Advanced MPI
Sh.Mem. & one-sided

SCIENTIFIC & DATA-INTENSIVE COMPUTING

Luca Tornatore - I.N.A.F.

Advanced HPC 2023-2024 @ Università di Trieste

Outline

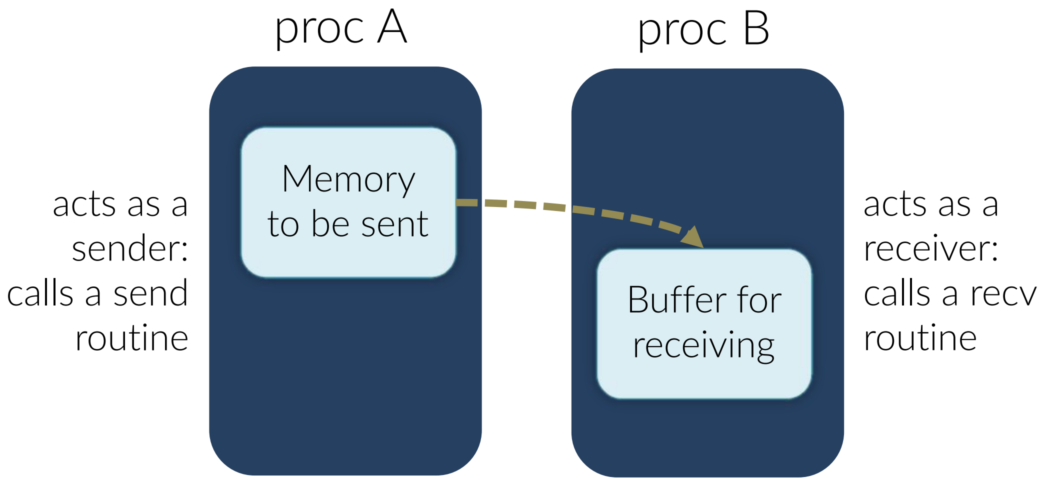
Advanced Usage of
some **MPI** features
on *topology* awareness,
shared memory &
one-sided communications

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

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.

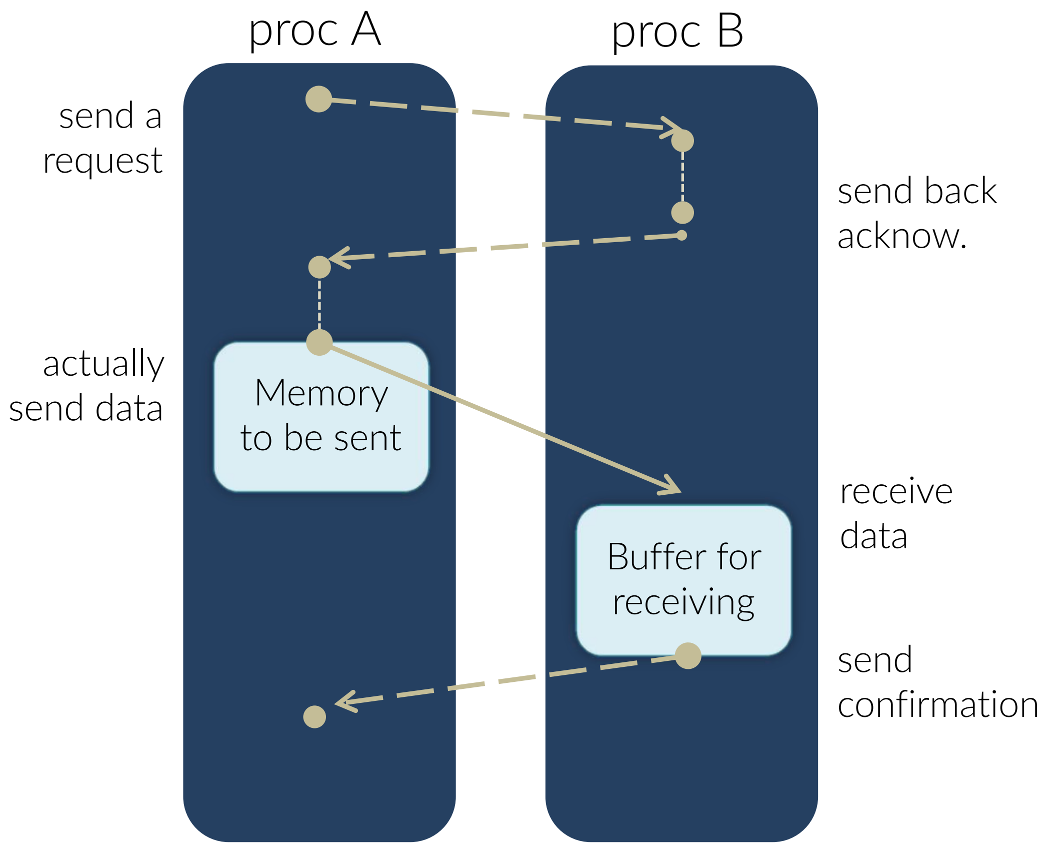
This model is very effective in protecting the memory access and in making clear what memory location will be modified and when that happens


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

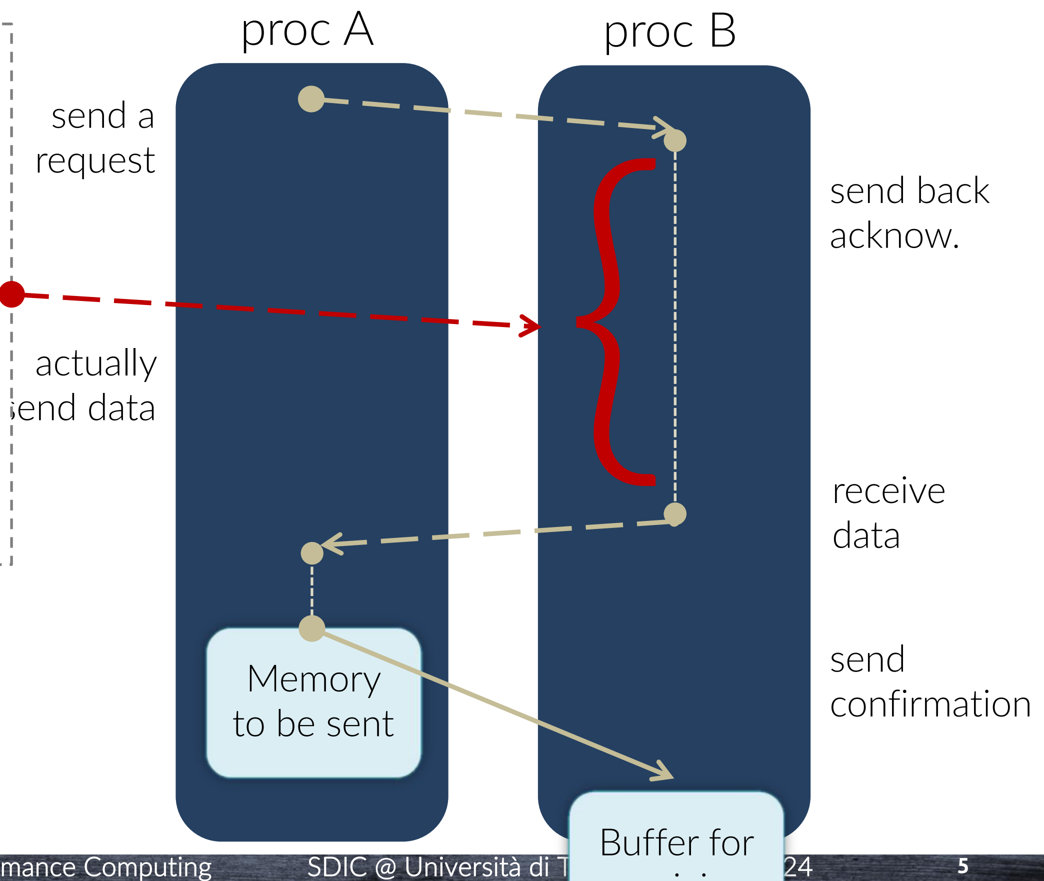


Two-Sided Communications

However, thi
cons.
The processe
collaborate; a
actually send
accepted by

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

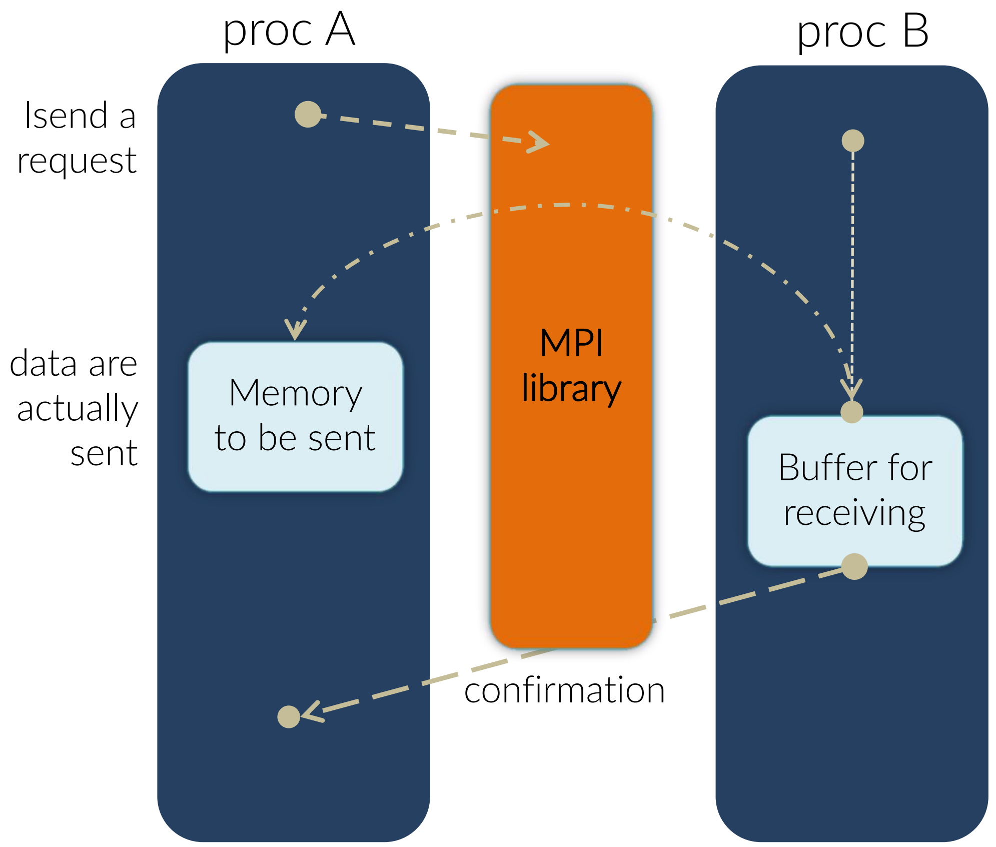
This delay may be large, depending on the status of process B.
Even the Send operation will be delayed as well, because it requires the cooperation of both processes



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.

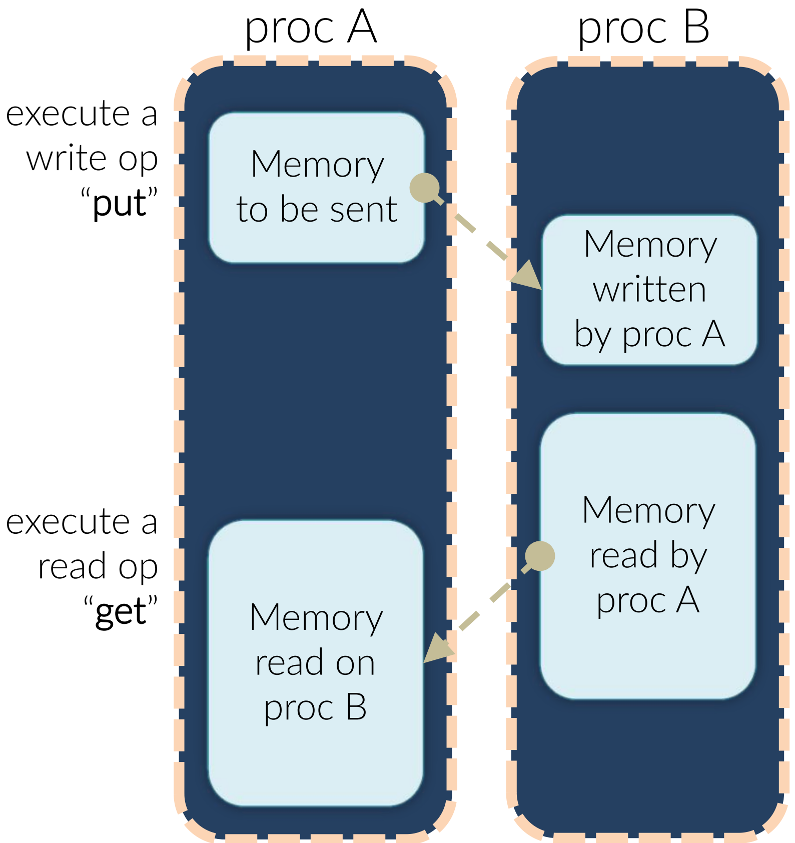
While this makes easier to implement several algorithms, it does not change the fact that the two process need to cooperate.



One-Sided Communications

On the other hand, if *Remote Memory Access (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

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

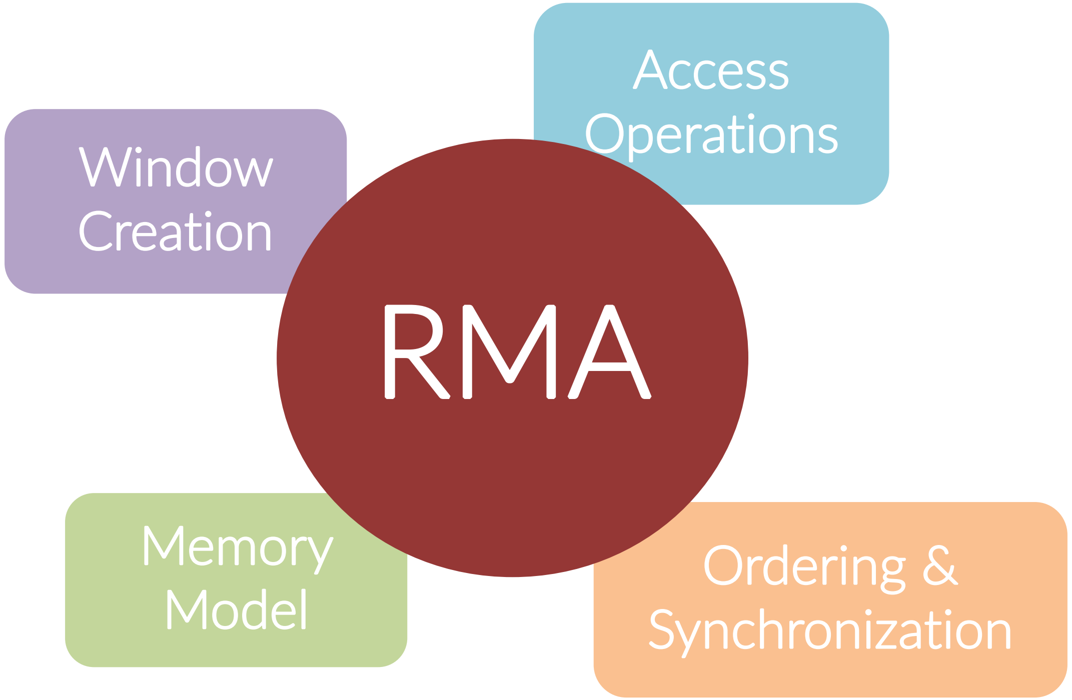
At abstract level, RMA workflow is something like:

```
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();
```

Key concepts in RMA

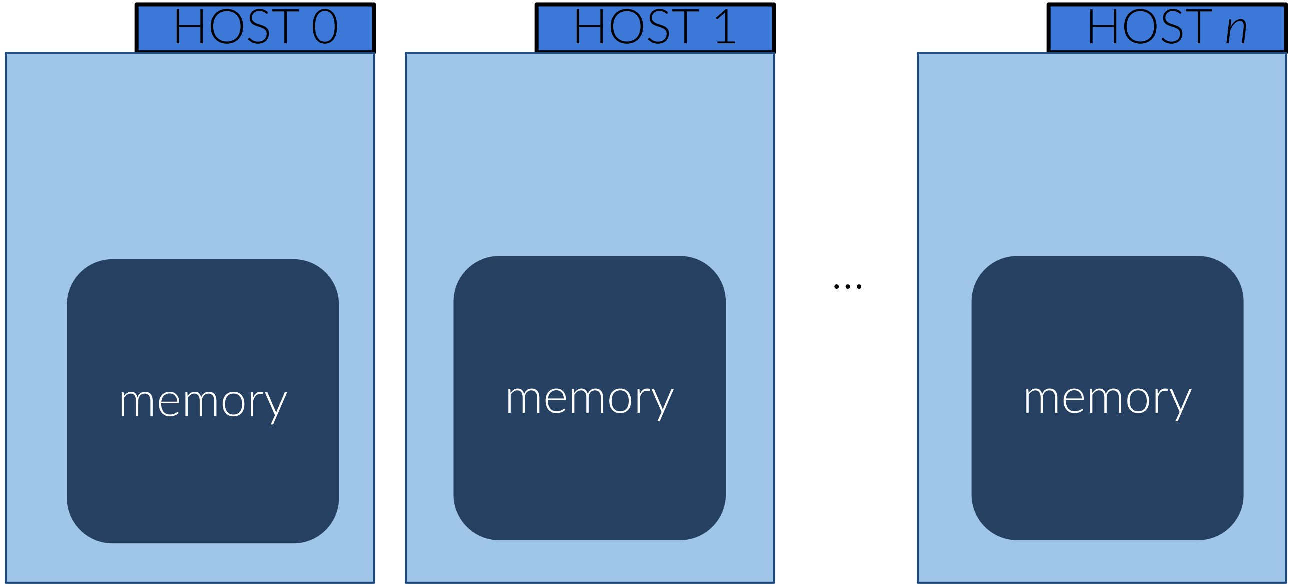


Key concepts in RMA

Creating Memory Windows

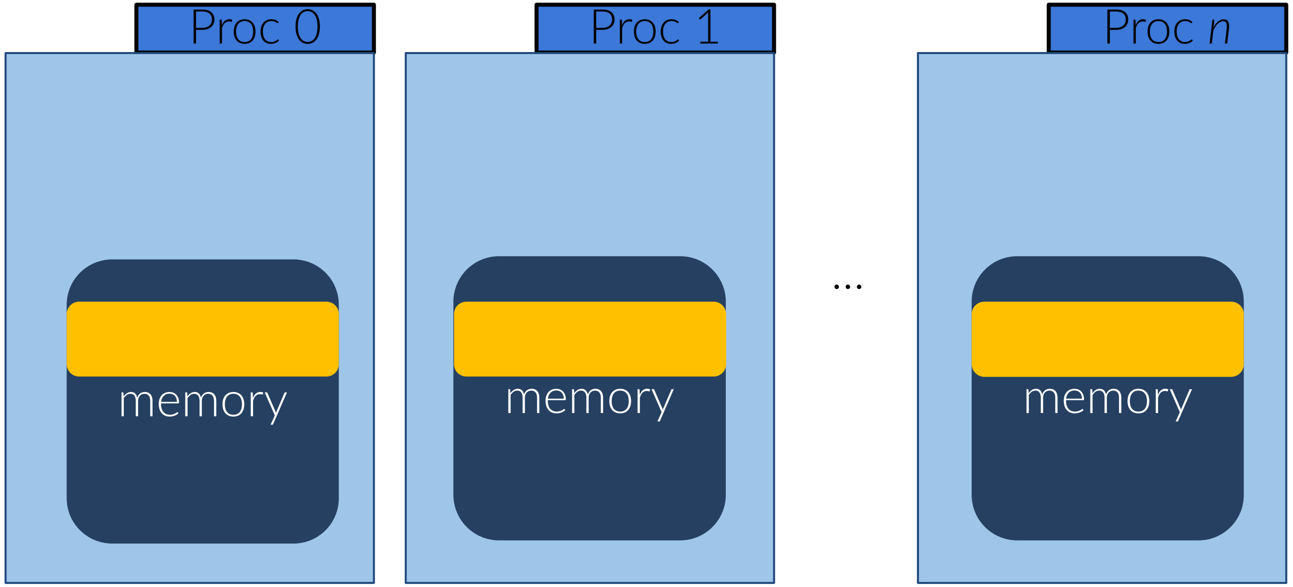
Creating memory windows

- The memory of each process is only locally accessible



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*

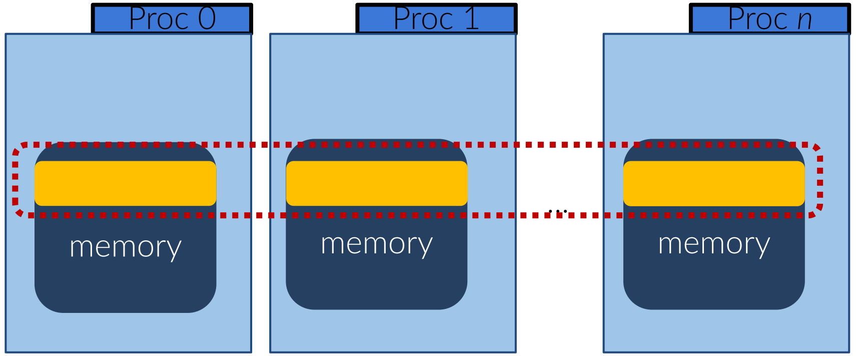


Creating memory windows

- The memory of each process is only locally accessible

- 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 sync 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;

...
if ( Rank == data_owner ) N = amount_of_data;
else N = 0;

MPI_Win_allocate ( N*sizeof(double),
sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD,
&data, &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_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 );
```

Key concepts in RMA

Data movement

Memory movement

You can read, write, modify data atomically

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

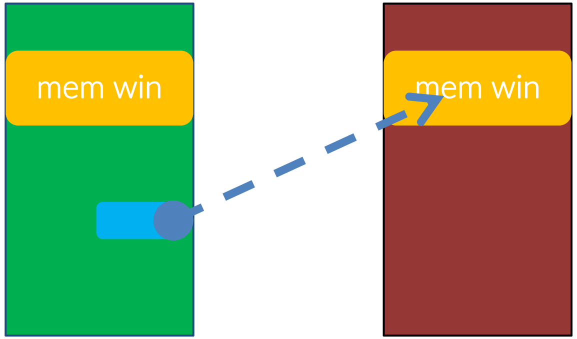
atomic operations

Memory movement: 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 )
```

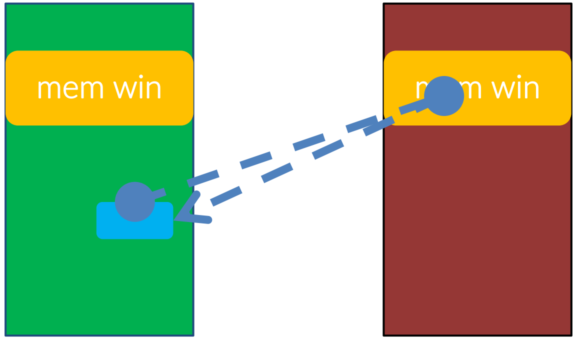
origin (calling proc) target (owner of accessed mem)



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)

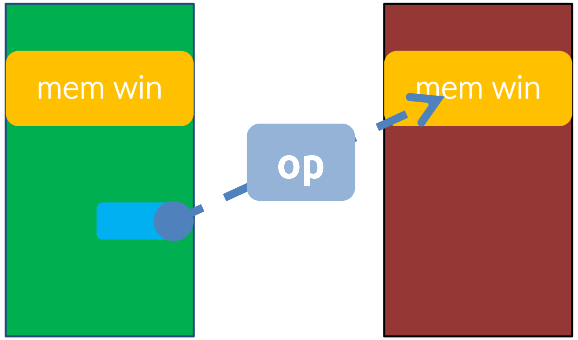


Memory movement: accumulate

```
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: 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?

note: the MPI standard does not enforce the order of exection of put and get operations

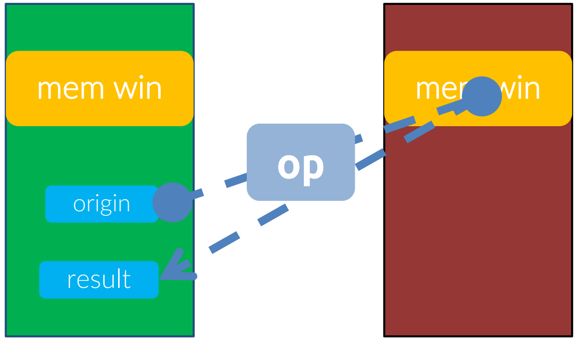
Memory movement: get_accumulate

```
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: 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



Memory movement: `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;

Memory movement

```
MPI_Rput
MPI_Rget
MPI_Raccumulate
MPI_Rget_accumulate
```

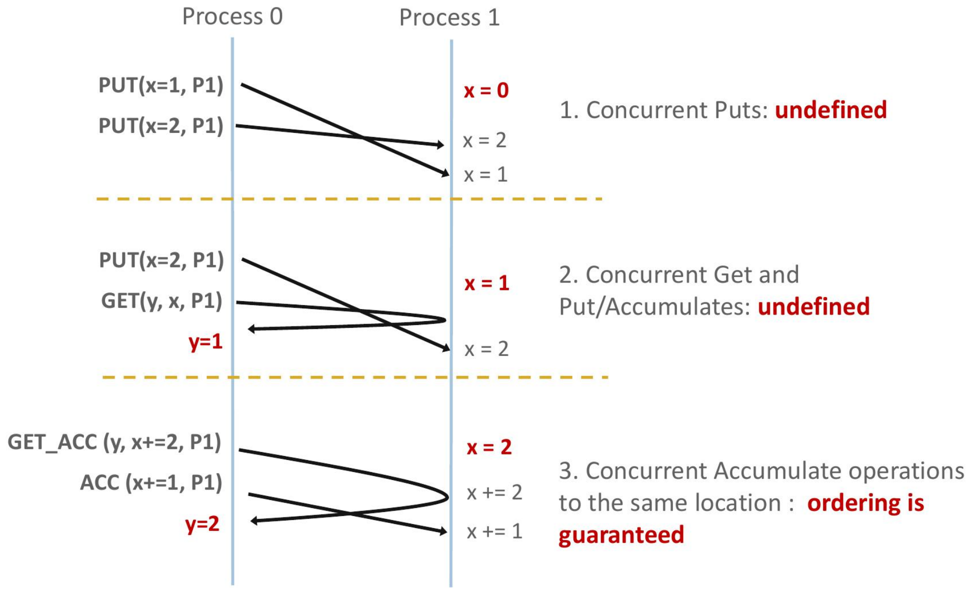
MPI offers also Request-based versions of put, get, accumulate, get_accumulate.

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 & Sync

Ordering and Sync with RMA


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, SC17

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

Ordering and Sync with RMA

All RMA routines are “non-blocking” routines: as such, to ensure the correctness of the operations, they need to be appropriately surrounded by synchronization calls

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

There two types of synchronization:

active	both origin and target have to call sync routines
passive	only the origin calls the sync routine

Ordering and Sync with RMA

active	both origin and target have to call sync routines	
	`MPI_Fence`	A collective call that surround RMA routines to isolate different access types
	`MPI_Win_post, MPI_Win_start, MPI_Win_complete, MPI_Win_wait`	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`	

Fences: semantics

```
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
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
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”)
MPI_MODE_NOSUCCEED	the symmetric than before: no RMA calls *will be made* until the next fence call (“no RMA start”)

PSCW: semantics

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

MPI_Win_post (MPI_Group from_group, int assert, MPI_Win win )
MPI_Win_complete ( 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: assert

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

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

The following asserts are used by MPI_Win_post. The only assert used by MPI_Win_start is `MPI_MODE_NOCHECK`

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 wih this process as a target (basically: no cross-RMAops). The matching `MPI_Win_start` *must* use the same assert.

PSCW

```
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 );

...
MPI_Group win_group, ngb_group;

MPI_Win_get_group( mywin, &win_group);   // MPI_Win_get_group( MPI_Win win, MPI_Group *group)

int ranks_of_my_neighbours[2] = {my_left_ngb_rank, my_right_ngb_rank };

MPI_Group_incl( win_group, 2, ranks_of_my_neighbours, &ngb_group) //MPI_Group_incl(MPI_Group group, int n,
                                                                   // const int ranks[],MPI_Group *newgroup)

// now use ngb_group as a group for MPI_Win_start and _post calls
```

Locks

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

MPI_Win_unlock (int rank, MPI_Win win )
```

Where `lock_type` can be either `MPI_LOCK_SHARED` or `MPI_LOCK_EXCLUSIVE`.

When you specify `MPI_LOCK_SHARED` you must also ensure that no race conditions happens (accumulate ops are always safe).

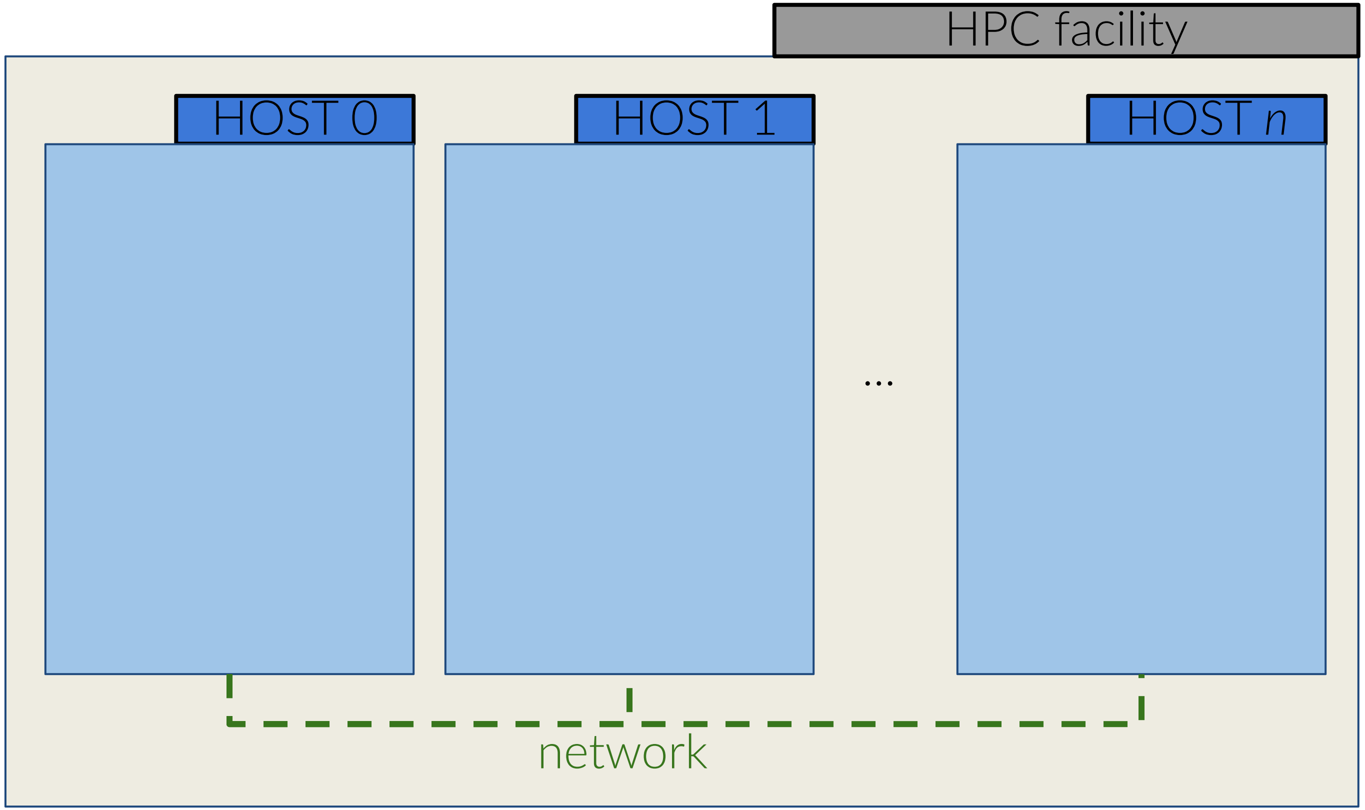
Key concepts in RMA

Shared-memory and the topology

Exploring the topology

An HPC machine is made up by several *nodes* that are connected by a top-level network.

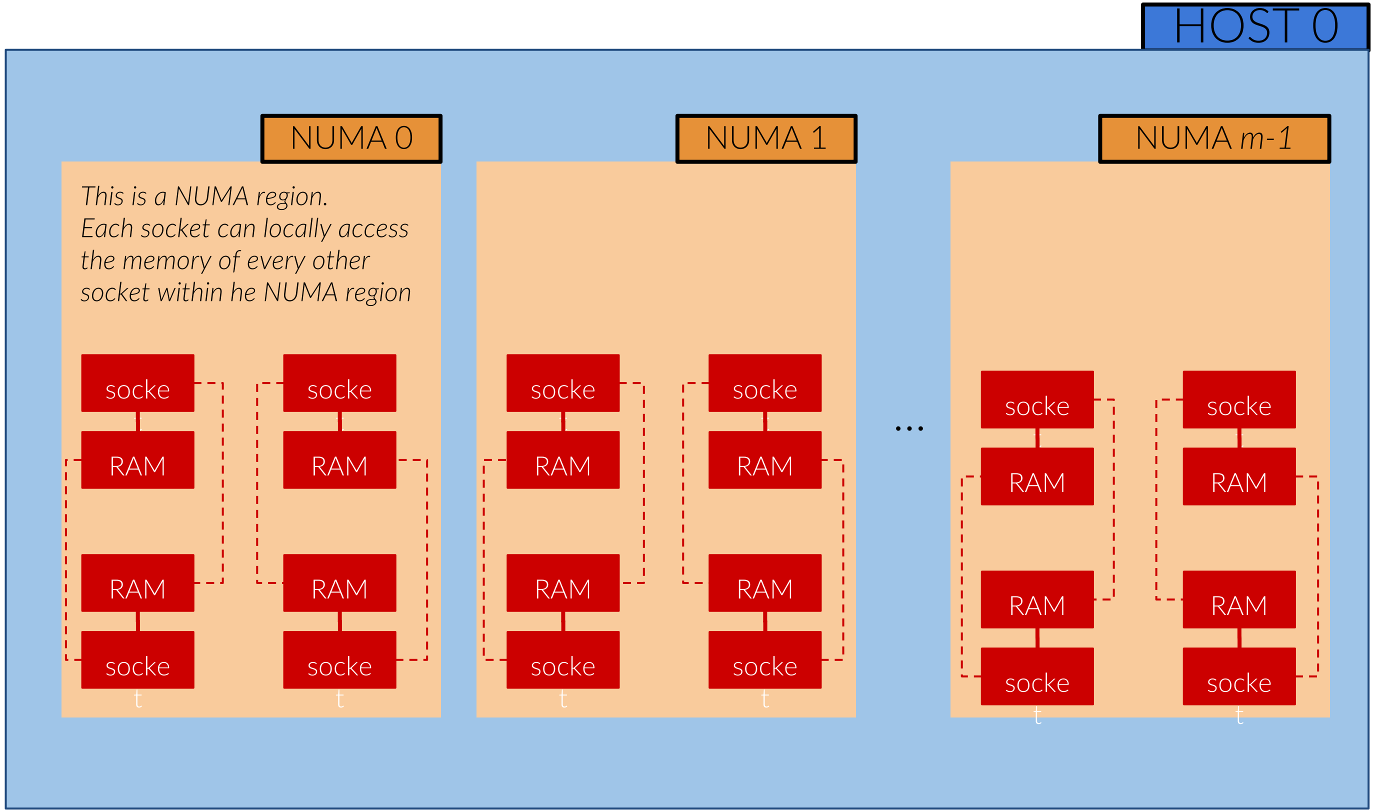
We'll call these *nodes* "hosts" - following the fact that their network name can be obtained by using `hostname`.



Exploring the topology

In principle, we may suppose that inside each host there may be several different clusters of *numa regions* (actually nowadays there is only one numa region, routinely)

Inside each numa region there may be more than one sockets (nowadays usually two), which are able to *physically* access each other's memory

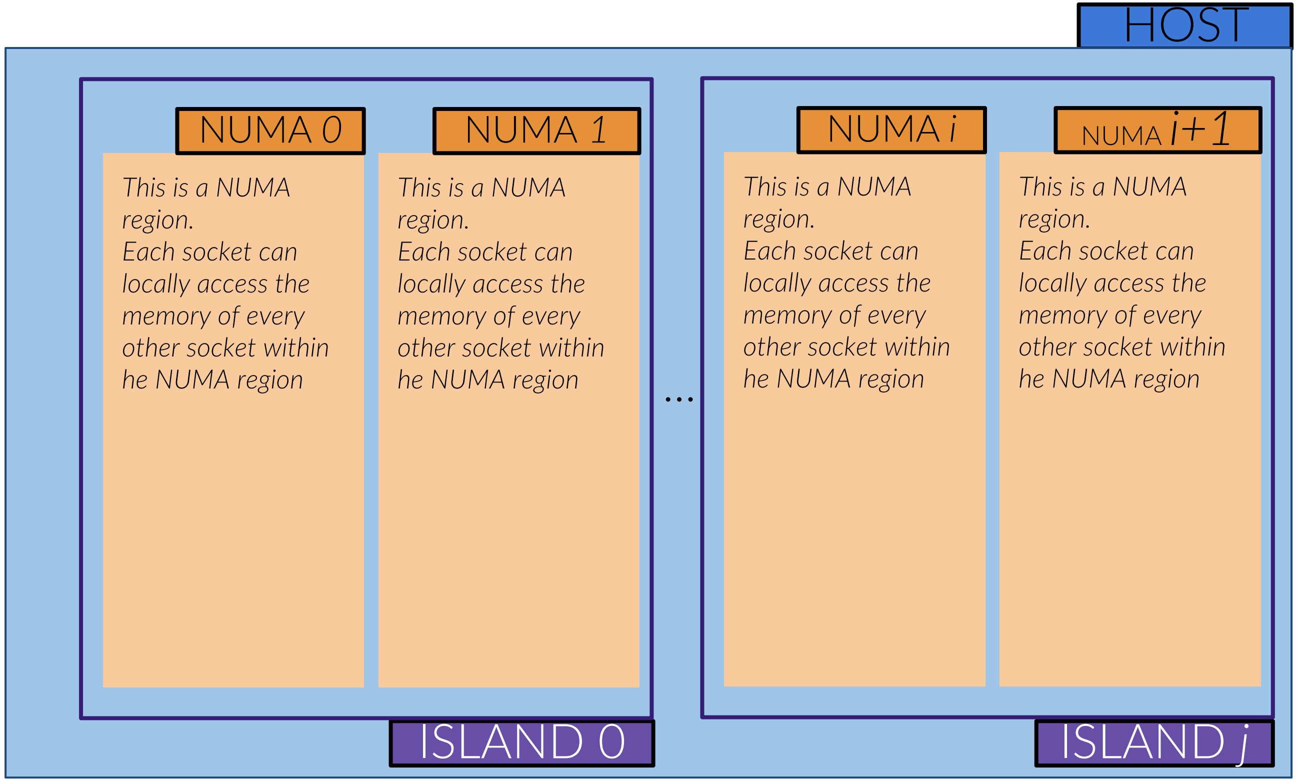

HOST 0
NUMA 0
NUMA 1
NUMA *m*-1
This is a NUMA region. Each socket can locally access the memory of every other socket within he NUMA region
socket
RAM
…

Exploring the topology

Inside each host, the numa regions (it there are more than one) may be clustered by some network/physical connection faster than the network that connects the hosts.

That is just to be very general in our abstraction.

Actually it seems that in the forthcoming future there will be only one numa region, however, the possibility to include in our abstraciton several levels of "contiguity" may always be useful.

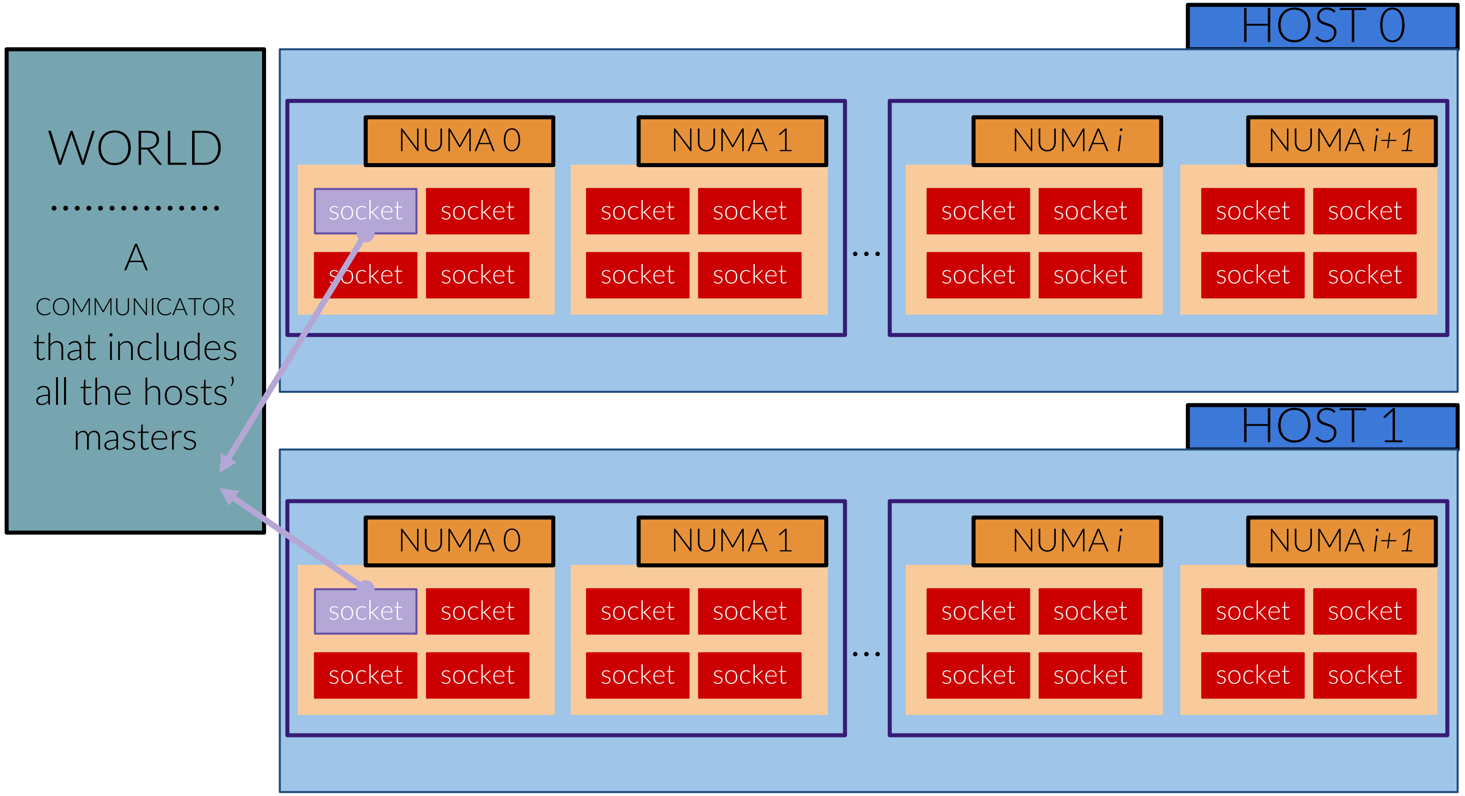

HOST
NUMA 0
This is a NUMA region. Each socket can locally access the memory of every other socket within he NUMA region
NUMA 1
This is a NUMA region. Each socket can locally access the memory of every other socket within he NUMA region
…
NUMA i
This is a NUMA region. Each socket can locally access the memory of every other socket within he NUMA region
NUMA i+1
This is a NUMA region. Each socket can locally access the memory of every other socket within he NUMA region
ISLAND 0
ISLAND j

Exploring the topology

First, it may be useful to build a communicator that includes all the MPI processes that serve as masters of their host.

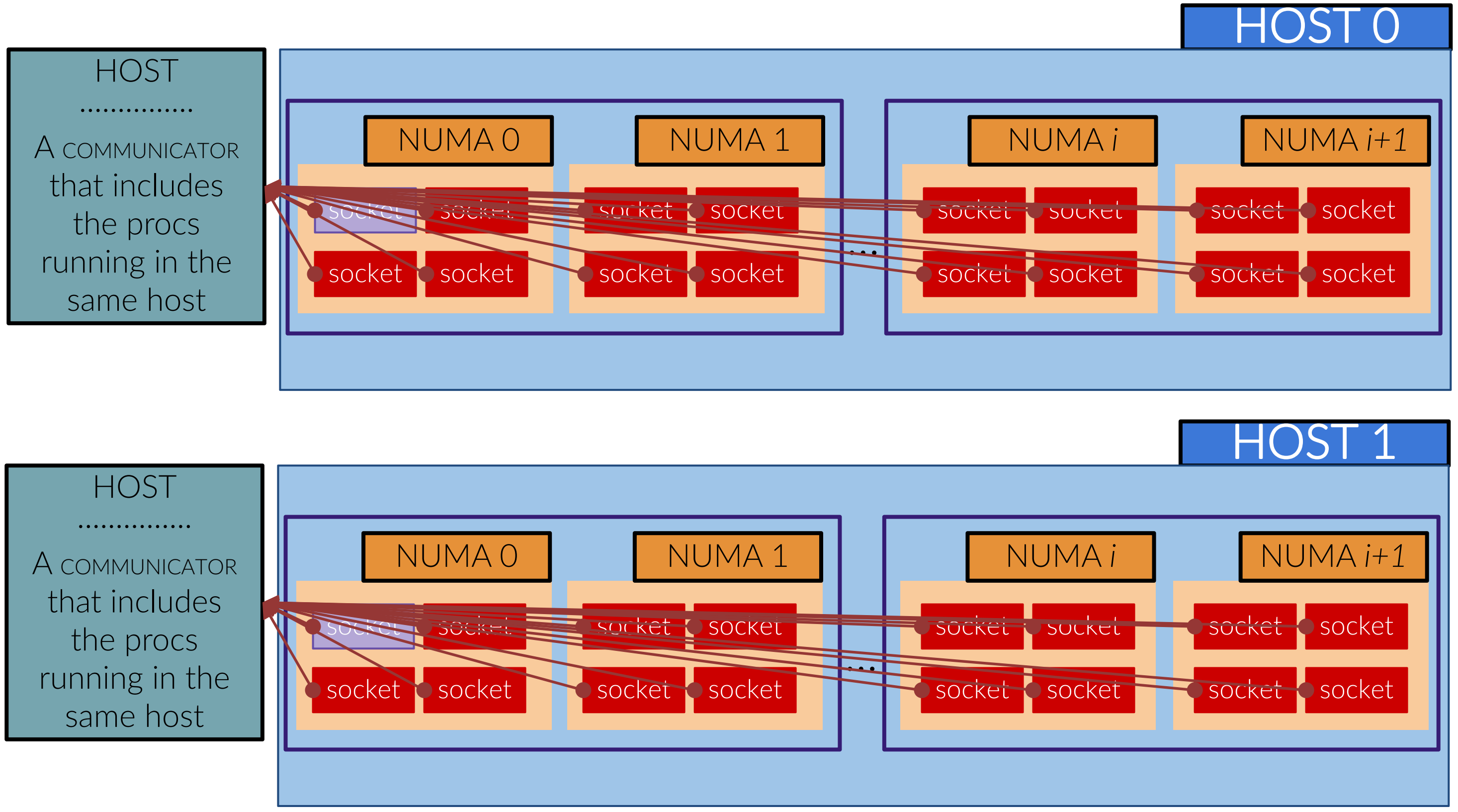
Communications among MPI tasks that live on different hosts can be grouped and exchanged in fewer messages among the host masters.

That results in a higher efficiency (smaller communication surface, smaller number of MPI calls, possible a better overlap of communication and computation)



Exploring the topology

Second, we can build a communicator that includes all the MPI procs that run on the same host.

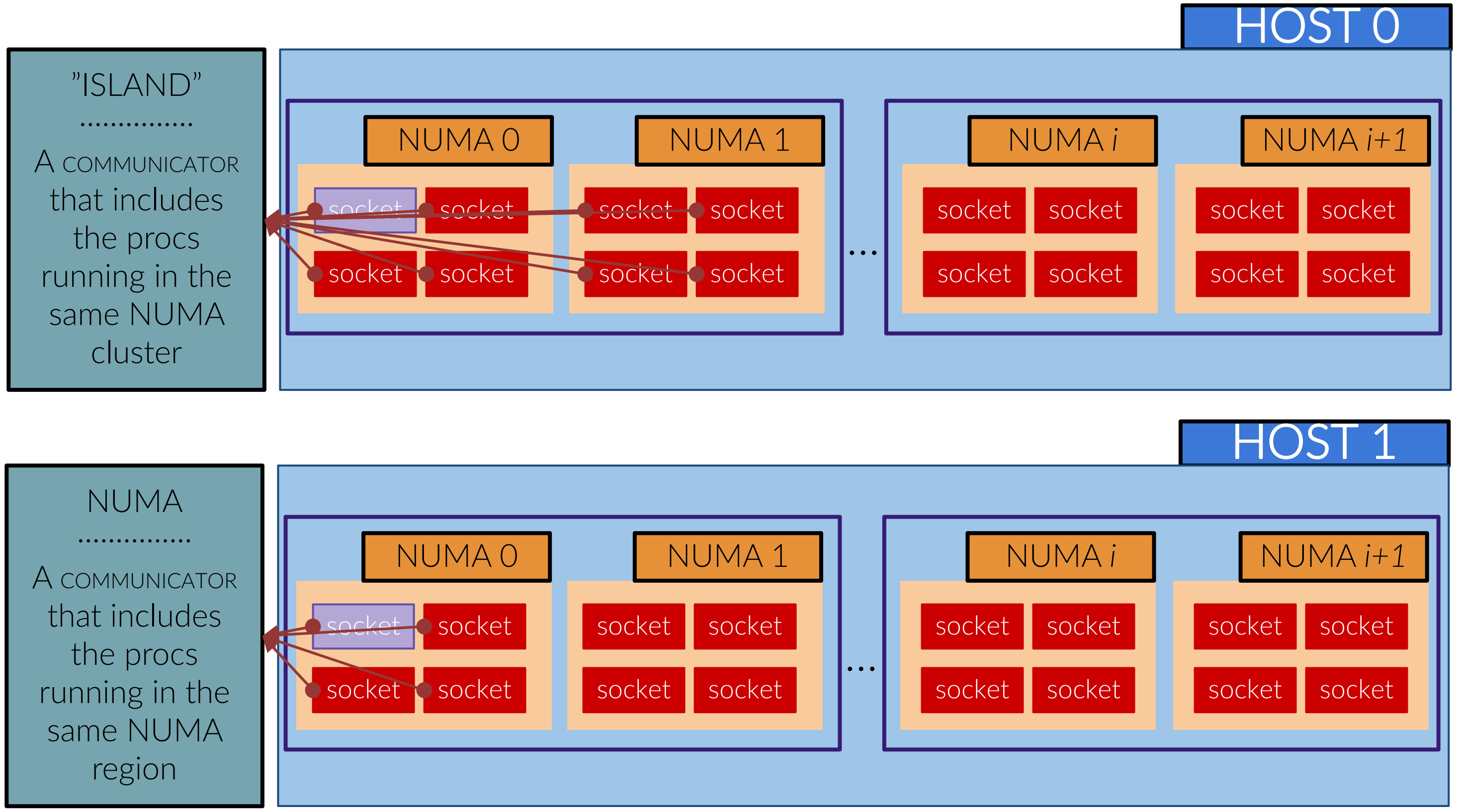

HOST 0
HOST
....
A COMMUNICATOR that includes the procs running in the same host
NUMA 0
NUMA 1
NUMA i
NUMA i+1
socket
...
HOST 1
HOST
....
A COMMUNICATOR that includes the procs running in the same host

Exploring the topology

Third, we can build a communicator for each level of "contiguity" that we are able to determine,

i.e. for which there is a precise definition and/or a system call that makes us able to decide where a process belongs to.

For instance, exploring the memory distance matrix, you may even decide to group procs that have the minimum memory distance.



"ISLAND"
....
A COMMUNICATOR that includes the procs running in the same NUMA cluster
HOST 0
NUMA 0
NUMA 1
NUMA i
NUMA i+1
socket
...
NUMA
....
A COMMUNICATOR that includes the procs running in the same NUMA region
HOST 1

Exploring the topology

In general each of these levels could be *collapsed*.

For instance, in the case in which only one host is involved, the `WORLD` does not exist and the top level is `HOST`.

If there are no "*islands*" (as is the case normally today), the `ISLAND` level collapses on the `NUMA` level.

There should be a `top_level` and a `bottom_level` "pointers" that keep track of the bottomost and topmost levels, and a `SHMEM` pointer that point to the highest level of shared memory.

WORLD

....

A COMMUNICATOR that includes all the hosts' masters

HOST

....

A COMMUNICATOR that includes all procs in the same host

"ISLAND"

....

A COMMUNICATOR that includes all procs in the same NUMA cluster

NUMA

....

A COMMUNICATOR that includes all procs in the same NUMA

Exploring the topology

1. get the cpu id for every MPI proc
2. get the socket for every MPI proc
3. get the host for every MPI proc

$\Downarrow$

4. create the HOST communicator that includes all the procs running on the same host
5. create the HOSTS communicator that includes all the host masters
6. create the NUMA communicator that includes all the procs that run on the same NUMA node

let's try to implement it

numa.c

Exploring the topology

```
1.  get the cpu id for every MPI proc
2.  get the socket for every MPI proc
3.  get the host for every MPI proc

...
```

let's try to implement it

numa.c

Exploring the topology

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

let's try to implement it

create the NUMA communicator that includes all the procs that run on the same NUMA node

numa.c

that's all, have fun