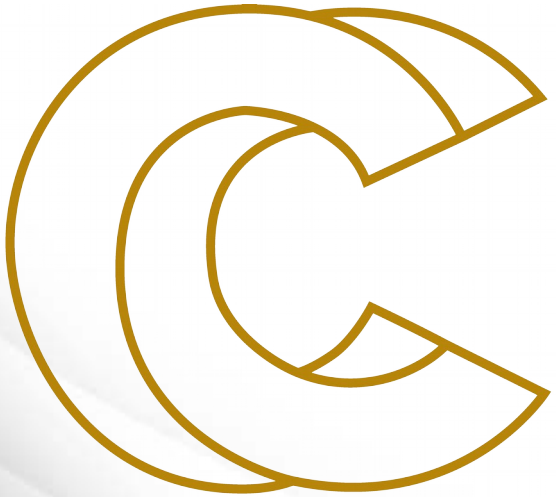
# Advanced MPI: MPI + OpenMP

SLING



EURO

Leon Kos

*University of Ljubljana, FME, LECAD lab*

# Acknowledgments


- *Shared MPI-3 memory* is Chapter 11 from *Introduction to the Message Passing Interface (MPI)* course by Rolf Rabenseifner from University of Stuttgart and High-Performance Computing-Center Stuttgart (HLRS)
- The MPI-1.1 part of this course is partially based on the MPI course developed by the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh.
- Thanks to the EPCC, especially to Neil MacDonald, Elspeth Minty, Tim Harding, and Simon Brown.
- Course Notes and exercises of the EPCC course can be used together with this slides.
- The MPI-2.0 part is partially based on the MPI-2 tutorial at the MPIDC 2000 by Anthony Skjellum, Purushotham Bangalore, Shane Hebert (High Performance Computing Lab, Mississippi State University, and Rolf Rabenseifner (HLRS)
- Some MPI-3.0 detailed slides are provided by the MPI-3.0 ticket authors, chapter authors, or chapter working groups, Richard Graham (chair of MPI-3.0), and Torsten Hoefler (additional example about new one-sided interfaces)
- Thanks to Dr. Claudia Blaas-Schenner from TU Wien (Vienna) and many other trainers and participants for all their helpful hints for optimizing this course over so many years.

# Motivation

- Computer systems in High-performance computing (HPC) feature a hierarchical hardware design (multi-core nodes connected via a network)
- Pure OpenMP performs better than pure MPI within node is a necessity to have hybrid code better than pure MPI across node
- Whether the hybrid code performs better than MPI code depends on whether the communication advantage outcomes the thread overhead, etc. or not.

# MPI + threading methods

- MPI + OpenMP
  - Often better one process per NUMA domain (not per ccNUMA node)
  - (Perfect) compiler support for threading
  - Called libraries must be thread-safe
- MPI + MPI-3 shared memory
  - Efficient placement of MPI processes on ccNUMA nodes *is not trivial*
    - Hard for applications with unstructured grids
    - Possible solution: Domain decomposition on core level.

      Then recombining for (cc)NUMA domains.
    - See in Chapter 9. Virtual topologies, (3) Optimization through reordering
  - Major usecase:
    - Replicated application data in one shared memory window

      per CPU or per ccNUMA nodeo

# MPI rules with OpenMP / Automatic SMP-parallelization

SLING EURO

- Special MPI-2 Init for multi-threaded MPI processes:

```
int MPI_Init_thread( int * argc, char ** argv[],
          int thread_level_required,
          int * thread_level_provided);
int MPI_Query_thread(int * thread_level_provided);
int MPI_Is_main_thread(int * flag);
```

- REQUIRED values (increasing order):
  - **MPI_THREAD_SINGLE:** **Only one thread will execute**
  - **MPI_THREAD_FUNNELED:** **Only master thread will make MPI-calls**
  - **MPI_THREAD_SERIALIZED:** **Multiple threads may make MPI-calls, but only one at a time**
  - **MPI_THREAD_MULTIPLE:** **Multiple threads may call MPI, with no restrictions**
- returned `provided` may be other than REQUIRED by the application

# MPI_THREAD_FUNNLED

**Safest (easiest) to use MPI_THREAD_FUNNLED**
See https://bit.ly/37w5g90 notebook example

- Fits nicely with most OpenMP models
  - Expensive loops parallelized with OpenMP
  - Communication and MPI calls between loops
- Eliminates need for true “thread-safe” MPI
- Parallel scaling efficiency may be limited (Amdahl’s law) by MPI_THREAD_FUNNLED approach
- Moving to MPI_THREAD_MULTIPLE does come at a performance price (and programming challenge)

## Hybrid MPI+OpenMP Masteronly Style

Masteronly MPI only outside of parallel regions

```
for (iteration ....) {
 #pragma omp parallel
  numerical code
  /*end omp parallel */
  /* on master thread only */
  MPI_Send (original data to  halo areas in
other SMP nodes)
  MPI_Recv (halo data from the neighbors)
} /*end for loop
```

- Advantages
  - No message passing inside of the SMP nodes
  - No topology problem
- Problems
  - All other threads are sleeping while master thread communicates!
  - Which inter-node bandwidth?
  - MPI-lib must support at least MPI_THREAD_FUNNELED

# Calling MPI inside of OMP MASTER

- Inside of a parallel region, with "**OMP MASTER**"
- Requires MPI_THREAD_FUNNELED,
  i.e., only master thread will make MPI-calls
- **Caution:** There isn't any synchronization with "OMP MASTER"!
  Therefore, "**OMP BARRIER**" normally necessary to guarantee, that data or buffer space from/for other threads is available before/after the MPI call!

```
!$OMP BARRIER    #pragma omp barrier
!$OMP MASTER     #pragma omp master
        call MPI_Xxx(...)          MPI_Xxx(...);
!$OMP END MASTER
!$OMP BARRIER    #pragma omp barrier
```

- But this implies that all other threads are sleeping!
- The additional barrier implies also the necessary cache flush!

# … the barrier is necessary – example with MPI_Recv

```
!$OMP PARALLEL
!$OMP DO
      do i=1,1000
      a(i) = buf(i)
      end do
!$OMP END DO NOWAIT
!$OMP BARRIER
!$OMP MASTER
      call MPI_RECV(buf,...)
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO
      do i=1,1000
      c(i) = buf(i)
      end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

```
#pragma omp parallel
{
#pragma omp for nowait
      for (i=0; i<1000; i++)
            a[i] = buf[i];

#pragma omp barrier
#pragma omp master
            MPI_Recv(buf,...);
#pragma omp barrier

#pragma omp for nowait
      for (i=0; i<1000; i++)
            c[i] = buf[i];

}
/* omp end parallel */
```

No barrier inside

Barriers needed to prevent data races

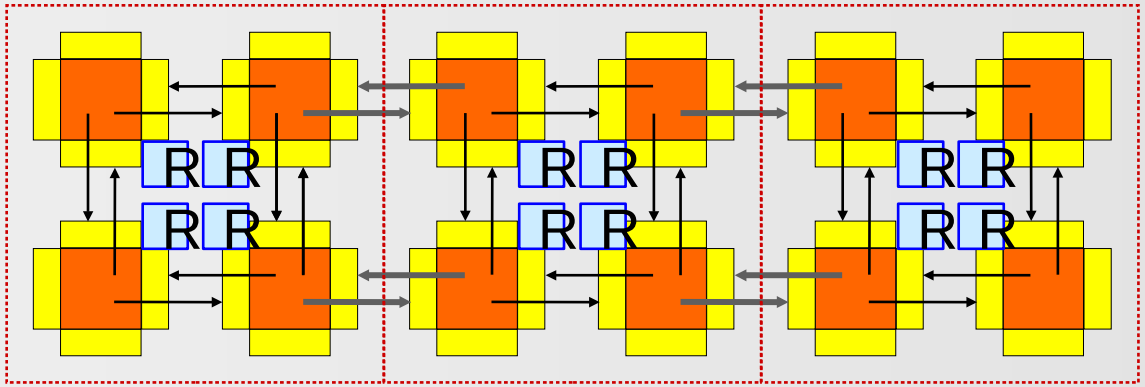
# MPI-3 shared memory

- 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!

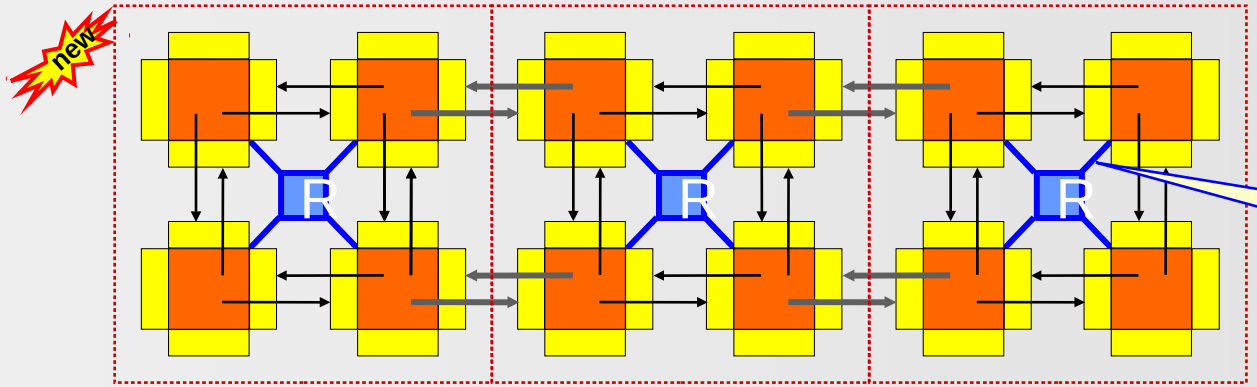
    (First touch rule: a memory page is pinned to the physical memory of the processor that first writes a byte into the page)

# MPI-3 shared memory (cont.)




R = Replicated data in each MPI process

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




R = Shared memory
replicated data
only once within each SMP node

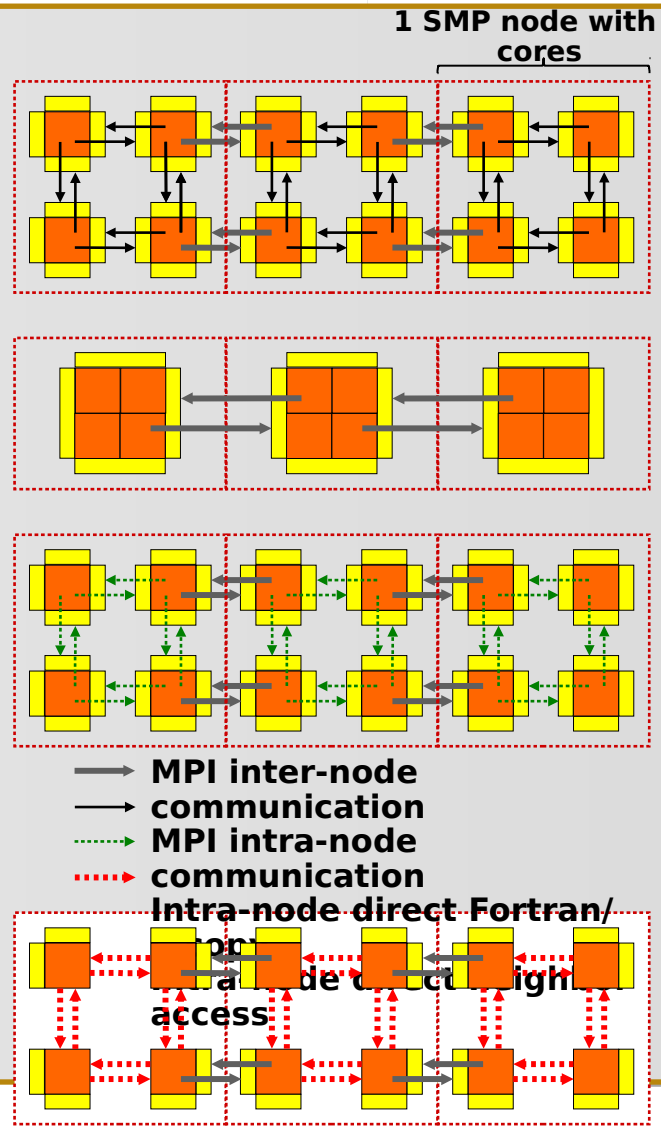
Direct loads & stores, no library calls

**Using MPI shared memory methods**

MPI-3.0 shared memory can be used to **significantly reduce the memory needs** for **replicated data**.

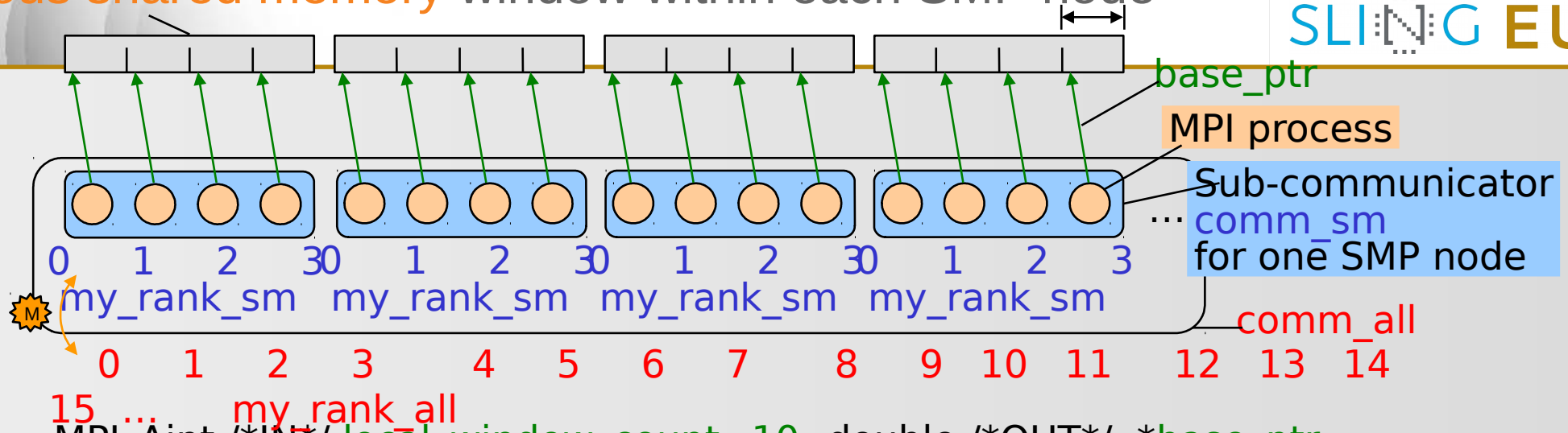
# Hybrid shared/cluster programming models

- 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 communicated 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



# Splitting the communicator & contiguous shared memory allocation

## Contiguous shared memory window within each SMP node




```
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 (local_window_count*disp_unit, disp_unit,
MPI_INFO_NULL,
                         comm_sm,  &base_ptr,  &win_sm);
```

Sequence in comm_sm as in comm_all

F In Fortran, MPI-3.1, page 339, Examples 8.1 (and 8.2) show how to convert buf_ptr into a usable array a.

M This mapping is based on a sequential ranking of the SMP nodes in comm_all.

# 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.

# Exercise 1: Shared memory ring communication

Exercise 1

- We use ring-1sided-put-**win-alloc**.c / _30.f90 as variant of ring-1sided-put.c / _30.f90
  - Using MPI_Win_allocate to allocate the rcv_buf

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, it uses C_F_POINTER – changed code lines:

```
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
  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 (cont.)

- Task of this exercise: C Fortran
  - Add **MPI_Comm_split_type** directly after MPI_Init. From there, use **comm_sm**
    - and of course also **my_rank** and **size** of **comm_sm**
  - Substitute **MPI_Win_allocate** by **MPI_Win_allocate_shared**
  - Use C C/Ch11/ring-1sided-[redacted]win-alloc-shared-skel.c
    or Fortran F_30/Ch11/ring-1sided-[redacted]t-win-alloc-shared-skel_30.f90
  - They 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

# Exercise 1: Shared memory ring communication (cont.)

Please **stay here in the main room** while you do this exercise

## During the Exercise

And have fun with this short exercise

Please do not look at the solution before you finished this exercise,

otherwise,

90% of your learning outcome may be lost

**As soon as you finished the exercise**,

please **go to your breakout room**

and continue your discussions with your fellow learners:

*Please start with assert==0!*

*Please go **already to your break out room** as soon as you program works*

*with assert == 0.*

*You may check and discuss as group, which assertions to apply*

*and which ones are forbidden*

# Conclusions

- MPI+OpenMP reduces communication overhead and does cheap load balancing
  - No intra-node communication
  - Longer messages between nodes and fewer parallel links may imply better bandwidth
  - Application developer can split the load-balancing issues between course-grained MPI and fine-grained OpenMP
- MPI+3.0 shared memory may be helpful to save memory
  - Thread-safety is not needed for libraries.
  - No reduction of MPI ranks

## Solution to Exercise 1

# 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("comm_sm consists of only one shared memory region\n");
  }else
  { printf("comm_sm 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(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++)
```

```
  snd_buf = *rcv_buf_ptr;
  sum += *rcv_buf_ptr;
```

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

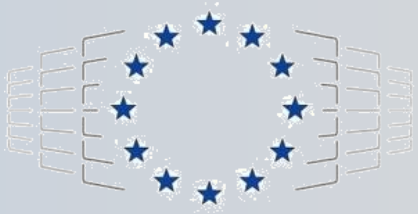
MPI/tasks/F_30/Ch11/solutions/ring-1sided-put-win-alloc-shared-30.f90

Fortran

```
USE mpi_f08
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
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)
```

SLING EURO

# Thanks!



EuroHPC
Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro