- one-sided communication
- shared memory one-sided communication

Introduction to the Message Passing Interface (MPI)

Rolf Rabenseifner

rabenseifner@hlrs.de

University of Stuttgart

High-Performance Computing-Center Stuttgart (HLRS)

www.hlrs.de

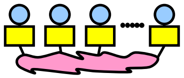
(for MPI-2.1, MPI-2.2, MPI-3.0, MPI-3.1, and MPI-4.0)

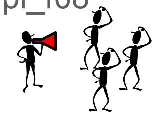
HLRS

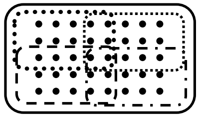
● REC → online

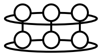
One-sided Communication

Chap.10 One-sided Communication



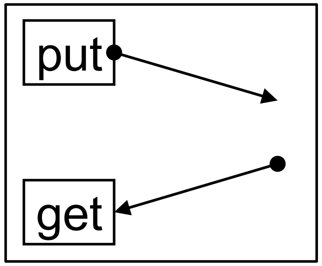


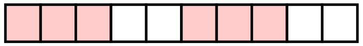


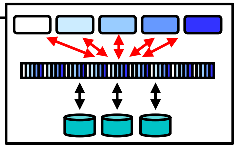


10. One-sided communication

– **Windows, remote memory access (RMA), synchronization**







Three skip-points:
1st after 1 slide
2nd after 11 slides
3rd: **Short tour** – 6 slides →
(total: 26 talk + 5 exercise-slides)

tour

 ● REC → online

One-Sided Operations

- Goals
  - PUT and GET data to/from memory of other processes
- Issues
  - Synchronization is separate from data movement
  - Automatically dealing with subtle memory behavior: cache coherence, sequential consistency
  - balancing efficiency and portability across a wide class of architectures
    - **shared-memory multiprocessor (SMP)**
    - **clusters of SMP nodes**
    - **NUMA architecture**
    - **distributed-memory MPP's**
    - **workstation networks**
- Interface
  - PUTs and GETs are surrounded by special synchronization calls

Advantages:

- Performance
  - For example, when calling PUT or GET, send and receive buffers are already defined, i.e., direct data transfer without further hand-shake is possible.
- Functionality
  - If the target process of many PUT and GET operations from other processes does not know whether it has to be part of such communications, then these many PUT/GET calls can be surrounded by a barrier-style synchronization (see example after Exercise 1+1b).

Synchronization Taxonomy

Message Passing:

explicit transfer, implicit synchronization,
implicit cache operations

Access to other processes' memory:

- **MPI 1-sided**

  explicit transfer, explicit synchronization,
  implicit cache operations (not trivial!)

- Shared Memory (e.g., in OpenMP)

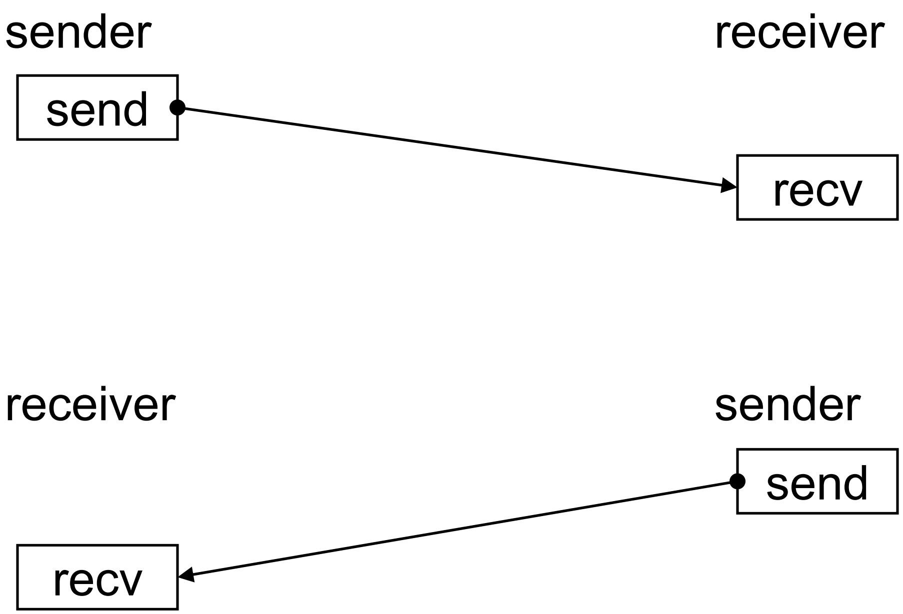
  implicit transfer, explicit synchronization,
  implicit cache operations

- shmem interface

  explicit transfer, explicit synchronization,
  explicit cache operations

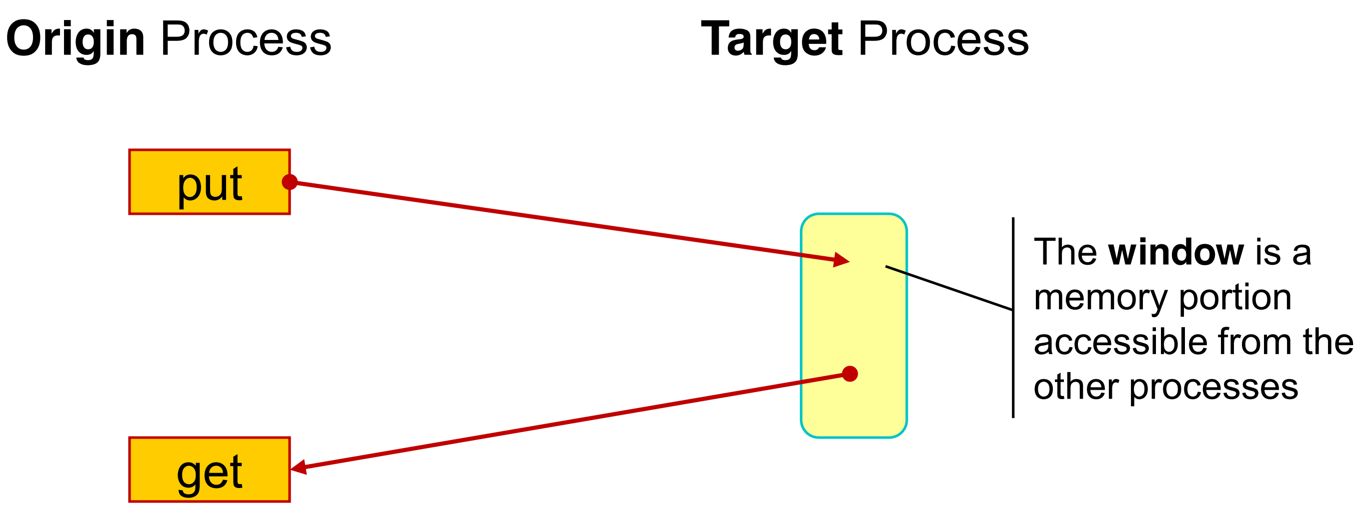
Cooperative Communication

- MPI-1 supports cooperative or 2-sided communication
- Both sender and receiver processes must participate in the communication



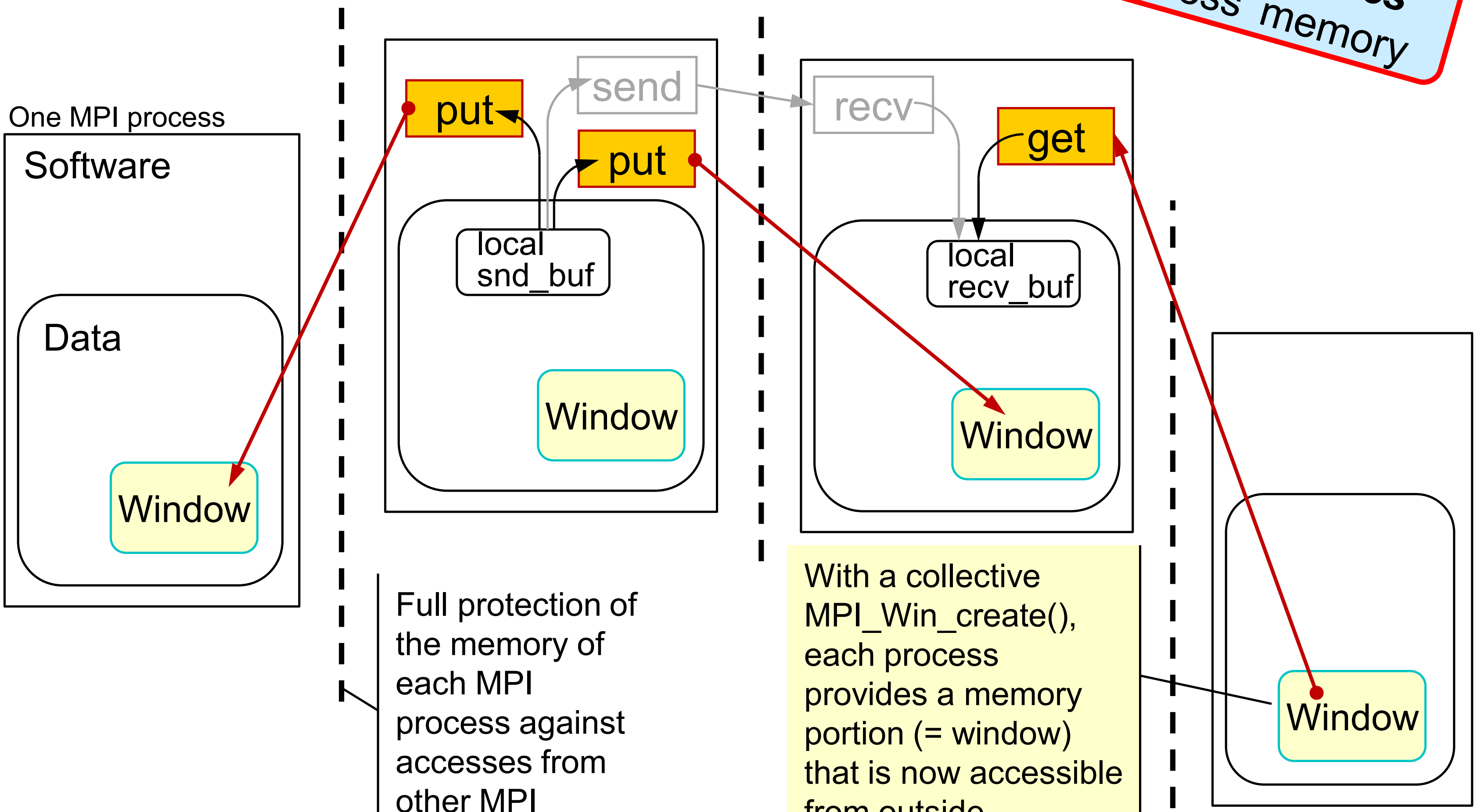
One-sided Communication

- Communication parameters for both the sender and receiver are specified by one process (origin)
- User must impose correct ordering of memory accesses



Typically, all processes are both, origin and target processes

Windows are peepholes into their process' memory



Full protection of the memory of each MPI process against accesses from other MPI processes

With a collective MPI_Win_create(), each process provides a memory portion (= window) that is now accessible from outside with put get …

 REC → online

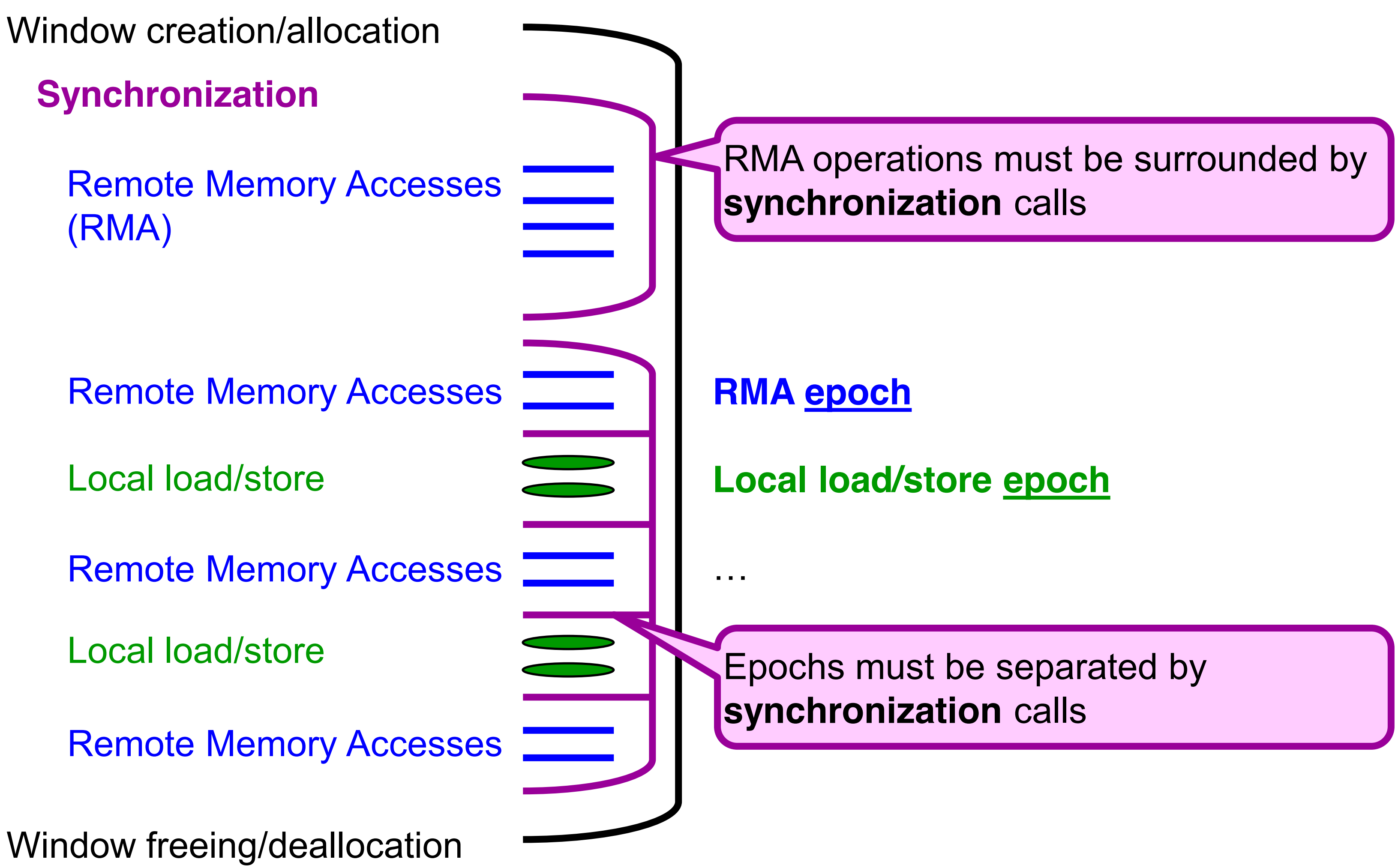
tour

One-sided Operations

Three major sets of routines:

- Window creation or allocation
  - Each process in a group of processes **(defined by a communicator)**
  - defines a chunk of own memory – named ***window***,
  - which can be afterwards accessed by all other processes of the group.
- **R**emote **M**emory **A**ccess (RMA, nonblocking) routines
  - Access to remote windows:
    - **put, get, accumulate, …**
- Synchronization
  - The RMA routines are nonblocking and
  - must be surrounded by synchronization routines,
  - which guarantee
    - **that the RMA is locally and remotely finished**
    - **and that all necessary cache operation are implicitly done.**

Sequence of One-sided Operations



Window creation or allocation

Four different methods

- Using existing memory as windows
  - MPI_Alloc_mem, **MPI_Win_create, MPI_Win_free,** MPI_Free_mem

New in MPI-3.0

- Allocating new memory as windows
  - **MPI_Win_allocate**
- Allocating shared memory windows – usable only within a shared memory node
  - **MPI_Win_allocate_shared, MPI_Win_shared_query**
- Using existing memory dynamically
  - **MPI_Win_create_dynamic, MPI_Win_attach, MPI_Win_detach**

MPI_Alloc_mem, MPI_Win_allocate, and MPI_Win_allocate_shared:

New in MPI-4.0

- Memory alignment must fit to all predefined MPI datatypes
  - alternative minimum alignment through info key "mpi_minimum_memory_alignment"

RMA Operations

- Nonblocking RMA routines
  - that are finished by subsequent window synchronization
    - **MPI_Get**
    - **MPI_Put**

      The outcome of concurrent puts to the same target location is undefined.

    - - - - - - - - - - - - - - - - - - -
    - **MPI_Accumulate**
    - **MPI_Get_accumulate**
    - **MPI_Fetch_and_op**

      Many calls by many processes can be issued for the same target element. Atomic operation for each target element.

      Get/Fetch is executed before the operation.

      Same as Get_accumulate, but only for 1 element.

    - - - - - - - - - - - - - - - - - - -
    - **MPI_Compare_and_swap**

      Substitute target element by origin buffer element if target element == compare buffer element.

  - that are completed with regular MPI_Wait, …
    - **MPI_Rget**
    - **MPI_Rput**
    - **MPI_Raccumulate**
    - **MPI_Rget_accumulate**

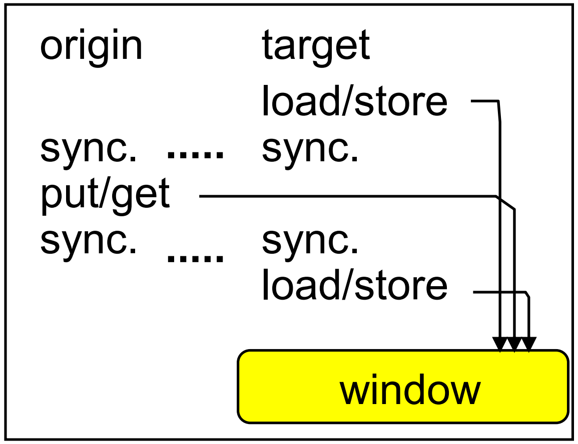
      Only within **passive** target communication, i.e., between lock & unlock, see next slide.

R = request-based

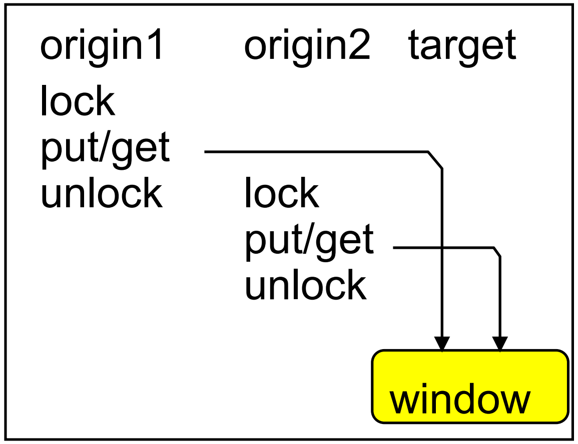
New in MPI-3.0

Synchronization Calls (1)

- Active target communication
  - communication paradigm similar to message passing model
  - target process participates only in the synchronization
  - fence or post-start-complete-wait



- Passive target communication
  - communication paradigm closer to shared memory model
  - only the origin process is involved in the communication
  - lock/unlock



 ● REC → online

Synchronization Calls (2)

- Active target communication
  - MPI_Win_fence (like a barrier)
  - MPI_Win_post, MPI_Win_start, MPI_Win_complete, MPI_Win_wait/test
- Passive target communication
  - MPI_Win_lock, MPI_Win_unlock,
  - MPI_Win_lock_all, MPI_Win_unlock_all,
  - MPI_Win_flush(_all), MPI_Win_flush_local(_all), MPI_Win_sync

New in MPI-3.0

New in MPI-3.0

Window Creation

- Specifies the region in memory (already allocated) that can be accessed by remote processes
- **Collective** call over all processes in the intracommunicator
- Returns an opaque object of type `MPI_Win` which can be used to perform the remote memory access (RMA) operations

A normal buffer argument

byte size, MPI_Aint

MPI_Win_create($\text{win\_base\_addr}_{\text{target}}$, $\text{win\_size}_{\text{target}}$, $\text{disp\_unit}_{\text{target}}$, info, comm, *win*)

byte size, int

Info handle for further customization, or just MPI_INFO_NULL.
See also course chapters 8-(2), 11-(1), 13-(1)
→ general rules,
→ alloc_shared_noncontig,
→ striping_factor

A window handle represents:
- all about the communicator
- and its processes,
- the location of the windows in all processes,
- the disp_units in all processes

Fortran C/C++ Python

language bindings
→ see next slide (skipped) or MPI Standard

 ● REC → online

skipped

Window Creation with MPI_Win_create

C

- C/C++: int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)

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

  Large count version, new in MPI-4.0

Fortran

- Fortran: MPI_Win_create(base, size, disp_unit, info, comm, *win*, ierror)

```
mpi_f08:      TYPE(*), DIMENSION(..), ASYNCHRONOUS             :: base
              INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN)       :: size
              INTEGER, INTENT(IN)                              :: disp_unit
        or    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN)       :: disp_unit
              TYPE(MPI_Info), INTENT(IN)                       :: info
              TYPE(MPI_Comm), INTENT(IN)                       :: comm
              TYPE(MPI_Win), INTENT(OUT)                       :: win
              INTEGER, OPTIONAL, INTENT(OUT)                   :: ierror
mpi & mpif.h: <type> base(*)
              INTEGER(KIND=MPI_ADDRESS_KIND) size
              INTEGER  disp_unit, info, comm, win, ierror
```

Overloaded large count version since MPI-4.0

Python

- Python: *win* = MPI.Win.Create(memory, disp_unit, info, comm)

  e.g., a numpy array

New in MPI-4.0

- Historical Fortran interface: Only in the mpi module and mpif.h
- **Modern interface with C-pointer, see next slide**

Fortran

MPI_ALLOC_MEM with old-style “Cray”-Pointer

MPI_ALLOC_MEM (size, info, *baseptr*)

MPI_FREE_MEM (base)

```
USE mpi
REAL a
POINTER (p, a(100))   ! no memory is allocated
INTEGER (KIND=MPI_ADDRESS_KIND) buf_size
INTEGER length_real, win, ierror
CALL MPI_TYPE_EXTENT(MPI_REAL, length_real, ierror)
Size = 100*length_real
CALL MPI_ALLOC_MEM(buf_size, MPI_INFO_NULL, P, ierror)
CALL MPI_WIN_CREATE(a, buf_size, length_real,
                    MPI_INFO_NULL, MPI_COMM_WORLD, win, ierror)
...
CALL MPI_WIN_FREE(win, ierror)
CALL MPI_FREE_MEM(a, ierror)
```

Fortran

New in MPI-3.0

In all three Fortran support methods

All Memory Allocation with modern C-Pointer

C

```
float *buf;  MPI_Win win; int max_length; max_length = …;
MPI_Win_allocate( (MPI_Aint)(max_length*sizeof(float)),  sizeof(float),
                           MPI_INFO_NULL,  MPI_COMM_WORLD,  &buf,  &win);
// the window elements are buf[0] .. buf[max_length-1]
```

Fortran

```
USE mpi_f08
USE, INTRINSIC :: ISO_C_BINDING
INTEGER :: max_length,  disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: lb, size_of_real, buf_size, target_disp
REAL, POINTER, ASYNCHRONOUS :: buf(:)
TYPE(MPI_Win) :: win;     TYPE(C_PTR) :: cptr_buf
max_length = …
CALL MPI_Type_get_extent(MPI_REAL, lb, size_of_real)
buf_size = max_length * size_of_real;      disp_unit = size_of_real
CALL MPI_Win_allocate(buf_size, disp_unit, MPI_INFO_NULL, MPI_COMM_WORLD,
                      cptr_buf, win)
CALL C_F_POINTER(cptr_buf, buf, (/max_length/) )
buf(0:) => buf    ! With this code, one may change the lower bound to 0 (instead of default 1)
! The window elements are buf(0) .. buf(max_length-1)
```

Python

```
np_dtype = np.single  # = C type float → MPI.FLOAT
max_length = …
win = MPI.Win.Allocate(np_dtype(0).itemsize*max_length, np_dtype(0).itemsize, MPI.INFO_NULL,
                       MPI.COMM_WORLD)
buf = np.frombuffer(win, dtype=np_dtype)
# the window elements are buf[0] .. buf[max_length-1]
# buf = np.reshape(buf,())  # in case of max_length==1 and using buf as a normal variable instead of a 1-dim array
```

 ● REC → online

MPI_Put

- Performs an operation equivalent to a **send** by the **origin process** and a matching **receive** by the target process
- The origin process specifies the arguments for both origin and target
- **Nonblocking call** → finished by subsequent synchronization call
  → don't modify the origin (=send) buffer until next synchron.

Where is the recv_buf in the target process **?**

- The target buffer is at address
  $target\_addr = win\_base_{\textbf{target\_process}}$
  $+ \ target\_disp_{\textbf{origin\_process}} * disp\_unit_{\textbf{target\_process}}$

As provided in MPI_Win_create or _allocate at the target process

Like **send_buf, count, datatype** in MPI_Send

```
MPI_Put( origin_address, origin_count, origin_datatype,
         target_rank, target_disp_origin_process,
         target_count, target_datatype, win)
```

Like **dest** in MPI_Send

Like **count, datatype** in an MPI_Recv at the target process

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

– skipped –

MPI_Put

C

- C/C++: 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)

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

  Large count version, new in MPI-4.0

Fortran

- Fortran: MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
  target_disp, target_count, target_datatype, win, ierror)

mpi_f08:
```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
INTEGER, INTENT(IN)                          :: origin_count, target_count
or INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN)  :: origin_count, target_count
INTEGER, INTENT(IN)                          :: target_rank
TYPE(MPI_Datatype), INTENT(IN)               :: origin_datatype, target_datatype
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN)   :: target_disp
TYPE(MPI_Win), INTENT(IN)                    :: win
INTEGER, OPTIONAL, INTENT(OUT)               :: ierror
```

Overloaded large count version since MPI-4.0

mpi & mpif.h:
```
<type> ORIGIN_ADDR(*)
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
INTEGER TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
```

Python

- Python: win.Put((origin_buf, origin_count, origin_datatype), target_rank,
  (target_disp, target_count, target_datatype))

New in MPI-4.0

 ● REC → online

MPI_Get

- Similar to the put operation, except that data is transferred from the target memory to the origin process
- To complete the transfer a synchronization call must be made on the window involved
- The local buffer should not be accessed until the synchronization call is completed

```
MPI_Get( origin_address, origin_count, origin_datatype,
         target_rank, target_disp, target_count,
         target_datatype, win)
```

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

MPI_Accumulate

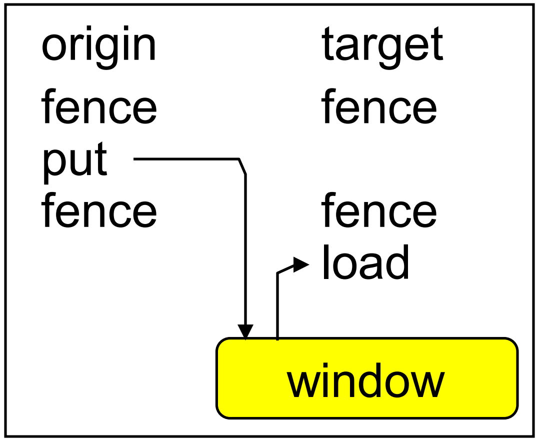
- Accumulates the contents of the origin buffer to the target area specified using the predefined operation `op`
- User-defined operations cannot be used
- Accumulate is **elementwise atomic**:
  many accumulates can be done by many origins to one target
  -> [*may be expensive*]

```
MPI_Accumulate(origin_address, origin_count,
               origin_datatype, target_rank, target_disp,
               target_count, target_datatype, op, win)
```

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

MPI_Win_fence

- Synchronizes RMA operations on specified window
- Collective over the window
- **Like a barrier**
- Used for active target communication
- Should be used before and after calls to put, get, and accumulate
- The `assert` argument is used to provide optimization hints to the implementation,
  - see MPI-3.1/MPI-4.0, Sect. 11.5.5/12.5.5 "Assertions" (page 450/607)
  - enables the optimization of internal cache operations
  - Integer 0 = no assertions
  - Several assertions with ***bitwise or*** operation



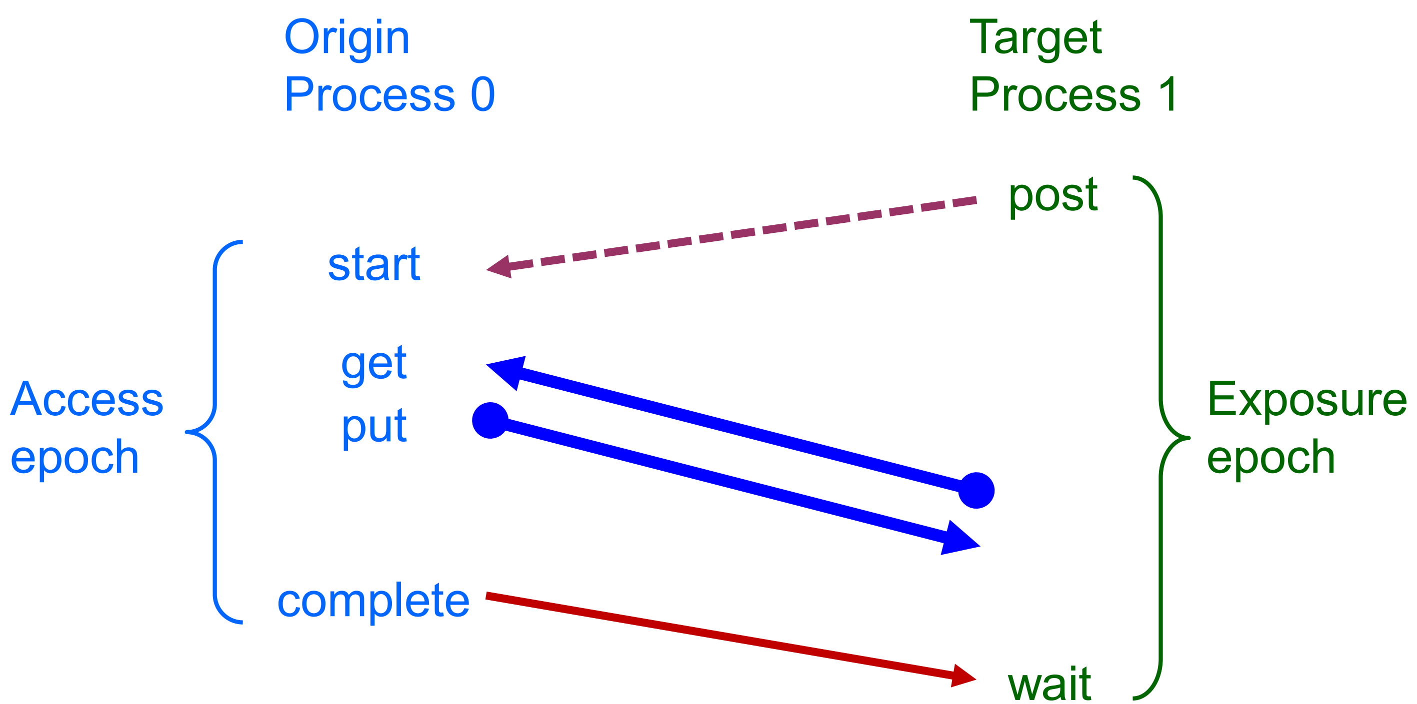
MPI_Win_fence(assert, win)

E.g., in C: `MPI_MODE_NOSTORE | MPI_MODE_... | MPI_MODE_...`
Fortran: `IOR(MPI_MODE_NOSTORE, IOR(MPI_MODE_..., MPI_...))`
Because assertions are bit-vectors, e.g.

- `MPI_MODE_NOSTORE   = 00L00`
- `MPI_MODE_PUT       = 000L0`
- `MPI_MODE_NOSUCCEED = 0000L`

Start/Complete and Post/Wait, I.

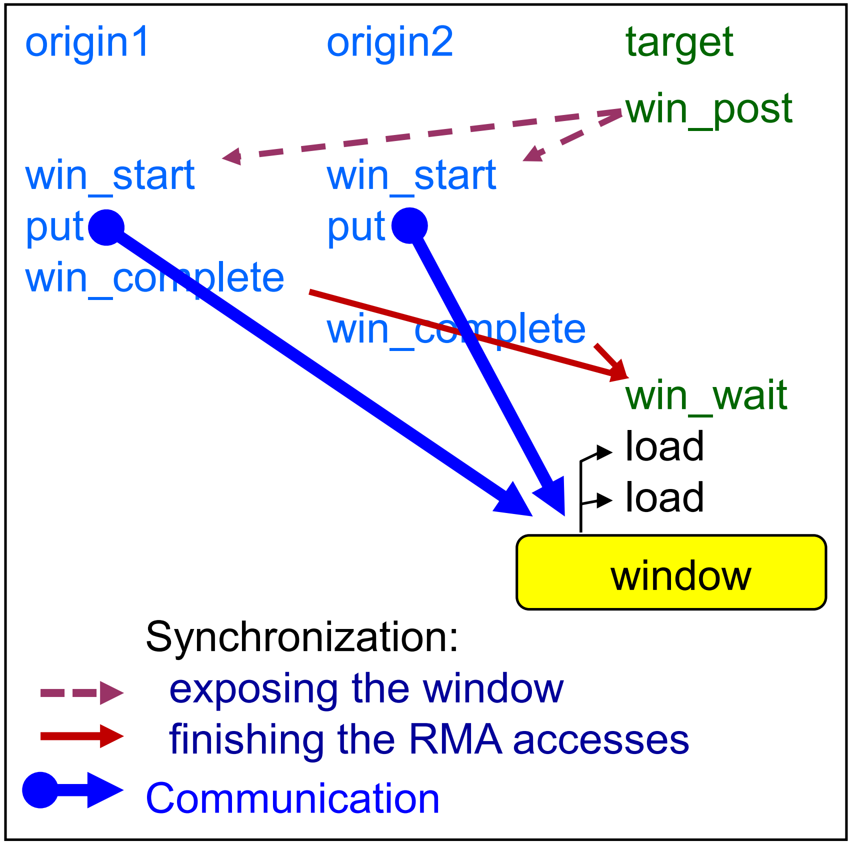
- Used for active target communication
  to restrict synchronization to a minimum



 ● REC → online

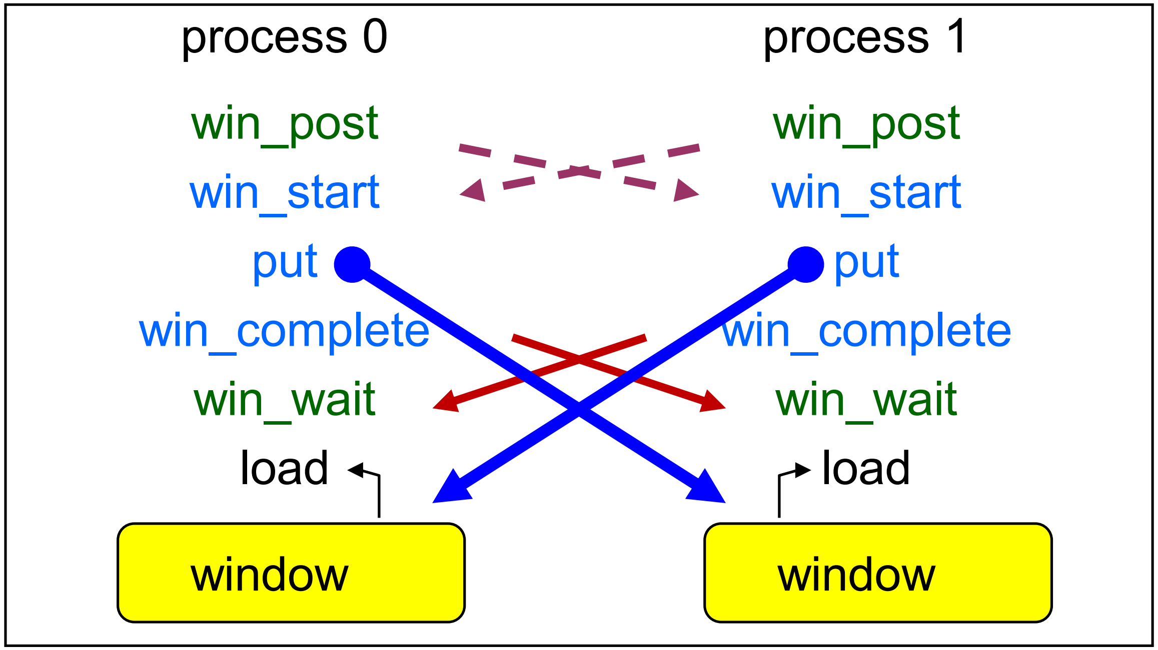
Start/Complete and Post/Wait, II.

- RMA (put, get, accumulate) are finished
  - locally after win_complete
  - at the target after win_wait
- local buffer must not be reused before RMA call locally finished
- communication partners must be known
- no atomicity for overlapping “puts”
- assertions may improve efficiency --> give all information you have



Start/Complete and Post/Wait, III.

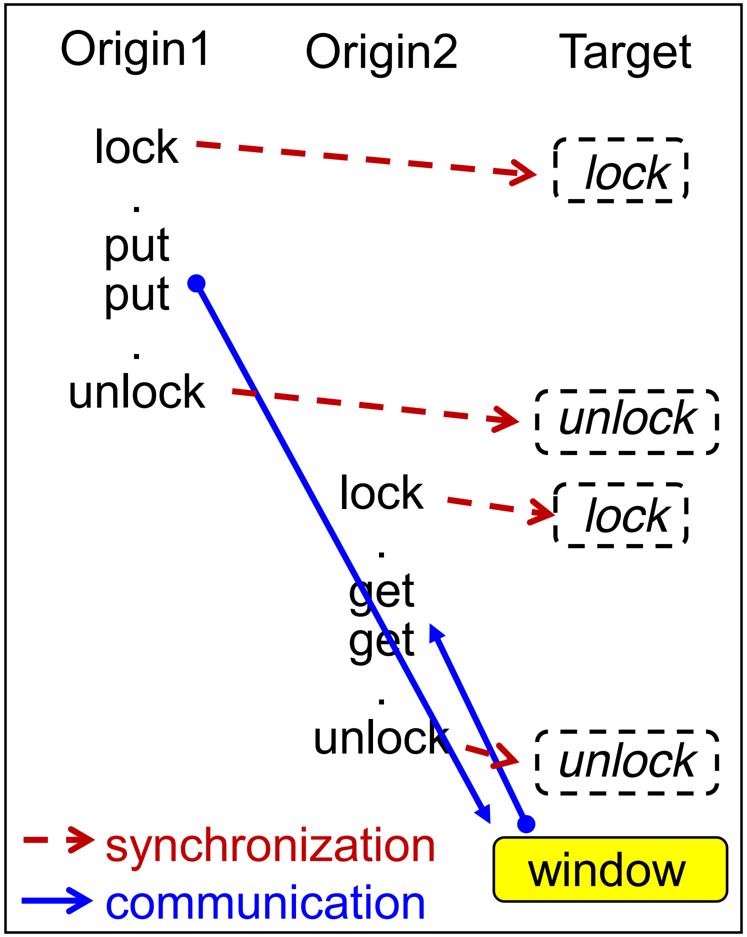
- symmetric communication possible,
  only win_start and win_wait may block



- Here, all processes are in the role of **target** *and* **origin**, i.e.
  - **expose** a window *and*
  - **access** windows per RMA *and*
  - complete the RMA accesses

Lock/Unlock

- Does not guarantee a sequence
- agent may be necessary on systems without (virtual) shared memory
- Portable programs can use lock calls to windows in memory allocated **only** by **`MPI_Alloc_mem`**, **`MPI_Win_allocate`**, or **`MPI_Win_attach`** or **`MPI_Win_allocate_shared`** New in MPI-4.0
- RMA completed after **`MPI_Unlock`** at both origin and target



Fortran

Fortran Problems with 1-Sided

Source of Process 1	Source of Process 2	Executed in Process 2
bbbb = 777	buff = 999	register_A := 999
call MPI_WIN_FENCE	call MPI_WIN_FENCE	
call MPI_PUT(bbbb		stop application thread
into buff of process 2)		buff := 777 in PUT handler
		continue application thread
call MPI_WIN_FENCE	call MPI_WIN_FENCE	
	print *, buff	print *, register_A

- Fortran register optimization
- Result: 999 is printed instead of expected 777
- How to avoid: (see MPI-3.1 / MPI-4.0, Sect. 17.1.17 / 19.1.17, pages 640ff / 826ff)
  - Window memory declared in COMMON blocks or as module data i.e. MPI_ALLOC_MEM cannot be used
  - Or declare window **buff** as **ASYNCHRONOUS** and **IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(buff)** before 1st and after 2nd FENCE in process 2
  - Same for bbbb due to nonblocking MPI_PUT: Declare also **bbbb** as **ASYNCHRONOUS** (because bbbb **not** in arg-list of 2nd=finishing FENCE) + **IF (…) CALL MPI_F_SYNC_REG(bbbb)**

See at end of course Chapter 4, slides on "*Nonblocking Receive and Register Optimization / Code Movement in Fortran*" and course Chapter 5

 ● REC → online

Other One-sided Routines

- Process group of a window
  - MPI_Win_get_group
- Attributes and names
  - MPI_Win_get/set_attr
  - MPI_Win_get/set_name
- Info attached to a window **New in MPI-3.0**
  - MPI_Win_set/get_info

One-sided: Functional Opportunities – an Example

- The receiver
  - needs information and
  - does not know the sending processes nor the number of sending processes (**nsp**)
  - and this number is small compared to the total number.
  - The sender knows all its neighbors, which need some data.
- Non-scalable solution to exchange number of neighbors:
  - MPI_ALLTOALL, MPI_REDUCE_SCATTER_BLOCK (array with one logical entry per process)
  - Each sender tells all processes whether they will get a message or not.
- Solution with 1-sided communication:
  - Each process in the role being a receiver:
    - **MPI_Win_create(&nsp, …); nsp=0;** (i.e., I do not yet know the number of my sending neighbors)
  - Each process as a sender tells the receiver "here is 1 neighbor from you"
    - **MPI_Win_fence**
    - **Multiple calls to MPI_Accumulate to add 1 in the nsp of its neighbors.**
    - **MPI_Win_fence**
  - Now, each process as a receiver knows in its nsp the number of its neighbors. Therefore:
    - **Loop over nsp with MPI_Irecv(MPI_ANY_SOURCE)**
  - Each process as a sender
    - **Loop over its neigbors, sending the data.**
  - As receiver: **MPI_Waitall()** – in the statuses array, the receiver can see the neighbor's ranks

Alter-native	**sender:**	**Isend to all neighbors**
	receiver:	Loop over nsp with **R**ecv or **Probe**+malloc+Recv
	sender:	Waitall

Another scalable solution: see Chapter 6-(2) → nonblocking barrier

ibarrier

tour

2[nd] skip-point: Skip rest of this chapter

 ● REC → online

Summary

One-sided: Summary

- Functional opportunities for some specific problems:
  - Scalable solutions with 1-sided compared to point-to-point or collective calls
- Several one-sided communication primitives
  - put / get / accumulate / ….
- Surrounded by several synchronization options
  - fence / post-start-complete-wait / lock-unlock …
- User must ensure that there are no conflicting accesses
- For better performance **assertions** should be used with fence, start, post, and lock/lockall operations
- Performance-opportunities depend largely on the quality of the MPI library
  - See also halo example in next course chapter

 ● REC → online

MPI–One-sided Exercise 1: Ring communication with fence

Exercise 1

In MPI/tasks/…

- Use C C/Ch10/ring-1sided-win-skel.c
  or Fortran F_30/Ch10/ring-1sided-win-skel_30.f90
  or Python PY/Ch10/ring-1sided-win-skel.py

- General goal of exercises 1 and 2:

(The substitution of Issend/Recv/Wait by Win_fence/Put/Win_fence comes later in Exercise 2)

  – Substitute the nonblocking communication by one-sided communication.
  – Two choices:
    - **either rcv_buf = window**
      – MPI_Win_fence - the rcv_buf can be used to receive data
      – MPI_Put - to write the content of the local variable snd_buf into the remote window (rcv_buf)
      – MPI_Win_fence - the one-sided communication is finished, rcv_buf is filled
    - **or snd_buf = window**
      – MPI_Win_fence - the snd_buf is filled
      – MPI_Get - to read the content of the remote window (snd_buf) into the local variable rcv_buf
      – MPI_Win_fence - the one-sided communication is finished, rcv_buf is filled

Please use this choice in this exercise!

- **Task of this Exercise 1: Create all rcv_buf as windows in their processes, that's all in this exercise!**

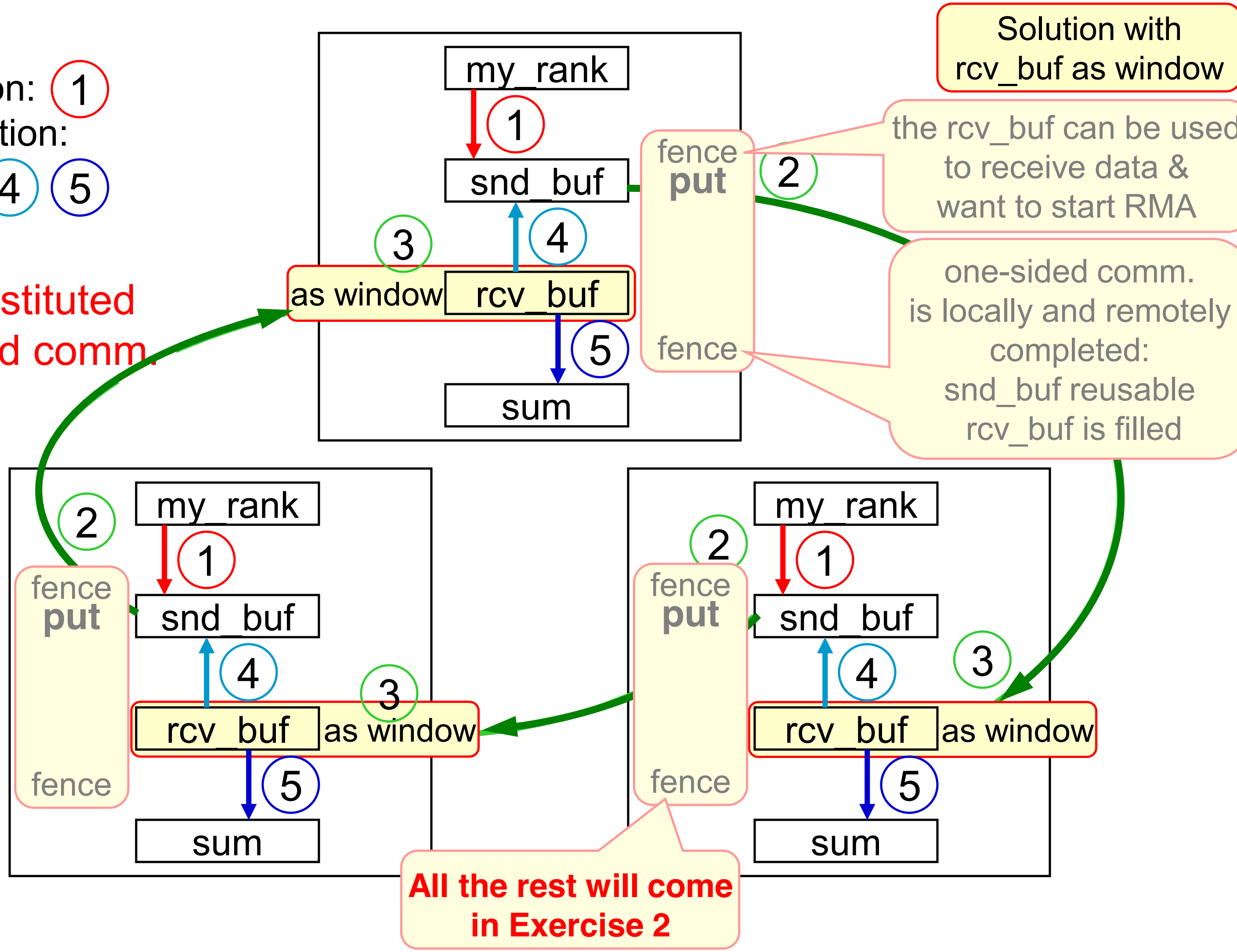
ring.c / .f: Rotating information around a ring

Initialization: 1

Each iteration:

2 3 4 5

to be substituted by 1-sided comm.



MPI–One-sided Exercise 1: additional hints

- MPI_Win_create:
  - base = reference to your rcv_buf or snd_buf variable
  - disp_unit = number of bytes of one int / integer, because this is the datatype of the buffer (=window)
  - size = same number of bytes, because buffer size = 1 value
  - size and disp_unit have different internal representations, therefore:

C

- **C/C++:** **MPI_Win_create(&rcv_buf, (MPI_Aint) sizeof(int), sizeof(int), MPI_INFO_NULL, …, &win);**

Fortran

- **Fortran:**

```
INTEGER disp_unit
INTEGER (KIND=MPI_ADDRESS_KIND) winsize, lb, extent
CALL MPI_TYPE_GET_EXTENT(MPI_INTEGER, lb, extent, ierror)
…
disp_unit = extent
winsize = disp_unit * 1
CALL MPI_WIN_CREATE(rcv_buf, winsize, disp_unit, MPI_INFO_NULL, …, ierror)
```

- MPI-3.1/MPI-4.0, Sect. 11.2.1, pages 403ff / Sect. 12.2.1, pages 553ff
- **Create all rcv_buf as windows in their processes, that's all in this exercise!**
- **(The substitution of Issend/Recv/Wait by Win_fence/Put/Win_fence comes later in Exe. 2)**

 ● REC → online

Exercise 2

MPI–One-sided Exercise 2: Ring communication with fence

- Use C C/Ch10/ring-1sided-put-skel.c
  or Fortran F_30/Ch10/ring-1sided-put-skel_30.f90
  or Python PY/Ch10/ring-1sided-put-skel.py
- General goal of exercises 1 and 2:
  - Substitute the nonblocking communication by one-sided communication.
  - Two choices:
    - **either rcv_buf = window**
      - MPI_Win_fence - the rcv_buf can be used to receive data
      - MPI_Put - to write the content of the local variable snd_buf into the remote window (rcv_buf)
      - MPI_Win_fence - the one-sided communication is finished, rcv_buf is filled
    - **or snd_buf = window**
      - MPI_Win_fence - the snd_buf is filled
      - MPI_Get - to read the content of the remote window (snd_buf) into the local variable rcv_buf
      - MPI_Win_fence - the one-sided communication is finished, rcv_buf is filled

Please use this choice in this exercise!

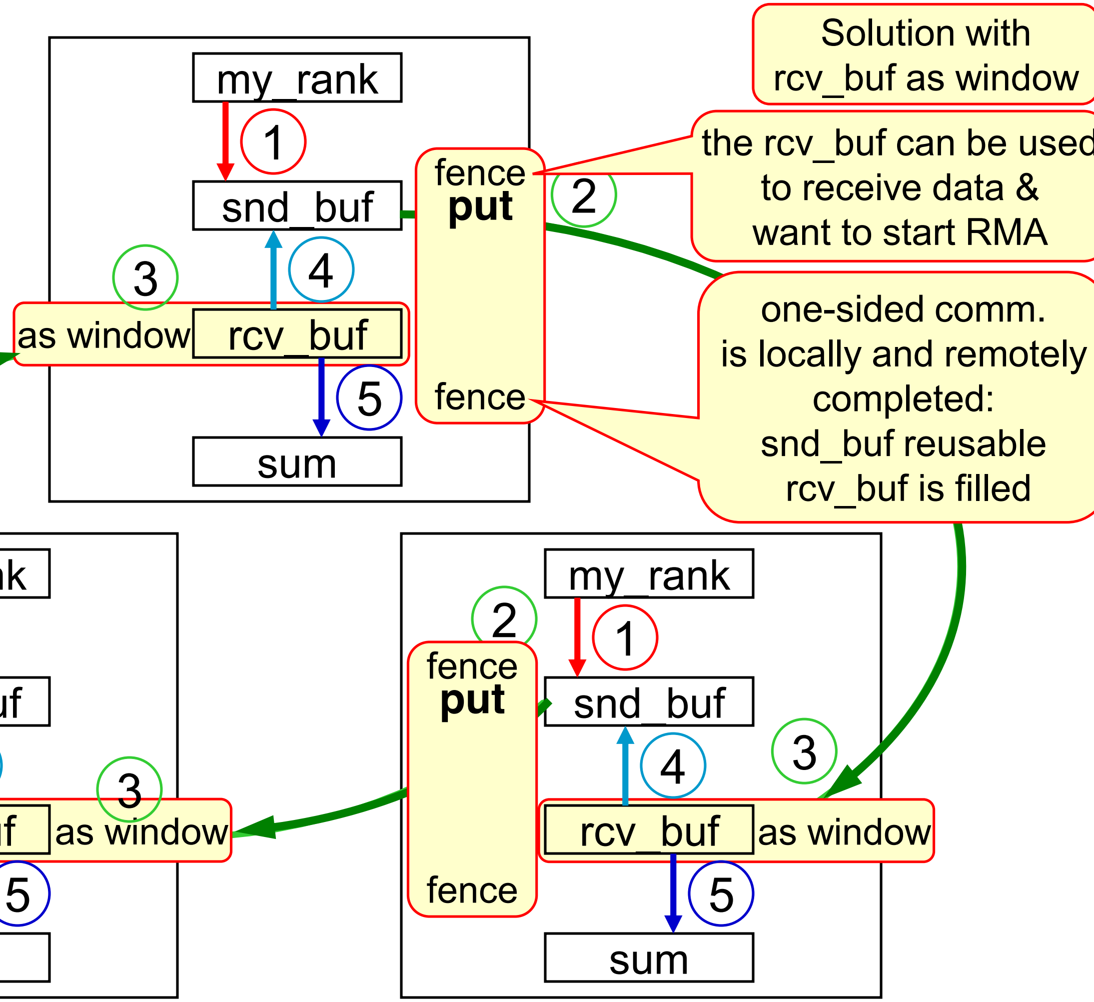
- In Exercise 1, you created the rcv_buf as windows, i.e., now accessible from outside through RMA operations.
- **Now, please substitute Issend/Recv/Wait by Win_fence/Put/Win_fence**

 ● REC → online

ring.c / .f: Rotating information around a ring

Initialization: 1
Each iteration:
2 3 4 5

to be substituted by 1-sided comm.



MPI–One-sided Exercise 2: additional hints

C

Fortran

Fortran

- MPI_Put (or MPI_Get):
  - target_disp
    - **C/C++:** MPI_Put(&snd_buf, 1, MPI_INT, right**, (MPI_Aint) 0**, 1, MPI_INT, win);
    - **Fortran:** **INTEGER (KIND=MPI_ADDRESS_KIND) target_disp**
      target_disp = 0
      …
      CALL MPI_PUT(snd_buf, 1, MPI_INTEGER, right, **target_disp**, 1,
      MPI_INTEGER, win, ierror)

      Or just *"long"* integer constant 0_MPI_ADDRESS_KIND
  - Register problem with Fortran with destination buffer of **non-blocking** RMA operation:
    - **Access to the rcv_buf before 1st <u>and</u> after 2nd MPI_WIN_FENCE:**

```
INTEGER, ASYNCHRONOUS :: snd_buf, rcv_buf
…
IF (.NOT.  MPI_ASYNC_PROTECTS_NONBLOCKING) &
&          CALL MPI_F_SYNC_REG(rcv_buf)
```

    - **Because MPI_PUT(snd_buf) is nonblocking → same with snd_buf after the 2nd FENCE**
- MPI_Put, see MPI-3.1, Sect. 11.3.1, pages 418f or MPI-4.0, Sect. 12.3.1, pages 570f
  and Fortran MPI-3.1, Sect. 17.1.10-19, p. 631-648 or MPI-4.0, Sect. 19.1.10-19, pages 817f
- Assertions for MPI_WIN_FENCE:
  See MPI-3.1, Sect. 11.5.5, pages 451 or MPI-4.0, Sect. 12.5.5, pages 607f

MPI–One-sided Exercise 3: Post-start-complete-wait

Exercise 3

- Use your result of exercise 2 or copy to your local directory:

C **cp** ~/MPI/tasks/**C**/Ch10/solutions/ring-1sided-put.c **my_1sided_exa3.c**

Fortran **cp** ~/MPI/tasks/**F_30**/Ch10/solutions/ring-1sided-put_30.f90 **my_1sided_exa3_30.f90**

Python **cp** ~/MPI/tasks/**PY**/Ch10/solutions/ring-1sided-put.py **my_1sided_exa3.py**

- Tasks:
  - Substitute the two calls to MPI_Win_fence by calls to MPI_Win_post / _start / _complete / _wait
  - Use of group mechanism to address the neighbors:
    - **MPI_Comm_group(comm, *group*)**
    - **MPI_Group_incl(group, n, ranks, *newgroup*)**
      - Fortran new mpi_f08: TYPE(MPI_Comm) :: comm; INTEGER n, ranks(...); TYPE(MPI_Group) :: group, newgroup
      - C: MPI_Comm comm; MPI_Group group, newgroup; int n, ranks[...];
  - Compile and run your `my_1sided_exa3.c / _30.f90`

Chapter 10: Ring with one-sided communication

C

MPI/tasks/C/Ch10/solutions/ring-1sided-win.c

```
MPI_Win  win;
---------------------------------------------------------------------------
/* Create the window once before the loop:  */
MPI_Win_create(&rcv_buf, (MPI_Aint) sizeof(int), sizeof(int), MPI_INFO_NULL,
               MPI_COMM_WORLD, &win);
---------------------------------------------------------------------------
```

Fortran

MPI/tasks/F_30/Ch10/solutions/ring-1sided-win_30.f90

```
INTEGER, ASYNCHRONOUS::snd_buf
TYPE(MPI_Win) :: win ; INTEGER :: disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: integer_size, lb, buf_size, target_disp
---------------------------------------------------------------------------
! Create the window once before the loop:
CALL MPI_TYPE_GET_EXTENT(MPI_INTEGER, lb, integer_size)
buf_size = 1 * integer_size; disp_unit = integer_size
CALL MPI_WIN_CREATE(rcv_buf, buf_size, disp_unit, MPI_INFO_NULL, &
 &                    MPI_COMM_WORLD, win)
---------------------------------------------------------------------------
```

Provided in the skeleton

Python

MPI/tasks/PY/Ch10/solutions/ring-1sided-win.py

```
np_dtype = np.intc
rcv_buf = np.empty((),dtype=np_dtype)
win = MPI.Win.Create(memory=rcv_buf, disp_unit=rcv_buf.itemsize,
                     info=MPI.INFO_NULL, comm=comm_world)
```

Chapter 10: Ring with one-sided communication

C

MPI/tasks/C/Ch10/solutions/ring-1sided-put.c

```
MPI_Win  win;
-------------------------------------------------------------
/* Create the window once before the loop:  */
MPI_Win_create(&rcv_buf, (MPI_Aint) sizeof(int), sizeof(int), MPI_INFO_NULL,
               MPI_COMM_WORLD, &win);
-------------------------------------------------------------
  MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPRECEDE, win);
  MPI_Put(&snd_buf, 1, MPI_INT, right, (MPI_Aint) 0, 1, MPI_INT, win);
  MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT | MPI_MODE_NOSUCCEED,win);
```

Inside of the loop; instead of Issend + Recv + Wait

Fortran

MPI/tasks/F_30/Ch10/solutions/ring-1sided-put_30.f90

```
INTEGER, ASYNCHRONOUS::snd_buf,rcv_buf
TYPE(MPI_Win) :: win ; INTEGER :: disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: integer_size, lb, buf_size, target_disp
-------------------------------------------------------------
! Create the window once before the loop:
CALL MPI_TYPE_GET_EXTENT(MPI_INTEGER, lb, integer_size)
buf_size = 1 * integer_size; disp_unit = integer_size
CALL MPI_WIN_CREATE(rcv_buf, buf_size, disp_unit, &
 &                  MPI_INFO_NULL, MPI_COMM_WORLD, win)
-------------------------------------------------------------
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
  CALL MPI_WIN_FENCE(IOR(MPI_MODE_NOSTORE,MPI_MODE_NOPRECEDE), win)
  target_disp=0 ! This "long" integer zero is needed in the call to MPI_PUT
  CALL MPI_PUT(snd_buf,1,MPI_INTEGER,right,target_disp,1,MPI_INTEGER, win)
  CALL MPI_WIN_FENCE(IOR(MPI_MODE_NOSTORE, IOR(MPI_MODE_NOPUT, MPI_MODE_NOSUCCEED)),win)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
```

In **ring-1sided-put-WRONG-S_30.f90**, these lines are commented out:
For example using gfortran with –O4, you may get completely wrong results.

Inside of the loop; instead of Issend + Recv + Wait

Python

MPI/tasks/PY/Ch10/solutions/ring-1sided-put.py

```
np_dtype = np.intc
rcv_buf = np.empty((),dtype=np_dtype)
win = MPI.Win.Create(memory=rcv_buf, disp_unit=rcv_buf.itemsize,
                     info=MPI.INFO_NULL, comm=comm_world)
-------------------------------------------------------------
  win.Fence(MPI.MODE_NOSTORE | MPI.MODE_NOPRECEDE)
  win.Put((snd_buf, 1, MPI.INT), right, (0, 1, MPI.INT))
  win.Fence(MPI.MODE_NOSTORE | MPI.MODE_NOPUT | MPI.MODE_NOSUCCEED)
```

Inside of the loop; instead of Issend + Recv + Wait

 ● REC → online

Assertions → next slide

Chapter 10: Ring with one-sided communication – Assertions

```
   /* in previous loop iterations */
     … = rcv_buf        /*the window*/
   /* Inside of the loop; instead of MPI_Issend / MPI_Recv / MPI_Wait:  */
A    MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPRECEDE, win);
     MPI_Put(&snd_buf, 1, MPI_INT, right, (MPI_Aint) 0, 1, MPI_INT, win);
B    MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT | MPI_MODE_NOSUCCEED,win);
     … = rcv_buf        /*the window*/
```

MPI_WIN_FENCE:

A B MPI_MODE_NOSTORE — the local window was not updated by stores (or local get or receive calls) since last synchronization.

B MPI_MODE_NOPUT — the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.

A MPI_MODE_NOPRECEDE — the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.

B MPI_MODE_NOSUCCEED — the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.

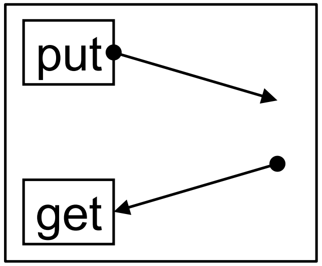
MPI-3.1, Sect.11.5.5., page 451 lines 26-38: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf#page=483
MPI-4.0, Sect.12.5.5., page 609 lines 1-11: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf#page=649

 ● REC → online

Chap.11 Shared Memory One-sided Communication

11. Shared memory one-sided communication

- **(1) MPI_Comm_split_type & MPI_Win_allocate_shared**
  Hybrid MPI and MPI shared memory programming
- **(2) MPI memory models and synchronization rules**



 ● REC → online

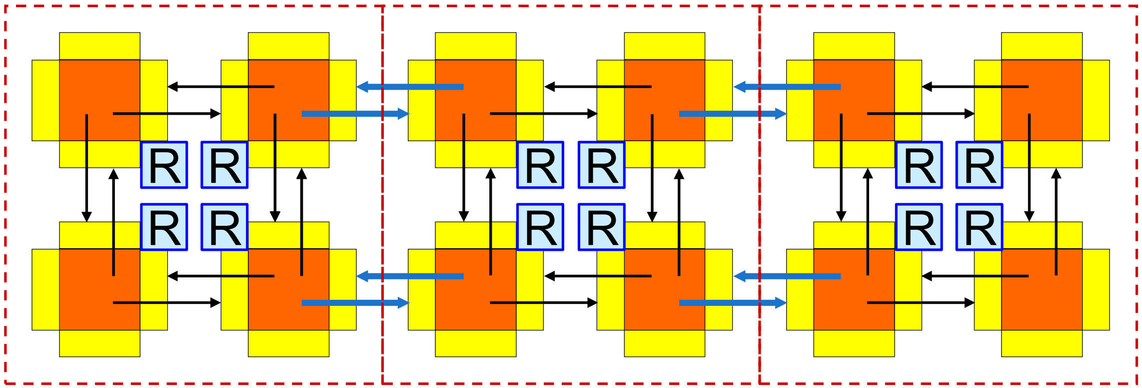
MPI shared memory

- Split main communicator into shared memory islands
  - **MPI_Comm_split_type**
- Define a shared memory window on each island
  - **MPI_Win_allocate_shared**
  - Result (by default):
    contiguous array, directly accessible by all processes of the island
- Accesses and synchronization
  - Normal assignments and expressions
  - No **MPI_Put/Get** !
  - Normal MPI one-sided synchronization, e.g., **MPI_Win_fence**
- Caution:
  - Memory may be already completely pinned to the physical memory of the process with rank 0, i.e., the first touch rule (as in OpenMP) does **not** apply!
    (First touch rule: a memory page is pinned to the physical memory of the processor that first writes a byte into the page)

tour

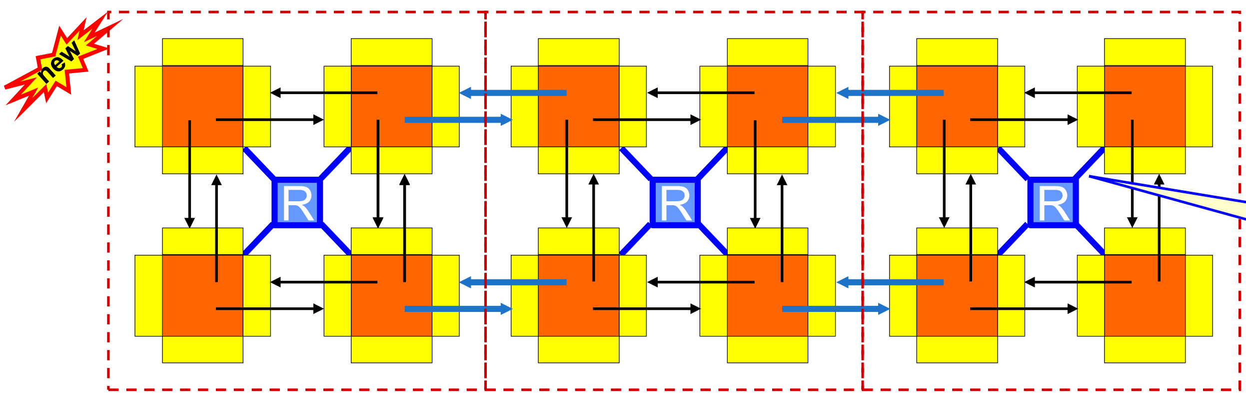
 ● REC → online

Programming opportunities with MPI shared memory: 1) Reducing memory space for replicated data



R = Replicated data in each MPI process

Example:
Cluster of 3 SMP nodes
without using MPI
shared memory methods



R = Shared memory
→ replicated data
<u>only once</u> within
each SMP node

Direct loads & stores,
no library calls

Using MPI shared memory methods

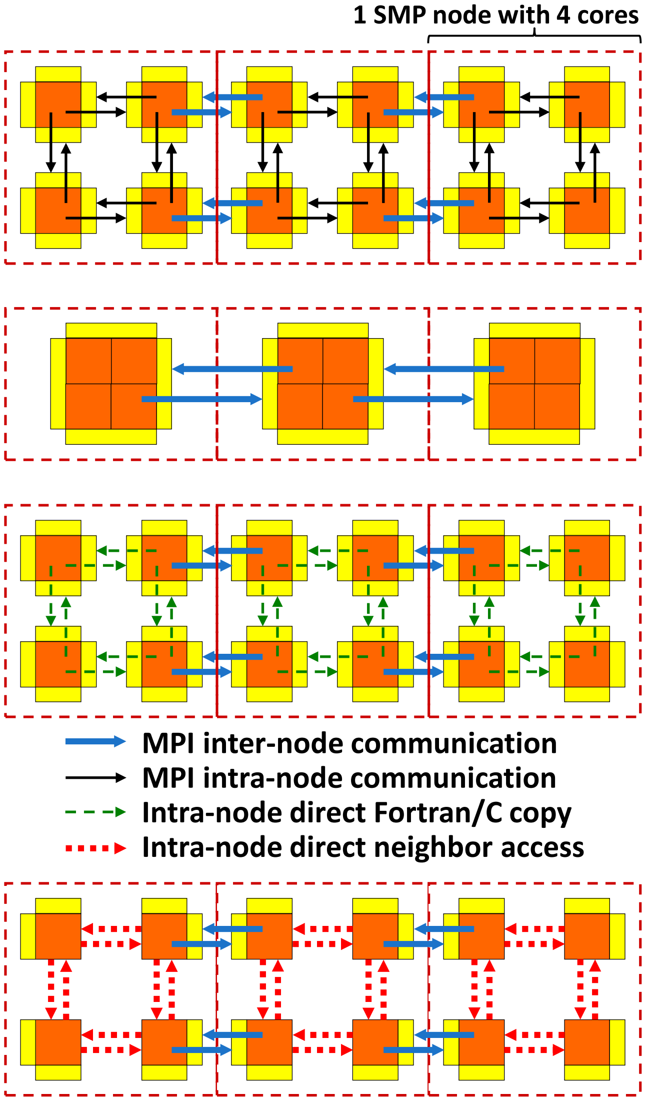
MPI shared memory can be used
to **significantly reduce the memory needs** for **replicated data**.

back
→Exa5

tour

Programming opportunities with MPI shared memory: 2) Hybrid shared/cluster programming models

- MPI on each core (not hybrid)
  - Halos between all cores
  - MPI uses internally shared memory and cluster communication protocols
- MPI+OpenMP
  - Multi-threaded MPI processes
  - Halos communica. only between MPI processes
- new MPI cluster communication + MPI shared memory communication
  - Same as “MPI on each core”, but
  - within the shared memory nodes, halo communication through direct copying with C or Fortran statements
- new MPI cluster comm. + MPI shared memory access
  - Similar to “MPI+OpenMP”, but
  - shared memory programming through work-sharing between the MPI processes within each SMP node

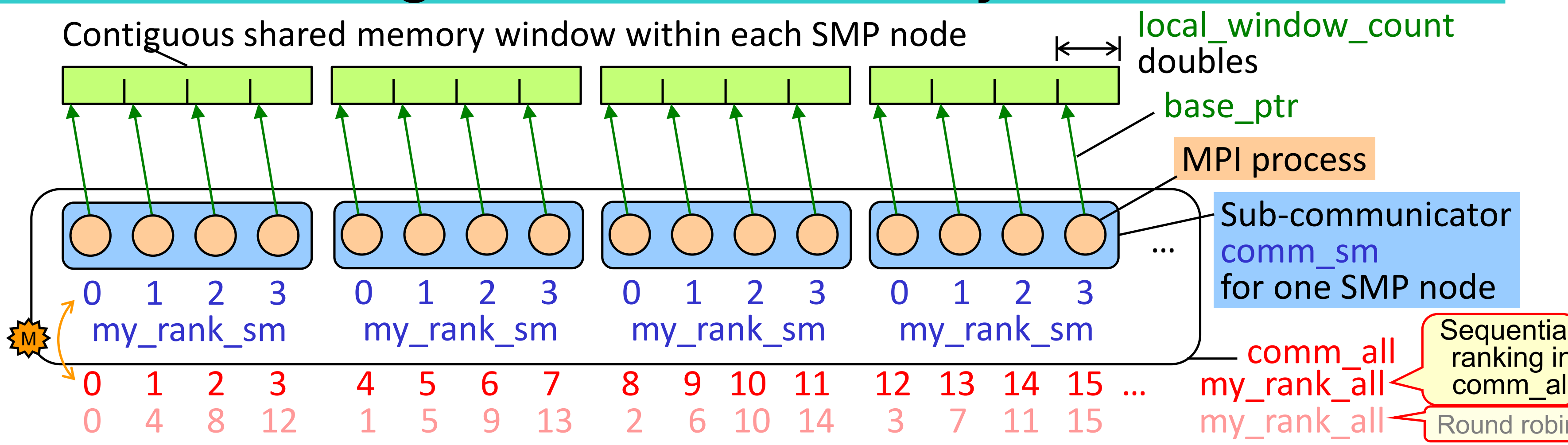


 REC → online

Skip rest of this course chapter

tour

Splitting the communicator & contiguous shared memory allocation




```
MPI_Aint /*IN*/ local_window_count=10;   double /*OUT*/  *base_ptr;
MPI_Comm  comm_all,  comm_sm;        int  my_rank_all,  my_rank_sm,  size_sm,  disp_unit;
MPI_Comm_rank (comm_all, &my_rank_all);
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0,
                           MPI_INFO_NULL,  &comm_sm);
MPI_Comm_rank (comm_sm, &my_rank_sm);  MPI_Comm_size (comm_sm, &size_sm);
disp_unit = sizeof(double);  /* shared memory should contain doubles */
MPI_Win_allocate_shared ((MPI_Aint) local_window_count*disp_unit,  disp_unit,
                             MPI_INFO_NULL,  comm_sm,  &base_ptr,  &win_sm);
```

Sequence in comm_sm as in comm_all

collective call

collective call

F In Fortran, MPI-3.1/MPI-4.0, page 339/457f, Examples 8/9.1 (and 8/9.2) show how to convert buf_ptr into a usable array a.

M This mapping is based on the ranking in comm_all.

 ● REC → online

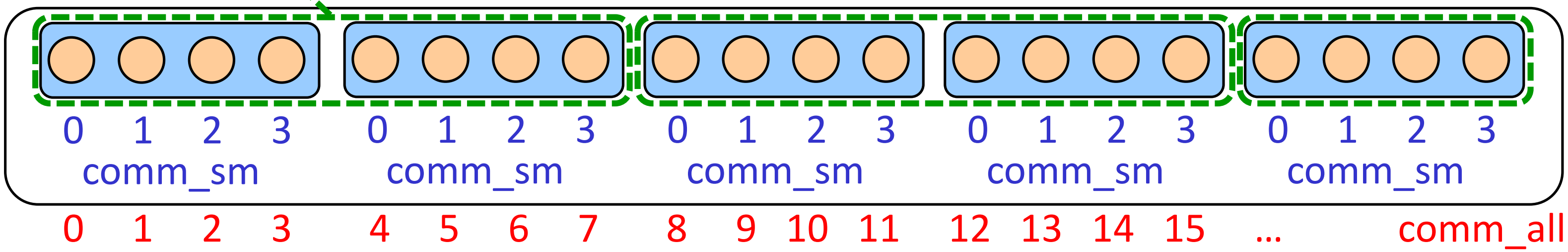
Within each SMP node – Essentials

- The allocated shared memory is contiguous across process ranks,
- i.e., the first byte of rank i starts right after the last byte of rank i-1.
- Processes can calculate remote addresses' offsets with local information only.
- Remote accesses through load/store operations,
- i.e., without MPI RMA operations (MPI_Get/Put, …)
- Although each process in comm_sm accesses the same physical memory, the virtual start address of the whole array may be different in all processes!
  → **linked lists** only with offsets in a shared array, but **not with binary pointer addresses!**

- Following slides show only the shared memory accesses, i.e., communication between the SMP nodes is not presented.

 ● REC → online

Splitting into smaller shared memory islands, e.g., NUMA nodes or sockets



- Subsets of shared memory nodes, e.g., one comm_sm on each socket with size_sm cores **(requires also sequential ranks in comm_all for each socket!)**

```
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm_large);
MPI_Comm_rank (comm_sm_large, &my_rank_sm_large); MPI_Comm_size (comm_sm_large, &size_sm_large);
MPI_Comm_split (comm_sm_large, /*color*/ my_rank_sm_large / size_sm,  0, &comm_sm);
MPI_Win_allocate_shared (..., comm_sm, ...);
```

or (size_sm_large /number_of_sockets) here 2

- Most MPI libraries have an non-standardized method to split a communicator into NUMA nodes (e.g., sockets): (see also Current support for split types in MPI implementations or MPI based libraries)
  - **OpenMPI:** choose split_type as OMPI_COMM_TYPE_NUMA
  - **HPE**: MPI_Info_create (&info); MPI_Info_set(info, "shmem_topo", "numa"); // or "socket"
    MPI_Comm_split_type(comm_all, MPI_COMM_TYPE_SHARED, 0, info, &*comm_sm*);
  - **mpich:** split_type=MPIX_COMM_TYPE_NEIGHBORHOOD, info_key= "SHMEM_INFO_KEY“ and value= "machine", "socket", "package", **"numa"**, "core", "hwthread", "pu", "l1cache", ... or "l5cache"

May not work with Intel-MPI

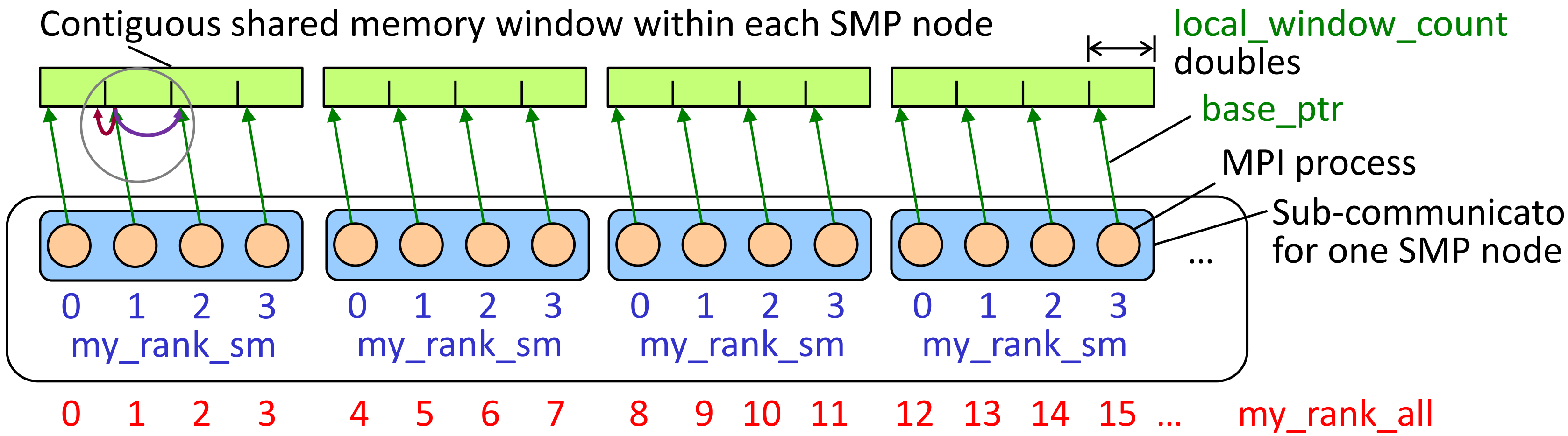
New in MPI-4.0

- Two additional standardized split types:
  - MPI_COMM_TYPE_HW_GUIDED and
  - MPI_COMM_TYPE_HW_UNGUIDED
- See also Exercise 3.

New in MPI-4.0

 ● REC → online

Shared memory access example

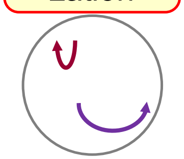



```
MPI_Aint /*IN*/ local_window_count;        double /*OUT*/  *base_ptr;
MPI_Win_allocate_shared ((MPI_Aint) local_window_count*disp_unit,  disp_unit,
                          MPI_INFO_NULL, comm_sm,  &base_ptr,  &win_sm);

MPI_Win_fence (0, win_sm);  /*local store epoch can start*/
for (i=0; i<local_window_count; i++)  base_ptr[i] = ... /* fill values into local portion */
MPI_Win_fence (0, win_sm);  /* local stores are finished, remote load epoch can start */
if (my_rank_sm > 0)              printf("left neighbor's rightmost value = %lf \n", base_ptr[-1] );
if (my_rank_sm < size_sm-1)  printf("right neighbor's leftmost value = %lf \n",
                                         base_ptr[local_window_count] );
```

Synchronization

Synchronization



Local stores

Direct load access to remote window portion

In Fortran, before and after the synchronization, on must add: CALL MPI_F_SYNC_REG (buffer) to guarantee that register copies of buffer are written back to memory, respectively read again from memory. The buffer should be declared as ASYNCHRONOUS, see course Chapter 10, slide "Fortran Problems with 1-Sided".

Such **out of bound addressing** is only available in C and Fortran. For Python, see and Exercise 2 .

Alternative: Non-contiguous shared memory

- Using info key "alloc_shared_noncontig“
- MPI library can put processes’ window portions
  - on page boundaries,
    - **(internally, e.g., only one OS shared memory segment with some unused padding zones)**
  - into the local ccNUMA memory domain + page boundaries
    - **(internally, e.g., each window portion is one OS shared memory segment)**

Pros:

- Faster local data accesses especially on ccNUMA nodes

Cons:

- Higher programming effort for neighbor accesses: MPI_WIN_SHARED_QUERY

Further reading:
Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, Rajeev Thakur:
MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory.
http://link.springer.com/content/pdf/10.1007%2Fs00607-013-0324-2.pdf

NUMA effects?
Significant impact of alloc_shared_noncontig

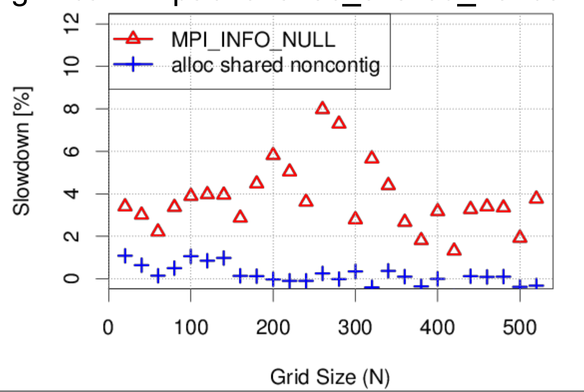
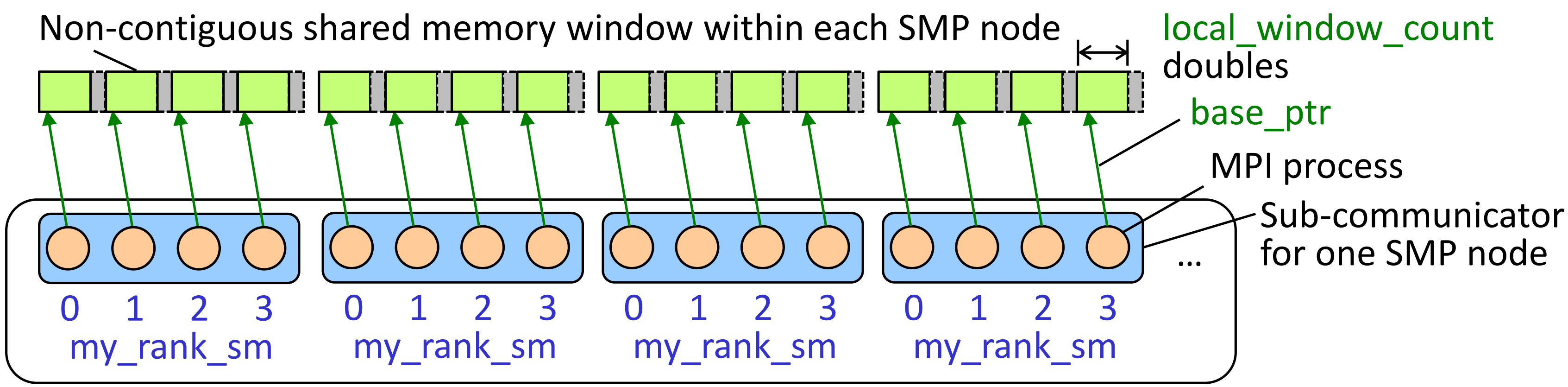


Image: Courtesy of Torsten Hoefler

Non-contiguous shared memory allocation



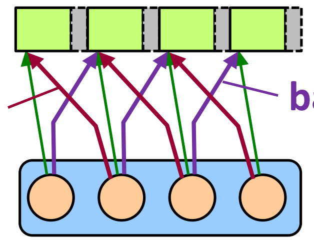

```
MPI_Aint /*IN*/ local_window_count;        double /*OUT*/  *base_ptr;
disp_unit = sizeof(double);  /* shared memory should contain doubles */
MPI_Info  info_noncontig;
MPI_Info_create (&info_noncontig);
MPI_Info_set (info_noncontig, "alloc_shared_noncontig", "true");
MPI_Win_allocate_shared ((MPI_Aint) local_window_count*disp_unit,  disp_unit,  info_noncontig,
                          comm_sm,  &base_ptr,  &win_sm );
```

Non-contiguous shared memory: Neighbor access through MPI_Win_shared_query

- Each process can retrieve each neighbor's base_ptr with calls to **MPI_Win_shared_query**

If only one process allocates the whole window → to get the base_ptr, all processes call MPI_WIN_SHARED_QUERY

- Example: only pointers to the window memory of the left & right neighbor



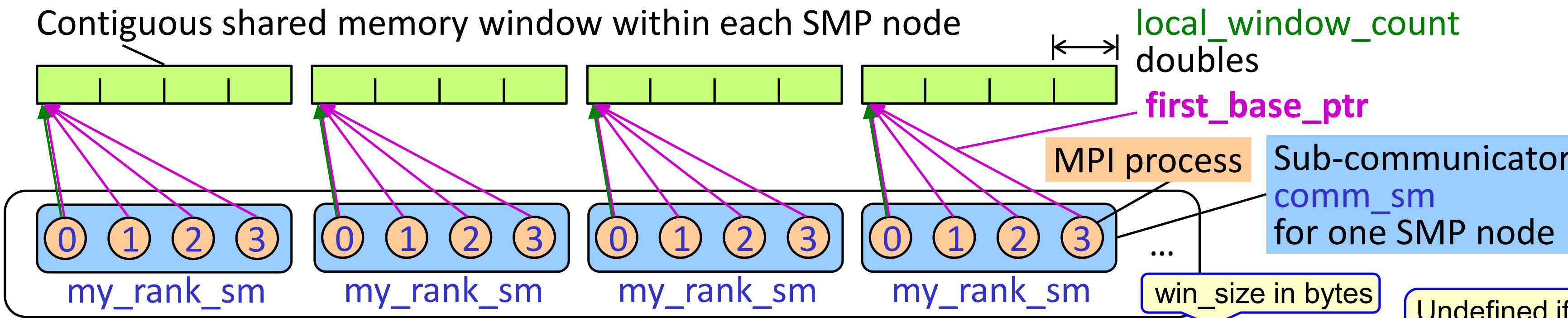
local call

```
if (my_rank_sm > 0)          MPI_Win_shared_query (win_sm, my_rank_sm – 1,
                                   &win_size_left,   &disp_unit_left,   &base_ptr_left);
if (my_rank_sm < size_sm-1) MPI_Win_shared_query (win_sm, my_rank_sm + 1,
                                   &win_size_right, &disp_unit_right, &base_ptr_right);
…
MPI_Win_fence (0, win_sm); /* local stores are finished, remote load epoch can start */
if (my_rank_sm > 0)          printf("left neighbor's rightmost value = %lf \n",
                                   base_ptr_left[ win_size_left/disp_unit_left – 1 ] );
if (my_rank_sm < size_sm-1) printf("right neighbor's leftmost value = %lf \n",
                                   base_ptr_right[ 0 ] );
```

Thanks to Steffen Weise (TU Freiberg) for testing and correcting the example codes.

 ● REC → online

Whole shared memory allocation by rank 0 in comm_sm



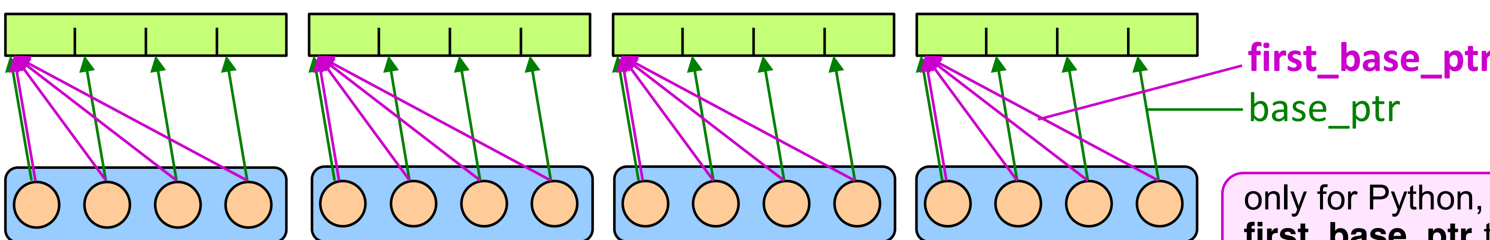
Undefined if win_size==0

if (my_rank_sm==0) win_size = local_window_count*disp_unit*_size_sm_ _else_ win_size = 0;

MPI_Win_allocate_shared (win_size, disp_unit, MPI_INFO_NULL, comm_sm, &_base_ptr_, &_win_sm_);

MPI_Win_shared_query (win_sm, /*rank=*/ 0, &_win_size_, &_disp_unit_, &_first_base_ptr_);

Describes the whole array



Python

only for Python, we use this **first_base_ptr** to define the **recv_buf array** in Exercise 2

win_size = local_window_count*disp_unit~~*size_sm~~;

MPI_Win_allocate_shared (win_size, disp_unit, MPI_INFO_NULL, comm_sm, &_base_ptr_, &_win_sm_);

MPI_Win_shared_query (win_sm, /*rank=*/ 0, &_win_size_, &_disp_unit_, **&_first_base_ptr_);**

Describes only first portion

CAUTION: Aliasing may be forbidden in your programming language, i.e., within one process, do not access the same window element through two different pointers. **Recommendation here**: use ↖ to access the _own_ window portion, and use ↖ to access _remote_ elements.

Other technical aspects with MPI_Win_allocate_shared

Caution: On some systems

- the number of shared memory windows, and
- the total size of shared memory windows

may be limited.

Some OS systems may provide options, e.g.,

- at job launch, or
- MPI process start,

to enlarge restricting defaults.

Another restriction in a low-quality MPI:
MPI_Comm_split_type
may return always
MPI_COMM_SELF

If MPI shared memory support is based on POSIX shared memory:

- Shared memory windows are located in memory-mapped /dev/shm or /run/shm
- Default: 25% or 50% of the physical memory, but a maximum of ~2043 windows!
- Root may change size with: mount –o remount,size=6G /dev/shm .

Due to default limit of context IDs in mpich

Cray XT/XE/XC (XPMEM): No limits.

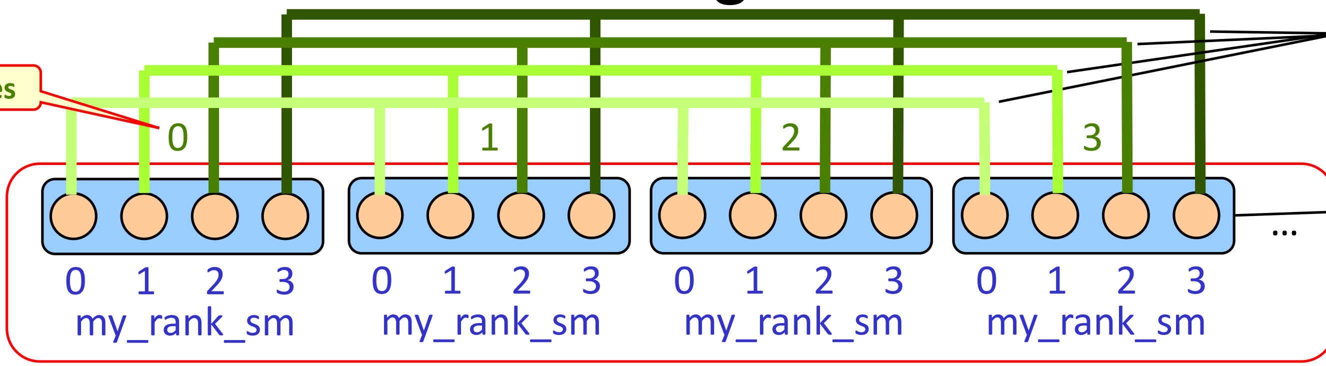
On a system without virtual memory (like CNK on BG/Q), you have to reserve a chunk of address space when the node is booted (default is 64 MB).

Thanks to Jeff Hammond and Jed Brown (ANL), Brian W Barrett (SANDIA), and Steffen Weise (TU Freiberg), for input and discussion.

skipped

Annex: Establish comm_sm, comm_nodes, comm_all, if SMPs are not contiguous within comm_orig

my_rank_nodes



comm_nodes
combining all processes with same my_rank_sm

Sub-communicator for one SMP node:
comm_sm

Establish a communicator **comm_sm** with ranks **my_rank_sm** on each SMP node

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

Input

```
MPI_Comm_split_type (comm_orig, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm);
MPI_Comm_size (comm_sm, &size_sm); MPI_Comm_rank (comm_sm, &my_rank_sm);
MPI_Comm_split (comm_orig, my_rank_sm, 0, &comm_nodes);
MPI_Comm_size (comm_nodes, &size_nodes);
if (my_rank_sm==0) {
  MPI_Comm_rank (comm_nodes, &my_rank_nodes);
  MPI_Exscan (&size_sm, &my_rank_all, 1, MPI_INT, MPI_SUM, comm_nodes);
  if (my_rank_nodes == 0) my_rank_all = 0;
}
MPI_Comm_free (&comm_nodes);
MPI_Bcast (&my_rank_nodes, 1, MPI_INT, 0, comm_sm);
MPI_Comm_split (comm_orig, my_rank_sm, my_rank_nodes, &comm_nodes);
MPI_Bcast (&my_rank_all, 1, MPI_INT, 0, comm_sm); my_rank_all = my_rank_all + my_rank_sm;
MPI_Comm_split (comm_orig, /*color*/ 0, my_rank_all, &comm_all);
```

Result: comm_nodes combines all processes with a given my_rank_sm into a separate communicator.

On processes with my_rank_sm > 0, this comm_nodes is unused because node-numbering within these comm_nodes may be different.

Exscan does not return value on the first rank, therefore

Expanding the numbering from **comm_nodes** with my_rank_sm == 0 to all new node-to-node communicators **comm_nodes**.

my_rank_nodes is not identical to the rank in comm_nodes if node sizes are not identical

Calculating **my_rank_all** and establishing global communicator **comm_all** with sequential SMP subsets.

Exercise 1: Shared memory ring communication

Exercise 1

- The following exercise is 1st based on ring-1sided-put.c / _30.f90
and 2nd on ring-1sided-put-**win-alloc**.c / _30.f90, which already includes:
  - Using MPI_Win_allocate to allocate the rcv_buf, **i.e., not yet the shared memory variant!**

C

  - Therefore in C, local rcv_buf is substituted by ***rcv_buf_ptr** – changed code lines:

```
   int snd_buf;    int *rcv_buf_ptr;
------------------------------------
   /* Allocate the window. */
   MPI_Win_allocate(&rcv_buf, sizeof(int), sizeof(int), MPI_INFO_NULL,
                    MPI_COMM_WORLD, &rcv_buf_ptr, &win);
------------------------------------
      snd_buf = *rcv_buf_ptr;
      sum += *rcv_buf_ptr;
```

Fortran

  - In Fortran, the skeleton uses C_F_POINTER – changed code lines:

```
   USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR, C_F_POINTER
------------------------------------
   INTEGER, ASYNCHRONOUS :: snd_buf
   INTEGER, POINTER, ASYNCHRONOUS :: rcv_buf !or rcv_buf(:) if it is an array
   TYPE(C_PTR) :: ptr_rcv_buf
------------------------------------
! ALLOCATE THE WINDOW.
   CALL MPI_Win_allocate(rcv_buf, rcv_buf_size, disp_unit,MPI_INFO_NULL,&
    &                    MPI_COMM_WORLD, ptr_rcv_buf, win)
! CALL C_F_POINTER(ptr_rcv_buf, rcv_buf, (/shape_of_number_of_elements/))
! rcv_buf(0:) => rcv_buf ! change lower bound to 0 (instead of default 1)
CALL C_F_POINTER(ptr_rcv_buf, rcv_buf) ! if rcv_buf is not an array
------------------------------------
snd_buf = rcv_buf
sum = sum + rcv_buf
```

if rcv_buf is an array

unchanged

Exercise 1: Shared memory ring communication

Python

```
– rcv_buf = np.empty((),dtype=np_dtype)
  win = MPI.Win.Create(memory=rcv_buf, disp_unit=rcv_buf.itemsize,
                       info=MPI.INFO_NULL, comm=comm_world)
→ win = MPI.Win.Allocate(np_dtype(0).itemsize, np_dtype(0).itemsize,
                         MPI.INFO_NULL, comm_world)
  rcv_buf = np.frombuffer(win, dtype=np_dtype)
  rcv_buf = np.reshape(rcv_buf,())
```

- And 3rd in Fortran, it is finally based on on ring-1sided-put-win-alloc-**arr**_30.f90, which declares rcv_buf as 0-based array

Fortran

– In Fortran, this **…-arr** skeleton uses C_F_POINTER for rcv_buf as an array:

```
   USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR, C_F_POINTER
 -----------------------------------------------
   INTEGER, ASYNCHRONOUS :: snd_buf
   INTEGER, POINTER, ASYNCHRONOUS :: rcv_buf(:)     <- if rcv_buf should be an array
   TYPE(C_PTR) :: ptr_rcv_buf
 -----------------------------------------------
 ! ALLOCATE THE WINDOW.
   CALL MPI_Win_allocate(rcv_buf, rcv_buf_size, disp_unit,MPI_INFO_NULL,&
    &                          MPI_COMM_WORLD, ptr_rcv_buf, win)
   CALL C_F_POINTER(ptr_rcv_buf, rcv_buf, (/1/)) !  1=length  } if rcv_buf
   rcv_buf(0:) => rcv_buf ! change lower bound to 0           } is an array
 ! CALL C_F_POINTER(ptr_rcv_buf, rcv_buf) ! if rcv_buf is not an array
 -----------------------------------------------
 snd_buf = rcv_buf(0)         } if rcv_buf is an array with lower bound 0
 sum = sum + rcv_buf(0)       }
```

- **All three steps are combined into the skeletons for the exercise on the next slide**

 ● REC → online

Exercise 1: Shared memory ring communication

- Tasks: In MPI/tasks/…
  - Use C C/Ch11/ring-1sided-put-win-alloc-shared-skel.c
    or Fortran F_30/Ch11/ring-1sided-put-win-alloc-shared-skel_30.f90
    or Python PY/Ch11/ring-1sided-put-win-alloc-shared-skel.py
  - Task A: Add **MPI_Comm_split_type** directly after MPI_Init.
    - **The ring algorithm should be executed only within the new comm_sm**
    - Therefore from there, use **comm_sm**
    - and of course also **my_rank_sm** and **size_sm** of **comm_sm**
    - Please, **be not confused**, if you are running this example **on a shared memory system**:
      In this case MPI_Comm_split_type will **not split** MPI_COMM_WORLD.
      It will return a copy of it instead. This is okay!
  - Task B: Substitute **MPI_Win_allocate** by **MPI_Win_allocate_shared**

    i.e., in C and Fortran, each process points to its own window portion
  - The skeletons are already prepared with
    - size_**world** and my_rank_**world** for **MPI_COMM_WORLD**
    - size_**sm** and my_rank_**sm** for **comm_sm**
  - And the print/write-statement already prints both my_ranks
- **(Please do not modify the MPI_Put – this will be done in Exercise 2 after the next talk i.e., ignore that the window portions are in one contiguous array)**

Exercise 2

Exercise 2: Shared memory ring communication

- Task of this exercise:
  - Use C C/Ch11/ring-1sided-**store**-win-alloc-shared-skel.c
    or Fortran F_30/Ch11/ring-1sided-**store**-win-alloc-shared-skel_30.f90
    or Python PY/Ch11/ring-1sided-**store**-win-alloc-shared-skel.py
  - Substitute **MPI_Put** by a direct assignment
    of the value of snd_buf into the rcv_buf of the right (i.e. my_rank_sm+1) neighbor
    - ***rcv_buf_ptr** (in C) and **rcv_buf(0)** (in Fortran) is the local rcv_buf
    - The rcv_buf of the right neighbor can be accessed through the word-offset **"+1"**
      in the direct assignment *(rcv_buf_ptr+(offset)) = snd_buf (in C)
      or rcv_buf(0+(offset)) = snd_buf (in Fortran)
    - In the ring, a word-offset with the value **+1** should be expressed with **(right – my_rank_sm)**, which is normally **+1**, except for the last process, where it is **–size+1**
    - Fortran: Be sure that that you add additional calls to MPI_F_SYNC_REG between both MPI_Win_fence and your direct assignment, i.e., directly before and after rcv_buf(0+(offset)) = snd_buf .
      Reason: One must prevent that the compiler may move the store to rcv_buf across the calls to MPI_Fence!

```
CALL MPI_Win_fence
…MPI_F_SYNC_REG(rcv_buf)
rcv_buf(…) = …
…MPI_F_SYNC_REG(rcv_buf)
CALL MPI_Win_fence
```

  - Problem with MPI-3.0 to MPI-4.0: The role of assertions in RMA synchronization used for direct shared memory accesses (i.e., without RMA calls) is not clearly defined!
    Implication: **MPI_Win_fence can be used, but only with assert = 0.** (State March 01, 2015)
  - **Python**: all processes shall point to the start of the **whole array**

i.e., In Python, add a call to MPI_Win_shared_query

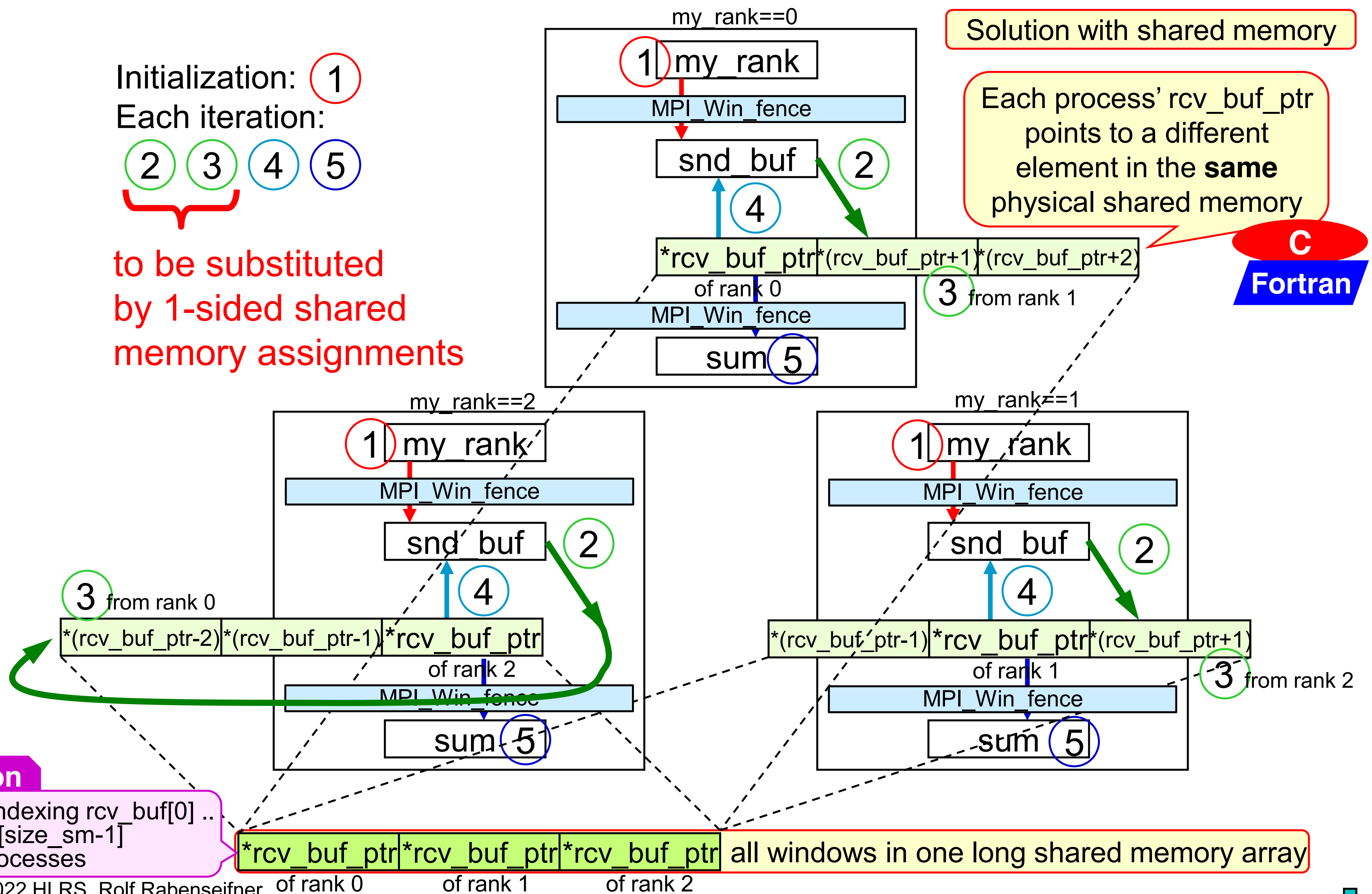
 ● REC → online

Exercise 2: Shared memory ring communication

Initialization: 1

Each iteration: 2 3 4 5

to be substituted by 1-sided shared memory assignments



Python

Same indexing rcv_buf[0] .. rcv_buf[size_sm-1] in all processes

Advanced Exercise 1b

Advanced Exercise 1b: Smaller Islands

- Task of this exercise:
  - Use C C/Ch11/ring-1sided-**put**-win-alloc-shared-subislands-skel.c
    or Fortran F_30/Ch11/ring-1sided-**put**-win-alloc-shared-subislands-skel_30.f90
    or Python PY/Ch11/ring-1sided-**put**-win-alloc-shared-subislands-skel.py
  - Split comm_sm into two comm_sm_sub
    - For example 12 processes into 2x 6 processes or 11 processes into 6+5 processes
  - For this, substitute the ______ lines
  - Compile and run: `mpirun -np 11 ./a.out | sed -e 's/World://' | sort -n`
  - Result may be

```
 0 of 11 comm_sm: 0 of 11 comm_sm_sub: 0 of 6 l/r=5/1 Sum = 15
MPI_COMM_WORLD consists of only one shared memory region
 1 of 11 comm_sm: 1 of 11 comm_sm_sub: 1 of 6 l/r=0/2 Sum = 15
 2 of 11 comm_sm: 2 of 11 comm_sm_sub: 2 of 6 l/r=1/3 Sum = 15
 3 of 11 comm_sm: 3 of 11 comm_sm_sub: 3 of 6 l/r=2/4 Sum = 15
 4 of 11 comm_sm: 4 of 11 comm_sm_sub: 4 of 6 l/r=3/5 Sum = 15
 5 of 11 comm_sm: 5 of 11 comm_sm_sub: 5 of 6 l/r=4/0 Sum = 15
 6 of 11 comm_sm: 6 of 11 comm_sm_sub: 0 of 5 l/r=4/1 Sum = 10
 7 of 11 comm_sm: 7 of 11 comm_sm_sub: 1 of 5 l/r=0/2 Sum = 10
 8 of 11 comm_sm: 8 of 11 comm_sm_sub: 2 of 5 l/r=1/3 Sum = 10
 9 of 11 comm_sm: 9 of 11 comm_sm_sub: 3 of 5 l/r=2/4 Sum = 10
 10 of 11 comm_sm: 10 of 11 comm_sm_sub: 4 of 5 l/r=3/0 Sum = 10
```

  - Maybe that your installation provides non-standardized methods to split a node with 2 CPUs into these CPUs (=NUMA domains, or SOCKETs)

 ● REC → online

Chapter 11-(1) Exercise 1: Ring with shared memory one-sided comm.

MPI/tasks/C/Ch11/solutions/ring_1sided_put_win_alloc_shared.c

C

```
int my_rank_world, size_world;
int my_rank_sm,    size_sm;
MPI_Comm comm_sm;
int snd_buf;
int *rcv_buf_ptr;
------------------------------------------------------------------------------------
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0,
                    MPI_INFO_NULL, &comm_sm);
MPI_Comm_rank(comm_sm, &my_rank_sm);
MPI_Comm_size(comm_sm, &size_sm);
if (my_rank_sm == 0)
{ if (size_sm == size_world)
  {  printf("MPI_COMM_WORLD consists of only one shared memory region\n");
  }else
  {  printf("MPI_COMM_WORLD is split into 2 or more shared memory islands\n");
} }
right = (my_rank_sm+1)          % size_sm;
left  = (my_rank_sm-1+size_sm) % size_sm;
MPI_Win_allocate_shared((MPI_Aint) sizeof(int), sizeof(int), MPI_INFO_NULL,
                        comm_sm, &rcv_buf_ptr, &win);
------------------------------------------------------------------------------------
snd_buf = my_rank_sm;
for( i = 0; i < size_sm; i++)
{
  MPI_Win_fence(0, win);
  MPI_Put(&snd_buf, 1, MPI_INT, right, (MPI_Aint) 0, 1, MPI_INT, win);
  MPI_Win_fence(0, win);
  snd_buf = *rcv_buf_ptr;
  sum += *rcv_buf_ptr;
}
```

 ● REC → online

Chapter 11-(1) Exercise 1:
Ring with shared memory one-sided comm.

Fortran

MPI/tasks/F_30/Ch11/solutions/ring_1sided_put_win_alloc_shared_30.f90

```
USE mpi_f08
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR, C_F_POINTER
----------------------------------------------------------------------
INTEGER :: my_rank_world, size_world
INTEGER :: my_rank_sm,    size_sm
TYPE(MPI_Comm)  :: comm_sm
----------------------------------------------------------------------
INTEGER, ASYNCHRONOUS :: snd_buf
INTEGER, POINTER, ASYNCHRONOUS :: rcv_buf(:) ! "(:)" because it is an array
TYPE(C_PTR) :: ptr_rcv_buf
----------------------------------------------------------------------
CALL MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, &
 &                       MPI_INFO_NULL, comm_sm)
CALL MPI_Comm_rank(comm_sm, my_rank_sm)
CALL MPI_Comm_size(comm_sm, size_sm)
IF (my_rank_sm == 0) THEN
  IF (size_sm == size_world) THEN
    write (*,*) 'comm_sm consists of only one shared memory region'
  ELSE
    write (*,*) 'comm_sm is split into 2 or more shared memory islands'
  END IF
END IF
----------------------------------------------------------------------
right = mod(my_rank_sm+1,         size_sm)
left  = mod(my_rank_sm-1+size_sm, size_sm)
----------------------------------------------------------------------
CALL MPI_Win_allocate_shared(rcv_buf_size, disp_unit, MPI_INFO_NULL, &
 &                                  comm_sm, ptr_rcv_buf, win)
CALL C_F_POINTER(ptr_rcv_buf, rcv_buf, (/1/)) ! if rcv_buf is an array
rcv_buf(0:) => rcv_buf ! change lower bound to 0
----------------------------------------------------------------------
snd_buf = my_rank_sm
DO i = 1, size_sm
----------------------------------------------------------------------
   snd_buf = rcv_buf(0)
   sum = sum + rcv_buf(0)
```

Chapter 11-(1) Exercise 1: Ring with shared memory one-sided comm.

Python

MPI/tasks/PY/Ch11/solutions/ring_1sided_put_win_alloc_shared.py

```
from mpi4py import MPI
import numpy as np
-------------------------------------------------------------------------
np_dtype = np.intc
status = MPI.Status()
-------------------------------------------------------------------------
comm_world = MPI.COMM_WORLD
my_rank_world = comm_world.Get_rank()
size_world = comm_world.Get_size()
-------------------------------------------------------------------------
comm_sm = comm_world.Split_type(MPI.COMM_TYPE_SHARED, 0, MPI.INFO_NULL)
my_rank_sm = comm_sm.Get_rank()
size_sm = comm_sm.Get_size()
if (my_rank_sm == 0):
   if (size_sm == size_world):
      print("MPI_COMM_WORLD consists of only one shared memory region")
   else:
      print("MPI_COMM_WORLD is split into 2 or more shared memory islands")
right = (my_rank_sm+1)          % size_sm
left  = (my_rank_sm-1+size_sm) % size_sm
-------------------------------------------------------------------------
# Allocate the window and use it as rcv_buf
win = MPI.Win.Allocate_shared(np_dtype(0).itemsize*1, np_dtype(0).itemsize,
                              MPI.INFO_NULL, comm_sm)
rcv_buf = np.frombuffer(win, dtype=np_dtype)
rcv_buf = np.reshape(rcv_buf, ())
-------------------------------------------------------------------------
sum = 0
snd_buf = np.array(my_rank_sm, dtype=np_dtype)
-------------------------------------------------------------------------
for i in range(size_sm):
   win.Fence(MPI.MODE_NOSTORE | MPI.MODE_NOPRECEDE)
   win.Put((snd_buf, 1, MPI.INT), right, (0, 1, MPI.INT))
   win.Fence(MPI.MODE_NOSTORE | MPI.MODE_NOPUT | MPI.MODE_NOSUCCEED)
   np.copyto(snd_buf,rcv_buf)
   sum += rcv_buf
-------------------------------------------------------------------------
print("World: {} of {} \tcomm_sm: {} of {} \tSum = {}".format(
       my_rank_world, size_world, my_rank_sm, size_sm, sum));   win.Free()
```

Only 1 element

The buffer interface is not implemented for the Win class prior to version 3.0.0.
This code will work with mpi4py 3.0.0 and above.

 ● REC → online

Chapter 11-(1) Exercise 2:
Ring with shared memory one-sided comm.

MPI/tasks/C/Ch11/solutions/ring_1sided_store_win_alloc_shared.c

C

And all fences without assertions (as long as not otherwise standardized):

```
MPI_Win_allocate_shared((MPI_Aint) sizeof(int), sizeof(int),
                        MPI_INFO_NULL, comm_sm, &rcv_buf_ptr, &win);
sum = 0;
snd_buf = my_rank_sm;

for( i = 0; i < size_sm; i++)
{
  MPI_Win_fence( /*workaround: no assertions:*/ 0, win);

  // MPI_Put(&snd_buf, 1, MPI_INT, right, (MPI_Aint) 0, 1, MPI_INT, win);
  // ... is substituted by
  //          (with offset "right-my_rank" to store into right neigbor's rcv_buf):
  *(rcv_buf_ptr+(right-my_rank_sm)) = snd_buf;

  MPI_Win_fence( /*workaround: no assertions:*/ 0, win);

  snd_buf = *rcv_buf_ptr;
  sum += *rcv_buf_ptr;
}

printf ("World: %i of %i \tcomm_sm: %i of %i \tSum = %i\n",
        my_rank_world, size_world, my_rank_sm, size_sm, sum);

MPI_Win_free(&win);
```

Chapter 11-(1) Exercise 2:
Ring with shared memory one-sided comm.

MPI/tasks/F_30/Ch11/solutions/ring_1sided_store_win_alloc_shared_30.f90

Fortran

```
USE mpi_f08
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR, C_F_POINTER
IMPLICIT NONE
------------------------------------------------------------------------------
INTEGER :: snd_buf ! no longer ASYNCHRONOUS, because no MPI_Put(snd_buf, ...)
INTEGER, POINTER, ASYNCHRONOUS :: rcv_buf(:)
TYPE(C_PTR) :: ptr_rcv_buf
------------------------------------------------------------------------------
sum = 0
snd_buf = my_rank_sm
DO i = 1, size_sm
   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
   CALL MPI_WIN_FENCE(0, win)   ! Workaround: no assertions
   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
   rcv_buf(0+(right-my_rank_sm)) = snd_buf
   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
   CALL MPI_WIN_FENCE(0, win)   ! Workaround: no assertions
   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
!  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
------------------------------------------------------------------------------
   snd_buf = rcv_buf(0)
   sum = sum + rcv_buf(0)
END DO
WRITE(*,*) 'World:', my_rank_world,' of ',size_world, &
 &         'comm_sm:',my_rank_sm,   ' of ',size_sm,    '; Sum =', sum
```

Needed to prevent code movement of load/store to rcv_buf across the fences in current and next loop iteration.

New: Needed to prevent movement of rcv_buf(…) =snd_buf across nearest fences

No longer needed, because the access to **snd_buf** is no longer a nonblocking MPI call. Now, it is a directly executed expression.

Chapter 11-(1) Exercise 2: Ring with shared memory one-sided comm.

Python

MPI/tasks/PY/Ch11/solutions/ring_1sided_store_win_alloc_shared.py

```
np_dtype = np.intc

# Allocate the window.
win = MPI.Win.Allocate_shared(np_dtype(0).itemsize*1, np_dtype(0).itemsize,
                              MPI.INFO_NULL, comm_sm)

# The buffer interface is not implemented
# for the Win class prior to version 3.0.0.
# This code will work with mpi4py 3.0.0 and above.
# We define an memory object with the rank 0 process' base address and
# length up to the last element of the shared memory allocated by
# Allocate_shared.
(buf_zero, itemsize) = win.Shared_query(0)
assert itemsize == MPI.INT.Get_size()
assert itemsize == np_dtype(0).itemsize
buf = MPI.memory.fromaddress(buf_zero.address, size_sm*1*itemsize)
# We use this memory object and consider it as an numpy ndarray
rcv_buf = np.frombuffer(buf, dtype=np_dtype)

sum = 0
snd_buf = np.array(my_rank_sm, dtype=np_dtype)

for i in range(size_sm):
    win.Fence() # workaround: no assertions

    # MPI_Put(&snd_buf, 1, MPI_INT, right, (MPI_Aint) 0, 1, MPI_INT, win);
    #    ... is substited by:
    rcv_buf[right] = snd_buf

    win.Fence() # workaround: no assertions

    snd_buf = rcv_buf[my_rank_sm]
    sum += rcv_buf[my_rank_sm]
```

Only 1 rcv_buf element per process

Number of processes

Only 1 rcv_buf element per process

 REC → online