MPI advanced features

Fabio Affinito – SCAI department
f.affinito@cineca.it

Summary

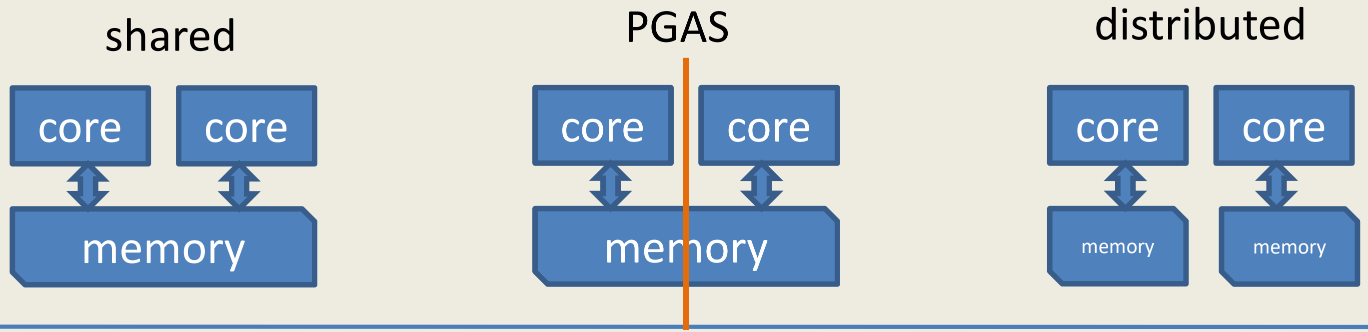
1. Introduction
2. MPI derived datatypes
3. Non blocking collective communications
4. Topologies and neighbourhood collectives
5. One-sided communication
6. MPI and MPI+X

Introduction

HPC machines are more and more oriented towards nodes with large number of cores (with a constant amount of memory).

Different programming models exist according to the model of memory consistency:

- distributed memory
- UMA/NUMA shared memory
- PGAS and similars



MPI

What is MPI?

- not a programming model
- not a language

But a standard relying on the distributed memory programming model (or paradigm).

Note that all models can be mapped to any architecture more or less efficiently (i.e. that depend on the execution model).

MPI3 is a standard, whose efficiency is highly dependent on the execution model (and on the implementation of the standard)

MPI cornerstones

- Communication concepts
  - Point to point communications
  - Collective communications
  - One sided communications
  - Collective I/O operations
- Declarative concepts
  - Groups and communicators
  - Derived datatypes
  - Process topologies
- Tool support
  - Linking and runtime

MPI history

MPI is an open standard library interface for message passing.
The standard is ratified by the MPI Forum.

1.0 – 1.1 – 1.2 – 1.3	1994-2008	Basic message-passing concepts
2.0 – 2.1	2008	Added one sided and I/O concepts
2.2	2009	Merging and smaller fixes
3.0	2012	Several new features

Best practises

Valid not only for MPI…

1. Identify a scalable algorithm
2. Check if there are existing libraries that can help my work.
    1. Computation libraries (MKL, PetSC, ScaLAPACK..)
    2. Utility libraries (LibXC, HDF5)
    3. Etc.
3. Increase modularity of your applications
    1. Writing (parallel) libraries has many benefits

1. Introduction
2. MPI derived datatypes
3. Non blocking collective communications
4. Topologies and neighbourhood collectives
5. One-sided communication
6. MPI and MPI+X

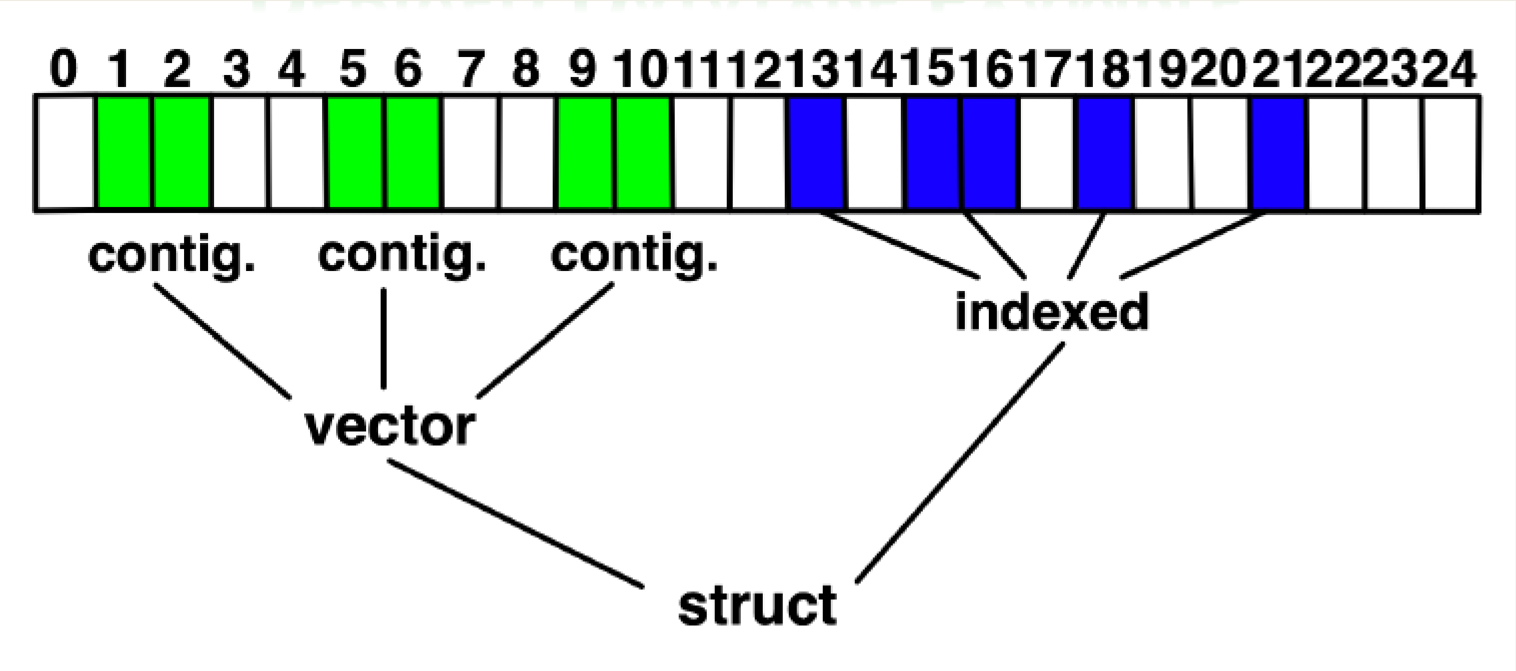
Derived datatypes

- Derived datatypes are in the MPI standard since v.1.0.
- Some extensions have been made in MPI-2.x and MPI-3.0
- Why is still an «advanced concept»?
  - not really popular (bad reputation…)
  - It enables many elegant optimization (zero copy)
  - It is a very elegant concept (and makes your code more clean!)

Terminology

- Type size
  - Size of the DDT signature (in bytes)
  - Important for matching
- Lower bound
  - Where does the DDT start
  - It can contain some «holes»
- Extent
  - Complete size of the DDT
  - Allow to «interleave» DDT. It can be dangerous

DDT overview

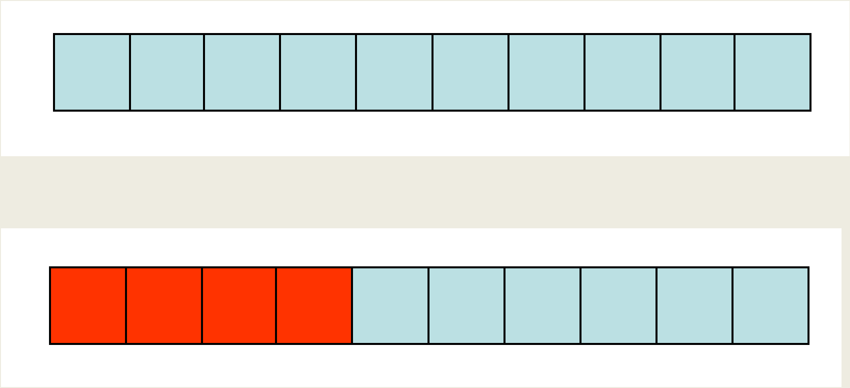


Basic datatypes

MPI has several pre-defined datatypes, used in elementary operations such as MPI_Send and MPI_Recv.

```
int [10]

MPI_Send(x,4,MPI_INT, …);
```

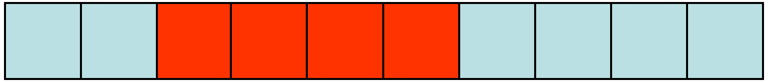


Basic datatypes

It is possible to define a different element of a buffer.

But you are limited to send contiguous data

```
MPI_Send(&x[2],4,MPI_INT, …);
```



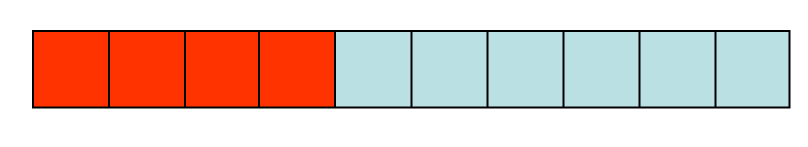
New derived datatypes permit to overcome (not only) this limitation

Contiguous datatype

It is the most simple datatype.
Apparently trivial, but it can be a building block for other DDT.

```
MPI Datatype mynewtype;
MPI_Type_contiguous(count=4,oldtype=MPI_INT, newtype=&mynewtype);
MPI_Type_commit(&mynewtype)
```

```
MPI_Send(x,1,mynewtype …);
```



Array layout in memory

1	2	3	4	5	6	7	8

C: row-major

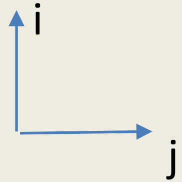
1	**2**
3	4
5	6
7	8

Fortran: col-major

1	**5**
2	6
3	7
4	8

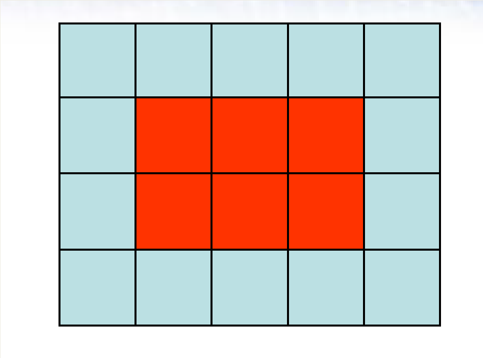
Process grid

- Use the C convention for the process coordinates, even in Fortran
  - Processes always ordered as for C arrays (and array indexes start with 0)
- This is what is returned by MPI for cartesian topologies
- Example: process rank layout on a 4x4 process grid
  - Rank 6 is at position (i=1,j=2) for C and Fortran

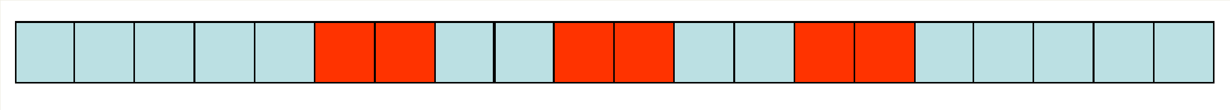


0	**1**	**2**	**3**
4	5	6	7
8	9	10	11
12	13	14	15

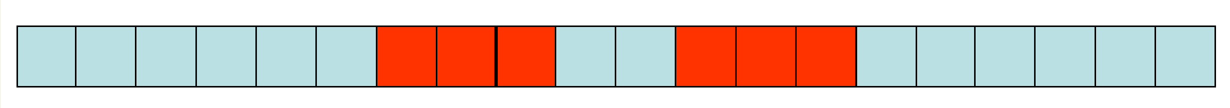
Sub-array layout in memory



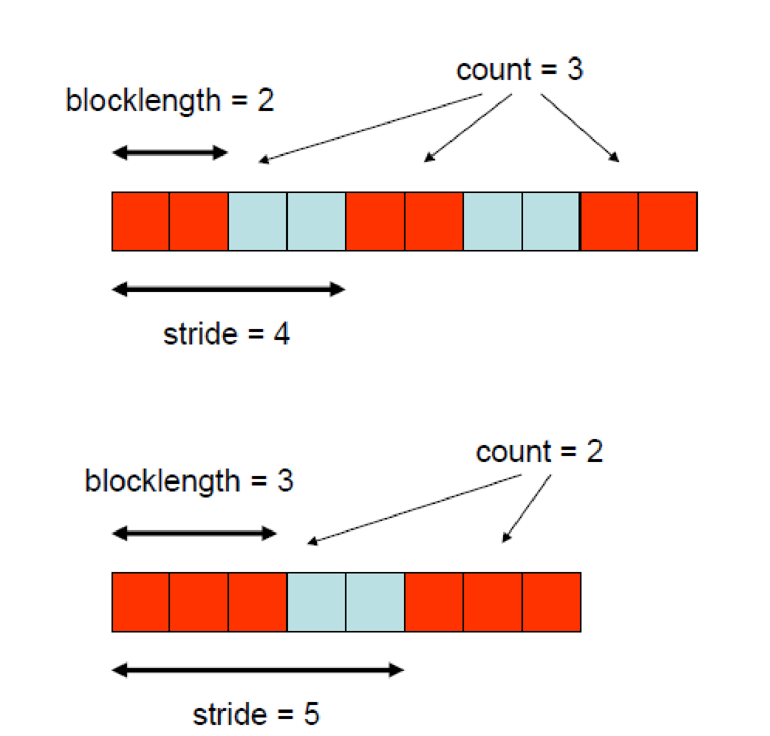
Fortran



C



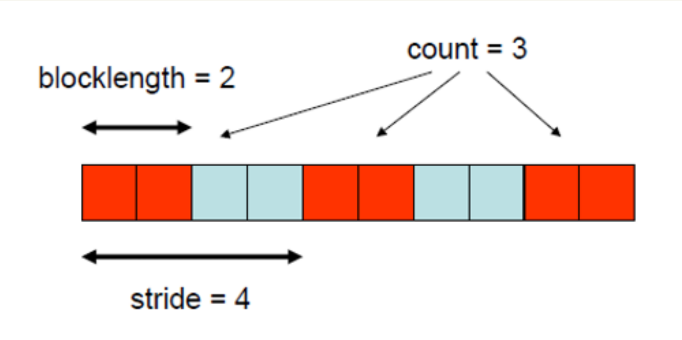
Vector datatypes



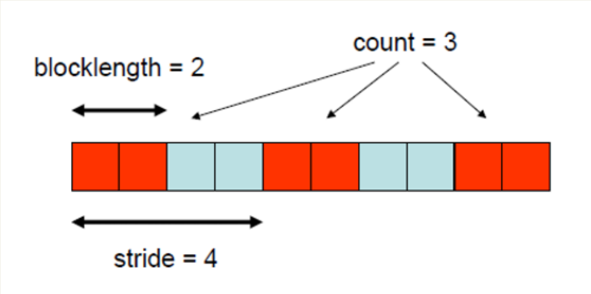
Vector datatype: definition

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

```
MPI_Datatype vector3x2;
MPI_Type_vector(3, 2, 4,MPI_FLOAT, &vector3x2)
MPI_Type_commit(&vector3x2)
```

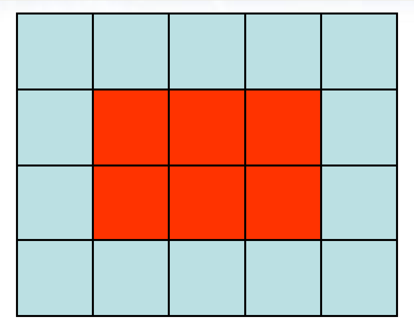


Vectors and subarrays

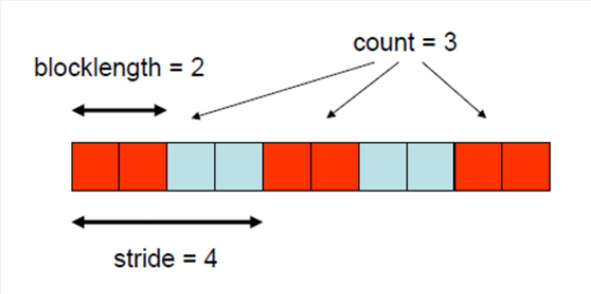


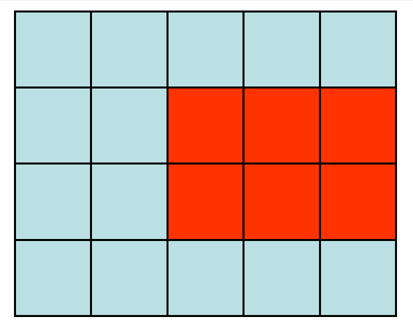

```
MPI_Send(&x[1][1],1,vector3x2, …);

MPI_SEND(x(2)(2), 1,vector3x2, …)
```



Vectors and subarrays



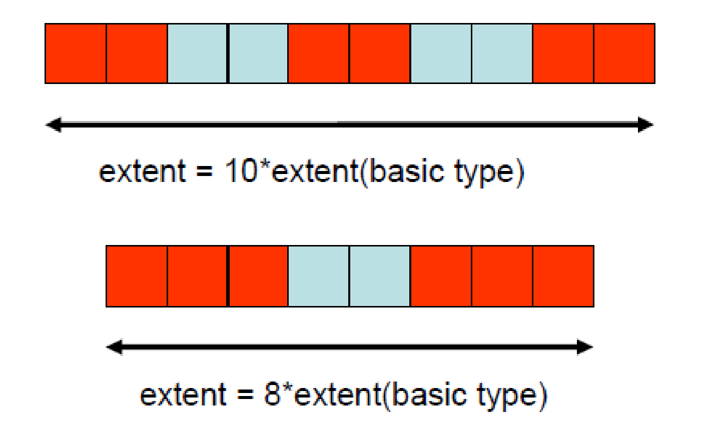


```
MPI_Send(&x[1][2],1,vector3x2, …);

MPI_SEND(x(2)(3), 1,vector3x2, …)
```

Datatype extent

- Datatypes are read from memory separated by their extent
- For basic datatypes, extent is the size of the object
- For vector datatypes, extent is the distance from first to last data



Subarray datatype

A single call defines multidimensional subsections:

- useful when working with 3D arrays

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

Subarray datatypes: C

```
#define NDIMS 2
MPI_Datatype subarray3x2;
int array_of_sizes[NDIMS], array_of_subsizes[NDIMS],
arrays_of_starts[NDIMS];

array_of_sizes[0] = 5;
array_of_sizes[1] = 4;
array_of_subsizes[0] = 3;
array_of_subsizes[1] = 2;
array_of_starts[0] = 2;
array_of_starts[1] = 1;

order = MPI_ORDER_C;

MPI_type_create_subarray(NDIMS, array_of_sizes, array_of_subsizes, array_of_starts,
    order, MPI_FLOAT, &subarray3x2);
MPI_TYPE_COMMIT(&subarray3x2);
```

Subarray datatypes: Fortran

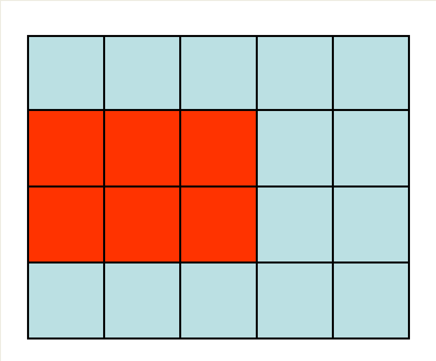
```
integer, parameter :: ndims = 2
integer subarray3x2
integer, dimension(ndims) :: array_of_sizes, array_of_subsizes,
arrays_of_starts

array_of_sizes(1) = 5
array_of_sizes(2) = 4
array_of_subsizes(1) = 3
array_of_subsizes(2) = 2
array_of_starts(1) = 2
array_of_starts(2) = 1
order = MPI_ORDER_FORTRAN

call MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts,
    order, MPI_REAL, subarray3x2, ierr)
call MPI_TYPE_COMMIT(subarray3x2, ierr)
```

Using subarray datatype

```
MPI_Send(&x[0][0], 1, subarray3x2, ...);
MPI_SEND(x , 1, subarray3x2, ...)
MPI_SEND(x(1,1) , 1, subarray3x2, ...)
```



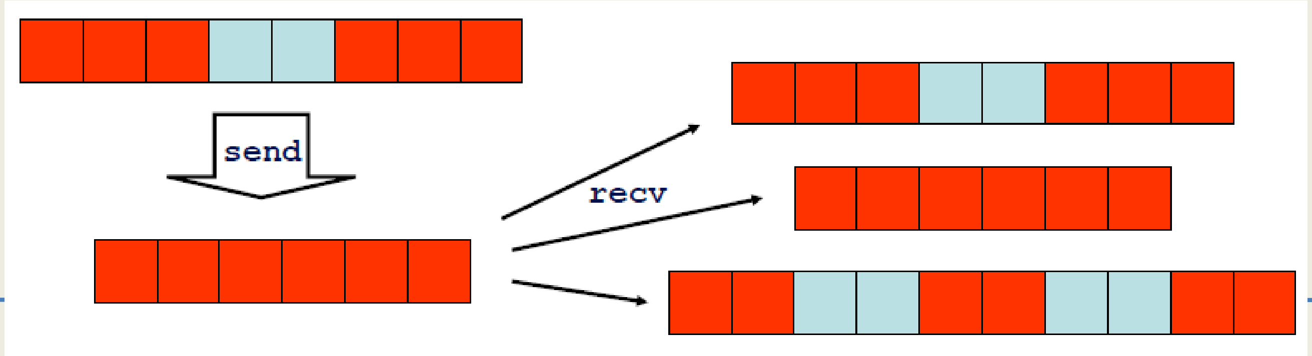
Matching messages

Messages are packed during the transmission between processes and empty entries are removed.

A datatype consists of:

- Type signature: an ordered list of the basic datatypes
- Type map: locations of each basic datatype

For a receive to match a send only signatures need to match.



Final considerations

1. Creation of datatypes requires an overhead
   1. No need to redefine datatypes every time
   2. Array sizes unlikely to change: define datatypes once for all
2. Beware of creating too many datatypes
   1. They consume memory
   2. Use MPI_Type_free whenever possible
3. Don’t:

```
do loop=1,1000000
        do stuff
        define type
        use type
        free type
end do
```

1. Introduction
2. MPI derived datatypes
3. Non blocking collective communications
4. Topologies and neighbourhood collectives
5. One-sided communication
6. MPI and MPI+X

Nonblocking and collective communication

- Nonblocking communication
  - Deadlock avoidance
  - Overlap communication and computation
- Collective communication
  - Optimized routines
- Nonblocking collective communications
  - Combine both advantages
  - System noise/imbalance resiliency
  - Semantic advantages

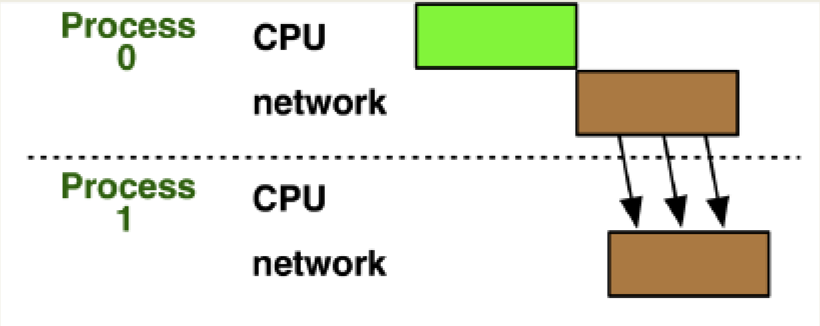
Nonblocking communication

Very simple semantics:

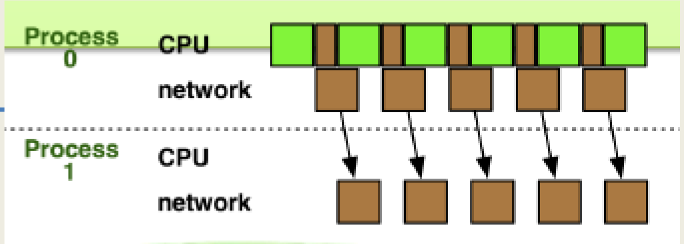
- Function returns no matter what
- No progress guarantee!
- Available:
  - Non blocking tests (test,testany,testsome…)
  - Blocking wait (wait,waitany, waitall…)

Motivation: pipelining

```
if(r == 0) {
   for(int i=0; i<size; ++i) {
   arr[i] = compute(arr, size);
   }
   MPI_Send(arr, size, MPI_DOUBLE, 1, 99, comm);
} else {
   MPI_Recv(arr, size, MPI_DOUBLE, 0, 99, comm, &stat);
}
```



Motivation: pipelining




```
if(r == 0) {
    MPI_Request req=MPI_REQUEST_NULL;
    for(int b=0; b<nblocks; ++b) {
          if(b) {
                if(req != MPI_REQUEST_NULL) MPI_Wait(&req, &stat);
                MPI_Isend(&arr[(b-1)*bs], bs, MPI_DOUBLE, 1, 99, comm, &req);
          }
          for(int i=b*bs; i<(b+1)*bs; ++i) arr[i] = compute(arr, size);
    }
    MPI_Send(&arr[(nblocks-1)*bs], bs, MPI_DOUBLE, 1, 99, comm);
} else {
    for(int b=0; b<nblocks; ++b)
    MPI_Recv(&arr[b*bs], bs, MPI_DOUBLE, 0, 99, comm, &stat);
}
```

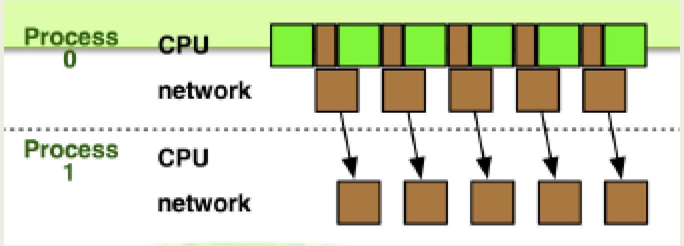
Pipeline performance model

No pipeline:

T = Tcomp(s) + Tcomm(s) + Tstartc(s)

Pipeline:

T = nblocks * [max(Tcomp(bs) , Tcomm(bs)) + Tstartc(bs)]



Collective communication

Three types:
- Synchronization (Barrier)
- Data Movement (Scatter, Gather, Alltoall, Allgather)
- Reductions (Reduce, Allreduce, (Ex)Scan, Red_scat)

Common semantics:
- no tags (communicators can serve as such)
- Blocking semantics (return when complete)
- Not necessarily synchronizing (only barrier and all*)

Nonblocking collective communication

Nonblocking variants of all collectives

- MPI_Ibcast(<bcast args>, MPI_Request *req);

Semantics:

- Function returns no matter what
- No guaranteed progress (quality of implementation)
- Usual completion calls (wait, test) + mixing
- Out-of order completion

Restrictions:

- No tags, in-order matching
- Send and vector buffers may not be touched during operation
- MPI_Cancel not supported
- No matching with blocking collectives

Nonblocking collective communication

Semantic advantages:

- Enable asynchronous progression (and manual)
- Software pipelinling
- Decouple data transfer and synchronization
- Noise resiliency!
- Allow overlapping communicators (see also neighborhood collectives)
- Multiple outstanding operations at any time
- Enables pipelining window

Noisy systems

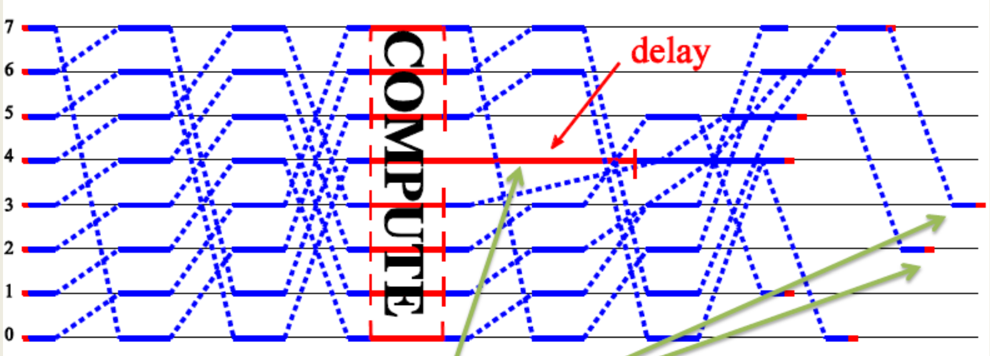
When dealing with a very large number of cores it can happen that one of them is delayed for a wide number of reasons (daemons, interrupts, steal cycles).

This delay is likely to propagate…

Characterizing the Influence of System Noise on Large-Scale Applications by Simulation

Torsten Hoefler
University of Illinois at Urbana-Champaign
Urbana IL 61801, USA
htor@illinois.edu

Timo Schneider and Andrew Lumsdaine
Indiana University
Bloomington IN 47405, USA
{timoschn,lums}@cs.indiana.edu



- Process 4 is delayed
  - Noise propagates "*wildly*" (of course deterministic)

Non-blocking barrier?

What can that be good for?

Semantics:

- MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
- Synchronization **may** happen asynchronously
- MPI_Test/Wait() – synchronization happens **if** necessary

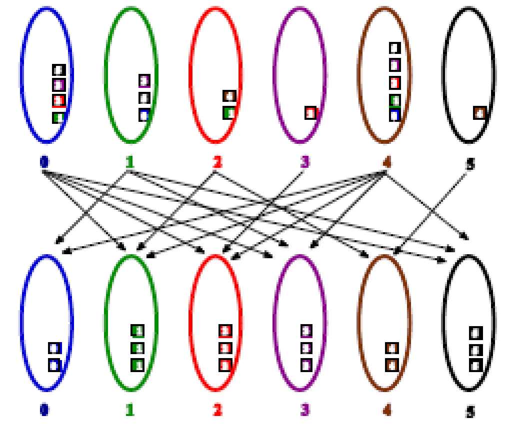
Uses:

- Overlap barrier latency (small benefit)
- Use the split semantics! Processes **notify** non-collectively but **synchronize** collectively!

Semantics example: DSDE

Dynamic Sparse Data Exchange

- Dynamic: comm. pattern varies across iterations
- Sparse: number of neighbors is limited (O(logP))
- Data exchange: only senders know neighbors



DSDE

Main Problem: metadata

Determine who wants to send how much data to me (I must post receive and reserve memory)

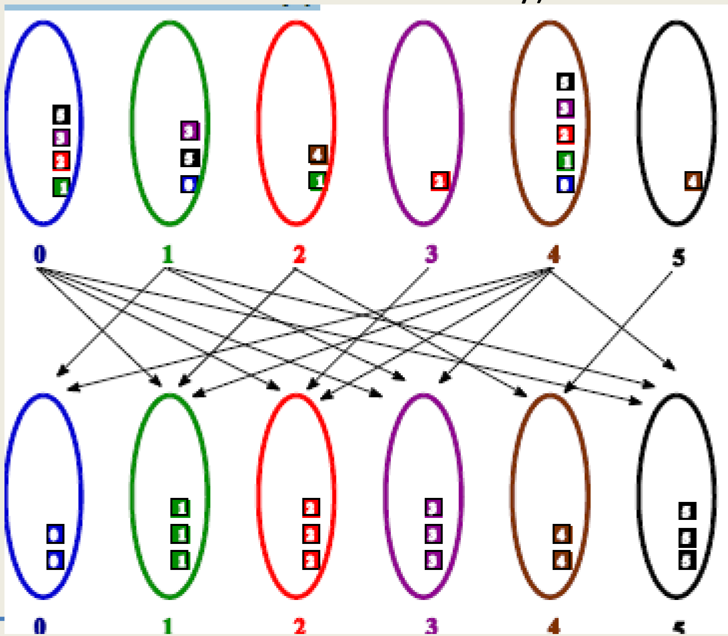
OR:

Use MPI semantics:

- Unknown sender (MPI_ANY_SOURCE)
- Unknown message size (MPI_PROBE)

Reduces problem to counting the number of neighbours

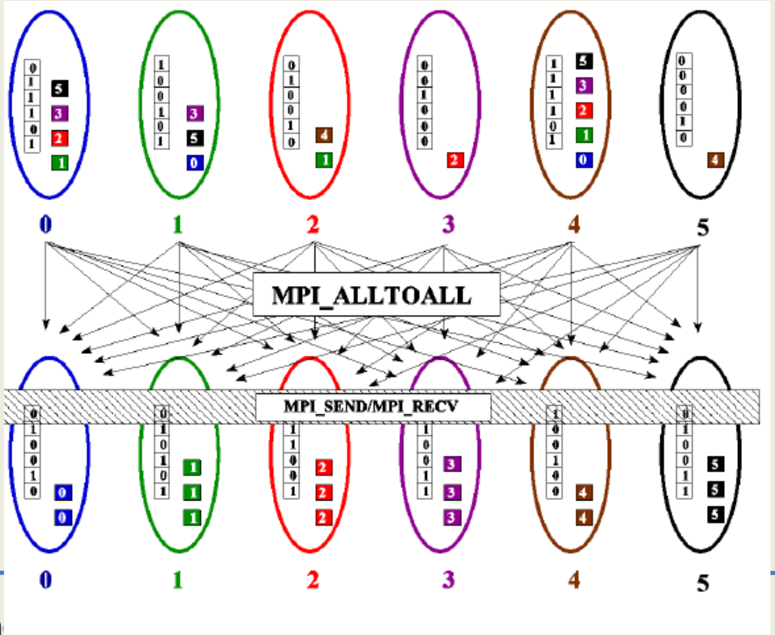
Allow faster implementation!



DSDE using Alltoall (PEX)

Based on Personalized Exchange (PEX)

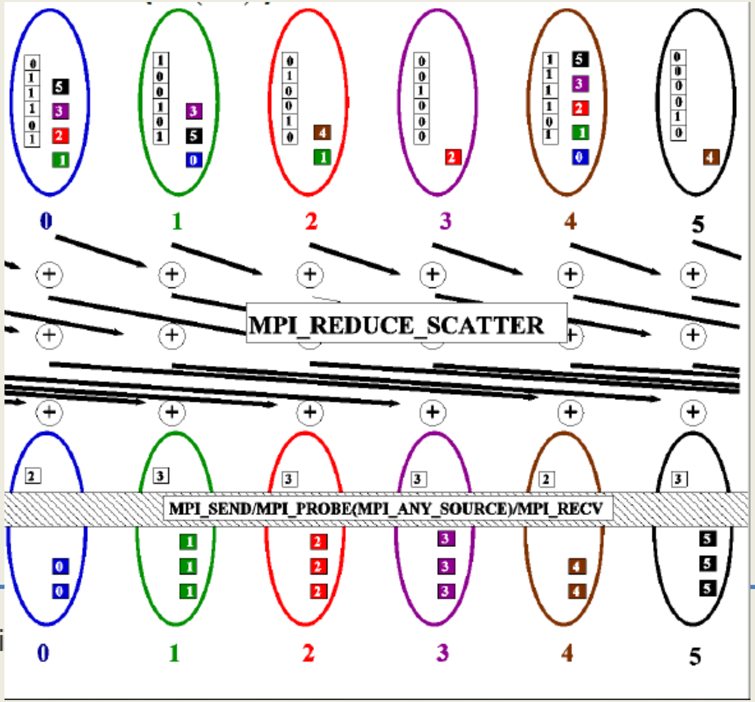
- Processes exchange metadata (sizes) about neighborhoods with all-to-all
- Processes post receives afterwards
- Most intuitive but least performance and scalability!



DSDE using Reduce/Scatter

Based on Personalized Census (PCX)

Processes exchange metadata (counts) about neighborhoods with reduce_scatter
- Receivers checks with wildcard MPI_IPROBE and receives messages
- Better than PEX but non-deterministic!



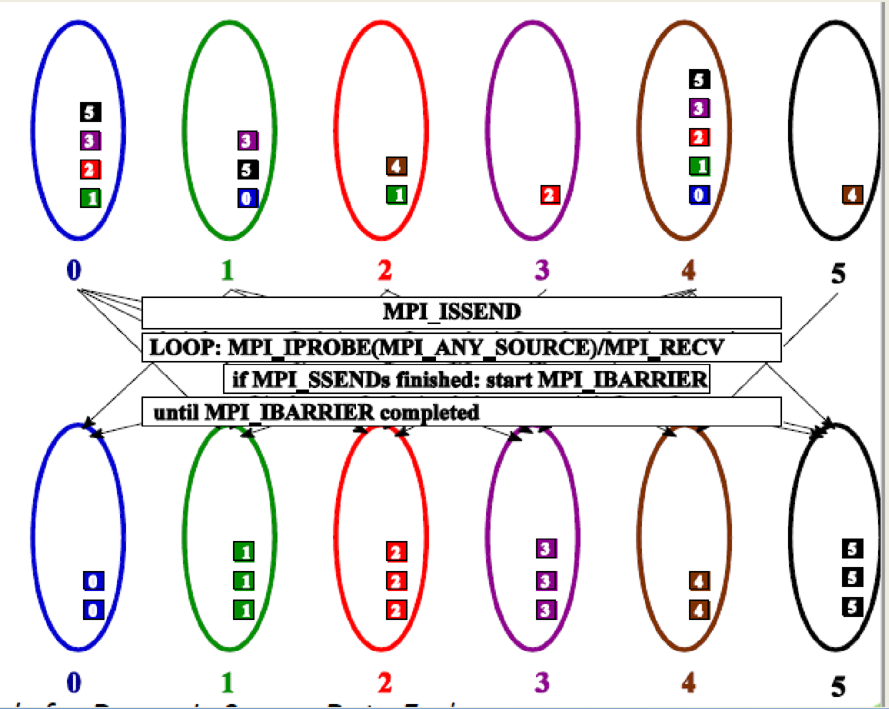
DSDE using nonblocking barrier

Combines metadata with actual transmission

- Point-to-point synchronization
- Continue receiving until barrier completes
- Processes start coll. synch. (barrier) when p2p phase ended

barrier = distributed marker!

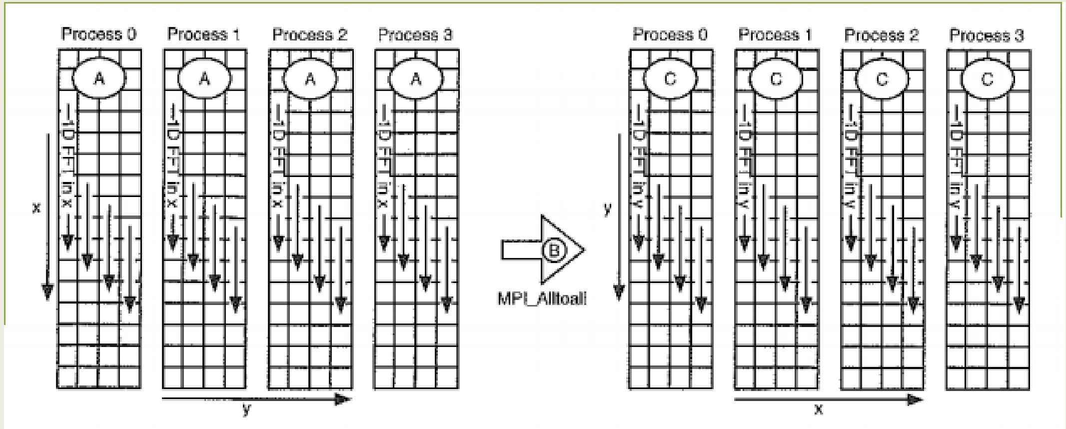
- Better than PEX, PCX, RSX!



```
MPI_Request reqs[m];
for (i=0, i<m,++i){
    MPI_Issend(sbuf[i],size[i],type,dest[i],tag,comm,&reqs[i]);
    MPI_Request barrier_req;
    int barrier_done=0, barrier_active=0;
    while(!barrier_done){
          MPI_Iprobe(MPI_ANY_SOURCE,tag,comm,&flag,&stat);
          if(flag){
                    //allocate buffer and receive msg
          }
          if(!barrier_active){
                    int flag;
                    MPI_Testall(m,reqs,&flag,MPI_STATUS_IGNORE);
                    if(flag){
                      MPI_Ibarrier(comm,&barrier_request);
                      barrier_active=1;
                    }
          }else{
                    MPI_Test(&barrier_request,&barrier_done,MPI_STATUS_IGNORE);
    }
}
```

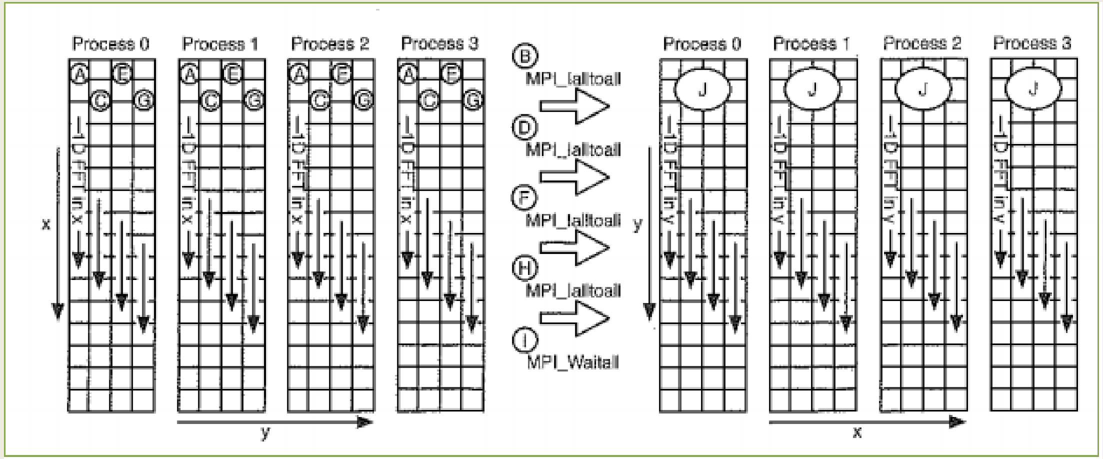
Nonblocking and 2D-FFT

- 2D-FFT can be distributed among different processes each of them executes a 1D-FFT
- After the 1D-FFT along, for example, the x-direction is completed, an MPI_Alltoall is performed.
- Now, each process can execute a 1D-FFT along the y-direction and a final MPI_Alltoall permits to obtain the complete 2D-FFT



Nonblocking and 2D-FFT

- The performance can be improved if we start the transposition and, without waiting for the completion, we start working on the FFT along the second direction



1. Introduction
2. MPI derived datatypes
3. Non blocking collective communications
4. Topologies and neighbourhood collectives
5. One-sided communication
6. MPI and MPI+X

Topologies

MPI topologies have been introduced in the v.2.0 of the MPI standard as a tool for a better usage of processes distributions.

MPI 3.0 introduces new neighbourhood collective operations:

- MPI_Neighbor_allgather[v]
- MPI_Neighbor_alltoall[v|w]

Recap of MPI topologies

Regular n-dimensional grid or torus topology:

- MPI_CART_CREATE

General graph topology

- MPI_GRAPH_CREATE

  all processes specify all edges in the graph (not scalable)

General graph topology (distributed version)

- MPI_DIST_GRAPH_CREATE_ADJACENT

  all processes specify their incoming and outgoing neighbours

- MPI_DIST_GRAPH_CREATE

  all processes can specify any edge in the graph

Recap of MPI topologies

Testing the topology type associated with a communicator
– MPI_TOPO_TEST
Finding the neighbours for a process
– MPI_CART_SHIFT
Find out how many neighbours there are:
– MPI_GRAPH_NEIGHBORS_COUNT
Get the ranks of all neighbours:
– MPI_GRAPH_NEIGHBORS
Find out how many neighbours there are:
– MPI_DIST_GRAPH_NEIGHBORS_COUNT
Get the ranks of all neighbours:
– MPI_DIST_GRAPH_NEIGHBORS

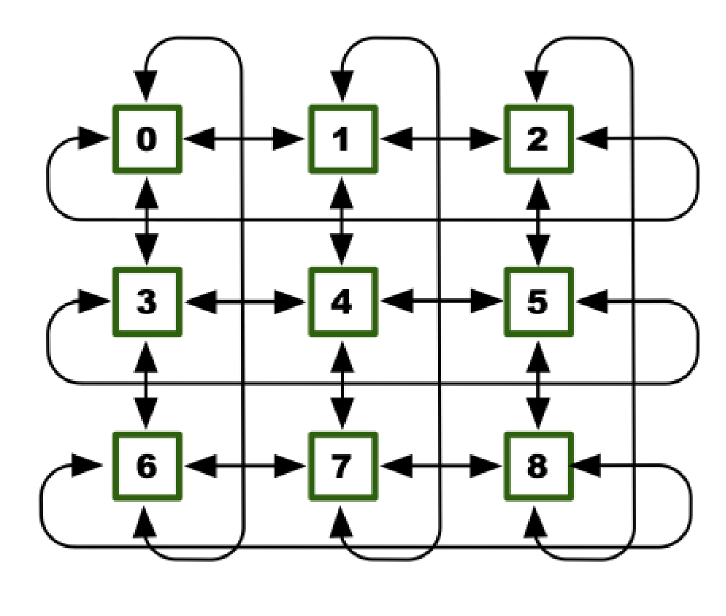
Neighbourhood collective operations

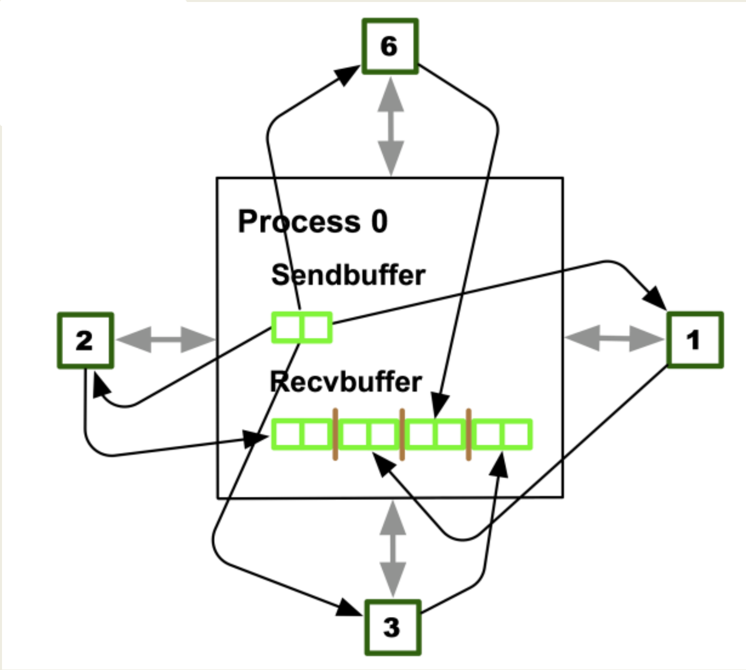
MPI_[N|In]eighbor_allgather[v]
– Send one piece of data to all neighbours
– Gather one piece of data from each neighbour

MPI_[N|In]eighbor_alltoall[v|w]
– Send different data to each neighbour
– Receive different data from each neighbour

Use-case: regular or irregular domain decomposition codes
– Where the decomposition is static or changes infrequently
– Because creating a topology communicator takes time

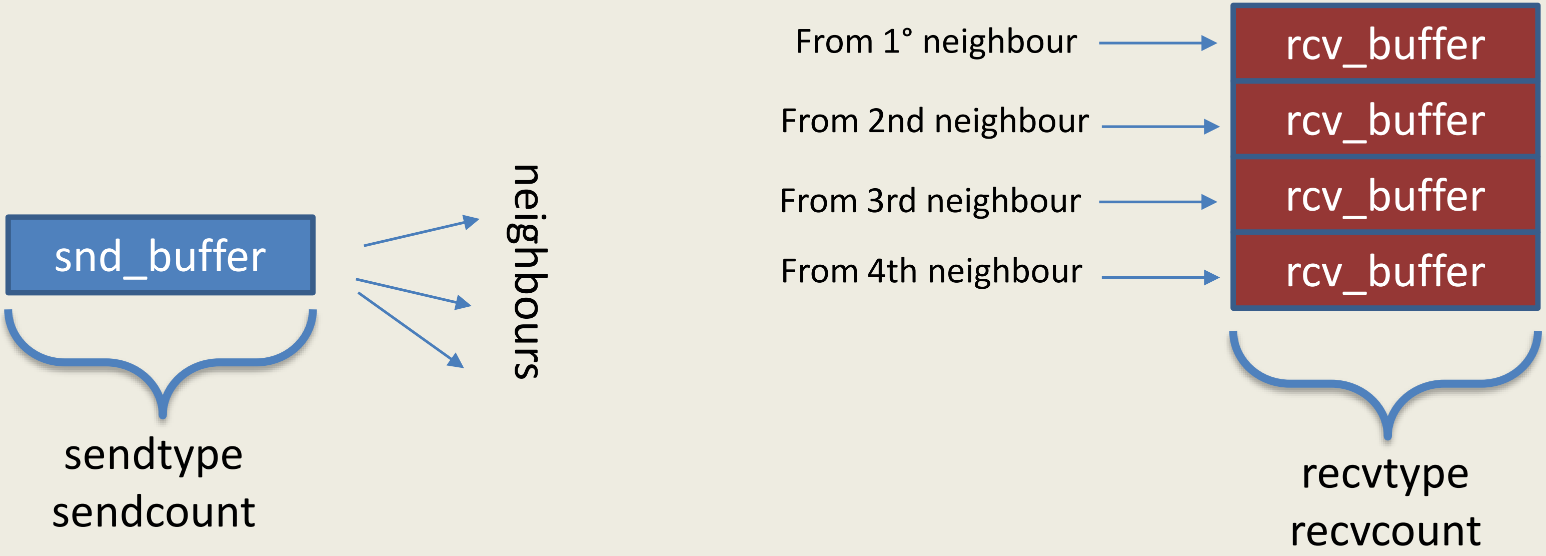
Buffer ordering





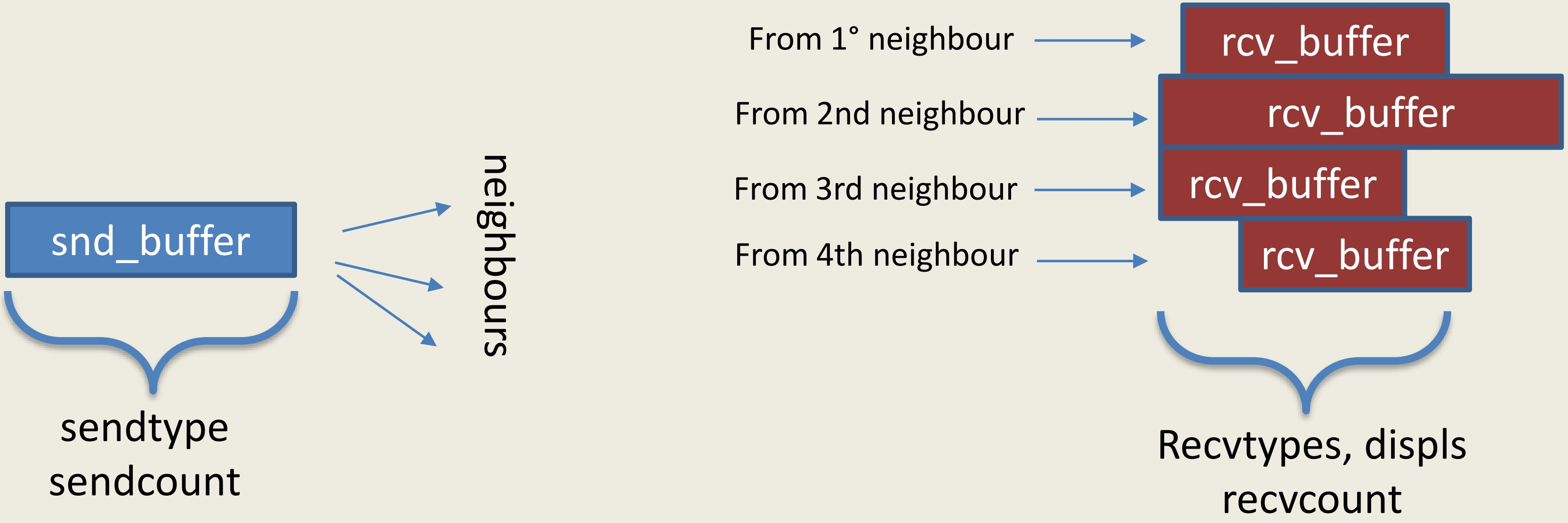
MPI_Neighbor_allgather

- Send the same message to all neighbours
- Contiguous chunks in in receive buffer from each incoming neighbour



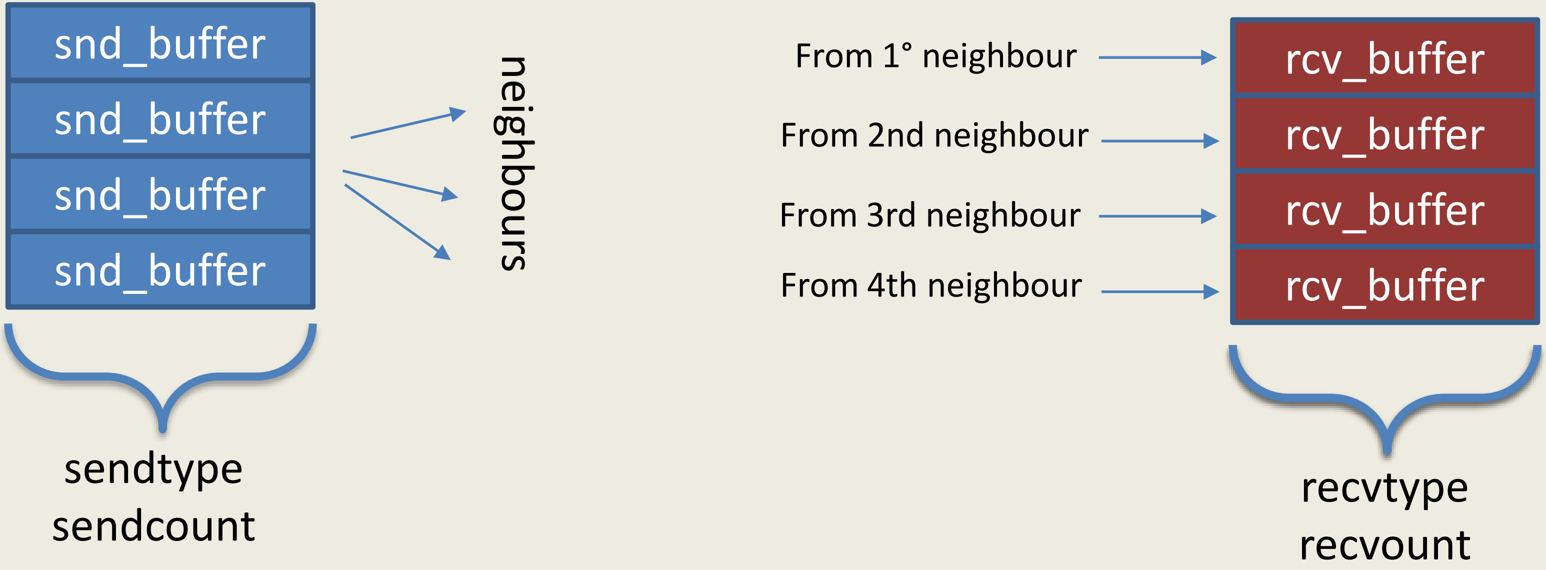
MPI_Neighbor_allgatherv

- Send the same message to all neighbours
- Non-contiguous variable-sized chunks in in receive buffer from each incoming neighbour



MPI_Neighbor_alltoall

- Send the same message to all neighbours
- Contiguous chunks in in receive buffer from each incoming neighbour



Neighborhoods collectives

Neighborhood collectives add communication functions to process topologies

- Collective optimization potential!

Allgather

- One item to all neighbors

Alltoall

- Personalized item to each neighbor

High optimization potential (similar to collective operations)

- Interface encourages use of topology mapping!

1. Introduction
2. MPI derived datatypes
3. Non blocking collective communications
4. Topologies and neighbourhood collectives
5. One-sided communication
6. MPI and MPI+X

Outline

MPI RMA basic concepts

- Why RMA?
- Terminology
- Program flow

Getting started with RMA

- Management of windows
- Fence synchronization
- Moving data around

Why RMA?

- One sided communication functions are an interface to MPI RMA
- It can provide a performance/scalability increase for your codes
  - Programmability reasons
  - Hardware (interconnect) reasons
  - But it is not a silver bullet!

RMA terminology

Origin is the process that performs the call.

Target is the process whose memory is accessed.

- All remote access are performed on windows of memory
- All accesses calls are non-blocking and issued inside an epoch

RMA program flow

1. Collectively initialize a window
2. Start a RMA epoch (synchronization)
3. Perform communication calls (put,get,etc.)
4. Close the RMA epoch (synchronization)
5. Collectively free the window

Window creation

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

- Window creation is a collective operation.
- Each process may specify different locations, sizes, displacement units and info arguments
- The same region of memory may appear in multiple windows that have been defined for a process. But concurrent communications to overlapping windows are disallowed.
- Performance may be improved by ensuring that the windows align with boundaries such as word or cache-line boundaries.

Window management

```
int MPI_Win_get_attr(MPI_Win win, int win_keyval,
void *attribute_val, int *flag)
```

This function permits to retrieve the window attributes.

- *min_keyval* options are: MPI_WIN_BASE, MPI_WIN_SIZE, MPI_WIN_DISP_UNIT, MPI_WIN_CREATE_FLAVOR, MPI_WIN_MODEL
- *Attribute_val* if the attribute is available and in this case (flag is true), otherwise flag will be false

Window management

After all RMA calls have been completed (i.e. after the epoch is closed), you should free the window.

```
int MPI_Win_free(MPI_Win *win)
```

Fences

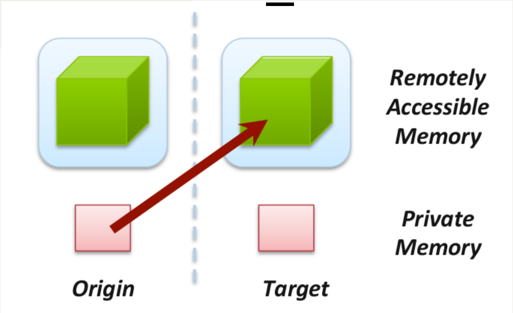
Synchronization calls are required to start and stop an epoch.

Fences are the simplest way of doing this.

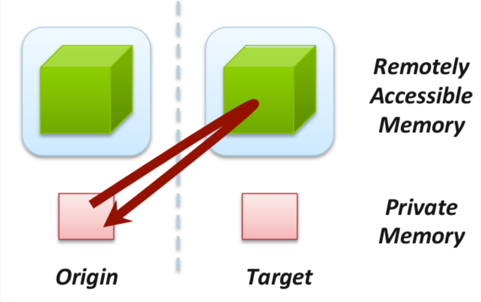
```
int MPI_Win_fence(int assert, MPI_Win win)
```

Data movement with RMA

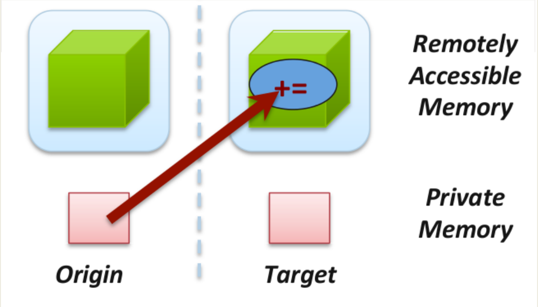
MPI_Put



MPI_Get



MPI_Accumulate



RMA communication

Get data from target’s memory

```
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)
```

Put data into target’s memory

```
int 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)
```

Accumulate data in target’s memory with some other data

```
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)
```

RMA communication

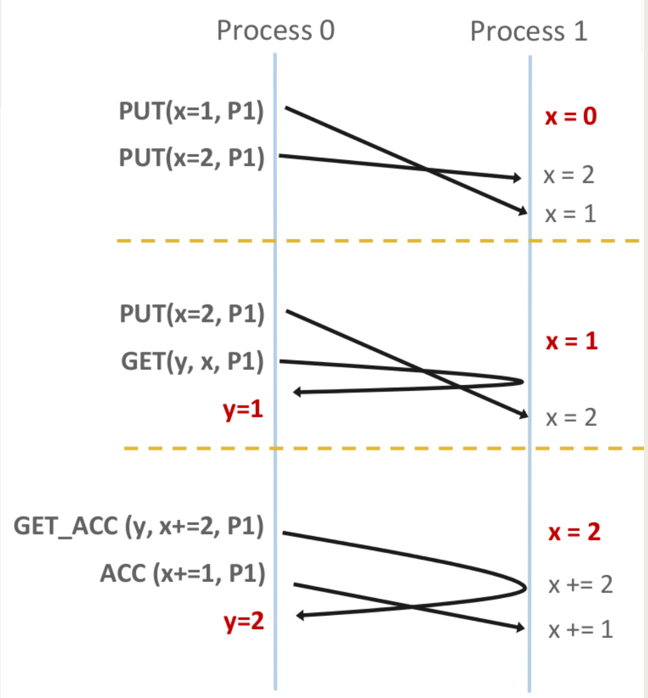
Similarly to non-blocking P2P one must wait for synchronisation (i.e. end of epoch) until accessing retrieved data (*get*) or overwriting written data (*put/accumulate*)

target_disp is in bytes (multiplied by window displacement unit), **origin_count** and **target_count** are in elements of data type

Undefined operations:

- Local stores/reads with a remote PUT in an epoch
- Several origin processes performing concurrent PUT to the same target location
- Single origin process performing multiple PUTs to the same target location in a single epoch
- Accumulate supports the MPI_Reduce operations, but NOT user defined operations. Also supports MPI_REPLACE which is effectively the same as a put.

RMA communication



No guaranteed ordering for put/get

Results for concurrent put/accumulate are undefined

For concurrent accumulate operations to the same location ordering is guaranteed

Fence example

```
MPI_Win win;
if (rank == 0) {
   MPI_Win_create(buf, sizeof(int)*20, 1, MPI_INFO_NULL, comm, &win);
} else {
   MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, comm, &win);
}
MPI_Win_fence(MPI_MODE_NOPRECEDE,win);

if (rank != 0) {
   MPI_Get(mybuf, 20 , MPI_INT, 0, 0, 20, MPI_INT, win);
}
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
MPI_Win_free(&win)
```

Only rank 0 attach an area of memory to the window

No RMA calls before the fence

Non-zero ranks get the 20 integer from rank zero

No RMA calls after the fence

Synchronization modes

Active target

- Both processes are explicitly involved in the data movement. Only one process issues the data transfer call but all processes issue the synchronisation.

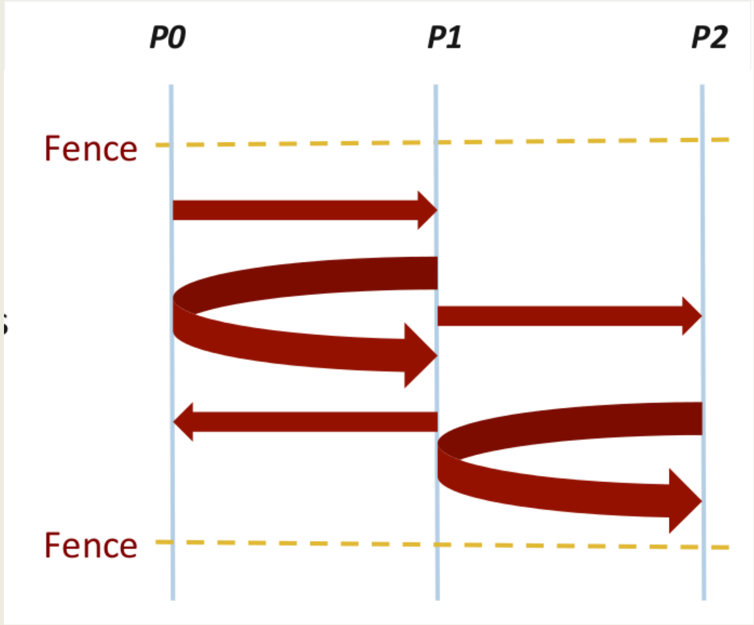
Passive target

- Only the origin process is involved in the data movement, there are no calls made on the target process. For instance two origin processes might communicate by accessing the same location in a target window, and the target process (which does not participate) might be distinct from the origin processes.

Fence is an example of active target as each process issues the fence calls

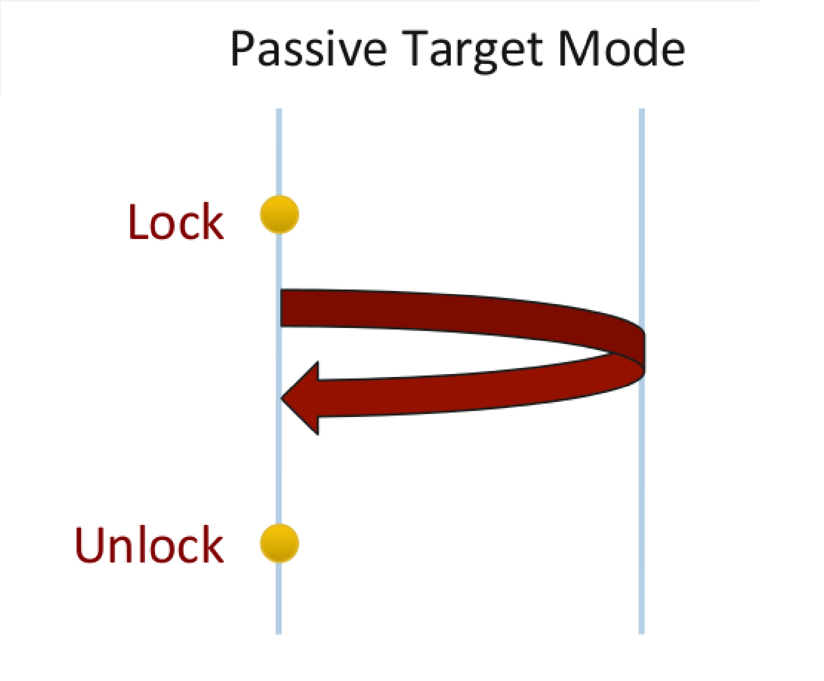
Fence: active target synchronization

- MPI_WIN_FENCE starts and ends access and exposure epoch of all processes in the window
- Collective synchronization model
- All operations complete at the second fence synchronization



Lock/unlock: passive target synchronization

- One sided asynchronous communication
- Target does not participate in communication operation
- Shared memory-like model



Epoch types

Access epoch definition

- RMA communication calls (get, put etc) can only be issued inside an access epoch. This is started with an RMA synchronisation call on the origin and completes with the next synchronisation call.
- i.e. it is used to access the remote memory of another process.

Exposure epoch

- Used in active target communication, this is required to expose memory on the target so it can be accessed by other processes' RMA operations.

Fences abstract the programmer from this as they will complete/start both access and exposure epochs automatically as required

Post-start-complete-wait

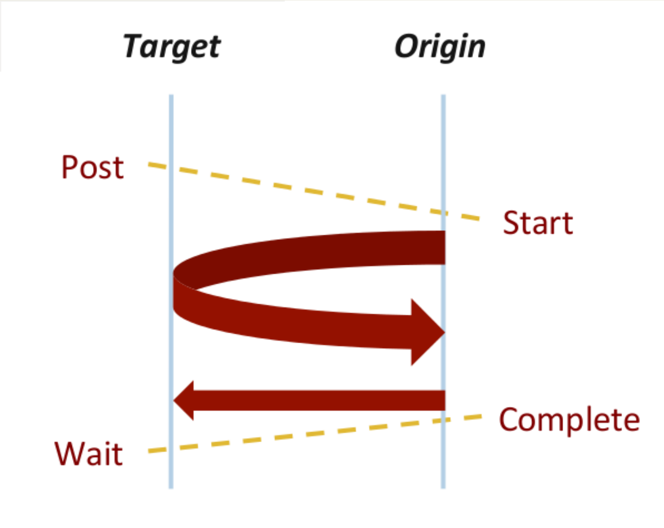
The programmer can handle explicitly different kinds of epochs

Post creates an exposure epoch

Wait ends an exposure epoch

Start creates an access epoch

Complete ends an access epoch

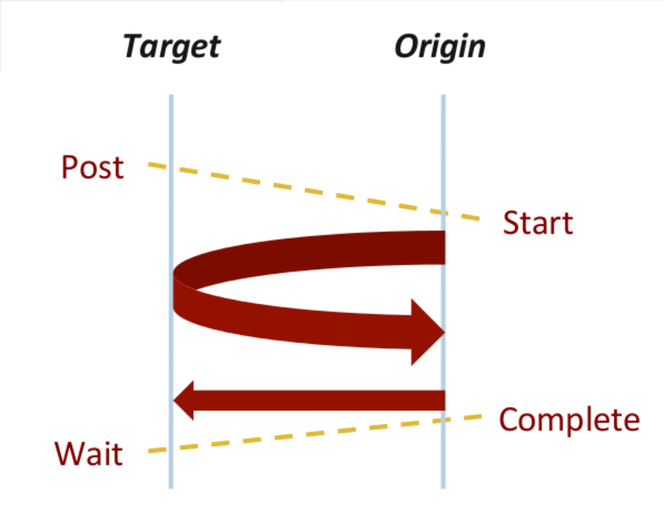


Post-start-complete-wait

Post will not block, **start** may or may not block

Wait will block until all matching complete calls and guarantees target RMA completion

Complete will block until RMA communications of that epoch have completed and guarantees origin RMA completion



PSCW example

```
int ranks[]={0,1,2};
if (rank == 0) {
     MPI_Win_create(buf, sizeof(int)*3, sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &win);
} else {
     MPI_Win_create(NULL, 0, sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD,
&win);
}
if (rank == 0) {
     MPI_Group_incl(comm_group, 2, ranks+1, &group);
     MPI_Win_post(group, 0, win);
     MPI_Win_wait(win);
} else {
     MPI_Group_incl(comm_group, 1, ranks, &group);
     MPI_Win_start(group, 0, win);
     MPI_Put(buf, 1, MPI_INT, 0, rank, 1, MPI_INT, win);
     MPI_Win_complete(win);
}
```

RMA memory model

Public and private window copies

- Public memory region is addressable by other processes (i.e. exposed main memory)
- Private memory (i.e. transparent caches or communication buffers) which is only locally visible but elements from public memory might be stored.
- Coherent if updates to main memory are automatically reflected in private copy consistently
- Non-coherent if updates need to be explicitly synchronised

RMA memory model

MPI therefore has two models

- Unified if public and private copies are identical – used if possible, realistic on cache coherent machines. *(This was added in MPI v3)*
- Separate if they are not, here there is only one copy of a variable in process memory but also a distinct public copy for each window that contains it. The old model

RMA memory model

In the separate model a suitable synchronisation call (i.e. end of an epoch) must be issued to make these consistent. In the unified model some synchronisation calls might be omitted for performance reasons

The window attribute tells you which model it follows

Locks and unlocks

PSCW is an example of active target synchronisation as the target must still explicitly create an exposure epoch

Locks/unlocks are an example of passive synchronisation where only the origin takes part.

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
int MPI_Win_unlock(int rank, MPI_Win win)
```

Inside the epoch (i.e. between lock & unlock) then RMA communication calls as normal, these complete ***for both the origin and target*** on the corresponding unlock.

Locks and unlocks

The lock type argument to lock is either:

- **MPI_LOCK_SHARED** where multiple processes may access the target window at any one time
- **MPI_LOCK_EXCLUSIVE** where only one process may access the target window at any one time

MPI 3 also added lock_all and unlock_all variants which control access to all processes associated with a window.

There is also

- **flush** to flush outstanding RMA operations on the window to the target rank
- **sync** to synchronise public & private window copies (separate memory model)

Locks and unlocks: example

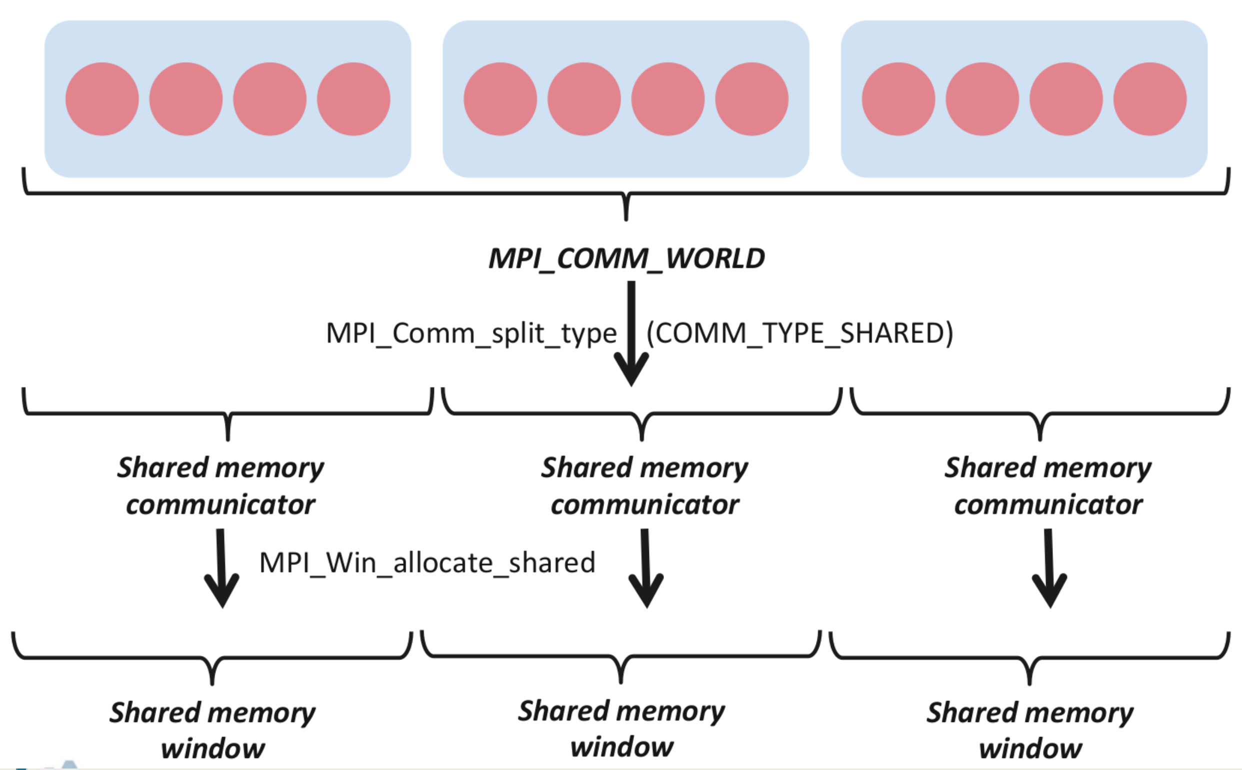
```
MPI_Win win;

if (rank == 0) {
    MPI_Win_create(NULL,0,1,MPI_INFO_NULL,MPI_COMM_WORLD,&win);
    MPI_Win_lock(MPI_LOCK_SHARED,1,0,win);
    MPI_Put(buf,1,MPI_INT,1,0,1,MPI_INT,win);
    MPI_Win_unlock(1,win);
    MPI_Win_free(&win);
} else {
    MPI_Win_create(buf,2*sizeof(int),sizeof(int), MPI_INFO_NULL,
    MPI_COMM_WORLD, &win);
    MPI_Win_free(&win);
}
```

Shared memory with MPI

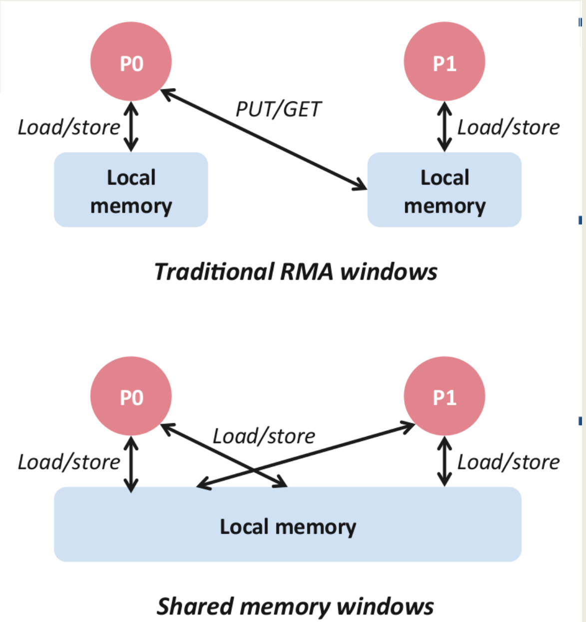
- MPI-3 permits to manage shared memory access to different processes
- It uses many of the concepts of one-sided communication
- Can be simpler to implement wrt OpenMP threads
- It can live together with other different MPI parallelization layers

MPI and shared memory



RMA windows vs shared memory windows

- Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
  - e.g. x[100]=10



Memory allocation and placement

- Shared memory allocation does not need to be uniform across processes
  - Processes can allocate a different amount of memory (even zero)
- The MPI standard does not specify where the memory would be placed (e.g. which physical memory it will be pinned to)
- The total allocated shared memory on a communicator is contiguous by default
  - Users can pass an info hint called “noncontig” that will allow the MPI implementation to align memory allocations from each process to appropriate boundaries to assist with placement

MPI Shared memory example

```
int main(int argc, char ** argv)
{
  int buf[100];
  MPI_Init(&argc, &argv);
  MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
  MPI_Win_allocate_shared(comm, ..., &win);
  MPI_Win_lockall(win);

  /* copy data to local part of shared memory */

  MPI_Win_sync(win);

  /* use shared memory */

  MPI_Win_unlock_all(win);
  MPI_Win_free(&win);
  MPI_Finalize();
  return 0;
}
```

1. Introduction
2. MPI derived datatypes
3. Non blocking collective communications
4. Topologies and neighbourhood collectives
5. One-sided communication
6. MPI and MPI+X

MPI+X

HPC systems based on multicore makes hybrid solution a must.

For heterogeneous systems with GPUs, the most natural approach is MPI+Cuda.

In most cases (homogenous systems) the most popular approach si MPI+OpenMP.

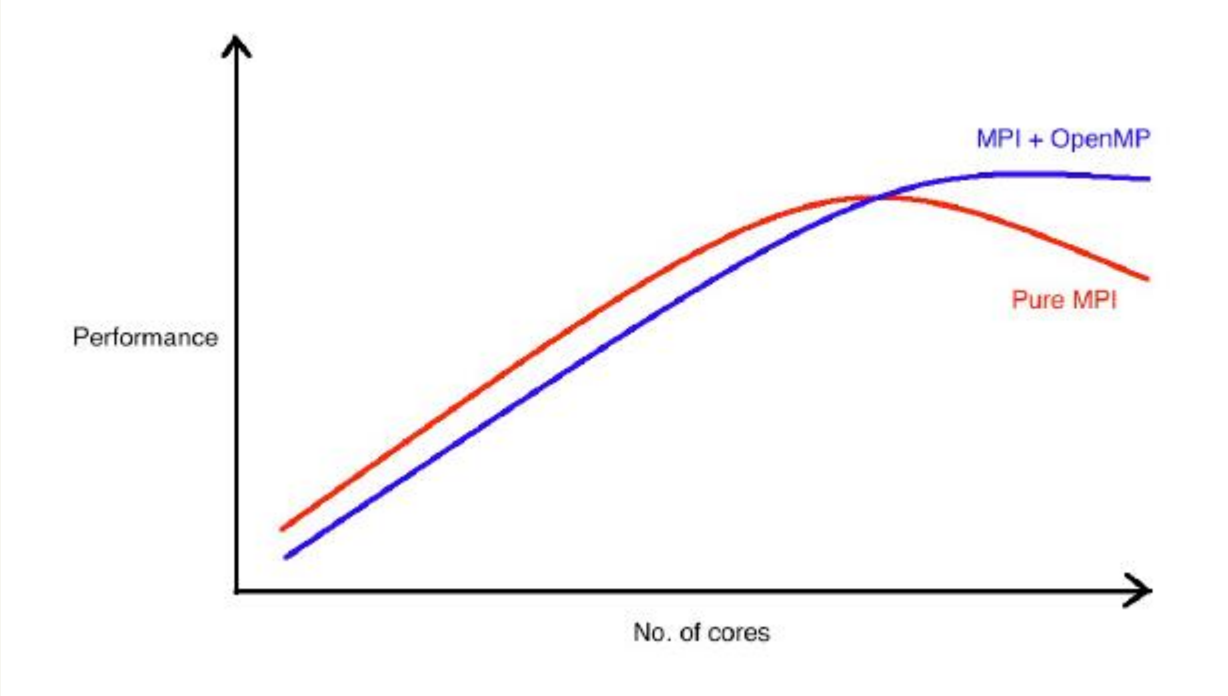
Recently, using RMA and Shared Memory capabilities of MPI, made MPI+MPI an option.

MPI+OpenMP

Using together MPI and OpenMP permits to create a hierarchy (intranode/internode) able to exploit the cores using threads.

Typically, MPI+OpenMP does not improve the scalability in the regime where MPI is scaling well, but it permits to increase the scalability when MPI reaches the saturation.

MPI+OpenMP



Maintenance costs

- Maintenance of a hybrid MPI+OpenMP code can be harder than for a pure MPI code
- When mixing MPI and OpenMP, typical problems of those approaches sum up
- In particular, in OpenMP there is the risk of race conditions and non deterministic bug. The presence of MPI on top can make the debugging much more complex.

Portability issues

- Even if both MPI and OpenMP are highly portable (in principle)
- MPI+OpenMP can have some portability issues
    - Thread safety: if the maximum level is assumed, portability will be reduced

Performance pitfalls

- Adding OpenMP may introduce additional overheads not present in the MPI code (e.g. synchronisation, false sharing, sequential sections, NUMA effects).

- Adding OpenMP introduces a tunable parameter – the number of threads per MPI process
    - optimal value depends on hardware, compiler, input data
    - hard to guess the right value without experiments

- Placement of MPI processes and their associated OpenMP threads within a node can have performance consequences.

Performance pitfalls

- The mixed implementation may require more synchronisation than a pure OpenMP version, if non-thread-safety of MPI is assumed.
- Implicit point-to-point synchronisation via messages may be replaced by (more expensive) barriers.
  - loose thread to thread synchronisation is hard to do in OpenMP
- In the pure MPI code, the intra-node messages will often be naturally overlapped with inter-node messages
  - harder to overlap inter-thread communication with inter-node messages – see later
- OpenMP codes can suffer from false sharing (cache-to-cache transfers caused by multiple threads accessing different words in thesame cache block)
  - MPI naturally avoids this
- Incremental parallelization can be insufficient to guarantee parallel efficiency

NUMA effects

- Nodes which have multiple sockets are NUMA: each socket has it's own block of RAM.
- OS allocates virtual memory pages to physical memory locations
  - has to choose a socket for every page
- Common policy (default in Linux) is *first touch* – allocate on socket where the first read/write comes from
  - right thing for MPI
  - worst possible for OpenMP if data initialisation is not parallelised
  - all data goes onto one socket
- NUMA effects can limit the scalability of OpenMP: it may be advantageous to run one MPI process per NUMA domain, rather than one MPI process per node.

NUMA effects

It is crucial to define the assignment of MPI tasks and threads explicitly

This is more important when the number of cores becomes very large (i.e. Intel Xeon Phi)

Use the APIs of your environment (for example KMP_AFFINITY on Intel Composer), the batch scheduler settings.

In any case, don’t trust the automatic assignment and check the assignment.

Using numactl can be a good idea.

Hybrid MPI+OpenMP

MPI provide several levels to implement OpenMP functions, depending on when/how threads are permitted to make MPI library calls.

Each of these levels has pros and cons.

Let’s review shortly these 4 degrees.

Hybrid MPI+OpenMP

Master-only

- all MPI communication takes place in the sequential part of the OpenMP program (no MPI in parallel regions)

Funneled

- all MPI communication takes place through the same (master) thread
- can be inside parallel regions

Serialized

- only one thread makes MPI calls at any one time
- distinguish sending/receiving threads via MPI tags or communicators
- be very careful about race conditions on send/recv buffers etc.

Multiple

- MPI communication simultaneously in more than one thread
- some MPI implementations don't support this
- …and those which do mostly don't perform well

Thread safety

Thread safety enforces an ordering in the calls to different threads/tasks.

Making MPI libraries thread-safe is difficult
- lock access to data structures
- multiple data structures: one per thread
- Adds significant overheads

MPI defines various classes of thread usage
- library can supply an appropriate implementation

MPI_Init_thread support

- **MPI_INIT_THREAD** (required, provided, ierr)
  - IN: required, desired level of thread support (integer).
  - OUT: provided, provided level (integer).

  provided may be less than required.

- Four levels are supported:
  - **MPI_THREAD_SINGLE**: Only one thread will runs. Equals to MPI_INIT.
  - **MPI_THREAD_FUNNELED**: processes may be multithreaded, but only the main thread can make MPI calls (MPI calls are delegated to main thread)
  - **MPI_THREAD_SERIALIZED**: processes could be multithreaded. More than one thread can make MPI calls, but only one at a time.
  - **MPI_THREAD_MULTIPLE**: multiple threads can make MPI calls, with no restrictions.

MPI_Init_thread

- The various implementations differs in levels of thread-safety
- If your application allow multiple threads to make MPI calls simultaneously, whitout MPI_THREAD_MULTIPLE, is not thread-safe
- Using OpenMPI, you have to use –enable-mpi-threads at configure time to activate all levels.
- Higher level corresponds higher thread-safety. Use the required safety needs.

MPI_THREAD_SINGLE

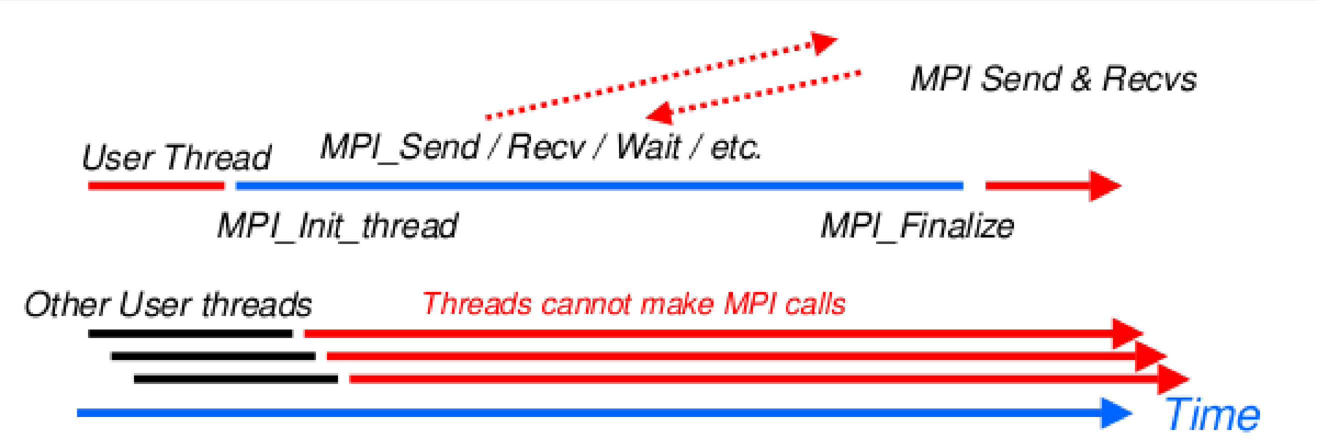
- It is fully equivalent to the master-only approach

```
!$OMP PARALLEL DO
   do i=1,10000
       a(i)=b(i)+f*d(i)
  enddo
!$OMP END PARALLEL DO
   call MPI_Xxx(...)
!$OMP PARALLEL DO
   do i=1,10000
       x(i)=a(i)+f*b(i)
   enddo
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for
      for (i=0; i<10000; i++)
      { a[i]=b[i]+f*d[i];
      }
/* end omp parallel for */
      MPI_Xxx(...);
#pragma omp parallel for
      for (i=0; i<10000; i++)
      { x[i]=a[i]+f*b[i];
      }
/* end omp parallel for */
```

MPI_THREAD_FUNNELED

- It adds the possibility to make MPI calls inside a parallel region, but only the master thread is allowed to do so



MPI_THREAD_FUNNELED

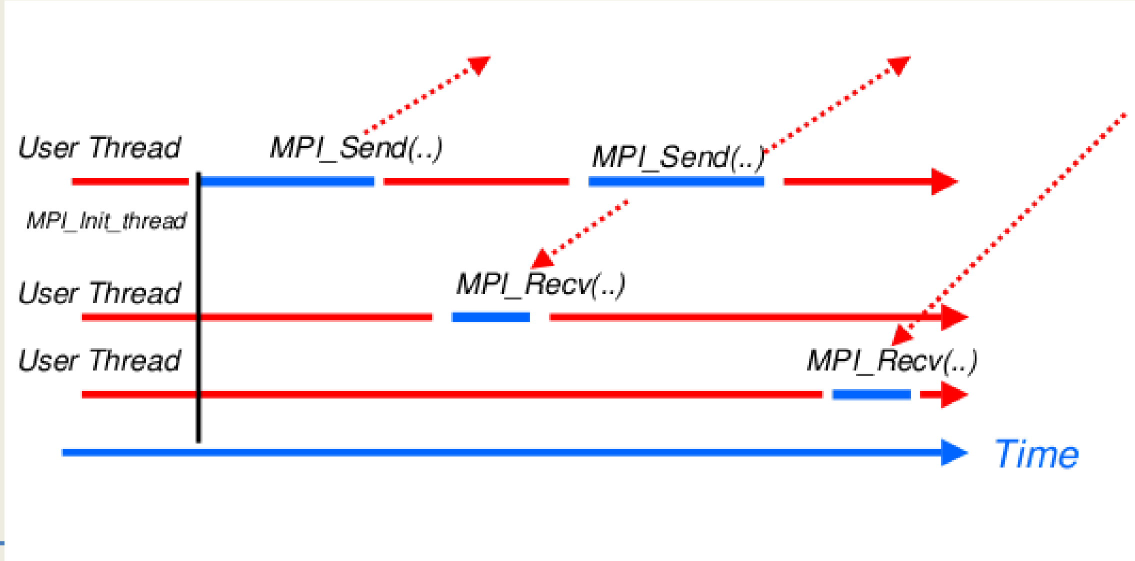
- MPI function calls can be: outside a parallel region or in a parallel region, enclosed in "omp master" clause
- There's no synchronization at the end of a "omp master" region, so a barrier is needed before and after to ensure that data buffers are available before/after the MPI communication

```
!$OMP BARRIER
!$OMP MASTER
      call MPI_Xxx(...)
!$OMP END MASTER
!$OMP BARRIER
```

```
#pragma omp barrier
#pragma omp master
    MPI_Xxx(...);
#pragma omp barrier
```

MPI_THREAD_SERIALIZED

- MPI calls are mad concurrently by two or more different threads. All the MPI communications are serialized.



MPI_THREAD_SERIALIZED

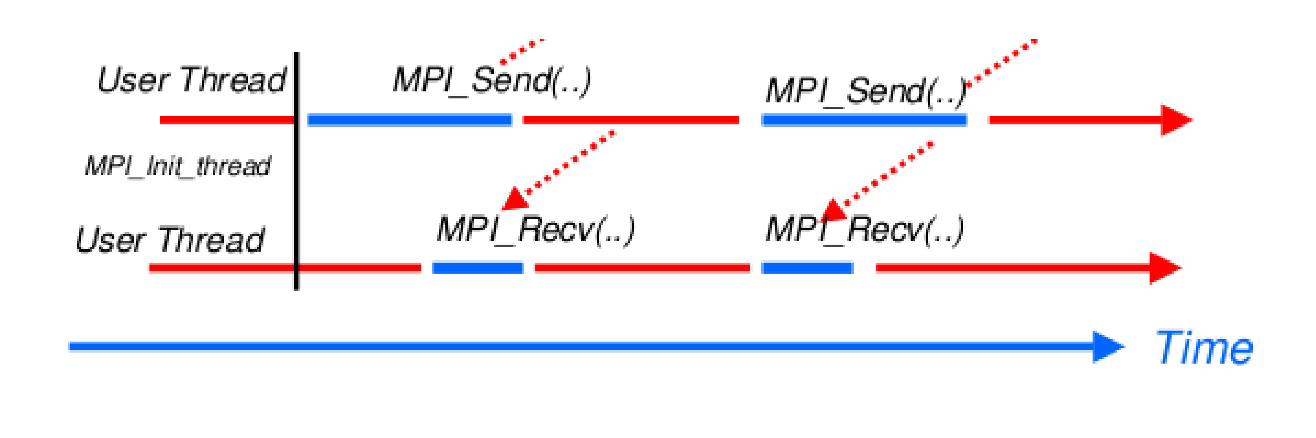
- MPI calls can be outside parallel regions, or inside, but enclosed in a "omp single" region (it enforces the serialization)
- Again, a barrier should ensure data consistency

```
!$OMP BARRIER
!$OMP SINGLE
      call MPI_Xxx(...)
!$OMP END SINGLE
```

```
#pragma omp barrier
#pragma omp single
    MPI_Xxx(...);
```

MPI_THREAD_MULTIPLE

- It is the most flexible mode, but also the most complicate one
- Any thread is allowed to perform MPI communications, without any restrictions.



Comparison to pure MPI

Funneled/serialized

- All threads but the master are sleeping during MPI communications
- Only one threads may not be able to lead up to max inter-node bandwith

Pure MPI

- Each CPU can lead up max inter-node bandwidth

Hints: overlap as much as possible communications and computations

Overlap communications and computations

- In order to overlap communications with computations, you require at least the MPI_THREAD_FUNNELED mode
- While the master thread is exchanging data, the other threads performs computation
- It is difficult to separate code that can run before or after the data exchanged are available

```
!$OMP PARALLEL
    if (thread_id==0) then
        call MPI_xxx(…)
    else
        do some computation
    endif
!$OMP END PARALLEL
```

1. Introduction
2. MPI derived datatypes
3. Non blocking collective communications
4. Topologies and neighbourhood collectives
5. One-sided communication
6. MPI and MPI+X
7. Bonus track: Endpoints

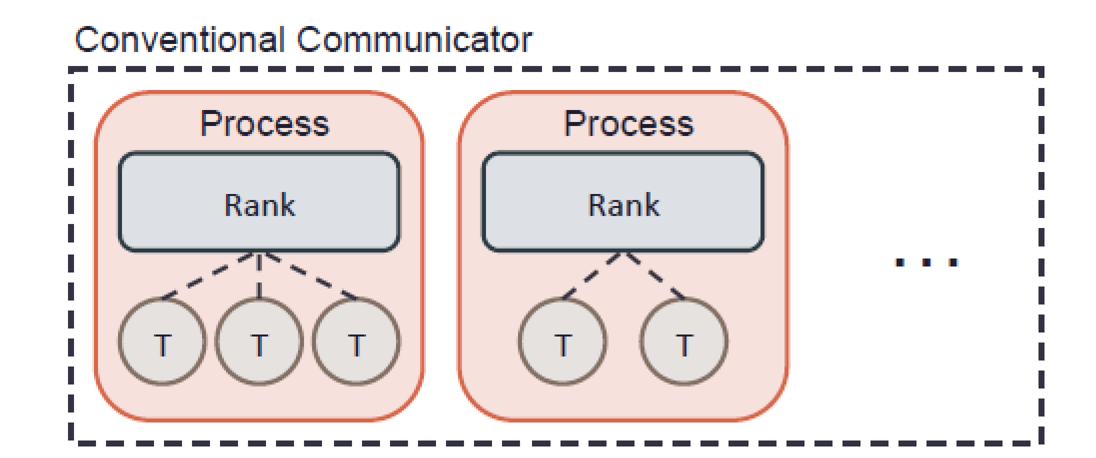
MPI endpoints

Not yet implemented in MPI3.0

Proposed for MPI4.0.

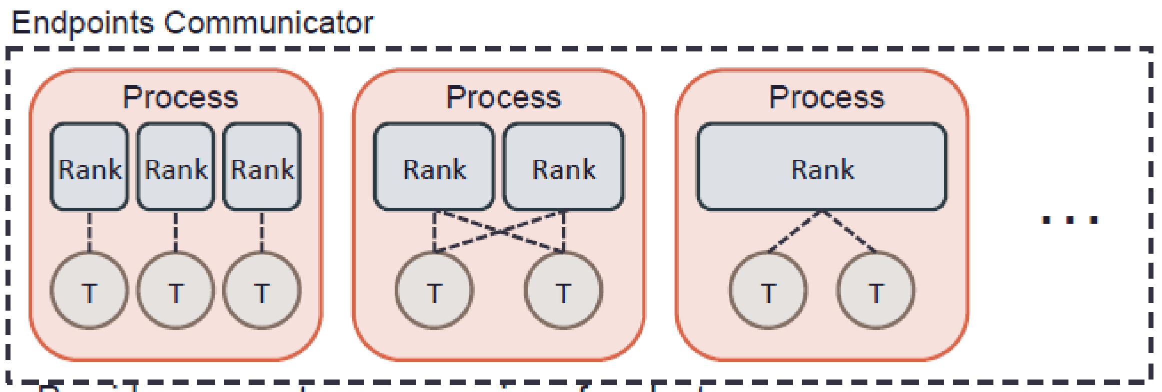
Idea is to make Multiple style easier to use and easier to implement efficiently.

Mapping of ranks to processes



In MPI there is a correspondance one-to-one between ranks.
Threads can be mapped many-to-one wrt to processes.

Mapping of ranks to processes



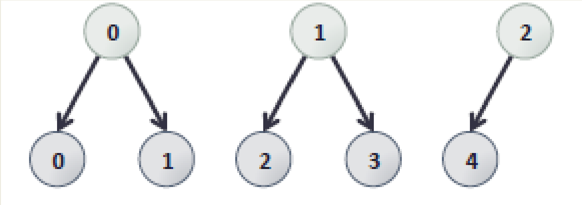
Provide a many-to-one mapping of ranks to processes

- Allows threads to act as first-class participants in MPI operations
- Improve programmability of MPI + node-level and MPI + system-level models
- Potential for improving performance of hybrid MPI + X
- A rank represents a communication “endpoint”
- Set of resources that supports the independent execution of MPI communications

Endpoints: proposed interface

```
int MPI_Comm_create_endpoints(MPI_Comm parent_comm, int
   my_num_ep, MPI_Info info, MPI_Comm *out_comm_hdls[])
```

- Each rank in *parent_comm* gets *my_num_ep* ranks in *out_comm*
- *my_num_ep* can be different at each process
- Rank order: process 0's ranks, process 1's ranks, etc.
- Output is an array of communicator handles
- *ith* handle corresponds to *ith* endpoint create by parent process
- To use that endpoint, use the corresponding handle



Endpoints: example

```
int main(int argc, char **argv) {
    int world_rank, tl;
    int max_threads = omp_get_max_threads();
    MPI_Comm ep_comm[max_threads];

    MPI_Init_thread(&argc, &argv, MULTIPLE, &tl);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

#pragma omp parallel
    {
        int nt = omp_get_num_threads();
        int tn = omp_get_thread_num();
        int ep_rank;
#pragma omp master
        {
            MPI_Comm_create_endpoints(MPI_COMM_WORLD,
                nt, MPI_INFO_NULL, ep_comm);
        }
#pragma omp barrier
        MPI_Comm_attach(ep_comm[tn]);
        MPI_Comm_rank(ep_comm[tn], &ep_rank);
        ... // divide up work based on 'ep_rank'
        MPI_Allreduce(..., ep_comm[tn]);

        MPI_Comm_free(&ep_comm[tn]);
    }
    MPI_Finalize();
}
```

Resources and credits

Many resources available online

- MPI version 3 standard is very comprehensive
- https://cvw.cac.cornell.edu/MPIoneSided is a good resource
- https://htor.inf.ethz.ch/publications/img/mpi3-rma-overview-and-model.pdf
- http://www.mpich.org/static/docs/v3.2/www3/
- Credits to EPCC online courses and T. Hoefler's