_Type_commit(&nt);
// use nt...
MPI_Send(&buf[0], 1, nt, …); // etc.
MPI_Type_free(&nt);
```

Most flexible type: `MPI_Type_create_struct`

Describe blocks with arbitrary data types and arbitrary displacements

```
MPI_Type_create_struct(int count, int block_lengths[],
  MPI_Aint displs[], MPI_Datatype types[],
  MPI_Datatype * newtype);
```



The contents of `displs` are either the displacements in bytes of the block bases or MPI addresses

How to obtain and handle addresses?

```
MPI_Get_address(const void *location, MPI_Aint *address);
MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2);
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp);
```

- Example:

```
double a[100];
MPI_Aint a1, a2, disp;
MPI_Get_address(&a[0],  &a1);
MPI_Get_address(&a[50], &a2);
disp = MPI_Aint_diff(a2,a1);
```

  Result would usually be **`disp = 400`** (50 x 8)

- When using absolute addresses, set buffer address = **`MPI_BOTTOM`**

Derived data types: summary

- A flexible tool to communicate complex data structures in MPI
- Most important calls:

```
MPI_Type_vector                 (second simplest)
MPI_Type_create_subarray
MPI_Type_create_struct          (most advanced)
MPI_Type_commit/MPI_Type_free
MPI_Get_address,
  MPI_Aint_add, MPI_Aint_diff
MPI_Type_get_extent, MPI_Type_size
```

- Other useful features:
  `MPI_Type_contiguous`, **`MPI_Type_indexed`**,…
- Matching rule: send and receive match if specified basic datatypes match one by one, regardless of displacements
  - Correct displacements at receiver side are automatically matched to the corresponding data items

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

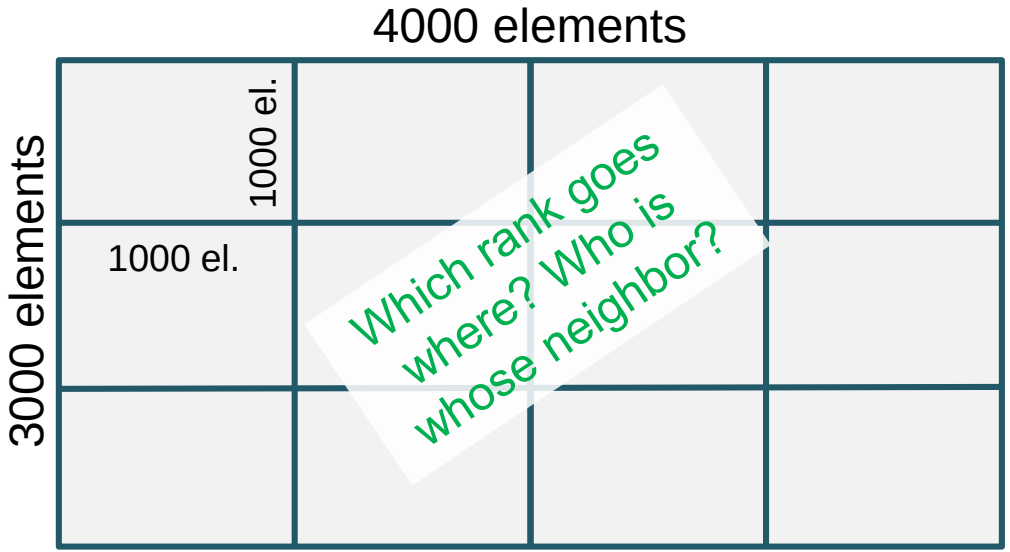
Virtual (Cartesian) topologies in MPI

High Performance Computing

A convenient process naming scheme for multi-dimensional problems

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

Example: distribute
2-D array of
4000 x 3000 elements
equally on 12 ranks



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

Creating a new Cartesian communicator

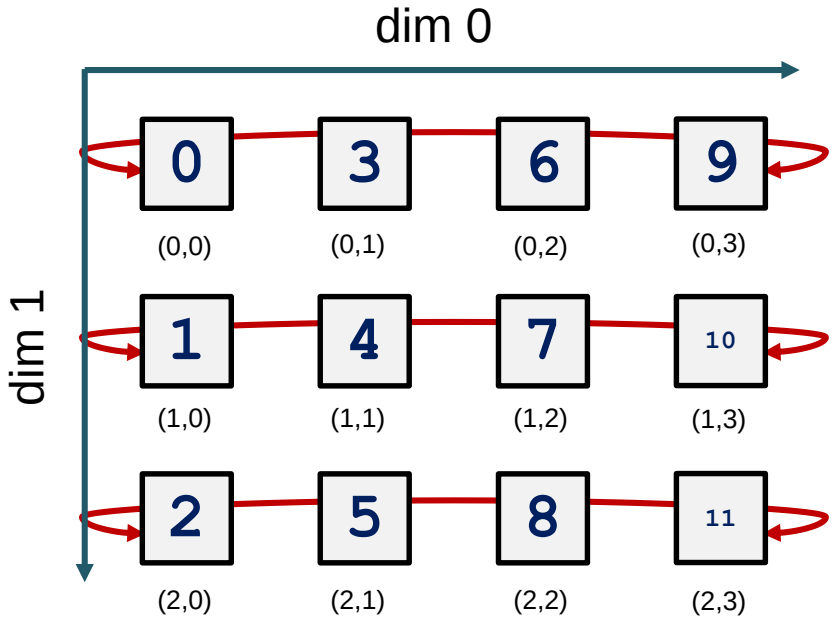
- Create new communicator attached to Cartesian topology

```
MPI_Cart_create(MPI_Comm oldcomm,
    int 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**

Cartesian topology example

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



Cartesian topology service functions

- Retrieve rank in new Cartesian communicator (“who am I in the grid?”)

```
MPI_Comm_rank(cart_comm, int *cart_rank);
```

- Map rank → coordinates (“where am I in the grid?”)

```
MPI_Cart_coords(cart_comm, rank, int maxdims, int coords[]);
```

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

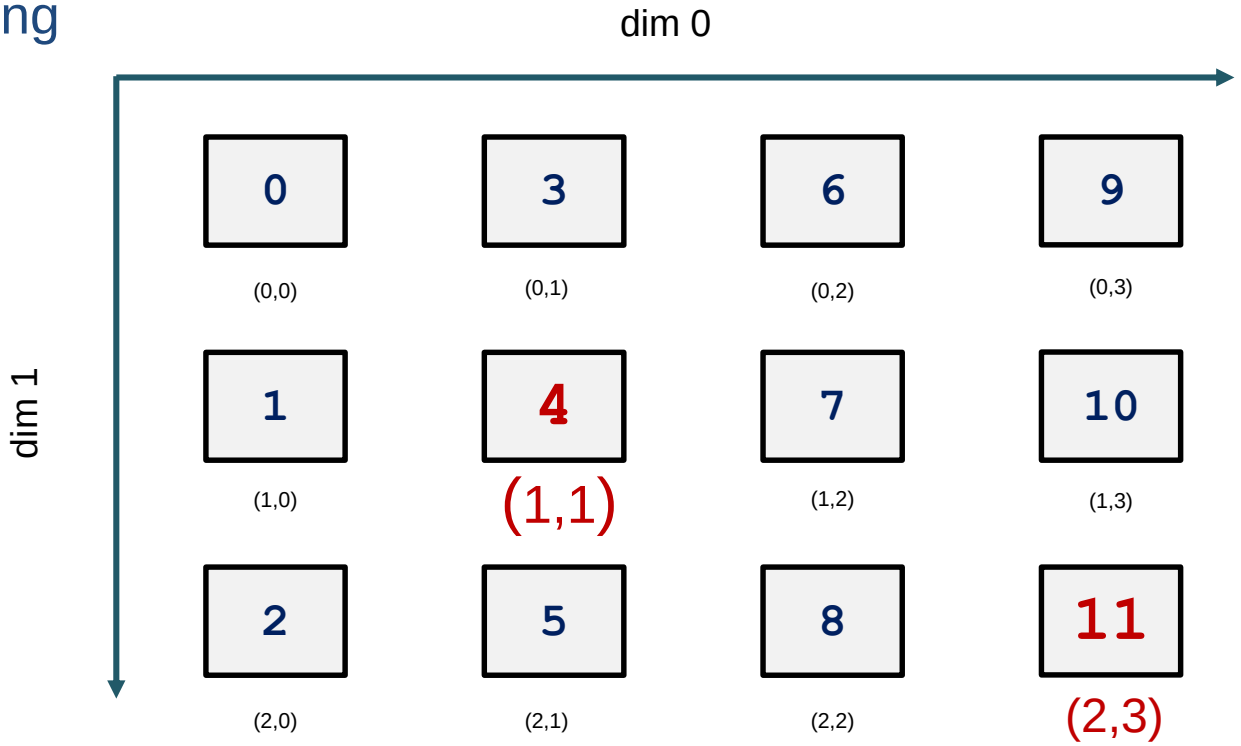
- Map coordinates → rank (“who is at that position?”)

```
MPI_Cart_rank(cart_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

Example

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



Next-neighbor communication

Sending/receiving from neighbors is a typical task in Cartesian topologies

```
MPI_Cart_shift(cart_comm, 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
source_rank/dest_rank: returned ranks as input into **MPI_Sendrecv*** calls

Next-neighbor communication

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

```
MPI_Cart_shift(cart_comm, 0, 1, &src, &dst);
MPI_Sendrecv_replace(&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

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

```
MPI_Cart_shift(cart_comm, 1, 1, &src, &dst);
MPI_Sendrecv_replace(&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

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Typical performance pitfalls in MPI

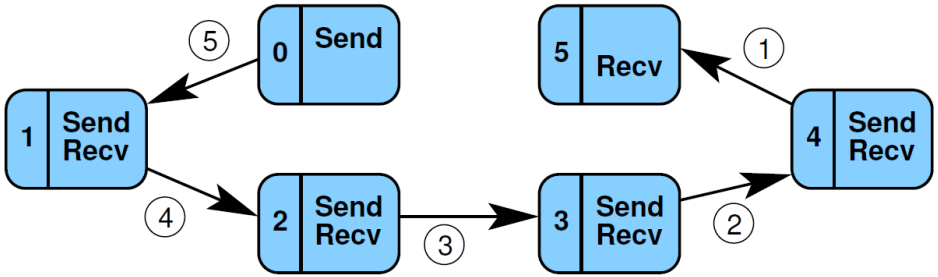
High Performance Computing

Implicit serialization and synchronization

- Very popular performance pitfall with MPI
- Consider linear shift in an open chain, e.g., each rank in the chain issues:

```
call MPI_Send(…,rank+1,…);
call MPI_Recv(…,rank-1,…);
```

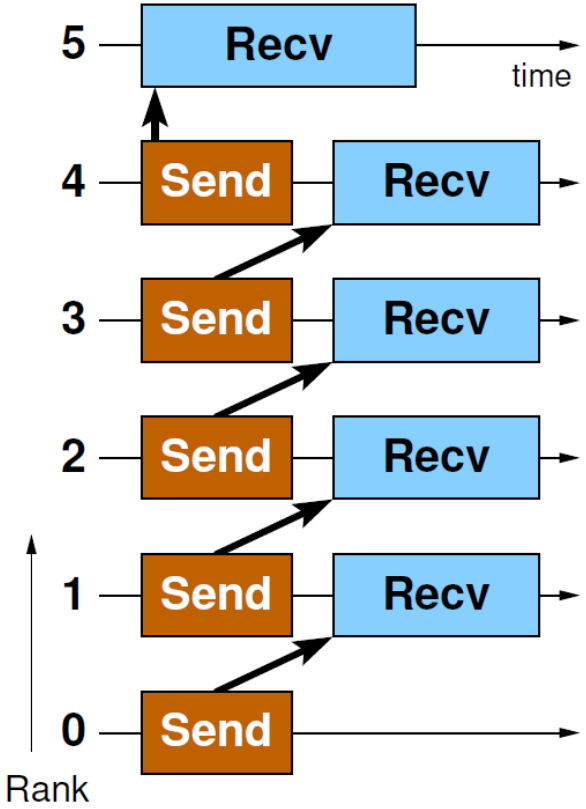
- First and last rank call **`MPI_Send`** and **`MPI_Recv`** only, respectively



- There is no danger of deadlocks but performance depends on implementation-specific parameters in the MPI library:
  - Buffered or synchronous **`MPI_Send`**
  - If synchronous **`MPI_Send`**: **Eager** or **Rendezvous** protocol?

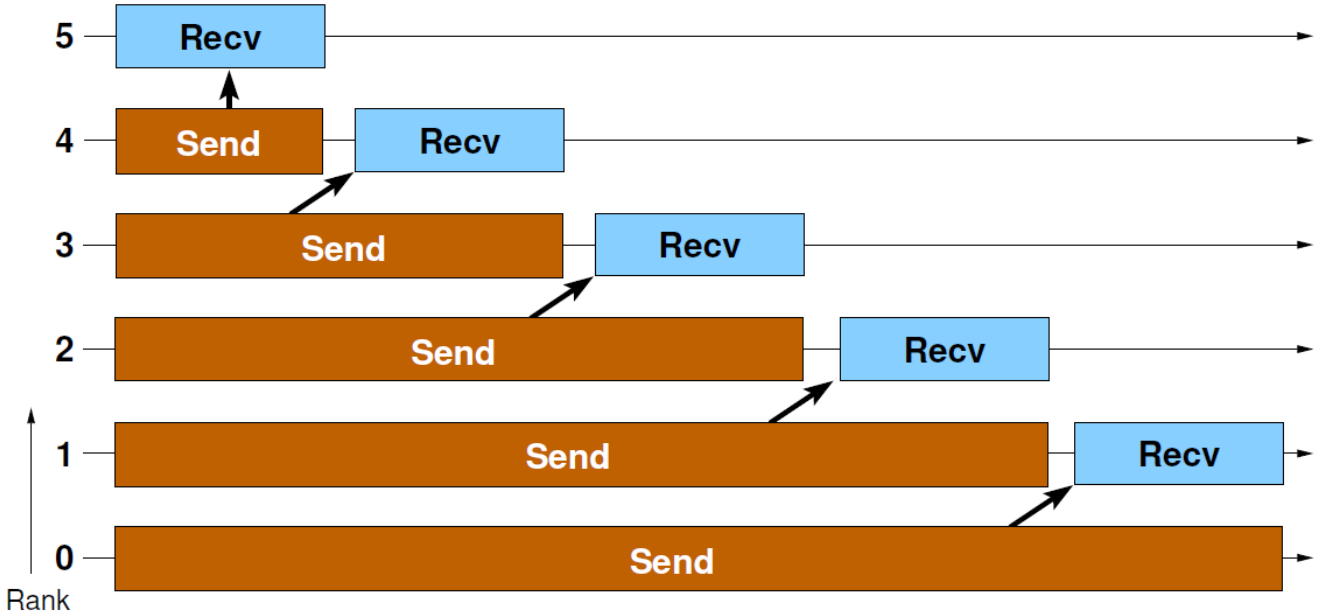
Implicit serialization and synchronization

- Best case scenario: **`MPI_Send`** operates in a buffered send mode
- **`MPI_Send`** returns after message is copied to a system buffer
- Send/Receive operations can be overlapped on nonblocking, bidirectional networks



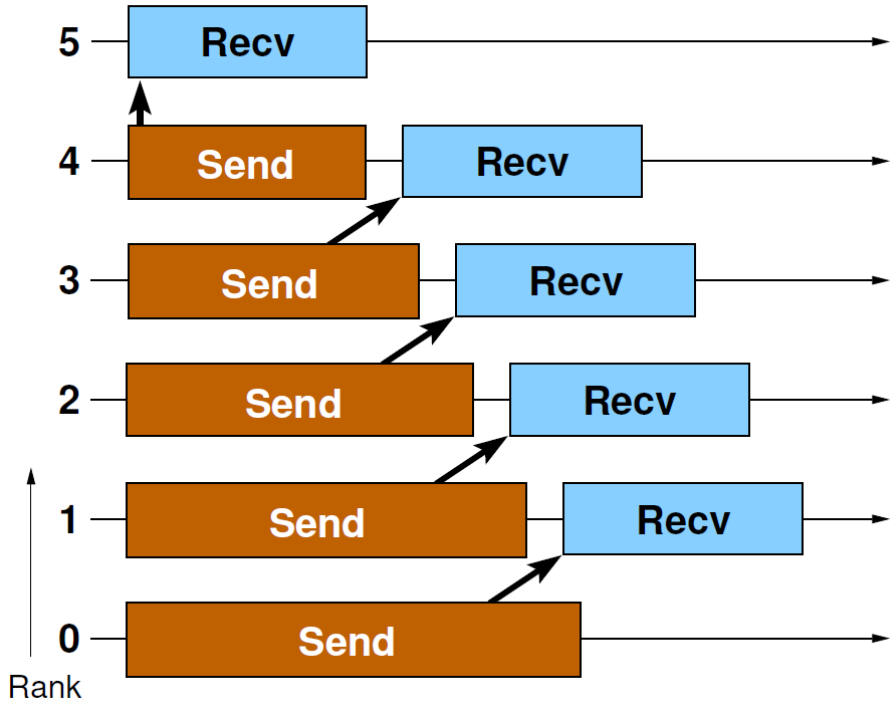
Implicit serialization and synchronization

- Worst case scenario 1: Synchronous send using the rendezvous protocol
- Rendezvous: Send operation blocks until complete message has been transferred!
- Serialization of all data transfers!



Implicit serialization and synchronization

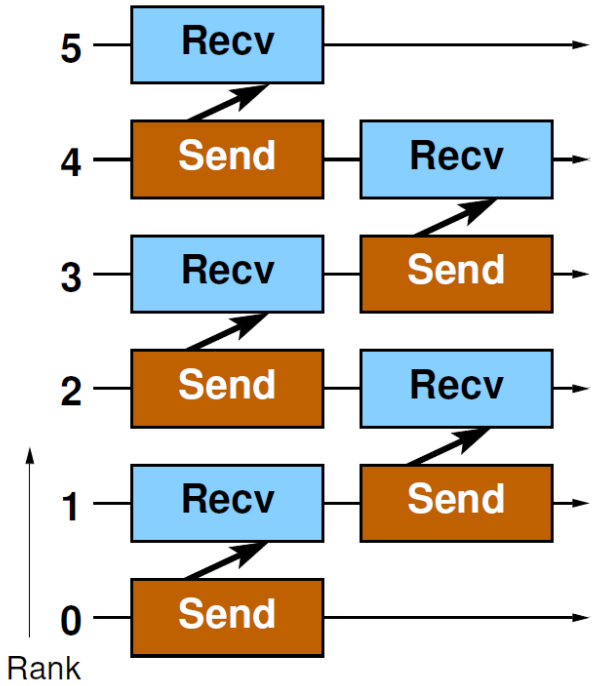
- Worst case scenario 2: Synchronous send using the eager protocol
- Eager: Message may be transmitted to receiver without a matching receive issued. Data is put in a local system buffer at receiver side
- Depends on message length and availability of system buffer space
- Handshake needs to be performed!



Implicit serialization and synchronization

Better implementation alternatives:

- Reverse order of send/receive calls on even and odd ranks



- Use non-blocking **`MPI_Isend/MPI_Irecv`** pairs. Multiple outstanding/open communication requests allow flexible scheduling. Also: potential for asynchronous data transfer.

- Use **`MPI_Sendrecv`** or **`MPI_Sendrecv_replace`**: Simple coding with flexible message scheduling; but no asynchronous transfer

Network contention

- Contention on network level may occur:
  - Multiple processes on a node try to use the network interface at the same time. Network bandwidth per process decreases linearly if a single process can already achieve full network bandwidth
  - Network topology is not fully non-blocking, i.e. bisection bandwidth/compute node decreases with increasing compute nodes. Examples: Torus networks, oversubscribed fat-tree networks
  - Even for full bisectional networks contention may occur on internal network links, due to non-optimal routing

- Communication pattern most vulnerable for contention: **`MPI_Alltoall`**
  → Every process wants to talk to everyone else at the same time (imagine 300.000 processes do that)

Non-blocking but non-asynchronous MPI calls

- Intention: Use non-blocking calls for communication overlap:

```
MPI_Isend(a,...,&request);
...do useful work and do not modify a ...
MPI_Wait(&request,...);
```

- Perfect world:
  - “Useful work” takes longer than communication
  - Nonblocking MPI call implements asynchronous data transfer

- However, the MPI standard does not guarantee asynchronous transfer

A simple test for asynchronicity

- Do some work for some configurable time (**`delay`**) while a message of **`count`** bytes is received or sent

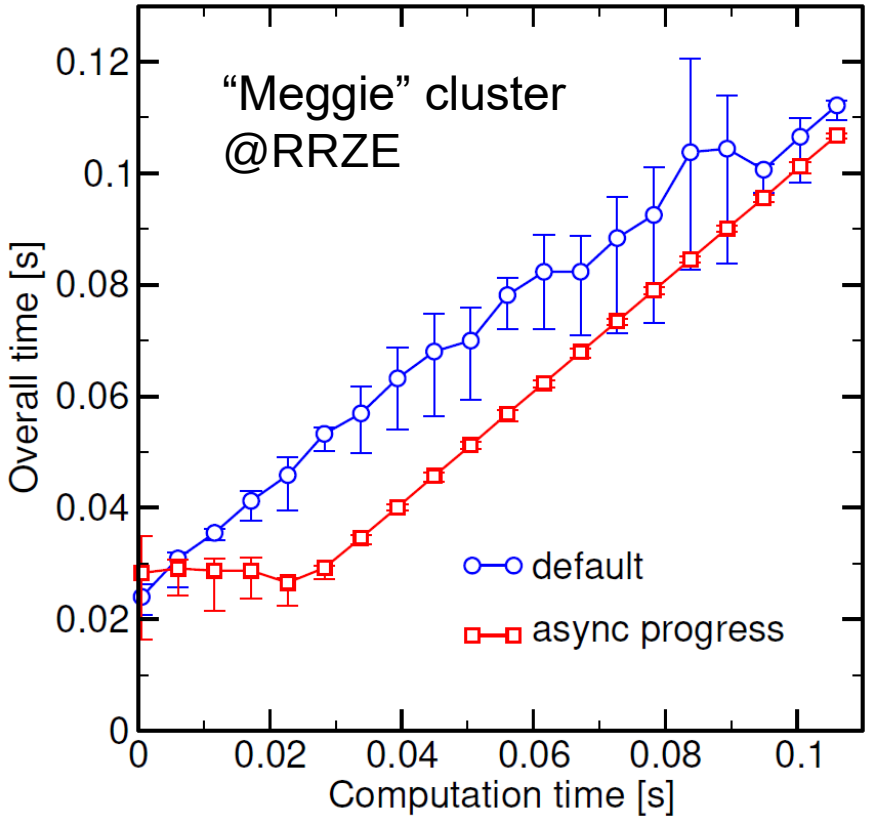
```
if(rank==0) {
  t = MPI_Wtime();
  MPI_Irecv(buf, count, MPI_BYTE, 1, 0, MPI_COMM_WORLD, &req);
  do_work(delay);
  MPI_Wait(&req, &status);
  t = MPI_Wtime() - t;
} else {
  MPI_Send(buf, count, MPI_BYTE, 0, 0, MPI_COMM_WORLD);
}
printf("Overall: %lf  Delay: %lf\n", t, delay);
```

- If overlap occurs, overall time will be constant w.r.t. the delay up to a certain point

Results

- Intel MPI 2019 update 8
- Message size = 240 MB
- Default behavior: no overlap
- **`I_MPI_ASYNC_PROGRESS=1`**: full overlap observed!

- General remarks
  - This is a moving target; MPI implementations change all the time
  - Depends on inter-/intranode, message size, network layer,…
  - MPI implementations provide tuning knobs



MPI performance pitfalls summary

- Always observe possible synchronizing properties of MPI calls
  - Assume the worst possible behavior
- Even if a deadlock does not occur, performance may be severely impacted
- Network contention is a fact
  - Most network connections can be saturated with a single connection
  - Most large-scale networks are not entirely non-blocking for cost reasons
- Nonblocking MPI communication is not necessarily asynchronous
  - Study possible tuning knobs