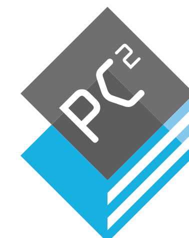
High-Performance Computing
– Advanced MPI –

Christian Plessl

High-Performance IT Systems Group
Paderborn University, Germany



version 1.0.2 2018-01-23

Outline

- Derived Data Types
- Non-Blocking Communication
- One-Sided Communication
- Hybrid Parallel Programming

Derived Data Types

Basic MPI Data Types

- Each communication in MPI requires to define data type and length
- MPI standard defines a set of basic (intrinsic) MPI data types
  - correspond native data types of C/Fortran
  - e.g. signed int → MPI_INT, double → MPI_DOUBLE, ..
  - single elements or contiguous arrays of same type can be transferred

- Example: send 100 double values in array a to rank 42

```
double buf[100];
MPI_Send(buf, len, MPI_DOUBLE, 42, 0, MPI_COMM_WORLD);
```

Derived Data Types

- Derived data types can express arbitrary data structures that are communicated
  - hierarchical construction based on basic or derived types
  - MPI runtime constructs efficient (de)serialization methods

- Purpose
  - communication of non-contiguous data (e.g. arrays with strided access)
  - communication heterogeneous data (e.g. structs comprising different types)
  - raise abstraction level of program (more expressive and shorter code)
  - increase communication efficiency (fewer data transfers)

- All communication types are supported
  - point-to-point, collective, blocking, non-blocking

Motivation: Sending Matrix Column in C

- Two dimensional arrays in C are stored in row-major order
- Communicating a row the array with basic MPI data types is not efficiently possible because data is non-contiguous
- Workarounds for communicating a row
  - one transfer per row element (a11, a21, a31)
  - transfer of whole array, discard unneeded elements
  - copying data to temporary contiguous buffer, which is then sent (manual marshalling)
- All workarounds are inefficient or cumbersome and increase complexity of code

a11	a12	a13
a21	a22	a23
a31	a32	a33

2D array a in C

storage in memory (C uses row major order)

Motivation: Sending Matrix Column in C (2)

- Solution: create derived data types for expressing a column in the array
  - enables to send a row with single MPI transfer
- Example
  - assume 2D NxN array of doubles
  - build custom data type for representing a row
- MPI_Type_vector constructor
  - N elements
  - groups of 1 (single) elements,
  - stride N (spacing between elements)
  - base type MPI_DOUBLE

```
double A[N][N];
MPI_Datatype row_t;
MPI_Type_vector(N, 1, N, MPI_DOUBLE, &row_t);
MPI_Type_commit(&row_t);
MPI_Send(&A[0][1], 1, row_t, 42, 0, MPI_COMM_WORLD);
...
MPI_Type_free(&row_t);
```

send second column of array

Type Lifecycle Management

- Creating a name for a derived data type
  - expressed with variable of type MPI_Datatype
- Declaration of new data type
  - MPI_Type_create constructor functions define new types based on existing types (flat or hierarchical)
- Finalizing the construction of data type
  - calling the MPI_Type_commit function instructs MPI that the type is final
  - triggers generation of optimized methods for (de)serialization
  - committing is only needed for types that are actually used in communication (intermediate types used just for hierarchical definitions do not need to be committed)
- Releasing resources
  - if a type is no longer needed, resources can be released with MPI_Type_free

Available Type Constructors

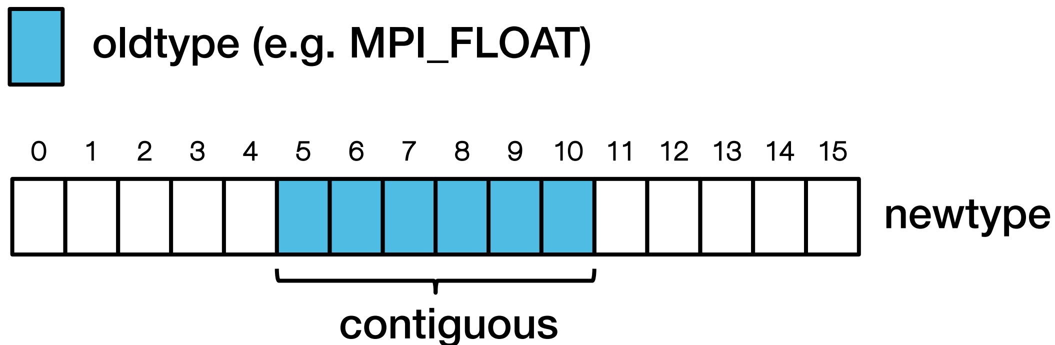
Constructor name	Purpose
MPI_Type_contiguous	Contiguous data types
MPI_Type_vector	Block of array elements with regular strides
MPI_Type_create_hvector	Block of array elements with regular stride (specified in bytes instead of size of oldtype)
MPI_Type_create_indexed_block	Blocks of array elements with irregular block lengths and strides
MPI_Type_indexed	Block of array elements with irregular strides
MPI_Type_create_struct	Most general data type
MPI_Type_create_subarray	Data type for n-dimensional array slices

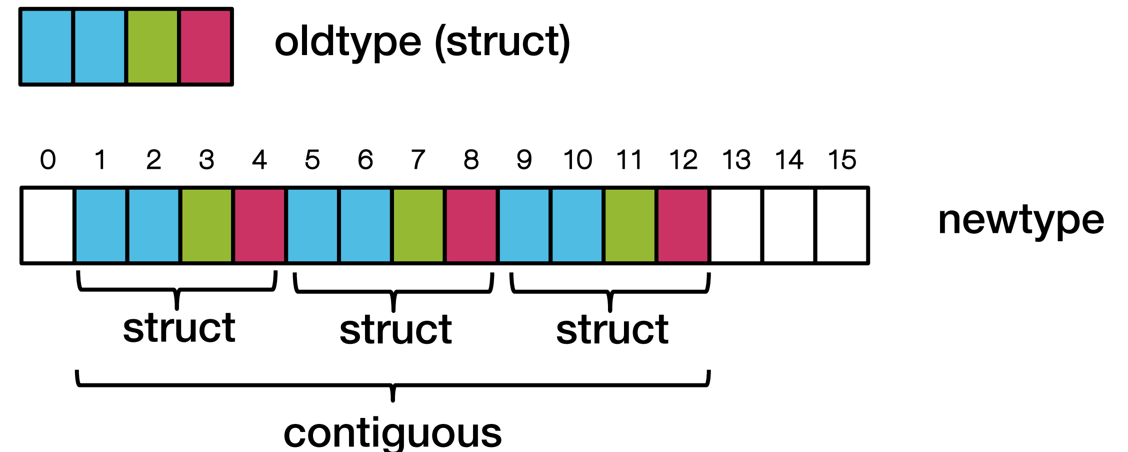
some frequently used type constructors (there are many more)

MPI_Type_Contiguous

```
MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Declare contiguous array of oldtype
  - count: number of elements
- Do not used as last type (use length parameter of send/recv instead)



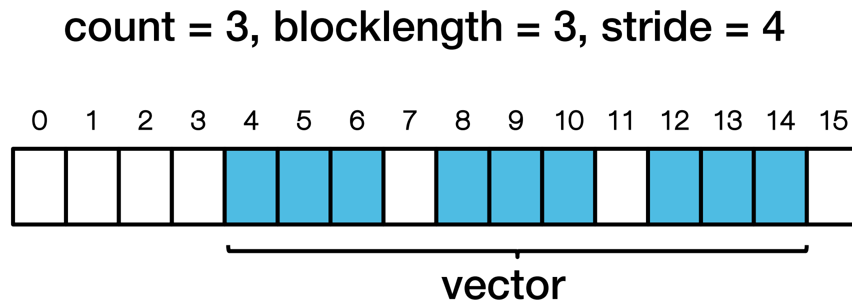


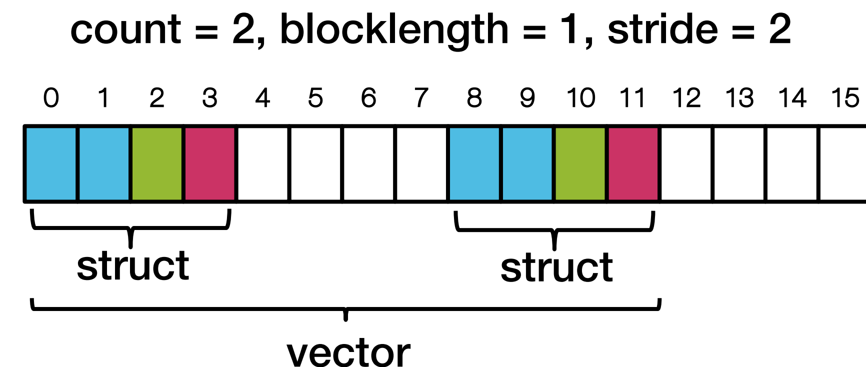
MPI_Type_vector

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

- Declare data type of identical blocks with fixed stride
  - count: number of blocks
  - blocklength: number of elements in each block
  - stride: displacements between blocks
- Use cases
  - communicating rows or planes in multi-dimensional arrays
  - arrays of more complex structures, e.g. vector of structs

stride measured in extent of oldtype, i.e. second struct could not start at offset 7

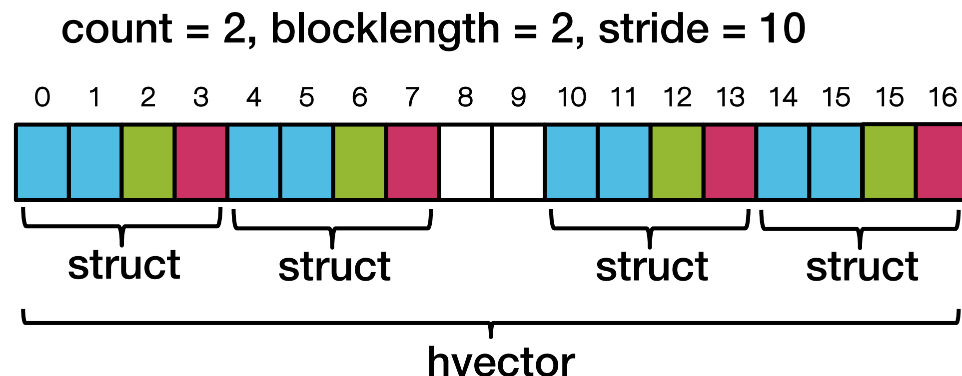




MPI_Type_create_hvector

```
MPI_Type_create_hvector( int count, int blocklength, MPI_Aint stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

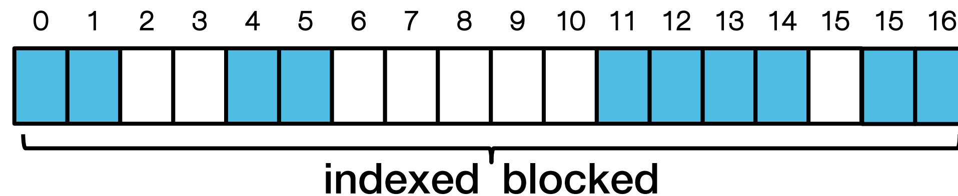
- Same function as regular vector, but stride is specified in bytes instead of size of oldtype
  - allows for using strides that are not evenly divisible by length of oldtype
- Declare data type of identical blocks with fixed stride
  - count: number of blocks
  - blocklength: number of elements in each block
  - stride: displacements between blocks in bytes (not extent of oldtype)



MPI_Type_create_indexed_block

```
MPI_Type_create_indexed_block( int count, const int blocklength,
  const int displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Extracts variable sized and spaced blocks of data comprising identical elements
  - blocklengh: length of blocks
  - displacements[]: displacements expressed in size (extent) of oldtype

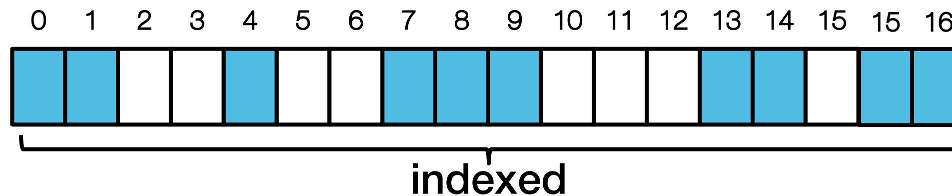


blocklength = 2
displacements = { 0, 4, 11, 13, 15 }

MPI_Type_indexed

```
MPI_Type_indexed( int count, const int blocklengths[], const int displacements[],
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Extracts variable sized and spaced blocks of data comprising identical elements
  - blocklengths[]: length of blocks as array
  - displacements[]: displacements expressed in size (extent) of oldtype
  - there is also a MPI_Type_create_hindexed variant that uses displacements in bytes



blocklengths = { 2, 1, 3, 2, 2 }
displacements = { 0, 4, 7, 13, 15 }

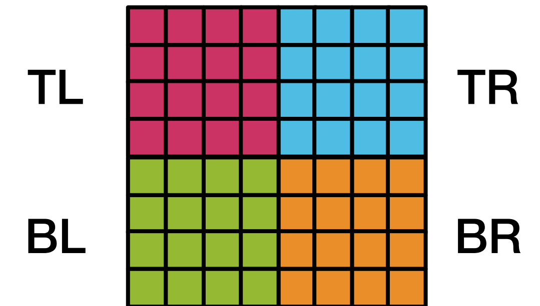
MPI_Type_create_subarray

```
MPI_Type_create_subarray( int ndims, const int size[], const int subsize[],
 const int start[], int order, MPI_Datatype *oldtype, MPI_Datatype *newtype)
```

- Create an n-dimensional subarrays from an n-dimensional array which is stored in a linearized way.
  - ndims: numbe of dimensions of full array (must match length of arrays size, subsize, start)
  - size[]: size of original array
  - subsize[]: size of subarray
  - start[]: start of subarray, indexes start at 0
  - order: MPI_ORDER_C (array is stored in row-major order), or MPI_ORDER_FORTRAN (column-major order)

MPI_Type_create_subarray Example

- Send 4 quadrants of array from master process to ranks 1–4 for further processing




```
double *array = ...;
int array_sz[2] = {8,8};
int sub_sz[2] = {4,4};
int off_TL[2] = {0,0}, off_TR[2] = {0,4}, off_BL[2] = {4,0}, off_BR[2] = {4,4};
MPI_Datatype TL, TR, BL, BR;

MPI_Type_create_subarray(2, array_sz, sub_sz, off_TL, MPI_ORDER_C, MPI_DOUBLE, &TL);
MPI_Type_commit(&TL);
MPI_Type_create_subarray(2, array_sz, sub_sz, off_TR, MPI_ORDER_C, MPI_DOUBLE, &TR);
MPI_Type_commit(&TR);
...

if(rank==0) {
    MPISend(array, 1, TL, 1, 0, MPI_COMM_WORLD);
    MPISend(array, 1, TR, 2, 0, MPI_COMM_WORLD);
    MPISend(array, 1, BL, 3, 0, MPI_COMM_WORLD);
    MPISend(array, 1, BR, 4, 0, MPI_COMM_WORLD);
} ...
```

MPI_Type_create_struct

```
MPI_Type_create_struct( int count, const int blocklengths[],
   const MPI_Aint displacements[], const MPI_Datatype types[],
   MPI_Datatype *newtype)
```

- Fully general constructor for creating new type with arbitrary many elements, displacements and types
  - blocklengths[]: length of blocks as array
  - displacements[]: byte displacements of each block as array
  - types[]: type of elements in each block (array of MPI_Datatype elements)
- The displacement can be determined in portable way using the function

```
MPI_Get_address(cosnt void *location, MPI_Aint *address)
```

- see example

MPI_Type_create_struct Example (simple case)

```
typedef struct {
  float x, y, z, velocity;
  char name[10];
  double mass;
} particle_t;
particle_t p[N];

MPI_Datatype particletype;

MPI_Datatype oldtypes[3] = {MPI_FLOAT, MPI_CHAR, MPI_DOUBLE};
int len[3] = {4, 10, 1};
int disp[3];
disp[0] = 0;
disp[1] = disp[0] + 4*sizeof(float);
disp[2] = disp[1] + 10*sizeof(char);

MPI_Type_create_struct(3, len, disp, oldtypes, &tmp);
MPI_Type_create_resized(tmp, 0, sizeof(particle_t), &particletype);
MPI_Type_commit(&particletype);
MPI_Send(p, N, particletype, dest, tag, comm);
```

declaration of C struct type for particles

what is the size of one particle_t structure?

The whole may be more than the sum of its parts

CAUTION: This example may be incorrect, depending on CPU architecture and compiler options/defaults

Complications by Struct Padding and Alignment

- The C compiler can exploit different performance / storage size trade-offs for structs
  - dense packing minimizes storage requirements but data may be poorly aligned for loads and stores, caching and vectorization
  - compiler can insert padding elements in struct for optimization
  - since handling of structs and unions is architecture and compiler specific, structs can cause problems with portability
- ISO C standard, “6.7.2.1 structure and union specifiers”
  - 14. Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.
  - 15. Within a structure object, the non-bit-field members [...] have addresses that increase in the order in which they are declared. [...] There may be unnamed padding within a structure object, but not at its beginning.
  - 17. There may be unnamed padding at the end of a structure or union

Example: Struct Alignment with GCC on x86 Linux

- GCC allows controlling struct packing and alignment in struct declaration and as variable attributes
  - __attribute__((packed)) use dense packing of struct elements
  - __attribute__((aligned (n))) force compiler to allocate and align variable at (at least) an n-byte boundary

```
typedef struct {
  float x, y, z, velocity;
  char name[10];
  double mass;
} __attribute__((packed))__ __attribute__((aligned (8))) particle_t;
```

			Index																																													
Packed	**Aligned**	**Sizeof**	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	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Default	**Default**	**40**	x	x	x	x	y	y	y	y	z	z	z	z	v	v	v	v	n	n	n	n	n	n	n	n	n	n	p	p	p	p	p	p	m	m	m	m	m	m	m	m								
Yes	**Default**	**34**	x	x	x	x	y	y	y	y	z	z	z	z	v	v	v	v	n	n	n	n	n	n	n	n	n	n	m	m	m	m	m	m	m	m																
Yes	**8**	**40**	x	x	x	x	y	y	y	y	z	z	z	z	v	v	v	v	n	n	n	n	n	n	n	n	n	n	m	m	m	m	m	m	m	m	p	p	p	p	p	p								
No	**8**	**40**	x	x	x	x	y	y	y	y	z	z	z	z	v	v	v	v	n	n	n	n	n	n	n	n	n	n	p	p	p	p	p	p	m	m	m	m	m	m	m	m								
Yes	**16**	**48**	x	x	x	x	y	y	y	y	z	z	z	z	v	v	v	v	n	n	n	n	n	n	n	n	n	n	m	m	m	m	m	m	m	m	p	p	p	p	p	p	p	p	p	p	p	p	p	p
No	**16**	**48**	x	x	x	x	y	y	y	y	z	z	z	z	v	v	v	v	n	n	n	n	n	n	n	n	n	n	p	p	p	p	p	p	m	m	m	m	m	m	m	m	p	p	p	p	p	p	p	p
			64bit								64bit								64bit								64bit								64bit								64bit							

MPI_Type_create_struct Example (max. Portability)

```
typedef struct {
  float x, y, z, velocity;
  char name[10];
  double mass;
} particle_t;
particle_t p[N];

MPI_Datatype particletype, tmp;

MPI_Datatype oldtypes[3] = {MPI_FLOAT, MPI_CHAR, MPI_DOUBLE};
int len[3] = {4, 10, 1};
MPI_Aint base, disp[3];
MPI_Get_address(particle[0].x, disp[0]);
MPI_Get_address(particle[0].name, disp[1]);
MPI_Get_address(particle[0].mass, disp[2]);
base = disp[0];
for (int i=0; i<3; i++) disp[i] = MPI_Aint_diff(disp[i], base);

MPI_Type_create_struct(3, len, disp, oldtypes, &tmp);
MPI_Type_create_resized(tmp, 0, sizeof(particle_t), &particletype);
MPI_Type_commit(&particletype);
MPI_Send(p, N, particletype, dest, tag, comm);
```

declaration of C struct type for particles

MPI_Get_address is a portable way of determining address of variables

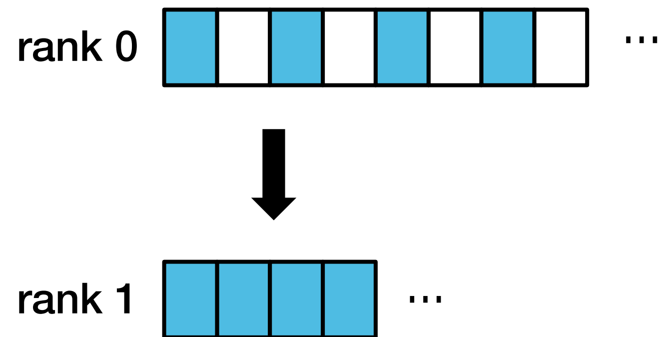
displacements are relative to base, use MPI_Aint_diff to compute in portable way

Compiler could add padding after each struct element in array. MPI_Type_create_resized adjusts size if needed

Data Type Conversion in MPI

- MPI offers limited forms of “data type” conversion
  - simple data layout conversions are supported, e.g. from contiguous to vector layouts
  - there is however no conversion between the actual data types (‘leaves’ of a data structure definition), e.g. no conversion from MPI_FLOAT to MPI_DOUBLE

- Example




```
MPI_Type my_vec_t;
MPI_Type_vector(N, 1, 2, MPI_FLOAT, my_vec_t);
float *a = (float*)malloc(N*sizeof(float));
init(a);

if (rank == 0) {
  MPI_Send(a, 1, my_vec_t, 1, 0, MPI_COMM_WORLD);
} else {
  MPI_Recv(buf, N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
}
```

Advise on Defining and Using MPI Datatypes

- Tradeoff between abstraction/convenience and performance
- Rule of thumb
  - the more parameter a MPI_Type_create constructor has, the slower the performance
  - predefined < contig < vector < index_block < index < struct
- Tips
  - construct data types hierarchically, from bottom up
  - use few, long data transfers instead of many small transfers
  - don’t use contiguous as the outermost MPI Datatype because multiple elements can be sent using the count argument of peer-to-peer or collective communication functions

Non-Blocking Communication

Non-Blocking Communication Objectives

- Blocking MPI_Send / MPI_Recv cause overheads
  - MPI_Send blocks until the message has been delivered to receiver (see MPI standard for precise semantics and guarantees)
  - when sending or receiving multiple independent messages, MPI_Send/Recv enforce ordering
  - overlapping of computation and communication is not possible

- Non-blocking MPI communication
  - non-blocking send (MPI_Isend) and receive (MPI_Irecv) immediately return and handle communication in background
  - completion of communication can be tested and enforced with additional functions
  - allows to overlap communication and computation
  - can avoid many common deadlocking problems

- Blocking and non-blocking communication can be mixed
  - MPI_Isend can be received by MPI_Recv

Non-Blocking Send and Receive

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

- Same parameters and types as MPI_Send
- Additional request parameter used for query status of communication or waiting for completion

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

- Same parameters and types as MPI_Recv but no status parameter
- Additional request parameter used for query status of communication or waiting for completion

Testing and Waiting for Non-Blocking Communication

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

- Test for completion of a single MPI request
  - request: handle to a request returned by MPI_Isend / MPI_Irecv
  - flag: returns true if operation has logically completed
  - status: delivers additional information, if application does not need additional status information MPI_STATUS_IGNORE can be passed to save resources

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

- Wait for completion of a single MPI requests
  - request: handle to a request returned by MPI_Isend / MPI_Irecv
  - status: delivers additional information, if application does not need additional status information MPI_STATUS_IGNORE can be passed to save resources

Testing and Waiting for Non-Blocking Communication

- Additional functions for testing of – or waiting on – multiple MPI requests concurrently
  - function return which requests have completed

Function	Purpose
MPI_Testall	Test for completion of **all** requests in a set
MPI_Testany	Test for completion of **zero or one** request in a set
MPI_Testsome	Test for completion of **one or more** requests
MPI_Waitall	Wait for completion of **all** requests in a set
MPI_Waitany	Wait for completion of **zero or one** request in a set
MPI_Waitsome	Wait for completion of **one or more** requests

Testing MPI Request Sets (1)

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

- Test for completion of all requests in a set
  - count: number of requests
  - requests: arrays of requests (length = count)
  - flag: returns true if all operations have completed
  - statuses: like in MPI_Test, use constant MPI_STATUSES_IGNORE if not needed

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

- Test for completion of zero or one request in a set
  - flag: returns true if a request has completed, index of request is returned in index
  - other parameters like MPI_Testall

Testing MPI Request Sets (2)

```
MPI_Testsome(int incount, MPI_Request requests[], int *outcount, int indices[],
  MPI_Status *statuses[])
```

- Test for completion of one or more request in a set
  - incount: number of requests
  - requests: arrays of requests (length = incount)
  - outcount: returns number of requests that have completed
  - indices: returns array with indices of requests that have completed

Waiting For MPI Request Sets

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

- Wait for completion of all requests in a set
  - count: number of requests
  - requests: arrays of requests (length = count)

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

- Wait for completion of zero or one request in a set
  - index: index of handle that completed

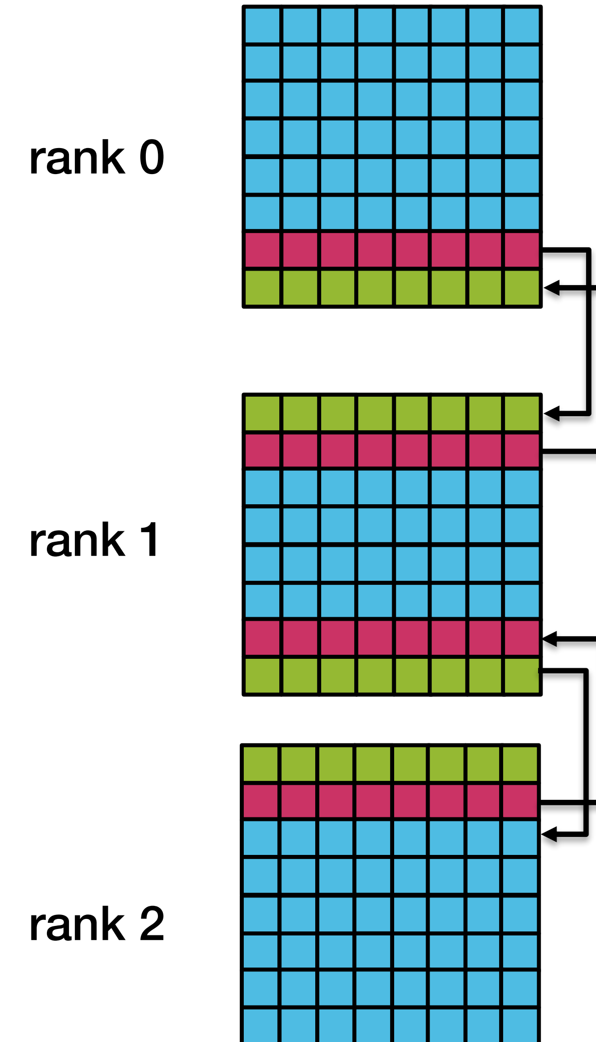
```
MPI_Waitsome(int incount, MPI_Request requests[], int *outcount, int indices[],
  MPI_Status *statuses[])
```

- Wait for completion of one or more request in a set
  - parameters analogous to MPI_Testsome

Typical Use Case for Non-Blocking Communication

- Example from exercise Conway's Game of Life
  - each cell updates requires data from 1-neighborhood
  - parallelization can be done by duplicate bordering data (so-called "halo" or "ghost-cells")
  - data not depending on halo can be computed concurrently with data exchange
  - after halo data arrives, the remaining computation can be completed

```
foreach timestep {
  MPI_Irecv(halo_data)
  MPI_Isend(border_data)
  compute(halo_independent_data)
  MPI_Waitall
  compute(border_data)
}
```



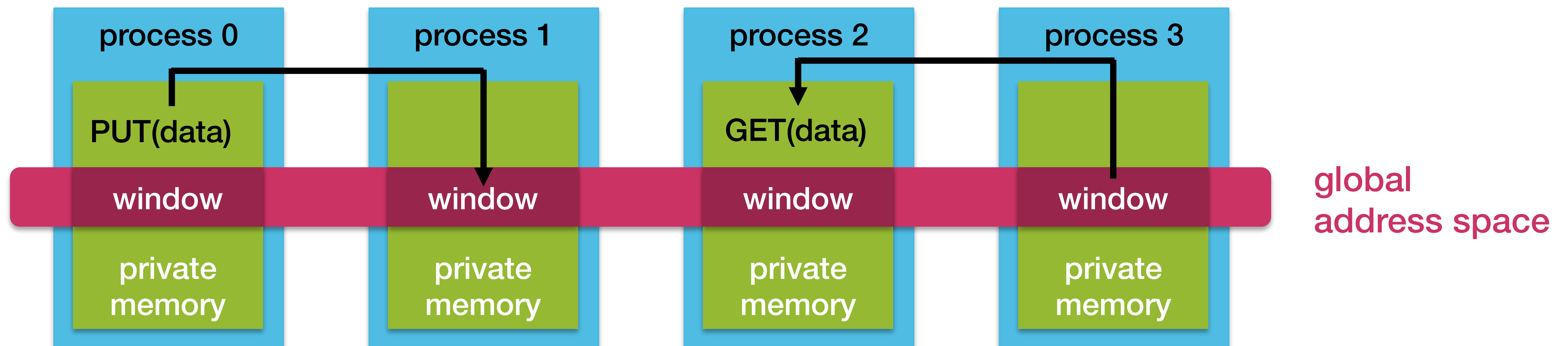
Further Non-Blocking Operations

- MPI-3 has added non-blocking collective operations in addition to the non-blocking point to point communication
  - MPI_Ibcast
  - MPI_Ireduce
  - ...

One-Sided Communication

Overview One-Sided Communication

- Two-sided communication (blocking and non-blocking)
  - two processes are involved: send and matching receive operation
  - combines data transfer and synchronization
- One-sided communication added in MPI-2
  - moves data without requiring the remote process to synchronize
  - each process exposes a section of memory (window) to other processes
  - other processes can directly read or write to this window (global address space)
  - communication is always non-blocking

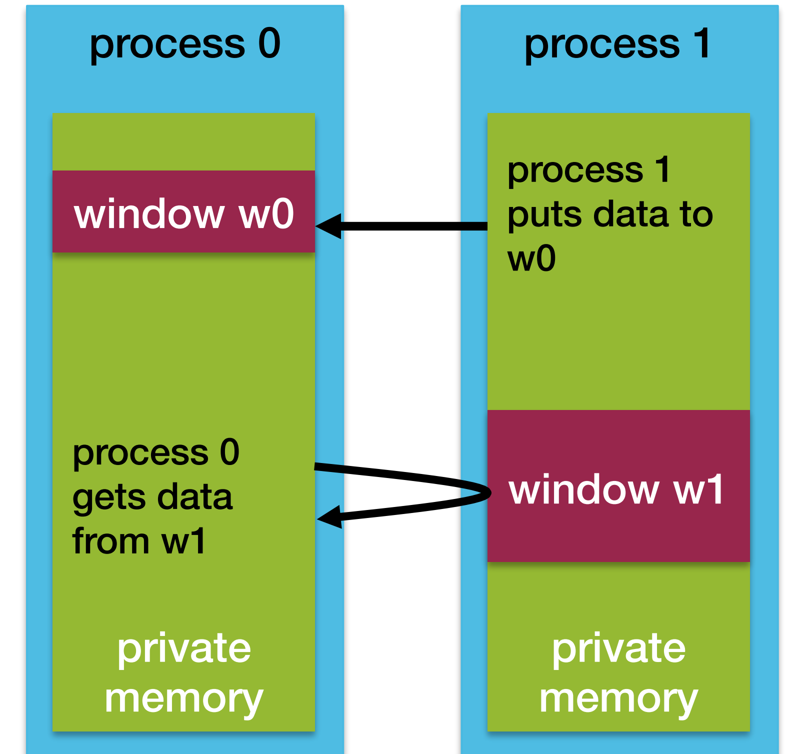


Remote Direct Memory Access (RDMA)

- The data transfers to and from remote memory are very efficient
  - Remote Direct Memory Access (RDMA) mechanism
  - network cards directly access memory and copy data through the network
- Ideally
  - no operating system interaction required
  - close to zero CPU load
  - all handled autonomously by hardware in special HPC networks and network cards
  - zero-copy, i.e. data is moved from main memory to networks without copying to OS kernel
- Operations that are typically supported
  - data copy (send and receive)
  - atomic operations

Motivation and Terminology

- Motivation
  - irregular communication patterns are easier to implement
  - lower overhead due to efficient RDMA transfers and explicit synchronization
- Origin / Target Process
  - processes can initiate a send to a remote location (PUT) and a receive from a remote location (GET), hence the usual terms sender/receiver are ambiguous
  - origin: process which initiates the data movement
  - target: process whose memory is accessed
- Remote Memory Access (RMA) Window
  - section of process memory that is available for one-sided (RMA) communication
  - created by collective calls
  - can differ between processes



Overview: RMA Operations in MPI-2

- MPI_Put
  - copy data from local buffer in origin to remote window in target process
- MPI_Get
  - copy data from remote window in target to local buffer in origin
- MPI_Accumulate
  - use data in local buffer at origin to modify data in window in target process
  - for example, add values in local buffer to remote buffer (one-sided reduction)

Overview: RMA Synchronization in MPI-2

- RMA data access model
  - when is a process allowed to perform RMA operations on target?
  - when is it safe for process Y to read data on target that was written by process X?

- Synchronization takes place in ”epochs” can be started and ended with multiple mechanisms
  - access epoch: origin my access window in different process with RMA operations
  - exposure epoch: target is offering other processes access to its window with RMA operations

- Three RMA synchronization models
  - active target: both origin and target explicitly start and end epochs with collective operations
  - generalized active target: post-start-complete-wait
  - passive target: use lock/unlock operations, no fence operations at target

Allocate Memory and Creating a Window

```
MPI_Win_allocate (MPI_Aint size, int disp_unit, MPI_Info info,
  MPI_Comm comm, void* baseptr, MPI_Win *win)
```

- Allocate new memory and expose it as an RMA window
  - collective operation that needs to be called by all processes in communicator
- Parameters
  - size: size of local data in bytes
  - disp_unit: local unit size for displacements in bytes
  - info: hints to MPI implementation for improving efficiency
  - comm: MPI communicator
  - base: returns initial address of created window
  - win: returns handle for identifying RMA window

Creating a Window to Existing Memory

```
MPI_Win_create (void *base, MPI_Aint size, int disp_unit, MPI_Info info,
  MPI_Comm comm, MPI_Win *win)
```

- Expose an existing memory region in an RMA window
  - collective operation that needs to be called by all processes in communicator
  - memory must be previously allocated with `MPI_Alloc_mem`
- Parameters
  - base: pointer to local data to expose
  - size: size of local data in bytes
  - disp_unit: local unit size for displacements in bytes
  - info: hints to MPI implementation for improving efficiency
  - comm: MPI communicator
  - win: returns handle for identifying RMA window

- If window is no longer used, it can be deallocated with `MPI_Win_free(win)`

MPI_Put

```
MPI_Put (const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
 int target_rank, MPI_Aint target_disp, int target_count,
 MPI_Datatype target_datatype, MPI_Win win)
```

- Move data from origin to target
- Parameters
  - origin_addr: pointer to local data to be sent to target
  - origin_count, origin_datatype: number of elements to put and its MPI data type
  - target_rank: rank of target process
  - target_disp: displacement from the beginning of the target window
  - target_count, target_datatype: number of elements and data type in target
  - win: RMA window to be used

MPI_Get

```
MPI_Get (void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
 int target_rank, MPI_Aint target_disp, int target_count,
 MPI_Datatype target_datatype, MPI_Win win)
```

- Move data from origin to target
- Parameters
  - origin_addr: initial address of origin buffer where data will be copied to
  - origin_count, origin_datatype: number of elements to get and its MPI data type
  - target_rank: rank of target process
  - target_disp: displacement from the beginning of the target window
  - target_count, target_datatype: number of elements and data type in target
  - win: RMA window to be used

MPI_Accumulate

```
MPI_Accumulate (void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
 int target_rank, MPI_Aint target_disp, int target_count,
 MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

- Update data at target atomically, generalization of a put
  - reduces origin and target into the target buffer using op as reduction operation
- Parameters (like `MPI_Put`)
  - op: MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, ...
    MPI_REPLACE acts like an MPI_Put

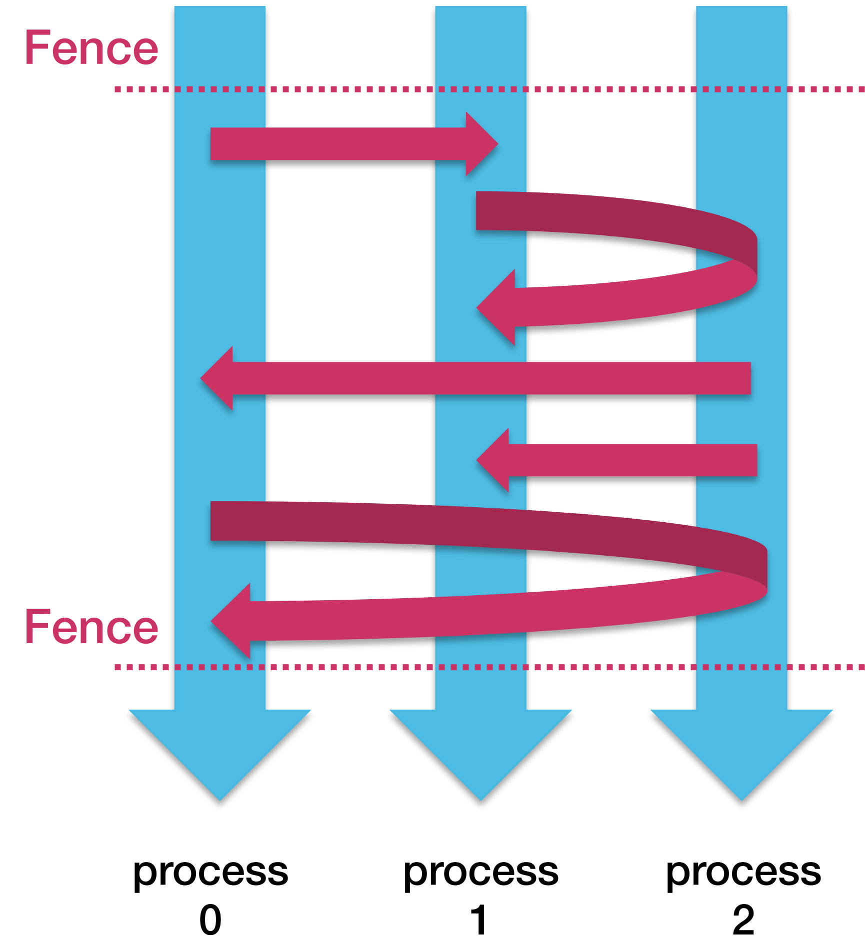
Ordering of RMA Operations

- Ordering of Get/Put operations is not guaranteed
  - result of concurrent Put to same location is undefined
  - result of Get is undefined if concurrent Put or Accumulate to same operations are active
- Results of concurrent Accumulates from same process to same location is defined
  - complete in the order of issue

Active Target Synchronization with Fences

```
MPI_Win_fence (int assert, MPI_Win win)
```

- Collective synchronization method for starting and ending both access and exposure epochs on all processes in window
  - first call to `MPI_Win_fence` starts the epoch
  - all processes can perform PUT/GET/ACCUMULATE operations now
  - all processes must call `MPI_Win_fence` again to close the epoch
- All operations complete at the second fence synchronization
- Within the epoch, all processes perform RMA operations on all targets



Active Target Synchronization with Fences (2)

- Assert argument for `MPI_Win_fence` can improve performance by specifying hints to runtime
  - MPI_MODE_NOSTORE: the local window was not updated by local stores (or local get or receive calls) since last synchronization
  - MPI_MODE_NOPUT: the local window will not be updated by put or accumulate calls after the fence call, until the following (fence) synchronization
  - MPI_MODE_NOPRECEDE: the fence does not complete any sequence of locally issued RMA calls
  - MPI_MODE_NOSUCCEED: the fence does not start any sequence of locally issued RMA calls

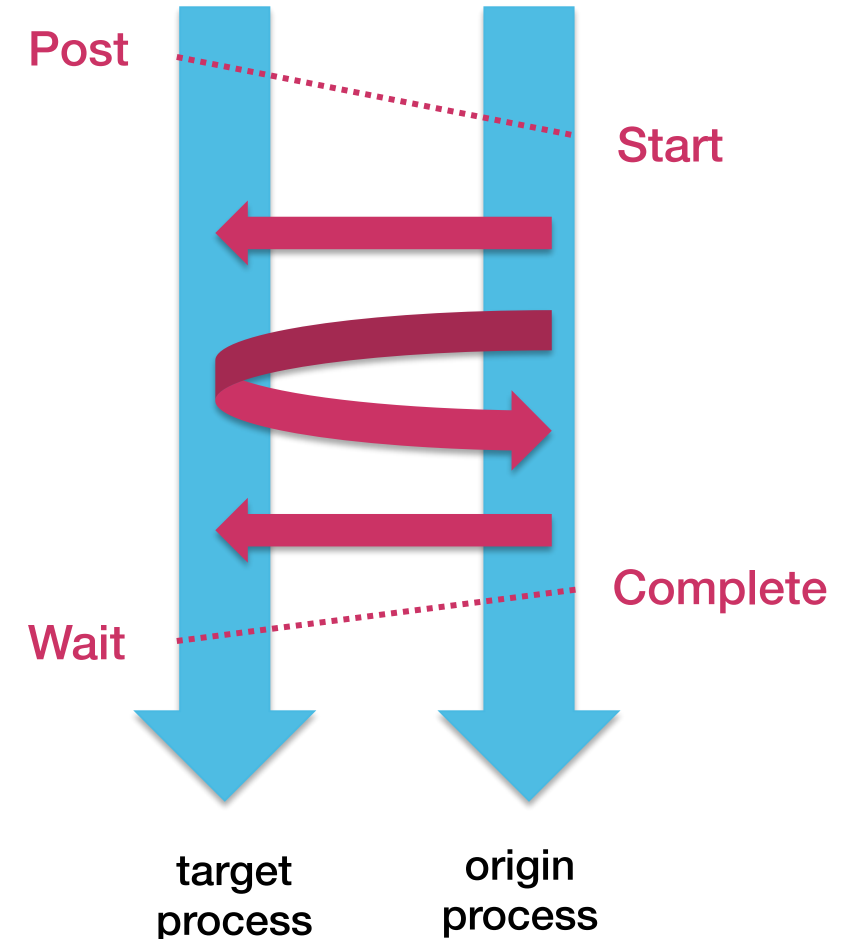
Example: MPI_Put with Active Target Synchronization

```
int data;
MPI_Win window;

data = rank;
// Create window
MPI_Win_create(&data, sizeof(int), sizeof(int),
  MPI_INFO_NULL, MPI_COMM_WORLD,&window);
...

MPI_Win_fence(0, window);
if (rank == 0)
  MPI_Put(&data, 1, MPI_INT, 1, 0, 1, MPI_INT, window);
MPI_Win_fence(0, window);
...
MPI_Win_free(&window);
```

Generalized Active Target Synchronization

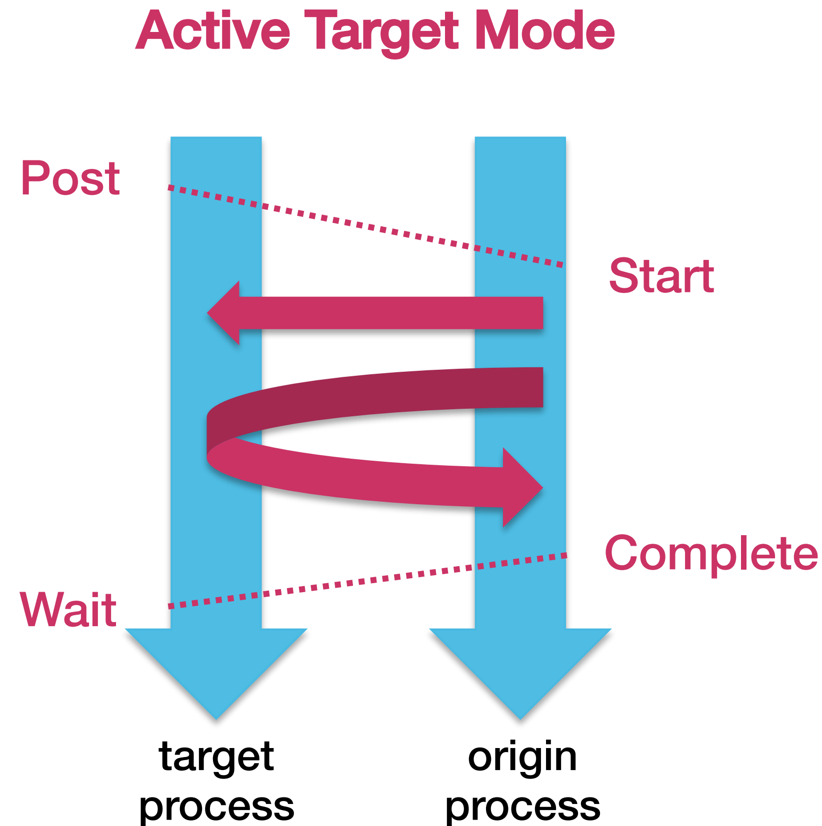
```
MPI_Win_post/start(MPI_Group grp, int assert, MPI_Win win)
MPI_Win_complete/wait(MPI_Win win)
```

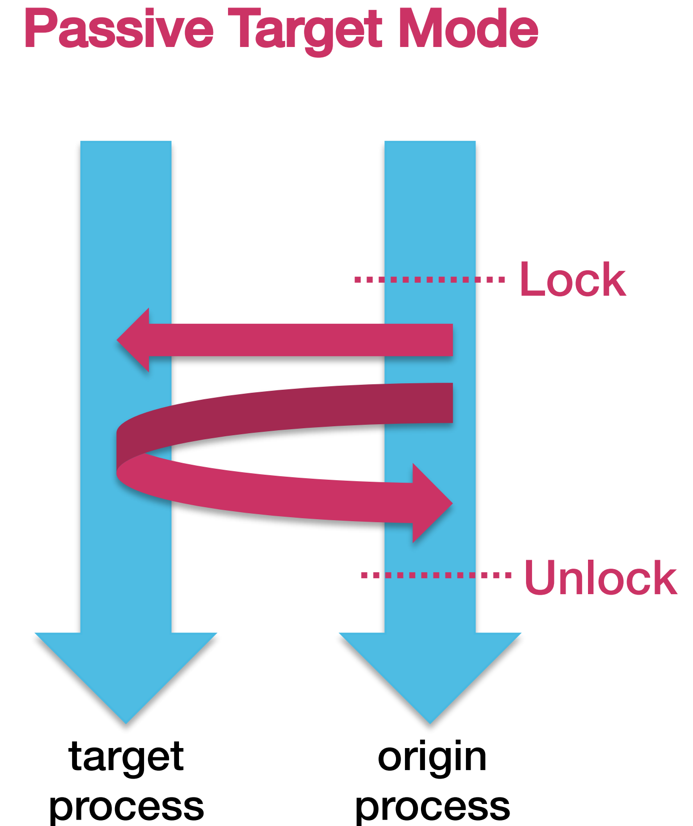
- Generalizes synchronization with fences but origin and target specify with whom they communicate
- Target: Exposure epoch
  - opened with MPI_Win_post
  - closed with MPI_Win_wait
- Origin: Access epoch
  - opened with MPI_Win_start
  - closed with MPI_Win_complete
- Synchronization methods may block to enforce Post-Start-Complete-Wait ordering



Passive Target Synchronization with Lock/Unlock

- Target does not participate in synchronization
  - true passive, one-sided asynchronous communication
  - shared memory-like model





Passive Target Synchronization with Lock/Unlock (2)

```
MPI_Win_lock/lock_all (int lock_type, int rank, int assert, MPI_Win win)
MPI_Win_unlock/unlock_all (int rank, MPI_Win win)
MPI_Win_flush/flush_local(int rank, MPI_Win win)
```

- `MPI_Win_lock/unlock`: start/end a passive mode epoch for rank
  - only called at origin (not target)
  - multiple passive target epochs to different processes can be active
  - concurrent epochs to same process not allowed
  - `lock_all/unlock_all` variants lock access to all processes in win with type MPI_LOCK_SHARED
- lock_type
  - MPI_LOCK_SHARED: other process using shared can access concurrently
  - MPI_LOCK_EXCLUSIVE: no other processes can access concurrently
- `MPI_Win_flush`
  - complete all outstanding RMA operations at origin and target, after completion target or other process can read consistent data in window
- `MPI_Win_flush_local`
  - complete all local RMA operations to the target process

How to Chose a Synchronization Model

- RMA communication has lower overheads than MPI_Send/Recv
  - two-sided : message matching, queuing, buffering, waiting for readiness to receive, etc.
  - one-sided: no message matching and buffering, always ready to receive
  - RDMA makes transfer even more efficient
- Active mode
  - useful for synchronizing after bulk data exchange, e.g. halo regions
- Passive mode
  - useful for moving data with unstructured access and synchronization pattern
  - distributed shared memory in global address space
  - lock/unlock: when exclusive epochs are needed
  - lock_all/unlock_all: when only shared epochs are needed

Hybrid Parallel Programming

MPI and Threads

- MPI dates back to time when CPUs only had a single (or very few) cores
  - single thread per rank
  - distributed memory
  - core-level parallelism must be exploited by running multiple MPI ranks per CPU
- Advantages of MPI-only programs
  - same code and programming model everywhere (reduce software complexity)
  - memory locality is also favorable for multi-cores
  - simple job scheduling, ranks can be placed anywhere
- Advantages of using multi-threading on node and MPI between nodes
  - eliminate need for domain decomposition on node
  - automatic memory sharing, coherency and high local bandwidth
  - faster synchronization routines

Thread-Safety of MPI (1)

- MPI can be used in multi-threaded environments
  - application must explicitly state, which level of thread-safety is required
  - higher degree of thread safety, comes with higher overheads
- Levels of thread safety
  - MPI_THREAD_SINGLE: only one thread will execute per rank
  - MPI_THREAD_FUNNELD: each rank may be multi-threaded but only the thread that called MPI_Init_thread is allowed to make MPI calls
  - MPI_THREAD_SERIALIZED: each rank my be multi-threaded but one thread at a time makes MPI calls
  - MPI_THREAD_MULTIPLE: each rank may be multi-threaded and multiple threads may call MPI at once without restrictions
- Increasing thread-safety levels include each other, i.e. an application that requires MPI_THREAD_FUNNELED runs with MPI_THREAD_SERIALIZED too

Thread-Safety of MPI (2)

- The application requests the desired thread-safety level using a variant of MPI_Init

```
MPI_Init_thread(int* argc, char** argv[], int required, int* provided)
```

  - required: specifies the desired thread-safety level, e.g. MPI_THREAD_FUNNELED
  - provided: returns the available level of thread support
- MPI implementations are not required to support higher levels than MPI_THREAD_SINGLE, hence provided may be different from requested
- Multi-threaded programs must call MPI_Init_thread (because MPI_Init implies MPI_THREAD_SINGLE)
- Levels FUNNELED and SERIAL are typically sufficient for bulk synchronous parallel programming (in particular OpenMP work sharing)
- Unrestricted multi-threading and MPI in MPI_THREAD_MULTIPLE mode, is tricky and can lead to very hard to find bugs related to thread-scheduling and race conditions (out of scope for this lecture)

MPI + OpenMP with MPI_THREAD_FUNNELED

- All MPI calls are made by the OpenMP master thread, either
  - outside OpenMP parallel region
  - or in an OpenMP master region within an OpenMP parallel region
- Example: MPI call outside of parallel region

```
int main(int argc, char * argv[]) {
  int provided;
  int a[N] = ...

  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
  if (provided < MPI_THREAD_FUNNELED) MPI_Abort(MPI_COMM_WORLD,1);

  // no MPI calls within this parallel region
  #pragma omp parallel for
  for(int i=0; i<N; i++){
     a[i] = f(i);
  }
  // ouside parallel region, MPI calls can be made
  MPI_Send(...);
  MPI_Finalize();
  return 0;
}
```

MPI + OpenMP with MPI_THREAD_FUNNELED (2)

- Example: MPI call from within a parallel region

```
int main(int argc, char * argv[]) {
  int provided;
  int a[N] = ...

  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
  if (provided < MPI_THREAD_FUNNELED) MPI_Abort(MPI_COMM_WORLD,1);

  // MPI calls only from within master region
  #pragma omp parallel for
  for(int i=0; i<N; i++){
     a[i] = f(i);
    if (i % 10) {
      #pragma omp barrier
      #pragma omp master
      MPI_Send(...);
      #pragma omp barrier
    }
  }
  MPI_Finalize();
  return 0;
}
```

- OpenMP master region has no implied barrier
- Explicit barrier needed to make sure memory state is consistent, in particular all buffers to be communicated with MPI are consistent before and after the MPI calls
- Second barrier also implies cache flush

Overlapping Computation and Communication

- Example: halo communication for stencils (e.g. Conway's Game of Life)
  - how do we create on thread for communication and let the others to the work?
  - here: create threads with nested parallelism, alternative: use OpenMP tasks

```
#pragma omp parallel num_threads(2)
{
  if(!omp_get_thread_num()) {
    MPI_Send/Recv(..)  // one thread exchanges halo data
  } else {
    #pragma omp parallel for num_threads(15)
    for{int i=0; i<N; i++) {
       // other threads do work not involving halos
    }
}

#pragma omp parallel num_threads(16)
{
   for{int i=0; i<N; i++) {
       // all threads work now on remaining data that need halos
    }
}
```

Running Hybrid MPI + OpenMP Programs on Oculus

- Example: hybrid MPI + OpenMP program and a resource budget of 64 cores
  - reminder: regular Oculus nodes have 2 sockets with 8 core CPUs, i.e. 16 cores per node (resource type 'norm')
- Variant 1: 64 MPI ranks (MPI-only) on 4 nodes with 16 MPI ranks per node

```
ccsalloc --res=rset=4:mpiprocs=16:ncpus=16:norm=true:place=:excl
```

- Variant 2: 4 nodes, 4 MPI ranks (1 per node), 16 OpenMP threads per MPI rank

```
ccsalloc --res=rset=4:ncpus=16:mpiprocs=1:ompthreads=16,place=:excl
```

- Variant 3: 4 nodes, 1 MPI ranks per CPU (2 per node), 8 OpenMP thr. per MPI rank

```
ccsalloc --res=rset=4:ncpus=16:mpiprocs=2:ompthreads=8,place=:excl
```

- Variant 4: 4 nodes, 2 MPI ranks per CPU, 4 OpenMP threads per MPI rank

```
ccsalloc --res=rset=4:ncpus=16:mpiprocs=4:ompthreads=4,place=:excl
```

- Variant 5: 16 chunks with 1 MPI rank and 4 OpenMP threads per MPI rank (let CCS decide whether

```
ccsalloc --res=rset=16:ncpus=4:mpiprocs=1:ompthreads=4
```

Acknowledgements

- This lecture is based materials from these sources
  - CSC.fi course materials on Advanced MPI
  - SC17 tutorial on Advanced MPI Programming

Change Log

- 1.0.2 (2018-01-23)
  - cosmetics
  - add warning to slide 18
  - fix struct declaration on slide 18 and 19 (last field is double mass, not int type)
- 1.0.1 (2018-01-22)
  - added section on hybrid parallel programming
- 1.0.0 (2018-01-16)
  - initial version of slides