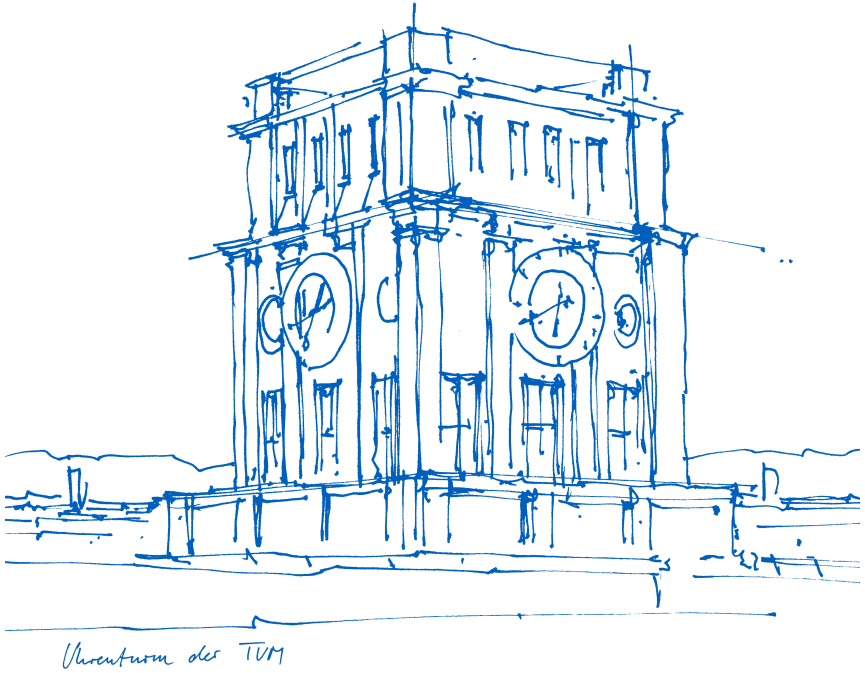
Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Exercises: Amir Raoofy
Technische Universität München
Fakultät für Informatik

Lecture 9:
Advanced MPI
Hybrid Programming



Summary From Last Time

MPI as the most widely used HPC standard for message passing
- Currently in version 3.1 with wide range of functionality
- Continues to evolve through the open standardization body MPI Forum

Basic message protocols
- Receive queues and Unexpected Message Queues
- Eager vs. Rendezvous communication

Non-blocking communication
- Functions return `MPI_Request` object
- Can be waited or tested for completion

Collective operations
- Operations involving all MPI processes in a communicator
- Enable faster and more efficient implementations

Communicator creation
- Duplication enables library isolation
- Communicator splitting to create subcommunicators

Central MPI Types and Objects

`MPI_Comm` – Communicators
- Group of MPI processes
- Communication context

`MPI_Request` – handle for non-blocking communication
- Symmetry of blocking and non-blocking functions
- Currently being made pervasive throughout the standard

`MPI_Info` – object
- Object object type for a key/value store
- Routines to create, query, set, destroy
- Passed to many routines to pass additional information

`MPI_Datatype` – structured datatypes for communication
- Describe format of any communicated data
- Passed into send/recv/collective call
- Describe message in combination with a “count”

MPI Datatypes

Datatypes are used to specify data layouts
- Describe data touched by MPI
- Basic types available matching the base language
- Ability to construct complex structures
  - Hierarchical
  - Gaps

Three main purposes
- Enable MPI implementations to introduce optimization
- Enable implicit conversions of data representations between sender and receiver
- Scatter/Gather like functionality for data transfers

Datatypes allow to serialize/deserialize data layouts into a message stream
- Can be handled as a message stream
- Enables different datatypes on sender/receiver side
  - Only serialization must match
  - Enables complex data reshuffling operations

Common Basic MPI Data Types (C Versions)

MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_C_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_BYTE	
MPI_PACKED	

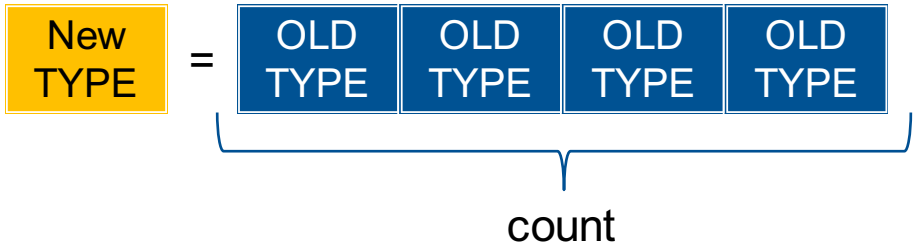
Derived MPI Datatypes

Several MPI functions available to create arbitrary layouts
- Recursive specification possible
- Declarative specification of data-layout

Constructor functions
- Additional arguments to describe new type
- Input: existing datatype (basetype for this operation)
- Output: newly created derived datatype

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

New datatype is “count” contiguous elements of ”oldtype”



Datatype Management

Datatype handle

```
MPI_Datatype
```

Datatype construction

Set of routines (see previous and next slides)

Datatype commit

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

necessary before type can be used,

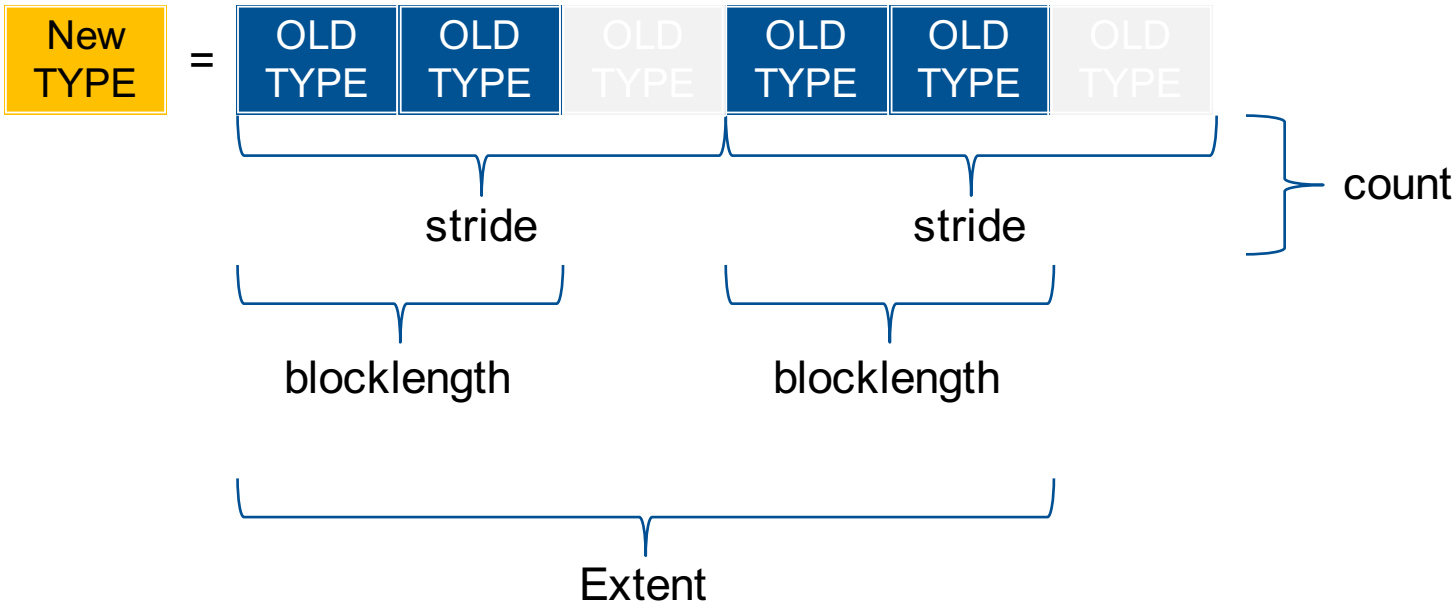
but uncommitted types can be used to create new types

Datatype free

```
int MPI_Type_free(MPI_Datatype *datatype)
```

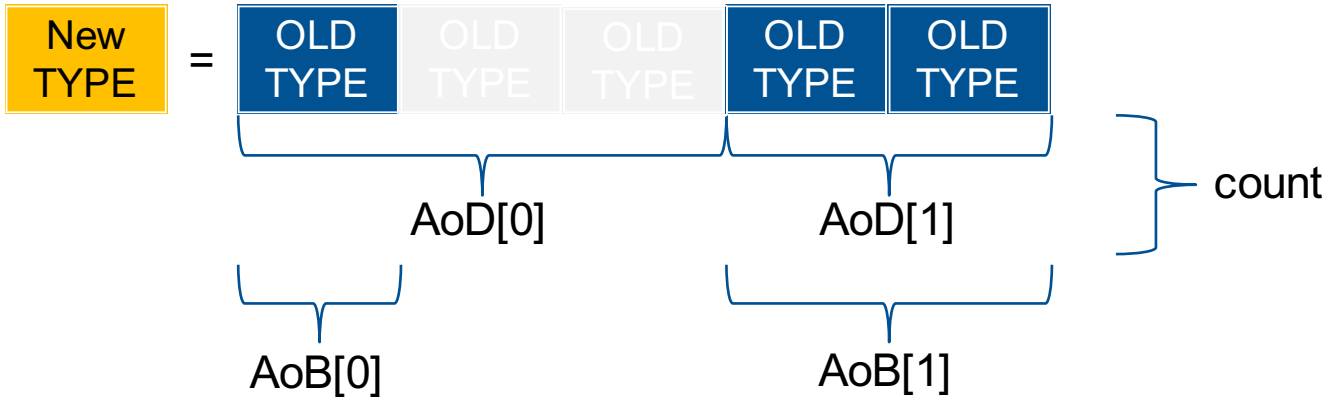
Constructing a Vector Datatype

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



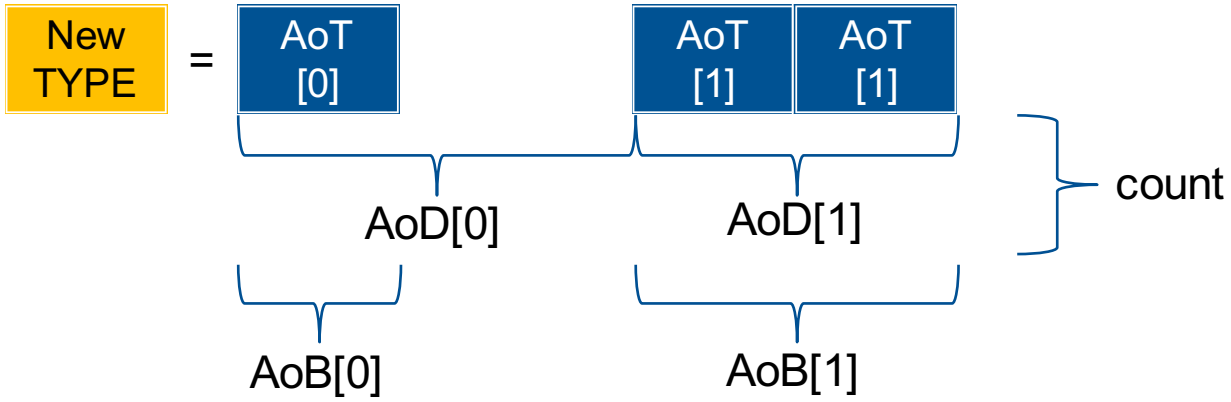
Constructing an Indexed Datatype

```
int MPI_Type_indexed(int  count,
                     const int array_of_blocklengths[],     (AoB)
                     const int array_of_displacements[],    (AoD)
                     MPI_Datatype  oldtype,
                     MPI_Datatype  *newtype)
```



Creating a Struct Datatype

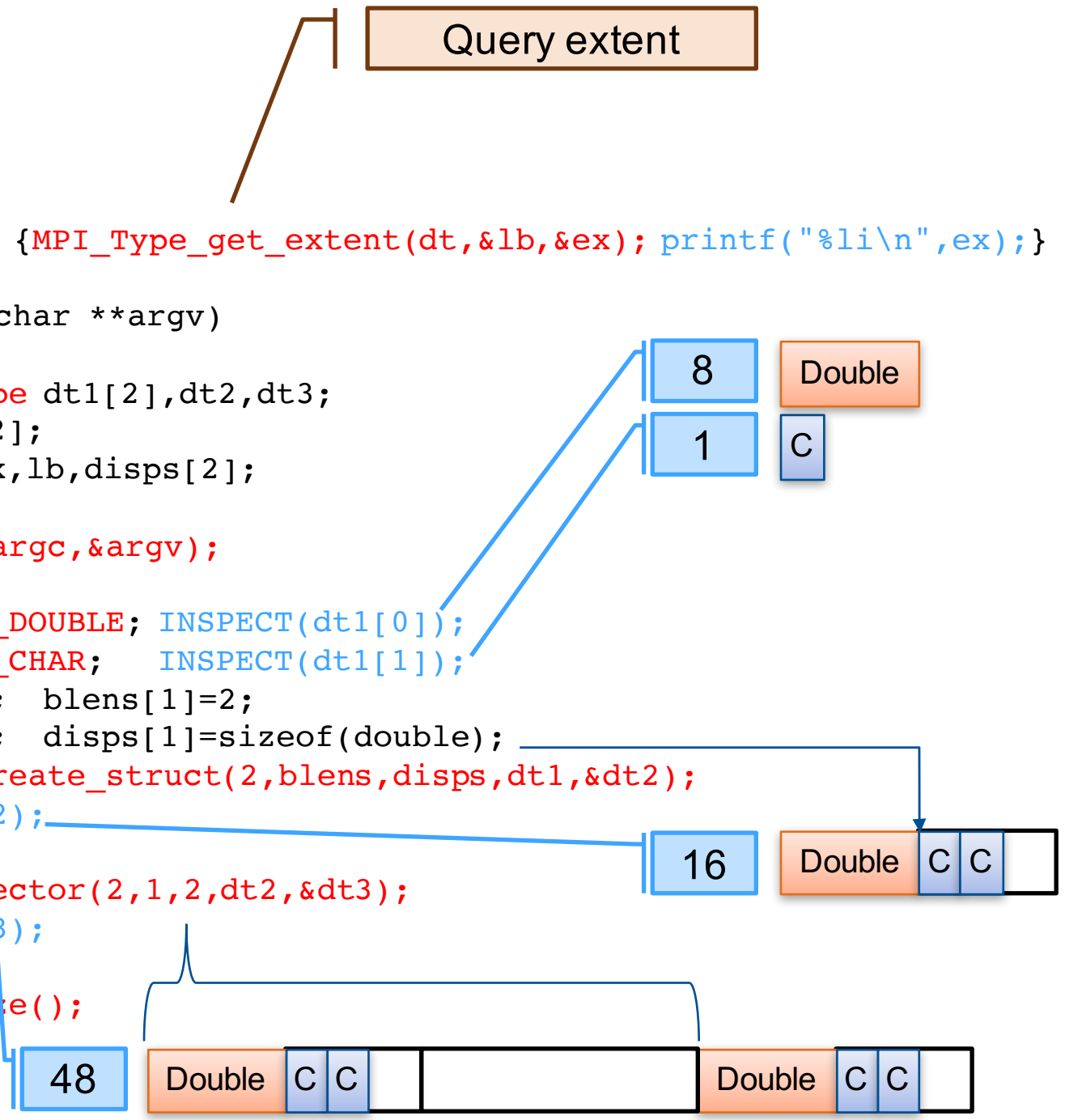
```
int MPI_Type_create_struct(int  count,
                const int array_of_blocklengths[],          (AoB)
                const MPI_Aint array_of_displacements[],    (AoD)
                const MPI_Datatype array_of_types[],        (AoT)
                MPI_Datatype *newtype)
```



Note: Displacements in Bytes (`MPI_Aint`)

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define INSPECT(dt) {MPI_Type_get_extent(dt,&lb,&ex); printf("%li\n",ex);}

int main(int argc, char **argv)
{
        MPI_Datatype dt1[2],dt2,dt3;
        int blens[2];
        MPI_Aint ex,lb,disps[2];

        MPI_Init(&argc,&argv);

        dt1[0]=MPI_DOUBLE; INSPECT(dt1[0]);
        dt1[1]=MPI_CHAR;   INSPECT(dt1[1]);
        blens[0]=1;  blens[1]=2;
        disps[0]=0;  disps[1]=sizeof(double);
        MPI_Type_create_struct(2,blens,disps,dt1,&dt2);
        INSPECT(dt2);

        MPI_Type_vector(2,1,2,dt2,&dt3);
        INSPECT(dt3);

        MPI_Finalize();
}
```



Constructing a Subarray Datatype

```
int MPI_Type_create_subarray(int ndims,
                const int array_of_sizes[],
                const int array_of_subsizes[],
                const int array_of_starts[],
                int order,
                MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Example:

ndims = 2

Sizes = 5,5
Subsizes = 3,2
Starts = 0,1

order = `MPI_ORDER_C`

Some Other Datatype Construction Functions

`MPI_Type_dup`
- Duplication of datatype

`MPI_Type_create_hvector`
- Like MPI_Type_create_vector only with Byte-sized strides

`MPI_Type_create_hindexed`
- Like MPI_Type_create_indexed only with Byte-sized strides

`MPI_Type_create_indexed_block`
- Like MPI_Type_create_indexed only with fixed block size

`MPI_Type_create_hindexed_block`
- Like MPI_Type_create_indexed_block only with Byte-sized strides

`MPI_Type_create_darray`
- Create a distributed array datatype

Inspecting Datatypes

```
int MPI_Type_get_envelope(MPI_Datatype datatype,
              int *num_integers, int *num_addresses,
              int *num_datatypes, int *combiner)
```

Query metadata information on a type

- Combiner: type of constructor used
- Returns number of elements used in construction

```
int MPI_Type_get_contents(MPI_Datatype datatype,
              int max_integers, int max_addresses,
              int max_datatypes, int array_of_integers[],
              MPI_Aint array_of_addresses[],
              MPI_Datatype array_of_datatypes[])
```

Query actual type information

- Pass in maximal size of arrays (too avoid overflow)
- Returns information used to construct datatype

Example: Inspecting Datatypes

```
#define LARGE 1000
int ni, na, nd, combiner, i[LARGE];
MPI_Aint a[LARGE];
MPI_Datatype type, d[LARGE];

/* construct datatype type (not shown) */

MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE))
{
  fprintf("ni, na, or nd = %d %d %d returned by ", ni, na, nd);
  fprintf(stderr, "MPI_Type_get_envelope is too\n");
  MPI_Abort(MPI_COMM_WORLD, 99);
}

MPI_Type_get_contents(type, ni, na, nd, i, a, d);
```

Datatype Discussions

MPI allows the creation of arbitrary datatypes

- Representation of complex datatypes
- Implicit Scatter/Gather operations
- Options for optimization in the MPI library

When to use datatypes?

- Messages with multiple datatypes
- Repeated access to spread out data structures
- Simplification of code

BUT: Warning

- Datatype creation/destruction can lead to overhead
- Not many MPI implementations are optimized for datatypes

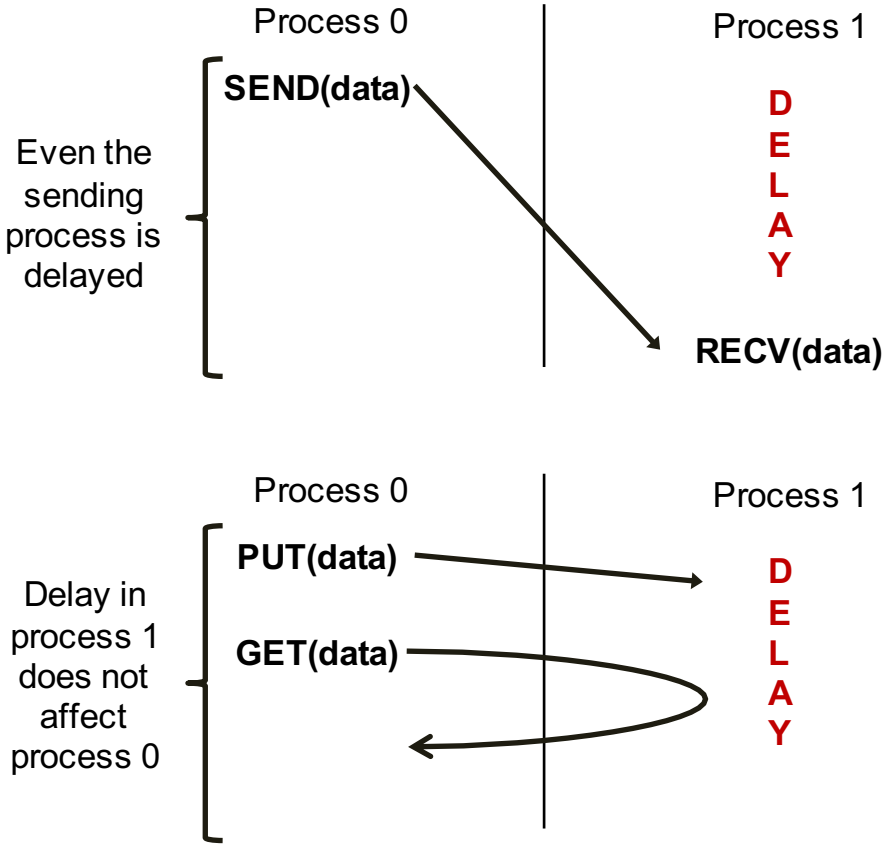
Communication Modes

So far, all communication has been ”two-sided”
- Explicit sender and receiver for P2P
- All processes call collectives

Advantages
- Simple concept
- Clear communication locations
- Implicit synchronization

Disadvantages
- Implicit synchronization
- Requires active involvement on both sides
- Receiver can delay sender

Alternative: One-sided communication
- Direct memory access to other memories
- Receiver does not get involved
- Requires extra synchronization

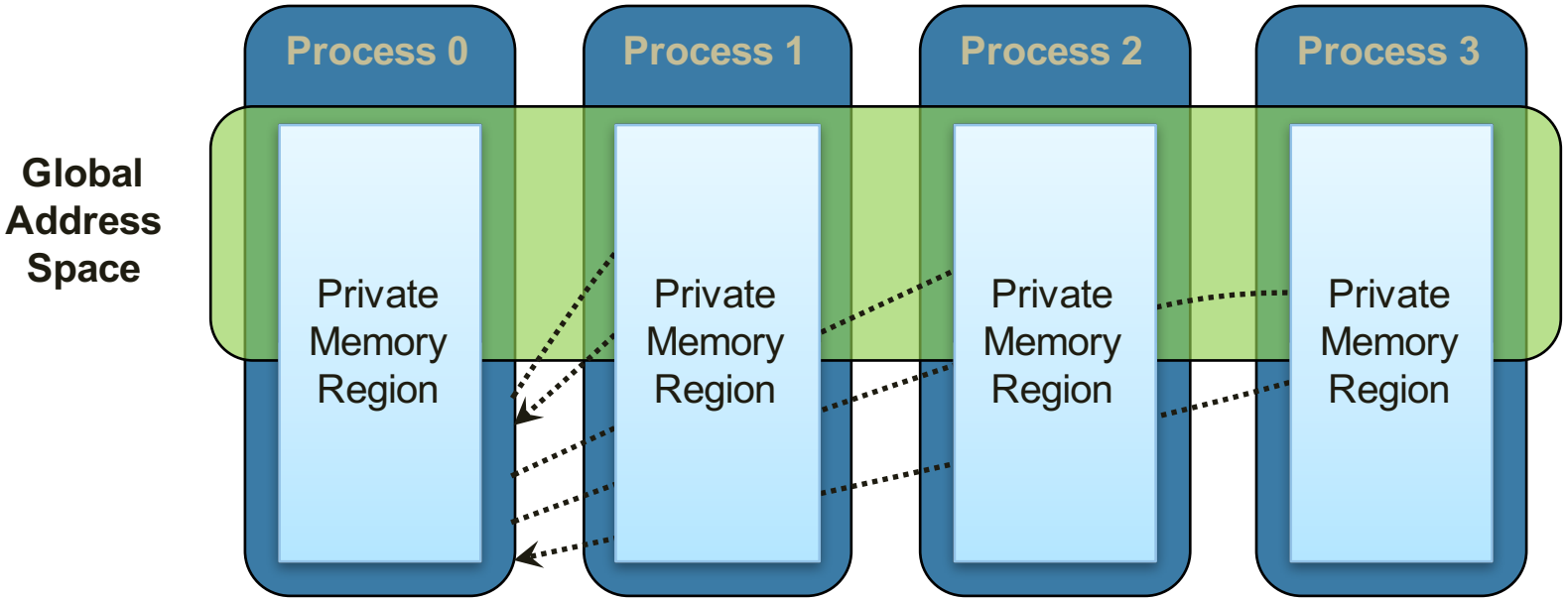


Material on RMA on the following slides from Tutorial “Advanced MPI”, ISC (06/16/2013) – Thakur, Hoefler, Schulz

One-sided Communication

The basic idea of one-sided communication models is to decouple data movement with process synchronization

- Should be able move data without requiring that the remote process synchronize
- Each process exposes a part of its memory to other processes
- Other processes can directly read from or write to this memory



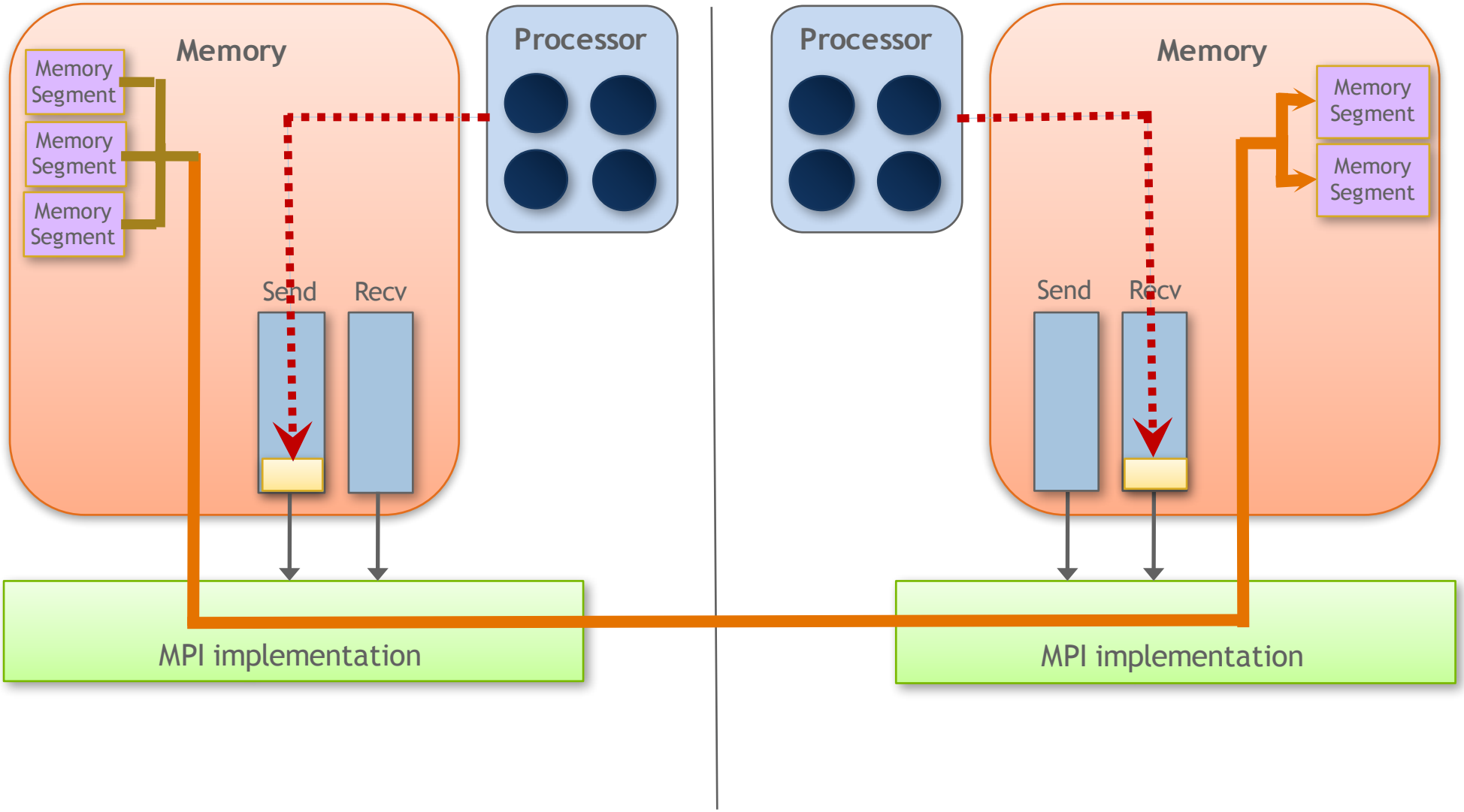
Remote Memory Access (RMA) in MPI

Introduction of shared memory programming to MPI
- Borrowing many concepts from shared memory models
- Limited by exposing only parts of the memory
- Limited by missing automatic consistency

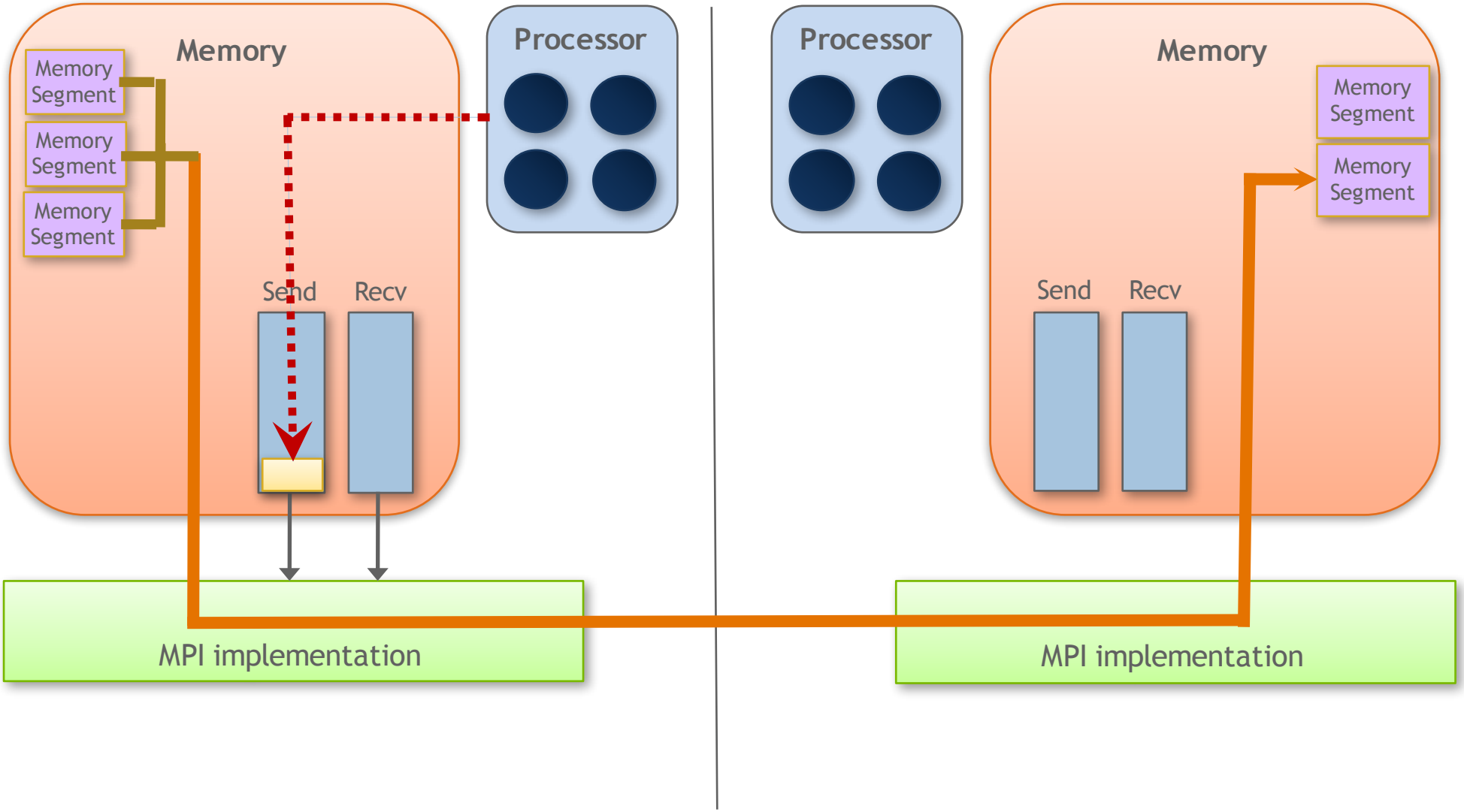
Differences to models like Pthreads and OpenMP
- Specific memory regions accessible only
- Still running in distinct MPI processes, which assume shared nothing
- No transparent access to remote memory
  - Explicit get and put calls
  - Still think of it as messages
- Still a library implementation with the need for hardware support

Necessary Steps to Use MPI RMA
- Making memory accessible
- Reading, writing and updating remote memory
- Adding needed data synchronization

Two-sided Communication Example



One-sided Communication Example



Creating Public Memory

Any memory created by a process is, by default, only locally accessible
- Separate MPI processes
  - Difference to shared memory programming models
- Example: X = malloc(100);

Once memory is created, user has to declare memory region as remotely accessible
- Done by explicit MPI call
- MPI terminology for remotely accessible memory is a “Window”

A group of processes collectively create a “Window”
- Tied to a communicator as the window context
- “Windows” are abstractions for limited memory regions

Once a memory region is declared as remotely accessible
- All processes in the window can read/write data to this memory
- Accesses don’t explicitly synchronizing with the target process

Several Window Creation Models

`MPI_WIN_CREATE`
- Make existing/allocated memory regions remotely accessible
- Data addressing relative to base address of window

`MPI_WIN_ALLOCATE`
- Allocate memory region and make it accessible
- Data addressing relative to base address of window

`MPI_WIN_CREATE_DYNAMIC`
- No buffer yet, but will have one in the future
- Additional calls to attach memory to window
  - `MPI_WIN_ATTACH`
    Add memory to a dynamic window
  - `MPI_WIN_DETACH`
    Remove memory from a dynamic window
- Data addressing based on absolute address

Making Local Memory Accessible

Expose a region of memory in an RMA window
- Only data exposed in a window can be accessed with RMA operations
- Access from remote processes only through RMA operations

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

IN base	*pointer to local data to expose*
IN size	*size of local data in bytes (nonnegative integer)*
IN disp_unit	*local unit size for displacements, in bytes (positive integer)*
IN Info	*info argument (handle)*
IN comm	*communicator (handle)*
OUT win	*newly created window (handle)*

Notes
- Collective operation by all MPI processes in comm
- Every process can contribute memory (or pass size=0)
- Info argument can be used to pass assumptions about the window

Example with MPI_WIN_CREATE

```
int main(int argc, char ** argv)
{
    int *a;
    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* create private memory in every MPI process */
    a = (void *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000, sizeof(int), MPI_INFO_NULL,
                           MPI_COMM_WORLD, &win);

     /* Array ‘a’ is now accessibly by all processes in MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize();
    return 0;
}
```

MPI_WIN_ALLOCATE

Create a remotely accessible memory region in an RMA window
- Only data exposed in a window can be accessed with RMA ops.

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

IN size	*size of local data in bytes (nonnegative integer)*
IN disp_unit	*local unit size for displacements, in bytes (positive integer)*
IN info	*info argument (handle)*
IN comm	*communicator (handle)*
OUT base	*pointer to exposed local data*
OUT win	*newly created window (handle)*

Notes
- Collective operation by all MPI processes in comm
- Every process can contribute memory (or pass size=0)
- Info argument can be used to pass assumptions about the window

Example with MPI_WIN_ALLOCATE

```
int main(int argc, char ** argv)
{
    int *a;
    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* collectively create remotely accessible memory in the window */

    MPI_Win_allocate(1000, sizeof(int), MPI_INFO_NULL,
                     MPI_COMM_WORLD, &a, &win);

     /* Array ‘a’ is now accessible from all processes in MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize();
    return 0;
}
```

MPI_WIN_CREATE_DYNAMIC

Create an RMA window, to which data can later be attached

- Only data exposed in a window can be accessed with RMA ops

```
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm,
                           MPI_Win *win)
```

IN info	*info argument (handle)*
IN comm	*communicator (handle)*
OUT win	*newly created window (handle)*

Notes

- Collective operation by all MPI processes in comm
- Info argument can be used to pass assumptions about the window
- Application can dynamically attach memory to this window
- Can access data on this window only after a memory region has been attached

Attaching and Detaching Memory

```
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
```

IN win	*window to attach memory to (handle)*
IN base	*pointer to base of memory region*
IN size	*size if memory region*

Notes:

- Local operation
- User has to ensure memory is attached before accessing it

```
int MPI_Win_detach(MPI_Win win, void *base)
```

IN win	*window to attach memory to (handle)*
IN base	*pointer to base of memory region*

Notes:

- Local operation
- Has to match a previously attached memory region

Example with MPI_WIN_CREATE_DYNAMIC

```
int main(int argc, char ** argv)
{
    int *a;
    MPI_Win win;

    MPI_Init(&argc, &argv);
    MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* create private memory in every MPI process */
    a = (void *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* locally declare memory as remotely accessible */
    MPI_Win_attach(win, a, 1000);

     /* Array ‘a’ is now accessible from all processes in
        MPI_COMM_WORLD*/

    /* undeclare public memory */
    MPI_Win_detach(win, a);
    MPI_Win_free(&win);

    MPI_Finalize();
    return 0;
}
```

Data Movement

Read, write and atomically modify data in remotely accessible memory regions

- `MPI_GET`
- `MPI_PUT`
- `MPI_ACCUMULATE`
- `MPI_GET_ACCUMULATE`
- `MPI_COMPARE_AND_SWAP`
- `MPI_FETCH_AND_OP`

Explicit calls for data transfers

- In contrast to true shared memory models
- Think of them as explicit data transfers

Have to have target/remote memory exposed

Data Movement / Get

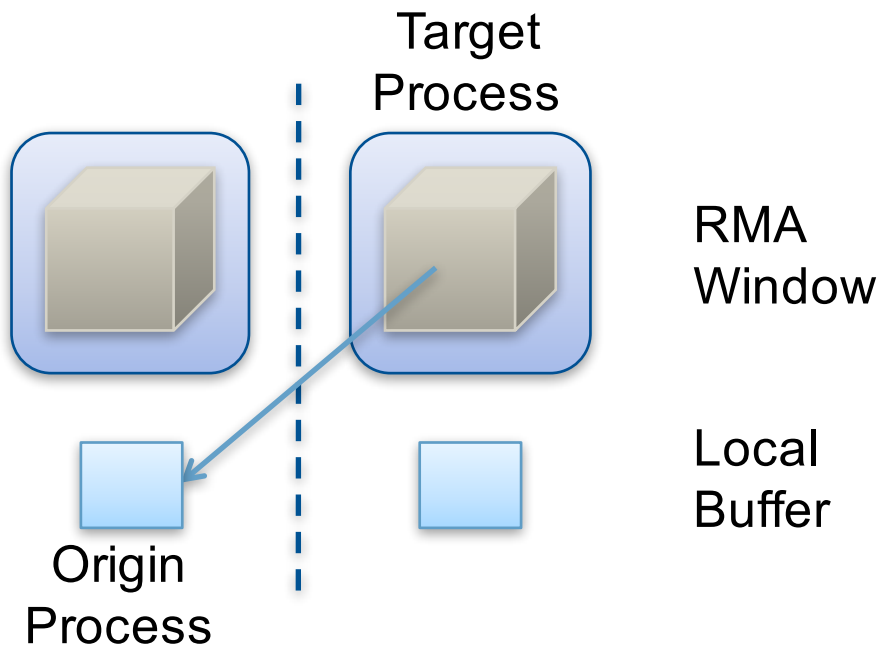
Move data to origin, from target

```
int 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)
```

Similar to a receive operation, but
data description triples for origin **and** target

Local/origin
- Address (origin_addr)
- Count/datatype

Remote/target
- Displacement (relative to window)
- Count/datatype



Data Movement / Put

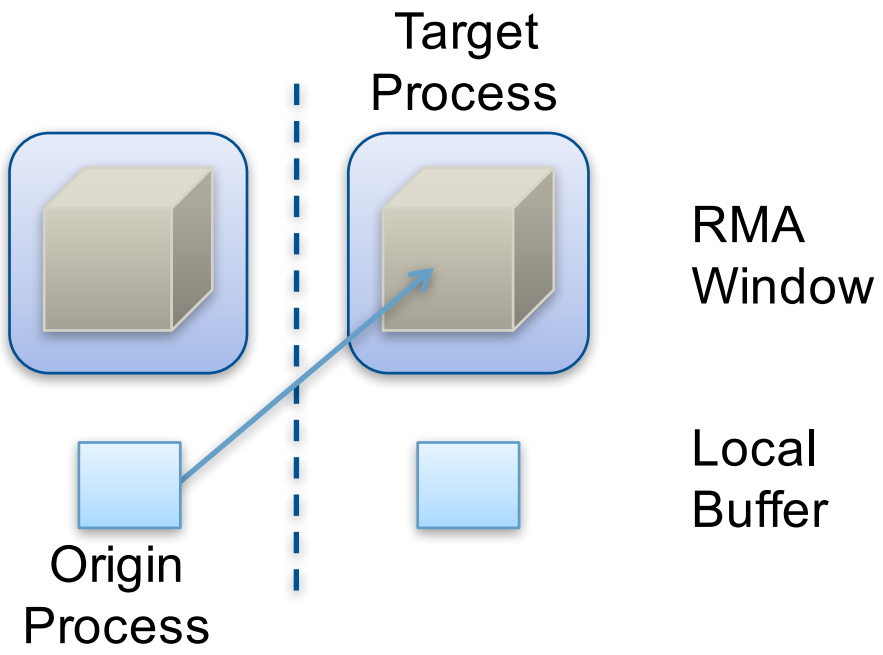
Move data from origin, to target

```
int MPI_Put(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)
```

Similar to a send operation, but
data description triples for origin **and** target

Local/origin
- Address (origin_addr)
- Count/datatype

Remote/target
- Displacement (relative to window)
- Count/datatype



Data Movement / Accumulate

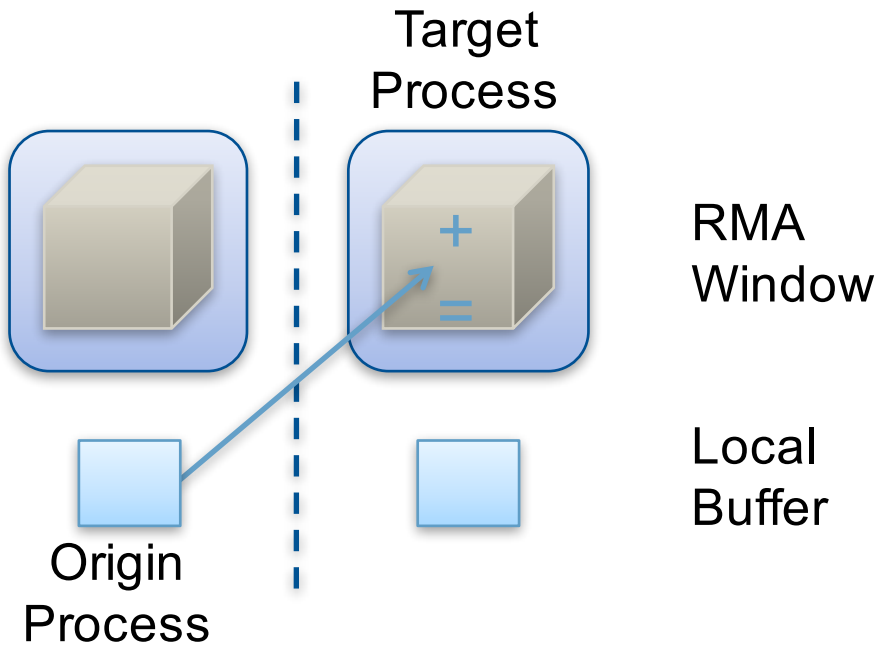
Move data from origin, to target

```
int 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)
```

Similar to `MPI_Put`

- Same arguments
- Applies an MPI_Op
  - Predefined ops only, no user-defined!
  - Special case: `MPI_REPLACE`
    - Leads to atomic update

Result ends up at target buffer



Additional RMA Operations

Get-accumulate
- Perform a get operation
- Additionally perform an accumulate locally

Compare-and-swap
- Compare the target value with an input value
- If they are the same, replace the target with some other value
- Useful for linked list creations
  - “If next pointer is NULL, do something”

Fetch-and-Op
- Special case of Get_accumulate for predefined datatypes
- Faster for the hardware to implement

Ordering of Operations in MPI RMA

For Put/Get operations, ordering does not matter
- If you do two PUTs to the same location, the result can be garbage
  – Writing full count/datatype pair (!)
- Similar to Write-after-Write races in shared memory programs with additional consistency issues

Two accumulate operations to the same location are valid
- If you want “atomic PUTs”, you can do accumulates with `MPI_REPLACE`

All accumulate operations are ordered by default
- User can tell the MPI implementation that ordering is not required
- MPI_Info object on the window

RMA Synchronization Models

RMA data visibility

- When is a process allowed to read/write from remotely accessible memory?
- How do I know when data written by process X is available for process Y to read?
- RMA synchronization models provide these capabilities

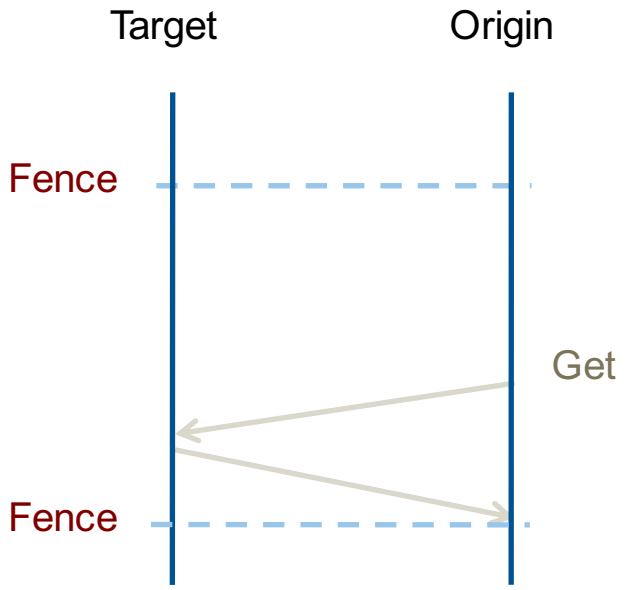
MPI RMA model allows data to be accessed only within an “epoch”

- Three types of epochs possible:
  - Fence (active target)
  - Post-start-complete-wait (active target)
  - Lock/Unlock (passive target)

Simplest model: fence synchronization

```
MPI_Win_fence(assert, win)
```

- Collective over all processes wrt. ”win”
- Start epoch with fence
  - Do exchange
- End epoch with fence

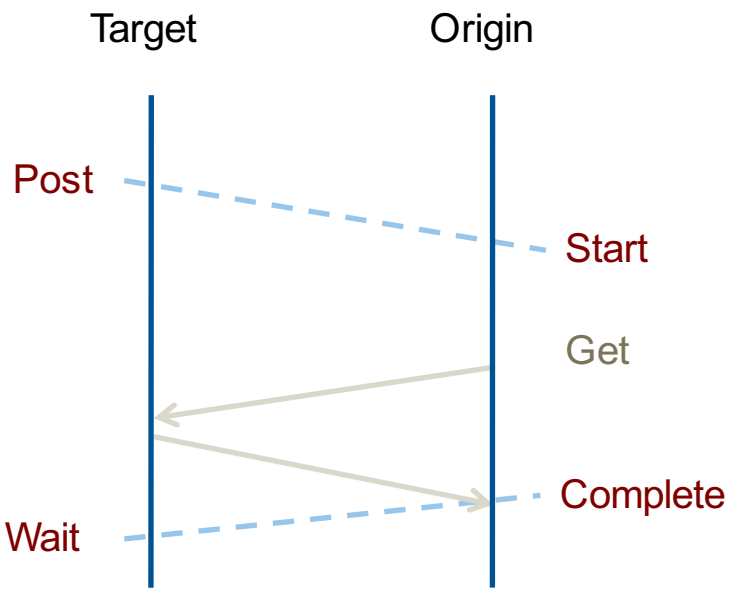


Post/Start/Complete/Wait Synchronization

Target: Exposure epoch
- Opened with `MPI_Win_post`
- Closed by `MPI_Win_wait`

Origin: Access epoch
- Opened by `MPI_Win_start`
- Closed by `MPI_Win_complete`

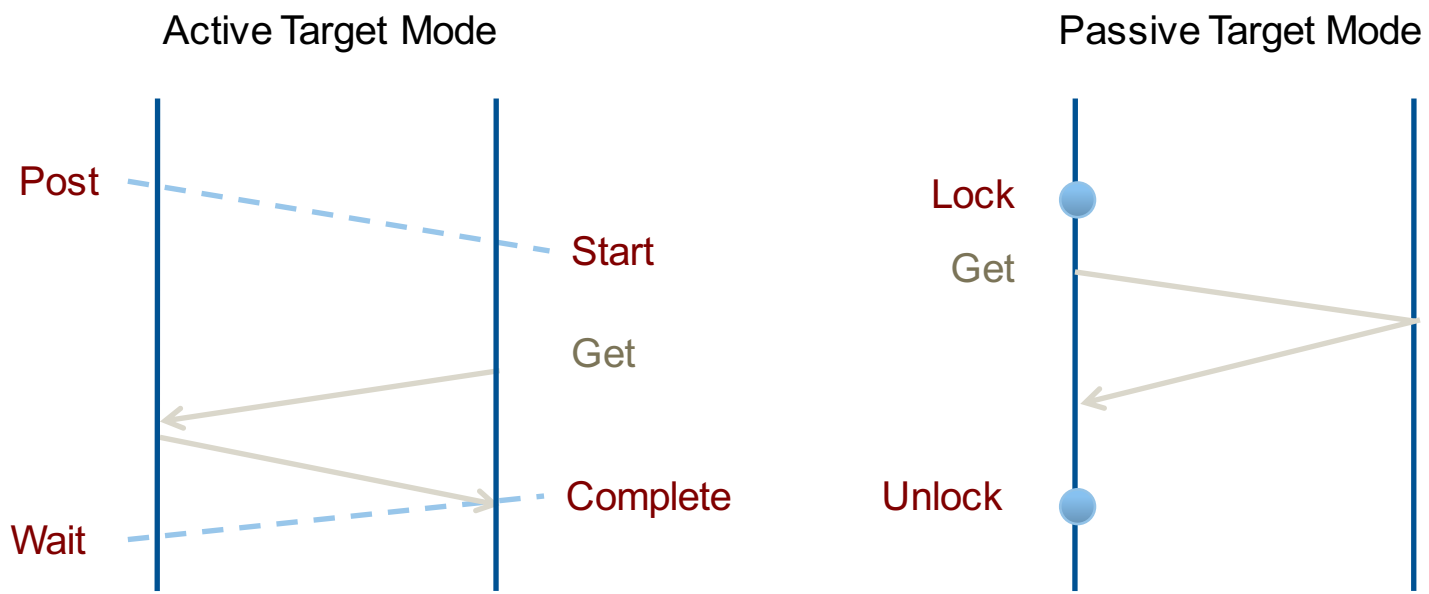
All may block
- Enforce P-S/C-W ordering
- Processes can be both origins and targets



Target may allow a smaller group of processes to access its data
- Post call takes a group of processes as argument

Origin can indicate a smaller group of processes to retrieve data from
- Start call takes a group of processes as argument

Lock/Unlock Synchronization



Passive mode: One-sided, *asynchronous* communication

- Target specified as parameter to lock call
- Target does **not** participate in communication operation
- Doesn't function like a mutex, name can be confusing
- Communication operations within epoch are all nonblocking
- Similar to release consistency

Which Communication to Use?

Two-sided communication easier to handle

- Explicit send/recv points
- Easier to reason about data and memory model
- Easy to use if send/recv points are known (e.g., SPMD codes)

RMA performance advantages from low protocol overheads

- Two-sided: Matching, queueing, buffering, unexpected receives, etc…
- Especially if supported from underlying interconnect (e.g. InfiniBand)
  - If not, RMA can be really slow

Passive mode: *asynchronous* one-sided communication

- Data characteristics:
  - Big data analysis requiring memory aggregation
  - Asynchronous data exchange
  - Data-dependent access pattern
- Computation characteristics:
  - Adaptive methods (e.g. AMR, MADNESS)
  - Asynchronous dynamic load balancing

Shared Memory in MPI

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

Create an MPI window that is also accessible with local load/store operations
- Collective operation
- Every process specifies a “minimum” size
- Function returns base pointer

Use cases
- Large read-only tables
- Often used in Equations of State (EoS)

Getting close to shared memory programming
- Load/store access
- Needs manual synchronization (outside of MPI)
- BUT: only part of address space is shared
- In most cases, still separate processes and address spaces

Combining Shared Memory and Distributed Memory Programming

Nodes in distributed memory systems are shared memory nodes

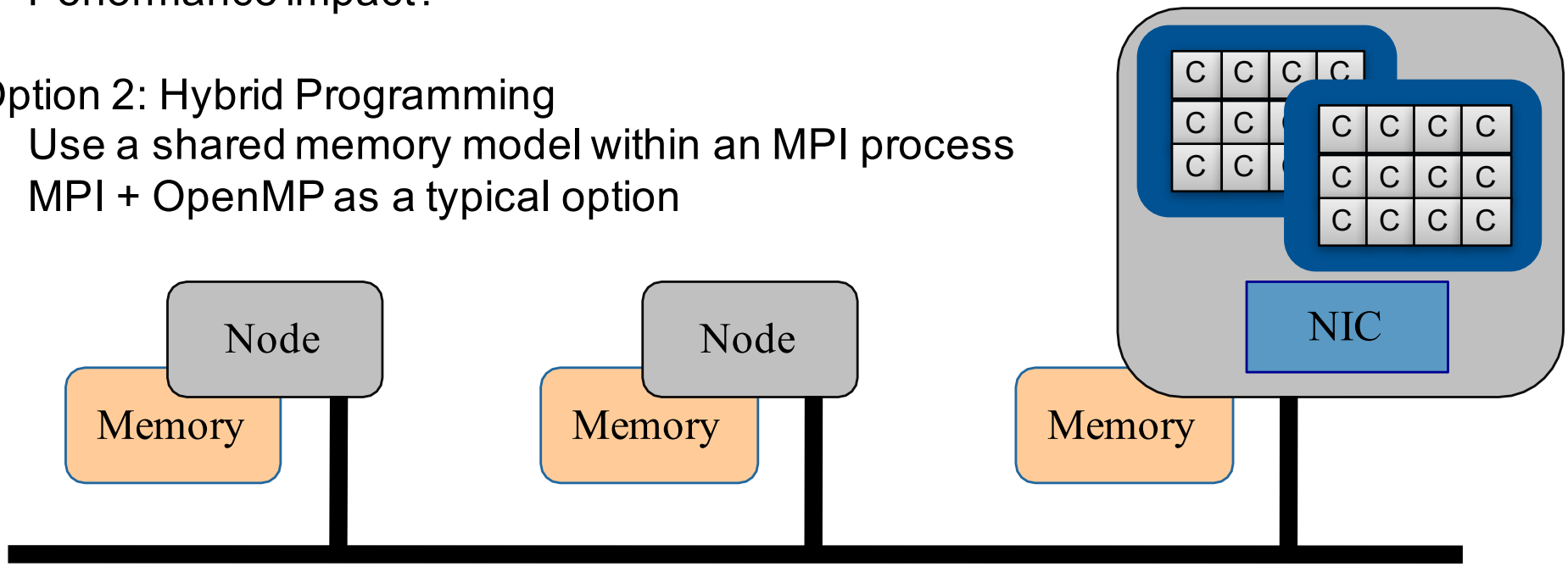
- We need a messaging library like MPI for cross-node communication
- But what on the node?

Option 1: MPI only

- Advantage: One single model, portability
- Disadvantage: no full use of sharing capabilities
- Performance impact?

Option 2: Hybrid Programming

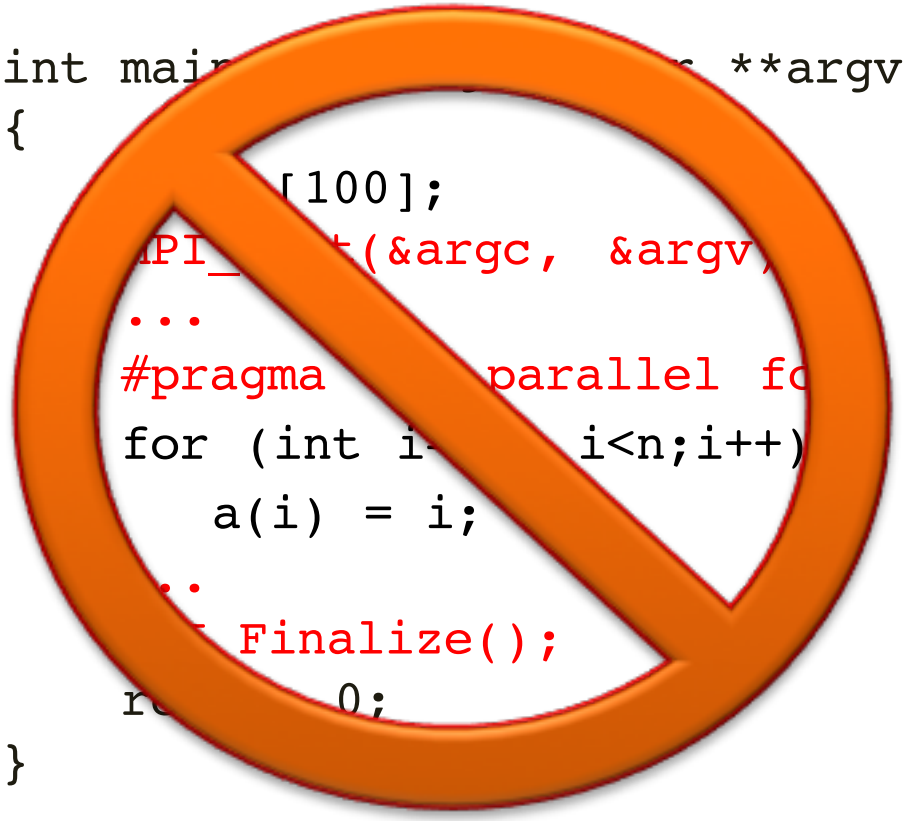
- Use a shared memory model within an MPI process
- MPI + OpenMP as a typical option



Threading and MPI

Base for any MPI program is a set of sequential programs
- Typically one per MPI process
- Each calls `MPI_Init`

What happens if we want to use a threaded program as base?



MPI 1.x was written without threads in mind
- `MPI_Init` assumes single thread
- MPI does not need to be thread safe

Accesses from several threads can cause problems during execution
- Shared data structures
- Coordinate access to NIC
- Callback coordination

MPI's Four Levels of Thread Safety

New MPI initialization routine

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

Application states needed level of thread support
MPI returns which one it supports

`MPI_THREAD_SINGLE`

- Only one thread exists in the application

`MPI_THREAD_FUNNELED`

- Multithreaded, but only the main thread makes MPI calls

`MPI_THREAD_SERIALIZED`

- Multithreaded, but only one thread at a time makes MPI calls

`MPI_THREAD_MULTIPLE`

- Multithreaded and any thread can make MPI calls at any time

Consequences of using `MPI_THREAD_MULTIPLE`

Each call to MPI completes on its own

- Effect is as if they would have been called sequentially in some order
- Blocking calls will only block the calling thread

```
T1: MPI_Send( to: 1 )
T2: MPI_Recv( from: 1 )
```

```
T1: MPI_Send( to: 0 )
T2: MPI_Recv( from: 0 )
```

User is responsible for making sure racing calls are avoided

- Example: access to an already freed object
- Collective operations must be ordered correctly among threads

```
T1: MPI_Bcast( comm )
T2: MPI_Barrier( comm )
```

```
T1: MPI_Bcast( comm )
T2: MPI_Barrier( comm )
```

Possible performance impact, e.g., caused by locking in MPI

Thread Safe Probing

`MPI_Probe/MPI_Iprobe` inspects unexpected message queue

- If message is found, this is indicated to application
- Application can then start the receive

```
T1: MPI_Probe( from: 0 )        T2: MPI_Probe( from: 0 )
    MPI_Recv ( from: 0 )            MPI_Recv ( from: 0 )
```

Problem in multithreaded programs

- Two threads call MPI_Probe and both return
- Subsequent receive may get the wrong message

Solution: matching probes and receive

```
int MPI_Mprobe/Improbe(…, MPI_Message *message, …)
```

Matching receive

```
int MPI_Mrecv(void* buf, int count, MPI_Datatype datatype,
              MPI_Message *message, MPI_Status *status)
```

Matching Receive

```
int MPI_Mprobe(int  source, int tag, MPI_Comm comm,
          MPI_Message *message, MPI_Status *status)
int MPI_Improbe(int  source, int tag, MPI_Comm comm,
          int *flag, MPI_Message *message, MPI_Status *status)
```

Wait for/Check if matching message

- Returns handle to matched message

```
int MPI_Mrecv(void* buf, int count, MPI_Datatype datatype,
          MPI_Message *message, MPI_Status *status)
int MPI_Imrecv(void* buf, int count, MPI_Datatype datatype,
          MPI_Message *message, MPI_Request *request)
```

Receive message identified by Mprobe

- Handle of message passed into receive

Thread-Levels When Using OpenMP

Must use `MPI_Init_thread` to initialize MPI with right thread level

`MPI_THREAD_SINGLE`
- There is no OpenMP multithreading in the program

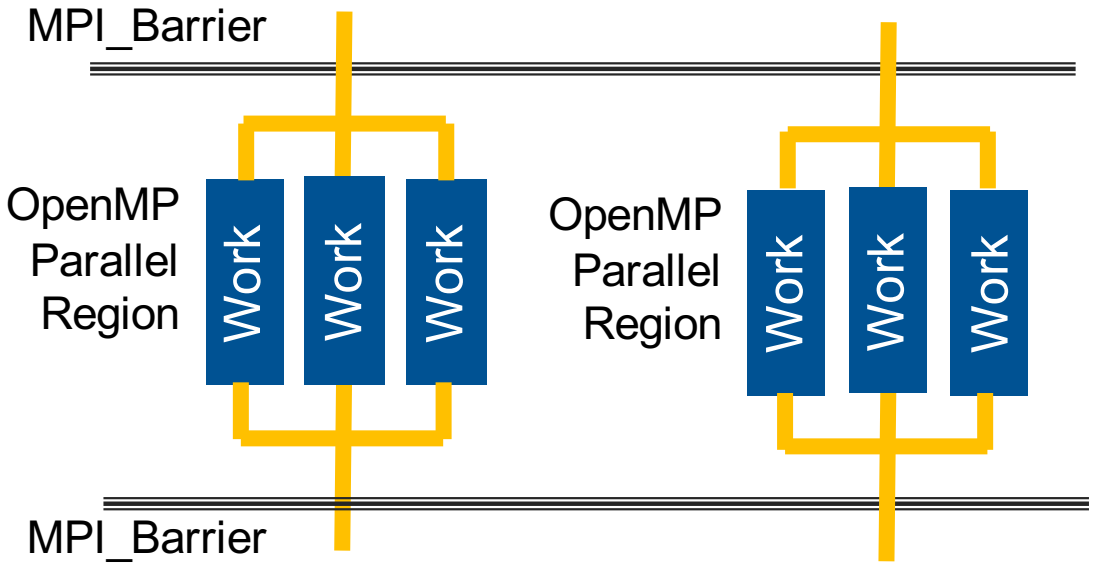
`MPI_THREAD_FUNNELED`
- Must ensure that all of the MPI calls are made by the master thread
  - Outside parallel regions
  - During master region
- Most common scenario

`MPI_THREAD_SERIALIZED`
- Must ensure that only thread calls MPI at one time
  - Critical or single regions
  - Through added synchronization

`MPI_THREAD_MULTIPLE`
- Any thread may make an MPI call at any time
- Also compatible with OpenMP tasking

Hybrid Programming MPI+OpenMP

OpenMP regions between MPI communication



Easy to understand

Fits many programs

Matches SPMD mode

Requires thread funneled only

Danger: Amdahl's Law in the non-OpenMP regions

Need to split data structures (for MPI), then share data parts (for OpenMP threads)

- Hierarchical data decomposition

When to Use Hybrid?

Depends on application
- Sharing requirements and granularity
- Suitable for hierarchical decomposition?
- Shared data structures?
- Impact on portability?

Option
- MPI only with one MPI process per HW thread
- One MPI process per node, one OpenMP thread per core on node
- Several MPI processes per node, rest of the concurrency in OpenMP

One MPI process per socket often fits well
- Large degree of parallelism in OpenMP
- Naturally avoids NUMA problems
- Need to pin threads to that socket

Summary

MPI Datatypes
- Enable flexible description of data elements used for communication
- Basic types matching the base language
- Derived types built from base types

MPI One-sided communication
- Create a window that is shared across processes
- Remote memory get, put and accumulate operations
- Synchronization based on Epochs

Ability, from within MPI, to create locally shared memory regions

Hybrid programming
- Need to negotiate with MPI on thread support
- Existence of thread has impact on MPI, e.g., probe operations
- Hybrid MPI+OpenMP programming
- Guideline: one MPI process per socket, rest by threading