Background Title
Foreground

SCIENTIFIC & DATA-INTENSIVE COMPUTING

Luca Tornatore - I.N.A.F.

Advanced HPC 2024-2025 @ Università di Trieste

Advanced MPI

SCIENTIFIC & DATA-INTENSIVE COMPUTING

Luca Tornatore - I.N.A.F.

Advanced HPC 2024-2025 @ Università di Trieste

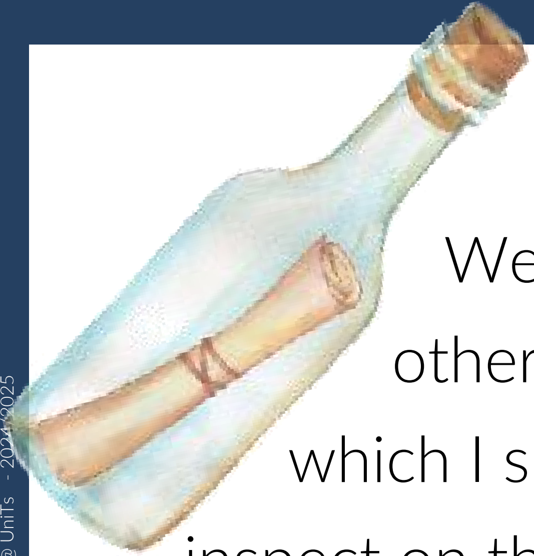
Outline

Advanced Usage of
some **MPI** features
on *topology* awareness,
shared memory &
one-sided communications

- Basics of **one-sided** communications
- Building a hierarchy of Communicators that reflects the **topology**
- Exchanging data in **shared-memory** windows among MPI processes

main reference: "Using Advanced MPI", Gropp, Hoefler, Thakur and Lusk

Outline



We will not cover other important topics, which I suggest you to inspect on the references that are in the last slides.

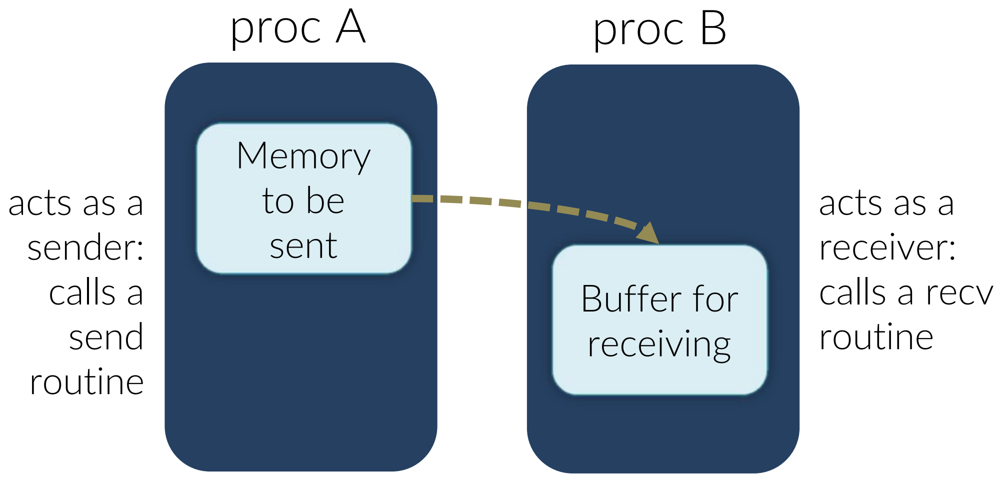
- Derived Data-types
- More details on Groups and Communicators
- Virtual topologies (cartesian and Graph communicators) and neighbourhood collectives
- Non-blocking collectives

main reference: "Using Advanced MPI", Gropp, Hoefler, Thakur and Lusk

Two-Sided communications

By its very nature, the message-passing paradigm is designed around the concept of cooperative exchange of informations among two or more processes whose address space are isolated and not directly inaccessible by other processes.

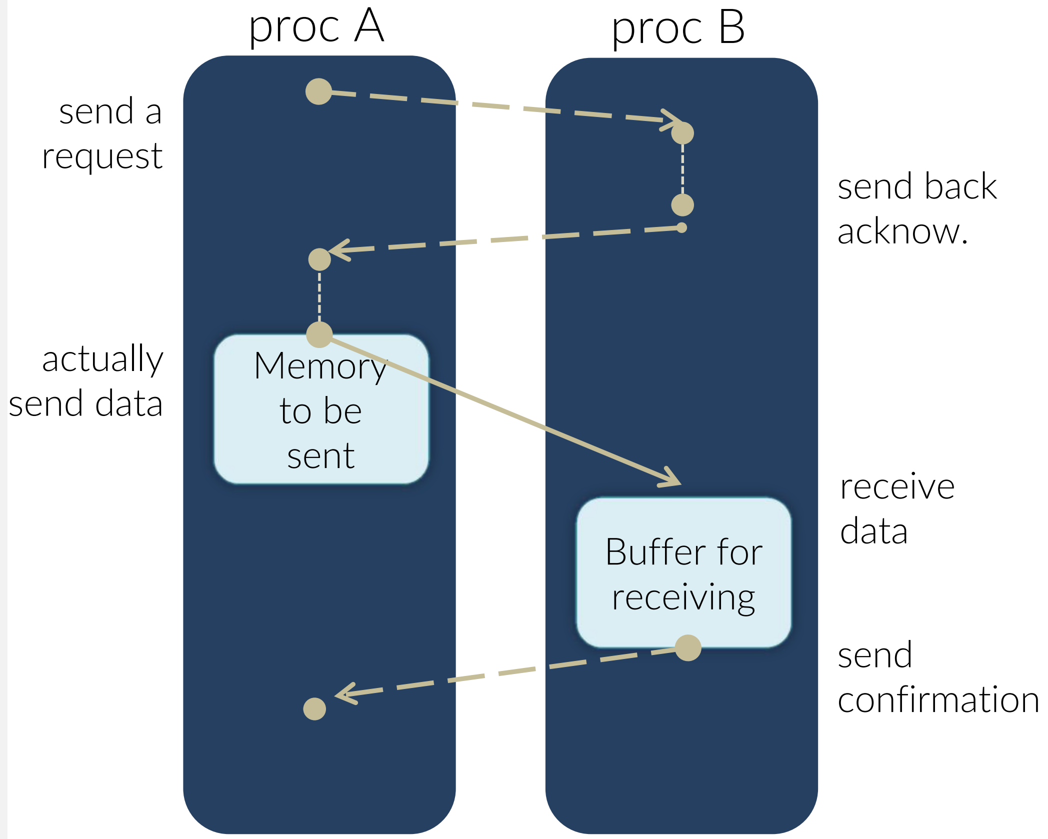
This model is very effective in protecting the memory access and in making clear what memory location will be modified and when that happens


proc A
proc B
Memory to be sent
Buffer for receiving
acts as a sender: calls a send routine
acts as a receiver: calls a recv routine

Two-Sided communications

However, this model has also several cons.
The processes act as peers and must collaborate; as such, the "sender" actually send a request that must be accepted by the receiver.

Moreover, every send must be matched by a receive, and that make certain types of code more complex.

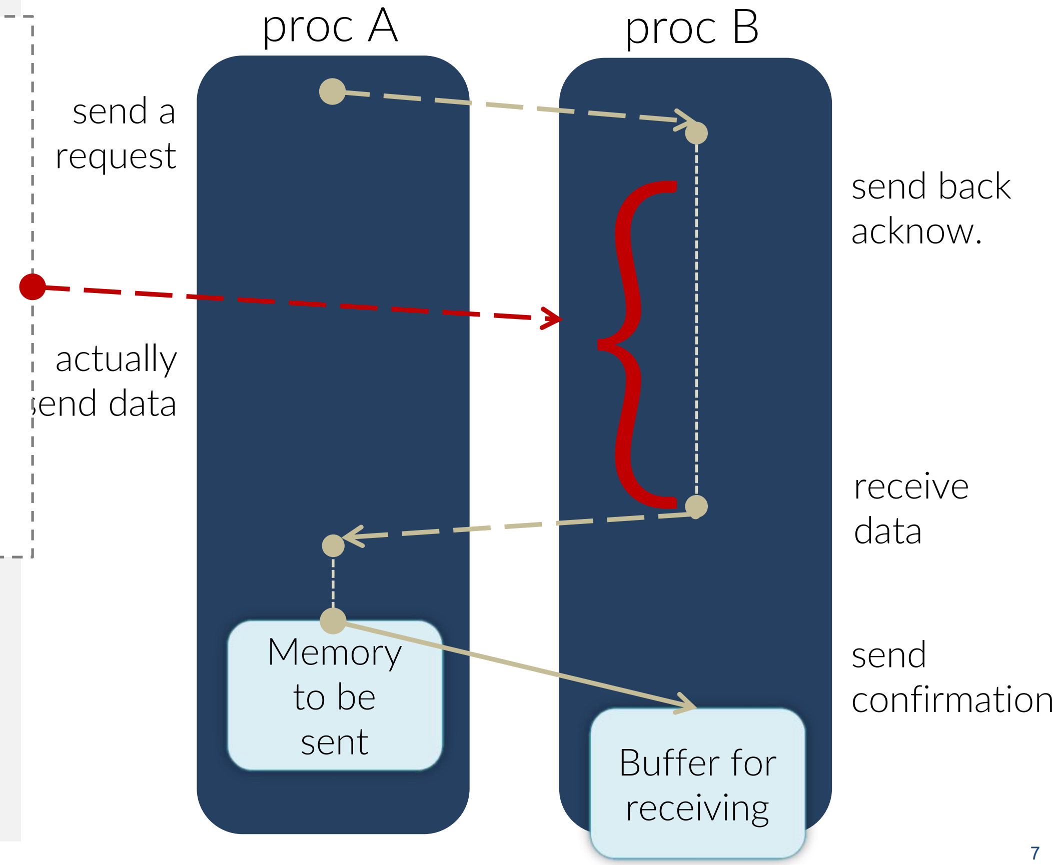


Two-Sided communications

However, t
cons.
The proces
collaborate
actually ser
accepted b

This delay may be large, depending on the status of process B.
Even the Send operation will be delayed as well, because it requires the cooperation of both processes

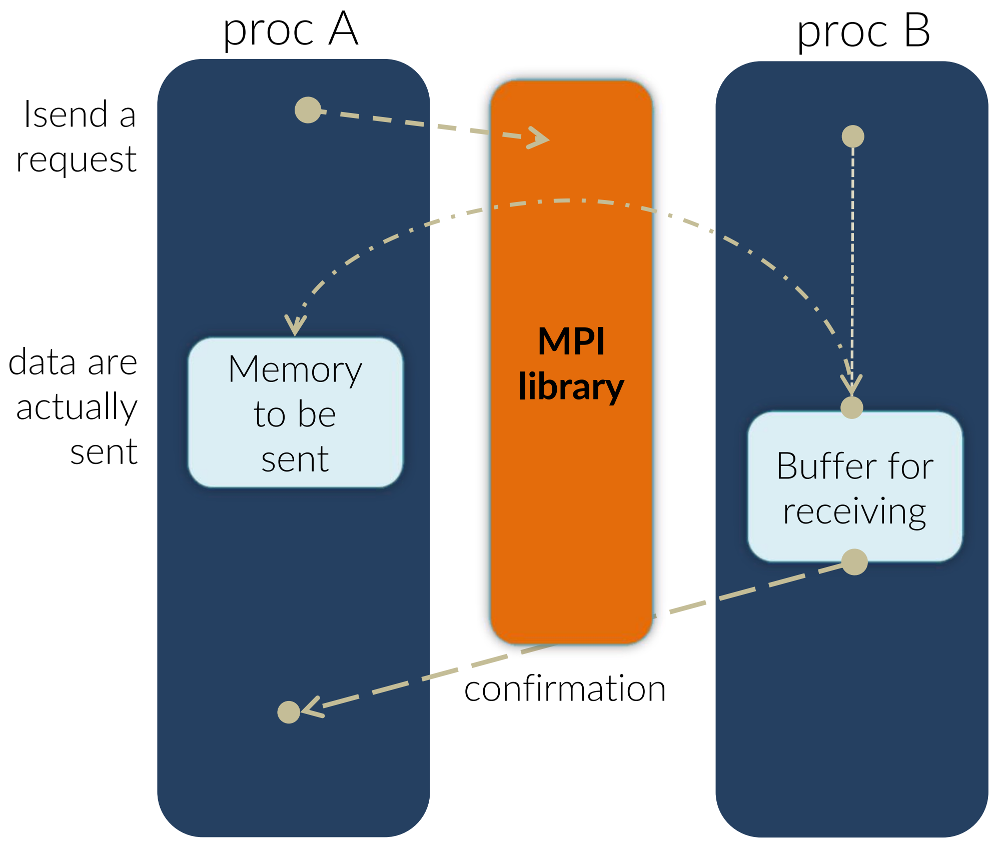
Moreover, every send must be matched by a receive, and that make certain types of code more complex.



Two-Sided communications

The *non-blocking* routines mitigate the difficulties linked to the synchronization: the MPI library manages the operations after the sending process posts his request for sending data.
The sendig process may even overlook the confirmation about the data receptions has concluded, if not needed by its semantics.

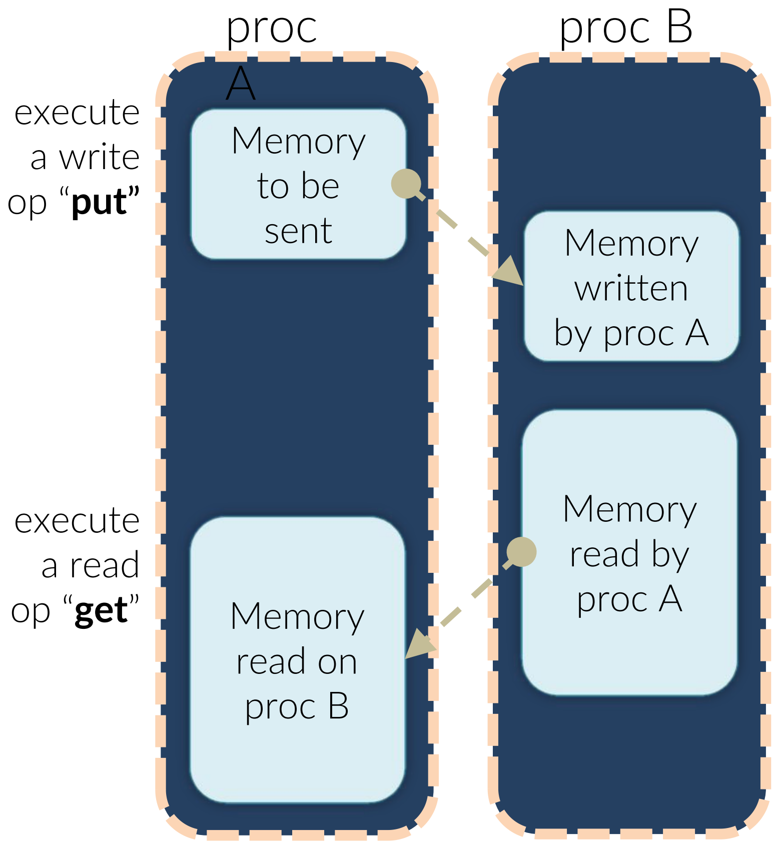
While this makes easier to implement several algorithms, it does not change the fact that the two process need to cooperate.



One-Sided communications

On the other hand, if ***R****emote* ***M****emory* ***A****ccess (RMA)* was possible, the protocol may be much more relaxed, at the cost of a much larger burden - on the shoulder of the developer:

to ensure a correct synchronization of the operations and the absence of data races or situations with an undefined behaviour.



Proc B does not need to collaborate. In fact, it may even not "know" that proc A is writing or reading.

well, for consistency reasons, it is advisable that it "knows" that somebody may be writing

One-Sided communications

The leading idea of one-sided communication models is to disentangle data exchange from the need of processes synchronization

– data movement possible without requiring that the “collaboration” (e.g. sync) of remote process
– Each process exposes a range of its memory (a “window”)
– Other processes can directly read from or write to the exposed memory window

Hence, using one-sided can help in

- reducing communication overhead
- overlapping communication and computation
- improving scalability

One-Sided communications

RMA and Shared-Memory are two different concepts.

In RMA the players offer a “window” to the other players to access precise memory locations and that access happens in well-defined moments that are circumscribed by *fences*, *epochs* or *locks*.

The Shared-Memory is more general: the players share the memory and the access, either in reading or writing, is possible on the entire (shared)memory and the correctness of the operations is entire responsibility of the programmer.

One-Sided communications

At abstract level, RMA workflow is something like:

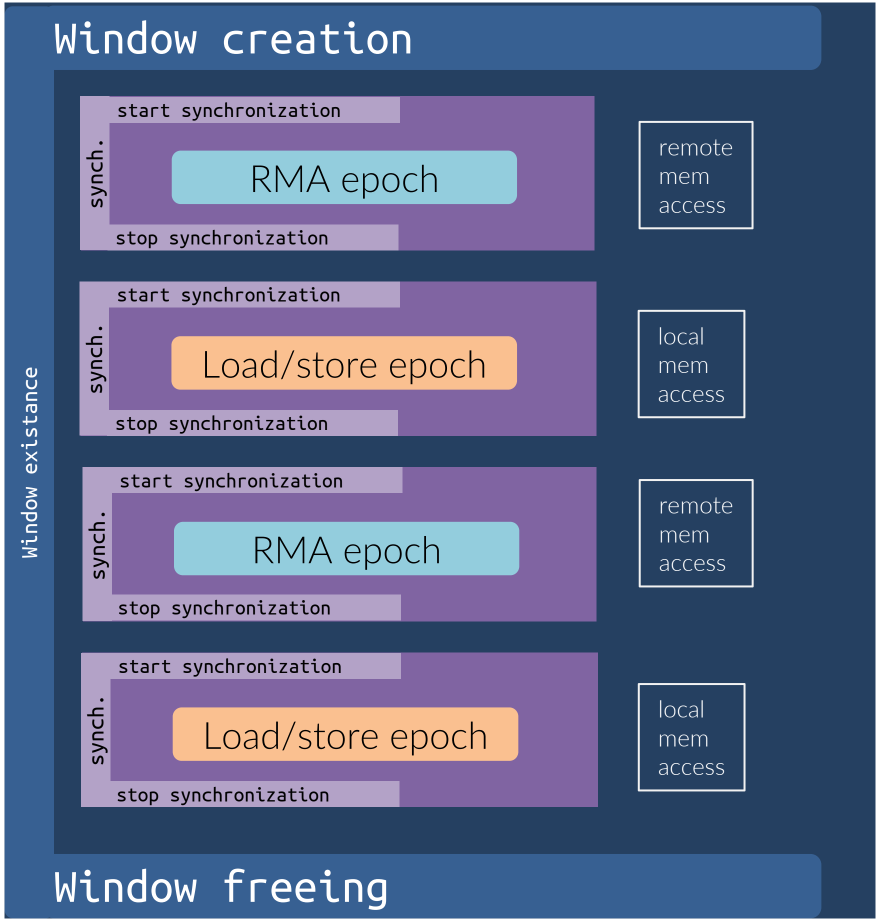
```
create_the_memory_window();

advertise_an_epoch_of_remote_write_access();
..
perform_remote_write_access()
..
close_the_epoch_of_remote_write_access();
..

close_the_memory_window();
```

Onse-sided communications

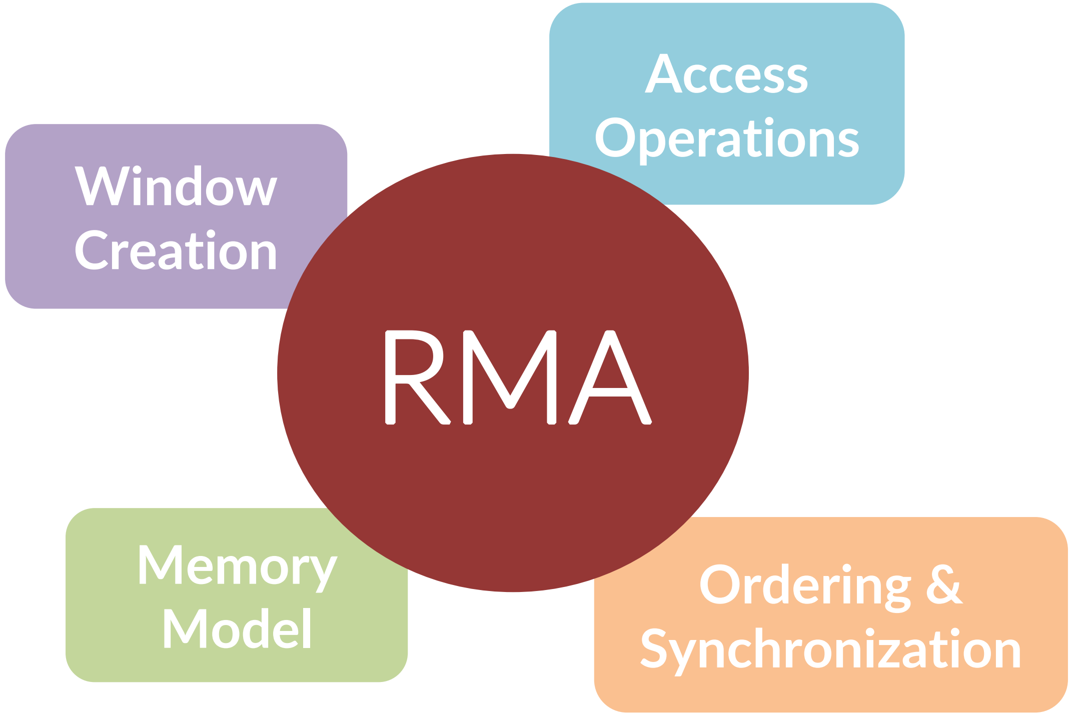
A broad general overview of the workflow with one-sided communications.

Within the existance of a memory window, which must be created and freed, many *epochs* subceed.

Epochs of *remote* memory access must be separated from epochs of *local* memory access by synchronization calls.



Key concepts in RMA

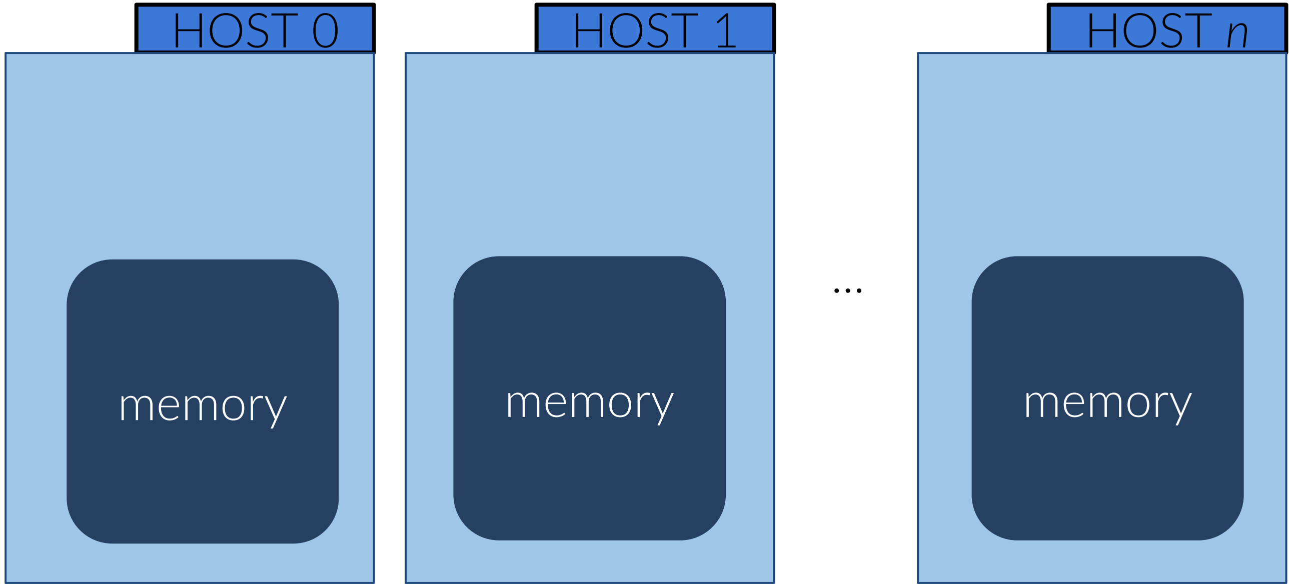


Key concepts in RMA

Creating Memory Windows

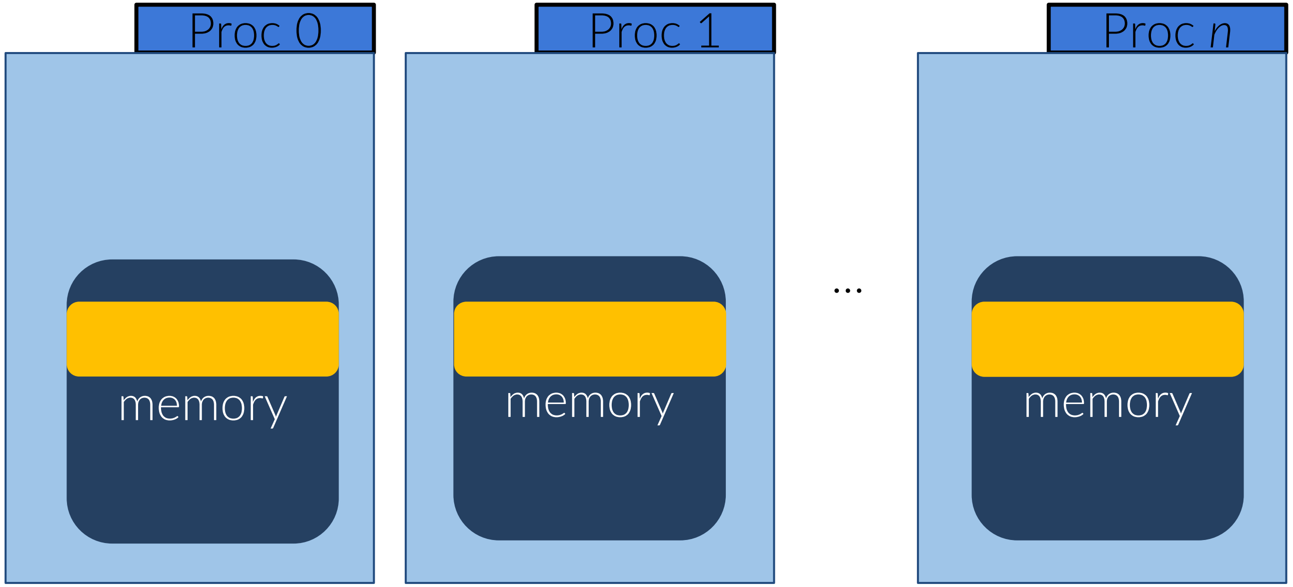
Creating memory windows

- The memory of each process is only locally accessible to each MPI process



Creating memory windows

- The programmer has to make an **explicit call to MPI** routines and declare that a region of memory is accessible from remote by the processes ***within a given communicator***

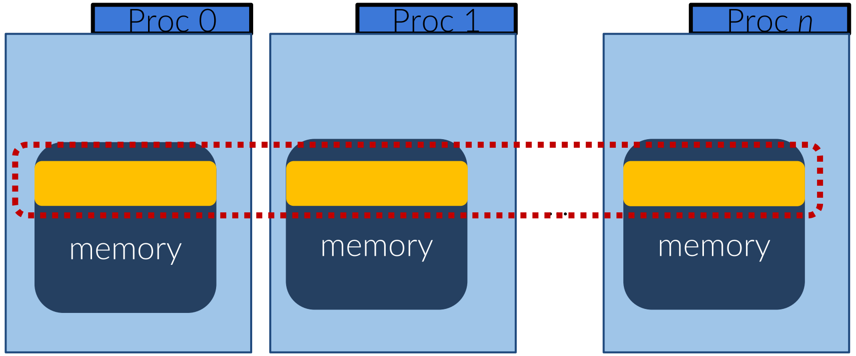


Creating memory windows

- The memory of each process is only locally accessible to each MPI process

- The programmer has to make an explicit call to MPI routines and declare that a region of memory is accessible from remote by the processes *within a given communicator*
  - the region made accessible by RMA ia called “a window”
  - the window creation is a *collective operation*

- Once a window is created, the processes are able to write on/read from it remotely without any collaboration with the process that owns the memory



Creating memory windows

There are **four possible ways** to create a window

- **`MPI_Win_allocate`**
  allocates a memory regione *and* makes it available - the memory will be released when you *free* the window

- **`MPI_Win_create`**
  a memory region already exists, it will be made a window remotely accessible

- **`MPI_Win_create_dynamic`**
  creates a window disjointly from a precise memory region; memory buffers will be dynamically added to / removed from the window at any time

- **`MPI_Win_allocate_shared`**
  allocates memory in a shared memory nodes *and* creates a window linked to it

Creating memory windows

Note on memory buffers to be linked to a window and memory allocation

Some MPI implementations may be more efficient if the memory address is *aligned* to the memory pages (and hence to cache lines).

A further important detail may be that the size of the memory buffer is a multiple of the pagse size.

On `posix` system you may use `posix_memalign` or the C11 `memalign`. An alternative is to use `MPI_Alloc_mem` or `MPI_Win_allocate`.

Creating memory windows

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

`size`	the size of the memory buffer, in bytes (integer)
`disp_unit`	local units for access offset, in bytes (pos. integer)
`info`	a handle to an info argument
`comm`	an handle to a comm object
`baseptr`	the pointer to the allocated mem buffer (returned by the call)
`win`	a pointer to a window object (returned by the call)

Creating memory windows

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

Note the type

```
int     N;
double *data;
MPI_win mywin;

...
MPI_Win_allocate ( N*sizeof(double),
sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD,
&data, &mywin );

... // use data
MPI_Win_free ( &mywin );
```

local data size of double type

always valid not to specify anything here; this is an object to pass hints that may be usefule to optimize memory management

It is collective within this communicator

Creating memory windows

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

```
int     N;
double *data;
MPI_win mywin;

...
MPI_Win_allocate ( N*sizeof(double),
sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD,
&data, &mywin );

... // use data
MPI_Win_free ( &mywin );
```

The call itself is a collective, but the window's size can be different at every MPI process

Creating memory windows

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

```
typedef struct { double d; int j;
                 char *buffer; } data_t;
data_t *dat;
MPI_win mywin;

...
MPI_Win_allocate ( N*sizeof(data_t), 1, MPI_INFO_NULL,
MPI_COMM_WORLD, &data, &mywin );

... // use data
MPI_Win_free ( &mywin );
```

```
typedef struct { double d; int j;
                 char *buffer; } data_t;
data_t *dat;
MPI_win mywin;

...
MPI_Win_allocate ( N*sizeof(data_t), sizeof(data_t),
MPI_INFO_NULL, MPI_COMM_WORLD, &data, &mywin );

... // use data
MPI_Win_free ( &mywin );
```

Creating memory windows

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

NOTE

In principle, whenever possible, **you should always prefer MPI_Win_allocate( )** over m-allocating memory and creating a windows attached to the returned pointer.

Window access performance depends critically on the fact that memory is *pinned*, and the MPI library is supposed to ask the OS such an allocation for each calling MPI task, so that the Network Interface can directly read/write without the CPU support (DMA - Direct Memory Access).

Creating memory windows

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

```
int     N;
double *data;
MPI_win mywin;

MPI_Alloc_mem ( N*sizeof(double), MPI_INFO_NULL, &data );
data[j] = ...;
...
MPI_Win_create ( data, N*sizeof(data_t), sizeof(double),
                 MPI_INFO_NULL, MPI_COMM_WORLD, &mywin );

... // use data
MPI_Win_free ( &mywin );
MPI_Free ( data );
```

Creating memory windows

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

```
int     N;
double *data;
MPI_win mywin;

MPI_Win_create_dynamic ( MPI_INFO_NULL, MPI_COMM_WORLD, &mywin );

MPI_Alloc_mem ( N*sizeof(double), MPI_INFO_NULL, &data );
data[j] = ...;
...
MPI_Win_attach ( mywin, data, N*sizeof(double) );
... // use data
MPI_Win_detach ( mywin, data );
MPI_Win_free ( &mywin );
```

Setting info for memory windows

The `MPI_info` argument provides optimization hints to the runtime about the expected usage pattern of the window. The following info keys are predefined:

```
"no_locks" (boolean, default: false)

"accumulate_ordering" (string, default rar,raw,war,waw)

"accumulate_ops" (string, default: same_op_no_op)

"mpi_accumulate_granularity" (integer, default 0)
```

You can set an info argument by

```
MPI_Info info;
MPI_Info_create( &info );
MPI_Info_set( info, "no_locks", "true" );
MPI_Win_create( ..., info, ... )
MPI_Info_free( &info );
```

Setting info for memory windows

`"no_locks" (boolean, default: false)`

if set to **`true,`** then the implementation may assume that passive target synchronization (i.e., `MPI_WIN_LOCK`, `MPI_WIN_LOCK_ALL`) will not be used on the given window. This implies that this window is not used for 3-party communication, and RMA can be implemented with no (less) asynchronous agent activity at this process.

`"accumulate_ordering" (string, default rar,raw,war,waw)`

controls the ordering of accumulate operations at the target.

`“accumulate_ops" (string, default: same_op_no_op)`

if set to "**`same_op`**", the implementation will assume that all concurrent accumulate calls to the same target address will use the same operator.
If set to "**`same_op_no_op`**", then the implementation will assume that all concurrent accumulate calls to the same target address will use the same operator or `MPI_NO_OP`.
This can eliminate the need to protect access for certain operators where the hardware can guarantee atomicity.

Setting info for memory windows

`"mpi_accumulate_granularity" (integer, default 0)”`

provides a hint to implementations about the desired *synchronization granularity for accumulate operations*, i.e., the size of memory ranges in bytes for which the implementation should acquire a synchronization primitive to ensure atomicity of updates.

If the specified granularity is not divisible by the size of the type used in an accumulate operation, it should be treated as if it was the next multiple of the element size.

For example, a granularity of 1 byte should be treated as 8 in an accumulate operation using MPI_UINT64_T. By default, this info key is set to 0, which leaves the choice of synchronization granularity to the implementation. If specified, all MPI processes in the group of a window must supply the same value.

Key concepts in RMA

Data movement

Move memory

You can read, write, modify data atomically

- `MPI_PUT, MPI_GET`
- `MPI_Accumulate`
- `MPI_Get_accumulate`
- `MPI_Compare_and_swap`
- `MPI_Fetch_and_op`

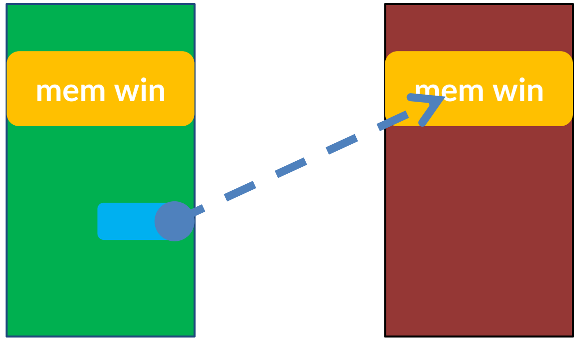
atomic operations

Move memory: put and get

move data from origin to target

```
MPI_Put ( void *origin_addr, int origin_count,
MPI_Datatype origin_dtype,
int target_rank,
MPI_Aint target_disp, int target_count,
MPI_Datatype target_dtype,
MPI_Win win )
```

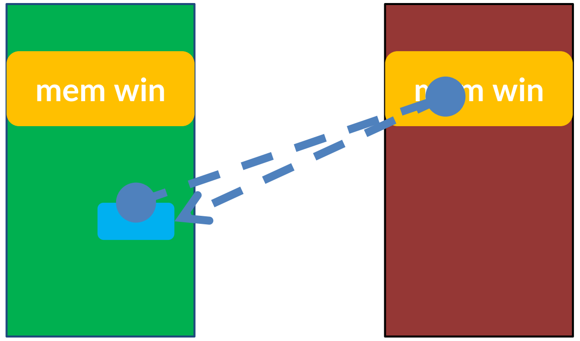
origin (calling proc) **target** (owner of accessed mem)



move data to origin from target

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

origin (calling proc) **target** (owner of accessed mem)

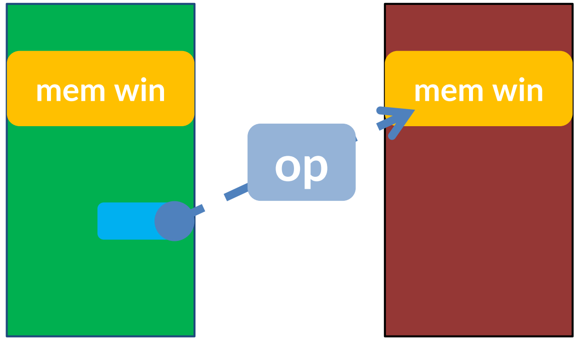


Move memory: accumulate

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

This implement an **atomic** update operation

- perform the **op** reduction operation between the origin data and the target data
  - OP are the reduction operations defined for MPI_Reduce: MPI_SUM, MPI_PROD, MPI_OR, MPI_NO_OP, …
  - user-defined operations are not allowed
- if **op**=MPI_REPLACE you have an **atomic put**



Why do we need atomics ?

```
if (Rank == 0) N = 1;
else N = 0;

MPI_Win_allocate ( N*sizeof(int), sizeof(int), MPI_INFO_NULL,
                   MPI_COMM_WORLD, &global_counter, &mywin );


...

int counter;
if (Rank > 0 )
 {
    MPI_Get( &counter, 1, MPI_INT,     // origin
             0,                        // target rank
             0, 1, MPI_INT, mywin );   // target + win

    while ( counter < MAX ) {
          do_something();
          counter++;
          MPI_Put( &origin, 1, MPI_INT, 0, 0, 1, MPI_INT, mywin); }
 }
```

Is this gonna work?

Why do we need atomics ?

```
if (Rank == 0) N = 1;
else N = 0;

MPI_Win_allocate ( N*sizeof(int), sizeof(int), MPI_INFO_NULL,
                   MPI_COMM_WORLD, &global_counter, &mywin );


...

int counter;
if (Rank == 1 )
 {
    MPI_Get( &counter, 1, MPI_INT, 0, 0, 1, MPI_INT, mywin );

    while ( there_is_something_todo ) {
      do_something();
      counter++;
      MPI_Put ( &counter, ... ); }
  }
```

Is this gonna work?

Why do we need atomics ?

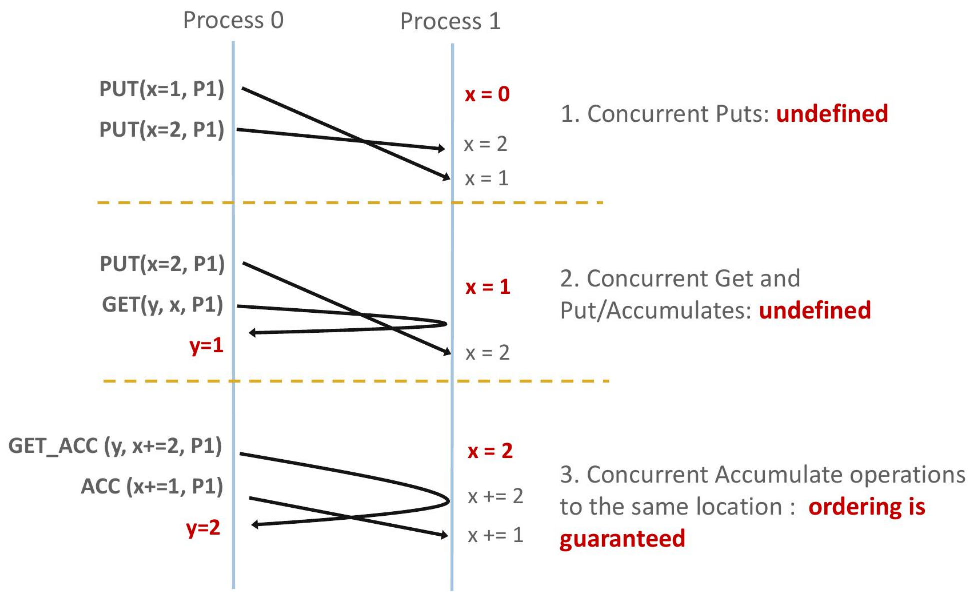
```
if (Rank == 0) N = 1;
else N = 0;

MPI_Win_allocate ( N*sizeof(int), sizeof(int), MPI_INFO_NULL,
                   MPI_COMM_WORLD, &global_counter, &mywin );

...

int counter;
if (Rank == 1 )
 {
    MPI_Get( &counter, 1, MPI_INT, 0, 0, 1, MPI_INT, mywin );

    while ( there_is_something_todo ) {
      do_something();
      counter++;
      MPI_Put ( &counter, ... ); }
  }
```

```
if (Rank == 0) N = 1;
else N = 0;

MPI_Win_allocate ( N*sizeof(int), sizeof(int), MPI_INFO_NULL,
                   MPI_COMM_WORLD, &global_counter, &mywin );

...

int counter;
if (Rank > 0 )
 {
    MPI_Get( &counter, 1, MPI_INT,     // origin
              0,                       // target rank
              0, 1, MPI_INT, mywin );  // target + win

    while ( counter < MAX ) {
          do_something();
          counter++;
          MPI_Put( &origin, 1, MPI_INT, 0, 0, 1, MPI_INT, mywin); }
 }
```

NO. Neither will work as intended.

The MPI standard does **not enforce the order** of execution of put and get operations. As such, the puts and gets in the two snippet above will be mixed randomly and, *since they are concurrent - i.e. on the same memory location* - the result will be undefined.

Ordering and Sync RMA ops


Process 0
Process 1
PUT(x=1, P1)
PUT(x=2, P1)
x = 0
x = 2
x = 1
1. Concurrent Puts: undefined
PUT(x=2, P1)
GET(y, x, P1)
y=1
x = 1
x = 2
2. Concurrent Get and Put/Accumulates: undefined
GET_ACC (y, x+=2, P1)
ACC (x+=1, P1)
y=2
x = 2
x += 2
x += 1
3. Concurrent Accumulate operations to the same location : ordering is guaranteed

from "Advanced MPI programming", SC24

- MPI does not ensure any ordering for put and get
  → Concurrent puts and gets have an undefined result
  → A get concurrent with put/accumulate have an undefined result
- concurrent accumulate have a result defined accordingly to the order of operations.
  As we'll see in next slides:
  Accumulate with op=MPI_REPLACE
  Get_accumulate with op=MPI_NO_OP
- Accumulate ops from the same process are ordered by default
  - you can tell the MPI not to order, as optmization hint
  - you can ask RAW, WAR, RAR or WAW

note: here above "concurrent" mean "concurrent on the same memory location"

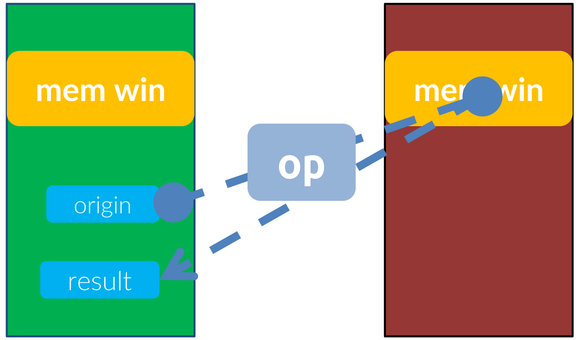
Move memory: get_accumulate

```
MPI_Get_accumulate ( void *origin_addr, int origin_count, MPI_Datatype origin_dtype,
                     void *result_addr, int result_count, MPI_Datatype result_dtype,
                     int target_rank,
                     MPI_Aint target_disp, int target_count, MPI_Datatype target_dtype,
                     MPI_Op op, MPI_Win win )
```

This implement an atomic read-modify-write operation

- perform the op reduction operation between the origin data and the target data
    - OP are the reduction operations defined for MPI_Reduce: MPI_SUM, MPI_PROD, MPI_OR, MPI_NO_OP, …
    - user-defined operations are not allowed

- if **op**=MPI_REPLACE you have an atomic swap
- if **op**=MPI_NO_OP you have an atomic get
- the result of the op is stored in the target buffer
- the value at target before op is stored in result buffer
- basic datatype must match



Move memory: cas and fop

```
MPI_Fetch_and_op ( void *origin_addr, void *target_addr,
                   MPI_Datatype origin_dtype, int target_rank,
                   MPI_Aint target_disp, MPI_Op op, MPI_Win win )

MPI_Compare_and_swap ( void *origin_addr, void *compare_addr, void *target_addr,
                       MPI_Datatype origin_dtype, int target_rank,
                       MPI_Aint target_disp, MPI_Win win )
```

FOP: it is an MPI_Get_accumulate but for 1 basic data at a time → more optimized

CAS: it is an atomic swap between origin and target if the target value is equal to compare value;

Move memory: request-based

```
MPI_Rput
MPI_Rget
MPI_Raccumulate
MPI_Rget_accumulate
```

MPI offers also Request-based versions of put, get, accumulate, get_accumulate.

I.e. routines, to be used only within *passive synchronization (i.e. lock/unlock; see later)*, that return a request handle that can be managed using MPI_Wait.

Key concepts in RMA

Ordering and Sync

Ordering and Sync RMA ops

Access model:

- When a process is allowed to read/write remote memory ?
- When data written by process A are available for process B ?

All RMA routines are “non-blocking” routines, meaning that when they return the actual data transfer may continue.
As such, to ensure the correctness of the subsequent operations (namely, the availability of results), they need to be appropriately surrounded by synchronization calls

- to ensure that operations are completed
- to ensure that cache sync have been done

Until the appropriate synchronization has not been performed, local buffer of put, accumulate or get should not be accessed until the op completes locally. In turn, the buffer at a target should not be accessed until any remote put/accumulate/get on it has not completed remotely.

Ordering and Sync RMA ops

There are two types of synchronization:

active both origin and target have to call sync routines

`MPI_Fence`	"active target" A collective call that surround RMA routines to isolate different access types

`MPI_Win_post, MPI_Win_start,`
`MPI_Win_complete,`
`MPI_Win_wait`
	"generalized active target"
	Collectives that apply to a sub-group, to restrict the overhead of the needed communication

passive only the origin calls the sync routine

`MPI_Win_lock, MPI_Win_unlock`

Ordering and Sync RMA ops

Data access happens between “epochs”

- ***Access*** *epochs:* a set of operations issued by origin processes
- ***Exposure*** *epochs:* remote processes can update a target’s memory window

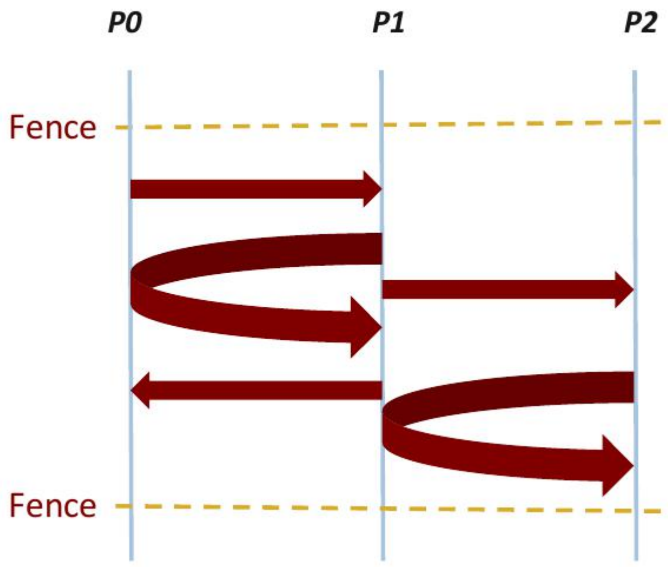
Epochs define operation ordering and completion semantics

The synchronization model provides a way to establish epochs

Fences

- A collective call, which opens and closes access and exposure epochs on *all processes* in the window

1. everyone in the windows post a **`MPI_Win_fence`** to open the epoch
2. everyone is allowed to issue MPI_Put and MPI_Get
3. everyone posts a **`MPI_Win_fence`** to close the epoch
4. all operations completes at the exit of the second fence synchronization



from "Advanced MPI programming", SC24

Fences: sematics

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

The assert argument is used to provide optimization hints to the implementation:
assert == 0 is always valid.

Valid values may be combined with a bitwise OR operation (assert1 | assert2)

```
MPI_Win_fence ( 0, mywin );
while ( there_is_something_todo ) {
        ret = do_something( );
        MPI_Accumulate( &ret, 1, MPI_INT, register[Rank], 0, 1, MPI_INT, MPI_SUM, mywin ); }
MPI_Win_fence ( 0, mywin );
```

Fences: assertions

`MPI_Win_fence (int assert, MPI_Win win )`

`MPI_MODE_NOSTORE`	The local window was not updated by any local store since the last call to MPI_Win_fence. This assert refers to operations *before* the present fence call. `[ can be different on processes ]`
`MPI_MODE_NOPUT`	The local window *will not* be remotely updated by put or accumulate between the present fence call and the next one. This assert involve *future* operations. `[ can be different on processes ]`
`MPI_MODE_NOPRECEDE`	The called fence *will* not conclude any RMA calls made by the process calling the fence; then, no RMA calls should have been made between this call and the previous call (basically it says “no RMA to complete”). `[ must be the same for all processes ]`
`MPI_MODE_NOSUCCEED`	the symmetric than before: no RMA calls *will be made* on this window before the next fence call (“no RMA to start”). `[ must be the same for all processes ]`

The attributes can be OR’d :

```
MPI_Win_fence ( MPI_MODE_NOSTORE | MPI_MODE_NOPRECEDE, win )
```

Fences: assertions

```
MPI_Win_fence (int assert, MPI_Win win )

  example

  MPI_Win_fence(MPI_MODE_NOPRECEDE, win);

  MPI_Put(..., target_a, ..., win);
  MPI_Put(..., target_b, ..., win);

  MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT | MPI_MODE_NOSUCCEED,
                win);
```

PSCW

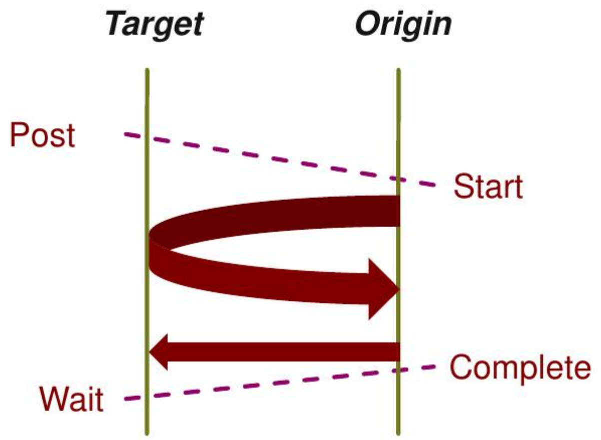
Similar to **Fence**, but not collective: origin and target declare themselves, in a sense

Target: declares **exposure** epoch
- `MPI_Win_post` initiates it
- `MPI_Win_wait` finalizes it

Origin: declares **access** epoch
- `MPI_Win_start` starts it
- `MPI_Win_complete` closes it

All synchronizations are allowed to block and defer to ensure the P-S/C-W ordering

Every process can be origin *and* target



from "Advanced MPI programming", SC24

PSCW

Similar to **Fence**, but not collective: origin and target declare themselves, in a sense

Target: declares **exposure** epoch

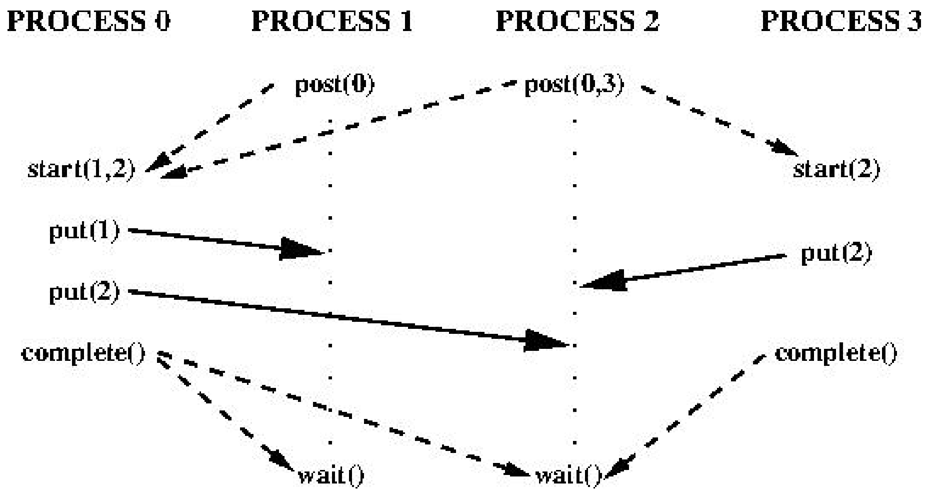
- **MPI_Win_post** initiates it
- **MPI_Win_wait** finalizes it

Origin: declares **access** epoch

- **MPI_Win_start** starts it
- **MPI_Win_complete** closes it

All synchronizations are allowed to block and defer to ensure the P-S/C-W ordering

Every process can be origin *and* target



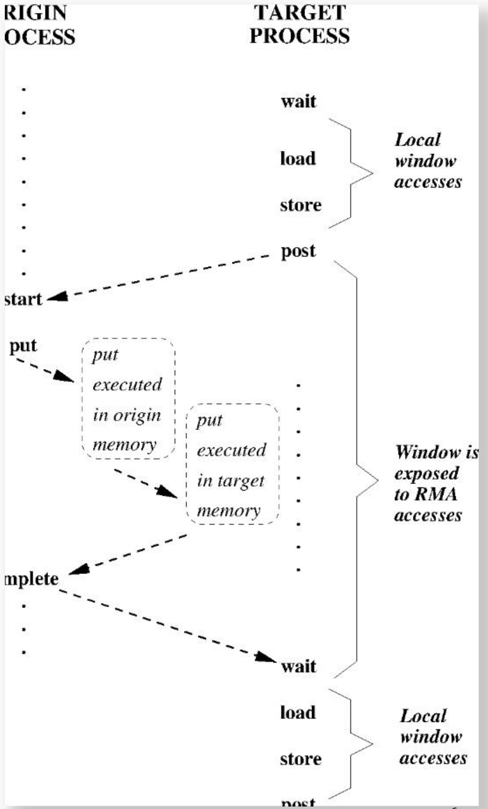
from MPI-forum: link

Advanced MPI

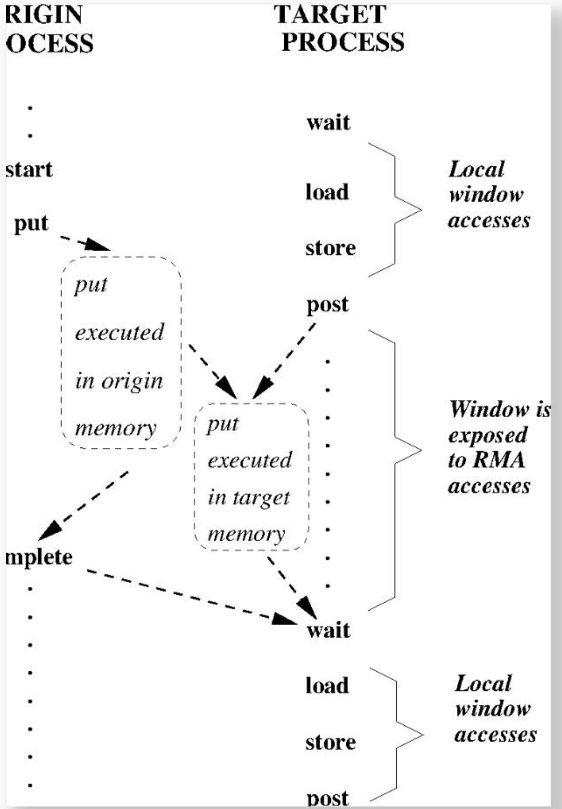
PSCW

What it means that "All synchronizations are allowed to block and defer to ensure the P-S/C-W ordering" ?

Active target communication



Active target communication, with *weak* synchronization



from MPI-forum: link

PSCW: sematics

```
MPI_Win_start    ( MPI_Group to_group, int assert, MPI_Win win )
MPI_Win_complete ( MPI_Win win )

MPI_Win_post ( MPI_Group from_group, int assert, MPI_Win win )
MPI_Win_wait ( MPI_Win win )
```

These routines are somehow equivalent to the fence call, but for the fact that they are not mandatorily executed by *all* the processes that are in the group of processes that created the window.
They may be execute by a sub-group, even by 2 processes.

The processes that *expose* their window initiate the *exposure epoch* with `MPI_Win_start` and ends it with `MPI_Win_wait`.
Instead, the processes that will *access* the windows initiate the *access epoch* by `MPI_Win_post` and close it by `MPI_Win_complete`.

PSCW: assertions

```
MPI_Win_start (MPI_Group to_group, int assert, MPI_Win win )

MPI_Win_post (MPI_Group from_group, int assert, MPI_Win win )
```

`MPI_Win_post`'s asserts	(zero is always correct)
`MPI_MODE_NOSTORE`	The local window was not updated by any local store since the last call to `MPI_Win_complete`.
`MPI_MODE_NOPUT`	The local window *will not* be remotely updated by put or accumulate between the present fence call and the next matching **`MPI_Win_complete`**
`MPI_MODE_NOCHECK`	The matching **`MPI_Win_start`** have not been issued by a process that is an origin of an RMA with this process as a target (basically: no cross-RMAops). The matching **`MPI_Win_start`** *must* use the same assert.
`MPI_Win_start`'s asserts	(zero is always correct)
`MPI_MODE_NOCHECK`	Guarantees that the matching calls to **`MPI_Win_post`** have already been made

PSCW: example

```
MPI_Group world_group;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);


if ( Me == origin ) {
            MPI_Group target_group;
            int ntargets = ...;
            int target_ranks[ntargets];
            MPI_Group_incl( world_group, ntargets, target_ranks, &target_group );
            MPI_Win_start( target_group, 0, win );
            // series of puts, gets, accumulate, etc.
            MPI_Put ( ... );
            MPI_Get ( ... );
            //
            MPI_Win_complete( win ); }
else {
            MPI_Group origin_group;
            int norigins = 1;
            int origin_rank = origin;
            MPI_Group_incl( world_group, norigins, &origin_ranks, &origin_group );
            MPI_Win_post( origin_group, 0, win );
            MPI_Win_wait( win );  }
```

a group with multiple targets

RMA ops

a group with one origin

Opens and closes exposure

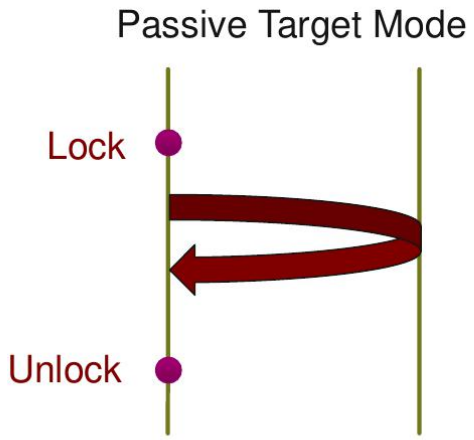
A snippet with one task being the origin of RMA ops towards multiple targets.

This can be easily generalized to a case in which many tasks are both origin and targets

Locks

Passive mode
the target does **not participate** in operations

One-sided *asynchronous* communication



from "Advanced MPI programming", SC24

Locks

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

MPI_Win_unlock (int rank, MPI_Win win )

MPI_Win_flush/flush_local (int rank, MPI_Win win )
```

Where `lock_type` can be:

`MPI_LOCK_SHARED`	concurrent access to the same target is allowed; you are in charge of ensuring that no race conditions happen.
`MPI_LOCK_EXCLUSIVE`	concurrent access to the same target is ***not*** allowed

NOTE: "lock" has been an unfortunate choice for the name. That is not a mutual exclusion, it is more similar to just start/stop of RMA operations

`Flush`: complete operations on remote target process; data will be then available to the target task, or to other tasks
`Flush_local`: locally complete operations to the target process

Locks

A snippet to exemplify how to use the lock

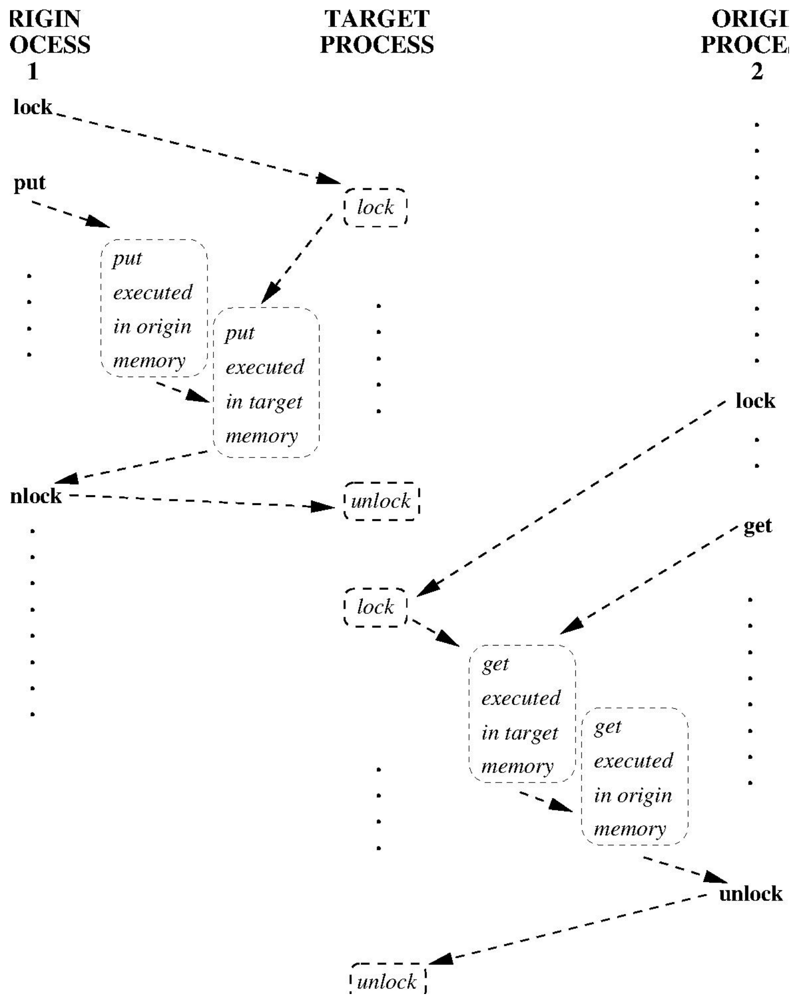
```
MPI_Win win;

if (rank == 0) {
    // Rank 0 will perform tyhe put, it does not need a window
    MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &win);
    // “locks” process 1
    MPI_Win_lock(MPI_LOCK_SHARED,1,0,win);
    // returns when succeded
    MPI_Put(...,1, ..., win);
    // returns when put has succedeed
    MPI_Win_unlock(1,win);
    MPI_Win_free(&win); }
else {
    // Rank 1 is the target, we need a window
    MPI_Win_create(..., ..., ... , MPI_INFO_NULL, MPI_COMM_WORLD, &win);
    //
    MPI_Win_free(&win);
}
```

Locks *all*

```
MPI_Win_lock_all (int lock_type, int assert, MPI_Win win )

MPI_Win_unlock_all ( MPI_Win win )

MPI_Win_flush_all/flush_local_all (int rank, MPI_Win win )
```

`Lock_all` starts an `MPI_LOCK_SHARED` on the group of tasks that participate in the window, wheres **`unlock_all`** ends it.
The routine is *not* collective

Locks *all*


RIGIN
OCESS
1
TARGET
PROCESS
ORIGI
PROCE
2
lock
lock
put
put executed in origin memory
put executed in target memory
nlock
unlock
lock
lock
get
get executed in target memory
get executed in origin memory
unlock
unlock

Key concepts in RMA

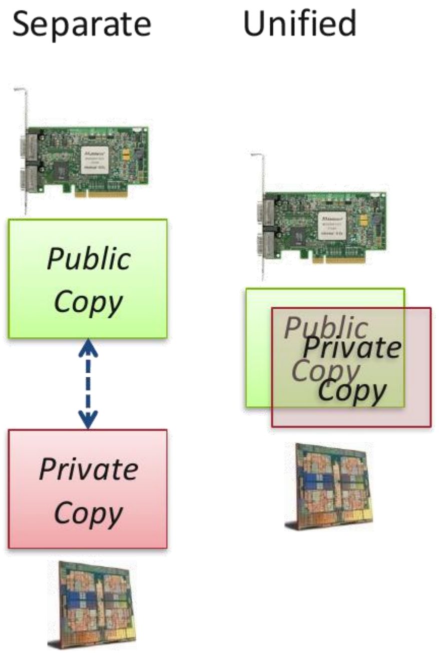
Memory model

Memory model in RMA

From MPI-3, two memory models are provided for RMA

- **Separate**, inherited from MPI-2
  designed to work on systems that do not provide cache coherence at hardware level.
  The "public" copy and the "private" memory are separated and MPI provides software coherence

- **`Unified`**
  there is only 1 copy of the window
  coherence between "public" and "private" provided at system level: yes, there still are a "private" (i.e. the cache) and a "public" (i.e. the RAM) copies that "*eventually*" will be synchronized at system level.



Memory model in RMA

If you want to check which model the MPI library has opted for your windows:

```
int *model, flag;
MPI_Win_get_attr(win, MPI_WIN_MODEL, &model, &flag);
int is_unified = (*model == MPI_WIN_UNIFIED);
```

For practical purposes, due to the nuances of the “eventually will be synchronized” in the unifed model, act as you want to ensure the synchronization whenever needed.

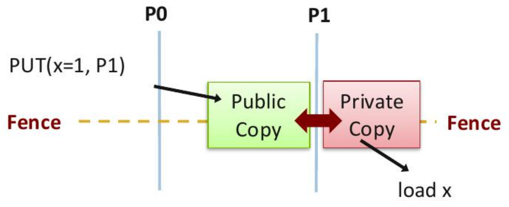
[Semantics and correctness @ mpi-forum]
[Examples @ mpi-forum]

Synchronization

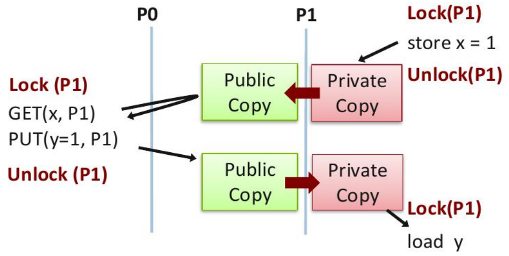
- RMA ops access the *public (RAM)* copy of the window
- local load/stores access the *private (cache)* copy of the window

RMA **`synch. calls`** imply a memory synchronization

- **`fence`** synchronizes public and private copies

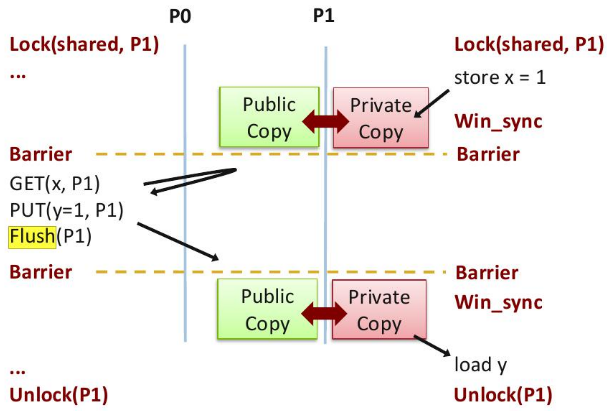


- **`lock/lock_all`** updates in private copy becomes visible in public copy
- **`unlock/unlock_all`** updates in private copy becomes visible in public copy



Synchronization

- `MPI_Win_sync` synchronizes public (reads: RAM) and private (reads: cache) copies



- An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to `MPI_WIN_POST`, `MPI_WIN_FENCE`, `MPI_WIN_UNLOCK`, `MPI_WIN_UNLOCK_ALL`, or `MPI_WIN_SYNC` is executed on that window *by the window owner*.
In the RMA unified memory model, an update of a location in a private window in process memory becomes visible without additional RMA calls.

- An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to `MPI_WIN_WAIT`, `MPI_WIN_FENCE`, `MPI_WIN_LOCK`, `MPI_WIN_LOCK_ALL`, or `MPI_WIN_SYNC` is executed on that window *by the window owner*.
In the RMA unified memory model, an update by a put or accumulate call to a public window copy eventually becomes visible in the private copy in process memory without additional RMA calls.

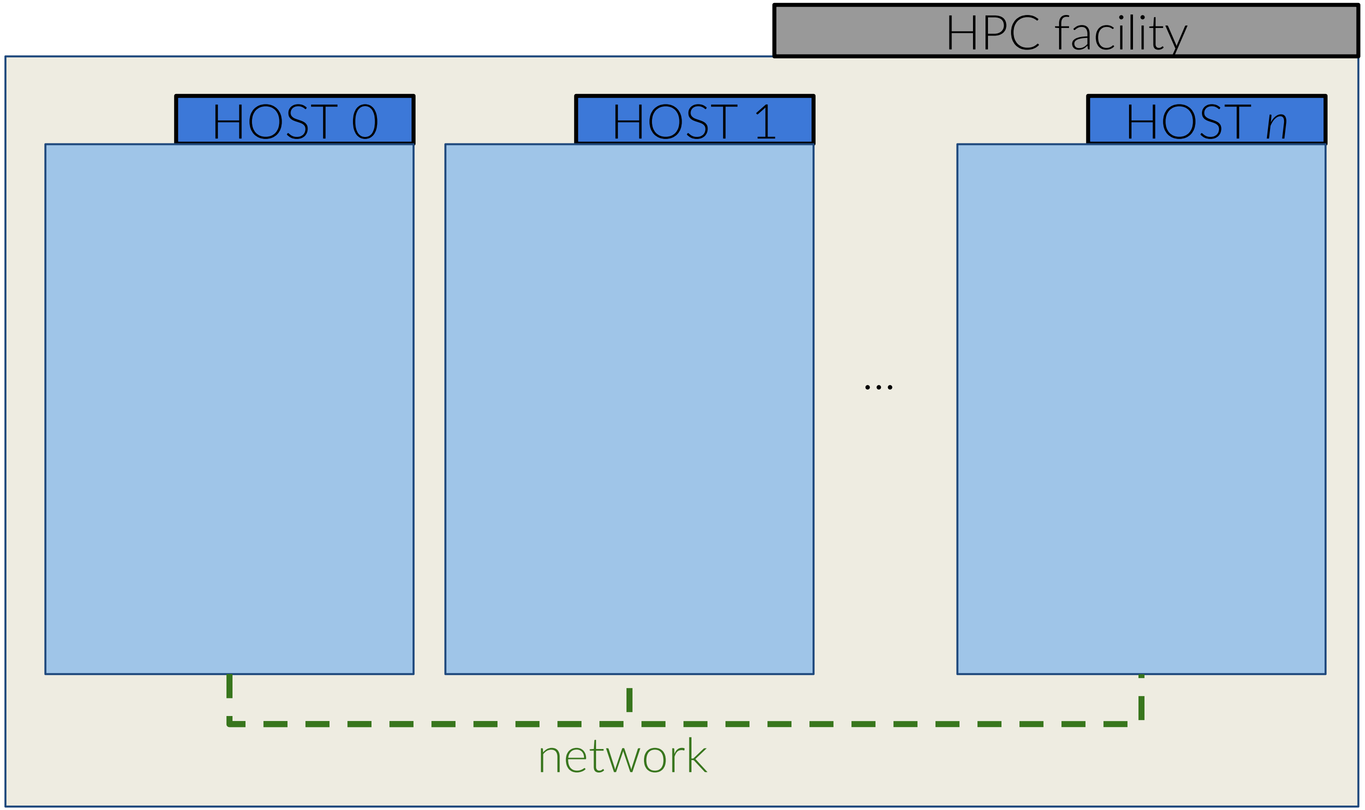
Focus: Shared Memory

Shared-Memory access

Shared-memory

As you know, an HPC machine is made of several *nodes* that are connected by a top-level network (with, possibile, a hierarchical topology)

We'll call these *nodes* "hosts"

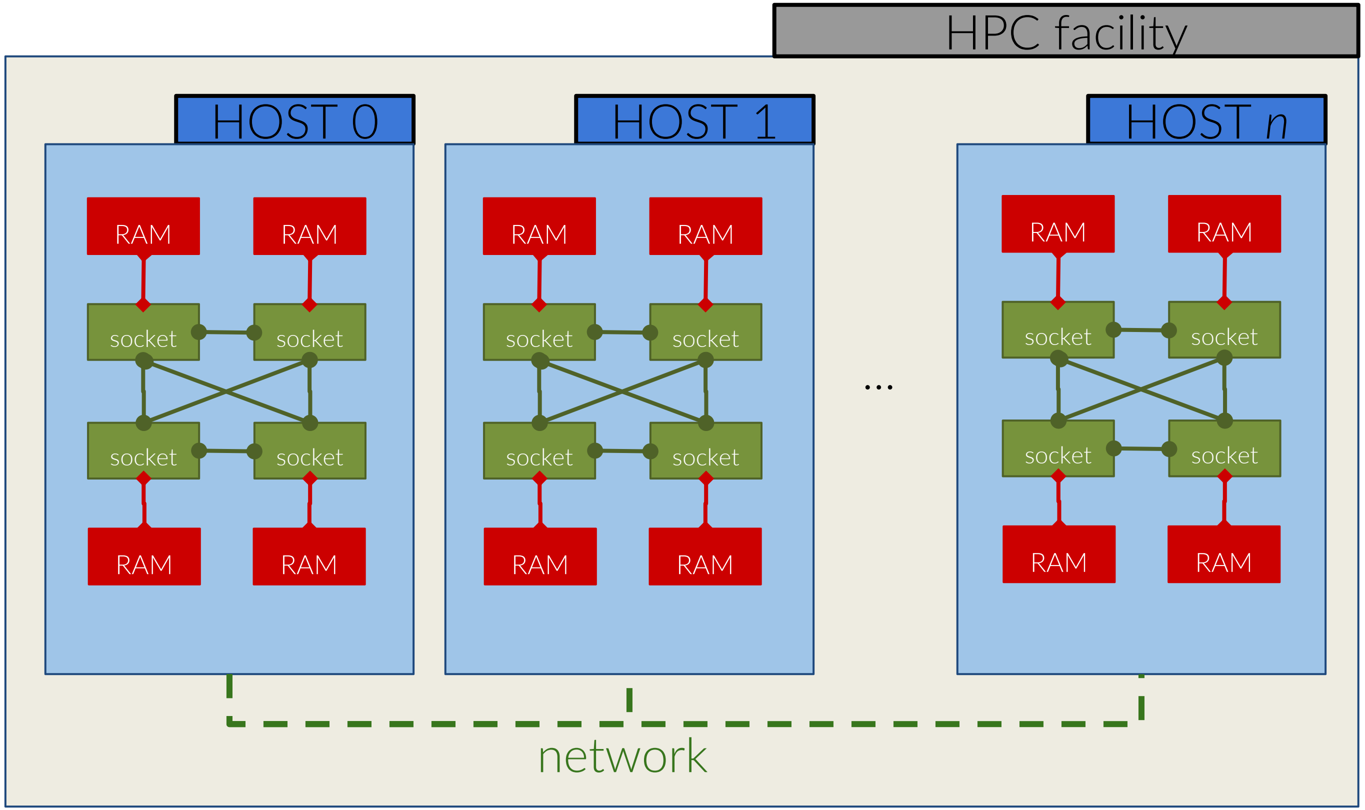


Shared-memory

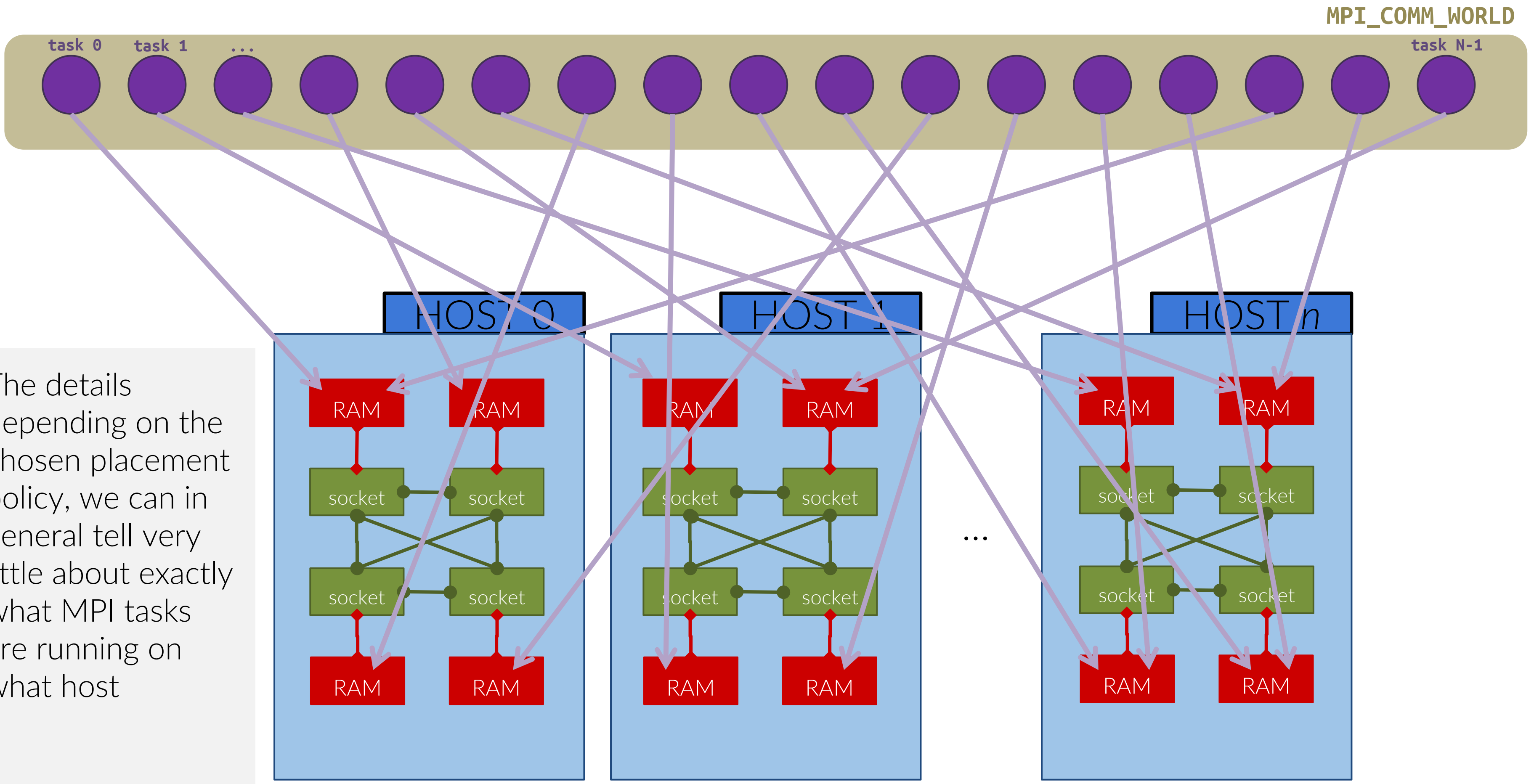
Currently every host contains a NUMA region which consists in a number of sockets (usually 2 nowadays, sometimes 4) each of which physically connected to RAM banks.

In turn, the sockets are inter-connected so that a process running on a given core can access a memory bank not physically attached to its socket.

The detailed topology may be significantly more complicated, but the main message is that a node consists of a NUMA region.
In any case, it is possible to generalize what will be discussed in the next slides to more complicated topologies or hierachies.



Shared-memory



The details depending on the chosen placement policy, we can in general tell very little about exactly what MPI tasks are running on what host

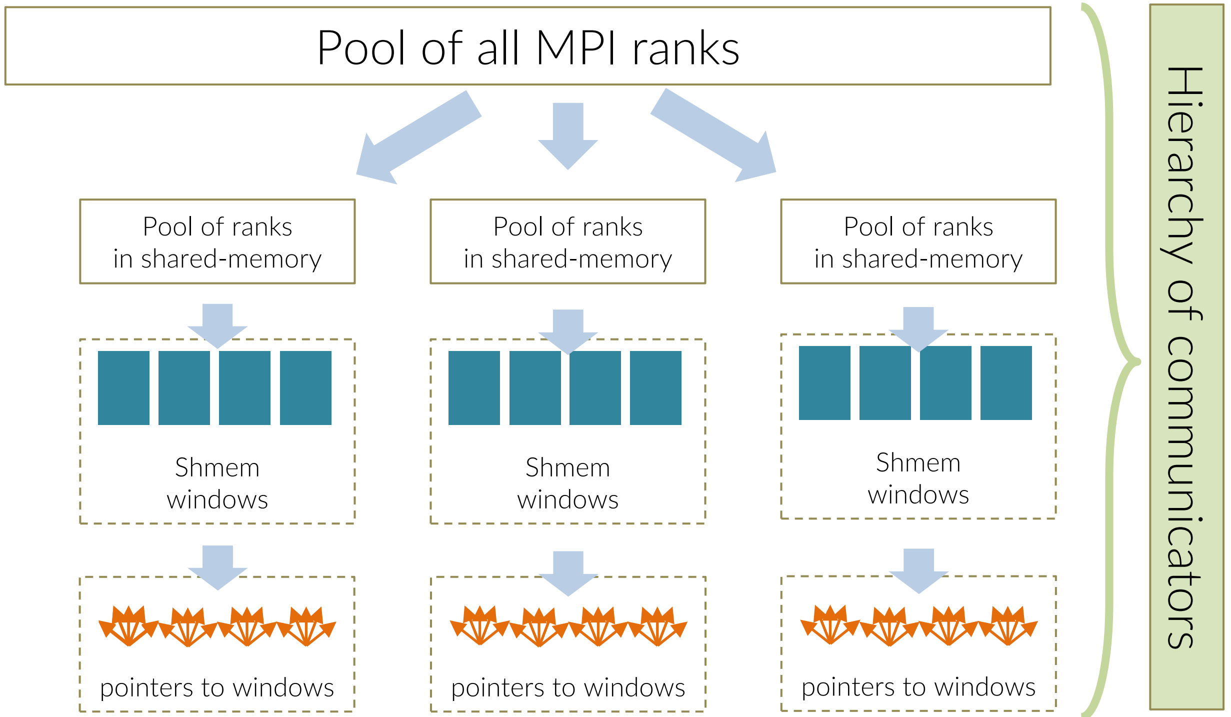
Shared-memory

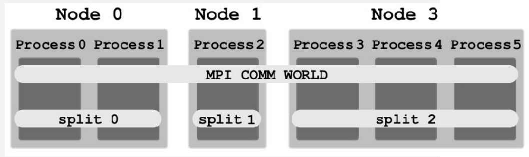
The rationale of the next slides is then:

1) how to re-group MPI tasks that belongs to the same NUMA regions

2) how to build shared-memory windows for every group of tasks

3) how to get the pointers to access the windows





from "Using advanced MPI"

Shared-memory

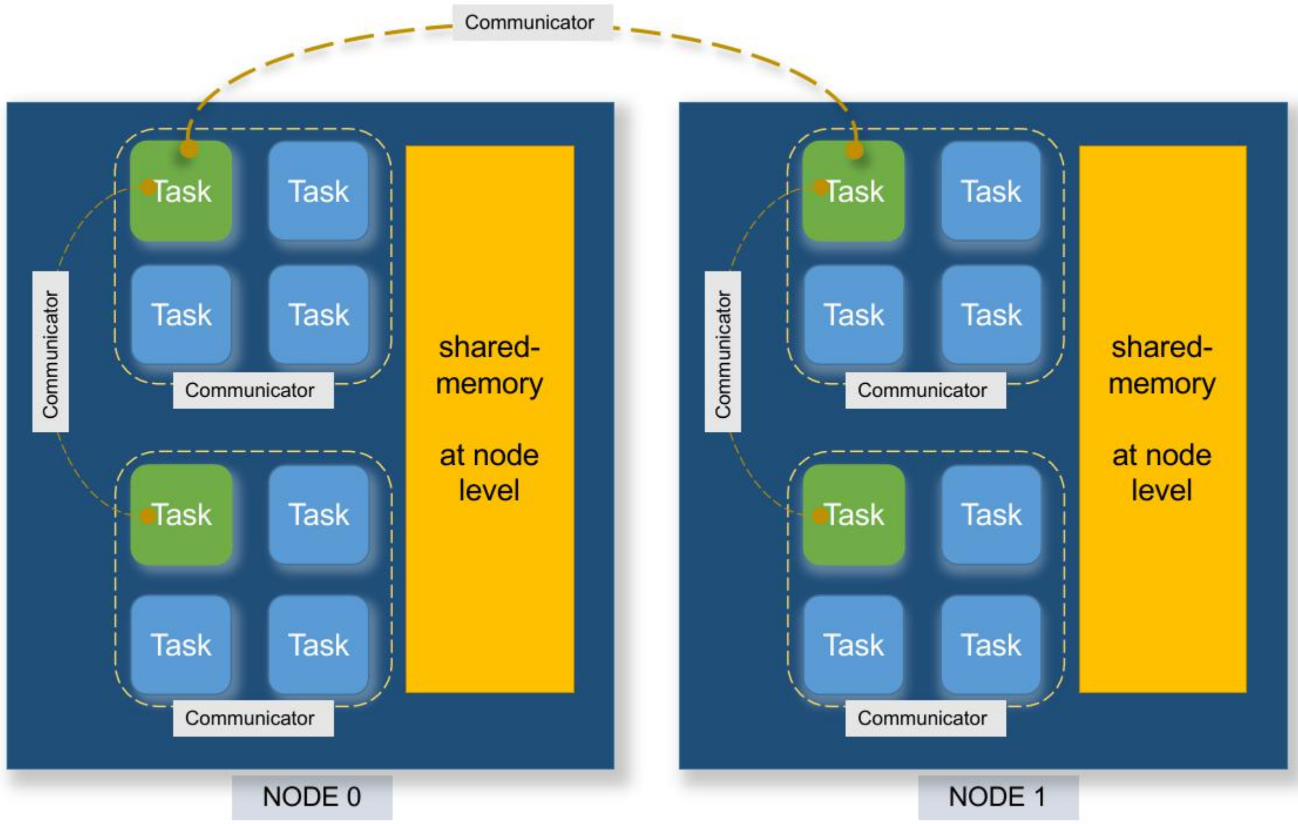
The rationale of the next slides is then:

1) how to re-group MPI tasks that belongs to the same NUMA regions

2) how to build shread-memory windows for every group of tasks

3) how to get the pointers to access the windows

4) how to build a hierarchy of communicators



1) Getting shared-memory pools

MPI provides a method to know about that and to distinguish the tasks depending some of their charateristics

A general routine to split a communicator into sub-communicators

```
MPI_Comm_split ( MPI_Comm comm, // the communicator to be splitted
          int color,            // discriminate tasks
          int key,              // hint for new ranks
        MPI_Comm *newcomm       // the new communicator )
```

A specific routine to split a communicator into sub-communicators following the hardware hierarchy

```
MPI_Comm_split_type ( MPI_Comm comm, // the communicator to be splitted
          int split_type,            // the splitting property
          int key,                   // int for the new ranks
          MPI_Info info,             // helper for the splitting property
        MPI_Comm *newcomm             // the new communicator )
```

reference

1) Getting shared-memory pools

MPI provides a method to know about that and to distinguish the tasks depending some of their charateristics

Let's start with the specific `Comm_split_type`

```
MPI_Comm_split_type ( MPI_Comm comm, int split_type,int key,
          MPI_Info info, MPI_Comm *newcomm )
```

`split_type` can have the values:

`MPI_COMM_TYPE_SHARED`

`MPI_COMM_TYPE_HW_GUIDED`

`MPI_COMM_TYPE_HW_UNGUIDED`

1) Getting shared-memory pools

`MPI_COMM_TYPE_SHARED`

« all MPI processes in the group of newcomm are part of the same shared memory domain and can create a shared memory segment (e.g., with a successful call to MPI_WIN_ALLOCATE_SHARED). This segment can subsequently be used for load/store accesses by all MPI processes in newcomm.»

`MPI_COMM_TYPE_HW_GUIDED`

« this value specifies that the communicator comm is split according to a hardware resource type (for example a computing core or an L3 cache) specified by the **`mpi_hw_resource_type`** info key. Each output communicator newcomm corresponds to a single instance of the specified hardware resource type. The MPI processes in the group associated with the output communicator newcomm utilize that specific hardware resource type instance, and no other instance of the same hardware resource type.

The value **`mpi_shared_memory`** is reserved and its use is equivalent to using **`MPI_COMM_TYPE_SHARED`** for the split_type parameter

the info key mpi_hw_resource_type is reserved and its associated value is an implementation-defined string designating the type of the requested hardware resource (e.g., ``NUMANode'', ``Package'' or ``L3Cache'') »

1) Getting shared-memory pools

`MPI_COMM_TYPE_HW_UNGUIDED`

« the group of MPI processes associated with newcomm must be a strict subset of the group associated with comm and each newcomm corresponds to a single instance of a hardware resource type (for example a computing core or an L3 cache).
All MPI processes in the group associated with comm that utilize that specific hardware resource type instance---and no other instance of the same hardware resource type---are included in the group of newcomm. »

1) Getting shared-memory pools

MPI_COMM_TYPE_HW_GUIDED

```
    MPI_Info info;
    MPI_Comm hwcomm;
    int      rank;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Info_create(&info);
    MPI_Info_set(info, "mpi_hw_resource_type", "NUMANode");
    MPI_Comm_split_type(MPI_COMM_WORLD,
                        MPI_COMM_TYPE_RESOURCE_GUIDED,
                        rank, info, &hwcomm);
```

MPI_COMM_TYPE_HW_UNGUIDED

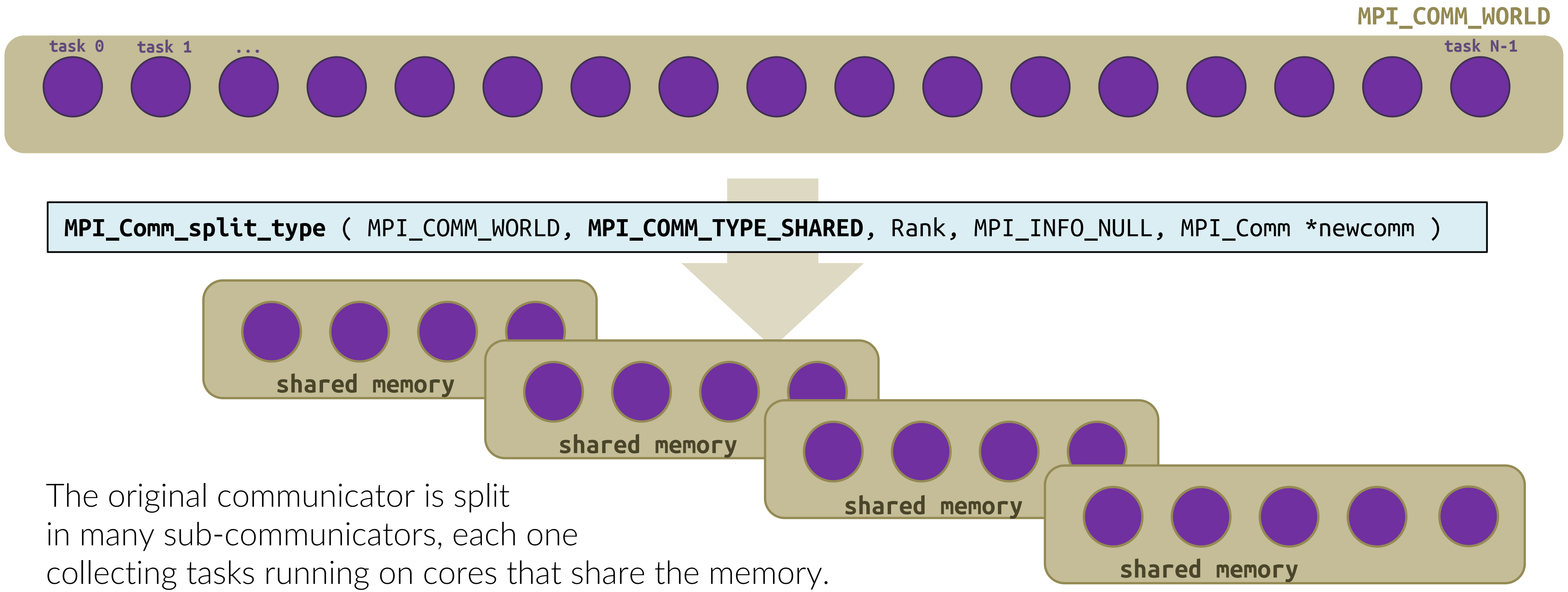
```
#define MAX_NUM_LEVELS 32

MPI_Comm hwcomm[MAX_NUM_LEVELS];
int      rank, level_num = 0;

hwcomm[level_num] = MPI_COMM_WORLD;

while((hwcomm[level_num] != MPI_COMM_NULL)
      && (level_num < MAX_NUM_LEVELS-1)){
  MPI_Comm_rank(hwcomm[level_num],&rank);
  MPI_Comm_split_type(hwcomm[level_num],
                      MPI_COMM_TYPE_HW_UNGUIDED,
                      rank,
                      MPI_INFO_NULL,
                      &hwcomm[level_num+1]);
  level_num++;
}
```

1) Getting shared-memory pools

The `MPI_COMM_TYPE_HW_GUIDED` would be very handy, but it is still not mature (real implementations are currently introducing it ; the exact key values are implementation-dependent).

For our purposes, let's explore the combination of `MPI_Comm_split_type()` with `split_type=MPI_COMM_TYPE_SHARED,` and the appropriate usage of `MPI_Comm_split()`

1) Getting shared-memory pools

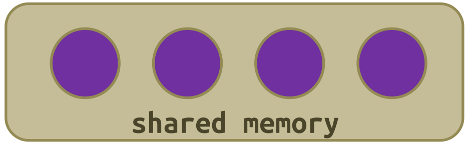


```
MPI_Comm_split_type ( MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, Rank, MPI_INFO_NULL, MPI_Comm *newcomm )
```

The original communicator is split in many sub-communicators, each one collecting tasks running on cores that share the memory.

Typically that is equivalent to say that they run on the same node.
Adivice: pin the tasks to avoid migration, wich would invalid the decomposition got from a previous call to `MPI_Comm_split_type`

2) Getting shared-memory windows



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

Create a remotely accessible memory region in an RMA window
Data exposed in a window can be accessed with either RMA ops or load/store

```
MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
MPI_Win_allocate_shared(comm, ..., &win);
MPI_Win_lockall(win);
   // local store on memory
MPI_Win_sync(win);
   // use shared memory
MPI_Win_unlock_all(win);
MPI_Win_free(&win);
```

Arguments:

- `size`	size of local data in bytes (nonnegative integer)
- `disp_unit`	local unit size for displacements, in bytes (positive integer)
- `info`	info argument (handle)
- `comm`	communicator (handle)
- `baseptr`	pointer to exposed local data
- `win`	window (handle)

2) Getting shared-memory windows

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

PLACEMENT

- As for the other window-creation routines, memory allocation is not mandatorily uniform
  allocated size can differ rom task to task

- Here is no specification about “where” the memory is physically placed
  reasonably, an MPI implementation will opt to maximize the affinity

- By default the allocation is *contiguous*
  that, however, does not only refer to the virtual address space: it would mean, generally to allocate the memory to the same physical bank since many systems do not allow address remapping at arbitraty sizes.
  Specifiying a non-contiguous allocation allows the MPI implementation to enhance the affinity for every task

2) Getting shared-memory windows

CAVEATS

- use the **`restrict`** attribute for the base pointer

  to signal that the allocated buffer does not contain pointers to other memory regions, which give to the compiler more freedom in optimizing the code

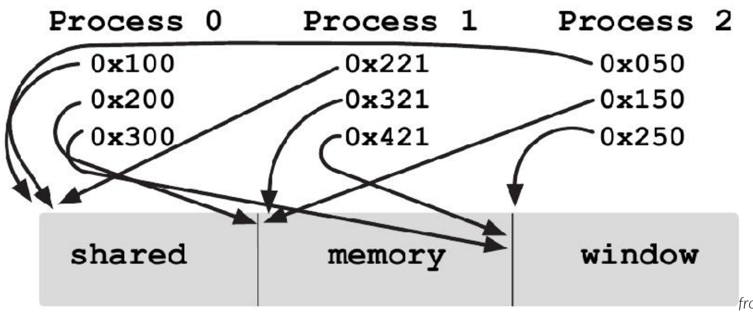
```
double restrict *ptr;
MPI_Win win;
...
MPI_Win_allocate_shared( size, ..., &mem, &win);
```

- enable **hugepage** if possible (explore that with your sys admin...)

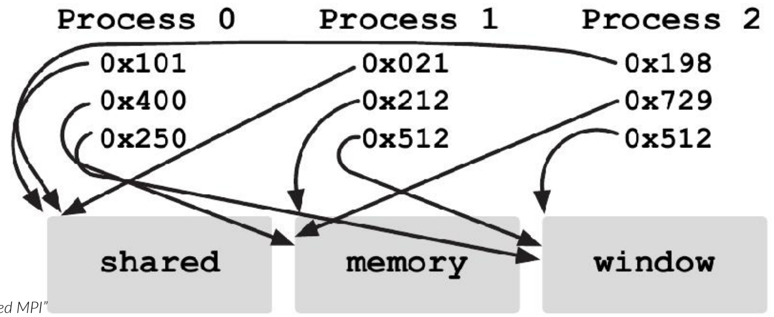
  in fact, OS uses large pages for in-process memory allocation, treated via anonymous **`mmap`**, whereas multi-process allocation is file-backed and assigned with regularly-sized pages (4KB)

3) Getting shared-memory pointers

CONTIGUITY



default contiguous allocation



from "Using advanced MPI"

non-contiguous allocation

```
MPI_Win_shared_query ( MPI_Win win, int rank, MPI_Aint *size, int *disp_unit, void *baseptr )
```

Allows to get the pointer for direct load/store from/to the window of `rank` (plus the size and the disp. units)
When called with MPI_PROC_NULL returns

- « the address of the first byte inthe shared window, regardless of which process allocated it », for contiguous allocation

- « the base address of the first process that specified non-zero `size` »

4) Building communicators

Using the `MPI_Comm_split( )` routine we can go further on

- using either `MPI_Get_processor_name()` or the stdc `gethostname()`, we could build a map of all the hosts on which the tasks runs and decompose them by host assigning a `color` to each host; the result is the same than using `MPI_COMM_TYPE_SHARED` (if hosts coincide with largest numa domains, which is usually true)

- if we had a routine like get_socket_id() and get_cpu_id() which return the socket and core on which a given task is running, we would beable to further decompose the tasks into smaller groups and more local numa domains

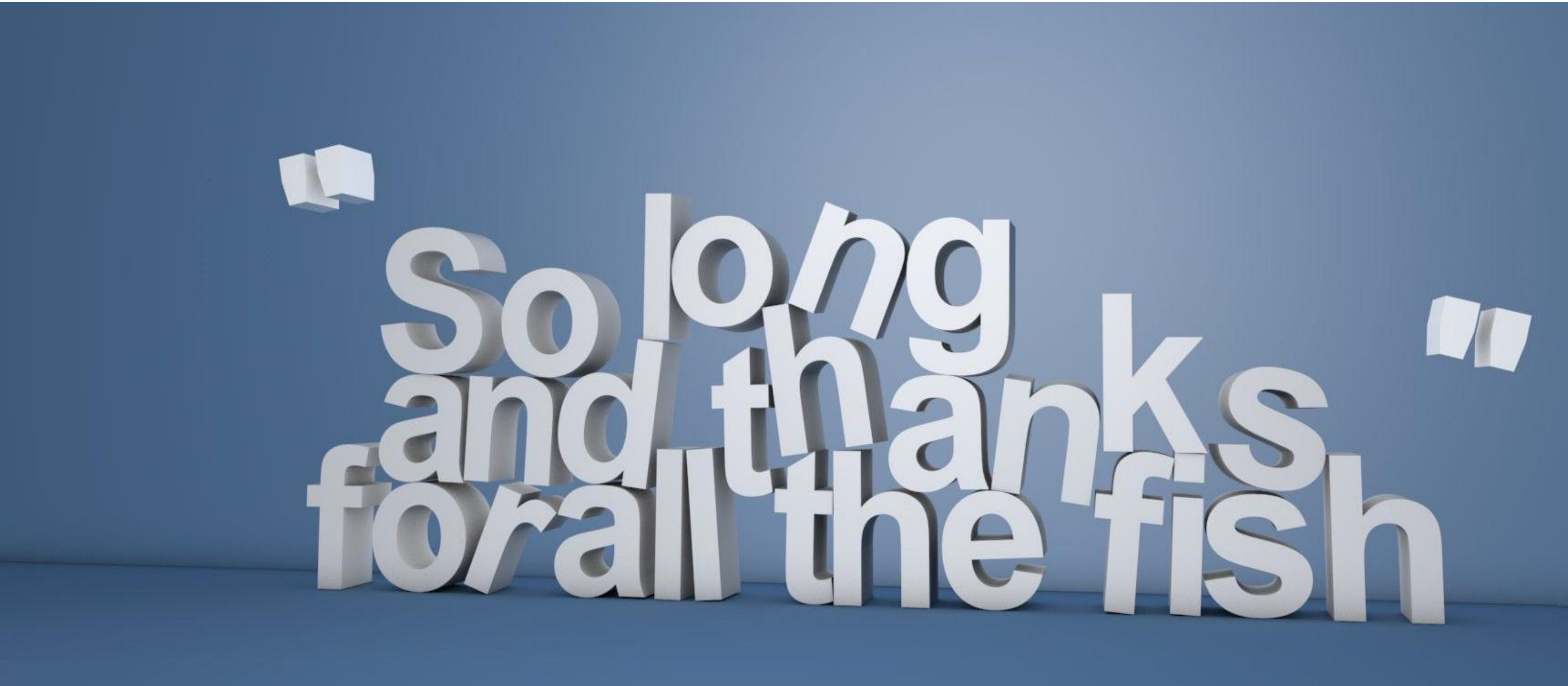
- Using the Groups and Communicators manipulating routines we can build a hierarchy of communicators

► Let's explore all this "in action" in the code `numa.c`

References

- mpi-forum, Index
- Future Learn @ HLRS, one-sided communications
- Future Learn @ HLRS, shared-memory + fast sync
- ENCCS intermediate MPI ENCCS intermediate-mpi - codes
- W. Gropp's course
- Using MPI and Using Advanced MPI

That’s all folks, have fun



That's all Folks!