# Message passing Interface: derived data types in MPI

Adam Belloum

Source: Parallel Programming 2020: Lecture 11- MPI data types, virtual topologies, and performance pitfalls

# MPI data types: Why?

```
MPI_Bcast (&cfg.nx, 1, MPI_INT, …);
MPI_Bcast (&cfg.ny, 1, MPI_INT, …);
MPI_Bcast (&cfg.du, 1, MPI_INT, …);
MPI_Bcast (&cfg.nx, 1, MPI_DOUBLE, …);
MPI_Bcast (&cfg.it, 1, MPI_INT, …);
```

Want to do:

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

```
MPI_Bcast (&cfg, 1, sizeof(cfg),  MPI_BYTE, …);
```

- Woks in practice
- But not portable

# MPI data types: Why?

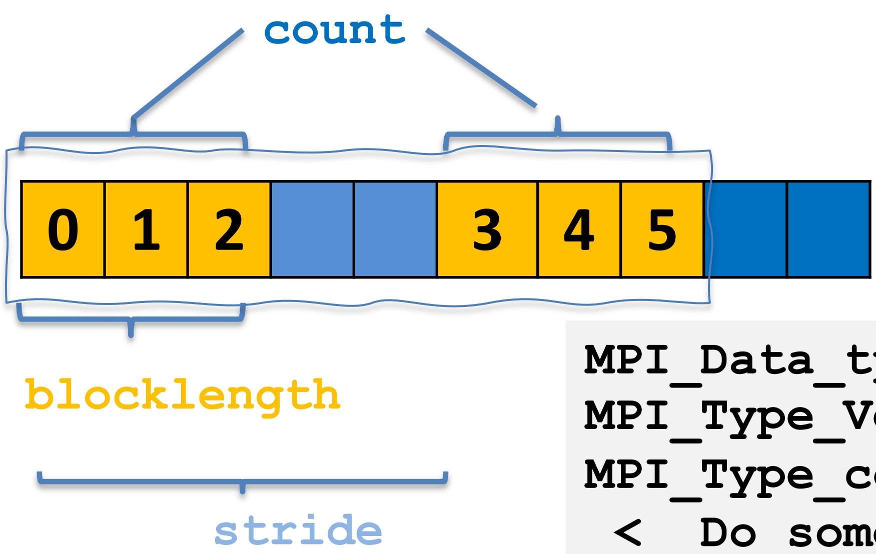
Example: Send column of a matrix (**non-contiguous** in C) ?
- Send each element alone ?
- Manually copy element into a contiguous buffer and sent it?

- New data type !!!
- Three step process
  1. **Contruct** with
     ```
     MPI_Type_*(MPI_Datatype *int);
     ```
  2. **Commit** new data type with
     ```
     MPI_Type_commit(MPI_Datatype *int);
     ```
  3. **Deallocate**
     ```
     MPI_Type_free(MPI_Datatype *int);
     ```

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 |

# A flexible, vector-like type: MPI_Type_vector

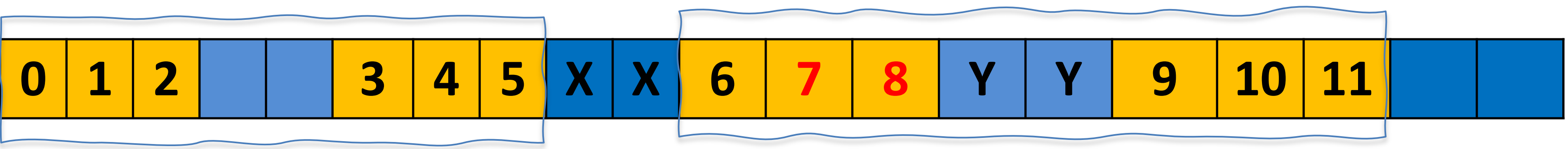
```
MPI_Type_vector (
    int count,          2 // no. of block
    int blocklength, 3 // no. of element/block
    int stride,         5 // no. of elements b/w start of each blck
    MPI_data Type oldtype.
    MPI_Datatype * new type);
```



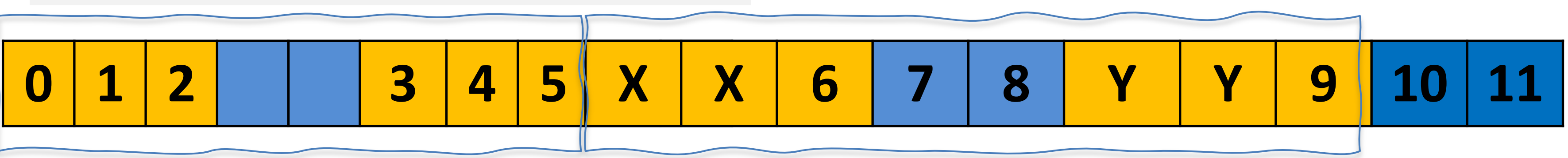

```
MPI_Data_type nt;
MPI_Type_Vector(2, 3, 5, MPI_INT, &nt);
MPI_Type_commit(&nt);
 <  Do something >
MPI_Type_free(&nt);
```

# Caveat when using a type

- Caution:
  - Concatenating such a type 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, …);
```



# Derived type size and extend

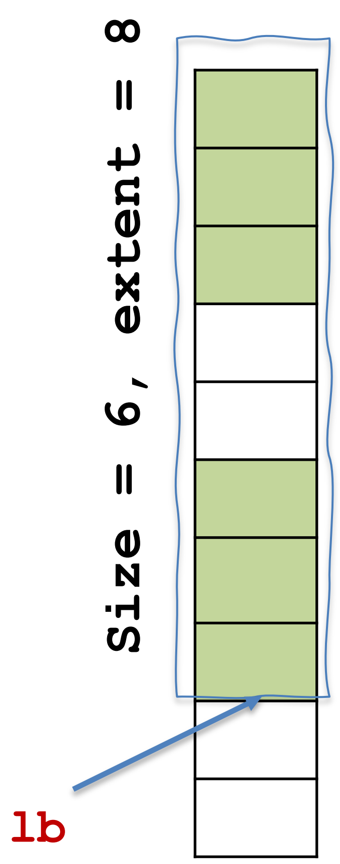
- Get the total size (in byte)of datatype in a message

```
MPI_Type_size(MPI_DataType new type, int *size);
```

- Get the
  - **lower bound**
  - the **extent** Span from **1st** to the **last byte** in a Datatype

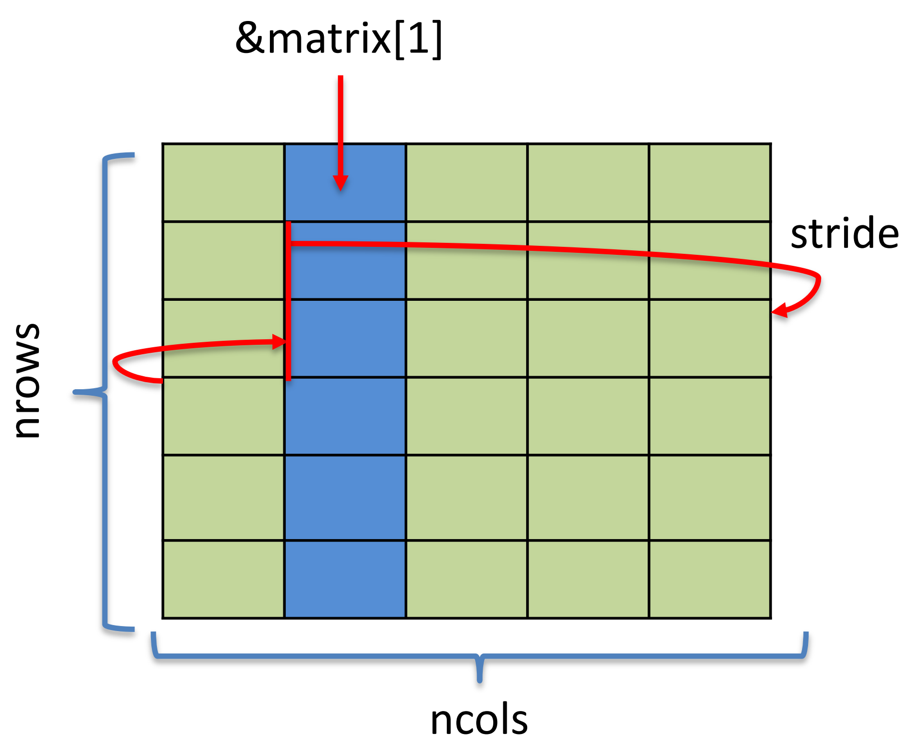
```
MPI_Type_get_extent(
        MPI_Datatype newtype,
        MPI_Aint *lb,
        MPI_Aint *extent);
```

- In MPI to change the extent of a datatype
  - Using `lb_marker` and `ub_marker`
  - Does not affect the size of the datatype
  - Does affect the outcome of the replication of this data type



# Sending a column of a Matrix in C

Raw-major datatype in C → cannot use plain array

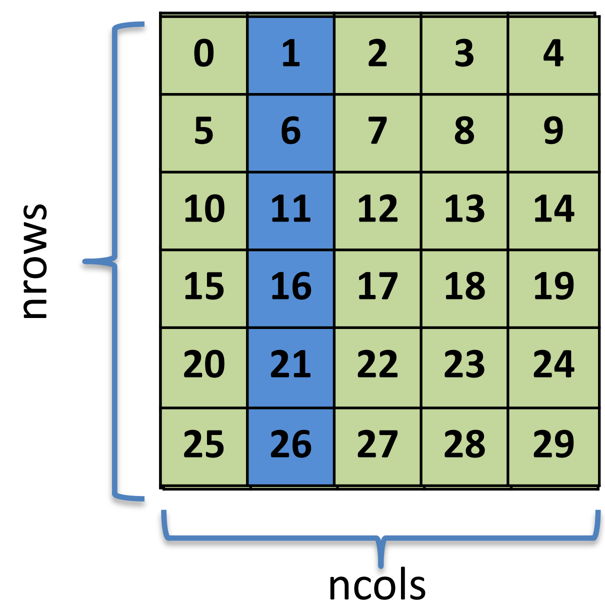



```
double matrix[30];
MPI_Datatype nt;

// 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

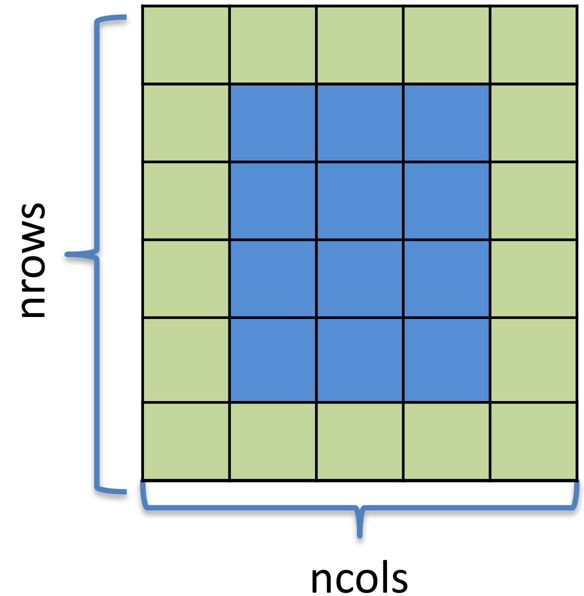
```
MPI_Type_create_subarray (
   int dims,          dim of the initial array
   int ar_sizes[],    array - sizes of array (in each dim)
   int arsizes[],     array - sizes of subarray(in each dim)
   int ar_starts[], Start indices of subarray
   int order,
   MPI_datatype oldtype
   MPI_Datatype * new type);
```

- Order:
  Row-major : **MPI_ORDER_C**
  Column-major: **MPI_ ORDER_Fortran**



# Example for a sub-array type: bulk of a matrix

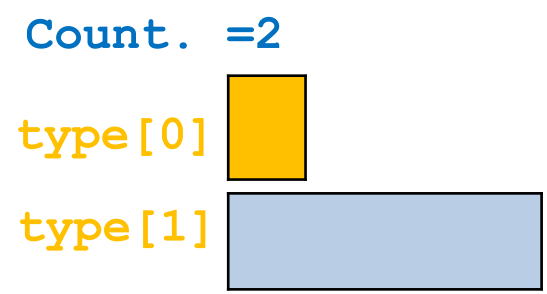
```
Dims         2
ar_sizes     {nclos, nrows}
ar_subsizes  {nclos-2 , nrows-2}
ar_starts    {1, 1}
Order        MPI_ORDER_C
Oldtype      MPI_INT
```

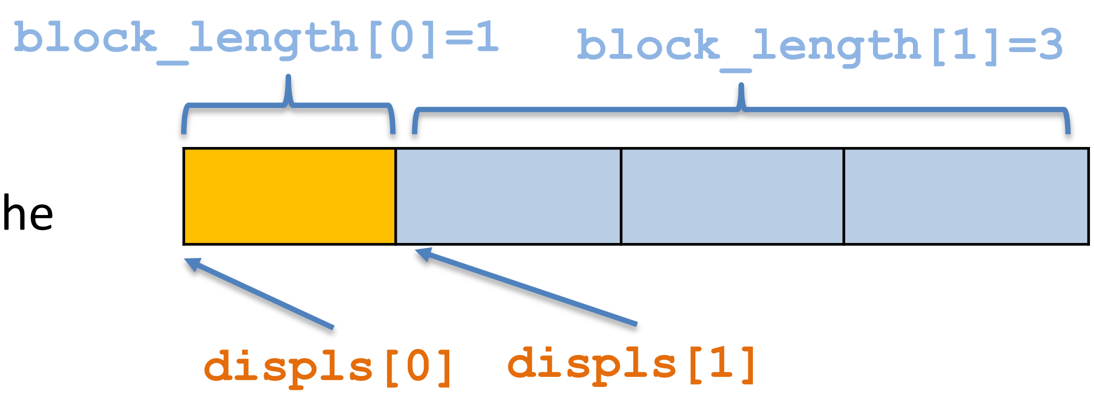



```
MPI_create_Subarray (dims, ar_sizes, ar-subsizes, ar_starts,
order, oldtype,&nt);

MPI_Type_commit (&nt);
//Use nt …
MPI_Send (&bud[0], 1, nt, …); //etc
MPI_type_free(&nt)
```

# Most flexible type: MPI_Type_create_struct

```
MPI_Type_create_struct (
    int count,              // number of type composing the struct
    int bloc_length[],      // the length of type composing the struct
    int 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 obtains 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[1000];
    MPI_Aint a1, a2, disp;
    MPI_Get_address(&a[0], &a1);
    MPI_Get_address(&a[50], &a2);
    Disp = MPI_Aint_diff (a2,a1)
```

# Derived data type: summary

- A flexible tool for 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,` …

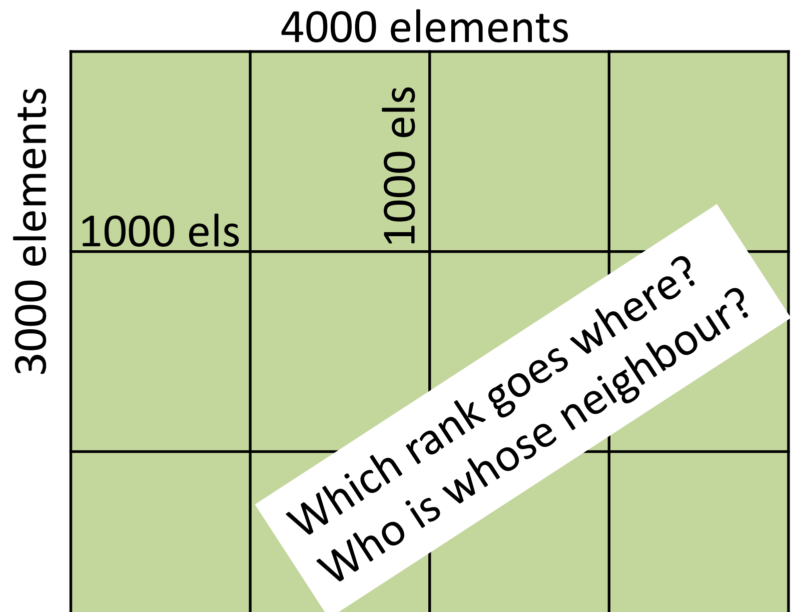
- **Marching rule**:
  - **send** and **receive** match if specified **basic datatypes** match **one by one**,
  - Correct displacement at receiver side are automatically matched to the corresponding data items

# Message passing Interface: Virtual (Cartesian) topologies in MPI

# A convenient process naming scheme for multi-dimension problems

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

Example distribute 2-D array of 400x 3000 elements equally on 12 ranks



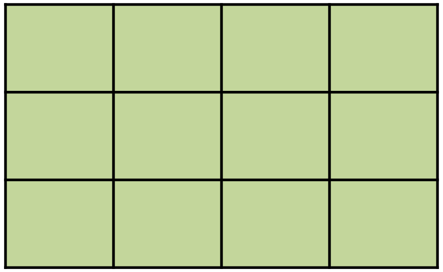

- Let **MPI map ranks to coordinates**
- User: map array segments to ranks

# Creating a Cartesian communicator

- Create a **new communicator** attached to cartesian topology

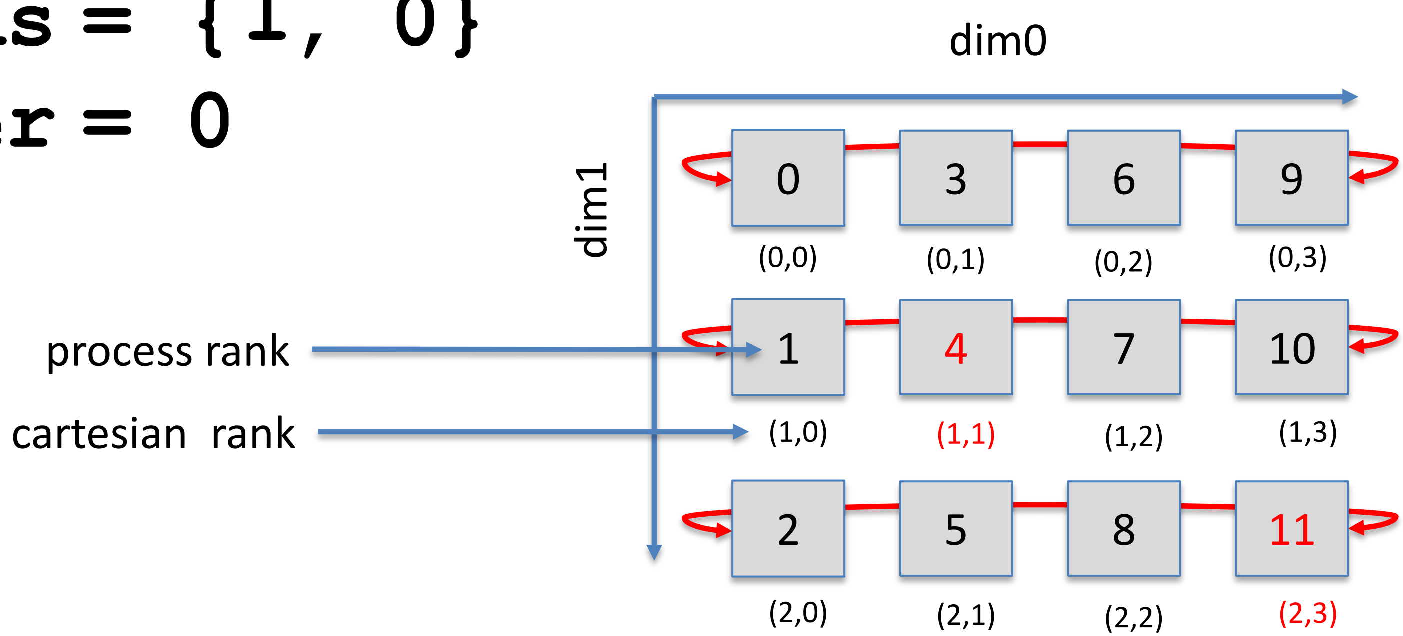
```
MPI_Cart_create(
   MPI_Comm oldcom,
   int ndims,           number of dimensions
   int dims[],          array with ndims elements, dims[i] specifies
                        the number of ranks in dimension i
   int periods[],       array with ndims elements, periods[i]
                        specifies if dimension i is periodic
   int reorder,         allow rank of oldcomm to have a different
                        rank in cart_comm
   MPI_Comm *cart_somm
);
```

```
ndims = 2
dims[0]=4
dims[1]=3
Periods=0/1
```



# 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 th**

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

- **Map rank → coordinates ("where am I in the grid ?")**

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

**rank:    any rank is part of the cartesian comm cart_comm**
**coords: array of maxdims elts,receives the coordinates for rank**

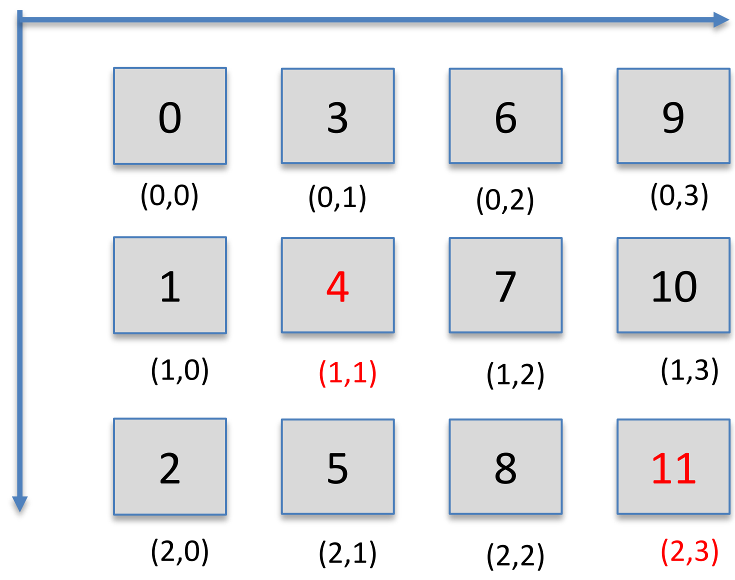
- **Map coordinates → rank (Who is at the position ?")**

```
MPI_Cart_rank(cart_comm, int coords[], int *rank);
```

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

# Example

- Example 12 processes arranged in 4X3 grid
- Column-major numbering
- Process coordinates begin with 0



# Next-neighbour communication

- Sending/receive from **neighbours** is a typical task in cartesian topologies

```
MPI_Cart_shift (
    cart_comm,
    direction,  direction: dimensions to shift
    disp,       offset to shift:
                   >0 shift in positive direction,
                   >0 shift in negative direction
    int *src_rank,    retuned ranks as input (argument)of
    int *dest_rank      MPI_Sendrec* calls
);
```

# Next-neighbour communication

Example :4X3 progress grid , **period on 1st dimension** (dim0 - horizontal),
each process has an int value which get shifted

```
MPI_Cart_shift (cart_comm, 0, 1, &src, &dest);
MPI_Sendrec_replace(&value, 1, MPI_INT, dst, 0, src, 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 |

```
MPI_Cart_shift (cart_comm, 1, 1, &src, &dest);
MPI_Sendrec_replace(&value, 1, MPI_INT, dst, 0, src, 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 |

For non-periodic topologies
- MPI_PROC_NULL is returned in boundaries

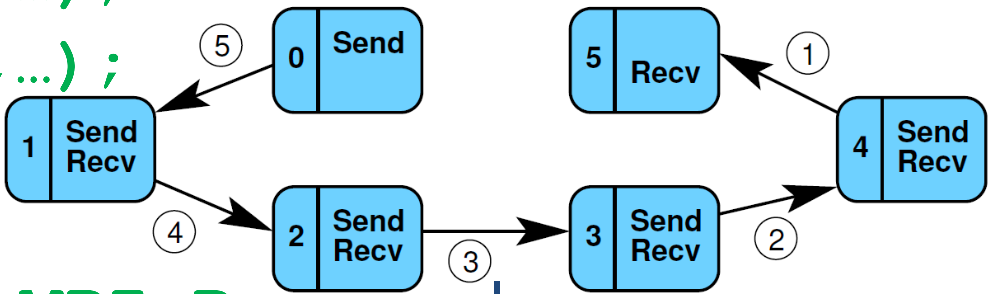
# Message passing Interface:
# typical performance pitfall in MPI

# Implicit serialization and synchronization

Common performance pitfall with MPI

Consider linear shift in an **open chain**, e.g.,

- **each rank** in the chain issues:
  - `MPI_Send(…,rank+1,…);`
  - `MPI_Recv(…,rank-1,…);`



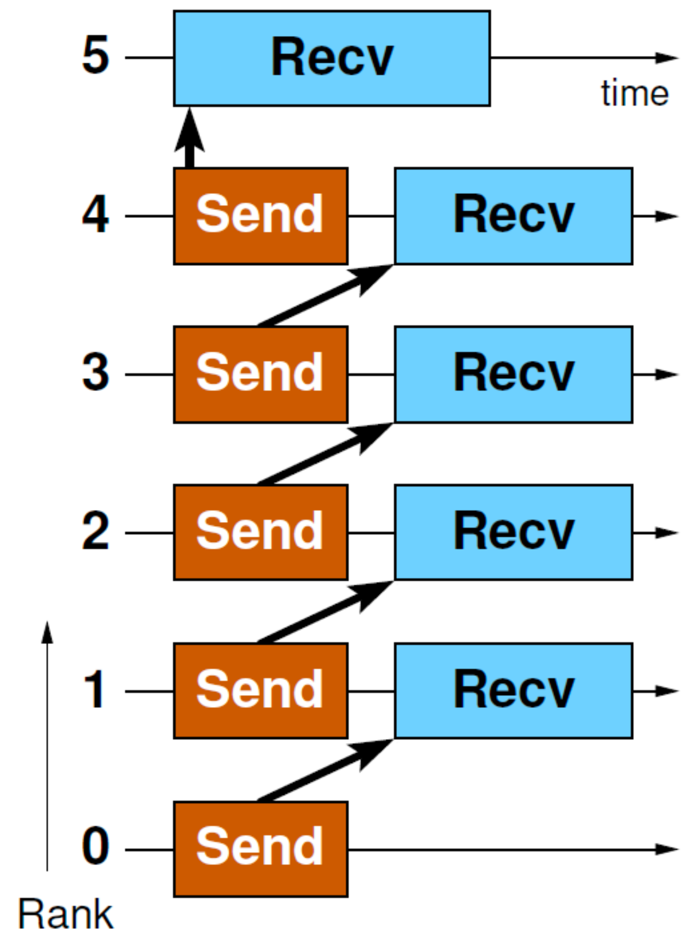

- **First** and **last** rank call → `MPI_Send` and `MPI_Recv` only
- There is no danger of deadlocks
- **But** performance depends on implementation-specific parameters in the MPI library: `MPI_Send`

# Implicit serialization and synchronization

- **Best case scenario**:
  - `MPI_Send` **operates in a buffered send mode**

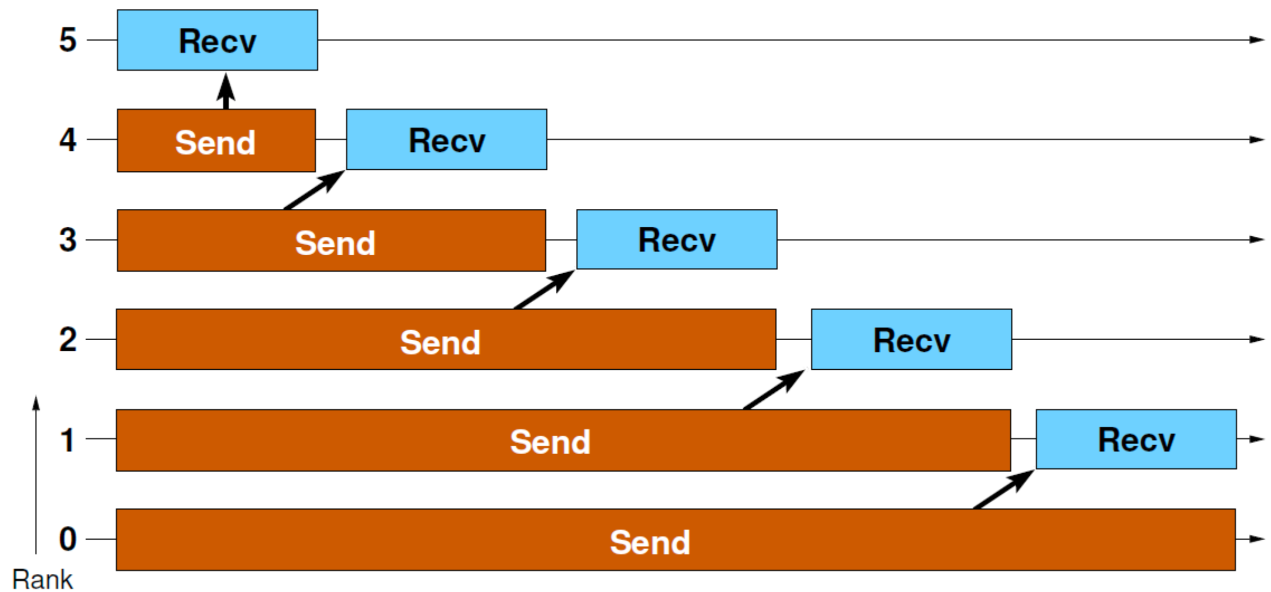
- `MPI_Send` returns after message is copied to a system buffer

- Send/Receive operations can be overlapped on nonblocking, bidirectional networks



# Implicit serialization and synchronization

- **Worst case scenario 1**: Synchronous send using
  - the rendezvous protocol

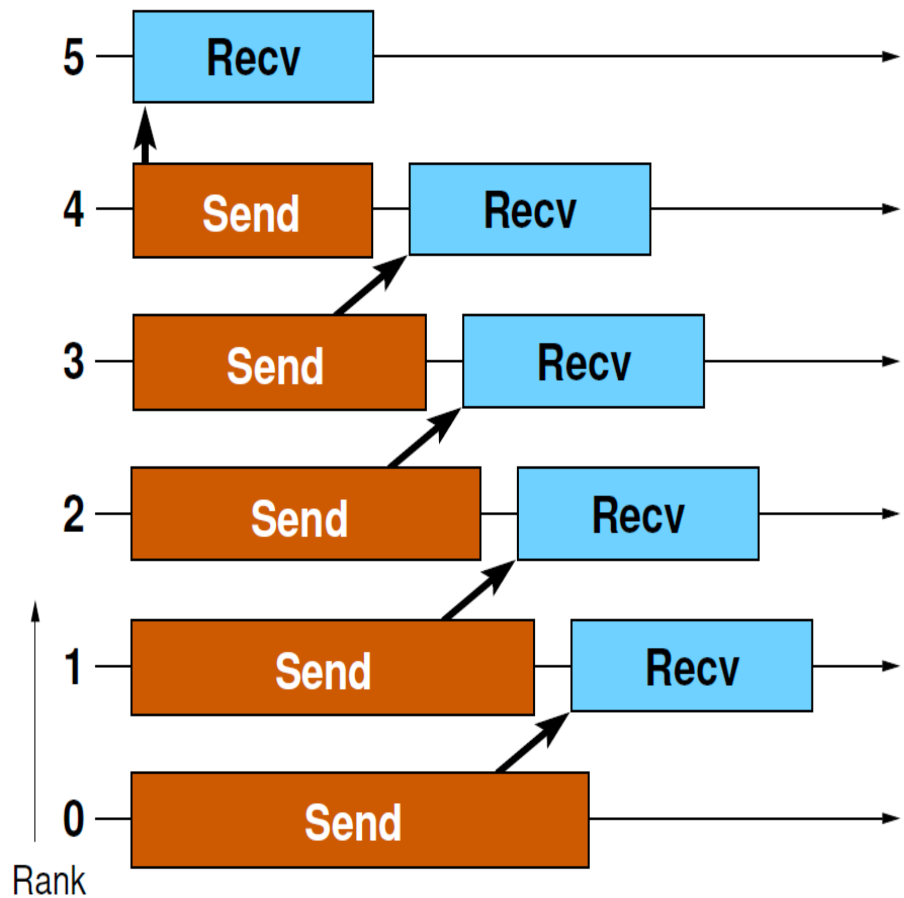



Rendezvous: Send blocks until complete message has been transferred!

# Implicit serialization and synchronization

- **Worst case scenario 2**: Synchronous send using the eager protocol

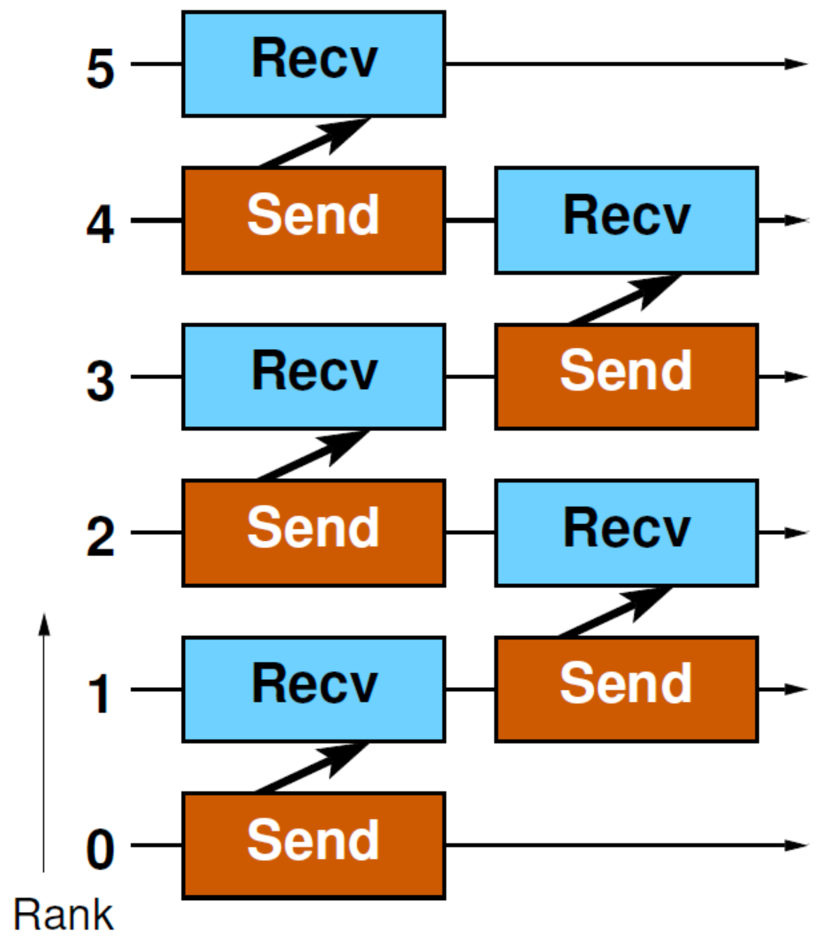
- Eager:
  - Message may be transmitted to receiver without a matching receive issued.
  - Data is put in a local system buffer at receiver side



# Implicit serialization and synchronization

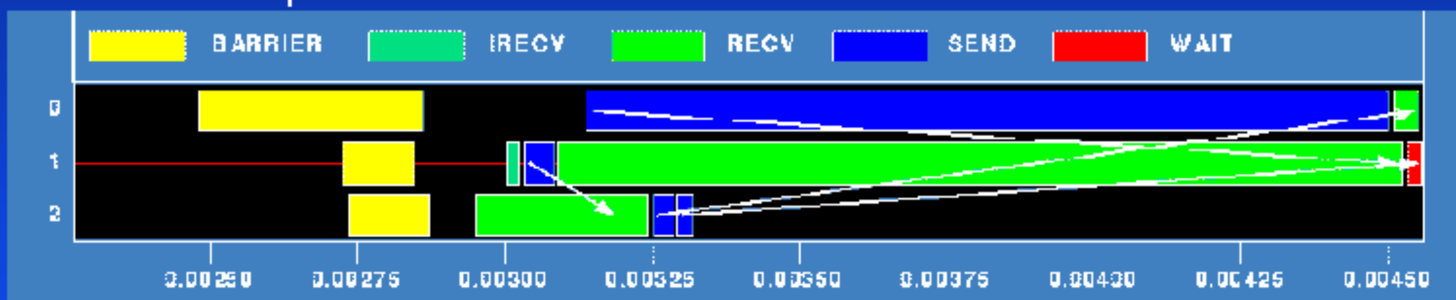
**Better implementation alternatives**:

- 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

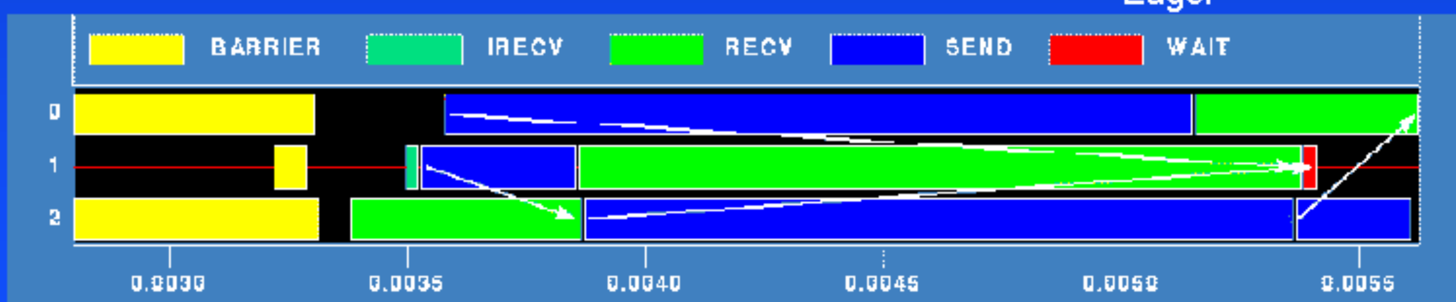


# Observing Synchronization Delays

- 3 processors sending data, with one sending a short message and another sending a long message to the same process:




Eager




Rendezvous

# Network contention

Contention on network level may occur:

- Multiple processes on a node 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.
- Non-optimal routing:
  - Even for full non-blocking networks contention may occur on internal network links
- **`MPI_Alltoall -`** Communication pattern most vulnerable:
  - 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: %lfDelay: %lf\n", t, delay);
```

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

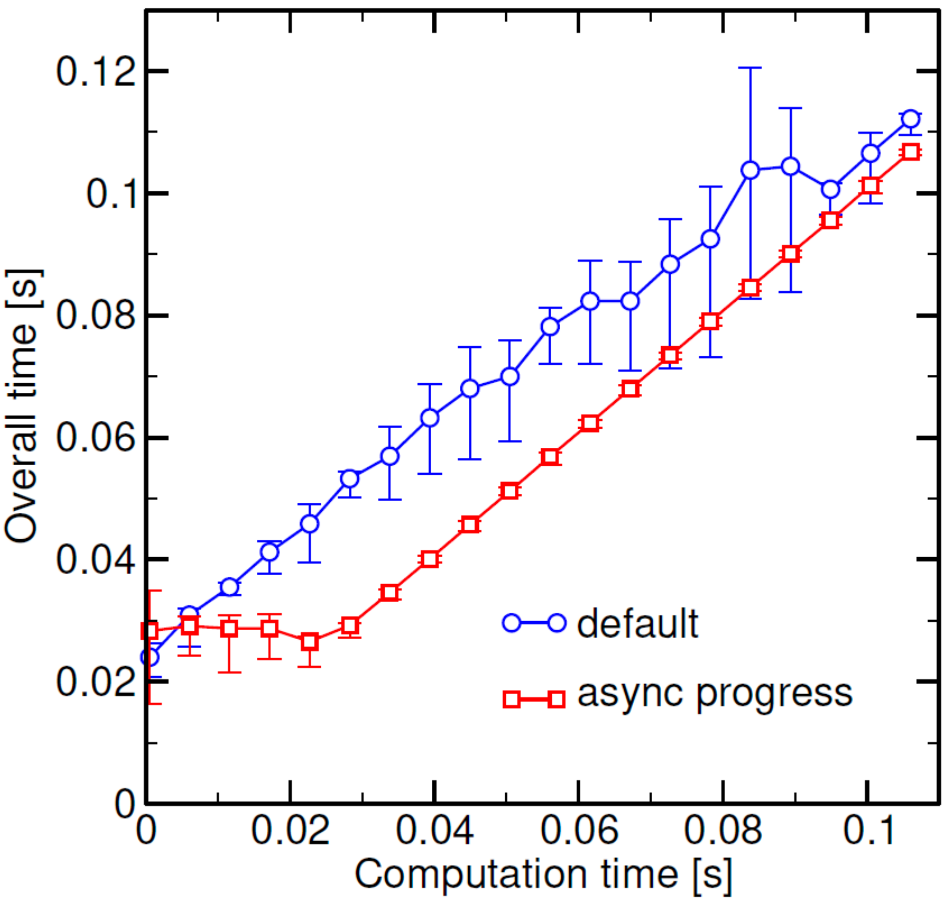
# Results

- Intel MPI 2019 update 8
- Message size = 240 MB
- Default behaviour: no overlap
- **I_MPI_ASYNC_PROGRESS=1**:

  // full overlap observed!

General remarks: 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 behaviour
- 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
- Non-blocking MPI communication is not necessarily asynchronous
- Study possible tuning knobs