
Introduction to the Message Passing Interface (MPI)

Rolf Rabenseifner
rabenseifner@hlrs.de

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

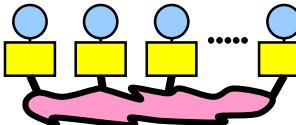
(for MPI-2.1, MPI-2.2, MPI-3.0, MPI-3.1, and MPI-4.0)



Outline

1. MPI Overview

- one program on several processors
- work and data distribution



Fast navigation
→ click on them

beginners

Lect. Exe.

[2.4, 2.7]
slides 9–...



2. Process model and language bindings

- starting several MPI processes

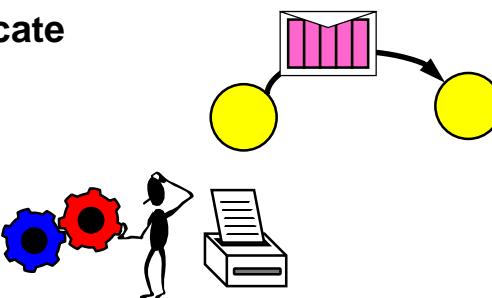
`MPI_Init()`
`MPI_Comm_rank()`

[2.6, 17.1, 6.4.1, 8.7-8]
slides 53–...

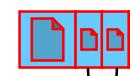


3. Messages and point-to-point communication

- the MPI processes can communicate

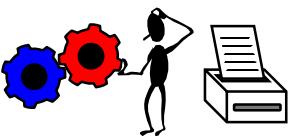


[3.1-3.6, 8.6]
slides 77–...



4. Nonblocking communication

- to avoid idle time, deadlocks and serializations



[3.7, 3.10]
slides 105–...



[...] = MPI-3.1 chapter

Exercises → slide 17



There is not one clear thread running through all the chapters.
But there are three threads:

- beginners' topics,
- intermediate,
- advanced

But parts of all three areas may be needed for your application in HPC (high performance computing)

Outline

5. The New Fortran Module mpi_f08



[17.1]
slides 141–...

advanced
Lect. Exe.



beginners

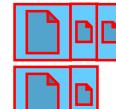
6. Collective communication

- (1) e.g., broadcast
- (2) e.g., nonblocking collectives, neighborhood communic.



[5]
slides 161–...
slides 185–...

beginners



advanced

7. Error handling

- error handler, codes, and classes



[2.8, 8.3-5, 11.6, 13.7-8, 17.2.6]
slides 201–...

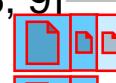


beginners

8. Groups & Communicators, Environment Management

- (1) MPI_Comm_split, intra- & inter-communicators
- (2) Re-numbering on a cluster, collective communication on inter-communicators, info object, naming & attribute caching, implementation information, Sessions Model

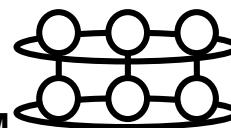
intermed.



advanced

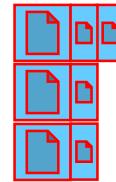
9. Virtual topologies

- (1) A multi-dimensional process naming scheme
- (2) Neighborhood communication + MPI_BOTTOM
- (3) Optimization through reordering



[7, 3.11]
slides 241–...
slides 273–...
slides 285–...

intermediate



advanced

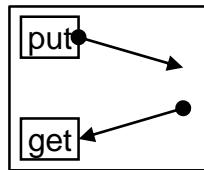
... whereas course is sorted by beginners / intermediate / advanced

[...] = MPI-3.1 chapter

Handout is sorted by content ...

Outline

10. One-sided Communication
- Windows, remote memory access (RMA)
 - Synchronization



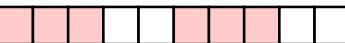
[11]
slides 321–...



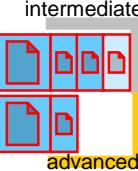
11. Shared Memory One-sided Communication
- (1) `MPI_Comm_split_type` & `MPI_Win_allocate_shared`
Hybrid MPI and MPI shared memory programming
 - (2) MPI memory models and synchronization rules

[11.2.3, 6.4.2]
slides 361–...

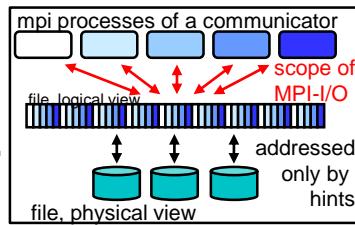


12. Derived datatypes
- 
- (1) transfer any combination of typed data
 - (2) advanced features, alignment, resizing

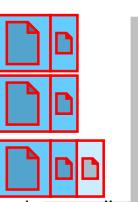
[4, 3.10, 2.5.8]
slides 425–...
slides 453–...



13. Parallel File I/O
- (1) Writing and reading a file in parallel
 - (2) Fileviews
 - (3) Shared Filepointers, Collective I/O ...



[13]
slides 473–...
slides 490–...
slides 511–...



14. MPI and Threads
- e.g., hybrid MPI and OpenMP

[12.4]
slides 533–...



[...] = MPI-3.1
chapter

Outline

15. Probe, Persistent Requests, Cancel	[3.8, 3.9] slides 541–...	advanced 
16. Process Creation and Management <ul style="list-style-type: none">– Spawning additional processes– Singleton MPI_INIT– Connecting two independent sets of MPI processes	[10] slides 553–...	
17. Other MPI features [1, 2, 12.1-3, 14, 15, 16, 17.2, A, B]	slides 569–...	
18. Best practice <ul style="list-style-type: none">– Parallelization strategies (e.g. Foster's Design Methodology)– Performance considerations– Pitfalls	[] slides 577–...	intermediate 
Summary	slides 593	
Appendix	slides 597	

[...] = MPI-3.1
chapter

Acknowledgments

- 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 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.
- Thanks to Tobias Haas from HLRS for his Python binding of the exercises. Thanks to Claudia Blaas-Schenner and David Fischak from TU Wien (Vienna) for their additional hints on the Python bindings.
Additional background was a first draft from the HiDALGO project at HLRS.

Information about MPI



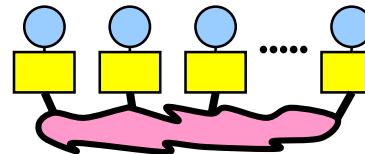
- **MPI: A Message-Passing Interface Standard**, Version 4.0 (June 9, 2021) (pdf & printed hardcover book [MPI-3.1 only] → online via [www.mpi-forum.org](http://www mpi-forum org))
- Marc Snir and William Gropp et al.: **MPI: The Complete Reference**, 1998. (**outdated**)
- William Gropp, Ewing Lusk and Anthony Skjellum:
Using MPI: Portable Parallel Programming With the Message-Passing Interface.
MIT Press, 3rd edition, Nov. 2014 (336 pages, ISBN 9780262527392), and
William Gropp, Torsten Hoefler, Rajeev Thakur and Ewing Lusk:
Using Advanced MPI: Modern Features of the Message-Passing Interface.
MIT Press, Nov. 2014 (392 pages, ISBN 9780262527637).
- Peter S. Pacheco: **Parallel Programming with MPI**. Morgan Kaufmann Publishers, 1997 (*very good introduction, can be used as accompanying text for MPI lectures*).
- Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti: **Parallel Programming with MPI**. Historical MPI course notes from EPCC.
http://www.archer.ac.uk/training/course-material/2014/10/MPI_UCL/Notes/MPP-notes.pdf
- All MPI standard documents and errata via [www.mpi-forum.org](http://www mpi-forum org)
- http://en.wikipedia.org/wiki/Message_Passing_Interface (English)
http://de.wikipedia.org/wiki/Message_Passing_Interface (German)
- **Tools:** see VI-HPS (Virtual Institute – High Productivity Supercomputing) <https://www.vi-hps.org/>
Tools Guide: <https://www.vi-hps.org/cms/upload/material/general/ToolsGuide.pdf> & [training events](#)
- **Python:** See [MPI for Python \(mpi4py.github.io\)](https://mpi4py.github.io), and [MPI for Python 3.1.1 documentation \(mpi4py.readthedocs.io\)](https://mpi4py.readthedocs.io), and [The API reference \(mpi4py.github.io/apiref/index.html\)](https://mpi4py.github.io/apiref/index.html)

For private notes

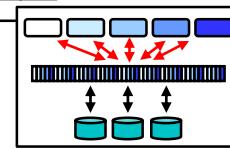
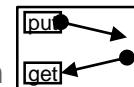
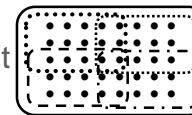
Chap.1 MPI Overview

1. MPI Overview

- one program on several processors
 - work and data distribution
 - the communication network
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module mpi_f08
6. Collective communication
7. Error Handling
8. Groups & communicators, environment management
9. Virtual topologies
10. One-sided communication
11. Shared memory one-sided communication
12. Derived datatypes
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

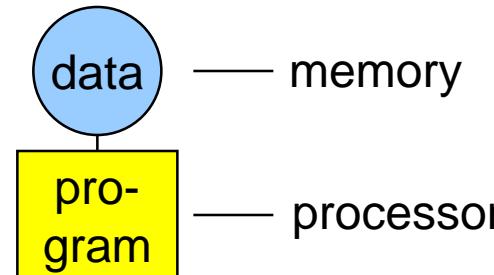


MPI_Init()
MPI_Comm_rank()

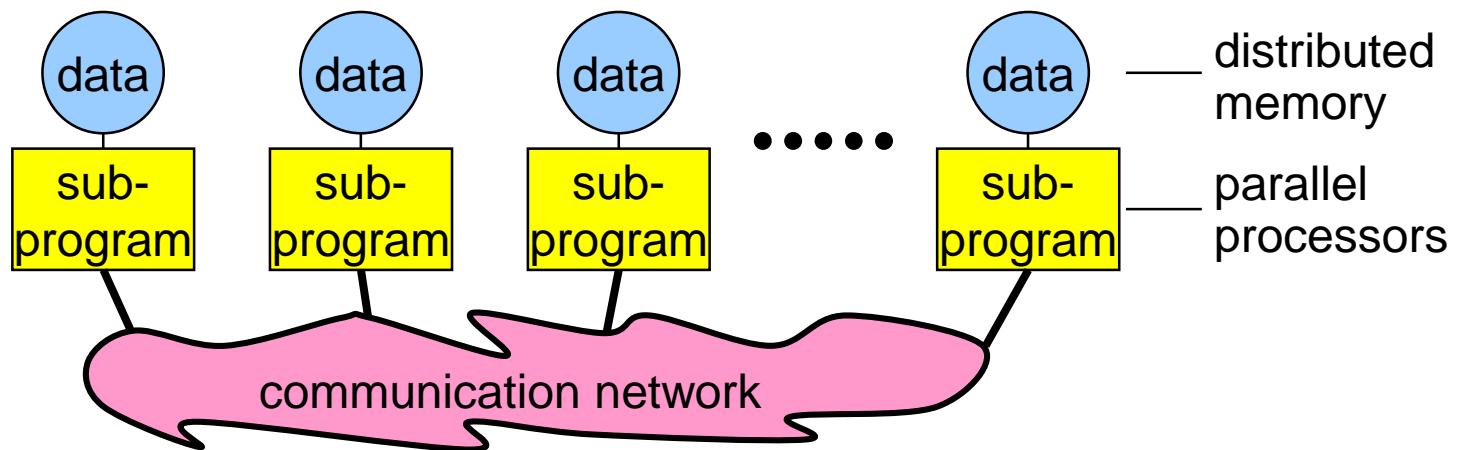


The Message-Passing Programming Paradigm

- Sequential Programming Paradigm

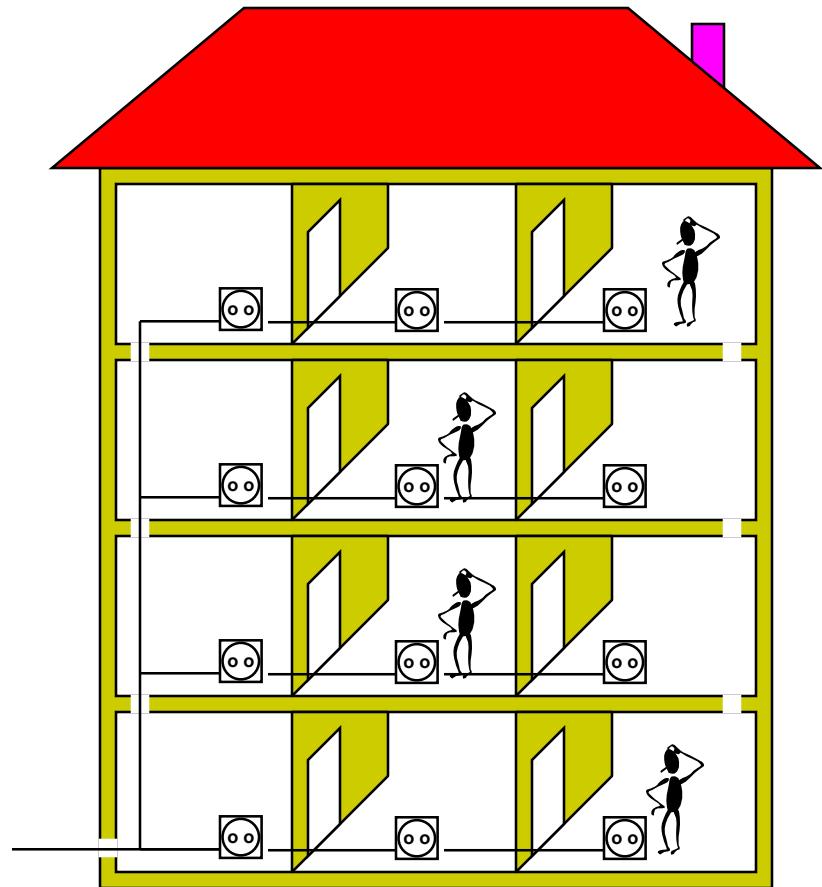


- Message-Passing Programming Paradigm



Analogy: Electric Installations in Parallel

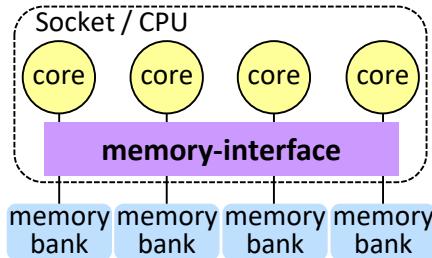
- MPI sub-program
= work of one electrician
on one floor
- data
= the electric installation
- MPI communication
= real communication
to guarantee that the wires
are coming at the same
position through the floor



Parallel hardware architectures



shared memory



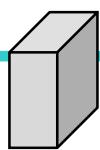
Socket/CPU

→ memory interface

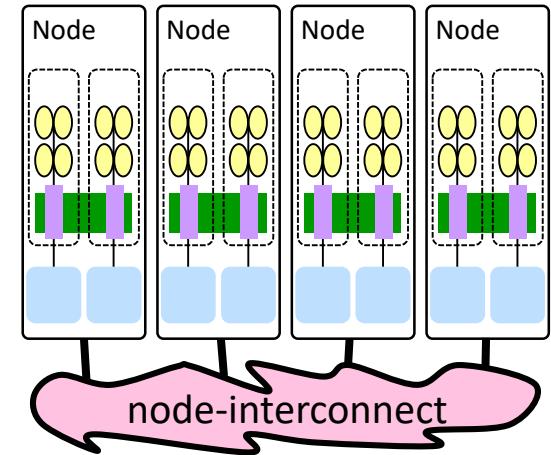
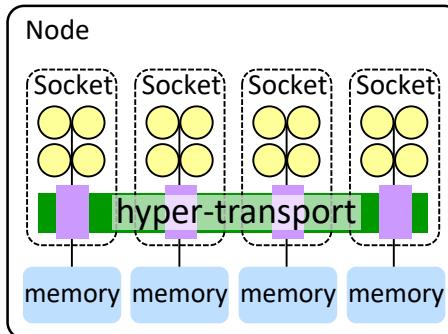
UMA (uniform memory access)

SMP (symmetric multi-processing)

All cores connected to all memory banks with same speed



distributed memory



Node

→ hyper-transport

ccNUMA (cache-coherent non-uniform
memory access)

Shared memory programming is possible

Performance problems:

- Threads should be **pinned** to the physical sockets
- First-touch** strategy is needed to minimize remote memory access

Cluster

→ node-interconnect

NUMA (non-uniform memory access)

!! fast access only on its own memory !!

Many programming options:

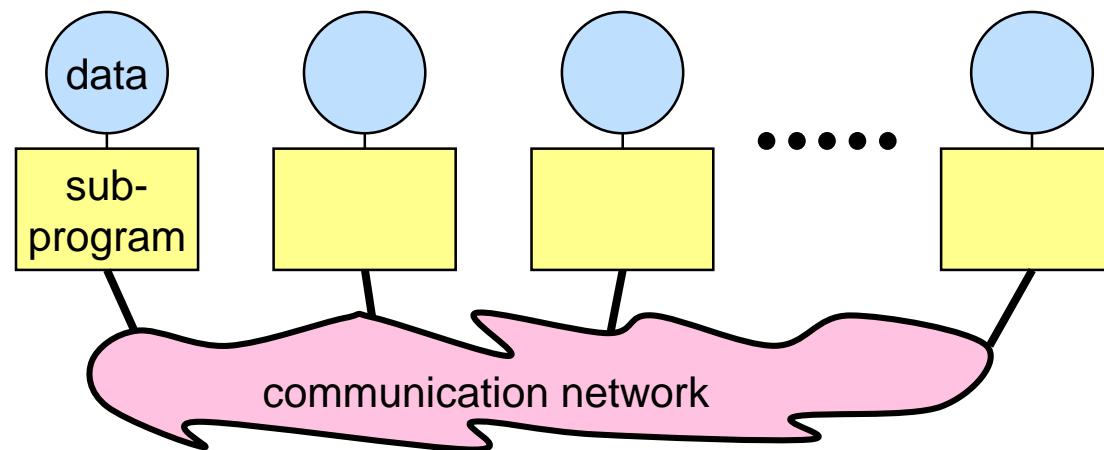
- Shared memory / symmetric multi-processing inside of each node
- distributed memory parallelization on the node interconnect
- Or simply one MPI process on each core

Shared memory programming with OpenMP

MPI works everywhere

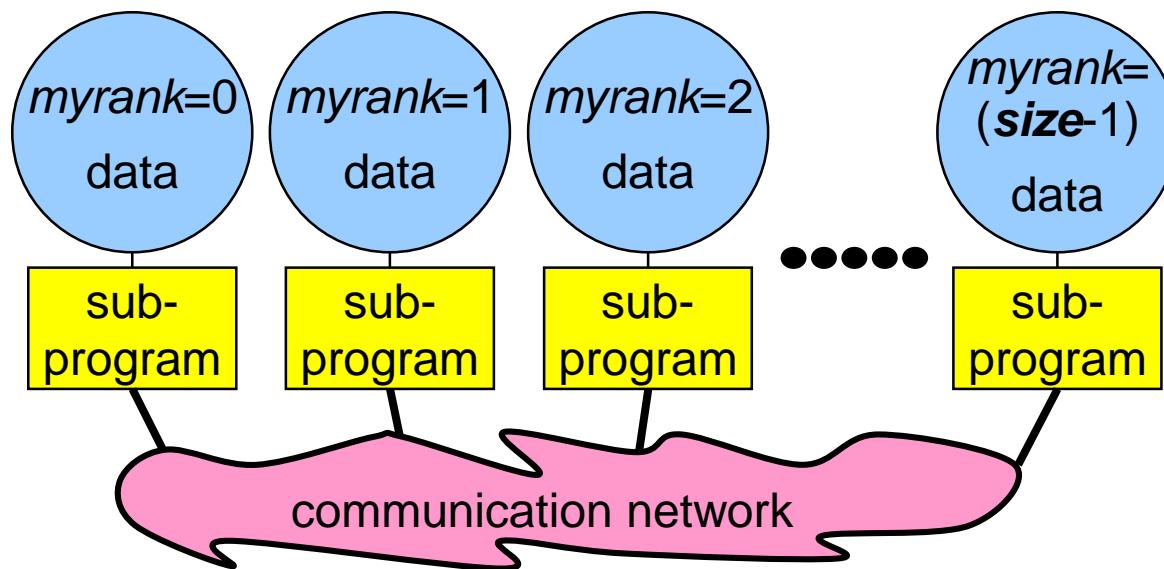
The Message-Passing Programming Paradigm

- Each processor in a message passing program runs a ***sub-program***:
 - written in a conventional sequential language, e.g., C, Fortran, or Python
 - typically the same on each processor (SPMD),
 - the variables of each sub-program have
 - **the same name**
 - **but different locations (distributed memory) and different data!**
 - **i.e., all variables are private**
 - communicate via special send & receive routines (***message passing***)



Data and Work Distribution

- the value of ***myrank*** is returned by special library routine
- the system of ***size*** processes is started by special MPI initialization program (mpirun or mpiexec)
- all distribution decisions are based on ***myrank***
- i.e., which process works on which data



What is SPMD?

- **S**ingle **P**rogram, **M**ultiple **D**ata
 - Same (sub-)program runs on each processor
-
- MPI allows also MPMD, i.e., **M**ultiple **P**rogram, ...
 - but some vendors may be restricted to SPMD
 - MPMD can be emulated with SPMD

Emulation of Multiple Program (MPMD), Example

- main(int argc, char **argv){
 if (myrank < /* process should run the ocean model */)
 {
 ocean(/* arguments */);
 } else {
 weather(/* arguments */);
 }
}
- PROGRAM
 IF (myrank < ...) THEN !! process should run the ocean model
 CALL ocean (some arguments)
 ELSE
 CALL weather (some arguments)
 ENDIF
 END
- if (myrank <): # process should run the ocean model
 ocean(...)
else:
 weather(...)

example for usage of
sub-groups of processes (\rightarrow Ch. 8)

```

#include <stdio.h> first-example.c Compiled, e.g., with: mpicc first-example.c
#include <mpi.h> Started, e.g., with: mpiexec -n 4 ./a.out
int main(int argc, char *argv[])
{
    int n;    double result;    application-related data
    int my_rank, num_procs;   MPI-related data
MPI_Init(&argc, &argv); Now, each process knows who it is:
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); number my_rank out of num_procs processes
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

if (my_rank == 0) reading the application data n from stdin only
{ printf("Enter the number of elements (n): \n"); by process 0
    scanf("%d", &n); process 0 is sender, all other
} processes are receivers
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD); broadcasting the content of variable n in process 0
into variables n in all other processes

result = 1.0 * my_rank * n; doing some application work in each process
printf("I am process %i out of %i handling the %i part of n=%i elements, result=%f\n",
       my_rank, num_procs, my_rank, n, result);

if (my_rank != 0) send to process 0
{ MPI_Send(&result, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD); sending some results from
} all processes (except 0) to process 0
else Process 0: receiving all these messages and, e.g., printing them
{ int rank;
  printf("I'm proc 0: My own result is %f \n", result);
  for (rank=1; rank<num_procs; rank++)
  {
    MPI_Recv(&result, 1, MPI_DOUBLE, rank, 99, receiving the message from process rank
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("I'm proc 0: received result of
          process %i is %f \n", rank, result);
  }
}
MPI_Finalize();

```

Then, this code is running 4 times in parallel !

Enter the number of elements (n): 100 input/output

I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0

I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0

I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0

I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0

I'm proc 0: My own result is 0.0

I'm proc 0: received result of process 1 is 100.0

I'm proc 0: received result of process 2 is 200.0

I'm proc 0: received result of process 3 is 300.0

well sorted output from process 0

Exercise 1

Please run this first example on your exercise system:

Already done as part of the course preparation

- Download **MPI31.tar.gz** or **MPI31.zip** from
https://fs.hlr.de/projects/par/par_prog_ws/practical/README.html
 or directly
https://fs.hlr.de/projects/par/par_prog_ws/practical/MPI31.tar.gz or [.zip](#)
- **C** `tar -xzf MPI31.tar.gz` (or equivalent commands on Windows)
Python `cd MPI/tasks/PY/Ch1`
- **Fortran** `cd MPI/tasks/F_30/Ch1`
- Initialize your compile and MPI environment,
 e.g., module load gnu openmpi (on my system ☺)
- `ls -l` → `first-example.c` or `first-example_30.f90` or `first-example.py`
- **C** `mpicc first-example.c`
Python `mpirun -np 4 ./a.out`
- **Fortran** `mpif90 first-example_30.f90`

Basis for the entire course

Caution: OpenMPI is an MPI library, whereas OpenMP is a shared memory parallel programming model

- **Python** `mpirun -np 4 python3 first-example.py` (no compilation, parallel start of the interpreter)
 - (or equivalent commands on your system)
 - As input, you may choose: 100 → output should be similar to previous slide)
 - Look at the sequence of the output lines for several runs with 4 to 10 processes

That's all, done.

During the Exercise (15 min.)



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



Important: To solve the exercises please **use previous slides and the provided .c/.f90 files**

(or in rare cases also the MPI standard)

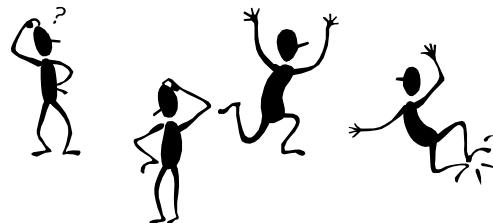
but **no google search on the web** – otherwise you are too slow



Please do not look at the next slide (=solution) before you finished this exercise,
otherwise in many exercises,
90% of your learning outcome may be lost



**As soon as you finished the exercise,
please go to your breakout room
and get to know your fellow learners,
where you all come from, why you all are here**?



Exercise 1 – Solution + Questions

```
mpif90 first-example_30.f90
```

```
mpirun -np 6 ./a.out
```

Enter the number of elements (n):

100

I am process 0 out of 6 handling the 0th part of n= 100 elements, result= 0.00

I am process 1 out of 6 handling the 1th part of n= 100 elements, result= 100.00

I am process 2 out of 6 handling the 2th part of n= 100 elements, result= 200.00

I am process 3 out of 6 handling the 3th part of n= 100 elements, result= 300.00

I am process 4 out of 6 handling the 4th part of n= 100 elements, result= 400.00

I am process 5 out of 6 handling the 5th part of n= 100 elements, result= 500.00

I'm proc 0: My own result is 0.00

I'm proc 0: received result of process 1 is 100.00

I'm proc 0: received result of process 2 is 200.00

I'm proc 0: received result of process 3 is 300.00

I'm proc 0: received result of process 4 is 400.00

I'm proc 0: received result of process 5 is 500.00

Normally, you'll never see this perfect output !



Why ?

Exercise 1 – Solution + Answers

```
mpif90 first-example_30.f90
```

```
mpirun -np 6 ./a.out
```

Enter the number of elements (n):

100

I am process 4 out of 6 handling the 4th part of n= 100 elements, result= 400.00

I am process 0 out of 6 handling the 0th part of n= 100 elements, result= 0.00

I'm proc 0: My own result is 0.00

I am process 2 out of 6 handling the 2th part of n= 100 elements, result= 200.00

I am process 1 out of 6 handling the 1th part of n= 100 elements, result= 100.00

I'm proc 0: received result of process 1 is 100.00

I'm proc 0: received result of process 2 is 200.00

I'm proc 0: received result of process 3 is 300.00

I am process 3 out of 6 handling the 3th part of n= 100 elements, result= 300.00

I'm proc 0: received result of process 4 is 400.00

I am process 5 out of 6 handling the 5th part of n= 100 elements, result= 500.00

I'm proc 0: received result of process 5 is 500.00

General rule:

The output of each process
is in the well defined sequence of its program,
see, e.g., the **bold text** from process 0!

The output from different processes can be intermixed in any sequence!

Most MPI libraries try to not intersect output lines ☺

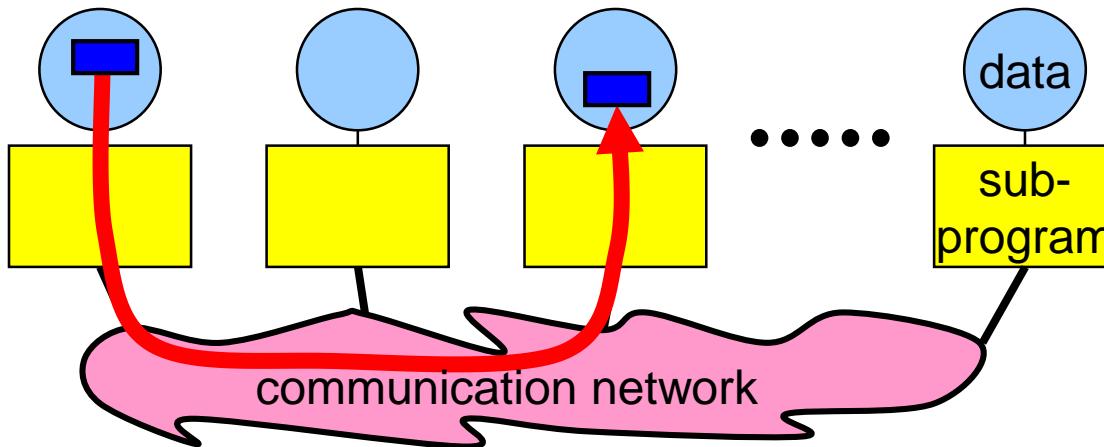
Access

- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
 - mail box
 - phone line
 - fax machine
 - etc.
- MPI:
 - sub-program must be linked with an MPI library
 - sub-program must use include file of this MPI library
 - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool



Process model and
language bindings (→Ch. 2)

Messages



- Messages are packets of data moving between sub-programs
 - Necessary information for the message passing system:
 - sending process
 - source location
 - source data type
 - source data size
 - receiving process
 - destination location
 - destination data type
 - destination buffer size
- i.e., the ranks } [blue square]
- basic or derived datatypes
(→ Ch. 3 + 12)**

Addressing

- Messages need to have addresses to be sent to.
- Addresses are similar to:
 - mail addresses
 - phone number
 - fax number
 - etc.
- MPI: addresses are ranks of the MPI processes (sub-programs)

Receiving

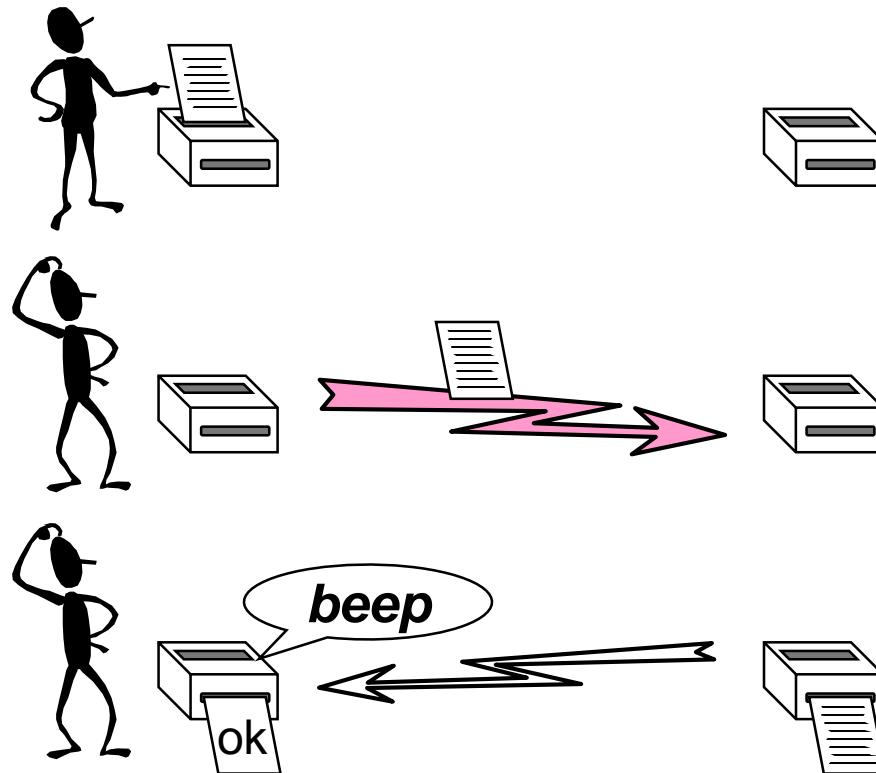
- All messages must be received.

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
 - synchronous send
 - buffered = asynchronous send

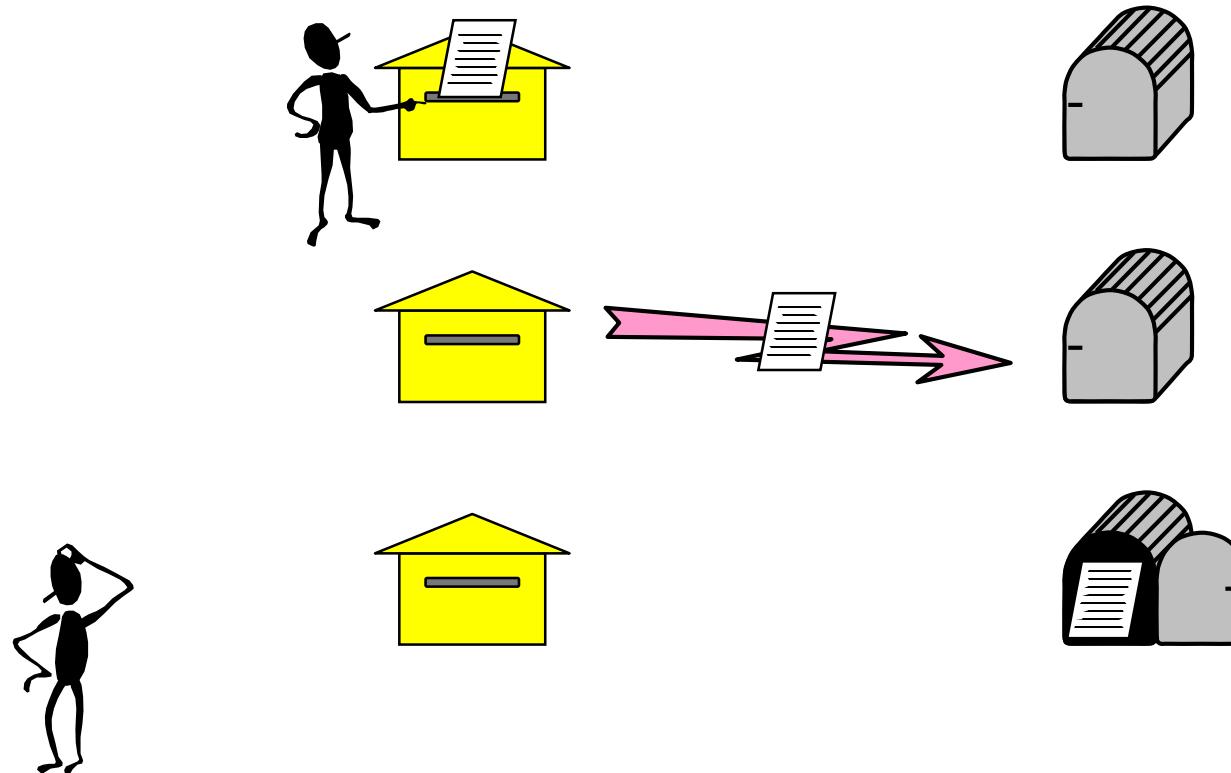
Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.



Buffered = Asynchronous Sends

- Only know when the message has left.



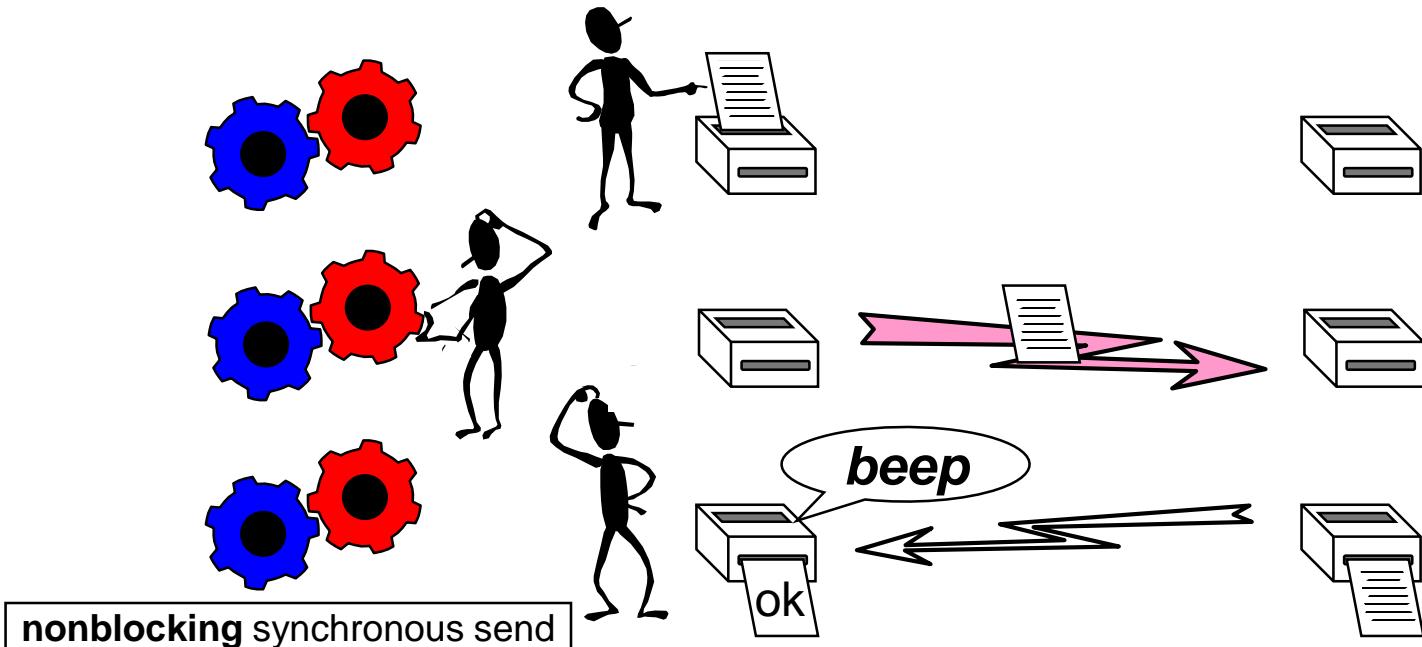
Blocking Operations

- Operations are activities, such as
 - sending (a message)
 - receiving (a message)
- Some operations may **block** until another process acts:
 - synchronous send operation **blocks until** receive is posted;
 - receive operation **blocks until** message was sent.
- Relates to the completion of an operation.
- Blocking subroutine returns only when the operation has completed.

Nonblocking Operations

Nonblocking operations consist of:

- A nonblocking procedure call: it returns immediately and allows the sub-program to perform other work
- At some later time the sub-program must **test** or **wait** for the completion of the nonblocking operation



Non-Blocking Operations (cont'd)



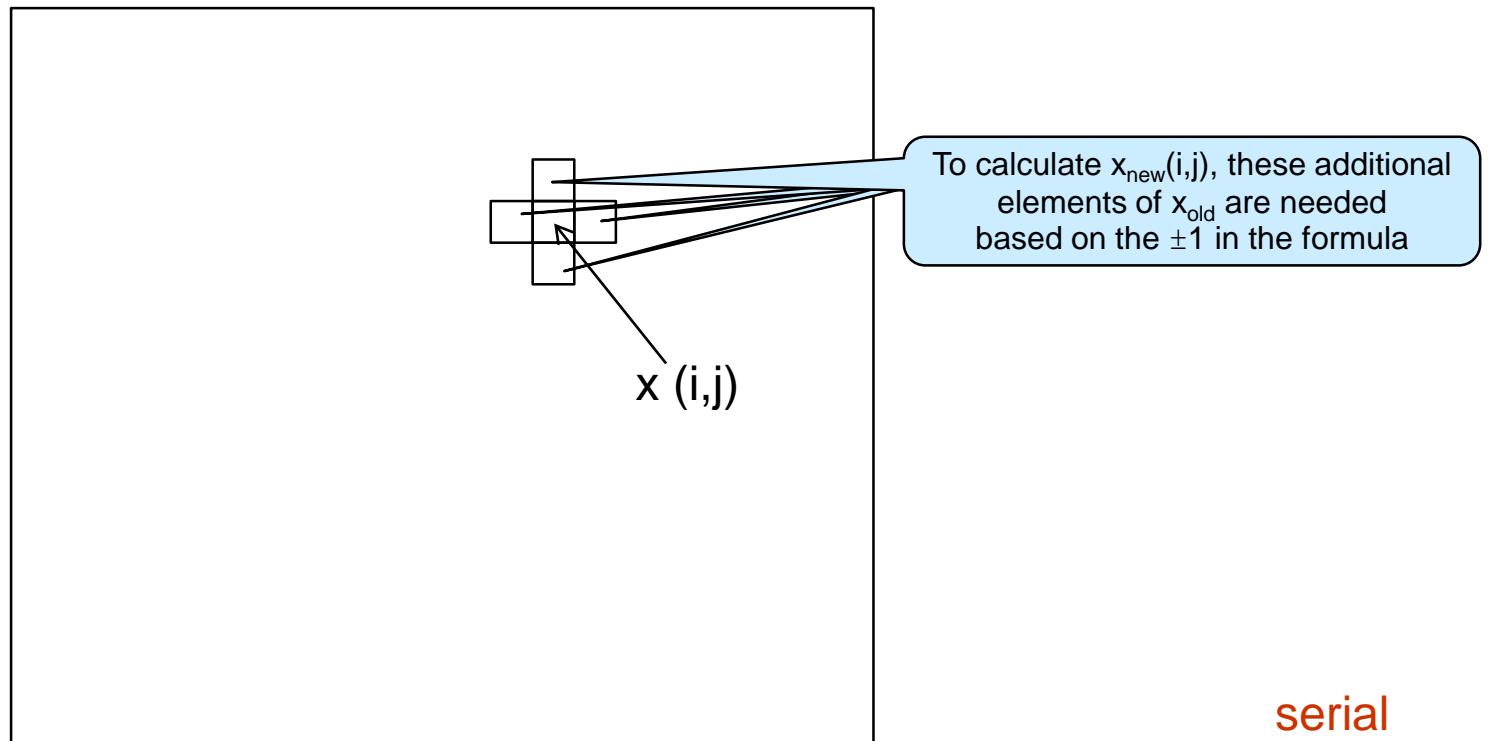
- All nonblocking procedures must have a matching wait (or test) procedure. (Some system or application resources can be freed only when the nonblocking operation is completed.)
- A nonblocking procedure immediately followed by a matching wait is equivalent to a blocking procedure.
- Nonblocking procedures are not the same as sequential subroutine calls:
 - the operation may continue while the application executes the next statements!

Interrupt: Example & Exercise 2

- Before we further go through the MPI chapter overview on
 - Collective Communication
 - Parallel file I/O
- Lets look at halo communication
- plus a short exercise 2

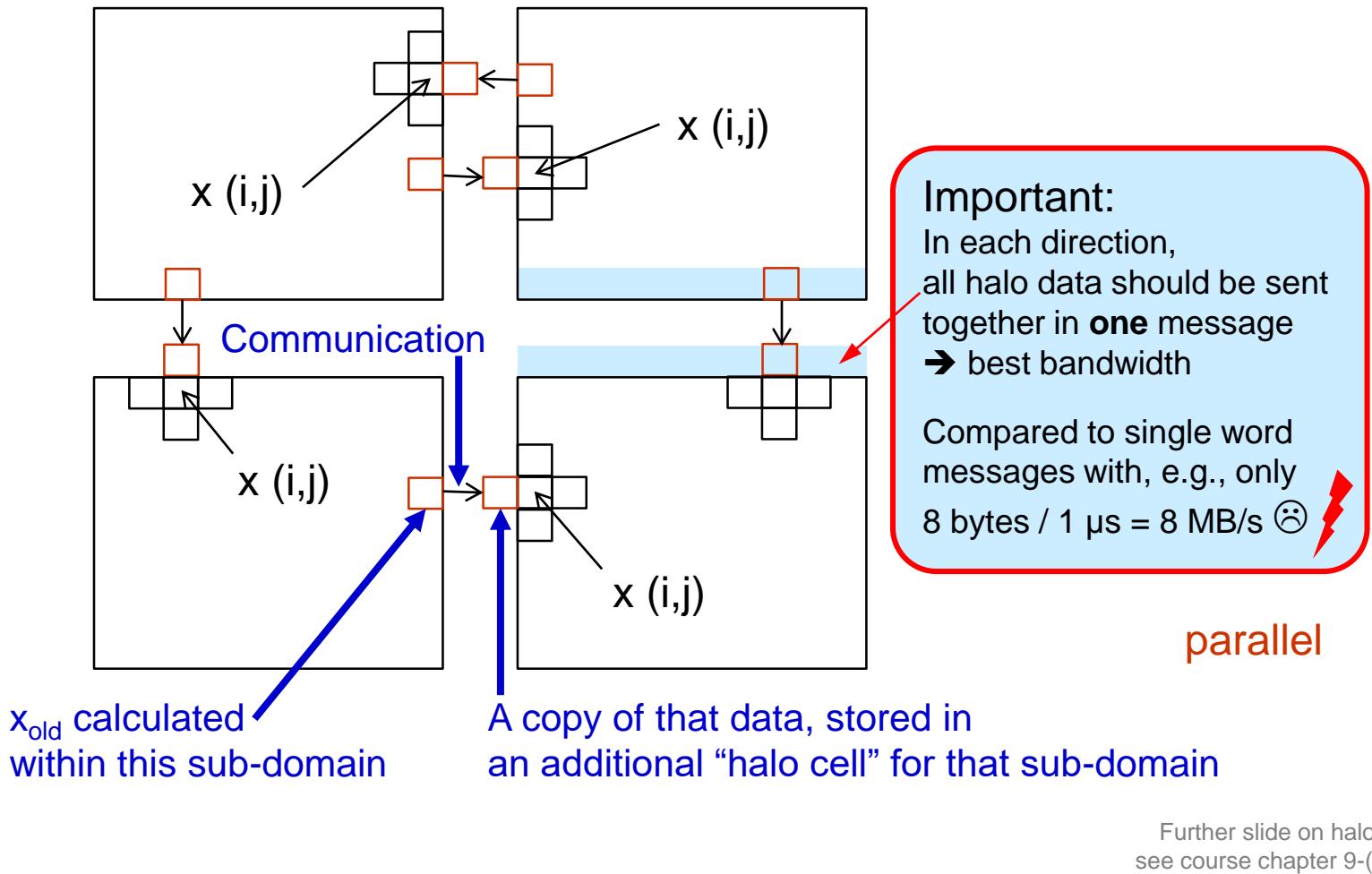
Example: Domain decomposition – serial

- $x_{\text{new}}(i,j) = f(x_{\text{old}}(i-1,j), x_{\text{old}}(i,j), x_{\text{old}}(i+1,j), x_{\text{old}}(i,j-1), x_{\text{old}}(i,j+1))$



Example: Domain decomposition – parallel

- $x_{\text{new}}(i,j) = f(x_{\text{old}}(i-1,j), x_{\text{old}}(i,j), x_{\text{old}}(i+1,j), x_{\text{old}}(i,j-1), x_{\text{old}}(i,j+1))$



9-(1)

Halos



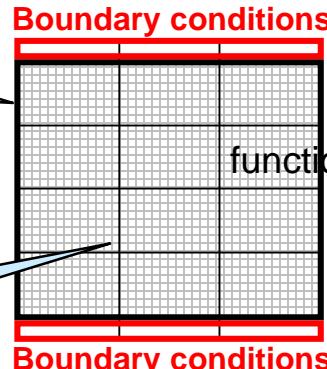
Further slide on halos,
see course chapter 9-(1)

Communication: Send inner data into halo storage

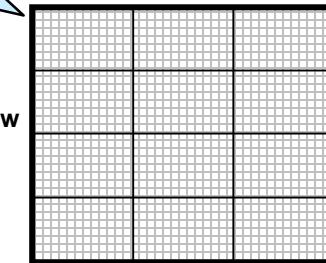
One iteration in the **serial** code:

- $x_{\text{new}} = \text{function}(x_{\text{old}})$
- $x_{\text{old}} = X_{\text{new}}$

The data mesh x_{old}



The data mesh x_{new}

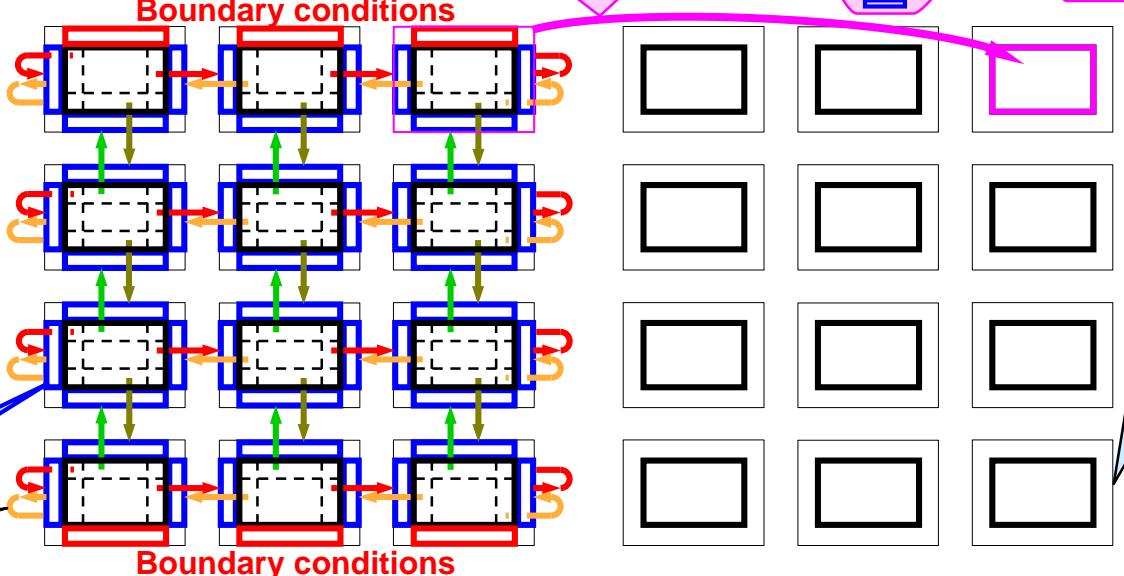


Preparing the domain decomposition of the data mesh into 3x4 subdomains for 12 processes

parallel code:

- Update halo
[=Communication, e.g., with 4 x MPI_Sendrecv ]
- $x_{\text{new}} \square = \text{function}(x_{\text{old}} \square)$
- $x_{\text{old}} \square = X_{\text{new}} \square$

halo cells



horizontally cyclic boundary conditions
 \rightarrow communication around rings

x_{old} & boundary conditions distributed on 3x4=12 MPI processes

x_{new} , distributed on same MPI processes

Maybe same memory layout for x_{old} and x_{new} to allow pointer exchange instead of data copy $x_{\text{old}} = x_{\text{new}}$

Example code

```
ib_global = 0; ie_global=n-1; // global xold, xnew: arrays with n elements and indexes 0 .. n-1
```

Chap. 2
Process Model

```
MPI_Init(null, null);
MPI_Comm_size(MPI_COMM_WORLD, &size);      // size = number of processes
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // myrank = index of the process from 0 to size-1
```

$$\text{ib_local} = \text{ib_global} + \text{myrank} * \left(\frac{(\text{ie_global}-\text{ib_global})/\text{size} + 1}{\lceil n/\text{size} \rceil} \right);$$

$$\text{ie_local} = \min(\text{ib_global} + (\text{myrank}+1) * \left(\frac{(\text{ie_global}-\text{ib_global})/\text{size} + 1}{\lceil n/\text{size} \rceil} \right), \text{ie_global});$$

Chap. 3
Messages +

```
left = (myrank-1 + size) % size; right = (myrank+1) % size; // neighbor ranks
To prohibit modulo of negative number
```

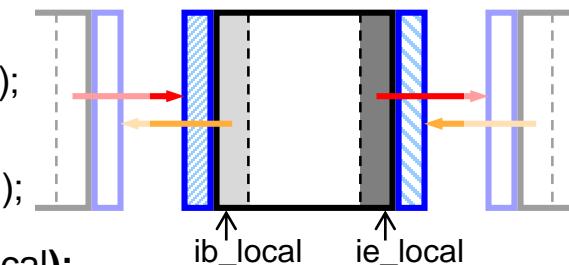
for(...) // e.g. timesteps

Chap. 4
Nonblocking
communic.

```
{ MPI_Sndrecv( /*sndbuf*/ xold, 1, right_black_part, right, 111,
                /*rcvbuf */ xold, 1, left_blue_halo, left, 111, ...);
    MPI_Sndrecv( /*sndbuf*/ xold, 1, left_black_part, left, 222,
                /*rcvbuf */ xold, 1, right_blue_halo, right, 222, ...);
```

Chap. 12
Derived types

```
numerical_func( xold, xnew, ib_global, ib_local, ie_global, ie_local);
tmp=xold; xold=xnew; xnew=tmp; // exchanging role of xold and xnew
}
```



Exercise 2: Calculating the size of the subdomains

Goal: Divide a given amount of mesh elements in one dimension into subdomains

- Given:
 - The number of processes: num_procs (e.g., 4, i.e., 4 subdomains)
 - The number of mesh elements: n (e.g., **17 or 5**)
 - The numerical workload of each element is identical
 - The mesh elements are numbered from **0** to **n-1**

#mesh elements per subdomain

- Two possible solutions:**
 - (A) $17=5+5+5+2$ or $5=2+2+1+0$
 - (B) $17=5+4+4+4$ or $5=2+1+1+1$
- Output should be like (with B)

I am process 1 out of 4, responsible for the 4 elements with indexes 5 .. 8

I am process 0 out of 4, responsible for the 5 elements with indexes 0 .. 4

I am process 3 out of 4, responsible for the 4 elements with indexes 13 .. 16

Or -1 if 0 elements

I am process 2 out of 4, responsible for the 4 elements with indexes 9 .. 12

In MPI/tasks/...

- Use:
 - C C/Ch1/first-dd-a.c and C/Ch1/first-dd-b.c
 - or Fortran F_30/Ch1/first-dd-a_30.f90 and F_30/Ch1/first-dd-b_30.f90
 - or Python PY/Ch1/first-dd-a_30.py and PY/Ch1/first-dd-b_30.py
- Test both programs with 4 processes and 9, 8, 7, ... 1 elements

Which algorithm would you prefer, and why?

Which are the major principles of A and B?

During the Exercise (15 min.)



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

And have fun with this short exercise



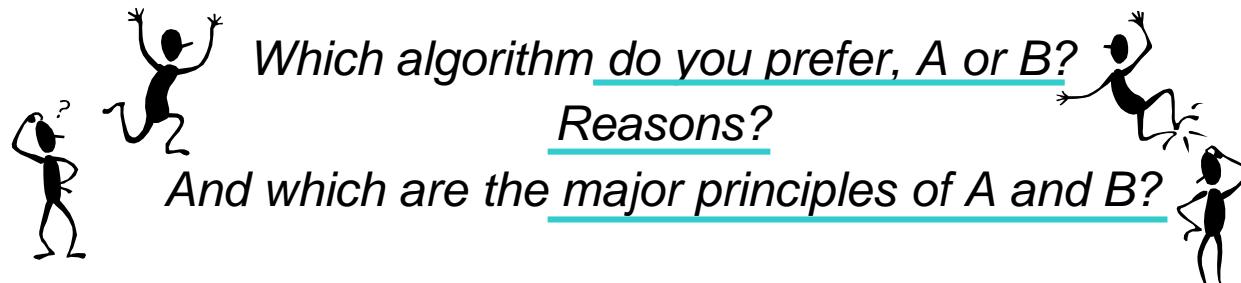
Please do not look at the solution before you finished this exercise,
otherwise,

90% of your learning outcome may be lost



**As soon as you finished your tests,
please go to your breakout room**

and continue your discussions with your fellow learners.

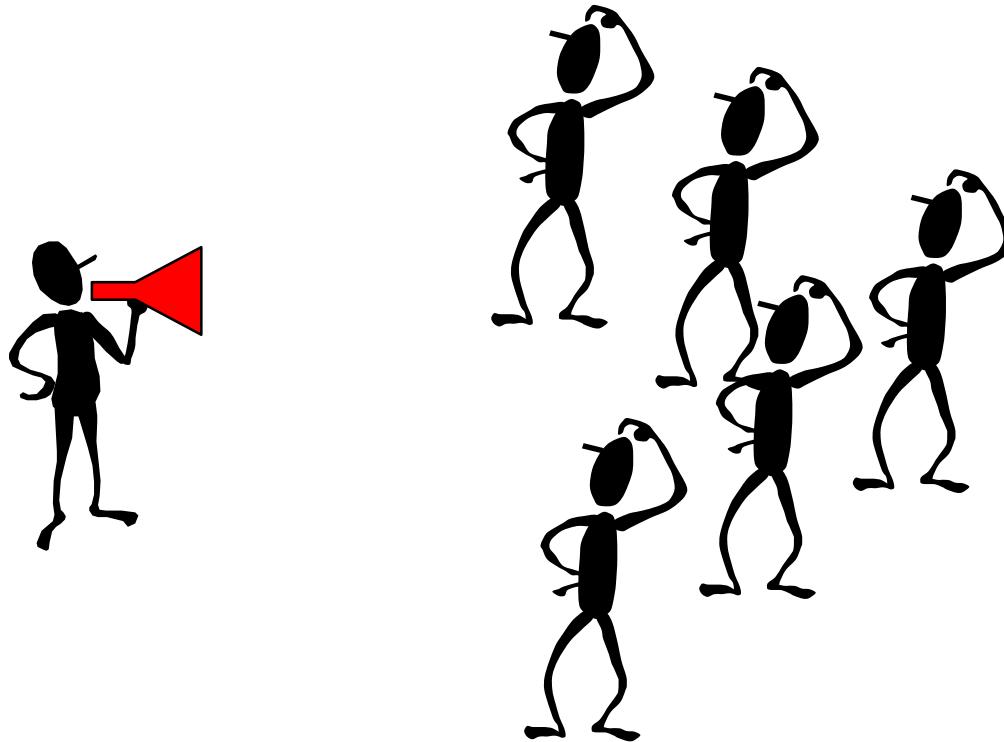


Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms.
- Can be built out of point-to-point communications.

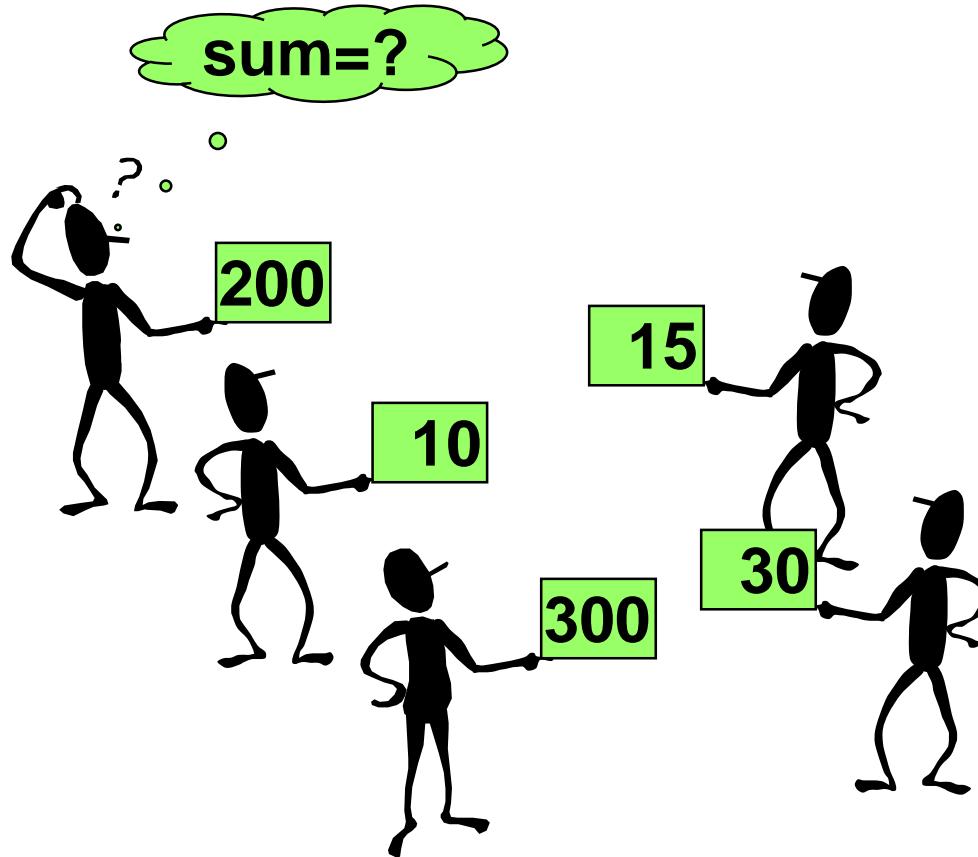
Broadcast

- A one-to-many communication.



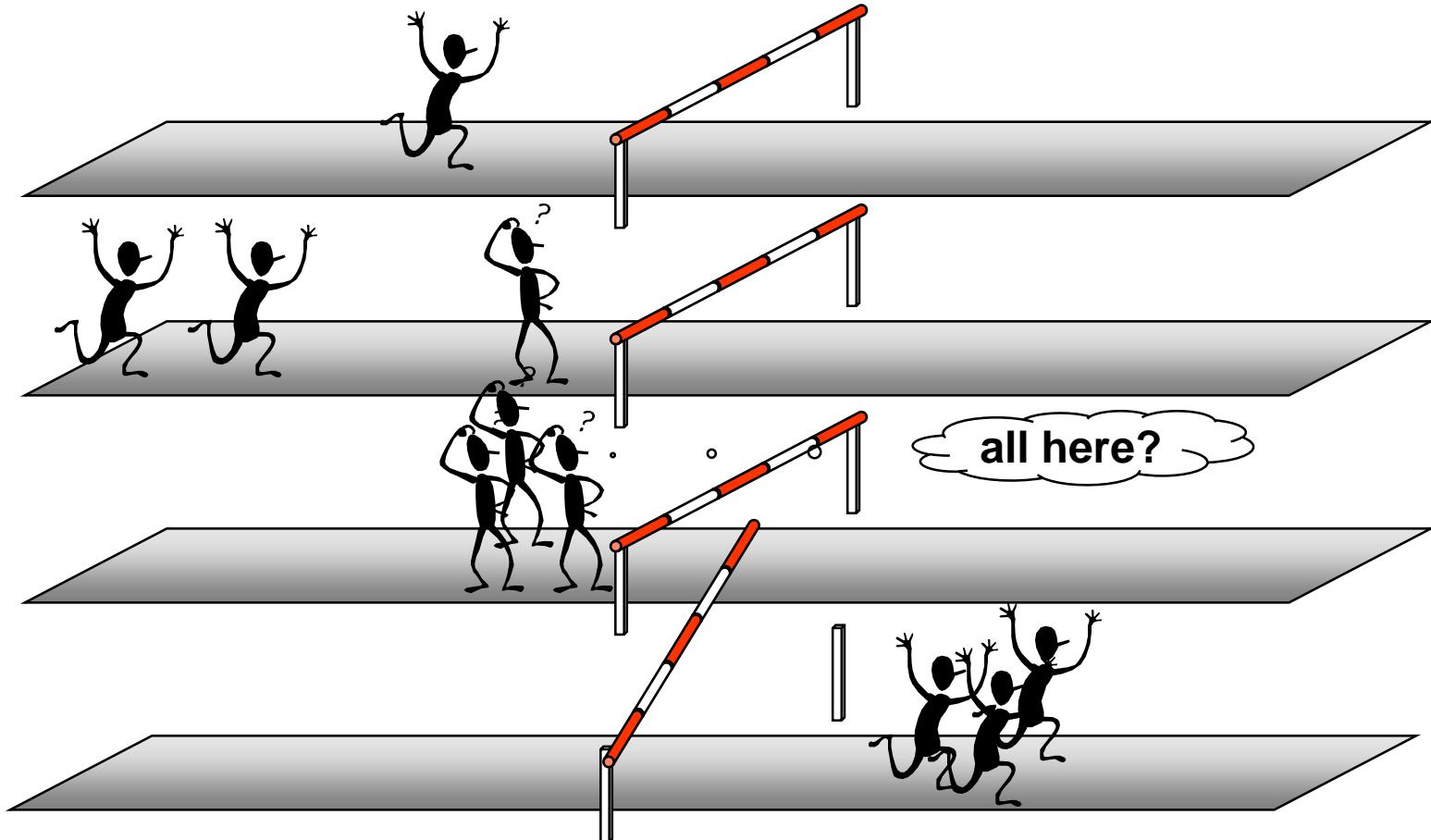
Reduction Operations

- Combine data from several processes to produce a single result.

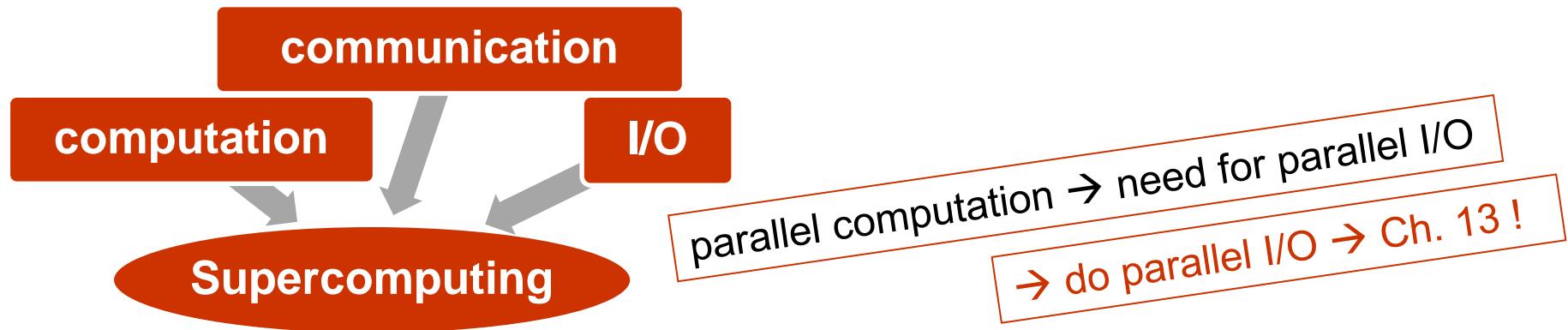


Barriers

- Synchronize processes.



Parallel File I/O



calculation on	time for computation	time for <u>serial</u> I/O
1 core	64 min = 98.5 % of total time	1 min = 1.5 % of total time
64 cores	1 min = 50 % of total time	1 min = 50 % of total time

Table: example with serial I/O

-
- 1) waste of resources
2) negative side effects on other users

Speedup, Efficiency, Scaleup, and Weak Scaling

- Definition: $T(p,N)$ = time to solve **problem of total size N** on **p processors**
- Parallel speedup: $S(p,N) = T(1,N) / T(p,N)$
compute **same problem** with more processors in **shorter time**
- Parallel Efficiency: $E(p,N) = S(p,N) / p$
- Scaleup: $Sc(p,N) = N / n$ with $T(1,n) = T(p,N)$
compute **larger problem** with more processors in **same time**
- Weak scaling: $T(p, p \cdot n) / T(1,n)$ is reported,
i.e., problem size per process (n) is fixed
- Problems:
 - Absolute MFLOPS rate / hardware peak performance?
 - Super-scalar speedup: $S(p,N) > p$, e.g., due to cache usage for large p :
 - $T(1,N)$ may be based on a huge number of N data elements in the memory in the one process, whereas
 - $T(p,N)$ may be based on *cache based execution* due to only N/p data elements per process
 - $S(p,N)$ close to **p** or far less? → see Amdahl's Law on next slide

Three different ways of reporting the success

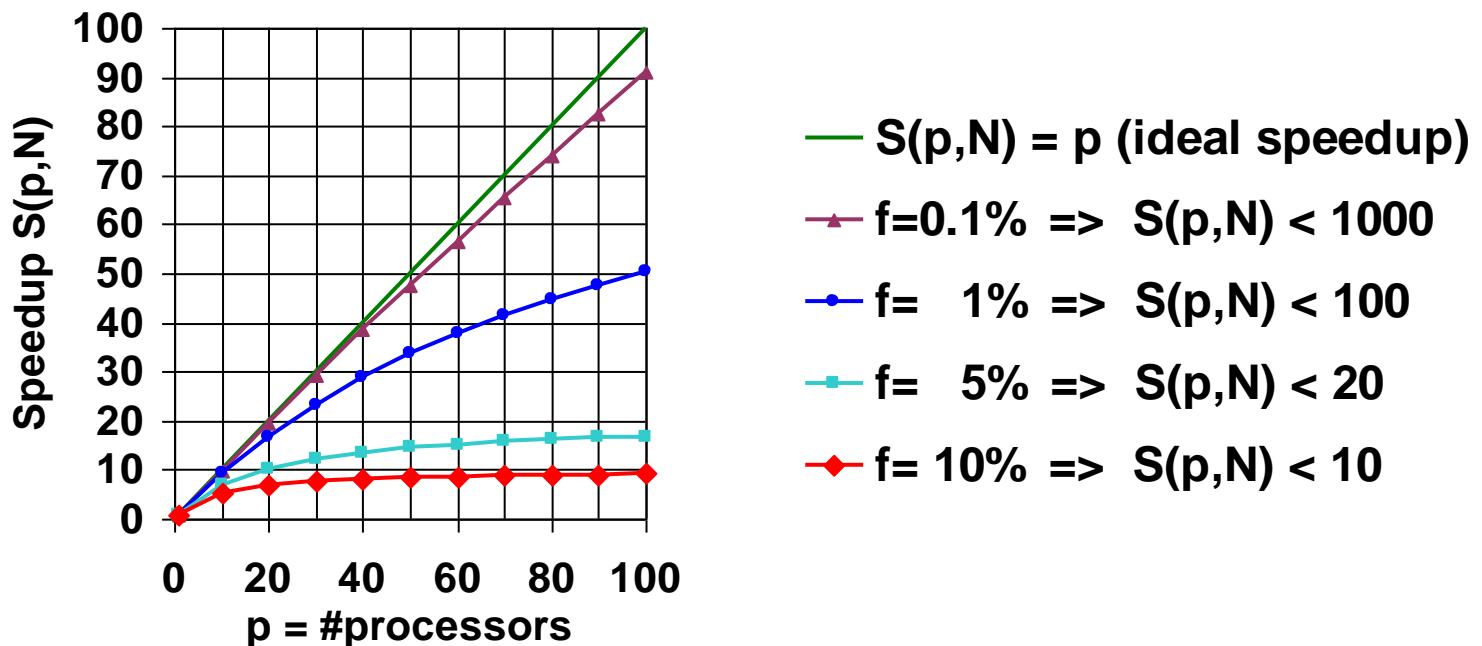
Amdahl's Law

$$T(p,N) = f \cdot T(1,N) + (1-f) \cdot T(1,N) / p$$

f ... sequential part of code that can not be done in parallel

$$S(p,N) = T(1,N) / T(p,N) = 1 / (f + (1-f) / p)$$

For $p \rightarrow \infty$, speedup is limited by $S(p,N) < 1 / f$



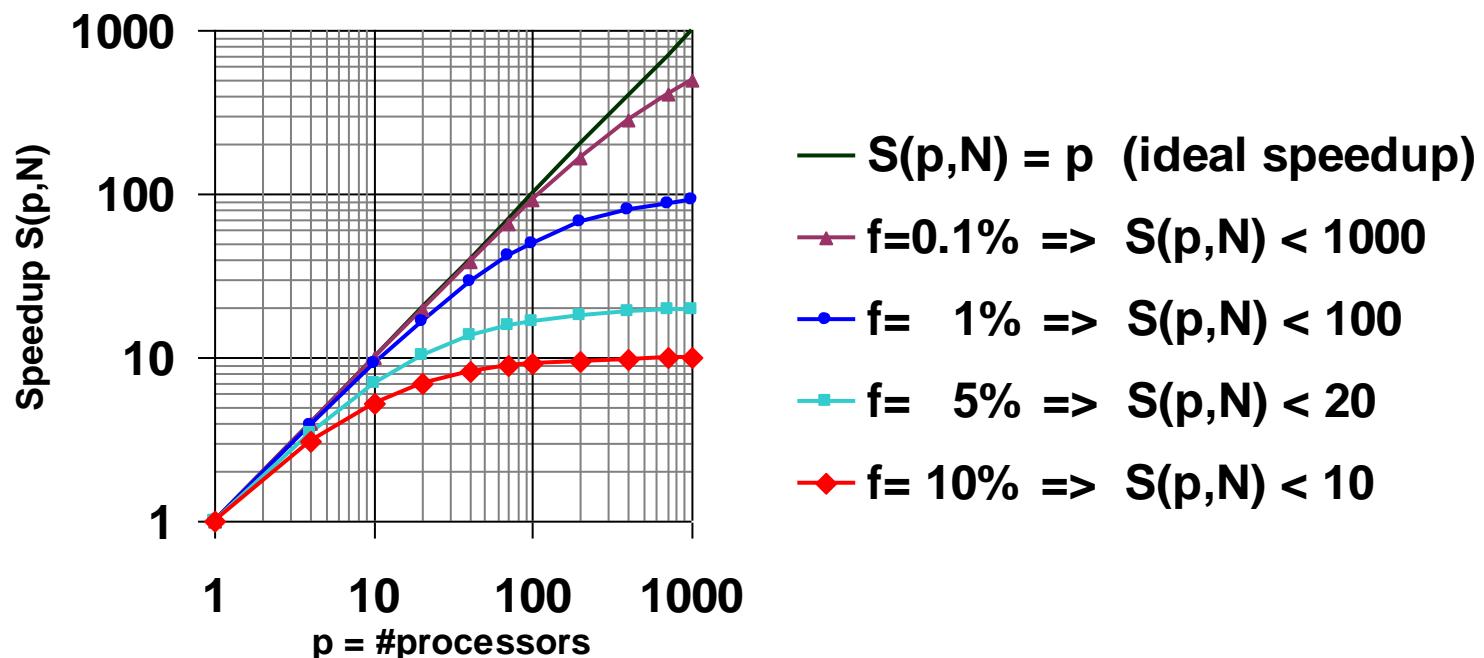
Amdahl's Law (double-logarithmic)

$$T(p,N) = f \cdot T(1,N) + (1-f) \cdot T(1,N) / p$$

f ... sequential part of code that can not be done in parallel

$$S(p,N) = T(1,N) / T(p,N) = 1 / (f + (1-f) / p)$$

For $p \rightarrow \infty$, speedup is limited by $S(p,N) < 1 / f$



MPI Forum

- **MPI-1 Forum**

- First message-passing interface standard.
- Sixty people from forty different organizations.
- Users and vendors represented, from US and Europe.
- Two-year process of proposals, meetings and review.
- *Message-Passing Interface* document produced.
- MPI-1.0 — June, 1994.
- MPI-1.1 — June 12, 1995.

MPI Forum, continued

- **MPI-2 Forum**

- Same procedure (e-mails, and meetings in Chicago, every 6 weeks).
- *MPI-2: Extensions to the Message-Passing Interface* (July 18, 1997). containing:
 - MPI-1.2 — mainly clarifications.
 - MPI-2.0 — extensions to MPI-1.2.

- **MPI-3 Forum → MPI-4 Forum**

- Started Jan. 14-16, 2008 (1st meeting in Chicago)
- Using e-mails, wiki, meetings every 8 weeks (Chicago and San Francisco), and telephone conferences
- MPI-2.1 — June 23, 2008
 - mainly combining MPI-1 and MPI-2 books to one book
- MPI-2.2 — September 4, 2009: Clarifications and a few new func.
- MPI-3.0 — September 21, 2012: Important new functionality
- MPI-3.1 — June 4, 2015: Errata & new: Nonblocking I/O, MPI_AINT_ADD
- MPI-4.0 — June 9, 2021: Several new functionalities, see tour  [New in MPI-4.0](#)



DIFF

Goals and Scope of MPI

- MPI's prime goals
 - To provide a message-passing interface.
 - To provide source-code portability.
 - To allow efficient implementations.
- It also offers:
 - A great deal of functionality.
 - Support for heterogeneous parallel architectures.
- With MPI-2:
 - Important additional functionality.
 - No changes to MPI-1.
- With MPI-2.1, 2.2, 3.0, 3.1, 4.0:
 - Important additional functionality to fit on new hardware principles.
 - Deprecated MPI routines moved to chapter “Deprecated Functions”.
 - With MPI-3.0, some deprecated features were removed.

About this course

- MPI-3 and -4 was developed for better **platform** and **application** support.
- MPI for HPC: Better support of clusters of SMP nodes
 - This is an MPI-3.1 (and 4.0) course
 - includes most (performance) features of MPI
- Only overview-information for less important features of MPI
 - This course is for applications on systems ranging
 - from small cluster
 - to large HPC systems
- Participants:
 - Beginners → basic exercises may be already challenging
 - Intermediate users → additional infos on the slides + additional exercises
 - Advanced users → fast basic exercises + challenging advanced exercises
 - Former participants → You may look for updates, see  **(New in) MPI-2.0 / 2.1 / 2.2, MPI-3.0, MPI-3.1, MPI-4.0**
or go to the tour  **New in MPI-4.0**

Quiz on Chapter 1 – Overview

Two developers report about their limited success when parallelizing an application:

- A. “My application is now running in parallel with 1000 MPI processes and my major limiting factor for scaling is
 - that I need about 50% of the whole compute time for MPI communication.”
- B. “My application is now running in parallel with 1000 MPI processes and my major limiting factor for scaling is
 - that I could not parallelize about 10% of the execution time of my sequential program.”

What are your answers for

- In your opinion, who of them was **more successful, A or B?**
- Can you calculate an estimate for the parallel efficiency of the parallel run reported by A and B?

For private notes

Chap.2 Process Model and Language Bindings

1. MPI Overview

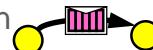


2. Process model and language bindings

- starting several MPI processes

**MPI_Init()
MPI_Comm_rank()**

3. Messages and point-to-point communication



4. Nonblocking communication



5. The New Fortran Module mpi_f08

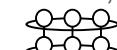


6. Collective communication

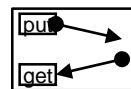


7. Error Handling

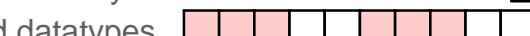
8. Groups & communicators, environment management



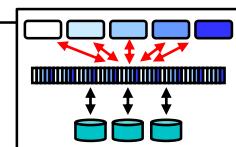
9. Virtual topologies



10. One-sided communication



11. Shared memory one-sided communication



12. Derived datatypes

13. Parallel file I/O

14. MPI and threads

15. Probe, Persistent Requests, Cancel

16. Process creation and management

17. Other MPI features

18. Best Practice

Header files

C

- C / C++

```
#include <mpi.h>
```

Python

- Python

```
from mpi4py import MPI
```

Fortran

- Fortran

```
use mpi_f08
```

```
use mpi
```

- Available since MPI-3.0
- Full consistency with Fortran standard
- Compile-time argument checking in all MPI libraries
→ Recommended

(or: include 'mpif.h')

Compile-time argument checking:
MPI-2.0 – 2.2: may be
MPI-3.0 and later: mandatory
but some MPI libraries still without.

Normally without
any
compile-time
argument
checking

MPI-3.0 and later:
The use of mpif.h is strongly discouraged!

MPI Function Format

In C and Python: case sensitive

C

- C / C++: `error = MPI_Xxxxxx(parameter, ...);`
`MPI_Xxxxxx(parameter, ...);`

Python

- Python: `result_value_or_object = input_mpi_object.mpi_action(parameter, ...)`

direct communication of numPy arrays (like in C)

`comm_world = MPI.COMM_WORLD`
`comm_world.Send((snd_buf, ...), ...)`
`comm_world.Recv((rcv_buf, ...), ...)`

Python interfaces: In analogy to the former MPI C++ interfaces in MPI-2.0 – MPI-2.2

Fortran

- Fortran: `CALL MPI_XXXXXXX(parameter, ..., ierror)`

Or with object-serialization:
`comm_world.send(snd_buf, ...)`
`rcv_buf = comm_world.recv(...)`

With mpi_f08 module:
ierror is optional (since MPI-3.0)

In Fortran: **not** case sensitive

With mpi module or mpif.h:

Absolutely Never forget!

Upper/mixed case	MPI standard	In this course
<code>MPI_Xxx_mixed</code>	<code>MPI procedures in C</code> and in Fortran <code>mpi_f08</code> module	<code>ditto in C and Python</code>
<code>MPI_XXX_UPPER</code>	<code>Language independent proc. specifications</code> <code>MPI procedures in mpi module and mpif.h</code>	<code>all Fortran MPI procedures</code> Recommendation: Use “mixed case” in your Fortran code
<code>MPI_Xxx_mixed</code>	<code>MPI type declarations</code>	<code>ditto.</code>
<code>MPI_XXX_UPPER</code>	<code>MPI constants</code>	<code>ditto.</code>

ierror with old mpif.h and new mpi_f08

- Unused ierror

INCLUDE 'mpif.h'

! wrong call:

CALL MPI_SEND(...., MPI_COMM_WORLD)

! → terrible implications because ierror=0 is written somewhere to the memory

mpi + mpif.h:
ierror is mandatory
→ NEVER FORGET!

- With the new module

USE mpi_f08

! Correct call, because ierror is **optional**:

CALL MPI_SEND(...., MPI_COMM_WORLD)

mpi_f08:
ierror is OPTIONAL

- Conclusion:** You may switch to the **mpi_f08** module

MPI Function Format Details

- Have a look into the MPI standard, e.g., MPI-3.1, page 28 or MPI-4.0 page 37. Each MPI routine is defined:
 - language independent (page:lines – p28:21-33), New in MPI-3.0
 - programming languages: C / Fortran **mpi_f08** / **mpi** & **mpif.h** (p28:34-48).

C

Output arguments in C/C++:

definition in the standard `MPI_Comm_rank(...., int *rank)`
 `MPI_Recv(..., MPI_Status *status)`

usage in your code: `main...`
 `{ int myrank; MPI_Status rcv_status;`
 `MPI_Comm_rank(..., &myrank);`
 `MPI_Recv(..., &rcv_status);`

New in MPI-3.1

- Several index sections at the end: **General Index**, Examples, **Constant and Predefined Handle**, Declarations, Callback Function Prototype, **Function Index**.
- `MPI.....` namespace is reserved for MPI constants and routines, i.e. application routines and variable names must not begin with `MPI_`.
- `mpi4py` is not part of the MPI standard. Internally a wrapper to the C binding.

Python

MPI 3.1 page 28

MPI 4.0 page 37

- Language independent definition

- C interface

- Fortran 2008 interface through mpi_f08 module

- Old Fortran interface through mpi module and mpif.h

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

MPI_RECV (buf, count, datatype, source, tag, comm, status)

23	OUT	buf	initial address of receive buffer (choice)
24	IN	count	number of elements in receive buffer (non-negative integer)
25	IN	datatype	datatype of each receive buffer element (handle)
26	IN	source	rank of source or MPI_ANY_SOURCE (integer)
27	IN	tag	message tag or MPI_ANY_TAG (integer)
28	IN	comm	communicator (handle)
32	OUT	status	status object (Status)

```
34 int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
35               int tag, MPI_Comm comm, MPI_Status *status)
```

```
37 MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
```

```
38      TYPE(*), DIMENSION(..) :: buf
```

```
39      INTEGER, INTENT(IN) :: count, source, tag
```

```
40      TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
41      TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
42      TYPE(MPI_Status) :: status
```

```
43      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Large count version in MPI-4.0
MPI_Recv_c(...) in C
with MPI_Count count
MPI_Recv(...)!(_c) in Fortran
with TYPE(MPI_Count) :: count

```
44 MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
```

```
45      <type> BUF(*)
```

```
46      INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
```

```
47      IERROR
```

No large count in mpi / mpif.h

<https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf#page=60>

<https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf#page=77>

Initializing MPI

`MPI_Init()` must be called before any other MPI routine
(only a few exceptions, e.g., `MPI_Initialized`)

C

- `int MPI_Init(int *argc, char ***argv)`

MPI-2.0 and higher:
Also
`MPI_Init(NULL, NULL);`

```
#include <mpi.h>
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ....
```

Fortran

- `MPI_INIT(IERROR)`
`INTEGER IERROR`

With MPI-3.0 and later recommended:
`use mpi_f08`

```
program xxxxx
use mpi_f08
implicit none
call MPI_INIT()
....
```

! With MPI-2.0:
program xxxxx
use mpi
implicit none
integer ierror
call MPI_INIT(ierror)

! With MPI-1.1:
program xxxxx
implicit none
include 'mpif.h'
integer ierror
call MPI_INIT(ierror)

If you install an MPI library with Fortran support:
Never install MPI without `mpi_f08` !

Test it with the version test
→ 2nd Advanced Exercise

Python

- `# MPI.Init()`

This call is not needed, because
automatically called at the import
of MPI at the begin of the program

```
from mpi4py import MPI
# MPI.Init() is not needed
....
```

The Fortran support methods

In MPI-4.0, new large count interfaces only in mpi_f08 !

Fortran support method	MPI-1.1	MPI-2	MPI-3	MPI-4.0	MPI-next	MPI-...	far future
USE mpi_f08	x	x	5	5	5	5	5
USE mpi	x	3	4	4	2b	2b	1
INCLUDE 'mpif.h'	3	3	2a	2a/b	1	0	0

Past Today Maybe in the future

Level of Quality:

- 5 – valid and consistent with the Fortran standard (Fortran 2008 + TS 29113)¹⁾
- 4 – valid and only partially consistent
- 3 – valid and small consistency (e.g., without argument checking)
- 2 – use is strongly (a) discouraged or (b) partially frozen (i.e., not with all new functions)
- 1 – deprecated
- 0 – removed
- x – not yet existing

¹⁾ For full consistency, Fortran 2003 + TS29113 is enough.

Fortran 2018 and later versions include TS 29113.

Without TS29113, same partial consistency as with the mpi module.

Exiting MPI

C

Fortran

Python

- C/C++: int MPI_Finalize()
- Fortran: MPI_FINALIZE(*ierror*)
mpi_f08: INTEGER, OPTIONAL :: ierror
mpi & mpif.h: INTEGER ierror
- Python: # MPI.Finalize()

This call is not needed,
because automatically called at the end of the program

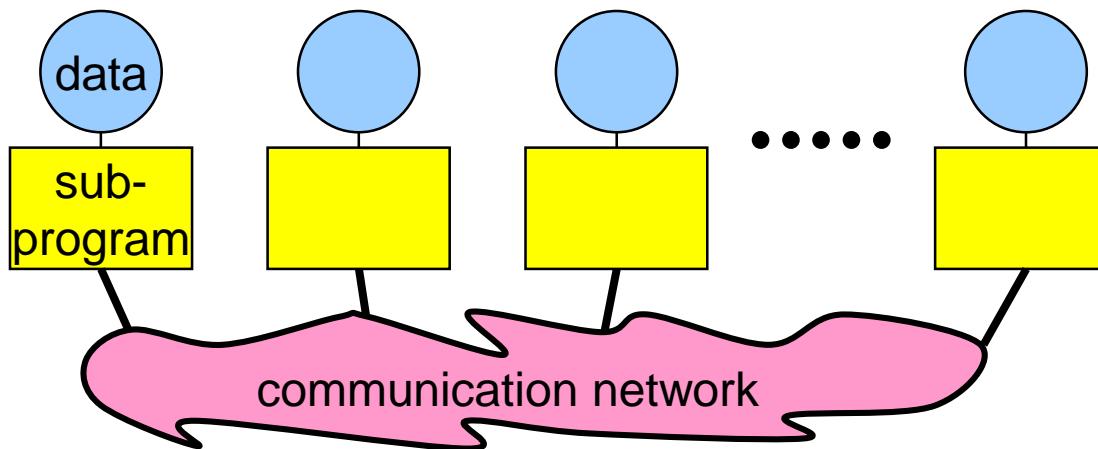
- **Must** be called last by all processes.
- User must ensure the completion of all pending communications (locally) before calling finalize
- After MPI_Finalize:
 - Further MPI-calls are forbidden
 - Especially re-initialization with MPI_Init is forbidden
 - **May** abort the calling process if its rank in MPI_COMM_WORLD is $\neq 0$
- Alternatives in MPI-4.0
 - World Model: MPI_Init/Finalize and MPI_COMM_WORLD
 - Sessions Model: See course chapter 8-(2)

New in MPI-4.0

 New in MPI-4.0

Starting the MPI Program

- Start mechanism is implementation dependent
- mpirun –np ***number_of_processes*** ***./executable*** (most implementations)
- mpiexec –n ***number_of_processes*** ***./executable*** (with MPI-2 and later)



- The parallel MPI processes exist at least after MPI_Init was called.

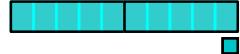
Exercise 1: Hello World

In MPI/tasks/...

- Use: **C** C/Ch2/hello-skel.c or **Fortran** F_30/Ch2/hello-skel_30.f90
or **Python** PY/Ch2/hello-skel.py
- Write a minimal **MPI** program which prints „hello world“ by each MPI process.
- Compile and run it on a single processor.
- Run it on several processors in parallel.
- Expected output on 4 processes

```
Hello world
Hello world
Hello world
Hello world
```

During the Exercise (10 min.)



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



Important: To solve the exercises please **use previous slides and the provided .c/.f90 files**
(or in rare cases also the MPI standard)
but **no google search on the web** – otherwise you are too slow



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.



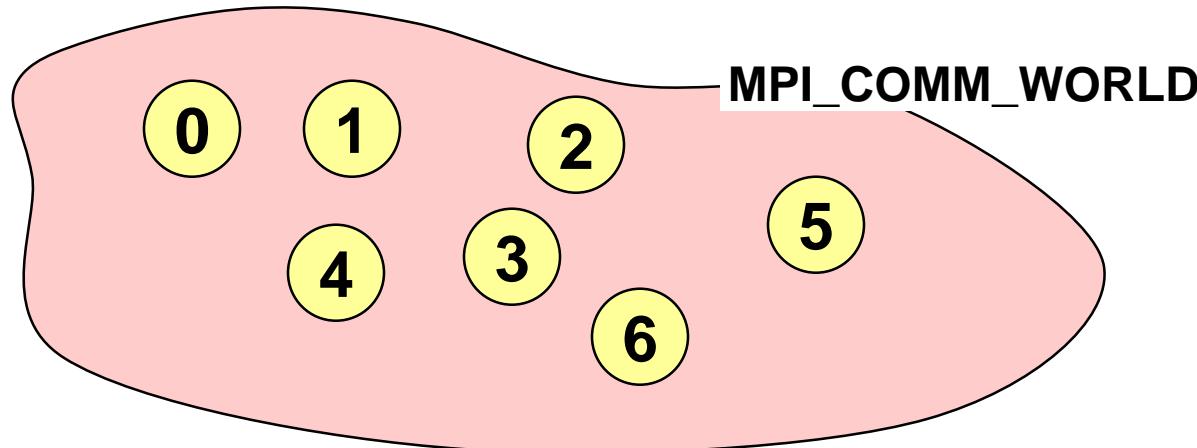
**Or, what do you believe what can happen,
if you have a print statement before the MPI_Init?**



(If your putty-keyboard gets locked due to unintended Ctrl/Strg+s then unlock with Ctrl/Strg+q)

Communicator MPI_COMM_WORLD

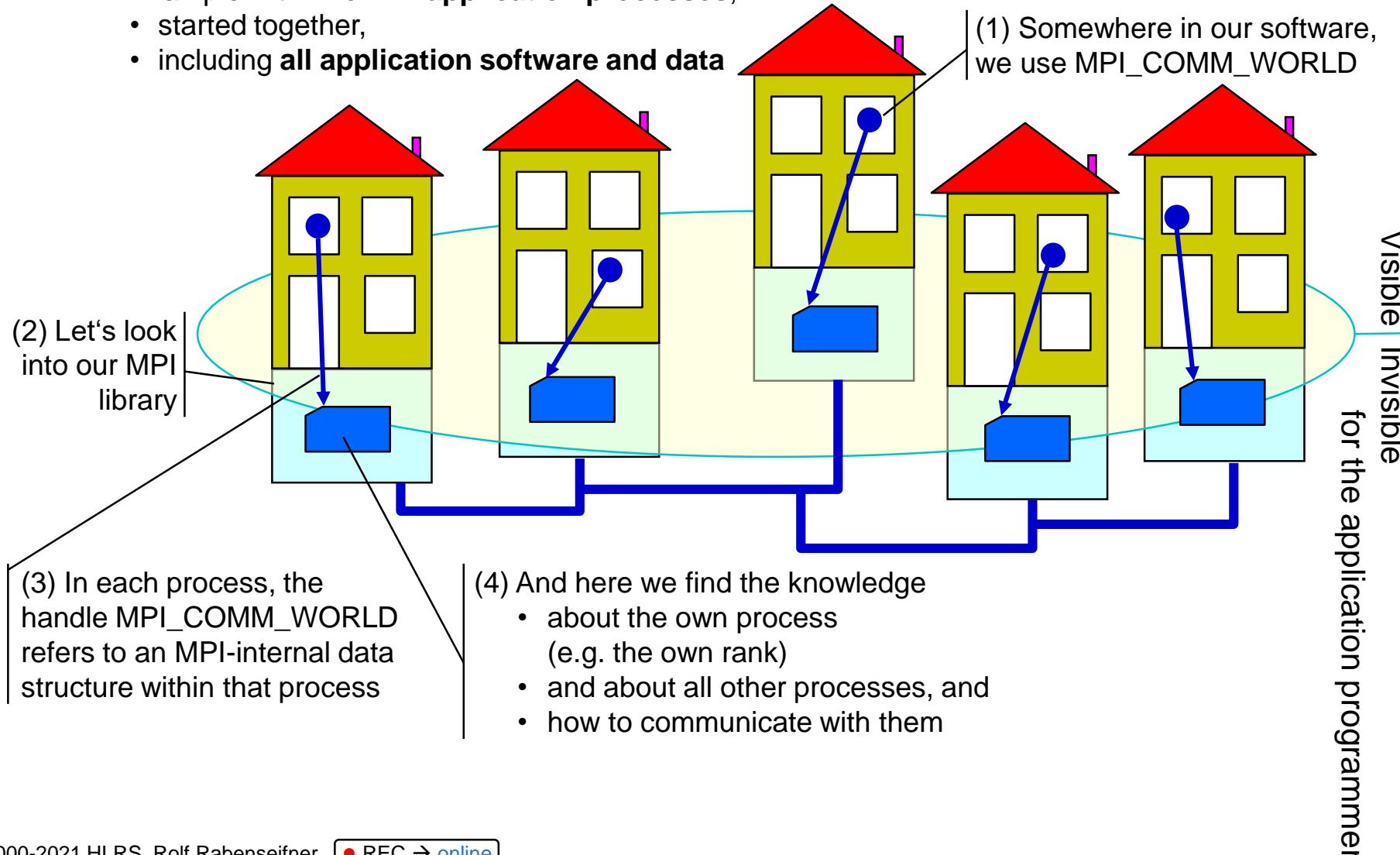
- All processes (= sub-programs) of one MPI program are combined in the **communicator MPI_COMM_WORLD**.
- MPI_COMM_WORLD is a predefined **handle** in
 - mpi.h and
 - mpi_f08 and mpi modules and mpif.h.
- Each process has its own **rank** in a communicator:
 - starting with 0
 - ending with (size-1)



Handles refer to internal MPI data structures

Example with **five MPI application processes**,

- started together,
- including **all application software and data**



Handles

- Handles identify MPI objects.
- For the programmer, handles are
 - **predefined constants** in C include file mpi.h or Fortran mpi_f08 or mpi modules or mpif.h or MPI module of mpi4py
 - Example: MPI_COMM_WORLD or MPI.COMM_WORLD ↗
 - Can be used in initialization expressions or assignments.
 - They are link-time constants, i.e., need not to be compile-time constants.
 - **values returned** by some MPI routines, to be stored in variables, that are defined as

Fortran

- Fortran:
 - mpi_f08 module: New in MPI-3.0 `TYPE(MPI_Comm) :: sub_comm`
 - mpi module and mpif.h: `INTEGER sub_comm`

C

- C: special MPI typedefs, e.g., `MPI_Comm sub_comm;`

Python

- Python: Type of object defined by the creating function, e.g., `sub_comm = MPI.COMM_WORLD.Split(...)`

- Handles refer to internal MPI data structures

Rank

- The rank identifies different processes.
- The rank is the basis for any work and data distribution.

C

Fortran

Python

• C/C++: int MPI_Comm_rank(MPI_Comm comm, int **rank*)

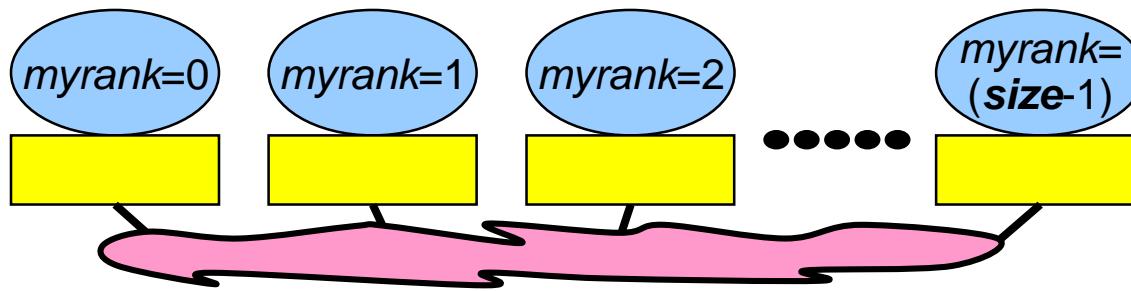
• Fortran: MPI_COMM_RANK(comm, *rank*, *ierror*)

mpi_f08: TYPE(MPI_Comm) :: comm
INTEGER :: rank; INTEGER, OPTIONAL :: ierror

mpi & mpif.h: INTEGER comm, rank, ierror

INTENT(IN/OUT)
is omitted
on these slides

• Python: *rank* = comm.Get_rank()



CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierror)

Size

- How many processes are contained within a communicator?

C

Fortran

Python

- C/C++: `int MPI_Comm_size(MPI_Comm comm, int *size)`
- Fortran: `MPI_COMM_SIZE(comm, size, ierror)`
mpi_f08: `TYPE(MPI_Comm) :: comm`
`INTEGER :: size`
`INTEGER, OPTIONAL :: ierror`
mpi & mpif.h: `INTEGER comm, size, ierror`
- Python: `size = comm.Get_size()`

Fortran & C: Interface definitions

- On these slides & in the MPI standard
- You have to write the corresponding procedure **calls**
- E.g., in C:
`MPI_Comm_size (MPI_COMM_WORLD, &size);`
- E.g., in Fortran:
`CALL MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierror)`

Python: Mix of usage of the interface and typed argument list

- See [MPI for Python \(mpi4py.github.io\)](https://mpi4py.github.io), and
[MPI for Python 3.1.1 documentation \(mpi4py.readthedocs.io\)](https://mpi4py.readthedocs.io), and
[The API reference \(mpi4py.github.io/apiref/index.html\)](https://mpi4py.github.io/apiref/index.html)

Exercise 2: I am my_rank of size

In MPI/tasks/...

- Use: **C** C/Ch2/myrank-skel.c or **Fortran** F_30/Ch2/myrank-skel_30.f90
or **Python** PY/Ch2/myrank-skel.py
- Modify this program so that
 - every process writes its rank and the size of MPI_COMM_WORLD,
 - only process ranked 0 in MPI_COMM_WORLD prints “hello world”.
- Why is the sequence of the output non-deterministic?

```
I am 2 of 4
Hello world
I am 0 of 4
I am 3 of 4
I am 1 of 4
```

During the Exercise (15 min.)



Please stay here in the main room while you do this 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

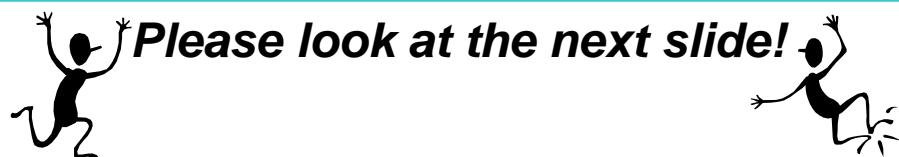


As soon as you finished the exercise,
please go to your breakout room

and continue your discussions with your fellow learners:



How can you resolve the question of the following advanced exercise?



Exercise 3 – Advanced Exercises: Hello World with deterministic output

- Discuss with your neighbor or in your break out group, what must be done, that the output of all MPI processes on the terminal window is in the sequence of the ranks.
- Or is there no chance to guarantee this?

Exercise 4: Version test

- Copy the version test programs into your local working directory
 - **C** C/Ch2/ and **Fortran** F_30/Ch2 and **Python** PY/Ch2 contain following version test programs
- Compile and run → **Besides the version of MPI, it also tests ...**
 - version_test.c → ... exists mpi.h and the C bindings
 - version_test_11.f → ... exists mpif.h and the Fortran bindings
 - version_test_20.f90 → ... exists the mpi module + bindings
 - version_test_30.f90 → ... exists the mpi_f08 module + bindings
 - version_test_keyarg_20.f90 → Contains the mpi module a correct bindings according to MPI-3.0 and higher, i.e., allowing also keyword based argument lists?
 - version_test_keyarg_30.f90 → Same test for the mpi_f08 module
 - version_test.py

See also
course
Chapter 5
**The New
Fortran
Module
mpi_f08**

Python

Previous exercises' directory

In the **previous courses** (until 6/2020 and without Python) we used **another directory**:

- C examples `~/MPI/course/C/Ch[2-13]/*.c`
- Fortran examples `~/MPI/course/F_[123]*/Ch[2-13]/*.f*`

C
Fortran

- Make sure you have completed the introductory exercises fully **before** checking the solution, otherwise you will lose out on 90% of the learning benefits.
- Time permitting, attempt to complete the advanced exercises and study their solutions.

- F_11 (*.f)
 - MPI-1.1
 - with mpif.h
- F_20 (*.f90)
 - MPI-2.x
 - mpi module
- F_30 (*.f90)
 - MPI-3.x
 - mpi_f08

Especially `~/MPI/course/F_11/Ch1` and `~/MPI/course/F_20/Ch1` contain the version tests for

- the old Fortran mpif.h include file (the usage is strongly discouraged), and for
- the old Fortran mpi module.

For private notes

For private notes

Chap.3 Messages and Point-to-Point Communication

1. MPI Overview
2. Process model and language bindings

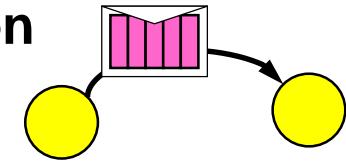
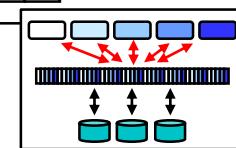
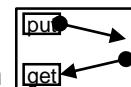
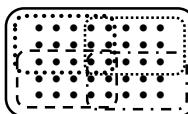
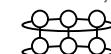


`MPI_Init()`
`MPI_Comm_rank()`

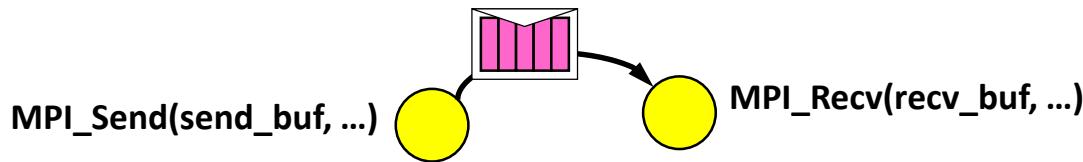
3. Messages and point-to-point communication

- the MPI processes can communicate

4. Nonblocking communication
5. The New Fortran Module `mpi_f08`
6. Collective communication
7. Error Handling
8. Groups & communicators, environment management
9. Virtual topologies
10. One-sided communication
11. Shared memory one-sided communication
12. Derived datatypes
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

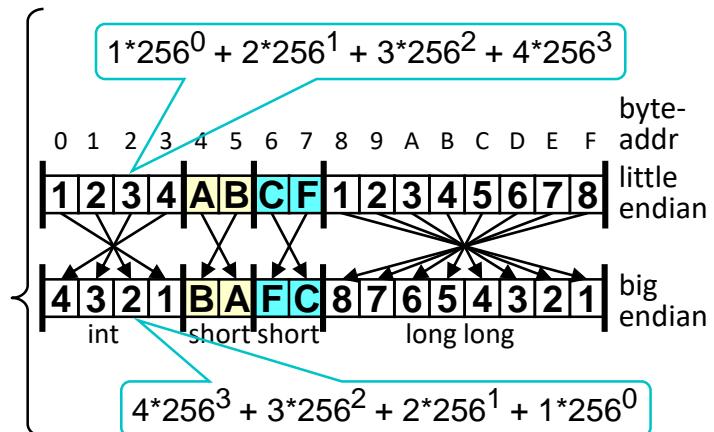


Major decisions — performance and functionality

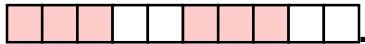


Important questions / decisions

- How to address the destination process?
- Which message content?
 - How to handle strided data?
 - Data conversion in inhomogeneous cluster
- Which protocol?
 - When must the **send** be completed?
 - As soon as possible → buffered protocol → low latency 😊 / bad bandwidth ☹
 - Only after the receive is called → synchronous protocol
→ sending process **blocked** by receiver → high latency ☹ / good bandwidth 😊
 - Application guarantees that corresponding receive is already called
→ direct protocol → sending process is **not blocked** / good bandwidth 😊
 - Automatic decision → may be blocked / low latency 😊 / good bandwidth 😊



Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
 - Basic datatype.
 - Derived datatypes 
- Derived datatypes can be built up from basic or derived datatypes.
- C types are different from Fortran types.
- Datatype handles are used to describe the type of the data in the memory.

Example: message with 5 integers

2345	654	96574	-12	7676
------	-----	-------	-----	------

- **Python:** messages can be stored in
 - Objects → using `send(...)`, `recv(...)`, ... mpi4py routines → slow object serialization
 - Buffers as numPy arrays → using `Send(...)`, `Recv(...)`, ... → fast communication

For other alternatives, see MPI\tasks\PY\Ch13\mpi_io_exa1-skel.py

Lower-case methods

Upper-case methods

MPI Basic Datatypes — C / C++

MPI Datatype handle	C datatype	Remarks
MPI_CHAR	char	Treated as printable character
MPI_SHORT	signed short int	
MPI_INT	signed int	
MPI_LONG	signed long int	
MPI_LONG_LONG	signed long long	
MPI_SIGNED_CHAR	signed char	Treated as integral value
MPI_UNSIGNED_CHAR	unsigned char	Treated as integral value
MPI_UNSIGNED_SHORT	unsigned short int	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_UNSIGNED_LONG_LONG	unsigned long long	
MPI_FLOAT	float	
MPI_DOUBLE	double	
MPI_LONG_DOUBLE	long double	
MPI_BYTE		
MPI_PACKED		

Further datatypes,
see, e.g., MPI-
3.1/4.0, Annex A.1

Includes also
special C++ types,
e.g., bool, see
MPI-3.1 page 674,
MPI-4.0 page 862

Python

All datatype handles can be used, syntax: e.g., MPI.FLOAT

MPI Basic Datatypes — Fortran

MPI Datatype handle	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Further datatypes,
e.g.,
MPI_REAL8 for
REAL*8,
see MPI-3.1/4.0,
Annex A.1

2345 654 96574 -12 7676

Arguments for MPI send/recv

count=5

datatype=MPI_INTEGER

Declaration of the buffers

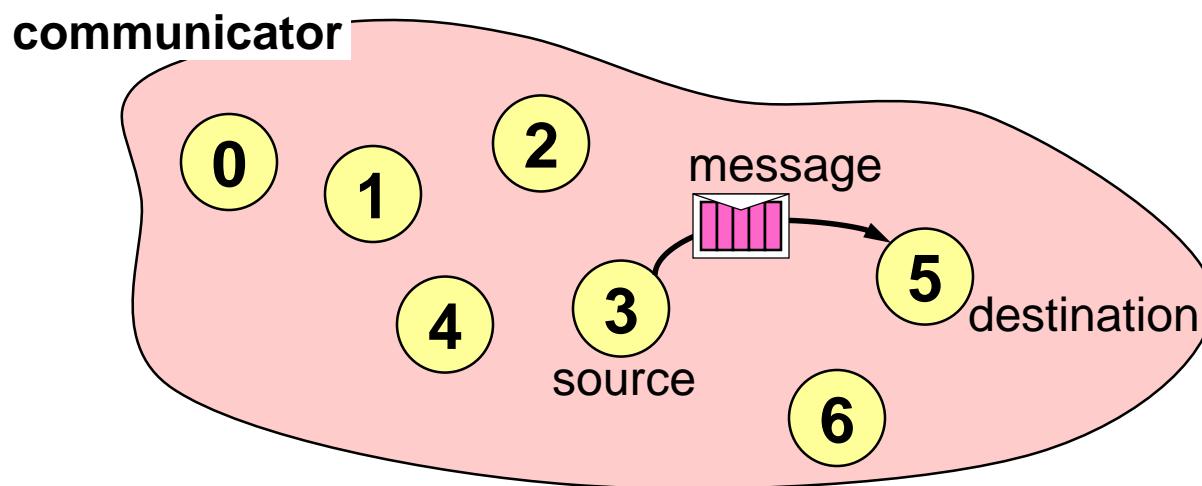
INTEGER arr(5)

For KIND-parameterized Fortran types, basic datatype handles must be generated with

- MPI_TYPE_CREATE_F90_INTEGER
- MPI_TYPE_CREATE_F90_REAL
- MPI_TYPE_CREATE_F90_COMPLEX

Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator, e.g., MPI_COMM_WORLD.
- Processes are identified by their ranks in the communicator.



Sending a Message

C

- C/C++: `int MPI_Send(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)`

Fortran

- Fortran: `MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)`
`mpi_f08:`
`TYPE(*), DIMENSION(..) :: buf`
`TYPE(MPI_Datatype) :: datatype;` `TYPE(MPI_Comm) :: comm`
`INTEGER :: count, dest, tag;` `INTEGER, OPTIONAL :: ierror`
`mpi & mpif.h: <type> buf(*); INTEGER count, datatype, dest, tag, comm, ierror`
- Python: `comm.Send(buf, int dest, int tag=0)`
`comm.send(obj, int dest, int tag=0)`

Python

- buf is the starting point of the message with count elements, each described with datatype.
- dest is the rank of the destination process within the communicator comm.
- tag is an additional nonnegative integer piggyback information, additionally transferred with the message.
- The tag can be used by the program to distinguish different types of messages.
- Python:
 - buf must implement the Python buffer protocol, e.g., numPy arrays
 - buf can be buf or (buf, datatype) or (buf, count, datatype)
 - with C datatypes in Python syntax, e.g., MPI.INT, MPI.FLOAT, ...
 - obj is any Python object that can be serialized with the pickle method

Receiving a Message

C

Fortran

Python

- C/C++: `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- Fortran: `MPI_RECV(buf,count,datatype, source, tag, comm, status, ierror)`
`mpi_f08:`
`TYPE(*), DIMENSION(..) :: buf`
`INTEGER :: count, source, tag`
`TYPE(MPI_Datatype) :: datatype;` `TYPE(MPI_Comm) :: comm`
`TYPE(MPI_Status) :: status;` `INTEGER, OPTIONAL :: ierror`
`mpi & mpif.h: <type> buf(*); INTEGER count, datatype, source, tag, comm, ierror`
`INTEGER status(MPI_STATUS_SIZE)`
- Python: `comm.Recv(buf, int source=ANY_SOURCE, int tag=ANY_TAG, Status status=None)`
`obj = comm.recv(buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,` `Status status=None)`
`buf is only a temporary buffer, deprecated since version 3.0.0`

- `buf/count/datatype` describe the receive buffer.
- Receiving the message sent by process with rank `source` in `comm`.
- Envelope information is returned in `status`.
- One can pass `MPI_STATUS_IGNORE` instead of a status argument.
- Output arguments are printed *blue-cursive*.
- **Message matching rule:** receives only if `comm`, `source`, and `tag` match.
- Python: `Send` requires that the matching receive is a `Recv` / ditto for `send` and `recv`

count, datatype
is not part of this
matching rule

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Type matching:


```
float sndbuf[n];
MPI_Send(sndbuf, n, MPI_FLOAT,...)
```

```
float rcvbuf[n];
MPI_Recv(rcvbuf, n, MPI_FLOAT,...)
```

 - ① Send-buffer's (C or Fortran) type must match with the send datatype handle
 - ② Send datatype handle must match with the receive datatype handle
 - ③ Receive datatype handle must match with receive-buffer's (C or Fortran) type
- Tags must match → typical usage: **different tags for different data**


```
#define TAG_velocity 111
MPI_Send( velocity_sndbuf, ... TAG_velocity, ...)
```

```
#define TAG_velocity 111
MPI_Recv( velocity_rcvbuf, ... TAG_velocity, ...)
```

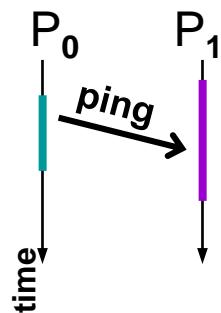
 - ①
 - ②
 - ③

→ The velocity message will never be received in, e.g., a temperature array
- Receiver's buffer must be large enough.

Exercise 1 — One Ping

In MPI/tasks/...

- Use: **C** C/Ch3/ping-skel.c or **Fortran** F_30/Ch3/ping-skel_30.f90
or **Python** PY/Ch3/ping-skel.py (hint: use **send** & **recv**)
- Write a program according to the time-line diagram:
 - Process 0 sends a message to process 1 (ping)
- We prepare a benchmark program → don't care on buffer contents
 - Just send 1 float (in C) / REAL (in Fortran) / [None] (in Python)



rank=0

rank=1

print("0: before send ping")

Send (dest=1)

(tag=17)

Recv (source=0)

print("1: after recv ping")

```
if (my_rank==0)          /* i.e., emulated multiple program */
    MPI_Send( ... dest=1 ...)
else
    MPI_Recv( ... source=0 ...)
fi
```

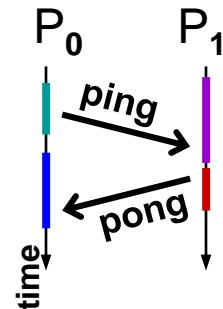
Start with only 2
processes:
mpirun -np 2 ...

Exercise 2 — One ping pong

- Before starting this exercise 2, you should have **compared your result of exercise 1 with ping.c / _30.f90 / .py** in the solution sub-directory

Exercise 2:

- Use: **C** C/Ch3/pingpong-skel.c or **Fortran** F_30/Ch3/pingpong-skel_30.f90
or **Python** PY/Ch3/pingpong-skel.py (hint: use **send** & **recv**)
- Write a program according to the time-line diagram:
 - process 0 sends a message to process 1 (ping)
 - after receiving this message,
process 1 sends a message back to process 0 (pong)
- For details, see next slide



Exercise 2 — One ping pong

rank=0

```
print("0: before send ping")
```

```
Send (dest=1)
```

(tag=17)

rank=1

```
Recv (source=0)
```

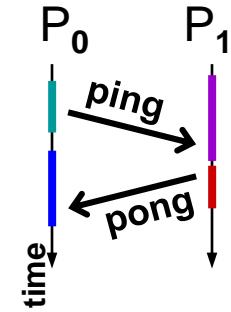
```
print("1: after recv ping")
```

```
print("1: before send pong")
```

```
Send (dest=0)
```

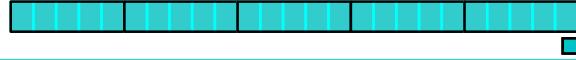
Recv (source=1)

```
print("0: after recv pong")
```



```
if (my_rank==0)          /* i.e., emulated multiple program */
    MPI_Send( ... dest=1 ...)
    MPI_Recv( ... source=1 ...)
else
    MPI_Recv( ... source=0 ...)
    MPI_Send( ... dest=0 ...)
fi
```

During the Exercise (25 min.)



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

And have fun with this longer 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:

If you are fast, then there is an **advanced exercise** waiting for you → next slide!

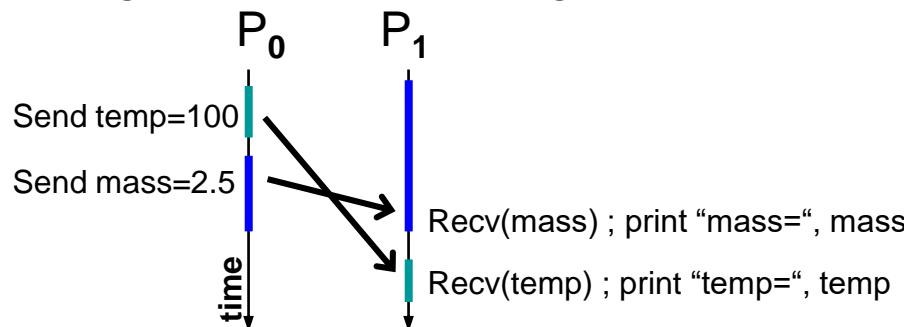


How do you like this programming of writing software parts
that should be **executed in parallel** by different process
separated by such `if(my_rank...)` **one after the other**
(i.e., somehow sequentially)  into your source code?



Advanced Exercise 2b — Overtaking messages

- Use: **C** C/Ch3/overtake-skel.c or **Fortran** F_30/Ch3/overtake-skel_30.f90
or **Python** PY/Ch3/overtake-skel.py (hint: use **send** & **recv**)
- Write a program according to the time-line diagram:



- Use float in C / REAL in Fortran for temp and mass
- 1st test: use same tags for both messages → expected: wrong result
- 2nd test: use different tags → correct result

Remarks:

- The complete rules for overtaking messages will come at the end of the chapter.
- Solutions: **C** / **F_30** / **PY**/Ch3/solutions/overtake.c / _30.f90 / .py
- Later we'll learn that this program may also cause a deadlock, because MPI_Send may synchronize; see additional solutions **overtake-arr.c** / **-arr_30.f90** / **-arr.py**



Wildcarding

- Receiver can wildcard.
 - To receive from any source — source = MPI_ANY_SOURCE
 - To receive from any tag — tag = MPI_ANY_TAG
 - Actual source and tag are returned in the receiver's status parameter.
-
- With info assertions New in MPI-4.0
 - "mpi_assert_no_any_source" = "true" and/or
 - "mpi_assert_no_any_tag" = "true"

stored on the communicator using MPI_Comm_set_info(),

 - an MPI application can tell the MPI library that it will never use MPI_ANY_SOURCE and/or MPI_ANY_TAG on this communicator
→ may enable lower latencies.
 - Other assertions:
 - "mpi_assert_exact_length" = "true" → receive buffer must have exact length
 - "mpi_assert_allow_overtaking" = "true" → message order need not to be preserved
- Further details, see course chapter 8-(2) *MPI_Info Object* 

Communication Envelope

- Envelope information is returned from MPI_RECV in *status*.
- C/C++:

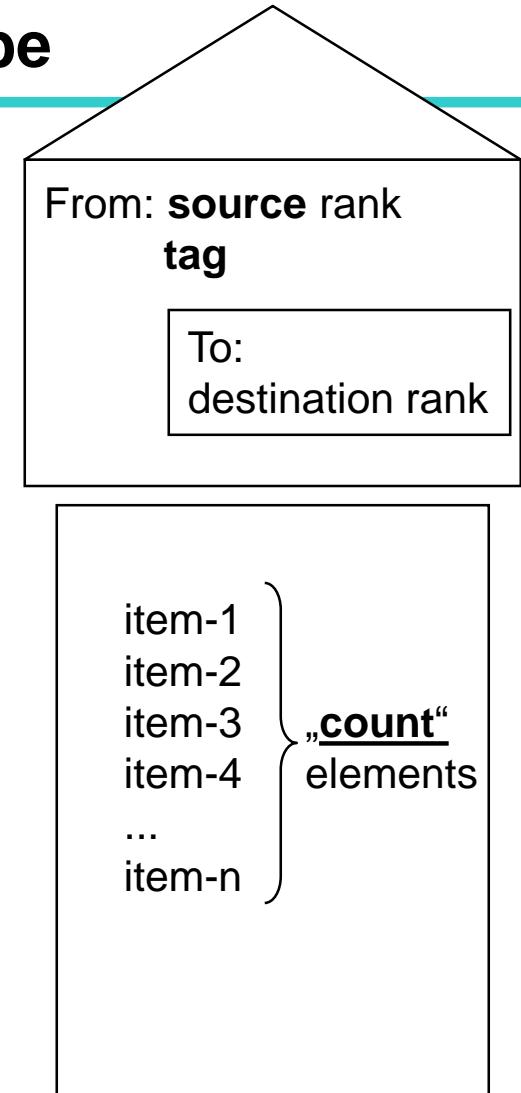
```
MPI_Status status;
status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR    *)
```
- Fortran:

```
mpi_f08:  TYPE(MPI_Status) :: status
           status%MPI_SOURCE
           status%MPI_TAG
           status%MPI_ERROR    *)
```
- mpi & mpif.h:

```
INTEGER status(MPI_STATUS_SIZE)
               status(MPI_SOURCE)
               status(MPI_TAG)
               status(MPI_ERROR)   *)
```
- Python:

```
status.Get_source()
status.Get_tag(), ...
count via MPI_GET_COUNT()
```

**) See slide on MPI_Waitall, ... in course Chapter 4 Nonblocking Communication*



Receive Message Count

C

- C/C++: `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`

Fortran

- Fortran: `MPI_GET_COUNT(status, datatype, count, ierror)`

mpi_f08:
 `TYPE(MPI_Status) :: status`
 `TYPE(MPI_Datatype) :: datatype`
 `INTEGER :: count`
 `INTEGER, OPTIONAL :: ierror`

mpi & mpif.h: `INTEGER status(MPI_STATUS_SIZE), datatype, count, ierror`

Python

- Python: `count = status.Get_count(Datatype datatype=BYTE)`

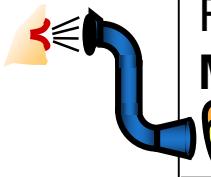
Caution:

```
buf = np.zeros((100,), dtype=np.double)
comm.Send((buf, 5, MPI.DOUBLE), ....)
comm.Recv((buf, 100, MPI.DOUBLE), ...., status)
count = status.Get_count(MPI.DOUBLE) # → 5
count = status.Get_count() # → 40
```

Communication Modes

- Send communication modes:
 - synchronous send → MPI_SSEND
 - buffered [asynchronous] send → MPI_BSEND
 - standard send → MPI_SEND
 - Ready send → MPI_RSEND
- Receiving all modes → MPI_RECV

Communication Modes — Definitions

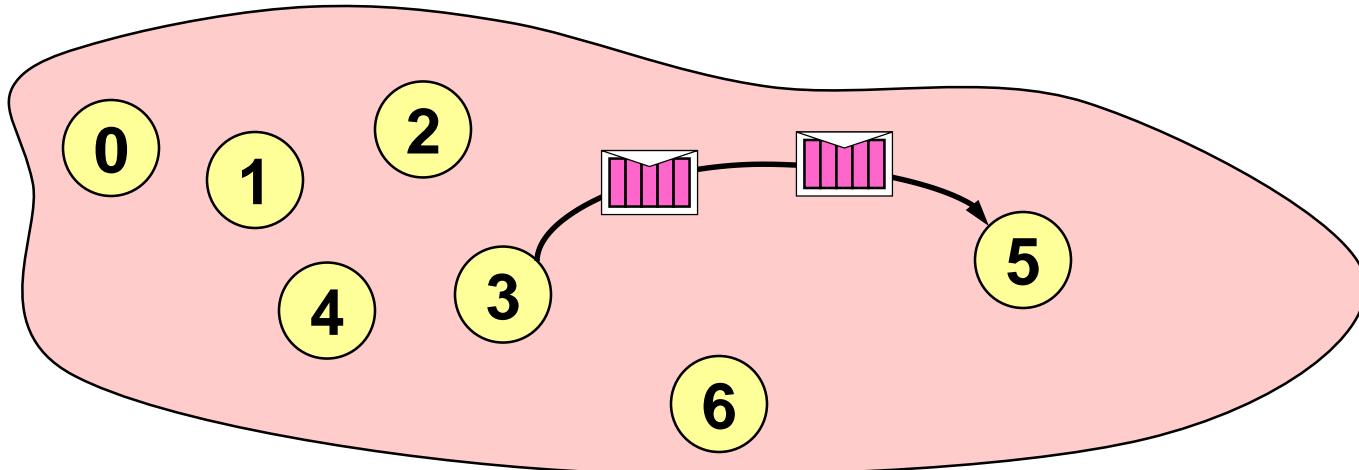
Sender mode	Definition	Notes
 Synchronous send MPI_SSEND	Only completes when the receive has started	
 Buffered send MPI_BSEND	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
 Standard send MPI_SEND	Either synchronous or buffered	uses an internal buffer
 Ready send MPI_RSEND	May be started only if the matching receive is already posted!	highly dangerous!
Receive MPI_RECV	Completes when a message has arrived	same routine for all communication modes

Rules for the communication modes

- Standard send (**MPI_SEND**)
 - minimal transfer time
 - may block due to synchronous mode
 - all risks of synchronous send
- Synchronous send (**MPI_SSEND**)
 - risk of deadlock
 - risk of serialization
 - risk of waiting → idle time
 - high latency / best bandwidth
- Buffered send (**MPI_BSEND**)
 - low latency / bad bandwidth
- Ready send (**MPI_RSEND**)
 - use **never**, except you have a *200% guarantee* that Recv is already called in the current version and all future versions of your code,
 - may be the fastest,
 - for a use case, see later → Chapter 4 (nonblocking) → Quiz C

Message Order Preservation

- Rule for messages on the same connection,
i.e., same communicator, source, and destination rank:
- **Messages do not overtake each other.**
- This is true even for non-synchronous sends.



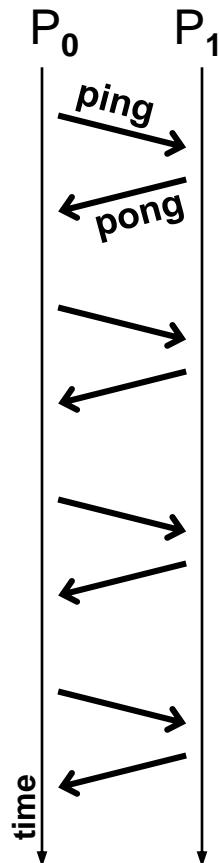
- If both receives match both messages, then the order is preserved.

Exercise 3 — Ping pong benchmark

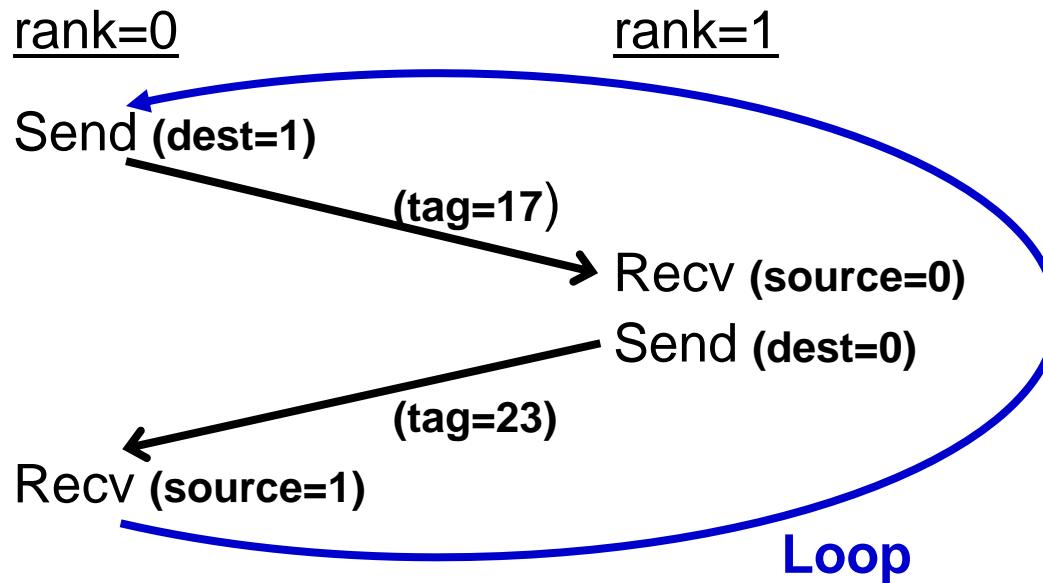
Use: **C** C/Ch3/pingpong-bench-skel.c or **Fortran** F_30/Ch3/pingpong-bench-skel_30.f90
Python PY/Ch3/pingpong-bench-skel.py

- Write a program according to the time-line diagram:
 - process 0 sends a message to process 1 (ping)
 - after receiving this message,
process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Add timing calls before and after the loop:
- **C/C++**: *double MPI_Wtime(void);*¹⁾
- **Fortran**: *DOUBLE PRECISION FUNCTION MPI_WTIME()*
- **Python**: *time = MPI.Wtime()*
- **MPI_WTIME** returns a wall-clock time in seconds.
- Only at process 0,
 - print out the transfer time of **one** message
 - in μs , i.e., $\text{delta_time} / (2*50) * 1\text{e}6$
- See also next slide

¹⁾ One of the rare routines that can be implemented as macros in C,
see MPI-3.1 / MPI-4.0, Sect.2.6.4, page 20 / 26

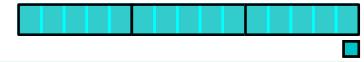


Exercise 3 — Ping pong benchmark



```
if (my_rank==0)          /* i.e., emulated multiple program */
    MPI_Send( ... dest=1 ...)
    MPI_Recv( ... source=1 ...)
else
    MPI_Recv( ... source=0 ...)
    MPI_Send( ... dest=0 ...)
fi
```

During the Exercise (15 min.)



Please stay here in the main room while you do this 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 **compare your measured latency** with your colleagues?

Please stay in the break out room and **continue with exercise 4 (next slide)!**

Now, you should have a more precise and reproducible latency?



And now, make a copy of your program and
use 2x synchronous MPI_Ssend instead of 2x standard MPI_Send.

Significantly longer latency?



Exercises 4+5 (advanced): Ping pong latency and bandwidth

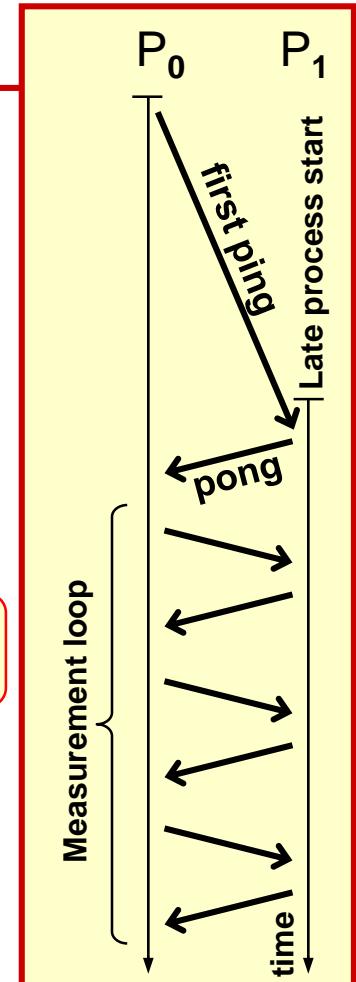
Exercise 4

- Exclude startup time problems from measurements:
 - Execute a first ping-pong outside of the measurement loop
- Solution: MPI/tasks/C/Ch3/solutions/pingpong-bench1.c
and MPI/tasks/F_30/Ch3/solutions/pingpong-bench1_30.f90
and MPI/tasks/PY/Ch3/solutions/pingpong-bench1.py

Exercise 5

- latency = transfer time for short messages
- bandwidth = message size (in bytes) / transfer time
- Print out message transfer time and bandwidth
 - for following send modes:
 - for standard send (`MPI_Send`)
 - for synchronous send (`MPI_Ssend`)
 - for following message sizes:
 - 8 bytes (e.g., one double or double precision value)
 - 512 B (= 8*64 bytes)
 - 32 kB (= 8*64**2 bytes)
 - 2 MB (= 8*64**3 bytes)
- Solution: MPI/tasks/C/Ch3/solutions/ping_pong_advanced_send.c / ...ssend...
and MPI/tasks/F_30/Ch3/solutions/ping_pong_advanced_send_30.f90 / ...ssend...
and MPI/tasks/PY/Ch3/solutions/ping_pong_advanced_send_obj.py / _ssend_obj.py / _Send.py / _Ssend.py

C unlimit or
 ulimit -s 200000
 once before calling mpirun



Optimized with `Send` / `Recv`
instead of `send` / `recv`

Quiz on Chapter 3 – Point-to-point communication

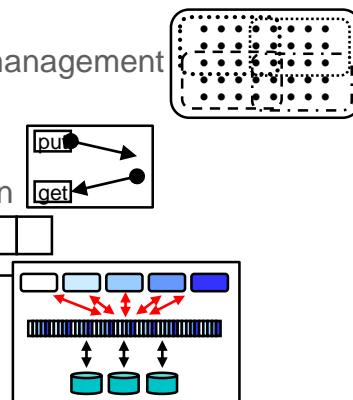
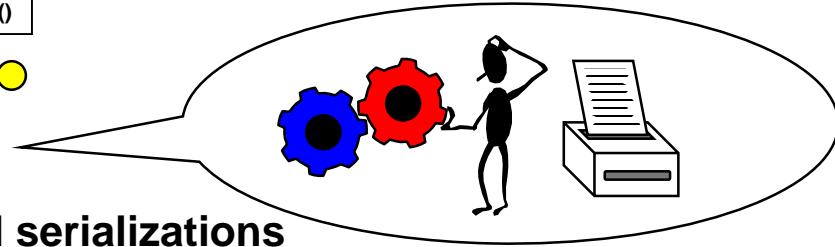
- A. How many different MPI point-to-point send protocols (=blocking APIs) exist?
- B. Which one requires that you first use MPI_Buffer_attach?
- C. Which one is recommended for smallest latency and highest bandwidth both together?
- D. If your buffer is an array `buf` with 5 double precision values that you want to send?
How do you describe your message in the call to MPI_Send
 - in C?
 - in Fortran?
- E. When calling MPI_Recv to receive this message which count values would be correct?
- F. When I use one of the MPI send routines, how many messages do I send?
- G. Which is the predefined communicator that can be used to exchange a message from process rank 3 to process rank 5?
- H. If you send two messages msg1 and msg2 from rank 3 to rank 5, is it possible that the second one can overtake, i.e., be received before the first one?
- I. Do you remember the major risks of synchronous send?
- J. Has standard send the same risks?
- K. What is the major use case for tags?

For private notes

For private notes

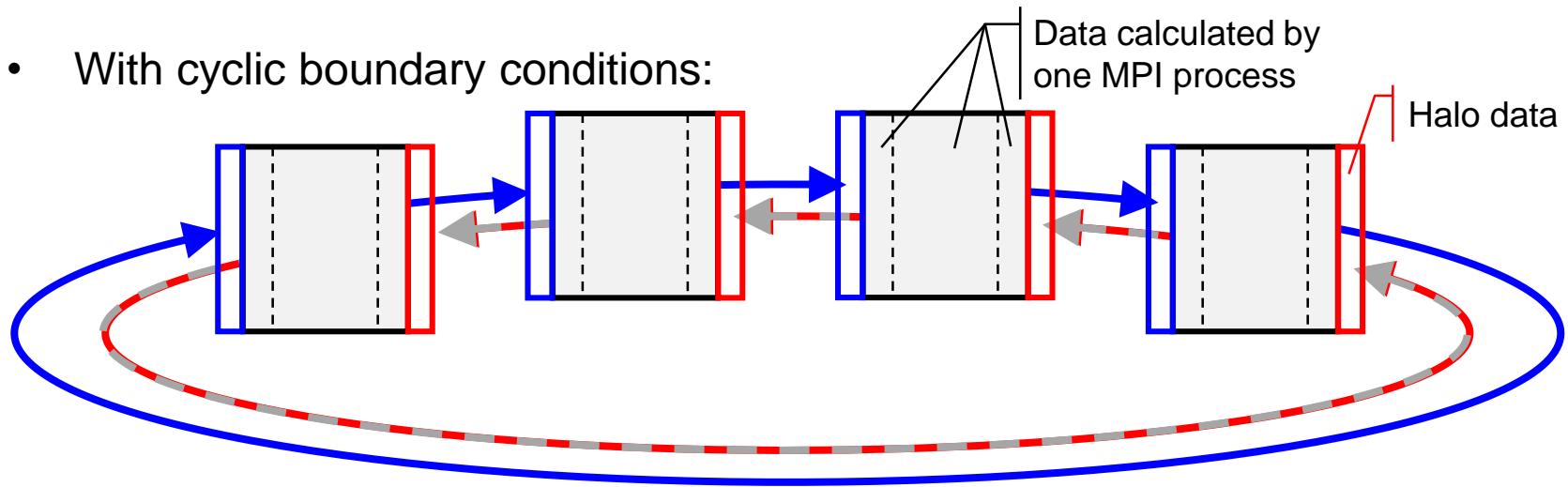
Chap.4 Nonblocking Communication

1. MPI Overview 
2. Process model and language bindings
`MPI_Init()`
`MPI_Comm_rank()`
3. Messages and point-to-point communication
- 4. Nonblocking communication**
 - **to avoid idle time, deadlocks and serializations**
5. The New Fortran Module mpi_f08
6. Collective communication
7. Error Handling
8. Groups & communicators, environment management
9. Virtual topologies
10. One-sided communication
11. Shared memory one-sided communication
12. Derived datatypes
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

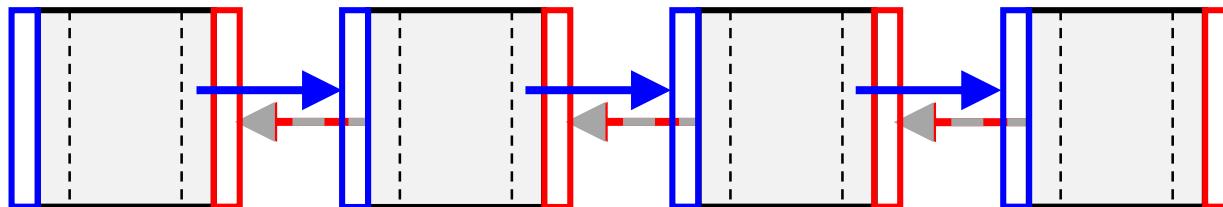


Task: Halo Communication

- With cyclic boundary conditions:



- Non-cyclic:



Let's concentrate on the blue direction (from left to right, clockwise)

Blocking Routines → Risk of Deadlocks & Serializations

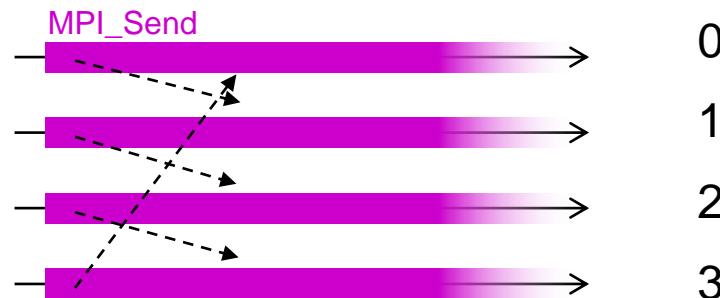
For cyclic boundary:

```
MPI_Send(..., right_rank, ...)  
MPI_Recv( ..., left_rank, ...)
```

For non-cyclic boundary:

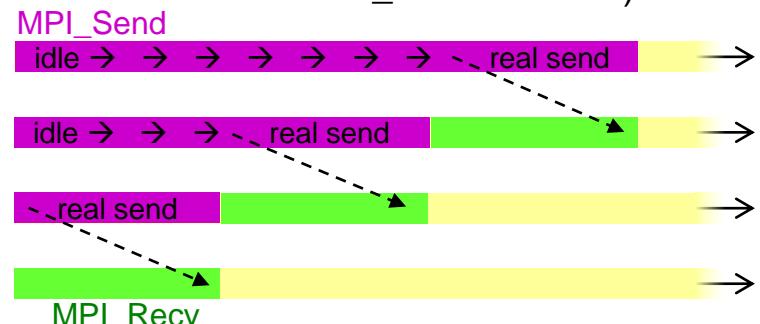
```
if (myrank < size-1)  
    MPI_Send(..., left, ...);  
if (myrank > 0)  
    MPI_Recv( ..., right, ...);
```

Timelines of all processes



→ Deadlock

(If the MPI library chooses the synchronous protocol, i.e. MPI_Send waits until MPI_Recv is called)



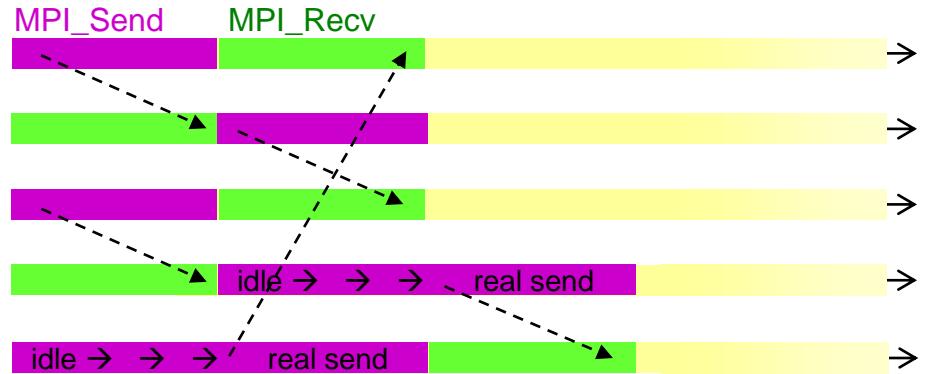
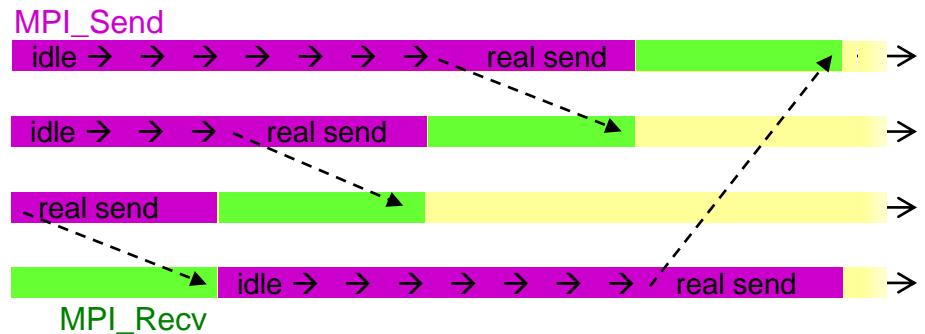
→ Serialization

(If the MPI library chooses the synchronous protocol)

Cyclic communication – other bad ideas

```
if (myrank < size-1) {  
    MPI_Send(..., left, ...);  
    MPI_Recv( ..., right, ...);  
} else {  
    MPI_Recv( ..., right, ...);  
    MPI_Send(..., left, ...);  
}
```

```
if (myrank%2 == 0) {  
    MPI_Send(..., left, ...);  
    MPI_Recv( ..., right, ...);  
} else {  
    MPI_Recv( ..., right, ...);  
    MPI_Send(..., left, ...);  
}
```



→ **Serialization**

(If the MPI library chooses
the synchronous protocol)



Non-Blocking Communications

Separate communication into **three phases**:

- Initiate nonblocking communication
 - returns immediately
 - routine name starting with MPI_I...
- it is local,
i.e., it returns independently of any other process' activity
- Do some work (perhaps involving other communications?)
- Wait for nonblocking communication to **complete**, i.e.,
 - the send buffer is read out, or
 - the receive buffer is filled in

“I” stands for

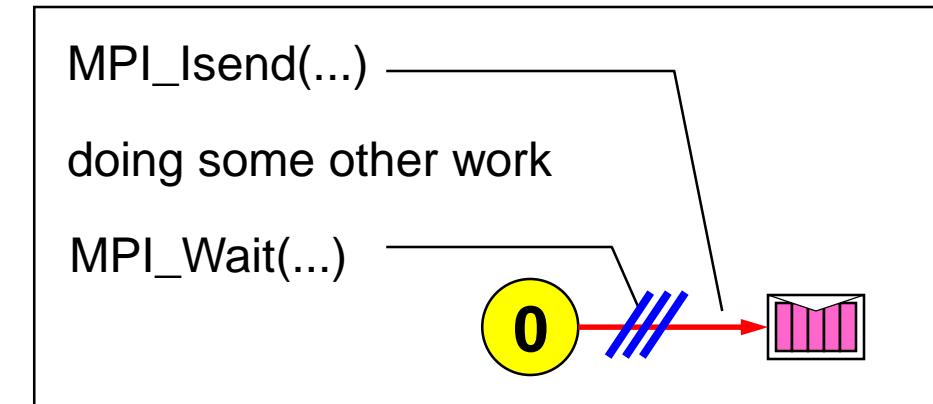
- Immediately (=local) and
- Incomplete (=nonblocking¹⁾)

¹⁾ The definition of nonblocking is clarified in MPI-4.0

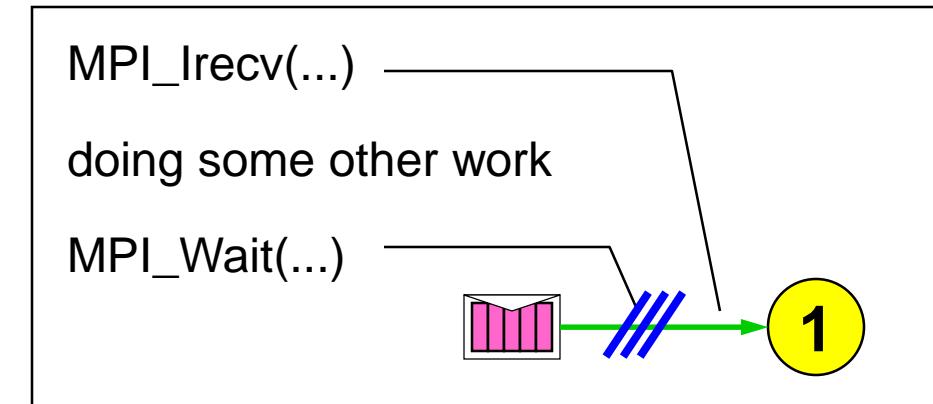
→ course Chapter 15 *Probe, Persistent Requests, Request_free, Cancel*

Non-Blocking Examples

- Nonblocking **send**



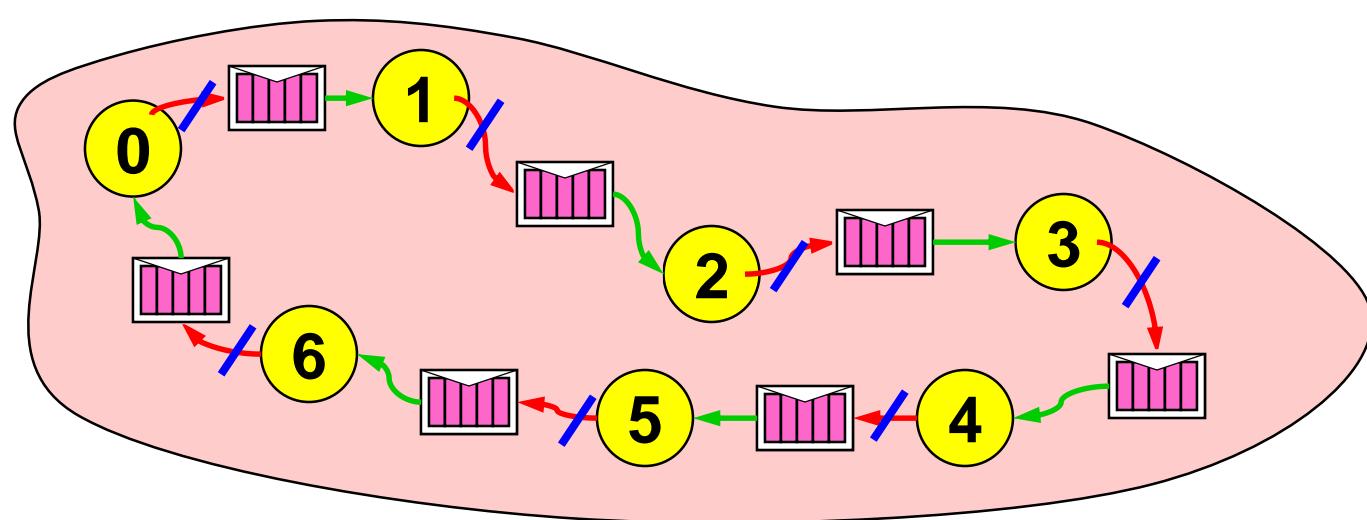
- Nonblocking **receive**



= waiting until operation locally completed

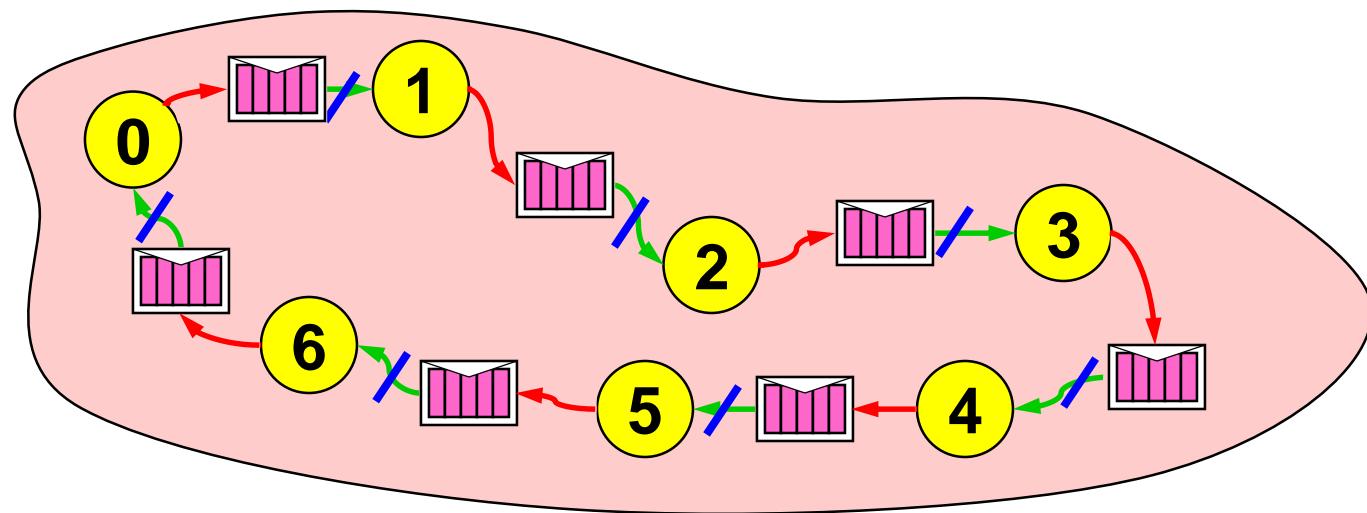
Non-Blocking Send

- Initiate nonblocking send
 - in the ring example: Initiate nonblocking send to the right neighbor
- Do some work:
 - in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for nonblocking send to complete ↔

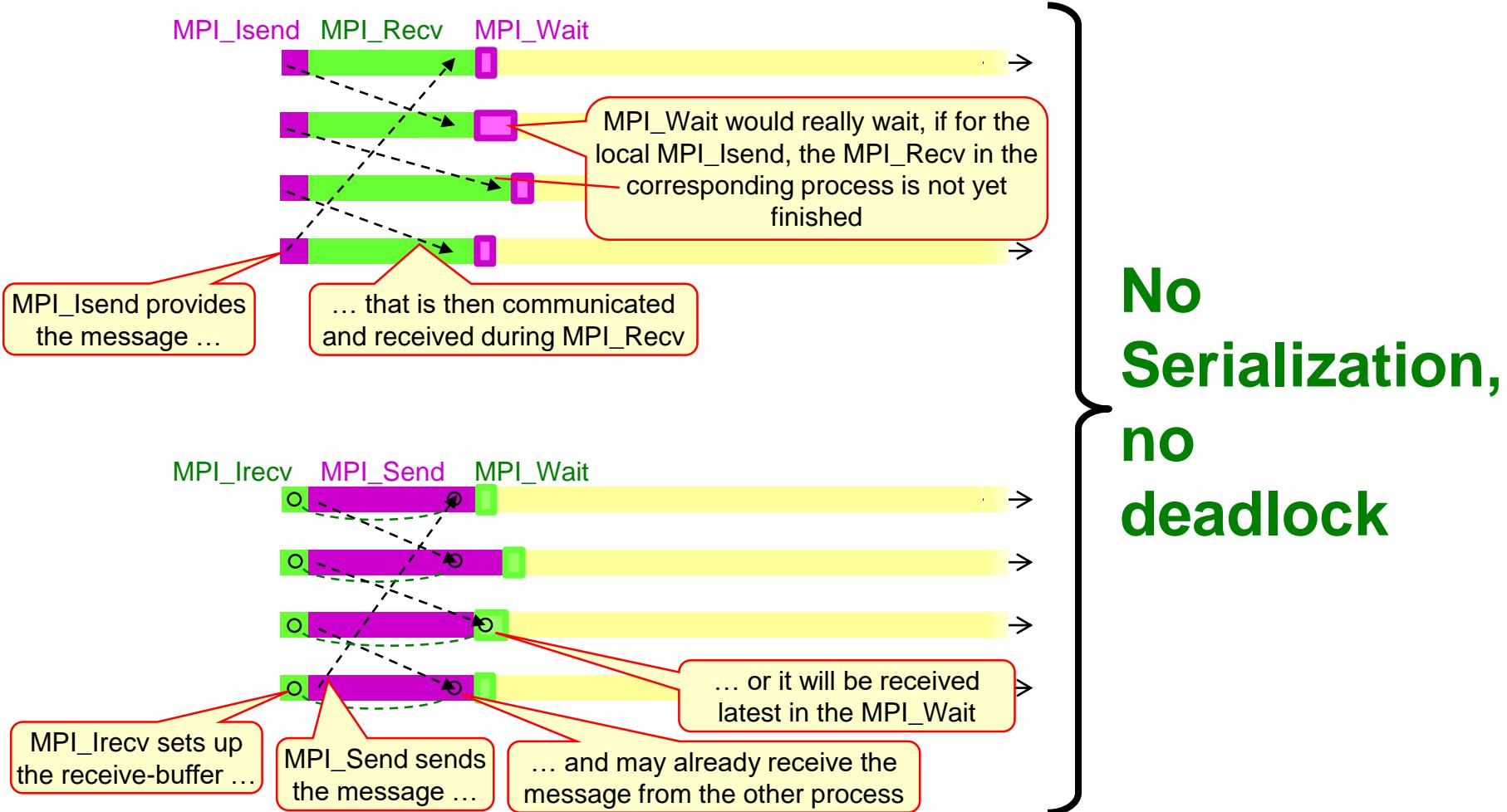


Non-Blocking Receive

- Initiate nonblocking receive
 - in the ring example: Initiate nonblocking receive from left neighbor
- Do some work:
 - in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for nonblocking receive to complete ↔



Timelines for both solutions

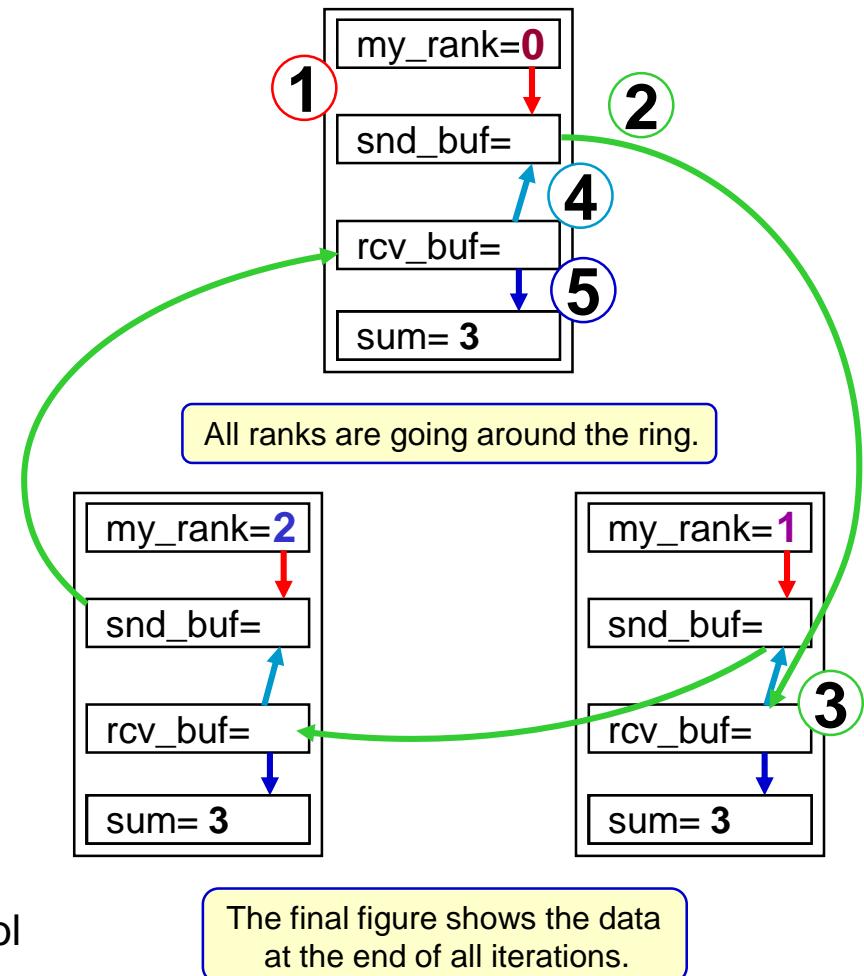


Exercise 1 — Rotating information around a ring

- This exercise is a **preparation of many exercises** during the whole course.
- It is the **smallest example of halo communication**:
 - `snd_buf` = the real data that is to sent to the neighbor
 - `rcv_buf` = a halo
- Instead of huge double precision arrays, our data is just **1 integer**
- It is communicating in a ring.
- Our source code is **wrong**, because it uses `MPI_Send` and `MPI_Recv` in a naïve way.
- Wrong programs may work ☺, here because we send only a small message.
- But it is still wrong ☹. In MPI/tasks/...
- We start with **C** `C/Ch4/ring-WRONG2.c` or
Fortran `F_30/Ch4/ring-WRONG2_30.f90` or **Python** `PY/Ch4/ring-WRONG2.py`
- It uses two different buffers for send and receive.  src2 back Object serialization substituted by direct communication in python
- Just for fun, the program send the `my_rank` values around the ring in a loop with `#process` iterations and sums up all values coming along.
- The following two slides explain, how the program works.

Exercise 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- 1 Each process stores its rank in MPI_COMM_WORLD into an integer variable `snd_buf`.
- 2 + 3 Each process passes this on to its neighbor on the right.
- 4 Preparation of next iteration.
- 5 Each process calculates the sum of all values.
- Repeat “2-5” with “size” iterations (size = number of processes), i.e.,
 - each process calculates sum of all ranks.
 - ring-WRONG2.c and _30.f90 programs use blocking MPI_Send and MPI_Recv
 - i.e., they will deadlock if MPI_Send is implemented with a synchronous protocol



Recap from previous course chapter:



Standard send
`MPI_SEND`

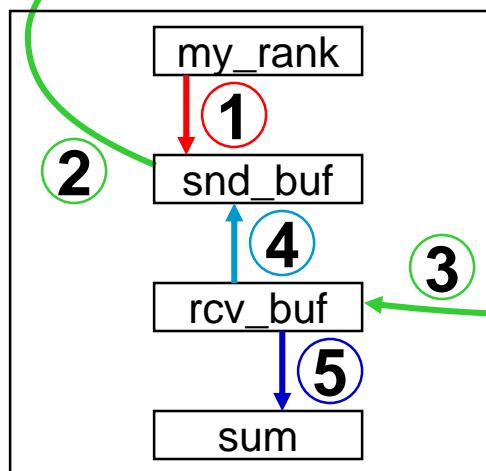
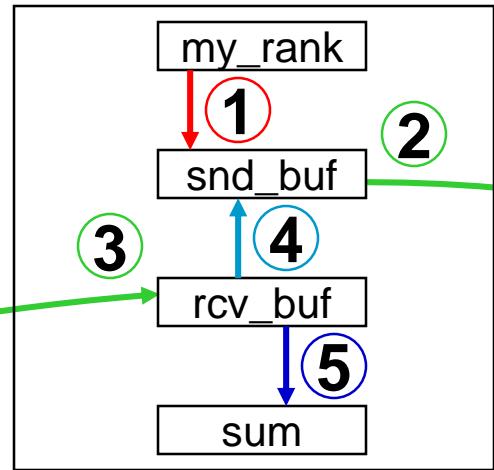
Either synchronous
or buffered

Exercise 1 — Rotating information around a ring

Initialization: 1

Each iteration:

2 3 4 5



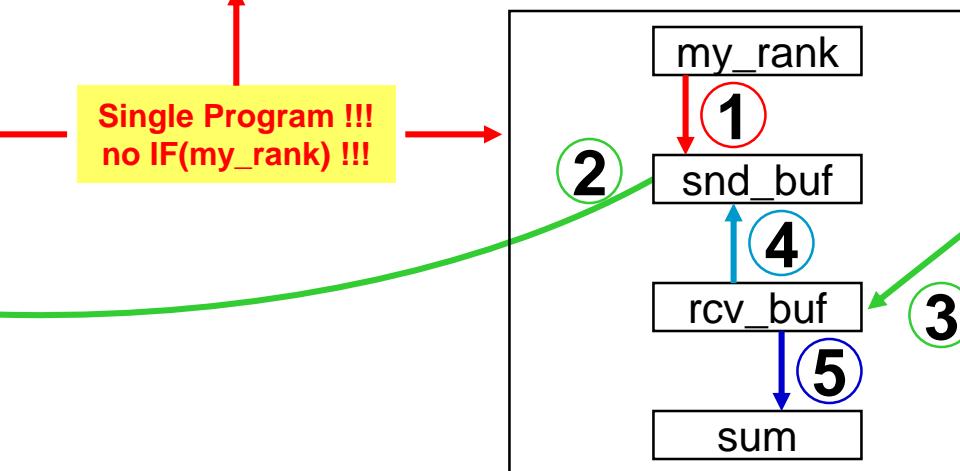
Fortran:

```
dest    = mod(my_rank+1,size)  
source = mod(my_rank-1+size,size)
```

C/C++:

```
dest    = (my_rank+1) % size;  
source = (my_rank-1+size) % size;
```

Single
Program !!!



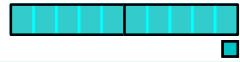
Exercise 1 — Rotating information around a ring

- We start with a more simple version with only one common buffer  **src1**
C [C/Ch4/ring-WRONG1.c](#) or **Fortran** [F_30/Ch4/ring-WRONG1_30.f90](#)
In MPI/tasks/... or **Python** [PY/Ch4/ring-WRONG1.py](#)
 - With 3 processes sum(all ranks) = $0+1+2 = 3$,
with 4 processes sum=6, with 5 processes sum=10, ...
 - Please compile and run the program with 3, 4, 5 processes.
- Now substitute MPI_Send by MPI_Ssend → your **ring-WRONG1-Ss.c** / **_30.f90** / **.py**
 - Expected result: Deadlock
 - Please try to run it and kill it (e.g., with Ctrl+c or Strg+c)
- Do same in version with 2 buffers  **src2**
C [C/Ch4/ring-WRONG2.c](#) or **Fortran** [F_30/Ch4/ring-WRONG2_30.f90](#)
or **Python** [PY/Ch4/ring-WRONG2.py](#)
 - Substitute MPI_Send by MPI_Ssend → your **ring-WRONG2-Ss.c** / **_30.f90** / **.py**
 - Keep the Ssend, but resolve deadlock through a serialization
→ your **ring-WRONG2-serialized.c** / **_30.f90** / **.py**
 - still wrong, not due to deadlock, but due to bad performance**
 - Please compile and run the program with 3, 4, 5 processes:
same results as in the first experiment.**



You may compare your solution with the file in the solution-directory

During the Exercise (10 min.)



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

Important: To solve the exercises please **use previous slides and the provided .c/.f90 files**

(or in rare cases also the MPI standard)

but **no google search on the web** – otherwise you are too slow

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:

Why did we use WRONG2 (and not WRONG1) for the serialized version?

Please **first go** **to your break out room**,
and then test your ring-WRONG2-serialized with only 1 process!

Expected result: original ring-WRONG works,
but “deadlock-free” ring-WRONG2-serialized deadlocks!

Do you all in the group have the same experience?

Why or why not?

Handles, already known

- Predefined handles
 - defined in mpi.h / mpi_f08 / mpi & mpif.h
 - communicator, e.g., MPI_COMM_WORLD
 - datatype, e.g., MPI_INT, MPI_INTEGER, ...

- Handles **can** also be stored in local variables

C	– memory for datatype handles	– in C/C++:	MPI_Datatype
Fortran		– in Fortran: mpi_f08: mpi & mpif.h:	TYPE(MPI_Datatype) INTEGER
Python		– in Python: automatically, e.g., my_dtype = MPI.FLOAT	
C	– memory for communicator handles	– in C/C++:	MPI_Comm
Fortran		– in Fortran: mpi_f08: mpi & mpif.h:	TYPE(MPI_Comm) INTEGER
Python		– in Python: automatically, e.g., comm_world = MPI.COMM_WORLD	

Request Handles

Request handles

- are used for nonblocking communication
- **must** be stored in local variables
 - in C/C++: MPI_Request
 - in Fortran:
mpi_f08: TYPE(MPI_Request)
mpi & mpif.h: INTEGER
 - in Python: automatically

C
Fortran

Python

the value

- **is generated** by a nonblocking communication routine
- **is used** (and freed) in the MPI_WAIT routine

Nonblocking Synchronous Send

c

Fortran

Python

Fortran

- C/C++: `MPI_Issend(&buf, count, datatype, dest, tag, comm,
[OUT] &request_handle);`
`MPI_Wait([INOUT] &request_handle, &status);`
- Fortran: ASYNCHRONOUS :: buf
`CALL MPI_ISSEND(buf, count, datatype, dest, tag, comm,
[OUT] request_handle, ierror)`
`CALL MPI_WAIT([INOUT] request_handle, status, ierror)`
IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING)
& `CALL MPI_F_SYNC_REG(buf)`
New in MPI-3.0
New in MPI-3.0
- Python: `request = comm_world.Issend(...)` / `status = MPI.Status()`; `request.Wait(status)`
- buf must not be modified between Issend and Wait (in all progr. languages)
(In MPI-2.1, this restriction was stronger: “should not access”, see MPI-2.1, page 52, lines 5-6)
- “Issend + Wait directly after Issend” is equivalent to blocking call (Ssend)
- Nothing returned in status (because send operations have no status)
- Fortran problems, see MPI-3.1 / MPI-4.0, Chap. 17.1.10-19 / 19.1.10-19,
pp 631-648 / 817-834, and slides at the end of this course chapter

Nonblocking Receive

c

- C/C++: MPI_Irecv (*buf*, count, datatype, source, tag, comm,
[OUT] &*request_handle*);

↓
MPI_Wait([INOUT] &*request_handle*, &*status*);

- Fortran: ASYNCHRONOUS :: buf
CALL MPI_IRecv (*buf*, count, datatype, source, tag, comm,
[OUT] *request_handle*, *ierror*)

New in MPI-3.0

↓
CALL MPI_WAIT([INOUT] *request_handle*, *status*, *ierror*)
IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING)
& CALL MPI_F_SYNC_REG(*buf*)

New in MPI-3.0

- Python: *request* = comm_world.Issend(...) / *status* = MPI.Status(); *request*.Wait(*status*)

- buf must not be used between Irecv and Wait (in all progr. languages)

- Message status is returned in Wait

- Fortran problems, see MPI-3.1 / MPI-4.0, Chap. 17.1.10-19 / 19.1.10-19,
pp 631-648 / 817-834, and slides at the end of this course chapter

Blocking and Non-Blocking

- Send and receive can be blocking or nonblocking.
- A blocking send can be used with a nonblocking receive, and vice-versa.
- Nonblocking sends can use any mode
 - standard – MPI_ISEND
 - synchronous – MPI_ISSEND
 - buffered – MPI_IBSEND
 - ready – MPI_IRSEND
- Synchronous mode affects completion, i.e. MPI_Wait / MPI_Test, not initiation, i.e., MPI_I....
- The nonblocking operation immediately followed by a matching wait is equivalent to the blocking operation, except the Fortran problems.

Completion

C

- C/C++:

```
MPI_Wait( &request_handle, &status);
```

```
MPI_Test( &request_handle, &flag, &status);
```

Fortran

- Fortran:

```
CALL MPI_WAIT( request_handle, status, ierror)
```

```
CALL MPI_TEST( request_handle, flag, status, ierror)
```

Python

- Python:

```
status = MPI.Status(); request_handle.Wait(status)
```

```
status = MPI.Status(); flag = request_handle.Test(status)
```

- one must
 - WAIT or
 - loop with TEST until request is completed,
i.e., flag == non-zero or .TRUE. or True

C

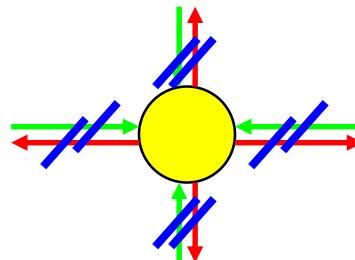
Fortran

Python

Multiple Non-Blocking Communications

You have several request handles:

- Wait or test for completion of **one** message
 - **MPI_Waitany / MPI_Testany**
- Wait or test for completion of **all** messages
 - **MPI_Waitall / MPI_Testall** *)
- Wait or test for completion of **at least one** messages
 - **MPI_Waitsome / MPI_Testsome** *)



*) Each status contains an additional error field.

This field is only used if **MPI_ERR_IN_STATUS** is returned (also valid for send operations).

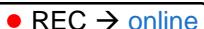
Other MPI features: Send-Receive in one routine

- MPI_Sendrecv & MPI_Sendrecv_replace
 - Combines the triple “MPI_Irecv + Send + Wait” into one routine
 - See advanced exercise at the end of course Chapter 12-(1) *Derived Datatypes*

New in MPI-4.0

- Nonblocking MPI_Isendrecv & MPI_Isendrecv_replace
 - Whereas blocking MPI_Sendrecv was used to prevent
 - **serializations and**
 - **deadlocks,**
 - the nonblocking MPI_Isendrecv can be used, e.g., to parallelize the existing communication calls in multiple directions
→ e.g., to minimize idle times if only some neighbors are delayed

 New in MPI-4.0

© 2000-2021 HLRS, Rolf Rabenseifner  REC → [online](#)

MPI course → Chap.4 Nonblocking Communication

Performance options

Which is the fastest neighbor communication?

- MPI_Irecv + MPI_Send
- MPI_Irecv + MPI_Isend
- MPI_Isend + MPI_Recv
- MPI_Isend + MPI_Irecv
- MPI_Sendrecv
- MPI_Neighbor_alltoall → see course Chapter 9-(2) *Virtual Topologies*

Caution: In the exercise, we use the *synchronous* MPI_Isend() only to demonstrate a deadlock if the nonblocking routine is not correctly used.

A real application would use *standard Isend()* !!!
Never *synchronous* Isend() !!!

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming
but

efficiency of MPI application-programming is **not portable!**

Use cases for nonblocking operations

- To prevent serializations and deadlocks
(as if overlapping of communication with other communication)
New in MPI-4.0 – Now also described in the intro of MPI-4.0 Section 3.7 Nonblocking Communication
- Real overlapping of
 - several communications
 - communication and computation→ see course Chapter 6-(2) *Nonblocking Collectives*
→ Slide: *General progress rule of MPI* 

Nonblocking Receive and Register Optimization / Code Movement in Fortran

Fortran PROBLEM 1:
nonblocking MPI routines

- Fortran source code:

```
CALL MPI_IRecv( buf, ..., request_handle, ierror)
```

```
CALL MPI_Wait( request_handle, status, ierror)
```

```
write (*,*) buf
```

buf is not part of the argument list

- may be compiled as

```
CALL MPI_IRecv( buf, ..., request_handle, ierror)
```

Data may be received in *buf* during
MPI_Wait

registerA = buf

```
CALL MPI_Wait( request_handle, status, ierror)
```

```
write (*,*) registerA
```

Therefore old data may be printed
instead of received data

PROBLEM 1 arises because Fortran compiler is an OPTIMIZING compiler.

Nonblocking Receive and Register Optimization / Code Movement in Fortran

- **Solution:**

With a Fortran 2018 or TS 29113 compiler,

code movements with buf across subroutine calls is prohibited.
Scope should including MPI_Wait and the subsequent use of buf.

<type>, ASYNCHRONOUS :: buf

CALL MPI_IRecv (*buf*, ..., *request_handle*, *ierror*)
CALL MPI_WAIT(*request_handle*, *status*, *ierror*)

buf is not part of the argument list, i.e., code movements with buf across this call is allowed for non-TS 29113 compiler

IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) &
& CALL MPI_F_SYNC_REG(*buf*)

write (*,*) buf

Directly after CALL MPI_Wait.
Needed for non-TS 29113 compiler, if buf is not defined in module data or a common block

With a TS 29113 compiler, the MPI library can set
MPI_ASYNC_PROTECTS_NONBLOCKING == .TRUE.
and at compile time, the compiler will remove the "CALL MPI_F_SYNC_REG"

- **Work-around in older MPI versions:**

CALL MPI_GET_ADDRESS(*buf*, *iaddrdummy*, *ierror*)
with INTEGER(KIND=MPI_ADDRESS_KIND) *iaddrdummy*

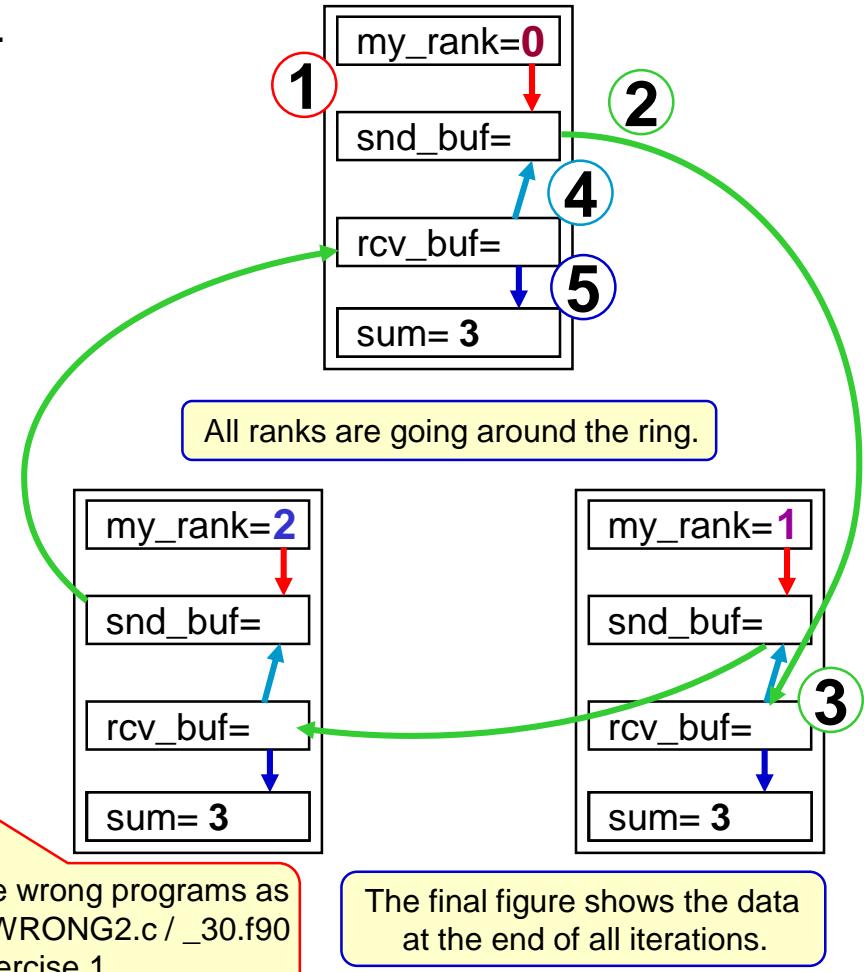
i.e., if MPI_F_SYNC_REG is not yet available

Exercise 2 — Rotating information around a ring

- A set of processes are arranged in a ring.
- 1** Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- 2** + **3** Each process passes this on to its neighbor on the right.
- 4** Preparation of next iteration.
- 5** Each process calculates the sum of all values.
- Repeat “**2**-**5**” with “size” iterations (size = number of processes), i.e.,
- each process calculates sum of all ranks.

C Use [C/Ch4/ring-skel.c](#)
Fortran or [F_30/Ch4/ring-skel_30.f90](#)
Python or [PY/Ch4/ring-skel.py](#)

- Use nonblocking MPI_Isend !**
 - to avoid deadlocks**
 - to verify the correctness,** because blocking synchronous send will cause a deadlock
- Keep normal blocking MPI_Recv !**

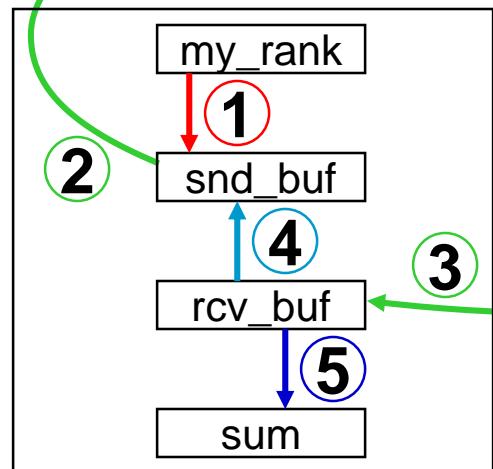
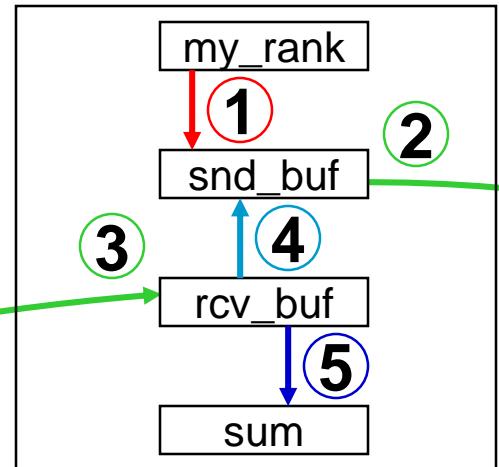


Exercise 2 — Rotating information around a ring

Initialization: 1

Each iteration:

2 3 4 5



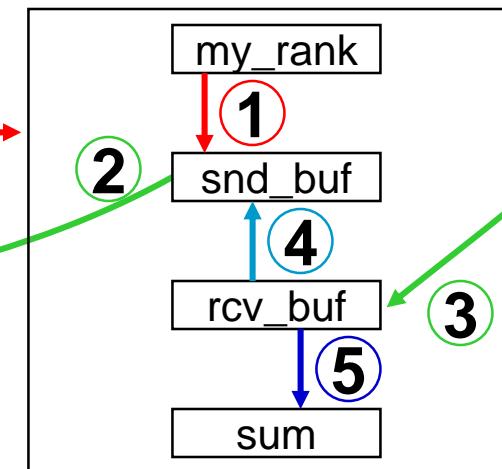
Fortran:

```
dest = mod(my_rank+1,size)  
source = mod(my_rank-1+size,size)
```

C/C++:

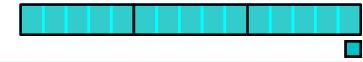
```
dest = (my_rank+1) % size;  
source = (my_rank-1+size) % size;
```

Single
Program !!!



Fortran: Do not forget MPI-3.0 → ..., ASYNCHRONOUS :: ..._buf and IF(.NOT.MPI...) CALL MPI_F_SYNC_REG(...)

During the Exercise (15 min.)



Please stay here in the main room while you do this 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:

⌚ Does this solution also work when you start it with only 1 MPI process?



Do you and your colleagues in your breakout room
have already some experience with MPI?

And already with preventing deadlocks?

Which methods have you used in the past?

Have you and your colleagues ever thought about serialization?



Exercises 3 (advanced) — `Irecv` instead of `Issend`

- Substitute the `Issend–Recv–Wait` method by the `Irecv–Ssend–Wait` method in your ring program.
 - Solution: MPI/tasks/C/Ch4/solutions/ring_advanced_irecv_ssend.c
and MPI/tasks/F_30/Ch4/solutions/ring_advanced_irecv_ssend_30.f90
- Or
- Substitute the `Issend–Recv–Wait` method by the `Irecv–Issend–Waitall` method in your ring program.
 - Solution: MPI/tasks/C/Ch4/solutions/ring_advanced_irecv_issend.c
and MPI/tasks/F_30/Ch4/solutions/ring_advanced_irecv_issend_30.f90

Quiz A+B on Chapter 4 – Nonblocking communication

- A. MPI nonblocking communication: Which are the three major use-cases and please sort from most important to less important:

1. _____
2. _____
3. _____

- B. [MPI/participant-test/ring-test.c](#) or [MPI/participant-test/ring-test_30.f90](#)

Is this **loop body** correctly programmed with MPI

- Yes / No

For the case that you answered with "No",

- which bug(s) do you see?
- And how could it/they be resolved?
- Which correction(s) would you apply
(still using nonblocking communication)?

Fortran

Full program on next slide

C

```
MPI_Isend(&buffer,1,MPI_INT,right,  
          17, MPI_COMM_WORLD, &request);  
MPI_Recv (&buffer,1,MPI_INT, left,  
          17, MPI_COMM_WORLD, &status);  
sum += buffer;
```

Python

```
CALL MPI_Isend(buffer,1,MPI_INTEGER,&  
   & right,17,MPI_COMM_WORLD,request)  
CALL MPI_Recv (buffer,1,MPI_INTEGER,&  
   & left, 17,MPI_COMM_WORLD,status)  
sum = sum + buffer  
  
| request=comm_world.Issend((buffer,1,MPI.INT),  
|                               dest=right, tag=17)  
| comm_world.Recv((buffer,1,MPI.INT),  
|                               source=left, tag=17, status=status)  
| sum += buffer
```

Quiz A+B on Chapter 4 – ring-test.c / _30.f90

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
    int my_rank, size;
    int buffer;
    int right, left;
    int sum, i;
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    right = (my_rank+1) % size;
    left = (my_rank-1+size) % size;

    sum = 0;
    buffer = my_rank;
    for( i = 0; i < size; i++)
    {
        MPI_Isend(&buffer,1,MPI_INT,right,
                  17, MPI_COMM_WORLD, &request);
        MPI_Recv (&buffer,1,MPI_INT, left,
                  17, MPI_COMM_WORLD, &status);
        sum += buffer;
    }
    printf ("PE%i:\tSum = %i\n", my_rank, sum);

    MPI_Finalize();
}
```

C

PROGRAM ring
USE mpi_f08
IMPLICIT NONE

INTEGER :: my_rank, size
INTEGER :: buffer
INTEGER :: right, left
INTEGER :: sum, i
TYPE(MPI_Status) :: status
TYPE(MPI_Request) :: request

CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
CALL MPI_Comm_size(MPI_COMM_WORLD, size)
right = mod(my_rank+1, size)
left = mod(my_rank-1+size, size)

sum = 0
buffer = my_rank

DO i = 1, size

CALL MPI_Isend(buffer,1,MPI_INTEGER,&
 & right,17,MPI_COMM_WORLD,request)
 CALL MPI_Recv (buffer,1,MPI_INTEGER,&
 & left, 17,MPI_COMM_WORLD,status)
 sum = sum + buffer

END DO
WRITE(*,*) "PE", my_rank, ": Sum =", sum

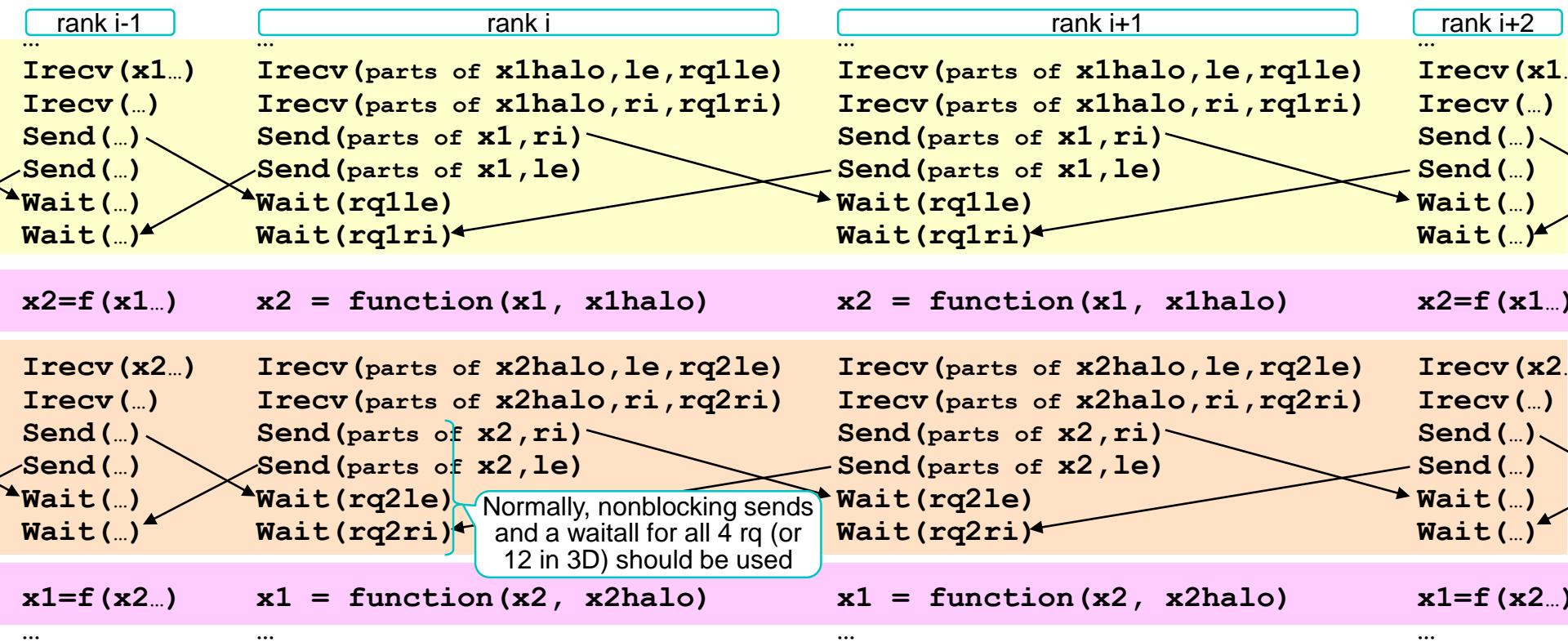
CALL MPI_Finalize()

END PROGRAM

Fortran

Quiz C on Chapter 4 – Double buffering and MPI_Rsend

- C. Let's have a look at several iteration of our halo exchange and $x_{\text{new}} = \text{function}(x_{\text{old}})$:
- We use here x_1 and x_2 instead of x_{new} and x_{old} and show two iterations
 - The halos are explicitly named with $x_1\text{halo}$ and $x_2\text{halo}$, and $\text{le}=\text{left}$, $\text{ri}=\text{right}$



Who can see, what must we change that we can use **MPI_Rsend** instead of **MPI_Send**?

For private notes

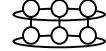
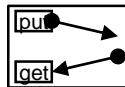
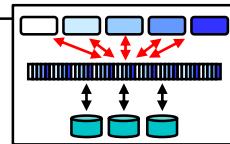
For private notes

For private notes

Chap.5 The New Fortran Module mpi_f08

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication

5. The New Fortran Module mpi_f08

6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication
12. Derived datatypes 
13. Parallel file I/O 
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

With additional notes,
courtesy to Claudia Blaas-Schenner

MPI and Fortran — Consistency problems

3 Fortran support methods:

1. mpif.h

- MPI-1 was inconsistent with Fortran 90
 - Several routines were deprecated and substituted
→ MPI-3.1, page 18, Table 2.1 or [MPI-4.0, page 25, Table 2.1](#)
 - Reason: they used INTEGER instead of INTEGER(KIND=MPI_ADDRESS_KIND)
 - Consequences
 - **For C programs:**
 - All was already MPI_Aint → no changes within declarations
 - Substitute deprecated/removed routine name by new one
 - **For Fortran programs:**
 - Change actual arguments into INTEGER(KIND=MPI_ADDRESS_KIND)
 - Substitute deprecated/removed routine name by new one

MPI and Fortran — Consistency problems

3 Fortran support methods:

2. mpi module

- MPI-2 was aware about Fortran problems
 - E.g., with nonblocking routines
 - And proposed a user-written dummy-routine dd()

```
CALL MPI_IRecv( buf, ..., request_handle... )
CALL MPI_Wait( request_handle, ... )
CALL dd(buf)
  or CALL MPI_Get_Address(buf, ...)
write (*,*) buf
```

This solution is only a work-around.

Fortran PROBLEM 1: nonblocking MPI routines

→ Course Chapter 4

Nonblocking Communication

```
CALL MPI_IRecv( buf, ..., request_handle, ierror )
CALL MPI_Wait( request_handle, status, ierror )
write (*,*) buf
```

may be compiled as

```
CALL MPI_IRecv( buf
registerA = buf
CALL MPI_Wait( request_handle, status, ierror )
write (*,*) registerA
```

buf is not part of
the argument list

Data may be received
in buf during MPI_Wait

Therefore old data
may be printed instead
of received data

MPI and Fortran — Consistency problems

3 Fortran support methods:

3. mpi_f08 module

- For MPI-3 and later, the new mpi_f08 module was designed to solve several inconsistencies with Fortran:
 - Problems with nonblocking routines, because Fortran enables special OPTIMIZATIONS → Course Chapter 4 
 - Optional ierror → Course Chapter 2 
 - Other problems → following slides

Routine declarations within the mpi_f08 module

To solve the strided-array problem

`MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)`

- `TYPE(*), DIMENSION(..), ASYNCHRONOUS1) :: buf`
- `INTEGER, INTENT(IN) :: count, source, tag`
- `TYPE(MPI_Datatype), INTENT(IN) :: datatype`
- `TYPE(MPI_Comm), INTENT(IN) :: comm`
- `TYPE(MPI_Request), INTENT(OUT) :: request`
- `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

Fortran compatible buffer declaration allows correct compiler optimizations

Unique handle types allow best compile-time argument checking

INTENT → Compiler-based optimizations & checking

`MPI_Wait(request, status, ierror) BIND(C)`

- `TYPE(MPI_Request), INTENT(INOUT) :: request`
- `TYPE(MPI_Status) :: status`
- `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

Status is now a Fortran structure, i.e., a Fortran derived type

¹⁾ ASYNCHRONOUS: only in nonblocking routines, not in MPI_Recv

OPTIONAL ierror:
MPI routine can be called without ierror argument

Keyword-based argument lists in mpi_f08 and mpi module

Positional and **keyword-based** argument lists

- CALL MPI_SEND(sndbuf, 5, MPI_REAL, right, 33, MPI_COMM_WORLD)
- CALL MPI_SEND(**buf**=sndbuf, **count**=5, **datatype**=MPI_REAL,
dest=right, **tag**=33, **comm**=MPI_COMM_WORLD)

The keywords are defined in the language bindings.
Same keywords for both modules.

Remarks:

- Some keywords are changed since MPI-2.2
 - For consistency reasons, or
 - To prohibit conflicts with Fortran keywords, e.g., *type*, *function*.
- Some MPI libraries show version numbers 3.0 and higher although
 - they do not correctly implement keyword based argument lists although the compiler would provide enough support for it
 - See next Exercise 1 in this course Chap. 5: ~/MPI/tasks/F_30/Ch5/version_*
 - Or advanced Exercise 4 of course Chap. 2: ~/MPI/tasks/F_30/Ch2/version_*

Do not use
outdated documents!

MPI_Status within the mpi_f08 module

Support method:

USE mpi or INCLUDE 'mpif.h' → **USE mpi_f08**

Status

INTEGER :: status(MPI_STATUS_SIZE) → **TYPE(MPI_Status) :: status**

status(MPI_SOURCE) → **status%MPI_SOURCE**

status(MPI_TAG) → **status%MPI_TAG**

status(MPI_ERROR) → **status%MPI_ERROR**

Additional routines and declarations are provided for the language interoperability of the status information between

- C,
- **Fortran mpi_f08, and**
- **Fortran mpi (and mpif.h)**

see MPI-3.1, Section 17.2.5 pages 656-658 or
MPI-4.0, Section 19.3.5 pages 843-846

Handles in mpi_f08

- Unique handle types, e.g.,
 - INTEGER comm
- Handle comparisons, e.g.,
 - comm .EQ. MPI_COMM_NULL
- Conversion in mixed applications:
 - Both modules (mpi & mpi_f08) contain the declarations for all handles.

Same names as in C

```
TYPE, BIND(C) :: MPI_Comm
  INTEGER :: MPI_VAL
END TYPE MPI_Comm
```

→ **TYPE(MPI_Comm) :: comm**

→ comm .EQ. MPI_COMM_NULL

No change through overloaded operator

```
SUBROUTINE a
USE mpi
INTEGER :: splitcomm
CALL MPI_COMM_SPLIT(..., splitcomm)
CALL b(splitcomm)
END
SUBROUTINE b(splitcomm)
USE mpi_f08
INTEGER :: splitcomm
TYPE(MPI_Comm) :: splitcomm_f08
CALL MPI_Send(..., MPI_Comm(splitcomm) )
! or
splitcomm_f08%MPI_VAL = splitcomm
CALL MPI_Send(..., splitcomm_f08)
END
```

```
SUBROUTINE a
USE mpi_f08
TYPE(MPI_Comm) :: splitcomm
CALL MPI_Comm_split(..., splitcomm)
CALL b(splitcomm)
END
SUBROUTINE b(splitcomm)
USE mpi
TYPE(MPI_Comm) :: splitcomm
INTEGER :: splitcomm_old
CALL MPI_SEND(..., splitcomm%MPI_VAL )
! or
splitcomm_old = splitcomm%MPI_VAL
CALL MPI_SEND(..., splitcomm_old)
END
```

Exercise 1 — Check the quality of your MPI library

Exercise 1

In MPI/tasks/...

- Please compile and run with 1 process
 - [F_30/Ch5/version_test_11.f](#) → checks for Fortran support through mpif.h
 - [F_30/Ch5/version_test_20.f90](#) → checks for the mpi module
 - [F_30/Ch5/version_test_30.f90](#) → checks for the mpi_f08 module
- Both mpi and mpi_f08 are required to provide explicit interface specifications and therefore to support
 - compile time argument checking, and
 - key word based argument lists

To check this, please compile and run with 1 process

- [F_30/Ch5/version_test_keyarg_20.f90](#) → it is required to compile (mpi mod.)
- [F_30/Ch5/version_test_keyarg_30.f90](#) → it is required to compile (mpi_f08),

otherwise your MPI library is not standard compliant:

- a typical problem of the mpi module in mpich based MPI libraries;
- the mpi_f08 module should work correctly in both mpich and OpenMPI,
but sometimes not automatically installed with mpich.

Caution: OpenMPI is an MPI library, whereas OpenMP is a shared memory parallel programming model

Exercise 1 — Check the quality of your MPI library

- Both mpi and mpi_f08 are required to provide the new types MPI_Status, MPI_Comm, ... and overloaded operators ==, /=, .EQ., and .NE. for them. The question is, whether they are really available in the mpi module:
To check this, please compile and run with 1 process

In MPI/tasks/...

- F_30/Ch5/version_test_handles_20.f90 → it is required to run with mpi,
- F_30/Ch5/version_test_handles_30.f90 → it is required to run with mpi_f08,
and also
 - F_30/Ch5/version_test_status+handles_20.f90
 - F_30/Ch5/version_test_status+handles_30.f90

otherwise your MPI library is not standard compliant:

Currently,

- OpenMPI may still have a problem with the mpi module
 - mpich may still not provide the mpi_f08 by default
 - and both may have a problem with the status conversion.
- Please check!**

During the Exercise (5 min.)



Please stay here in the main room while you do this 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



Please stay in the main room!

When finished, please write a short “finished” in the chat.



Any quality problems with your MPI installation?

- Could you compile `version_test_keyarg_20.f90` ?
- Could you compile `version_test_keyarg_30.f90` ?
- Could you compile and run `version_test_handles_20.f90` ?
- Could you compile and run `version_test_handles_30.f90` ?
- Could you compile and run `version_test_status+handles_20.f90` ?
- Could you compile and run `version_test_status+handles_30.f90` ?



Data with longer steps between the portions,
i.e., non-contiguous data in memory

Nonblocking MPI routines and strided sub-arrays in Fortran

- Fortran:

```
CALL MPI_ISEND ( buf(7,:,:), ..., request_handle, ierror)
```

**Fortran PROBLEM 2:
strided sub-arrays**

- The content of this non-contiguous sub-array is stored in a temporary array.
- Then MPI_ISEND is called.
- On return, the temporary array is **released**.

other work

- The data may be transferred while other work is done, ...
- ... or inside of MPI_Wait, but the
data in the temporary array is already lost!

```
CALL MPI_WAIT( request_handle, status, ierror)
```

- Since MPI-3.0: Works if **MPI_SUBARRAYS_SUPPORTED == .TRUE.**
- MPI-1.0 – MPI-2.2 and MPI-3/-4 without TS 29113:
Do not use non-contiguous sub-arrays in nonblocking calls!!!
 - Use first sub-array element (`buf(1,1,9)`) instead of contiguous sub-array (`buf(:,:,9:13)`)
 - Call by reference necessary → Call by in-and-out-copy forbidden
 - → use the correct compiler flags!

(requires Fortran 200x + TS29113
or Fortran 2018 compiler)

but it arises in Fortran in a more subtle way

This problem is NOT specific to Fortran!

obviously wrong code:

(same in **C** with malloc & free
or auto-deallocated data on the stack)

```
allocate(buf(n))
CALL MPI_ISEND(buf,n,...)
deallocate(buf)
```

It is only safe to deallocate the memory
after the matching Wait.

**Do NOT use non-contiguous
subarrays in nonblocking calls!!!**

(work only with: TS 29113 compiler & mpi_f08)

contiguous array sections: pass the starting element (array(1,1)) instead of (array(1:m,1))
non-contiguous sections: do an explicit copy to a contiguous temporary buffer (kept after Wait)
or define an appropriate vector derived data type
→ Course Chapter 12 *Derived Datatypes*

same problem (but not so obvious):

```
real, dimension(m,n) :: arr
...
CALL MPI_ISEND(arr(1,1:n),n,...)
```

since it will be compiled

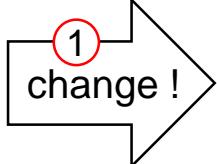
(without TS 29113 compiler & mpi_f08) **as:**

```
allocate( scratch_buf(n) )
scratch_buf(1:n) = array(1,1:n)
CALL MPI_ISEND(scratch_buf,n,...)
array(1,1:n) = scratch_buf(1:n)
deallocate(scratch_buf)
```

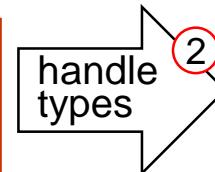
Fortran and MPI – Best Practice

Switch to the new **mpi_f08** module to be consistent with Fortran standard

```
program xxxxx  
implicit none  
include 'mpif.h'
```



```
program xxxxx  
use mpi  
implicit none
```



```
program xxxxx  
use mpi_f08  
implicit none
```

- ① Compile with a library that provides compile-time argument checking
 - Can be tested with
 - Try to compile `~/MPI/tasks/F_30/Ch5/version_test_keyarg_20.f90`
 - **Compiles only if the mpi module provides correct bindings according to MPI-3.0 and higher, i.e., allowing also keyword based argument lists!**
 - E.g., OpenMPI provides compile-time argument checking for the mpi module
- ② INTEGER rq, comm, datatype, status(MPI_STATUS_SIZE)
→ TYPE(MPI_Request) :: rq
TYPE(MPI_Comm) :: comm
TYPE(MPI_Datatype) :: datatype
TYPE(MPI_Status) :: status

Caution: OpenMPI is an MPI library, whereas OpenMP is a shared memory parallel programming model

Full consistency requires Fortran 2003/2008 + TS 29113 or Fortran 2018

Fortran and MPI – Best Practice

- **in your code** always do (as long as **TS 29113** is not everywhere available):

Non-contiguous subarrays: Do NOT use in nonblocking routines ! (workaround: see previous two slides)

Buffers in nonblocking routines or together with MPI_BOTTOM or in 1-sided communication:

```
<type>, ASYNCHRONOUS :: buffer
```

```
IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(buffer)  
after MPI_Wait or before and after blocking calls with MPI_BOTTOM
```

or before a nonblocking routine with MPI_BOTTOM and after final MPI_Wait / ...
or in 1-sided communication before the 1st and after 2nd CALL MPI_Win_fence

in older MPI versions or if the MPI-3.0 Fortran support methods are incomplete:

```
INTEGER(KIND=MPI_ADDRESS_KIND) :: iadummy  
CALL MPI_GET_ADDRESS(buffer, iadummy, ierror)
```

Major enhancement with a full MPI-3.0 implementation

- The following features require Fortran 2003/2008 + **TS 29113** or **Fortran 2018**
 - Subarrays may be passed to nonblocking routines, e.g., array(0:12:3)
 - This feature is available if the **LOGICAL** compile-time constant **MPI_SUBARRAYS_SUPPORTED == .TRUE.**
 - Correct handling of buffers passed to nonblocking routines,
 - if the application has declared the buffer as **ASYNCHRONOUS** within the scope from which the nonblocking MPI routine and its MPI_Wait/Test is called,
 - and the **LOGICAL** compile-time constant **MPI_ASYNC_PROTECTS_NONBLOCKING == .TRUE.**
 - mpi_f08 module:
 - These features must be available in MPI-3.0 if the target compiler is **TS 29113** compliant.
 - mpi module and mpif.h:
 - These features are a question of the quality of the MPI library.
 - If feature is not available in a Fortran support method:
 - Constant is set to **.FALSE.**
- **Conclusions:**
 - Non-contiguous subarrays: Don't use in nonblocking routines until **TS 29113** compilers are available
 - Buffers in nonblocking routines or together with **MPI_BOTTOM**:
 - Declare buffer as **ASYNCHRONOUS**
 - IF (.NOT. **MPI_ASYNC_PROTECTS_NONBLOCKING**) CALL **MPI_F_SYNC_REG(buffer)** after **MPI_Wait** or before and after blocking calls with **MPI_BOTTOM**

Detailed description of problems, mainly with the old support methods, or if the compiler does not support TS 29113:

MPI 4.0 Sections

- 19.1.8 Additional Support for Fortran Register-Memory-Synchronization
- 19.1.10 Problems With Fortran Bindings for MPI
- 19.1.11 Problems Due to Strong Typing
- 19.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets
- 19.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts
- 19.1.14 Special Constants
- 19.1.15 Fortran Derived Types
- 19.1.16 Optimization Problems, an Overview
- 19.1.17 Problems with Code Movement and Register Optimization
 - **Nonblocking Operations**
 - **One-sided Communication**
 - **MPI_BOTTOM and Combining Independent Variables in Datatypes**
 - **Solutions**
 - **The Fortran ASYNCHRONOUS Attribute**
 - **Calling MPI_F_SYNC_REG (new routine, defined in Section 19.1.7)**
 - **A User Defined Routine Instead of MPI_F_SYNC_REG**
 - **Module Variables and COMMON Blocks**
 - **The (Poorly Performing) Fortran VOLATILE Attribute**
 - **The Fortran TARGET Attribute**
- 19.1.18 Temporary Data Movement and Temporary Memory Modication
- 19.1.19 Permanent Data Movement
- 19.1.20 Comparison with C

New in MPI-3.0

For private notes

For private notes

For private notes

Chap.6 Collective Communication

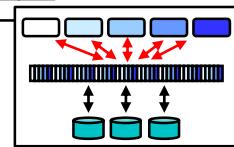
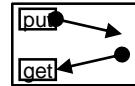
1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module mpi_f08



`MPI_Init()`
`MPI_Comm_rank()`

6. Collective communication

- (1) e.g., broadcast
 - (2) advanced topics, e.g., nonblocking collectives, neighborhood comm.
7. Error Handling
 8. Groups & communicators, environment management
 9. Virtual topologies
 10. One-sided communication
 11. Shared memory one-sided communication
 12. Derived datatypes
 13. Parallel file I/O
 14. MPI and threads
 15. Probe, Persistent Requests, Cancel
 16. Process creation and management
 17. Other MPI features
 18. Best Practice



Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
 - Barrier synchronization.
 - Broadcast, scatter, gather.
 - Global sum, global maximum, etc.
 - Neighbor communication in a virtual process grid

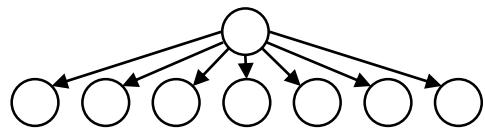
Should be faster than
any programming
with point-to-point
messages!

New in MPI-3.0

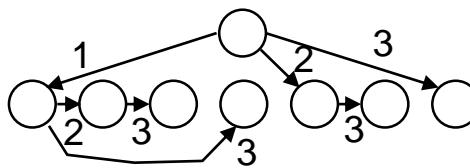
Short tour = 4 slides 
tour

Internally: tree-based algorithms

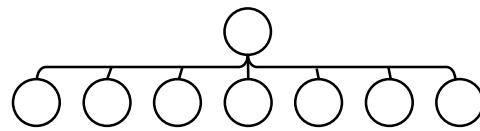
E.g., broadcast



Sequential algorithm
 $O(\# \text{ processes})$



Tree based algorithm
 $O(\log_2(\# \text{ processes}))$



Hardware-broadcast
 $O(1)$

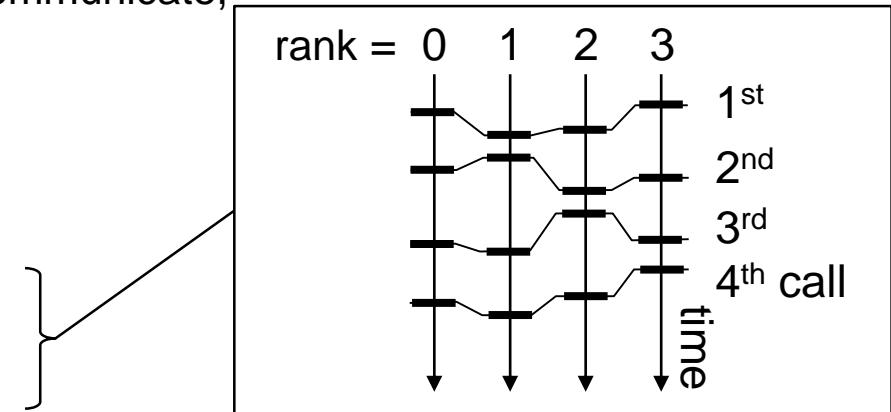
You need not to care about !
It is the job of the MPI lib !!!

And optimized algorithms on **clusters of SMP nodes** are even more complicated!

Characteristics of Collective Communication

- Collective action over a communicator.
- All processes of the communicator must communicate, i.e., must call the collective routine.
- Synchronization may or may not occur, therefore all processes must be able to start the collective routine.
- On a given communicator, the n-th collective call must match on all processes of the communicator.
- In MPI-1.0 – MPI-2.2, all collective operations are blocking. Nonblocking versions since MPI-3.0.
- No tags.
- For each message, the amount of data sent must exactly match the amount of data specified by the receiver
 - It is forbidden to provide receive buffer count arguments that are too long (and also too short, of course)

very important



Exception with Python (mpi4py): if a buffer argument represents #processes of messages (e.g. `snd_buf` in `comm.Scatter`) and the argument `count` is to be derived from the buffer argument (i.e. is not explicitly defined in the argument list), then this `count` argument is derived from the inferred number of elements of the buffer divided by the size of the communicator.

e.g., when passing `snd_buf`, or `(snd_buf, datatype)`.

For buffer options such as `BufSpec`, `BufSpecV`, ..., see e.g. [mpi4py/typing.pyi at master · mpi4py/mpi4py \(github.com\)](https://github.com/mpi4py/mpi4py/blob/master/mpi4py/typing.pyi)

Barrier Synchronization

C

Fortran

Python

- C/C++: `int MPI_Barrier(MPI_Comm comm)`
- Fortran: `MPI_BARRIER(comm, ierror)`
`mpi_f08: TYPE(MPI_Comm) :: comm ; INTEGER, OPTIONAL :: ierror`
`mpi & mpif.h: INTEGER comm, ierror`
- Python: `comm.Barrier()` or `comm.barrier()`

- `MPI_Barrier` is normally never needed:
 - all synchronization is done automatically by the data communication:
 - a process cannot continue before it has the data that it needs.
 - if used for debugging:
 - please guarantee, that it is removed in production.
 - for profiling: to separate time measurement of
 - Load imbalance of computation [`MPI_Wtime(); MPI_Barrier(); MPI_Wtime()`]
 - communication epochs [`MPI_Wtime(); MPI_Allreduce(); ...; MPI_Wtime()`]
 - ~~if used for synchronizing external communication (e.g. I/O):~~
 - ~~exchanging tokens may be more efficient and scalable than a barrier on `MPI_COMM_WORLD`,~~
 - ~~see also advanced exercise of this course chapter.~~

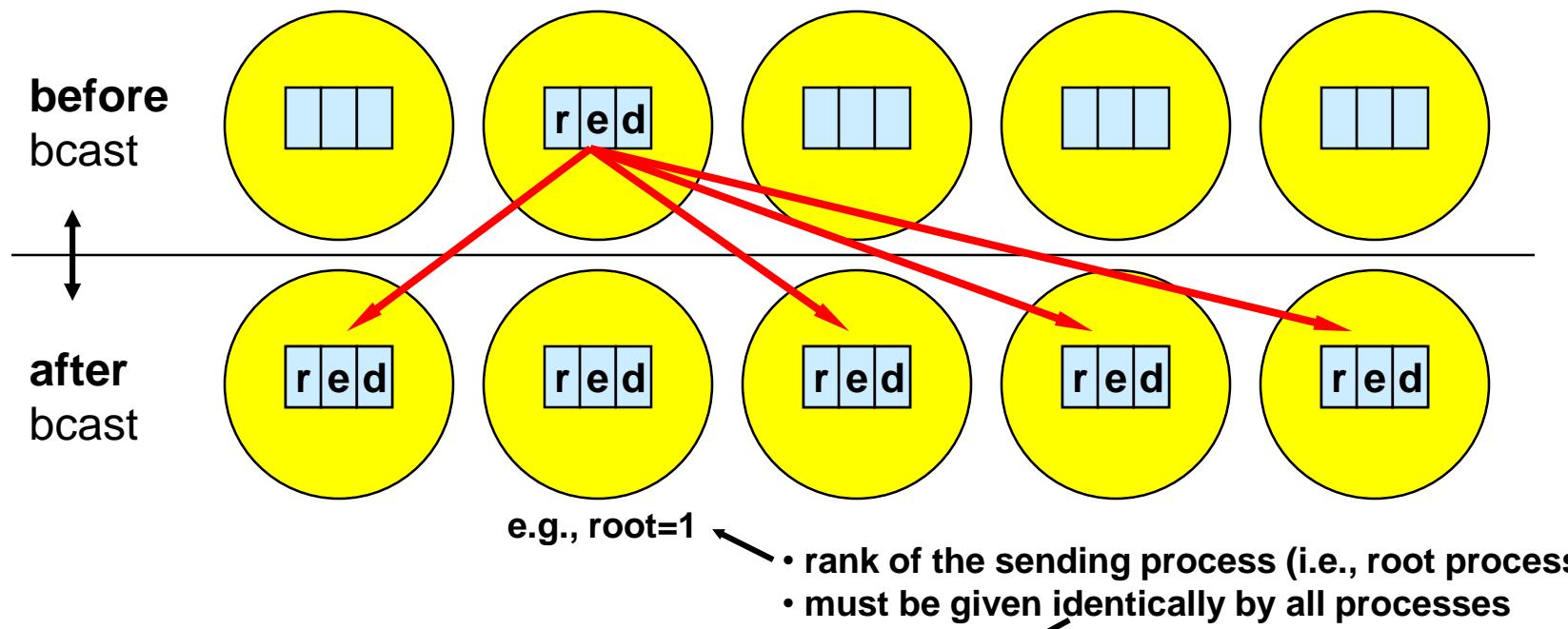
Broadcast

C

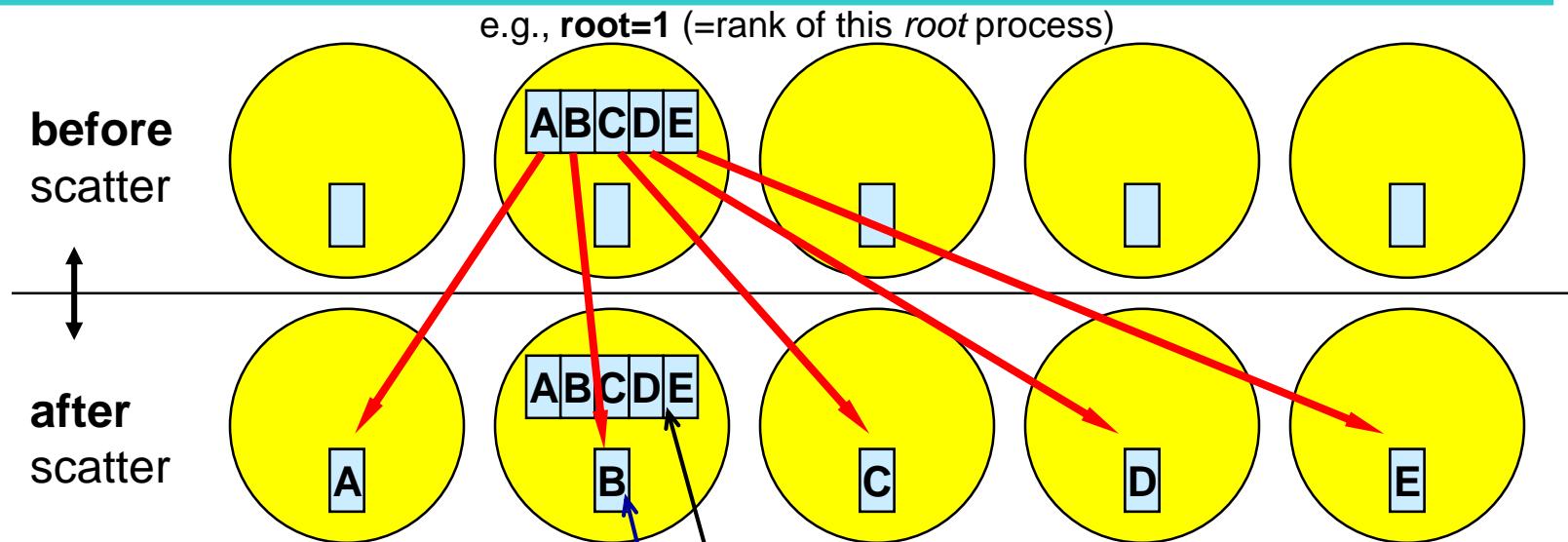
Fortran

Python

- C/C++: `int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
- Fortran:
 - mpi_f08:
`TYPE(*), DIMENSION(..) :: buf`
`TYPE(MPI_Datatype) :: datatype;`
`INTEGER :: count, root;`
`<type> buf(*); INTEGER count, datatype, root, comm, ierror`
 - mpi & mpif.h:
`TYPE(MPI_Comm) :: comm`
`INTEGER, OPTIONAL :: ierror`
- Python: `comm.Bcast(buf, int root=0)` or `comm.bcast(obj, int root=0)`



Scatter



C

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
                void *recvbuf, int recvcount, MPI_Datatype recvtype,  
                int root, MPI_Comm comm)
```

Fortran

```
MPI_SCATTER(sendbuf, sendcount, sendtype,  
            recvbuf, recvcount, recvtype, root, comm, ierror)
```

```
mpi_f08:   TYPE(*), DIMENSION(..) :: sendbuf, recvbuf;   INTEGER :: sendcount, recvcount, root;  
            TYPE(MPI_Datatype) :: sendtype, recvtype;   TYPE(MPI_Comm) :: comm;   INTEGER, OPTIONAL :: ierror
```

```
mpi & mpif.h: <type> sendbuf(*), recvbuf(*); INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror
```

Python

```
comm.Scatter(sendbuf or None, recvbuf, int root=0)  
recvobj = comm.scatter(sendobj or None, int root=0)
```

See, e.g., [Tutorial — MPI for Python 3.1.1 documentation \(mpi4py.readthedocs.io\)](#)

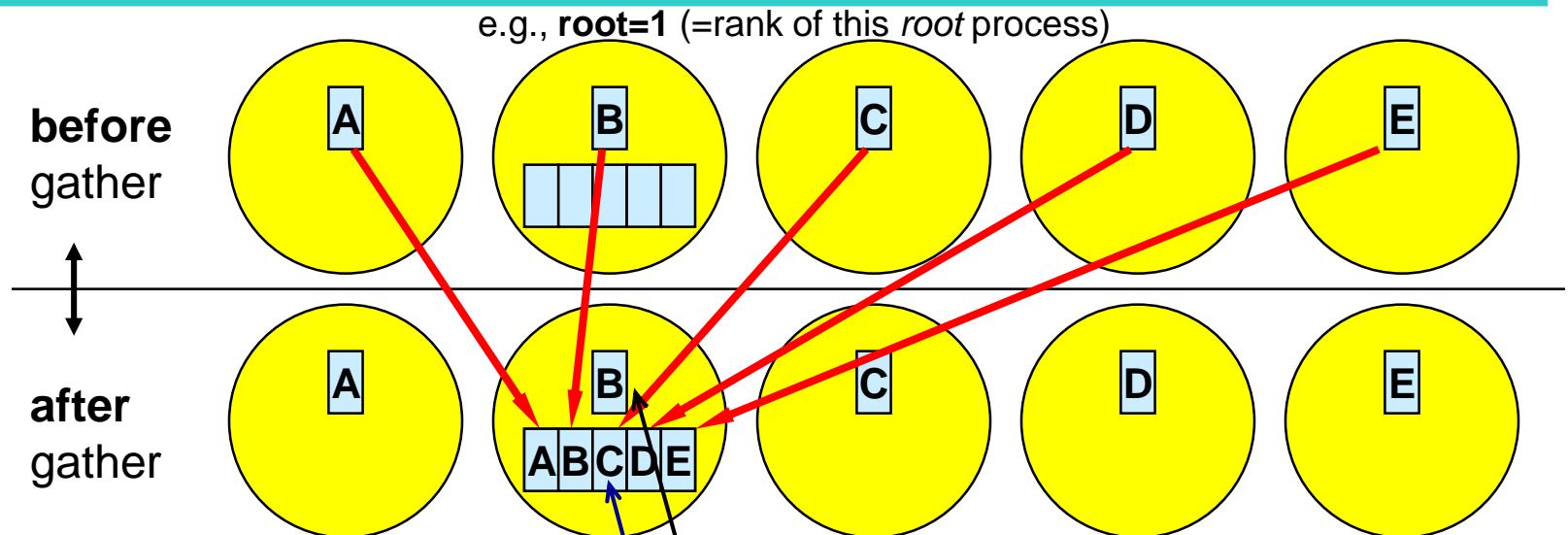
sendcount describes only one message

Example: `MPI_Scatter(sbuf, 1, MPI_CHAR, rbuf, 1, MPI_CHAR, 1, MPI_COMM_WORLD);`

Completely ignored at all
processes except *root*



Gather



C

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
               void *recvbuf, int recvcount, MPI_Datatype recvtype,  
               int root, MPI_Comm comm)
```

Fortran

```
MPI_GATHER(sendbuf, sendcount, sendtype,  
            recvbuf, recvcount, recvtype, root, comm, ierror)
```

mpi_f08: `TYPE(*), DIMENSION(..) :: sendbuf, recvbuf;` `INTEGER :: sendcount, recvcount, root;`
 `TYPE(MPI_Datatype) :: sendtype, recvtype;` `TYPE(MPI_Comm) :: comm; INTEGER, OPTIONAL :: ierror`

mpi & mpif.h: `<type> sendbuf(*), recvbuf(*);` `INTEGER sendcount, sendtype, recvcount, recvtype, root, comm, ierror`

Python

```
comm.Gather(sendbuf, recvbuf or None, int root=0)  
recvobj = comm.gather(sendobj, int root=0)
```

See, e.g., [Tutorial — MPI for Python 3.1.1 documentation \(mpi4py.readthedocs.io\)](#)

CALL MPI_Gather(sbuf, 1, MPI_CHARACTER, *rbuf*, 1, MPI_CHARACTER, 1, MPI_COMM_WORLD, ierror);

recvcount describes only one message

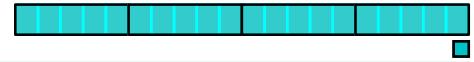
Completely ignored at all
processes except *root*

Exercise 1 — Gather

In MPI/tasks/...

- Use **C** C/Ch6/gather-skel.c or **Fortran** F_30/Ch6/gather-skel_30.f90 or **Python** PY/Ch6/gather-skel.py
- The skeleton is based on our first example in course Chapter 1.
- Differences:
 - This skeleton first gathers the data into an array at process 0
 - And then, process 0 prints the array.
- In this exercise, you should substitute the point-to-point communication by one call to MPI_Gather

During the Exercise (20 min.)



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



And have fun with this a bit longer 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 enjoy the (short) break



Advanced Exercise 1b — Barrier / profiling

- Based on C/Ch6/solutions/pi.c → pi-mpi.c → pi-mpi-inbalance.c
 - balanced
 - inbalanced
- Use C C/Ch6/pi-mpi-inbalance-profiling-skel.c
or Python PY/Ch6/pi-mpi-inbalance-profiling-skel.py
or Fortran (my apologies, Fortran does not yet exists, but this shouldn't be a problem)
- This program has several parts:
 - Perfect work-distribution for n=10,000,000 intervals.
 - If 3 or more processes:
Introducing an imbalance: The last 2 processes get double and zero intervals.
 - Calculation of π with a distributed integral → partial sums in p_sum.
 - Global reduction of all p_sum into one global sum.
 - Time measurements for all parts
- Your task, see “// EXERCISE” in the skeleton:
 - Add MPI_Barrier wherever useful, and especially to measure idle time due to the bad load balance.
 - Substitute all wt? by wt1 .. wt4 as needed
 - Compile and run it with 2 processes
 - expected result 99,9% parallel efficiency
 - Run with more than 3 processes
 - about 50% parallel efficiency and 50% in idle time



Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- $d_0 \circ d_1 \circ d_2 \circ d_3 \circ \dots \circ d_{s-2} \circ d_{s-1}$
 - d_i = data in process rank i
 - single variable, or
 - vector
 - \circ = associative operation
 - Example:
 - global sum or product
 - global maximum or minimum
 - global user-defined operation
- floating point rounding may depend on usage of associative law:
 - $[(d_0 \circ d_1) \circ (d_2 \circ d_3)] \circ [\dots \circ (d_{s-2} \circ d_{s-1})]$
 - $(((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \dots) \circ d_{s-2}) \circ d_{s-1}$
 - May be even worse through partial sums in each process:
$$\sum_{i=0}^{n-1} x_i \rightarrow [[[(\sum_{i=0}^{n/s-1} x_i \circ \sum_{i=n/s}^{2n/s-1} x_i) \circ (\dots \circ \dots)] \circ [\dots \circ (\dots \circ \dots)]]]$$

E.g., with $n=10^8$ rounding errors may modify last 3 or 4 digits!

Example of Global Reduction

- Global integer sum.
- Sum of all inbuf values should be returned in *resultbuf*.

C

- C/C++: root=0;

```
MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT,  
          MPI_SUM, root, MPI_COMM_WORLD);
```

Fortran

- Fortran: root=0

```
CALL MPI_REDUCE(inbuf, resultbuf, 1, MPI_INTEGER,  
                 MPI_SUM, root, MPI_COMM_WORLD, IERROR)
```

Python

- Python:

```
comm_world = MPI.COMM_WORLD  
snd_buf = np.array(value, dtype=np.intc)  
resultbuf = np.empty((), dtype=np.intc)  
comm_world.Reduce(snd_buf, resultbuf, op=MPI.SUM)
```
- The result is only placed in *resultbuf* at the root process.

op=MPI.SUM
and root=0
are defaults

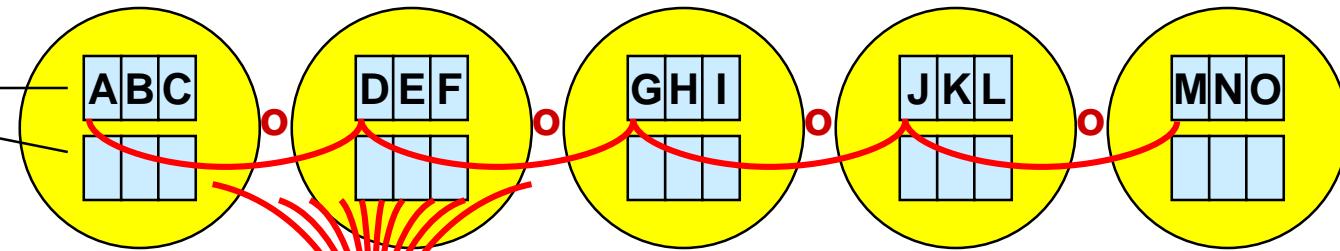
Predefined Reduction Operation Handles

Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

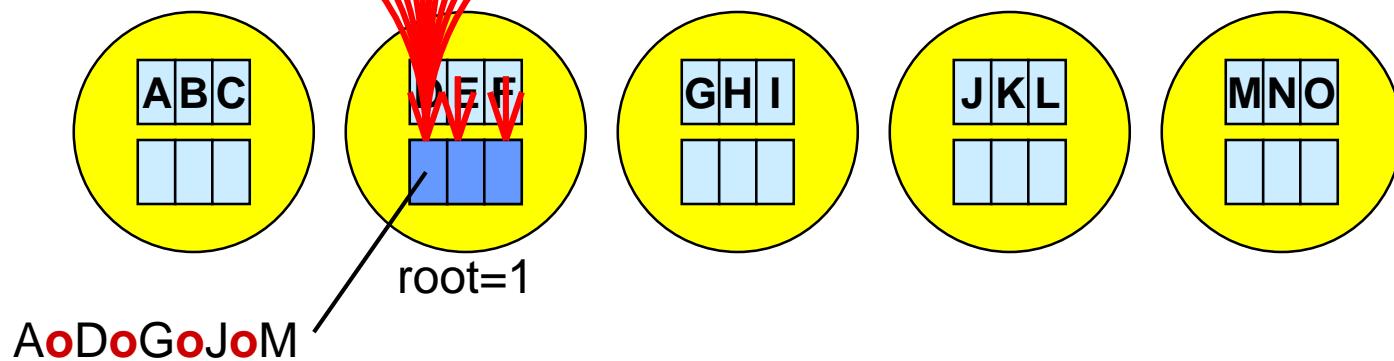
MPI_Reduce

before MPI_Reduce

- inbuf
- result



after



User-Defined Reduction Operations

- Operator handles
 - predefined – see table above
 - user-defined
- User-defined operation □:
 - associative
 - user-defined function must perform the operation $\text{vector_A} \square \text{vector_B}$
 - syntax of the user-defined function → MPI standard
- Registering a user-defined reduction function:
 - C/C++: `MPI_Op_create(MPI_User_function *func, int commute,
MPI_Op *op)`
 - Fortran: `MPI_OP_CREATE(FUNC, COMMUTE, OP, IERROR)`
 - Python: `op = MPI.Op.Create(func, commute=True or False)`
- COMMUTE tells the MPI library whether FUNC is commutative.

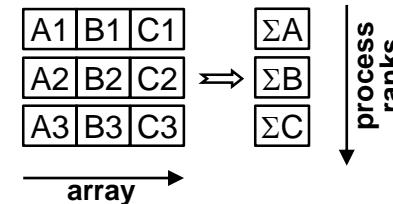
C

Fortran

Python

Variants of Reduction Operations

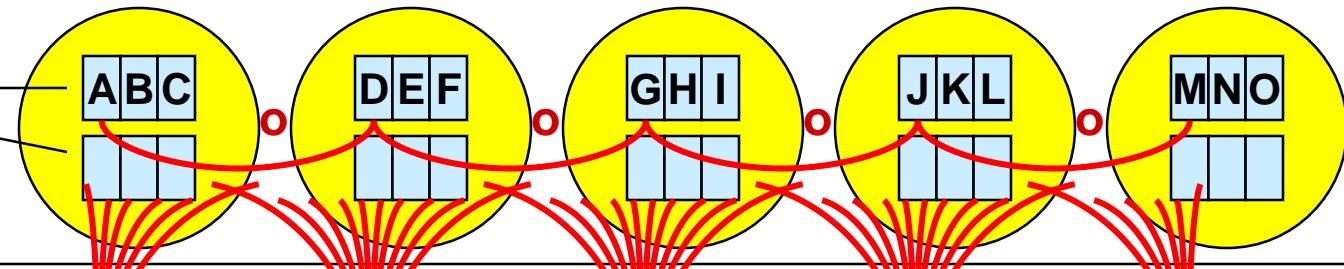
- MPI_Allreduce
 - no root,
 - returns the result in all processes
- MPI_Reduce_scatter_block and MPI_Reduce_scatter
 - result vector of the reduction operation is scattered to the processes into the real result buffers
- MPI_Scan
 - prefix reduction
 - result at process with rank $i :=$ reduction of inbuf-values from rank 0 to rank i
- MPI_Exscan
 - result at process with rank $i :=$ reduction of inbuf-values from rank 0 to rank **$i-1$**



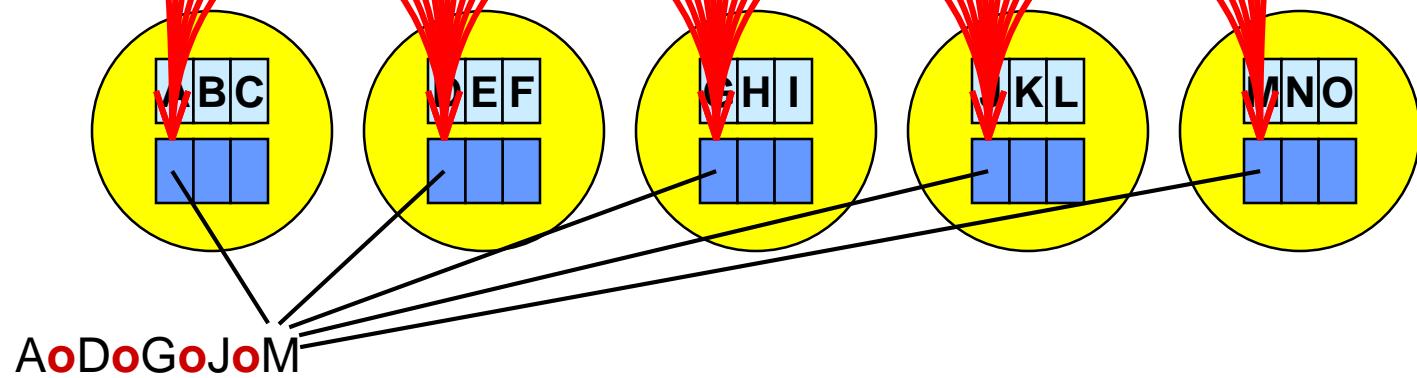
MPI_Allreduce

before MPI_Allreduce

- inbuf
- result

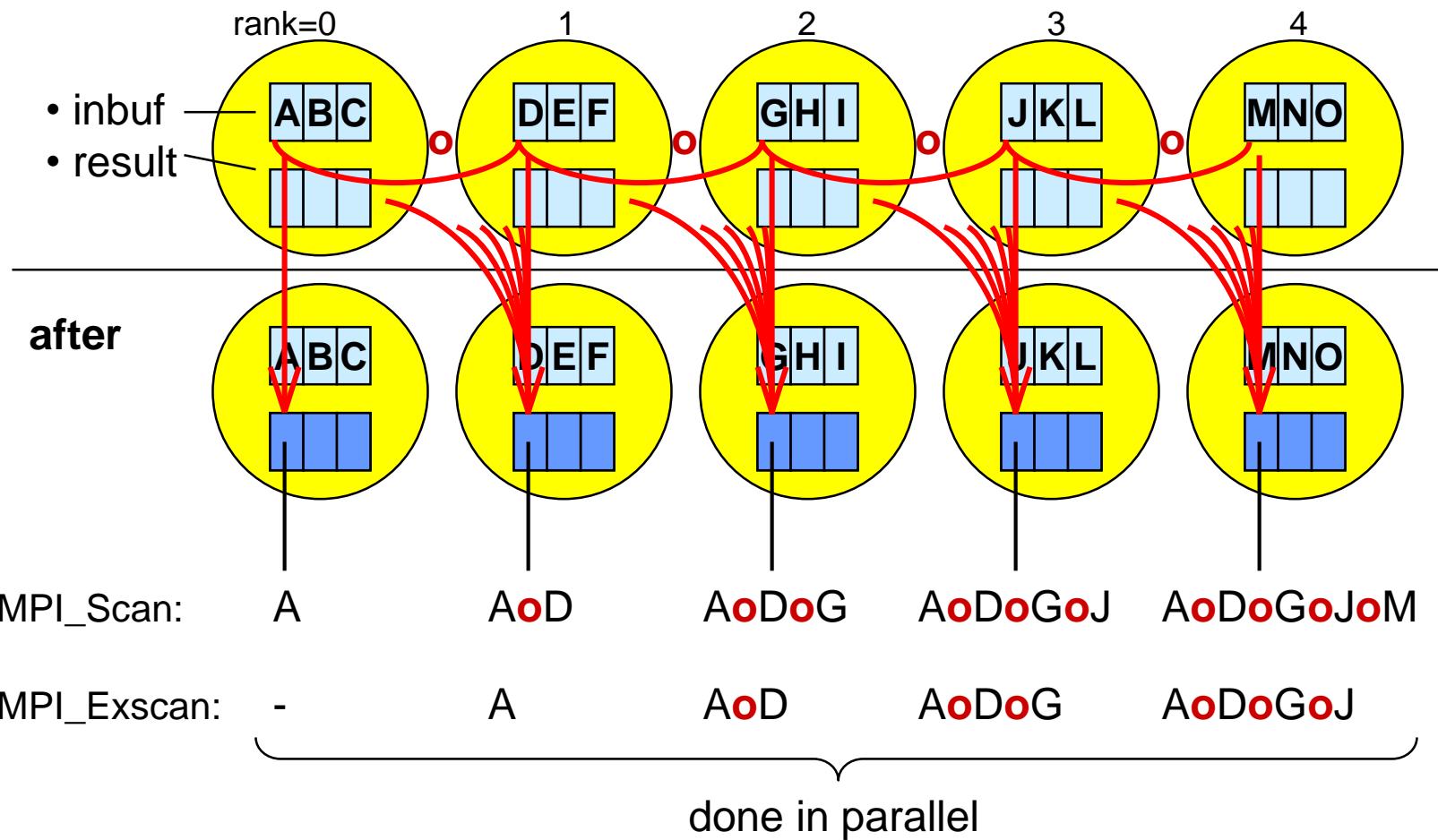


after

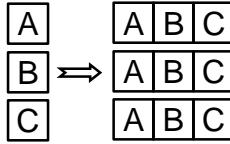
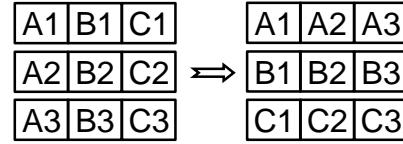


MPI_Scan and MPI_Exscan

before the call



Other Collective Communication Routines

- MPI_Allgather → similar to MPI_Gather, but all processes receive the result vector
- MPI_Alltoall → each process sends messages to all processes
- MPI_.....v (Gatherv, Scatterv, Allgatherv, Alltoallv, Alltoallw)
 - Each message has a different count and displacement
 - array of counts and array of displs (Alltoallw: also array of types)
 - interface does **not scale** to thousands of MPI processes!
 - Recommendation: One should try to use data structures with same communication size on all ranks.



Exercise 2 — Global reduction

- Rewrite the pass-around-the-ring program to use the MPI global reduction to perform the global sum of all ranks of the processes in the ring (and print it from all processes).
- Use **C** C/Ch6/allreduce-skel.c or **Fortran** F_30/Ch6/allreduce-skel_30.f90 or **Python** PY/Ch6/allreduce-skel.py
- I.e., the pass-around-the-ring communication loop must be totally substituted by one call to the MPI collective reduction routine.
- For the argument list, of MPI_Allreduce, please look into the MPI standard:
 - Go to the end of the standard (=[end of the MPI function index of MPI-4.0](#))
 - Go backward in the alphabet to MPI_Allreduce
 - Click on the underlined reference
 - MPI_Allreduce, [279](#), (in MPI-4.0)
 - Python: see also, e.g., [mpi4py.MPI.Comm — MPI for Python 3.1.1 documentation](#)

During the Exercise (15 min.)



Please stay here in the main room while you do this 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 enjoy the (short) break



Advanced Exercises — Global scan and sub-groups

1. Global scan:

- Rewrite the last program so that each process computes a partial sum, i.e., with MPI_Scan().
- mpirun -np 5 ./a.out | **sort -n** to get the output sorted by the ranks:

```
rank= 0 → sum=0
rank= 1 → sum=1
rank= 2 → sum=3
rank= 3 → sum=6
rank= 4 → sum=10
```

- Solution: MPI/tasks/C/Ch6/solutions/ring_advanced1_scan.c
and MPI/tasks/F_30/Ch6/solutions/ring_advanced1_scan_30.f90

2. Global sum in sub-groups:

- Rewrite the result of the advanced exercise of course Chapter 9.
- Compute the sum in each slice with the global reduction.
- (see course chapter 9-(1), Exercise 4b)
- Solution: MPI/tasks/C/Ch6/solutions/cylinder_advanced2_subtopology.c
and MPI/tasks/F_30/Ch6/solutions/cylinder_advanced2_subtopology_30.f90
and MPI/tasks/PY/Ch6/solutions/cylinder_advanced2_subtopology.py

Quiz on Chapter 6-(1) – Collective communication

- MPI Collective communication: Which are the **major rules when using collective communication** routines and that **do not apply** to point to point communication? Please try to find at least two or three:

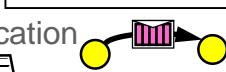
1. _____
2. _____
3. _____
4. _____

Chap.6 Collective Communication

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module mpi_f08



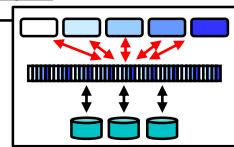
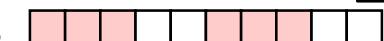
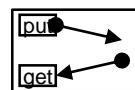
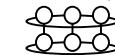
`MPI_Init()`
`MPI_Comm_rank()`



6. Collective communication

- (1) e.g., broadcast
- (2) advanced topics, e.g., nonblocking collectives, neighborhood comm.

7. Error Handling
8. Groups & communicators, environment management
9. Virtual topologies
10. One-sided communication
11. Shared memory one-sided communication
12. Derived datatypes
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I..... Nonblocking variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, ...

- **Nonblocking** collective operations do **not match** with **blocking** collective operations
With point-to-point message passing,
such matching is allowed
- Collective initiation and completion are separated
- **MPI_I...** calls are **local** (i.e., not synchronizing),
whereas the **corresponding MPI_Wait** collectively **synchronizes**
in same way as corresponding blocking collective procedure
- May have multiple outstanding collective communications on same communicator
- Ordered initialization on each communicator
- Parallel MPI I/O (except with shared file pointer):
The split collective interface may be deprecated in a future version of MPI

New in MPI-3.1

General progress rule of MPI

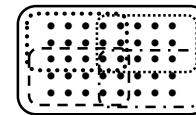
- MPI is mainly defined in a way that **progress** on communication (and ...) is **required only during MPI procedure calls**.
- But then, progress is required
 - for **all** outstanding (incomplete/nonblocking) communications
 - together with operation of the current communication (...) procedure call.
- See, e.g., in MPI-3.1
 - Sect. 3.5, page 41, and 3.7.4, page 56; Paragraphs “Progress”
 - Sect. 3.8.1 and 3.8.2 about MPI_(I)(M)PROBE
 - Sect. 3.8.4 Cancel, esp. page 73 lines 17-25 & MPI_Finalize Example 8.9, page 358-359
 - Sect. 5.12: Nonblocking collectives: Same rules as for nonblocking pt-to-pt
 - Sect. 11.7.3: Progress with one-sided communication, especially the **rationale at the end**
 - Sect. 12.4: MPI and Threads
 - Sect. 13.6.3: Progress with MPI-I/O
- None of these rules require progress outside of called MPI routines,
 - But MPI_Test and each MPI routine that blocks must do progress on any ongoing (i.e. nonblocking) communication
- Additional progress
 - By several calls to MPI_Test(), which enables progress
 - Use non-standard extensions to switch on asynchronous progress
 - E.g., with MPICH:
export MPICH_ASYNC_PROGRESS=1

Implies a helper thread and
MPI_THREAD_MULTIPLE,
see course Chapter 14. MPI and Threads



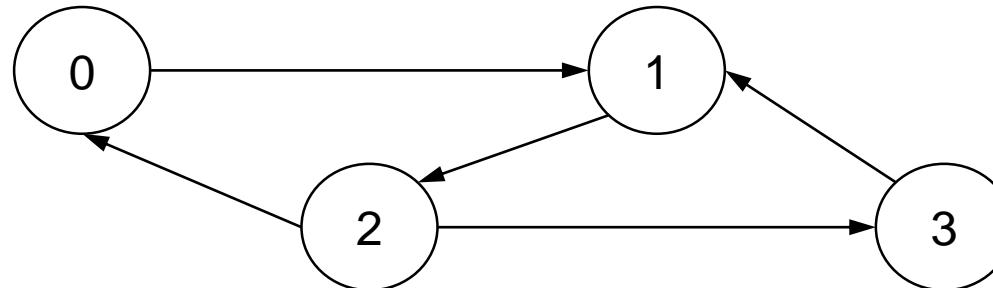
Opportunities with Nonblocking Collectives

- Offers opportunity to overlap
 - several collective communications,
e.g., on several overlapping communicators
 - **Without deadlocks or serializations!**
 - computation and communication
 - **For this, progress is needed**
 - **See previous slide**



Nonblocking Barrier: 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** (array with one logical entry per process)
 - Each sender tells all processes whether they will get a message or not.



- For the example with MPI_Ibarrier on next slide, we also need the following *local inquiry procedure*:
 - **MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);**
 - Result: flag == non-zero or .TRUE. → a message arrived and can be received with a local MPI_Recv, i.e., a subsequent corresponding MPI_Recv will **not** block
 - flag == 0 or .FALSE. → currently no incoming message with given source rank & tag & comm
 - See also course chapter 15

Nonblocking Barrier: Functional Opportunities – an Example

Principles:

1. Ssend
reports to
the sender
that **Recv** is
called on
the other
side.

2. Ibarrier
completes
when **all**
processes
reported (by
starting the
Ibarrier)
that **all** their
Ssend calls
are received
on their
other sides,
i.e., comple-
tely all **Recv**
calls are
called.

- The receiver (a) needs information, and (b) does not know the sending processes nor the number of sending processes (**nsp**), and (c) this number is small compared to the total number, and (d) The sender knows all its neighbors, which need some data.

- Solution with nonblocking barrier:

- Each process as a sender

- Loop over its neighbors, sending the data with **MPI_ISSEND**

- LOOP

- Process in the role being a receiver:

MPI_Iprobe(MPI_ANY_SOURCE); If there is a message then **MPI_Recv** for this one msg

- Process in the role being a sender:

Check whether all **Issend** calls are finished

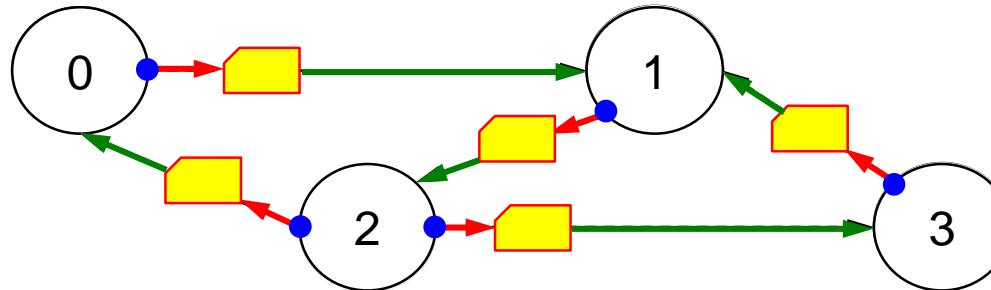
Important: The S=synchronous reports back to the sender that the RECV is called !

→ then start **MPI_Ibarrier** to signal to all other processes

that all **MPI_Issend** of this process are already received

(i.e. the corresponding **MPI_Recv** is already called)

- UNTIL **MPI_Ibarrier** finished (i.e. all processes signaled that all receives are called)



T. Hoefler, C. Siebert and A. Lumsdaine: Scalable Communication Protocols for Dynamic Sparse Data Exchange.
(In Proceedings of the 2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10), presented in Bangalore, India, pages 159--168, ACM, ISBN: 978-1-60558-708-0, Jan. 2010)



Collective Operations for Intercommunicators

- In MPI-1, collective operations are restricted to ordinary (intra) communicators.
- In MPI-2, most collective operations are extended by an additional functionality for intercommunicators
 - e.g., Bcast on a parents-children intercommunicator: sends data from one parent process to all children.
- Intercommunicators do not apply in
 - MPI_Scan, MPI_Iscan, MPI_Exscan, MPI_Iexscan,
 - MPI_(I)Neighbor_allgather(v)
 - MPI_(I)Neighbor_alltoall(v,w)

See course
Chapter 8-(2)
**Groups &
communicators**
→ Inter-
communicators

Sparse Collective Operations on Process Topology

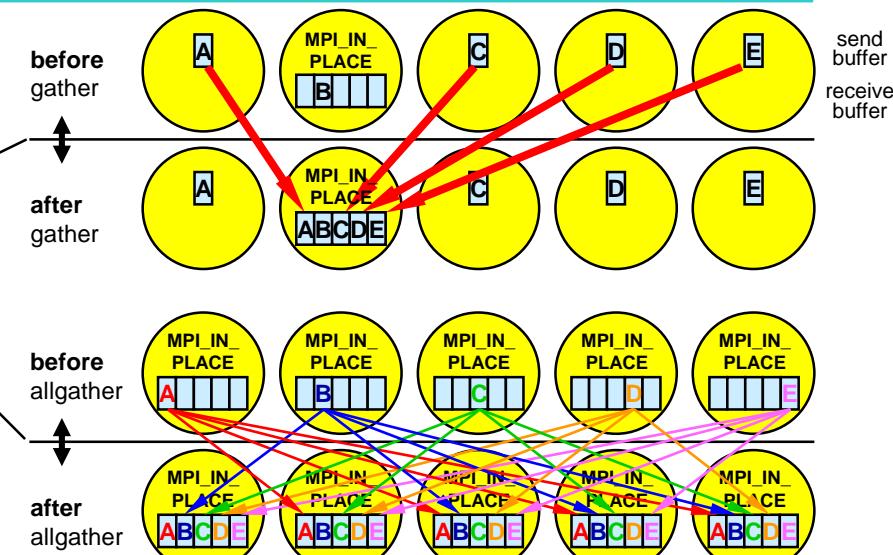
- MPI process topologies (Cartesian and (distributed) graph) usable for communication
 - MPI_(I)NEIGHBOR_ALLGATHER(V)
 - MPI_(I)NEIGHBOR_ALLTOALL(V,W)
- If the topology is the full graph, then neighbor routine is identical to full collective communication routine
 - Exception: s/rdispls in MPI_NEIGHBOR_ALLTOALLW are MPI_Aint
- Allows for optimized communication scheduling and scalable resource binding
- Cartesian topology:
 - Sequence of buffer segments is communicated with:
 - **direction=0 source, direction=0 dest, direction=1 source, direction=1 dest, ...**
 - Defined only for disp=1
 - If a source or dest rank is MPI_PROC_NULL then the buffer location is still there but the content is not touched.
 - See 2nd and 3rd advanced exercise of course Chapter 9 “**Virtual Topologies**”

See course
Chapter 9-(2)
Virtual Topologies

Extended Collective Operations — “In place” Buffer Specification

The **MPI_IN_PLACE** has two meanings:

- to **prohibit the local copy** with
→ $\text{sendbuf} = \text{MPI_IN_PLACE}$:
 - (I)GATHER(V) at root process
 - (I)ALLGATHER(V) at all processes
- to **overwrite input buffer** with the result:
($\text{sendbuf} = \text{MPI_IN_PLACE}$, input is taken from recvbuf , which is then overwritten)
 - (I)REDUCE at root
 - (I)ALLREDUCE, (I)REDUCE_SCATTER(_BLOCK), (I)SCAN, (I)EXSCAN, (I)ALLTOALL(V,W) at all processes
- Not available for
 - (I)BARRIER, (I)BCAST, (I)NEIGHBOR_ALLGATHER/ALLTOALL(V,W)
- Python: the constant is **MPI.IN_PLACE**

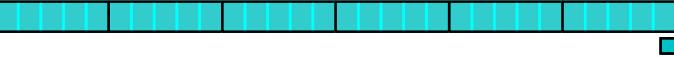


Exercise 3 — nonblocking barrier

In MPI/tasks/...

- Use **C** `C/Ch6/ibarrier-skel.c` or **Fortran** `F_30/Ch6/ibarrier-skel_30.f90` or **Python** `PY/Ch6/ibarrier-skel.py`
- Each process sends 0-4 messages to some other processes (see `number_of_dests`).
- The skeletons include already the `Issends` of these messages.
- The receiving processes do not know
 - how many messages must be received, and
 - from which processes they will come.
- The skeleton also includes the `Iprobe(...)` [please add the arguments] and the `Recv()`
- You should add the sender-side part of the nonblocking barrier algorithm presented within this course chapter. Hints:
 - With which one call can you check for the completeness of all nonblocking send requests?
 - `MPI_Ibarrier(..., &ib_rq)` should be called only once!
 - The `MPI_Test(&ib_rq, ...)` can be done only when `MPI_Ibarrier` is already called
- Please only fill in the _____ parts. Please do not modify the already given source code.
- `mpirun -np 4 ./a.out | sort +0 -1 +6 -7 +4r -5` (to check whether all messages are received)
- `mpirun -np 4 ./a.out | sort +0 -1 +2 -3 +4r -5 +6 -7` (to sort by processes / snd/rcv / partners)
- Finally, make a diff between your `ibarrier-skel.c` / `_30.f90` / `.py` and `solutions/ibarrier.c` / `_30.f90` / `.py`
 - `diff ibarrier-skel.c solutions/ibarrier.c`
 - `diff ibarrier-skel_30.f90 solutions/ibarrier_30.f90`
 - `diff ibarrier-skel.py solutions/ibarrier.py`

During the Exercise (30 min.)



For this complex exercise , I recommend:

- Work together with your colleagues **in your break out room**.
- Of course you all should look at the ibarrier-skel.c / _30.f90 by yourself
 - One of your group  should provide screen and editing.
 - And the others assist with ideas  to resolve each step. 

And *have fun with this complex exercise*

At the end you may use ZOOM file exchange to share the result.

Please do not look at the solution before you finished this exercise,
otherwise,
90% of your learning outcome may be lost



Exercise — nonblocking barrier — solutions

In the Ch6/solutions directory, you find

- ibarrier.c / _30.f90 / .py
 - the solution for the ..ibarrier-skel.c / _30.f90 / .py
- ibarrier-optimized.c / _30.f90 / .py
 - an optimized solution that additionally loops over the iprobe & recv
- ibarrier-optimized-test.c / _30.f90 / .py
 - same, but executes only each 10th iprobe & recv
- ibarrier-wrong.c, ibarrier-optimized-wrong.c, ibarrier-optimized-test-wrong.c / _30.f90 / .py
 - All *synchronous MPI_Ssend* calls are substituted by *standard MPI_Send*.
 - Therefore, the algorithm will start the ibarrier too early.
 - And therefore may stop before all messages are received.
 - Especially the test version shows always wrong results,
whereas the optimized version may sometimes receive all message by luck.
 - Incorrect programs may produce correct results ☺
→ therefore correct results never prove that the program is correct ☺

Advanced Exercise 4 — MPI_IN_PLACE

- Use **C** C/Ch6/in-place-skel.c or **Fortran** F_30/Ch6/in-place-skel_30.f90
- It is based on solutions/gather.c / _30.f90
- Your tasks:
 - Substitute the several 0 by a `root` variable initialized with `root=0`, compile and run
 - Substitute `root=0` by `root=num_procs-1`, compile and run
 - Modify your program that the `MPI_IN_PLACE` option is used for `MPI_Gather` (read the appropriate paragraph in the MPI description of `MPI_Gather`), compile and run
- Finally, make a diff between your in-place-skel.c / _30.f90 and solutions/in-place.c / _30.f90
 - `diff in-place-skel.c solution/in-place.c`
 - `diff in-place-skel_30.f90 solution/in-place_30.f90`

Any significant difference to your solution?



For private notes

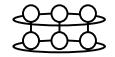
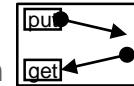
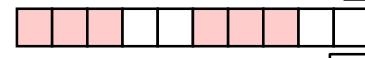
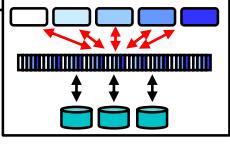
For private notes

For private notes

Chap.7 Error Handling

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 

7. Error Handling – error handler, codes, and classes

8. Groups & communicators, environment management 
9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication
12. Derived datatypes 
13. Parallel file I/O 
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

Error Handling → “assembler for parallel computing”

2-level-concept with **error codes** and **error classes**, see MPI-3.1/-4.0 Sect. 8/9.3-5

Most important aspects:

- The communication should be reliable (same rule as for processor and memory)
- If the MPI program is erroneous → **no warranties**:
 - by default: abort, if error detected by MPI library
i.e., error handler **MPI_ERRORS_ARE_FATAL** is the default
 - C/C++: **MPI_Comm_set_errhandler** (comm, **MPI_ERRORS_RETURN**);
Fortran: call **MPI_Comm_set_errhandler**(comm, **MPI_ERRORS_RETURN**, ierr)
Python: **comm.Set_errhandler(MPI.ERRORS_RETURN)** Newly added in MPI-4.0
 - directly after **MPI_INIT** with both **comm = MPI_COMM_WORLD** and **MPI_COMM_SELF**, then
 - **ierror returned by each MPI routine (except MPI window and MPI file routines)**
 - **undefined state after an erroneous MPI call has occurred (only MPI_Abort(...) should be still callable)**
 - Exception: MPI-I/O has default **MPI_ERRORS_RETURN**
 - Default can be changed through **MPI_FILE_NULL**:
 - **MPI_File_set_errhandler** (**MPI_FILE_NULL**, **MPI_ERRORS_ARE_FATAL**)
Python: **MPI.FILE_NULL.Set_errhandler(MPI.ERRORS_ARE_FATAL)**
 - See MPI-3.1 Sect. 13.7, page 555 / MPI-4.0 Sect. 14.7, page 719, and course Chapter 7
 - **MPI_ERRORS_ARE_FATAL** aborts the process and all connected processes
 - **MPI_ERRORS_ABORT** aborts only all processes of the related communicator New in MPI-4.0



For private notes

Chap.8 Groups & Communicators, Environment managem.

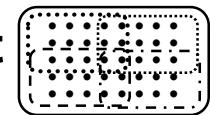
1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module mpi_f08
6. Collective communication
7. Error Handling



`MPI_Init()`
`MPI_Comm_rank()`

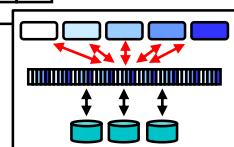
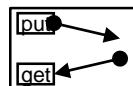
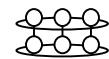


8. Groups & communicators, environment management



- (1) `MPI_Comm_split`, intra- & inter-communicators
- (2) Re-numbering on a cluster, collectives on inter-communicators, info object, naming & attribute caching, implementation information, Sessions Model

9. Virtual topologies
10. One-sided communication
11. Shared memory one-sided communication
12. Derived datatypes
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



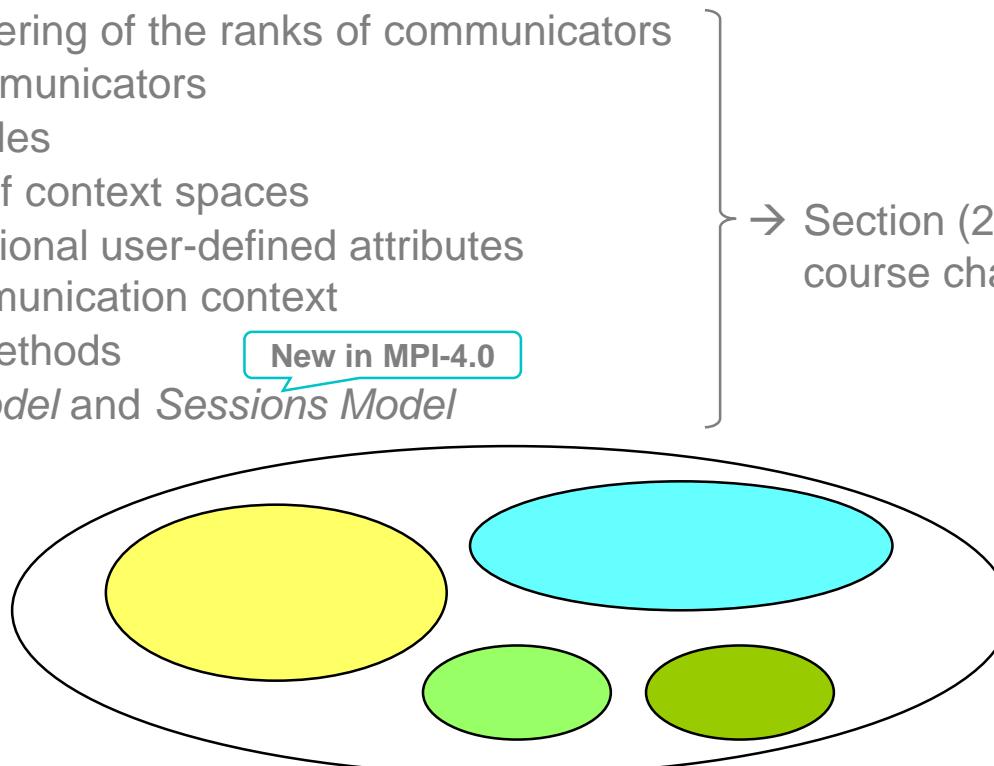
Goals

Support for libraries or application sub-spaces

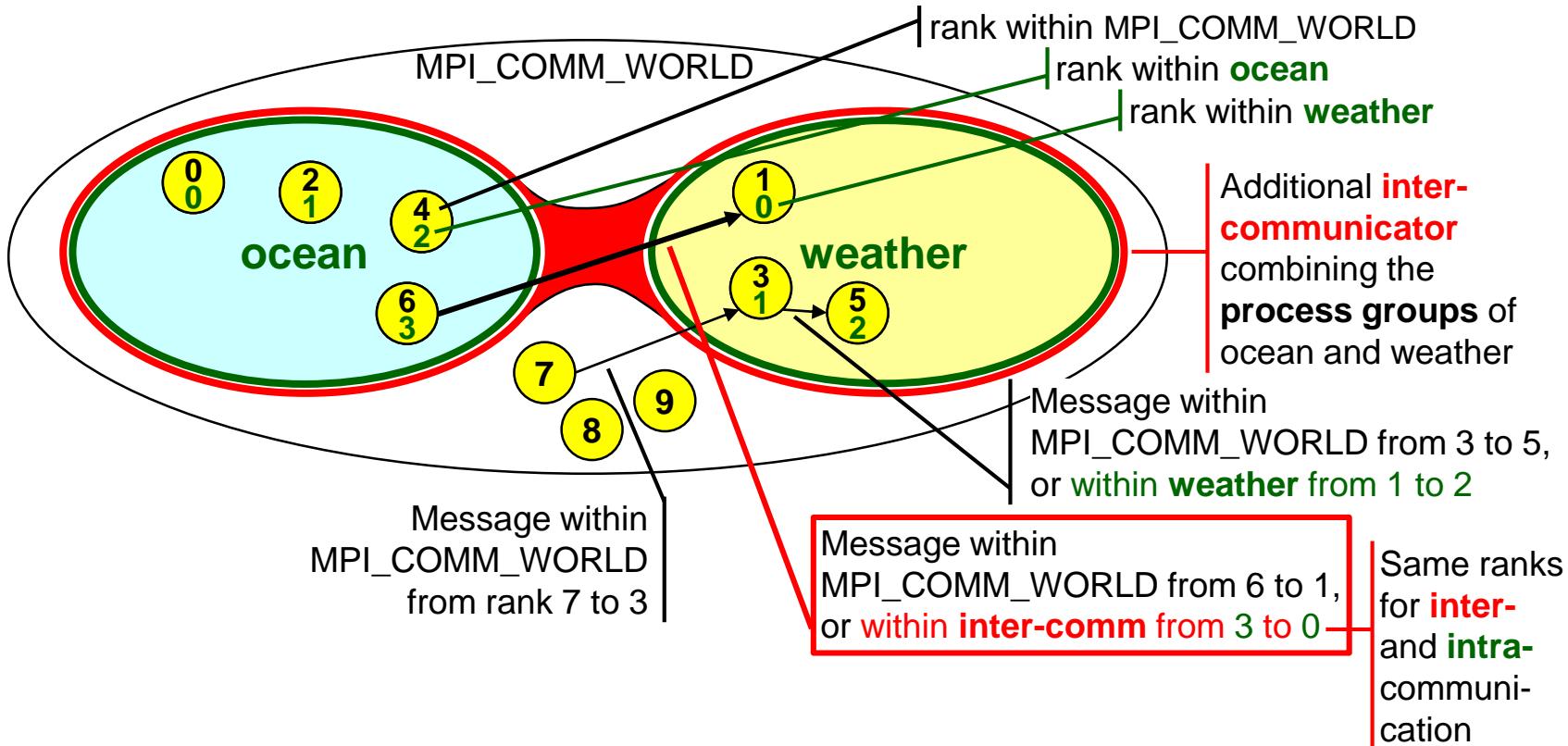
- Safe communication context spaces
 - e.g., for subsets of processes,
 - or duplicated communicators for independent software layers (middle-ware)
- Collective operations (→course Chapter 6) on a subset of processes
- Re-numbering of the ranks of communicators
- Inter-communicators
- Info handles
- Naming of context spaces
- Add additional user-defined attributes to a communication context
- Inquiry methods
- *World Model and Sessions Model*

A library should always use a duplicate of MPI_COMM_WORLD, and never MPI_COMM_WORLD itself.

→ Section (2) of this course chapter



Methods – e.g., for coupled applications

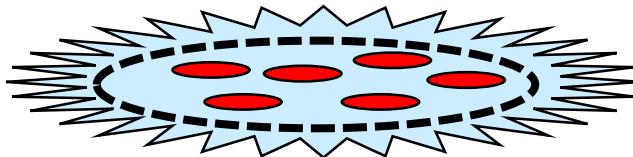


- **Sub-communicators:** Collectively defined communication sub-spaces
- **Intra- and inter-communicators**

Perfect for any communication
between processes of the two groups
(ocean and weather)

Sub-groups and sub-communicators (1)

Several ways to establish
sub-communicators



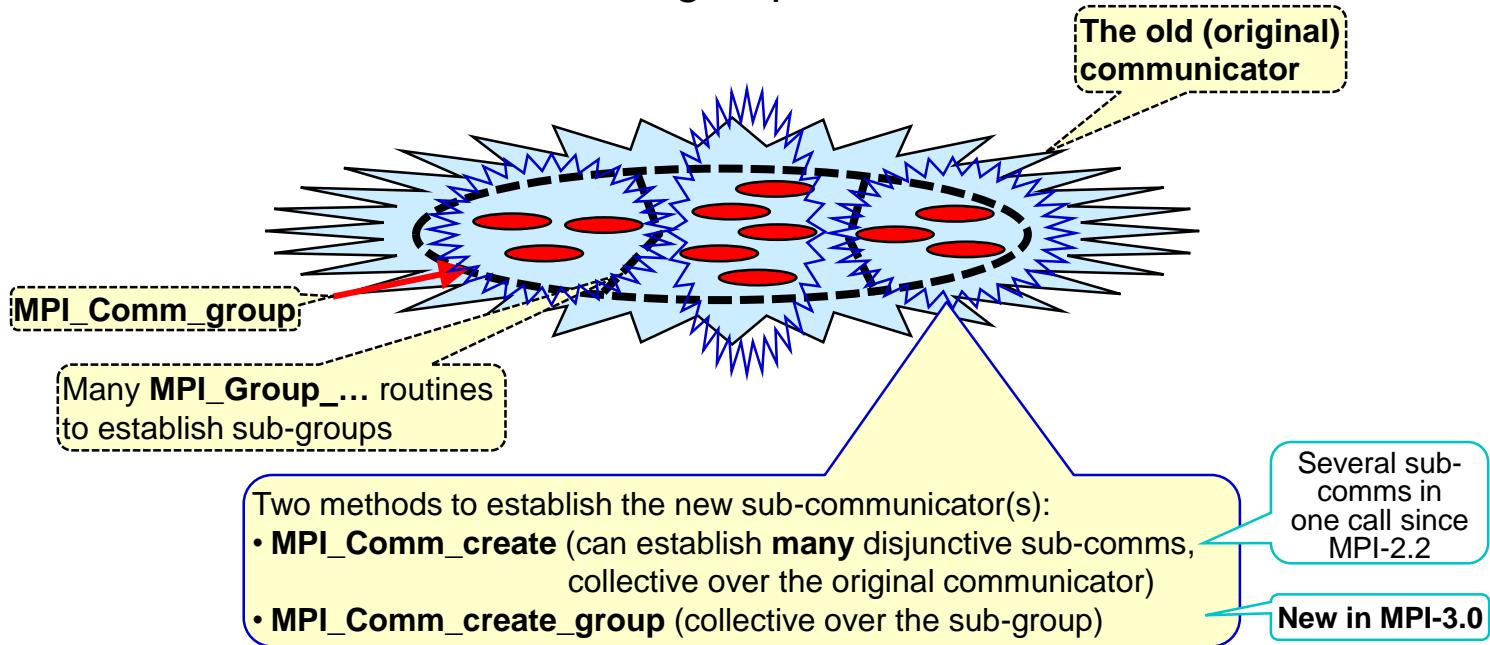
Two levels:

Scalability problems
when handling many
processes in each
process

- Group <----> of processes 
 - Without the ability to communicate
 - Local routines to build group & sub-sets
 - Same ranks as in related communicator
- Communicators 
 - Group of processes with additional ability to communicate

Sub-groups and sub-communicators (2)

- New sub-communicators via sub-groups



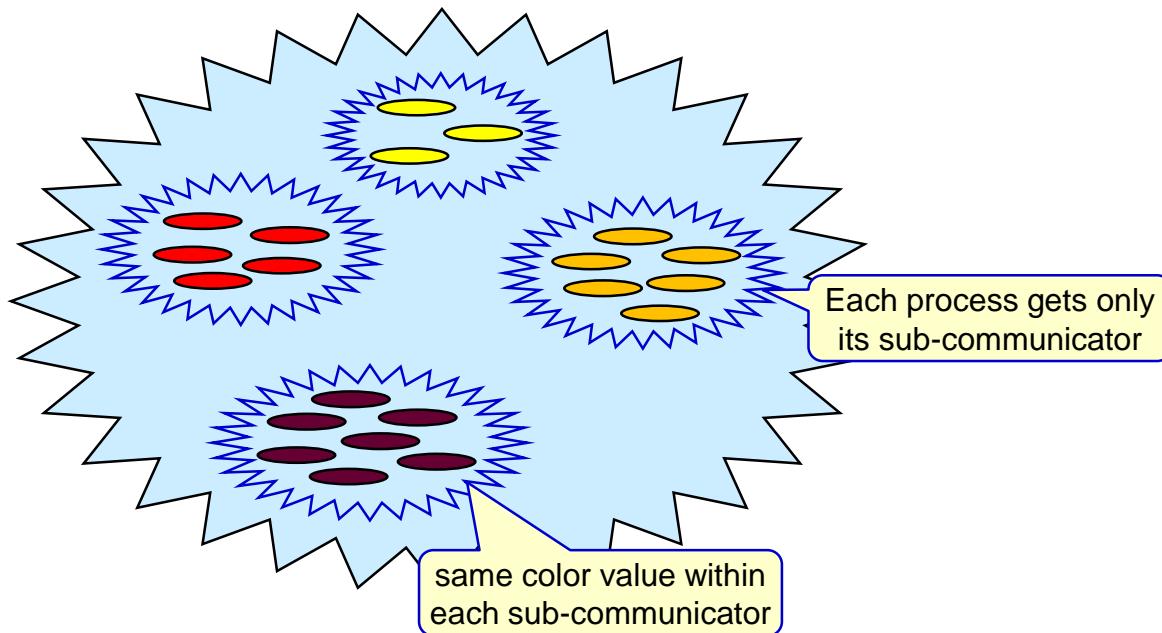
Sub-groups and sub-communicators (3)

- New sub-communicators via MPI_Comm_split

& MPI_Comm_split_type
→ course Chapter 11



New in
MPI-3.0



Example: MPI_Comm_split()

Creation is **collective** in the **old** communicator.

All processes with same color are grouped into separate sub-communicators

C

- C/C++: int MPI_Comm_split (MPI_Comm comm, int color, int key,
MPI_Comm *newcomm)

Each process
gets only its own
sub-communicator

Fortran

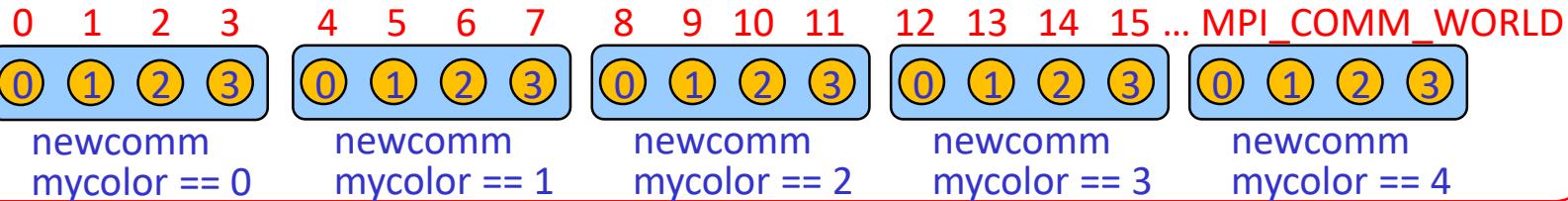
- Fortran: MPI_COMM_SPLIT (comm, color, key, newcomm, ierror)
mpi_f08:
TYPE(MPI_Comm) :: comm, newcomm
INTEGER :: color, key;
INTEGER, OPTIONAL :: ierror
mpi & mpif.h: INTEGER comm, color, key, newcomm, ierror

Python

- Python: newcomm = comm.Split(color=0, key=0)

Example:

```
int my_rank, mycolor, key, my_newrank;
MPI_Comm newcomm; Always 4 process get same color → grouped in an own newcomm
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
mymcolor = my_rank/4; key==0 → ranking in newcomm is sorted as in old comm
key = 0; key ≠ 0 → ranking in newcomm is sorted according key values
MPI_Comm_split (MPI_COMM_WORLD, mycolor, key, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```



Example: MPI_Group_range_incl() + MPI_Comm_create()

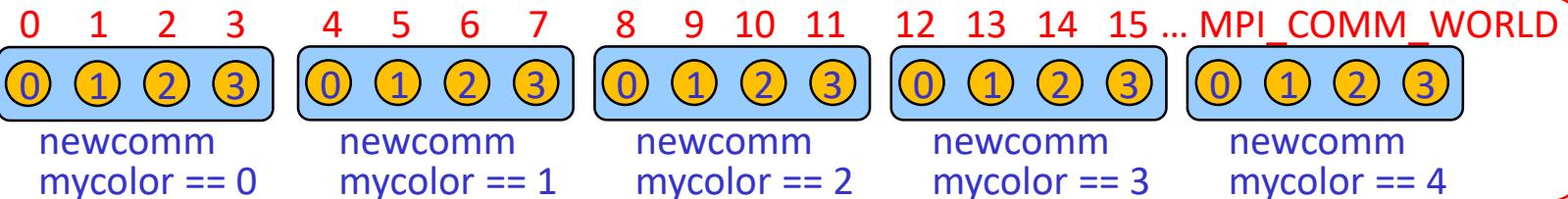
```
int my_rank, mycolor, my_newrank, ranges[1][3];
MPI_Group world_group, sub_group;
MPI_Comm newcomm;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_group (MPI_COMM_WORLD, &world_group)
mycolor = my_rank/4;
/* first rank of my range: */ ranges[0][0] = mycolor*4;
/* last rank of my range: */ ranges[0][1] = mycolor*4 + (4-1);
/* stride of ranks: */ ranges[0][2] = 1; Must be restricted to < num_procs
MPI_Group_range_incl ( world_group, 1, ranges, &sub_group );
MPI_Comm_create (MPI_COMM_WORLD, sub_group, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

Only one range
Three values per range:
[0]: first rank
[1]: last rank
[2]: stride

Group of the processes in MPI_COMM_WORLD.
Group and sub-group creation is local (non-collective).

Always 4 process get same color
→ grouped in an own sub_group
→ grouped in an own newcomm

(Sub-)communicator creation is collective.



Move to next course chapter, i.e.,
skip practical and (2)=advanced topics (13 slides)



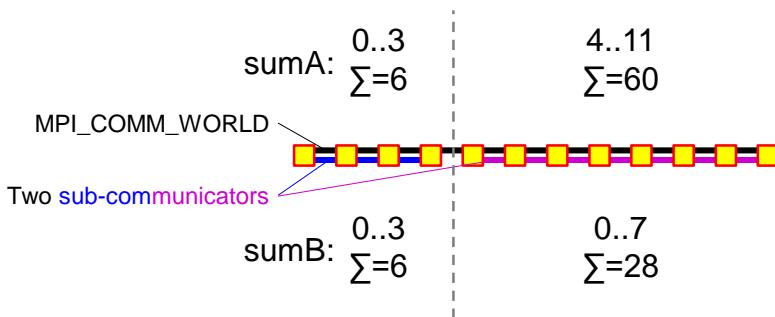
Skip practical, move to (2)=advanced topics



Exercise 1 — Two independent sub-communicators

In MPI/tasks/...

- Use **C** C/Ch8/comm-split-skel.c or **Fortran** F_30/Ch8/comm-split-skel_30.f90 or **Python** PY/Ch8/comm-split-skel.py
- Modify the *allreduce* program:
 - Split the communicator into 1/3 and 2/3, e.g., with $\text{color} = (\text{rank} > \left\lfloor \frac{\text{size}-1}{3} \right\rfloor)$ as input for **MPI_Comm_split**
 - Calculate **sumA** and **sumB** over all processes within each **sub-communicator**
 - **sumA: ranks in MPI_COMM_WORLD** (but summed up only within each sub-communicator)
 - E.g., with 12 processes → split into 4 & 8 with world ranks 0..3 & 4..11 and sums 6 & 60 → sumA
 - **sumB: ranks in new sub-communicators** (and summed up only within each sub-comm.)
 - E.g., with 12 processes → split into 4 & 8 with sub-comm ranks 0..3 & 0..7 and sums 6 & 28 → sumB
 - Use mpirun | sort +2n -3



Expected results with 12 processes:

```

PE world: 0, color=0 sub: 0, SumA= 6, SumB= 6 in sub_comm
PE world: 1, color=0 sub: 1, SumA= 6, SumB= 6 in sub_comm
PE world: 2, color=0 sub: 2, SumA= 6, SumB= 6 in sub_comm
PE world: 3, color=0 sub: 3, SumA= 6, SumB= 6 in sub_comm
PE world: 4, color=1 sub: 0, SumA= 60, SumB= 28 in sub_comm
PE world: 5, color=1 sub: 1, SumA= 60, SumB= 28 in sub_comm
PE world: 6, color=1 sub: 2, SumA= 60, SumB= 28 in sub_comm
PE world: 7, color=1 sub: 3, SumA= 60, SumB= 28 in sub_comm
PE world: 8, color=1 sub: 4, SumA= 60, SumB= 28 in sub_comm
PE world: 9, color=1 sub: 5, SumA= 60, SumB= 28 in sub_comm
PE world: 10, color=1 sub: 6, SumA= 60, SumB= 28 in sub_comm
PE world: 11, color=1 sub: 7, SumA= 60, SumB= 28 in sub_comm
  
```

During the Exercise (20 min.)



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



Important: To solve the exercises please **use previous slides and the provided .c/.f90 files**
(or in rare cases also the MPI standard)
but **no google search on the web** – otherwise you are too slow



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:

You may want to continue with the advanced exercise –

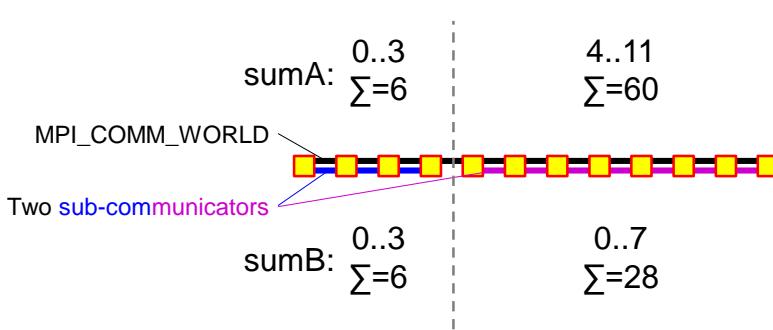


Please go to your breakout room before you start with the advance exercise!



Exercise 2 (advanced) — MPI_Comm_create

- Use **C** C/Ch8/comm-create-skel.c or **Fortran** F_30/Ch8/comm-create-skel_30.f90 or **Python** PY/Ch8/comm-create-skel.py
- Same as Exercise 1, but with **MPI_Comm_group()**, **MPI_Group_range_incl()**, and **MPI_Comm_create()**
 - instead of **MPI_Comm_split()**
 - Two different ranges for color 0 and 1 !!!
 - Same results in sumA/B as in Exercise 1
- Same details as in Exercise 1:
 - Split the communicator into 1/3 and 2/3, e.g., with $\text{color} = (\text{rank} > \left\lfloor \frac{\text{size}-1}{3} \right\rfloor)$
 - Calculate **sumA** and **sumB** over all processes within each **sub-communicator**
 - **sumA:** **ranks in MPI_COMM_WORLD** (but summed up only within each sub-communicator)
 - **sumB:** **ranks in new sub-communicators** (and summed up only within each sub-comm.)
 - Use mpirun | sort +2n -3



Expected results with 12 processes:

```
PE world:  0, color=0 sub:  0, SumA= 6, SumB= 6 in sub_comm
PE world:  1, color=0 sub:  1, SumA= 6, SumB= 6 in sub_comm
PE world:  2, color=0 sub:  2, SumA= 6, SumB= 6 in sub_comm
PE world:  3, color=0 sub:  3, SumA= 6, SumB= 6 in sub_comm
PE world:  4, color=1 sub:  0, SumA= 60, SumB= 28 in sub_comm
PE world:  5, color=1 sub:  1, SumA= 60, SumB= 28 in sub_comm
PE world:  6, color=1 sub:  2, SumA= 60, SumB= 28 in sub_comm
PE world:  7, color=1 sub:  3, SumA= 60, SumB= 28 in sub_comm
PE world:  8, color=1 sub:  4, SumA= 60, SumB= 28 in sub_comm
PE world:  9, color=1 sub:  5, SumA= 60, SumB= 28 in sub_comm
PE world: 10, color=1 sub:  6, SumA= 60, SumB= 28 in sub_comm
PE world: 11, color=1 sub:  7, SumA= 60, SumB= 28 in sub_comm
```



Quiz on Chapter 8-(1) – Groups & Communicators

A. Which is the easiest way to build a set of disjoint subcommunicators?

B. What are the major differences between

- a group of processes referenced by a group handle, and
- a communicator referenced by a communicator handle?

C. Can you produce with one call to MPI_Comm_create

- Only one subcommunicator?
- One or more disjoint subcommunicators?
- One or more overlapping subcommunicators?

D. If you split a communicator in five subcommunicators,
must you then use an array for 5 handles as *newcomm* output argument
instead of only a single *newcomm* handle variable?

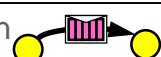
For private notes

For private notes

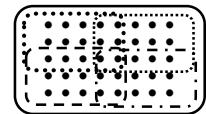
For private notes

For private notes

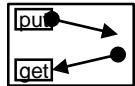
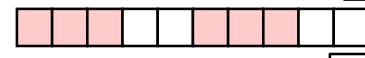
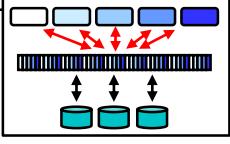
Chap.8 Groups & Communicators, Environment managem.

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling

8. Groups & communicators, environment management



- (1) MPI_Comm_split, intra- & inter-communicators
- (2) Re-numbering on a cluster, collectives on inter-communicators, info object, naming & attribute caching, implementation information, Sessions Model 

9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication
12. Derived datatypes 
13. Parallel file I/O 
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

Changing (= reordering / = re-numbering) ranks of a communicator

- Same rank-mapping provided by all processes:
 - communicator → MPI_Comm_group() → group handle
 - group handle → MPI_Group_incl(mapping_array) → reordered group
 - Communicator + reordered group → MPI_Comm_create() → reordered comm.
- Each process provides its new rank:
 - MPI_Comm_split (comm_old, /*color=*/ 0, /*key=*/ new_rank, &comm_new);

*see Advice to implementors of MPI_CART_MAP,
MPI-4.0, Sec. 8.5.8, page 415, lines 33-38*
- See also course Chapter 9-(3) *Optimization through reordering*

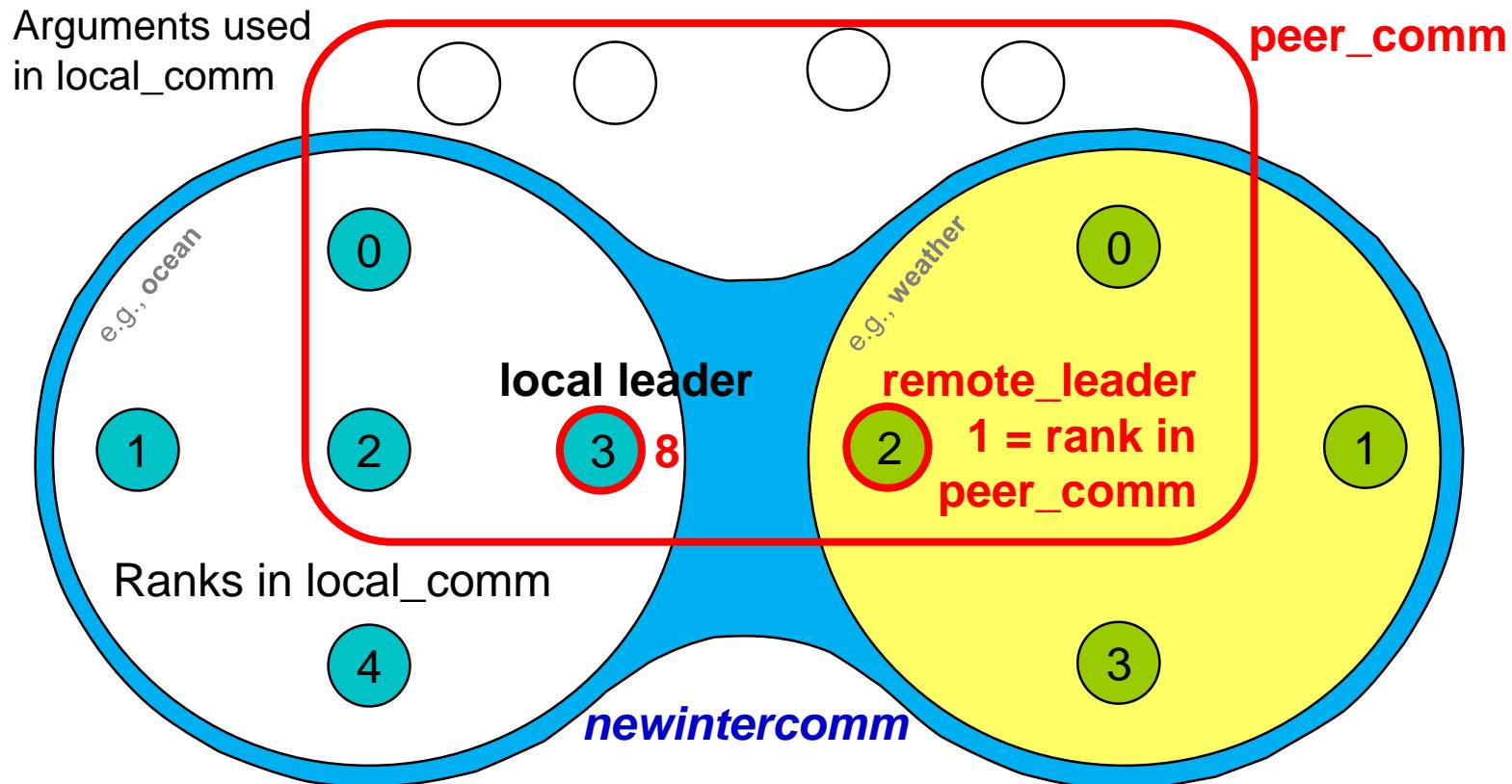
Inter-communicator –

combines a local and a remote communicator

Perfect for any communication
between the processes of two groups
(e.g., ocean and weather)

Since MPI-3.0:
tag is unused

- `MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm)`
In left processes: 3
In right processes: 2
- Significant only in left process 3
and right process 2



The arguments in the remote communicator are defined in the same way, but local and remote role is interchanged.

Inter-communicator – Accessors

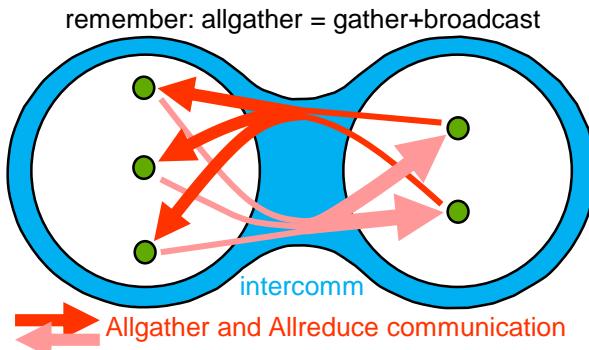
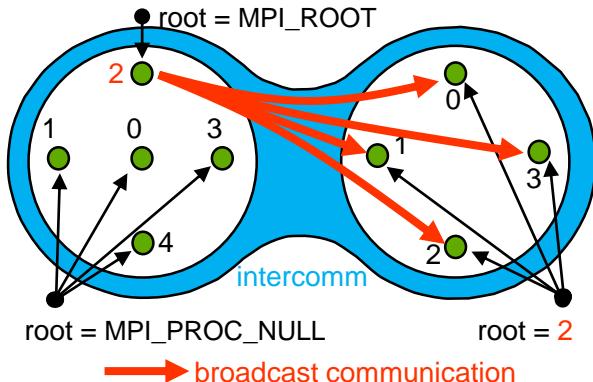
- Which routines can be applied for inter-communicator handles?
 - **MPI_Comm_Size, MPI_Comm_rank**
return same result as if applied to the local group
→ used in next Exercise 3
 - **MPI_Comm_inter_test(comm, flag)**
returns true in flag if comm is an inter-communicator
 - **MPI_Comm_remote_size(inter_comm, size)**
returns the size of the remote group
 - **MPI_Comm_group**
returns the local group
 - **MPI_Comm_remote_group(inter_comm, group)**
return the remote group
 - **MPI_Comm_compare**, see MPI-3.1 Chap. 6.6.1 or MPI-4.0 Chap. 7.6.1

— skipped —

Collective Operations for Intercommunicators

- Most collective operations are extended by an additional functionality for intercommunicators, e.g.,
 - Bcast on a *parents-children* intercommunicator:
Sends data
 - from one *parent* process
 - to all *children*
 - MPI_Allgather and MPI_Allreduce:
 - collects on each group
 - and sends it to the other group
- Intercommunicators do not apply in
 - MPI_(I)(Ex)Scan,
 - MPI_(I)Neighbor_allgather(v)
 - MPI_(I)Neighbor_alltoall(v,w)

Since MPI-2.0



MPI_Info Object

A general service for many MPI procedures

- An **MPI_Info** is an opaque object that consists of a set of (key,value) pairs
 - Both key and value are **strings**
 - A **key** should have a **unique** name within one info handle
 - Several keys are reserved by standard / implementation
 - Portable programs may use **MPI_INFO_NULL** as the info argument
 - Vendor keys are also portable, may be ignored by other libraries
 - Several sets of vendor-specific keys may be used
- Allows applications to **pass environment-specific information**
- Allow applications to **provide assertions** regarding their usage of MPI objects and operations → to improve performance or resource utilization (course chap. 2)
- Several functions provided to manipulate the info objects
- Used in:
 - Process Creation,*
 - Window Creation,*
 - MPI-I/O,* **New in MPI-4.0**
 - MPI_Comm_(i)dup_with_info,*
 - MPI_INFO_ENV*
- The key/value list returned by **MPI_Comm|File|Win_get_info** in the handle may differ from those set by the application during Comm|File|Win creation or stored with **MPI_Comm|File|Win_set_info**: The MPI library may or may not set or recognize some (system specific) hints.

Info handle

key1	value1
key2	value2
...	...

Internally stored in the MPI library

Adds 1 new entry, or modifies the value if key already exists

Example:

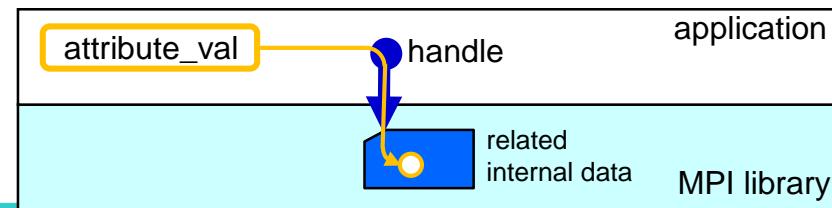
```
MPI_Info_info_noncontig;  
MPI_Info_create (&info_noncontig);  
MPI_Info_set (info_noncontig,  
              "alloc_shared_noncontig", "true");  
MPI_Win_allocate_shared (... , info_noncontig, ...);
```

Creates the list with 0 entries

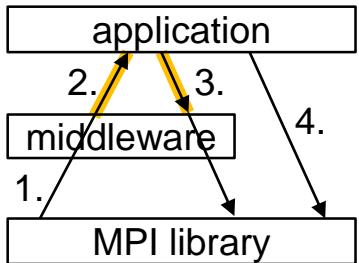
New in MPI-4.0

New in MPI-4.0

Naming & attribute caching



Problem:



1. The MPI library provides communicators for a middleware, and
2. the middleware hands it over to the application,
3. which gives it back to the middleware, or
4. the MPI library, and the middleware wants to remember middleware-specific data — with such a communicator handle

Caching attributes on handles in two steps:

- 1st step – generating a keyval:
 - MPI_Comm_create_keyval
(comm_copy_attr_fn, comm_delete_attr_fn, **comm_keyval**, extra_state)
- 2nd step – storing & retrieving an attribute on/from a communicator handle:
 - MPI_Comm_set_attr (comm, comm_keyval, **attribute_val**)
 - MPI_Comm_get_attr (comm, comm_keyval, **attribute_val**, flag)
- Other routines:
 - MPI_Comm_delete_attr & MPI_Comm_free_keyval

Other objects: Same method for **datatypes** and **windows**

Examples: See MPI-3.1/-4.0 Sect. 17.2.7/19.3.7 *Attributes*

Name an object:

- MPI_Comm_set_name(comm, comm_name),
MPI_Comm_get_name(...)

Environment inquiry – implementation information (1)

Version of MPI

- Compile time information
 - integer MPI_VERSION=3, MPI_SUBVERSION=1
 - Valid pairs: (3,1), (3,0), (2,2), (2,1), (2,0), and (1,2).
- Runtime information
 - MPI_Get_version(*version, subversion*)
 - Can be called before MPI_Init and after MPI_Finalize

New in MPI-3.0

Inquire start environment

- Predefined info object **MPI_INFO_ENV** (in the World Model) or info handle created with **MPI_Info_create_env** (in the Sessions Model) holds arguments from
 - mpiexec, or
 - MPI_COMM_SPAWN

New in MPI-4.0

see a few slides later

Inquire processor name

- MPI_Get_processor_name(*name, resultlen*)

Caution: several MPI ranks may return the same name,
e.g., the node name

 New in MPI-4.0

Environment inquiry – implementation information (2)

Environmental inquiries

C

Fortran

Python

- C: `MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &p, &flag)`
 - Will return in *p* a pointer to an int containing the *attribute_val*
- Fortran: `MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, attribute_val, flag, ierror)`
- Python: `attribute_val = MPI.COMM_WORLD.Get_attr(keyval)`
- with keyval =
 - **MPI_TAG_UB** Python: `MPI.TAG_UB` C: pointer based attributes
Fortran: `integer(kind=MPI_ADDRESS_KIND)` based attributes
 - **MPI_HOST** May be deprecated in MPI-4.1 → returns host-rank (if exists) or `MPI_PROC_NULL` (if there is no host)
 - **MPI_IO** → returns `MPI_ANY_SOURCE` in *attribute_val* (if every process can provide I/O)
 - **MPI_WTIME_IS_GLOBAL** → returns 1 in *attribute_val* (if clocks are synchronized), otherwise, 0

Examples: see MPI-3.1, Sect. 17.2.7, page 664, line 43 – page 665, line 13 or
MPI-4.0, Sect. 19.3.7, page 852, line 29-47

World Model and Sessions Model

- **The World Model**
 - MPI_COMM_WORLD can be used between MPI_Init and MPI_Finalize
 - Exactly one call to MPI_Init and MPI_Finalize
 - Problem, if several independent software layers want to use MPI:
 - Each layer can duplicate MPI_COMM_WORLD using MPI_COMM_DUP()
 - But there is no rule on which layer calls MPI_Init and which one MPI_Finalize
- **The Sessions Model**
 - Each independent software layer **xxx** can initialize and finalize MPI, e.g., as follows:
 - As part of layer_xxx_init
 - MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &session);
 - MPI_Group_from_session_pset(session, "mpi://WORLD", &xxx_world_group);
 - MPI_Comm_create_from_group(xxx_world_group, "stringtag_xxx", MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &xxx_world_comm);
 - MPI_Group_free(&xxx_world_group);
 - As part of layer_xxx_finalize
 - MPI_Comm_free(&xxx_world_comm);
 - MPI_Session_finalize(&session);
 - **Caution:** MPI objects derived from different MPI Session handles shall **not** be intermixed with each other in a single MPI procedure call.
- An MPI application may use the World Model (not more than once) together with the Sessions Model (with several overlapping or non-overlapping sessions)

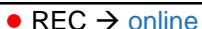
Since MPI-2.0:
duplicates with associated key values, topology and info hints.
Since MPI-4.0:
Now without info hints

Conclusions of this course chapter

- Sub-communicators
 - Scalability problems
 - methods with local data with $O(\#MPI_COMM_WORLD)$ are not scalable
 - e.g., `MPI_Comm_group(MPI_COMM_WORLD, group)`
 - Sub-communicator splitting is a scalable interface
 - This does not guarantee that an MPI implementation is scalable
 - Inter-communicators
 - mainly used in coupled applications
 - Also used for `MPI_Comm_spawn`
(See course Chapter 16 Process creation and management)
- Info Object → used in several interfaces → `MPI_INFO_NULL` is always a choice
- Object naming & attribute caching – useful only for libraries between MPI and appl.
- Environment inquiry → small functionality, `MPI_INFO_ENV` new in MPI-3.0
- The Sessions Model → a method to init/finalize MPI within independent application components / software layers

New in MPI-4.0

 New in MPI-4.0

© 2000-2021 HLRS, Rolf Rabenseifner  REC → [online](#)

MPI course → Chap.8-(2) Groups & Communicators, advanced topics

Go back to practical of (1) →
`MPI_Comm_split & MPI_Comm_create`

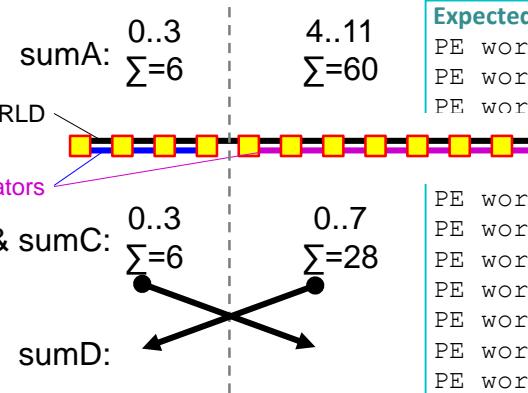


Slide 235 / 593

Exercise 3 — Create an inter-communicator

In MPI/tasks/...

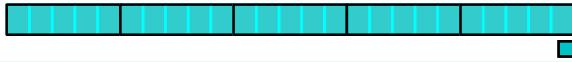
- Use **C** C/Ch8/intercomm-skel.c or **Fortran** F_30/Ch8/intercomm-skel_30.f90
or **Python** PY/Ch8/intercomm-skel.py
- Same details as in Exercise 1:
 - Split the communicator into 1/3 and 2/3, e.g., with color = ($\text{rank} > \lfloor \frac{\text{size}-1}{3} \rfloor$)
 - Calculate **sumA** and **sumB** over all processes within each **sub-communicator**
 - **sumA**: ranks in **MPI_COMM_WORLD** (but summed up only within each sub-communicator)
 - **sumB**: **ranks in new sub-communicators** (and summed up only within each sub-comm.)
 - Use mpirun | sort +2n -3
- And additionally:
 - **Create an inter-communicator from the two sub-communicators**
 - **Choose rank 0 as local leader in both sub_comm**
 - **sumC: ranks in inter_comm** summed up over the sub_comm
 - **sumD: ranks in inter_comm** summed up over the **inter_comm**



Expected results with 12 processes:

PE world: 0, color=0 sub:	0 inter:	0 SumA= 6, SumB= 6, SumC= 6, SumD= 28
PE world: 1, color=0 sub:	1 inter:	1 SumA= 6, SumB= 6, SumC= 6, SumD= 28
PE world: 2, color=0 sub:	2 inter:	2 SumA= 6, SumB= 6, SumC= 6, SumD= 28
PE world: 3, color=0 sub:	3 inter:	3 SumA= 6, SumB= 6, SumC= 6, SumD= 28
PE world: 4, color=1 sub:	0 inter:	0 SumA= 60, SumB= 28, SumC= 28, SumD= 6
PE world: 5, color=1 sub:	1 inter:	1 SumA= 60, SumB= 28, SumC= 28, SumD= 6
PE world: 6, color=1 sub:	2 inter:	2 SumA= 60, SumB= 28, SumC= 28, SumD= 6
PE world: 7, color=1 sub:	3 inter:	3 SumA= 60, SumB= 28, SumC= 28, SumD= 6
PE world: 8, color=1 sub:	4 inter:	4 SumA= 60, SumB= 28, SumC= 28, SumD= 6
PE world: 9, color=1 sub:	5 inter:	5 SumA= 60, SumB= 28, SumC= 28, SumD= 6
PE world: 10, color=1 sub:	6 inter:	6 SumA= 60, SumB= 28, SumC= 28, SumD= 6
PE world: 11, color=1 sub:	7 inter:	7 SumA= 60, SumB= 28, SumC= 28, SumD= 6

During the Exercise (25 min.)



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



And have fun with this a bit longer 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:

What all could you do wrong in this exercise?



Advanced Exercise 4 — MPI_TAG_UB

- Use **C** C/Ch8/tag-ub-skel.c or **Fortran** F_30/Ch8/tag-ub-skel_30.f90 or **Python** PY/Ch8/tag-ub-skel.py
- Goal: Inquiry the upper bound for MPI tag arguments
- Hint:
 - See the reference to the MPI standard on the previous slide on “*Environment inquiry – implementation information (2)*”
- Expected results (with mpirun -np 1 ./a.out)
 - PE 0, tag_ub=2147483647, flag=1 ($=2^{31}-1$) with OpenMPI 2.1.6.0
 - PE 0, tag_ub=268435455, flag=1 ($=2^{28}-1$) with mpich 3.2.1
 - PE 0, tag_ub=32767, flag=1 ($=2^{15}-1$) at least required
- Solutions: **C** C/Ch8/solutions/tag-ub.c or **Fortran** F_30/Ch8/solutions/tag-ub_30.f90 or **Python** PY/Ch8/solutions/tag-ub.py



Quiz on Chapter 8-(2) – Groups & Communicators

A. Which data can be stored in an info handle?

B. Which rules apply to such content?

1. _____

2. _____

3. _____

For private notes

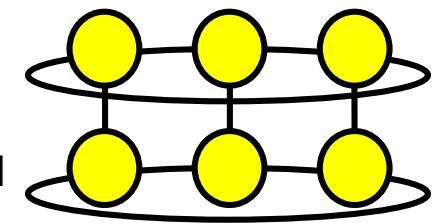
Chap.9 Virtual Topologies

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module mpi_f08
6. Collective communication
7. Error Handling
8. Groups & communicators, environment management

9. Virtual topologies

- **(1) A multi-dimensional process naming scheme**
- **(2) Neighborhood communication + MPI_BOTTOM**
- **(3) Optimization through reordering**

10. One-sided communication
11. Shared memory one-sided communication
12. Derived datatypes
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



Example

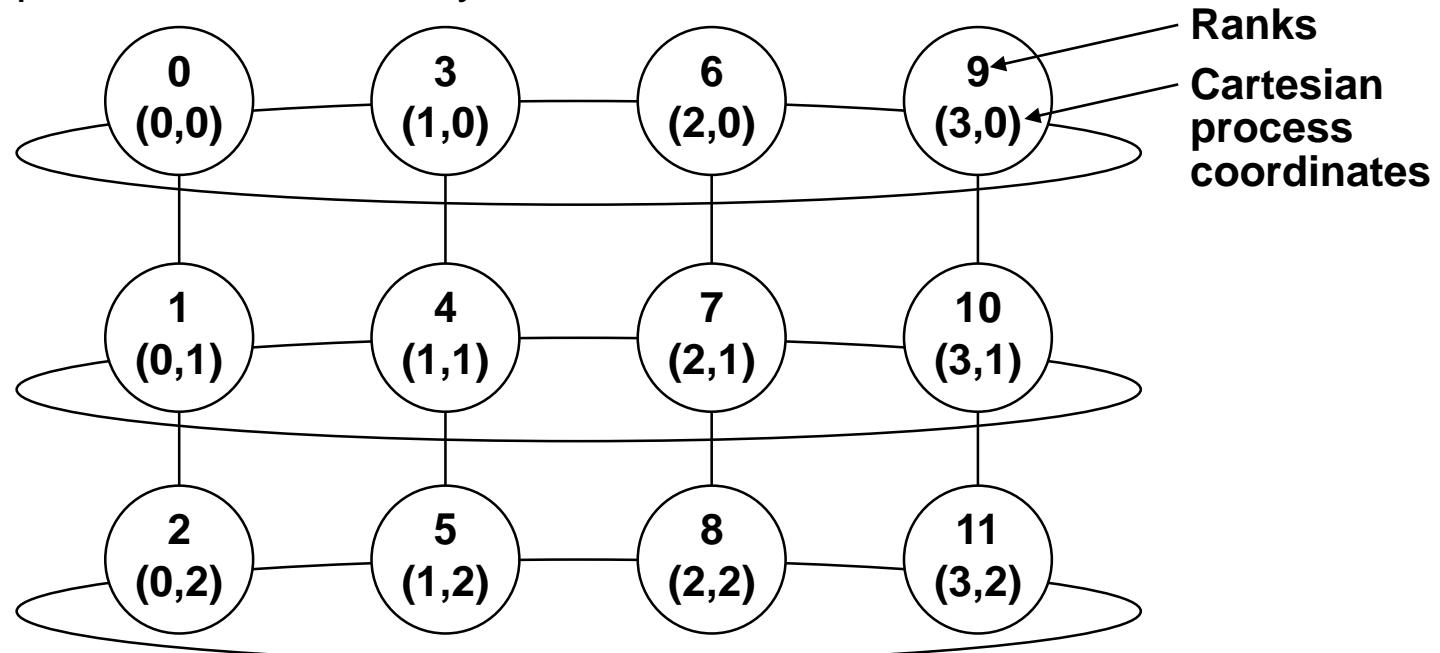
- Global data array $A(1:3000, \quad 1:4000, \quad 1:500) = 6 \cdot 10^9$ words
- on 3 x 4 x 5 = **60 processes**
- process coordinates **0..2,** **0..3,** **0..4**
- example:
on process **ic₀=2,** **ic₁=0,** **ic₂=3** **(rank=43)**
decomposition, e.g., $A(2001:3000, \quad 1:1000, \quad 301:400) = 0.1 \cdot 10^9$ words
- **process coordinates:** handled with **virtual Cartesian topologies**
- Array decomposition: handled by the application program directly

Virtual Topologies

- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimize communications → see course Chapter 9-(3)

How to use a Virtual Topology

- Creating a topology produces a new communicator.
- MPI provides mapping functions:
 - to compute process ranks, based on the topology naming scheme,
 - and vice versa.
- Example: 2-dimensional cylinder



Topology Types

- Cartesian Topologies
 - each process is *connected* to its neighbor in a virtual process grid,
 - boundaries can be cyclic, or not,
 - processes are identified by Cartesian coordinates,
 - of course,
communication between any two processes is still allowed.
- Graph Topologies
 - general graphs,
 - two interfaces:
 - **`MPI_Graph_create`** (since MPI-1)
 - **`MPI_Dist_graph_create_adjacent`** & **`MPI_Dist_graph_create`** (new scalable interface since MPI-2.2)
 - not covered here.
 - See also slides on “**Unstructured Grids**” at the end of course Chapter 9-(3)
 - See also T. Hoefler and M. Snir. 2011. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*. ACM, 75–85.

Creating a Cartesian Virtual Topology

C

Fortran

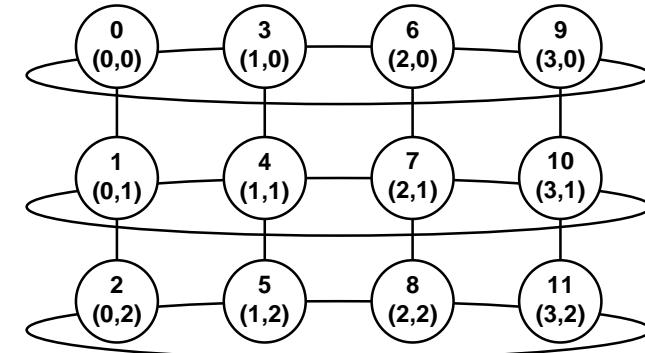
Python

- C/C++: `int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
int *periods, int reorder, MPI_Comm *comm_cart)`
- Fortran: `MPI_CART_CREATE(comm_old, ndims, dims, periods,
reorder, comm_cart, ierror)`
`mpi_f08: TYPE(MPI_Comm) :: comm_old, comm_cart
INTEGER :: ndims, dims(*),
LOGICAL :: periods(*), reorder
INTEGER, OPTIONAL :: ierror`
`mpi & mpif.h: INTEGER comm_old, ndims, dims(*), comm_cart, ierror ; LOGICAL periods(*), reorder`
- Python: `comm_cart = comm_old.Create_cart(dims, periods, reorder)`

see [mpi4py.MPI.Intracomm — MPI for Python 3.1.1 documentation](#)

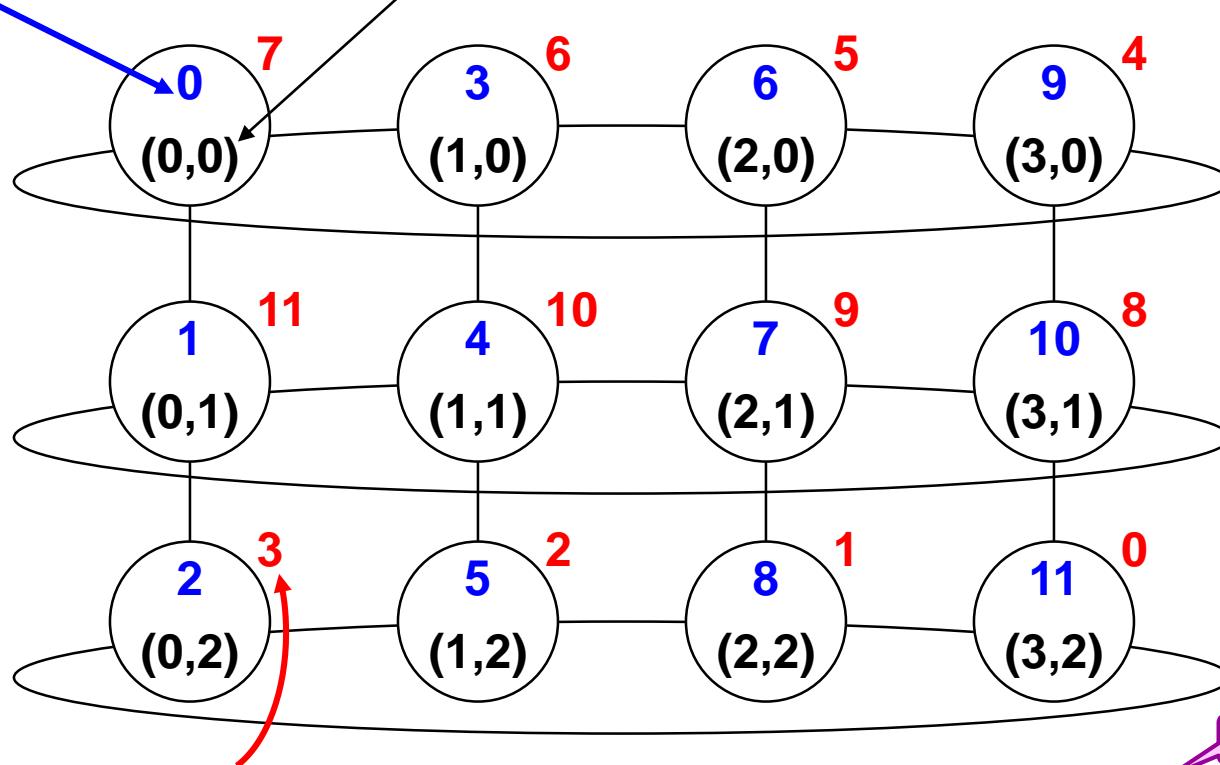
```
comm_old = MPI_COMM_WORLD
ndims = 2
dims = ( 4,      3      )
periods = ( 1,      0      ) (in C)
periods = ( .true., .false. ) (in Fortran)
reorder = see next slide
```

e.g., size==12 factorized
with `MPI_Dims_create()`,
see later the slide „Typical usage of
`MPI_Cart_create & MPI_Dims_create`“
and the advanced exercise 1b



Example – A 2-dimensional Cylinder

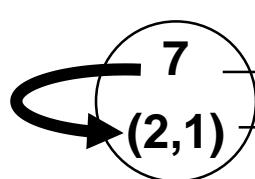
- Ranks and Cartesian process coordinates in `comm_cart`



- Ranks in `comm` and `comm_cart` may differ, if reorder == non-zero or `.TRUE.`
- This reordering can allow MPI to optimize communications

e.g., 1

Cartesian Mapping Functions



- Mapping ranks to virtual process grid coordinates

C

- C/C++: `int MPI_Cart_coords(MPI_Comm comm_cart, int rank, int maxdims, int *coords)`

- Fortran: `MPI_CART_COORDS(comm_cart, rank, maxdims, coords, ierror)`

mpi_f08: `TYPE(MPI_Comm) :: comm_cart`
 `INTEGER :: rank, maxdims, coords(*)`
 `INTEGER, OPTIONAL :: ierror`

mpi & mpif.h: `INTEGER comm_cart, rank, maxdims, coords(*), ierror`

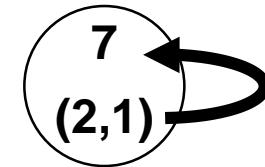
- Python: `coords = comm_cart.Get_coords(rank)`

see [mpi4py.MPI.Cartcomm — MPI for Python 3.1.1 documentation](#)

Python

Cartesian Mapping Functions

- Mapping virtual process grid coordinates to ranks



C

- C/C++: `int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)`

- Fortran: `MPI_CART_RANK(comm_cart, coords, rank, ierror)`

```
mpi_f08:      TYPE(MPI_Comm)      :: comm_cart  
              INTEGER             :: coords(*), rank  
              INTEGER, OPTIONAL    :: ierror
```

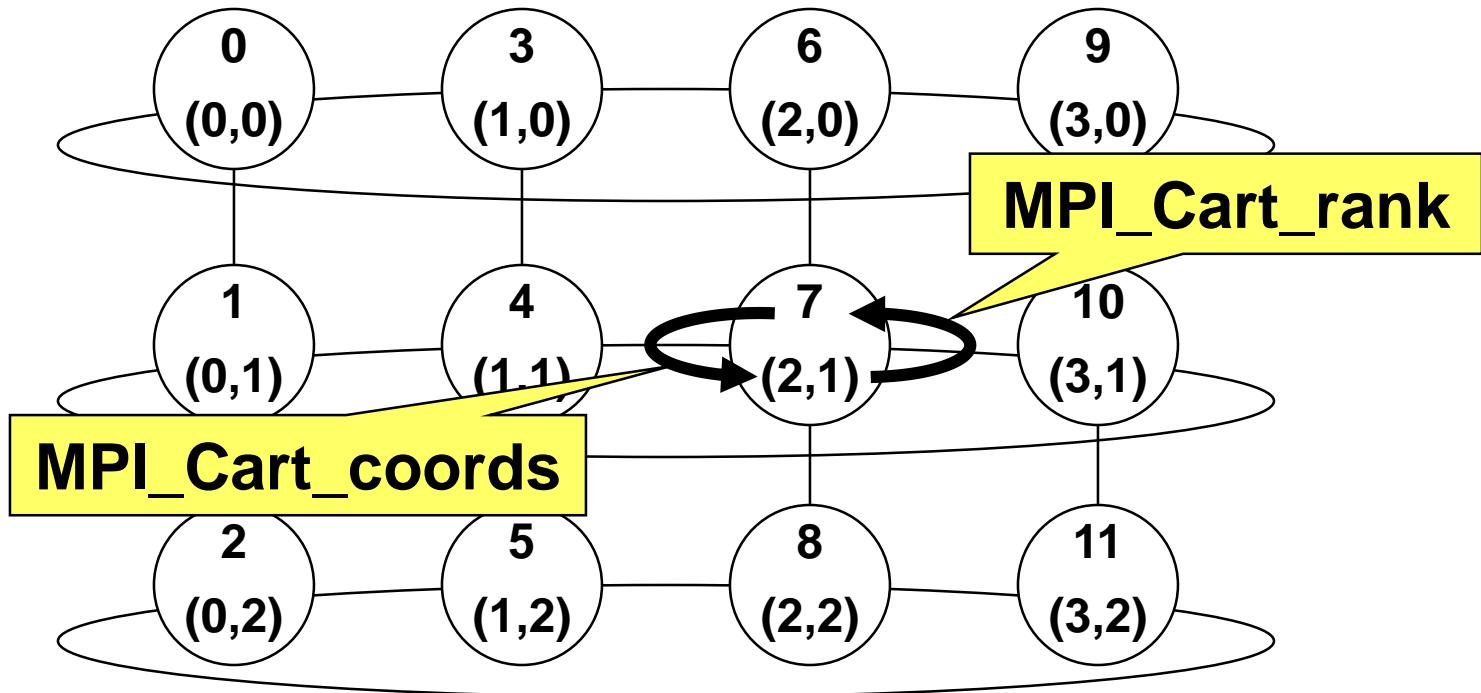
```
mpi & mpif.h:  INTEGER  comm_cart, coords(*), rank, ierror
```

- Python: `rank = comm_cart.Get_cart_rank(coords)`

see [mpi4py.MPI.Cartcomm — MPI for Python 3.1.1 documentation](#)

Python

Own coordinates



- Each process gets its own coordinates with (example in **Fortran**)
CALL MPI_Comm_rank(comm_cart, *my_rank*, *ierror*)
CALL MPI_Cart_coords(comm_cart, *my_rank*, maxdims, *my_coords*, *ierror*)

Typical usage of MPI_Cart_create & MPI_Dims_create

```
#define ndims 3
int i, nnodes, world_myrank, cart_myrank, dims[ndims], periods[ndims], my_coords[ndims]; MPI_Comm comm_cart;
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &world_myrank);
for (i=0; i<ndims; i++) { dims[i]=0; periods[i]=...; }
MPI_Dims_create(numprocs, ndims, dims); // computes factorization of numprocs
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods,1, &comm_cart);
MPI_Comm_rank(comm_cart, &cart_myrank);
MPI_Cart_coords(comm_cart, cart_myrank, ndims, my_coords, ierror)
```

From now, all communication should be based on
comm_cart & cart_myrank & my_coords

C

- C/C++: int MPI_Dims_create(int nnodes, int ndims, int **dims*)

- Fortran: MPI_DIMS_CREATE(nnodes, ndims, *dims*, *IERROR*)

mpi_f08: INTEGER :: nnodes, ndims, dims(*)
INTEGER, OPTIONAL :: ierror

mpi & mpif.h: INTEGER nnodes, ndims, dims(*), ierror

Array *dims* must be **initialized with zeros**
(other possibilities, see MPI standard)

- Python: *dims_out* = MPI.Compute_dims(nnodes, dims)

Fortran

Python

Exercise 1 — One-dimensional ring topology

- Use a one-dimensional virtual Cartesian topology in the pass-around-the-ring program:

Add a call to **MPI_Cart_create**, of course with reorder == non-zero or .TRUE.

e.g., 1

- Use **C** C/Ch9/cart-create-skel.c or **Fortran** F_30/Ch9/cart-create-skel_30.f90
or **Python** PY/Ch9/cart-create-skel.py
- **Caution:** Do only the prepared **one-dimensional virtual Cartesian topology**
- Hints:
 - After calling **MPI_Cart_create**,
 - there should be no further usage of **MPI_COMM_WORLD**, and
 - the **my_rank** must be recomputed on the base of **comm_cart**.
 - Only **one-dimensional**:
 - → coordinates are not necessary, because **coord==rank**

In this exercise not relevant, because the skeleton already uses arrays:

- → In C: **dims** and **period** as normal variables, i.e., no arrays, but call by reference with &dims, ...
- → In Fortran: **dims** and **period** must be arrays (i.e., with only 1 element, e.g., (/ .TRUE./))

Slide from Chap. 4 — Rotating information around a ring

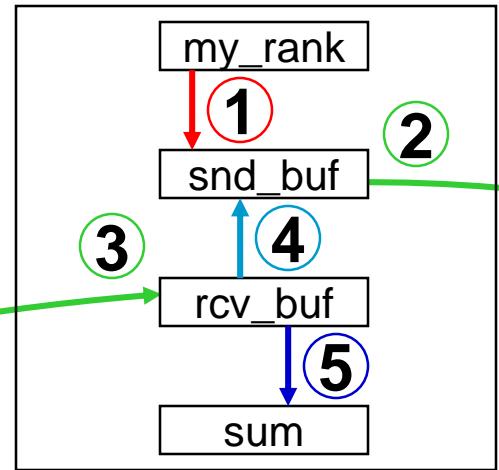
Initialization: 1

Each iteration:

2 3 4 5

(1) Communication through a new reordered Cartesian communicator

(2) my_rank based on this new communicator

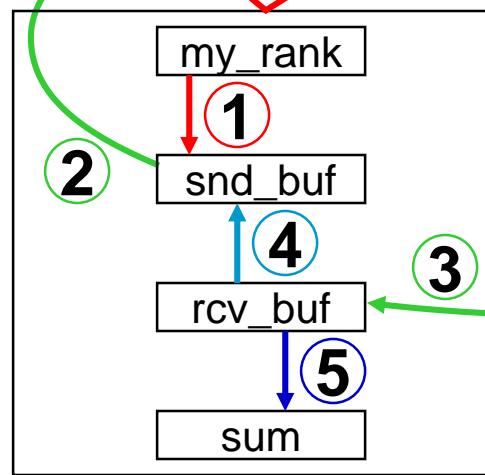


Fortran:

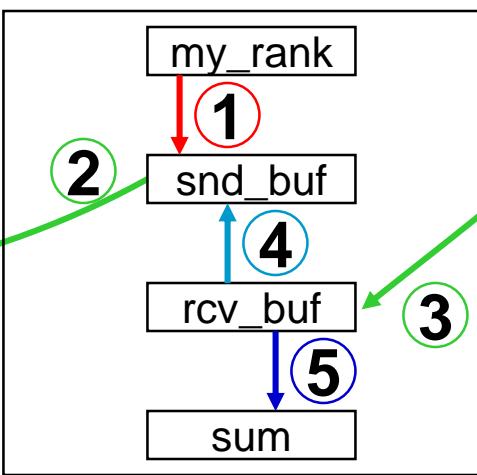
```
dest = mod(my_rank+1,size)  
source = mod(my_rank-1+size,size)
```

C/C++:

```
dest = (my_rank+1) % size;  
source = (my_rank-1+size) % size;
```



From
Chap.4 Nonblocking Communication



During the Exercise (20 min.)



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

And have fun with this a bit longer 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:

*It is a **longer exercise**, with about 3 lines completed (with correct types),
5 lines added, and 2 lines modified.*



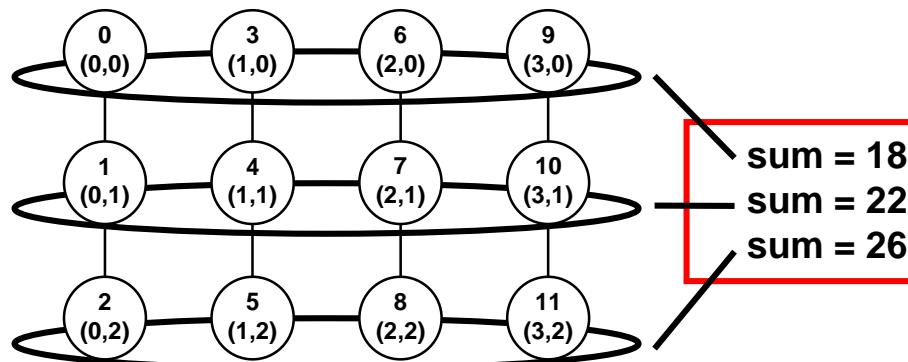
**Please, go already to your breakout room after finishing the source code
and before first compilation and mpirun.**

**You may be able to find help or to help your colleagues
if there are any problems.**



Advanced Exercise 1b — Two-dimensional topology

- Task: Rewrite the exercise in two dimensions, as a cylinder.
 - Each row of the cylinder, i.e. each ring, should compute its own separate sum of the original ranks in the two dimensional comm_cart.
 - Compute the two dimensional factorization with MPI_Dims_create().
 - Array *dims* must be **initialized** with **(0,0)** !
 - Execute the ring algorithm in direction 0, i.e., communicating only to its left and right neighbors.
 - Calculate the neighbor ranks `left` and `right` using MPI_Cart_rank().
- Use **C** C/Ch9/cylinder-skel.c or **Fortran** F_30/Ch9/cylinder-skel_30.f90 or **Python** PY/Ch9/cylinder-skel.py
- Run with mpirun -np 12 ./a.out | sed -e 's/PE//' | sort



Cartesian Mapping Functions

- Computing ranks of neighboring processes

C

- C/C++: `int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int disp,
int *rank_source, int *rank_dest)`

Fortran

- Fortran: `MPI_CART_SHIFT(comm_cart, direction, disp,
rank_source, rank_dest, ierror)`

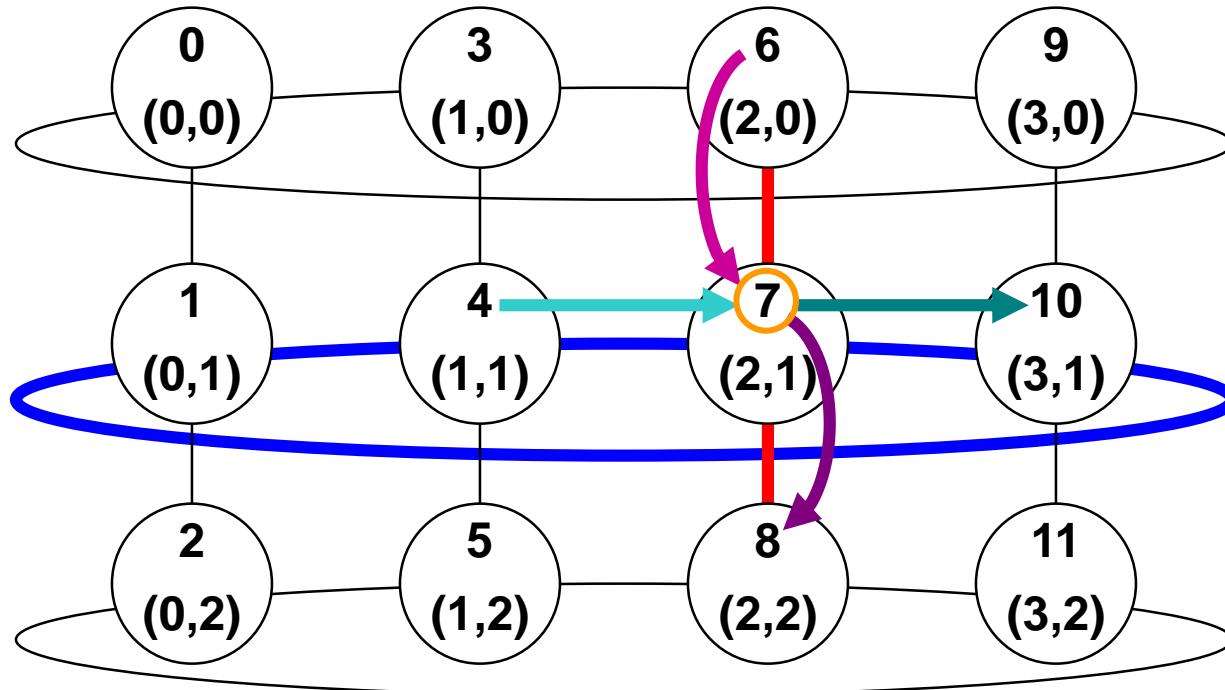
mpi_f08: `TYPE(MPI_Comm) :: comm_cart`
 `INTEGER :: direction, disp, rank_source, rank_dest`
 `INTEGER, OPTIONAL :: ierror`
mpi & mpif.h: `INTEGER comm_cart, direction, disp, rank_source, rank_dest, ierror`

- Python: `(rank_source, rank_dest) = comm_cart.Shift(direction, disp)`

Python

- Returns `MPI_PROC_NULL` if there is no neighbor.
- `MPI_PROC_NULL` can be used as source or destination rank in each communication → Then, this communication will be a no-operation!

MPI_Cart_shift – Example



○ ... invisible input argument: **my_rank** in `comm_cart`

CALL MPI_Cart_shift (`comm_cart`, direction, disp, `rank_source`, `rank_dest`, `ierror`)

example on

process rank=7

0 or	+1	4	10
1	+1	6	8

Cartesian Partitioning

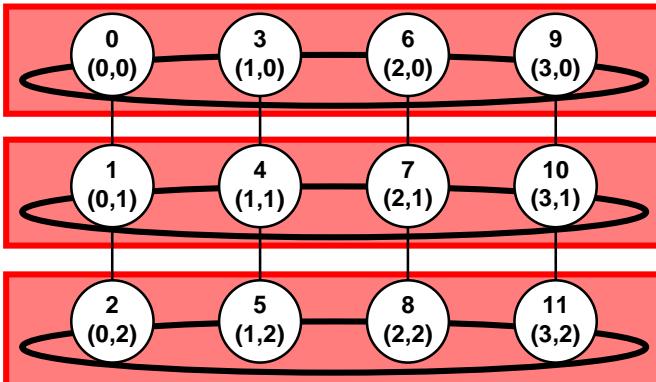
- Cut a virtual process grid up into *slices*.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.

c

- C/C++: `int MPI_Cart_sub(MPI_Comm comm_cart, int *remain_dims,
MPI_Comm *comm_slice)`

Fortran

- Fortran: `MPI_CART_SUB(comm_cart, remain_dims, comm_slice, ierror)`



mpi_f08: `TYPE(MPI_Comm) :: comm_cart`
 `LOGICAL :: remain_dims(*)`
 `TYPE(MPI_Comm) :: comm_slice`
 `INTEGER, OPTIONAL :: ierror`

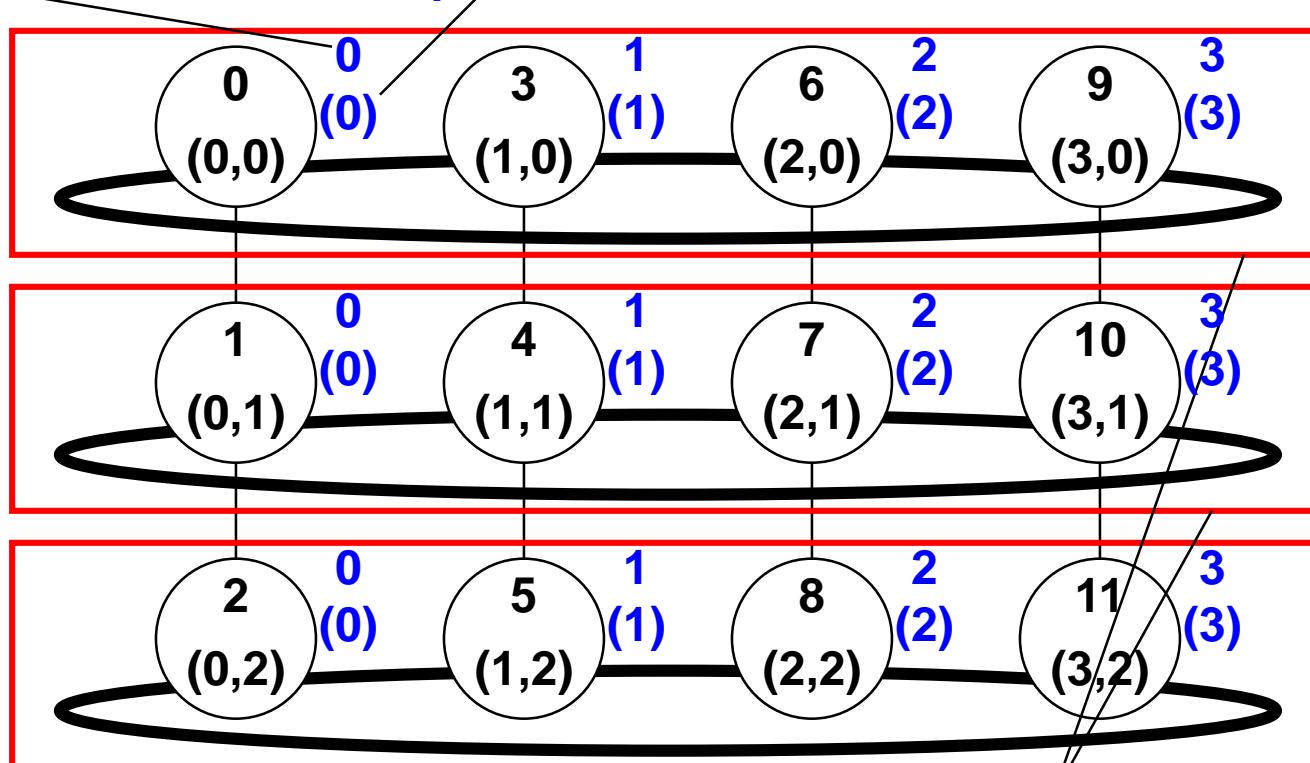
mpi & mpif.h: `INTEGER comm_cart, comm_slice, ierror`
`LOGICAL remain_dims(*)`

Python

- Python: `comm_slice = comm_cart.Sub(remain_dims)`

MPI_Cart_sub – Example

- Ranks and Cartesian process coordinates in **comm_slice**



- CALL MPI_Cart_sub(comm_cart, remain_dims, **comm_slice**, ierror)

(true, false)

Each process gets only
its own sub-communicator

Four slides with general remarks
before next exercise

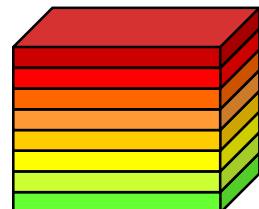
Multidimensional domain decomposition

- Applications with 3 dimensions

- each sub-domain (computed by one CPU) should
- have the same size → optimal load balance
- minimal surface → minimal communication
- Usually optimum with 3-dim. domain decomposition & cubic sub-domains

- Same rule for 2 dimensional application → 2-D domain decomposition & quadratic sub-domains

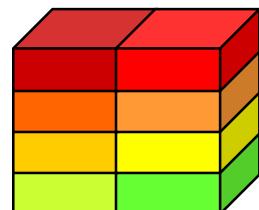
Exception: The total domain has extremely different dimensions, e.g., weather/climate:
40,000 km x 40,000 km x 15 km
(→ only 2-dim domain decomp.)



Splitting in

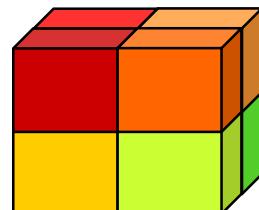
- **one dimension:**

communication
 $= n^2 * 2 * w * 1 / p^0$



- **two dimensions:**

communication
 $= n^2 * 2 * w * 2 / p^{1/2}$



- **three dimensions:**

communication
 $= n^2 * 2 * w * 3 / p^{2/3}$

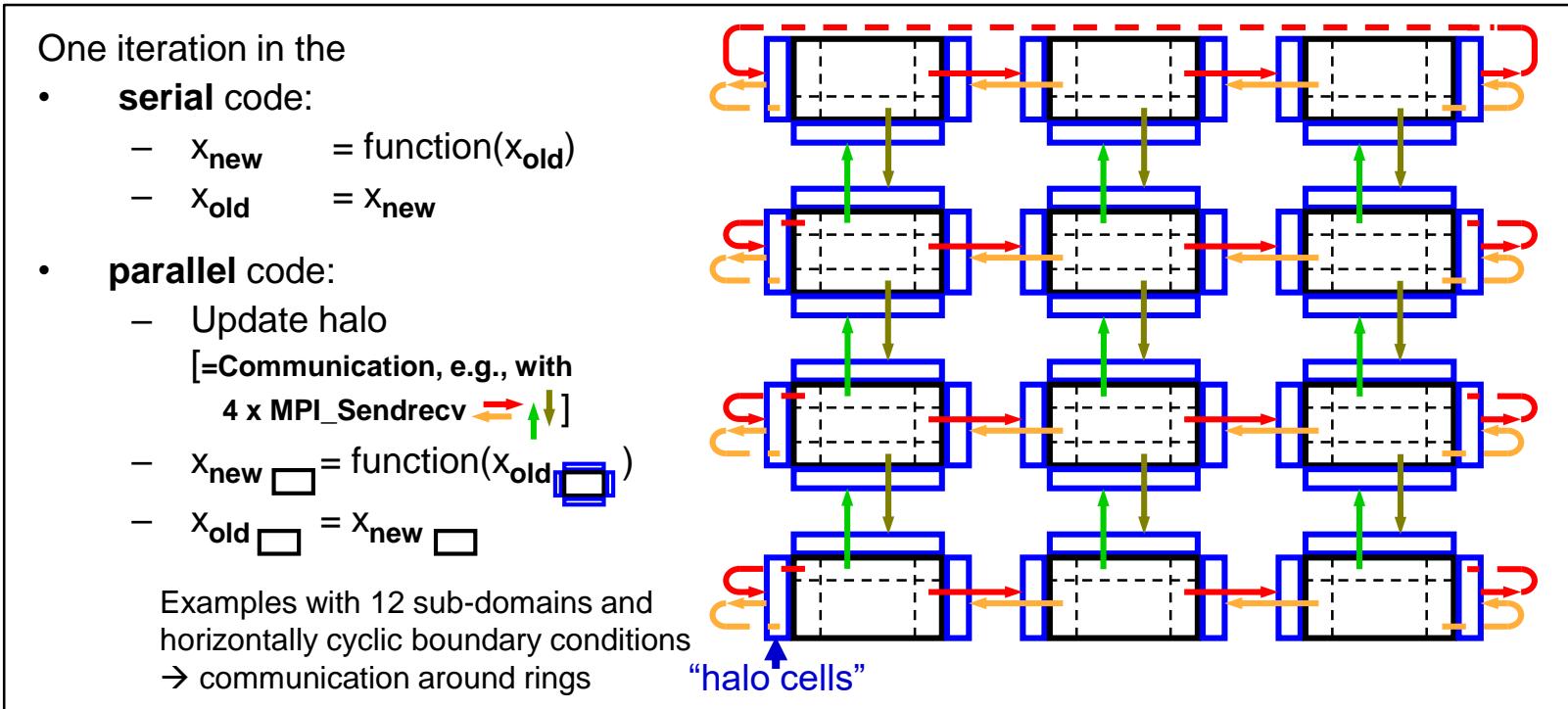
w = width of halo
 n^3 = size of matrix
 p = number of processes
cyclic boundary
—> **two** neighbors
in each direction

optimal for $p \geq 12$

General rule:

Symmetric vs. asymmetric manager/worker parallelization

- We know this example already from course chapter 1

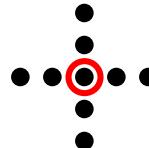


- General rule:
 - Always try to implement such a **symmetric parallelization design**
 - Avoid (asymmetric) manager-worker¹⁾-paradigm**
→ the manager always tends to **limit the scaling** to a larger number of processes

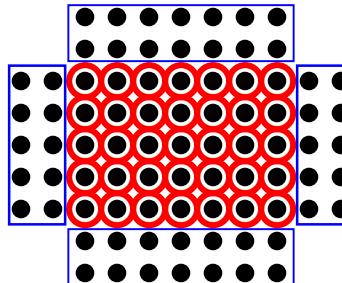
¹⁾ The outdated wording “master/slave” should be avoided

Halo

- Stencil:
 - To calculate a new data mesh point (○), old data from the stencil mesh points (●) are needed
 - E.g., 9 point stencil



- Halo
 - To calculate the new data mesh points of a sub-domain, additional mesh points from other sub-domains are needed.
 - They are stored in **halos** (ghost cells, shadows)
 - Halo depends on form of stencil



Back
to Ch.1
on next
slide

Diagonals Problem

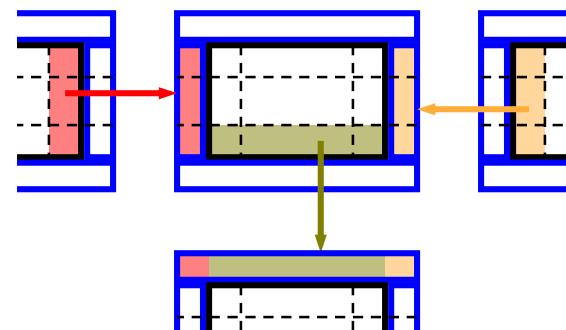
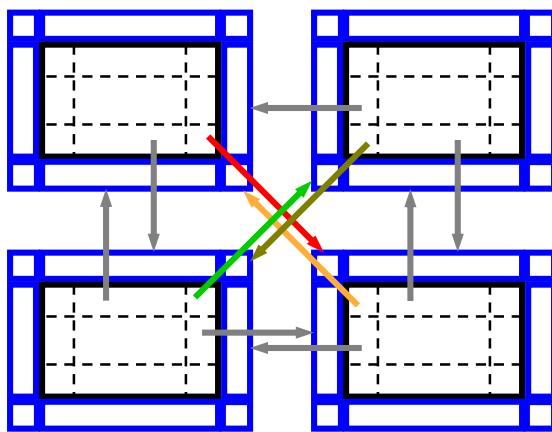
- Stencil with diagonal point, e.g.,



- i.e., halos include corners →→→

substitute small corner messages:

- one may use 2-phase-protocol:
- normal horizontal halo communication
- include corner into vertical exchange



Chris Ding and Yun He: A ghost cell expansion method for reducing communications in solving PDE problems.
Proc. SC2001. DOI:10.1145/582034.582084

Back
to Ch.1



Exercise 2 — One-dimensional ring topology

- Use a one-dimensional in the pass-around-the-ring program:
Add a call to **MPI_Cart_shift** to calculate left and right
- Use **C** C/Ch9/cart-shift-skel.c or **Fortran** F_30/Ch9/cart-shift-skel_30.f90
or **Python** PY/Ch9/cart-shift-skel.py
- Goal:
 - the cryptic way to compute the neighbor ranks should be substituted by one call to MPI_Cart_shift, that should be before starting the loop.

Slide from Chap. 4 — Rotating information around a ring

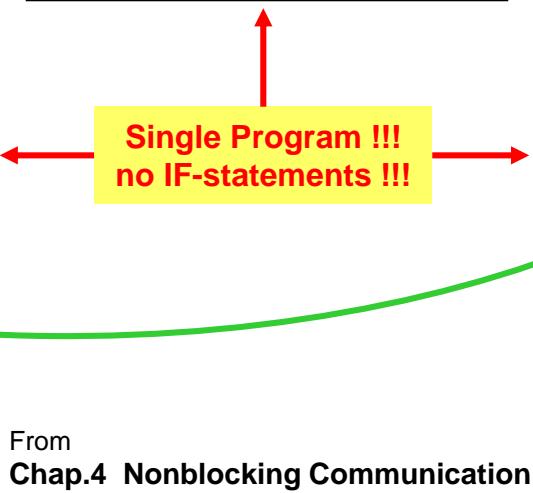
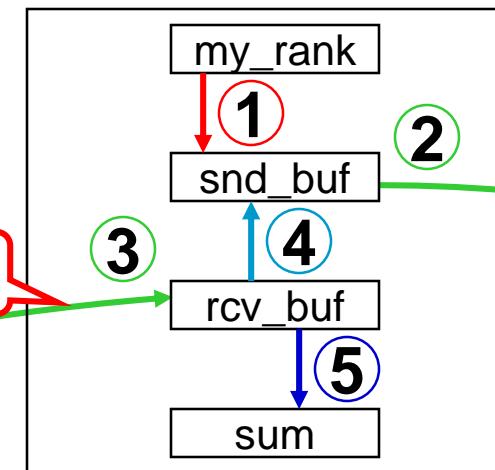
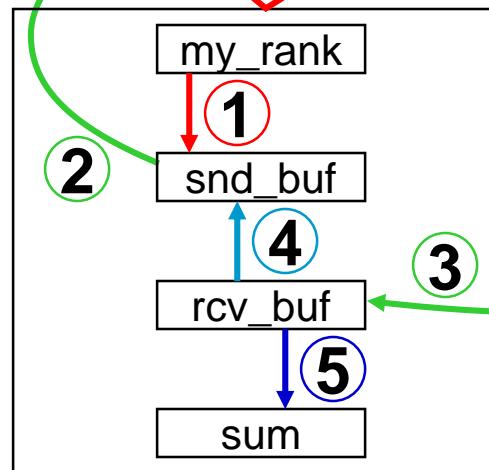
Initialization: 1

Each iteration:

2 3 4 5

Done in Exercise 1: (1) Communication through
a new reordered Cartesian communicator

Done in Exercise 1: (2) my_rank based
on this new communicator



Fortran:

```
dest = mod(my_rank+1,size)  
source = mod(my_rank-1+size,size)
```

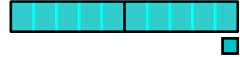
C/C++:

```
dest = (my_rank+1) % size;  
source = (my_rank-1+size) % size;
```

(3) To be substituted by
MPI_Cart_shift(... source, dest ...),
called only once,
before starting the loop



During the Exercise (10 min.)



Please stay here in the main room while you do this 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

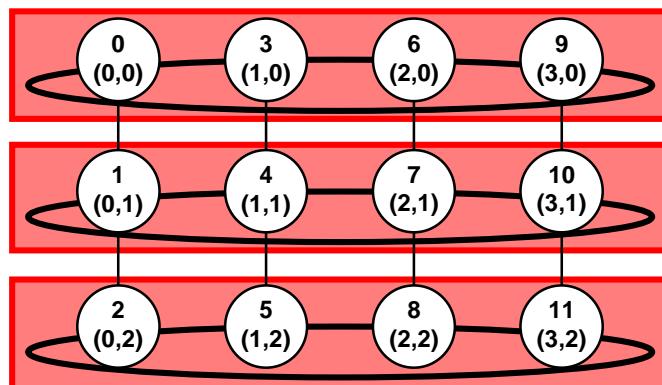
and continue your discussions with your fellow learners:

Any problems with this small exercise?



Exercise 3+4 (advanced) — Two-dimensional topology

- **Exercise 3:** Rewrite the exercise in two dimensions, as a cylinder.
 - Each row of the cylinder, i.e. each ring, should compute its own separate sum of the original ranks in the two dimensional comm_cart.
 - Task: substitute 2x MPI_Cart_rank by 1x MPI_Cart_shift
 - **Use** (your) solution of Ch.9-(1) Advanced exercise 1b:
 - Your modified **C, F_30, PY/Ch9/cylinder-skel.c, _30.f90, .py**
 - Or copy provided **C, F_30, PY/Ch9/solutions/cylinder.c, _30.f90, .py**
- **Exercise 4:** Use MPI_Cart_sub to create the one-dimensional slice communicators
 - Results are the same



sum = 18
sum = 22
sum = 26

Summing up the myrank of the 2-dimensional Cartesian topology:
Advanced Exercise 4a:
Ring-communication in the comm_slice, and using the ring with myrank, left, right and size of the comm_slice.

Additional Advanced Exercise 4b:
Using MPI_Allreduce within the comm_slice instead of the ring communication algorithm.
Solution, see 2nd Adv. Exe Chapter 6-(1)

Exercise 3+4 (advanced) — Two-dimensional ring topology

- In the solutions directories

C C/Ch9/solutions/ & **Fortran** F_30/Ch9/solutions/ & **Python** PY/Ch9/solutions/

- topology_advanced3_cylinder.c / _30.f90 / .py → 2-dim topology (Exa.3)
- topology_advanced4_cart_sub.c / _30.f90 / .py → using MPI_Cart_sub (Exa. 4a)
- And in directories

C C/Ch6/solutions/ & **Fortran** F_30/Ch6/solutions/ & **Python** PY/Ch6/solutions/

- cylinder_advanced2_subtopology.c / _30.f90 / .py
→ MPI_Cart_sub and MPI_allreduce (Exa. 4b)

Quiz on Chapter 9-(1) – Virtual topologies

A. What are the two major benefits from virtual topology process grids?

1. _____
2. _____

B. What must the application programmer do to enable these opportunities?

1. _____
2. _____
3. _____

C. Which MPI procedure can be used to create several non-overlapping subcommunicators?
(do not forget to also mention the routines from course chapter 8!)

And are their specific prerequisites?

1. _____
2. _____
3. _____

For private notes

For private notes

For private notes

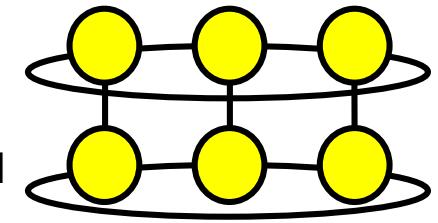
Chap.9 Virtual Topologies

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module mpi_f08
6. Collective communication
7. Error Handling
8. Groups & communicators, environment management

9. Virtual topologies

- (1) A multi-dimensional process naming scheme
- (2) Neighborhood communication + MPI_BOTTOM
- (3) Optimization through reordering

10. One-sided communication
11. Shared memory one-sided communication
12. Derived datatypes
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



= perfect scalable !?

Course Chap. 9-(2):

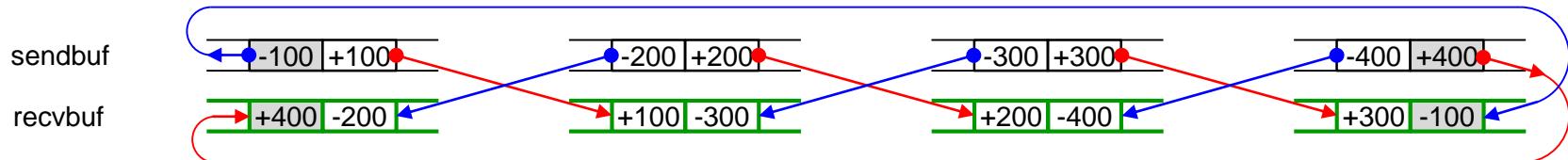
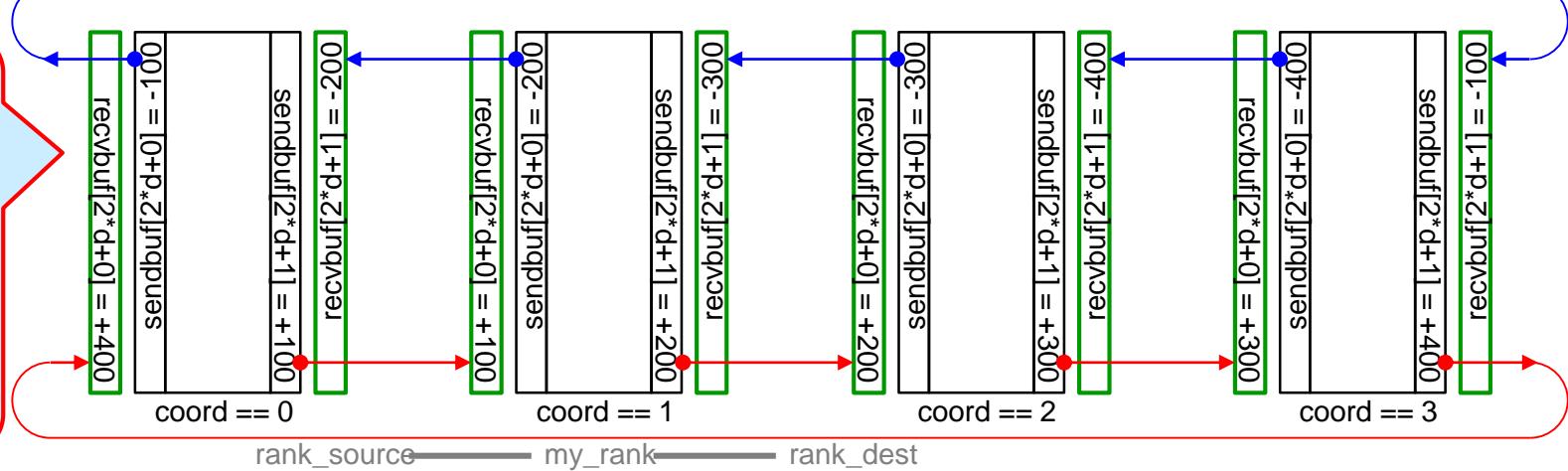
Sparse Collective Operations on Process Topologies

New in MPI-3.0

- MPI process topologies (Cartesian and (distributed) graph) usable for communication
 - MPI_(I)NEIGHBOR_ALLGATHER(V)
 - MPI_(I)NEIGHBOR_ALLTOALL(V,W)
- If the topology is the full graph, then neighbor routine is identical to full collective communication routine
 - Exception: s/rdispls in MPI_NEIGHBOR_ALLTOALLW are MPI_Aint
- Allows for optimized communication scheduling and scalable resource binding
- Cartesian topology:
 - Sequence of buffer segments is communicated with:
 - **direction=0 source, direction=0 dest, direction=1 source, direction=1 dest, ...**
 - Defined only for disp=1 (direction, source, dest and disp are defined as in MPI_CART_SHIFT)
 - If a source or dest rank is MPI_PROC_NULL then the buffer location is still there but the content is not touched.
 - See exercise 5 and advanced exercise 6

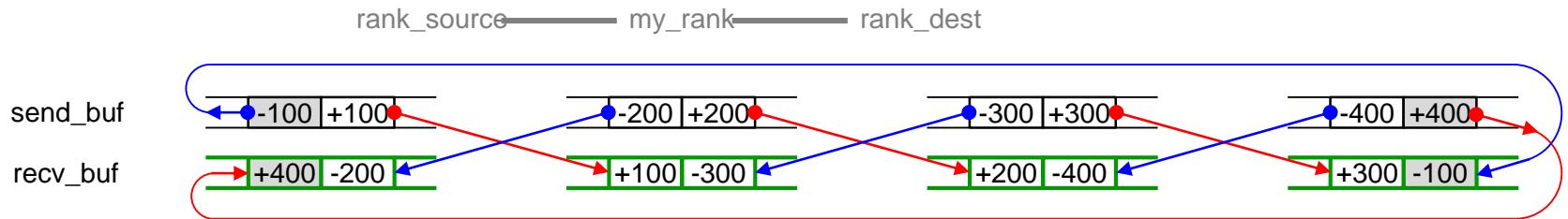
Periodic MPI_NEIGHBOR_ALLTOALL in direction d with 4 processes

This figure represents one direction d . Of course, it is valid for any direction



... grey array entries are used only if periods[d] == non-zero in C or .TRUE. in Fortran

As if ...

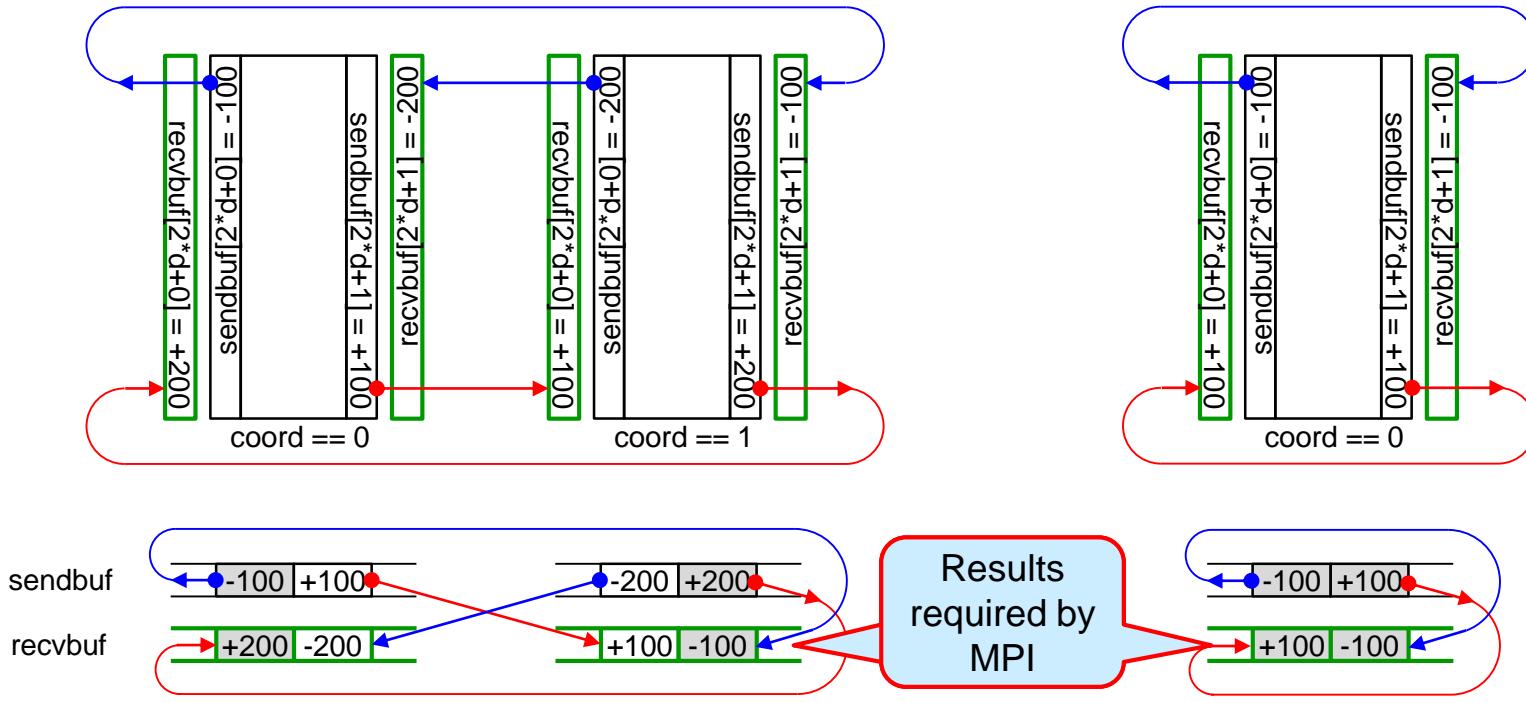


After MPI_NEIGHBOR_ALLTOALL on a Cartesian communicator returned, the content of the `recvbuf` is **as if** the following code is executed:

```
MPI_Cartdim_get(comm, &ndims);
for( /*direction*/ d = 0; d < ndims; d++) {
    MPI_Cart_shift(comm, /*direction*/ d, /*disp*/ 1, &rank_source, &rank_dest);
    MPI_Sendrecv(sendbuf[d*2+0], sendcount, sendtype, rank_source, /*sendtag*/ d*2,
                recvbuf[d*2+1], recvcount, recvtype, rank_dest, /*recvtag*/ d*2,
                comm, &status); /* 1st communication in direction of displacement -1 */
    MPI_Sendrecv(sendbuf[d*2+1], sendcount, sendtype, rank_dest, /*sendtag*/ d*2+1,
                recvbuf[d*2+0], recvcount, recvtype, rank_source, /*recvtag*/ d*2+1,
                comm, &status); /* 2nd communication in direction of displacement +1 */
}
```

The tags are chosen to guarantee that both communications (i.e., in negative and positive direction) cannot be mixed up, even if the MPI_SENDRECV is substituted by nonblocking communication and the MPI_ISEND and MPI_RECV calls are started in any sequence.

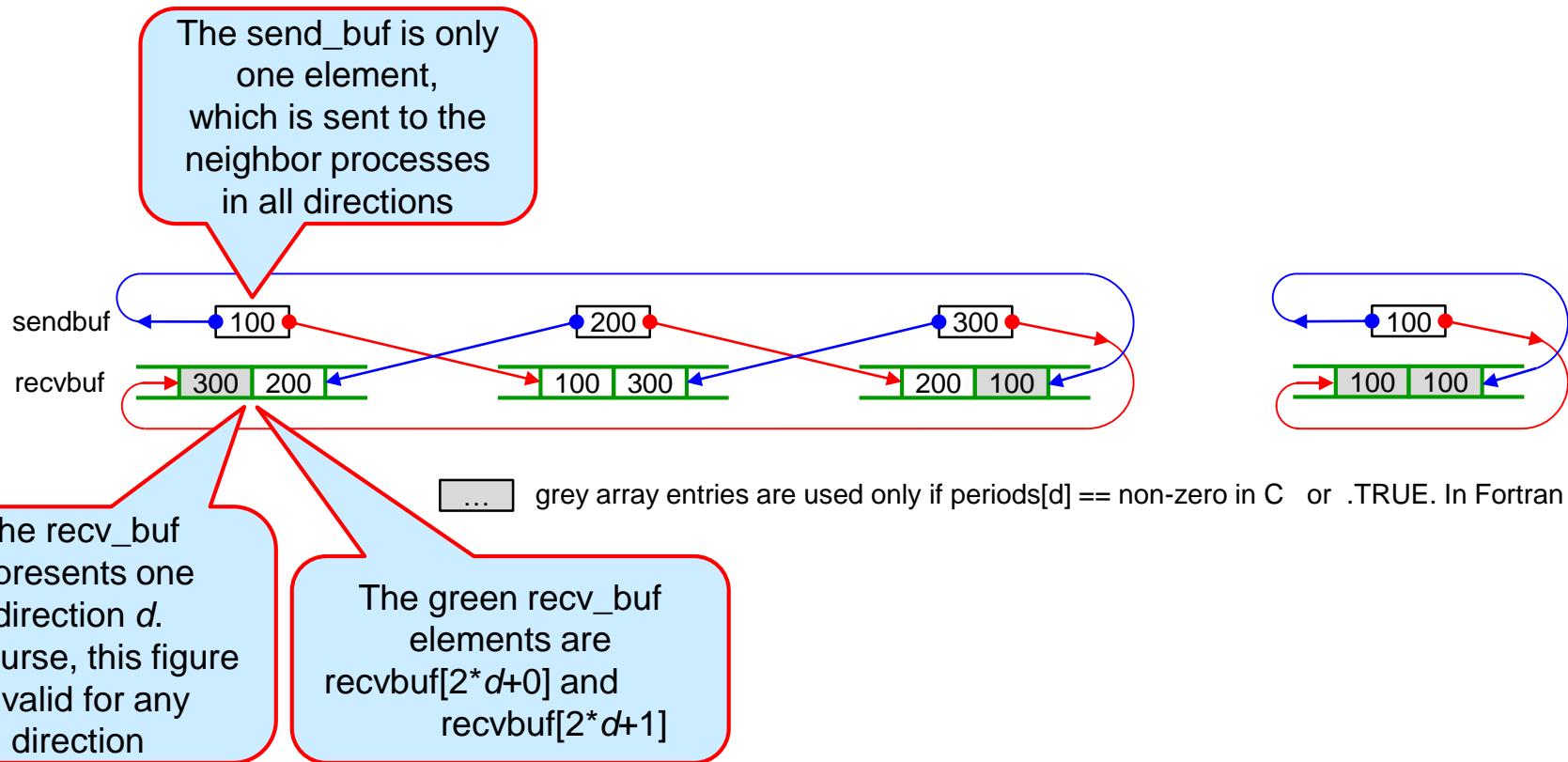
Wrong implementations of periodic MPI_NEIGHBOR_ALLTOALL with only 2 and 1 processes



Wrong results with **openmpi/4.0.1-gnu-8.3.0** and **cray-mpich/7.7.6** with 2 and 1 processes:

`recvbuf` **WRONG**

Communication pattern of MPI_NEIGHBOR_ALLGATHER



Other MPI features: MPI_BOTTOM and absolute addresses

- MPI_BOTTOM in point-to-point and collective communication:
 - Buffer argument is MPI_BOTTOM
 - Then absolute addresses can be used in
 - **Communication routines with byte displacement arguments, e.g., MPI_(I)NEIGHBOR_ALLTOALLW**
 - **Derived datatypes with byte displacements**
 - Displacements must be retrieved with MPI_GET_ADDRESS()
 - MPI_BOTTOM is an address,
i.e., **cannot be assigned to a Fortran variable!**
 - MPI-3.1/-4.0, Section 2.5.4, page 15 line 45 – page 16 line 6 / page 21 lines 14-23 shows all such address constants
that cannot be used in expressions or assignments **in Fortran**, e.g.,
 - **MPI_STATUS_IGNORE** (→ point-to-point comm.)
 - **MPI_IN_PLACE** (→ collective comm.)
 - Fortran: Using MPI_BOTTOM & absolute displacement of variable X
→ **<type>, ASYNCHRONOUS :: X** and **MPI_F_SYNC_REG(X)** is needed:
 - **MPI_BOTTOM** in a blocking MPI routine → **MPI_F_SYNC_REG** before and after this routine
 - in a nonblocking routine → **MPI_F_SYNC_REG** before this routine & after final WAIT/TEST

Fortran

} Further information,
see course
Chapter 12
**Derived
Datatypes**
→
MPI_GET_ADDRESS
MPI_AINT_ADD
MPI_AINT_DIFF

Exercise 5 — Neighbor Collective Communication

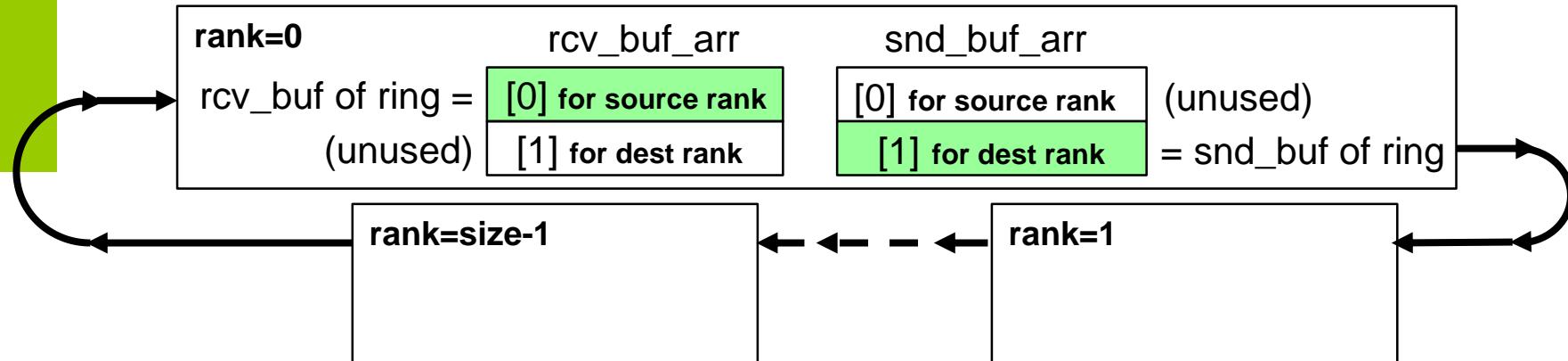
In MPI/tasks/...

Use **C** C/Ch9/ring_neighbor_alltoall_skel.c
 or **Fortran** F_30/Ch9/ring_neighbor_alltoall_skel_30.f90
 or **Python** PY/Ch9/ring_neighbor_alltoall_skel.py

In this example, we ignore the communication in the other direction.
 Of course in real applications, both communications (to the left and to the right) are used.

Keep the ring communication in the virtual topology example, but substitute the point-to-point communication by neighborhood collective:

- I.e., Isend-Recv-Wait → one call to MPI_Neighbor_alltoall
- rcv_buf and snd_buf must be extended to a rcv_buf_arr and snd_buf_arr with **rcv_buf_arr[0]** as rcv_buf and **snd_buf_arr[1]** as snd_buf, i.e., according to the sequence rule for the buffer segments.
- snd_count and recv_count are both 1 (not 2!), describing one buffer, not the array of buffers (i.e., one message)!



Exercise 6 (advanced) —

Neighbor Collective Communication & MPI_BOTTOM

Use **C** C/Ch9/ring_neighbor_alltoallw_skel.c

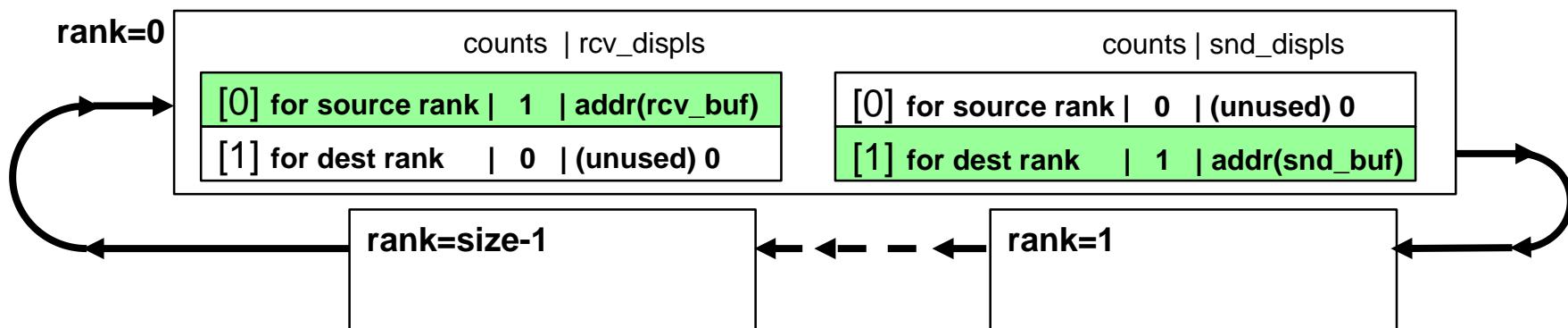
or **Fortran** F_30/Ch9/ring_neighbor_alltoallw_skel_30.f90

or **Python** PY/Ch9/ring_neighbor_alltoallw_skel.py

mpi4py may require `mem_from_bottom = MPI.memory.fromaddress(MPI.BOTTOM,0,0)` and passing `mem_from_bottom` instead of `MPI.BOTTOM`, e.g., in `(mem_from_bottom, snd_counts, snd_displs, snd_types)` as send buffer in `comm.Neighbor_alltoallw(...)`

You start again from the virtual topology example, but substitute the point-to-point communication by **MPI_NEIGHBOR_ALLTOALLW** with **MPI_BOTTOM** and absolute addresses of `rcv_buf` and `snd_buf`:

- I.e., Isend-Recv-Wait → one call to `MPI_Neighbor_alltoallw`
- Fortran: Do not forget to call `MPI_F_SYNC_REG` for the real variables behind `MPI_BOTTOM` (i.e., `snd_buf`, `rcv_buf`) **before & after** the communication call!



CAUTION: Officially, this example is not portable, because address differences are allowed only inside of structures or arrays, i.e., `snd_buf` and `rcv_buf` need to be part of a common space → MPI-3.1, 4.1.12

During the Exercise (20 min.)



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



And have fun with this a bit longer 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 check what happens, if you use only one or two processes.
If your software works with 3, 4, 5, ... processes,
then the problem is based on a **bug in nearly all MPI libraries** detected 2019.



Who of you could see this bug?

Do you have an idea how to proceed with this problem? Please discuss.



For private notes

For private notes

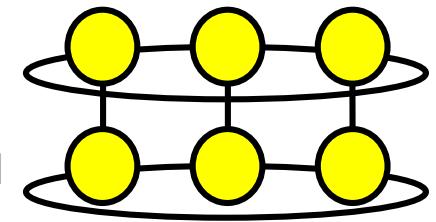
Chap.9 Virtual Topologies

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module mpi_f08
6. Collective communication
7. Error Handling
8. Groups & communicators, environment management

9. Virtual topologies

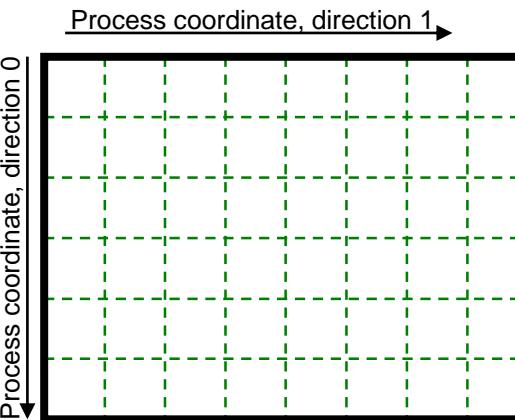
- (1) A multi-dimensional process naming scheme
- (2) Neighborhood communication + MPI_BOTTOM
- (3) Optimization through reordering

10. One-sided communication
11. Shared memory one-sided communication
12. Derived datatypes
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

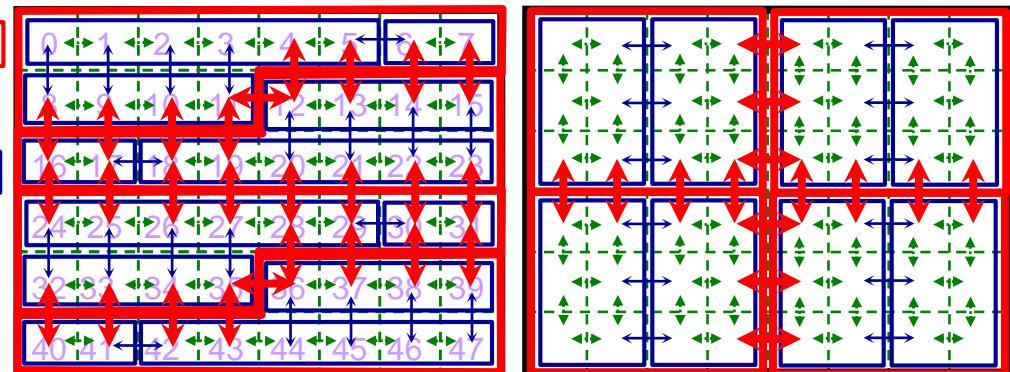


Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

- Example:
 - 2-dim 6000 x 8080 data mesh points
 - To be parallelized on 48 cores
- Minimal communication
 - Subdomains as quadratic as possible
 - minimal circumference
 - minimal halo communication
 - virtual 2-dim process grid: 6 x 8 with 1000 x 1010 mesh points/core



- Hardware example: 48 cores:
 - 4 ccNUMA nodes
 - each node with 2 CPUs
 - each CPU with 6 cores
- How to locate the MPI processes on the hardware?
 - Using sequential ranks in `MPI_COMM_WORLD`
 - Optimized placement
 - See next slides and example code



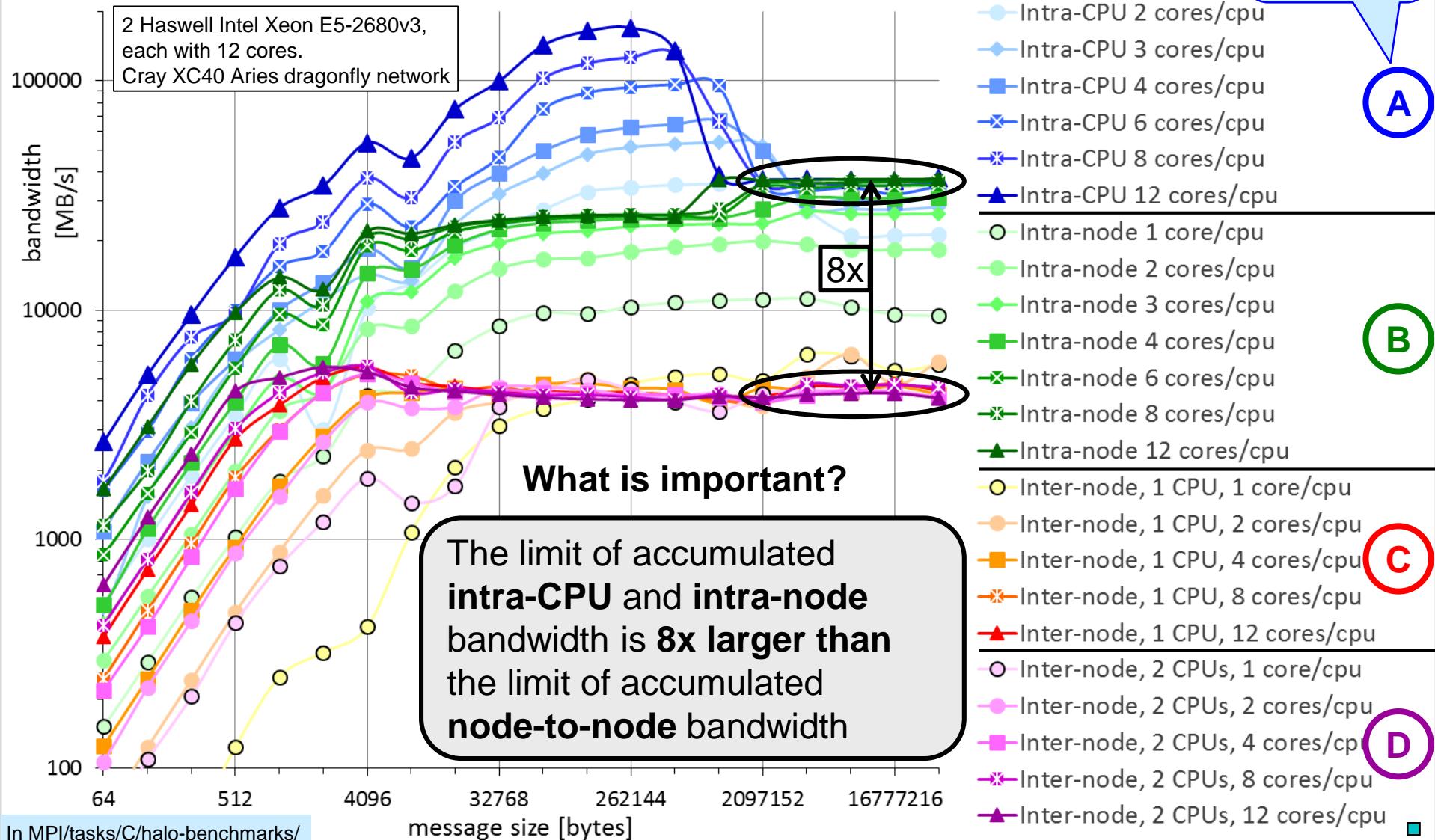
Non-optimal communications:
↔ 26 node-to-node (outer)
↔ 20 CPU-to-CPU (middle)
↔ 36 core-to-core (inner)

Optimized placement:
↔ Only 14 node-to-node
↔ Only 12 CPU-to-CPU
↔ 56 core-to-core

Duplex accumulated ring bandwidth per node

Further details
on the
benchmarks,
see next slide

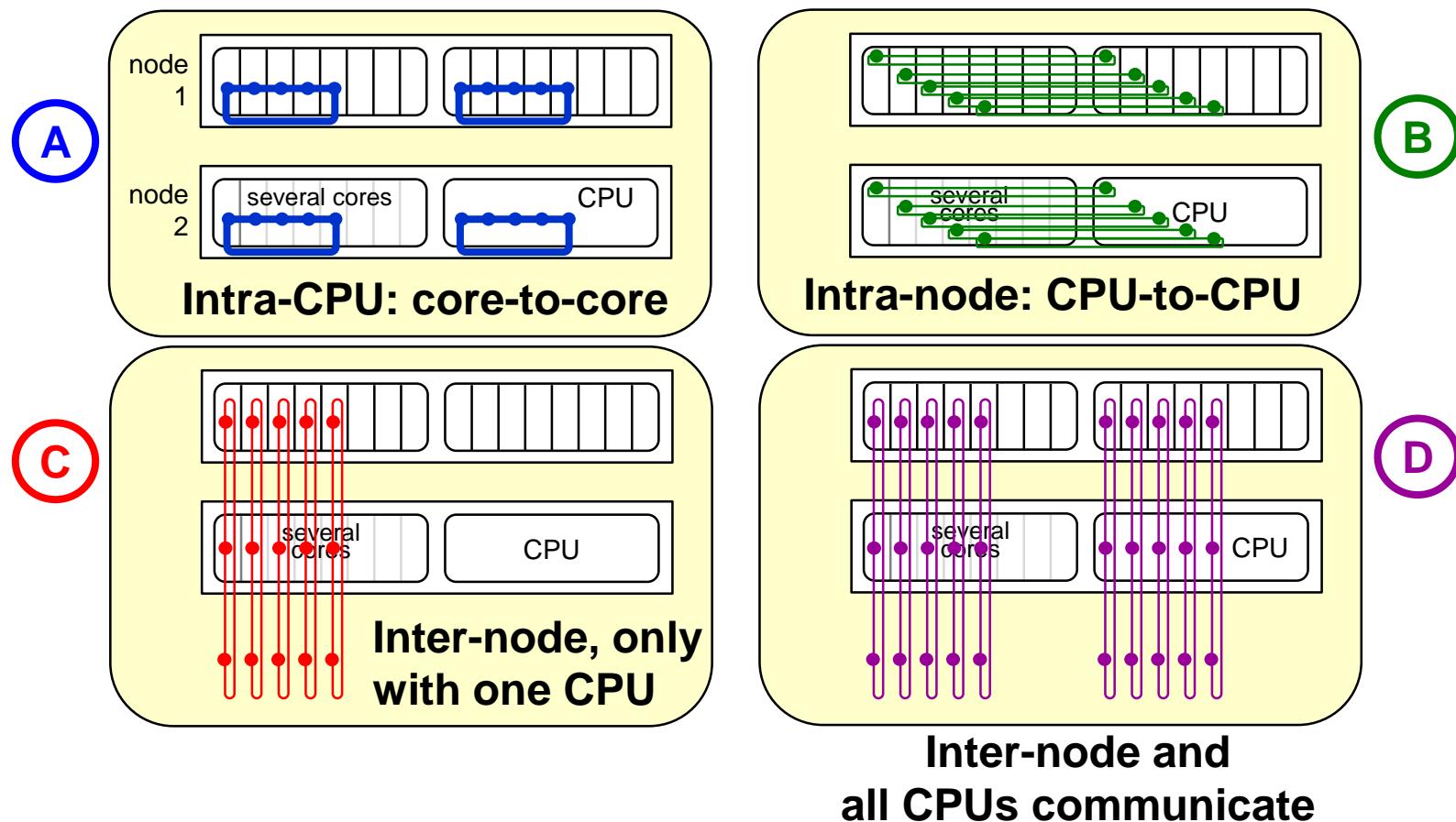
(each message is counted twice, as outgoing and incoming)



Multiple communicating rings

Benchmark MPI/tasks/C/halo-benchmarks/**halo_irecv_send_multiplelinks_toggle.c**

- Varying message size,
- number of **communication cores per CPU**, and
- four communication schemes (example with 5 **communicating cores per CPU**)



The problems

1. All MPI libraries provide the necessary interfaces 😊 😊 😊,
but **without** re-numbering in nearly all MPI-libraries 😞 😞 😞
 - You may substitute **MPI_Cart_create()** by Bill Grop's solution

William D. Gropp, Using Node [and Socket] Information to Implement MPI Cartesian Topologies, Parallel Computing, 2019, and
in: Proceedings of the 25th European MPI User' Group Meeting, EuroMPI'18, ACM, New York, NY, USA, 2018, pp. 18:1-18:9.
doi:10.1145/3236367.3236377. Slides: <http://wgropp.cs.illinois.edu/bib/talks/tdata/2018/nodecart-final.pdf>.

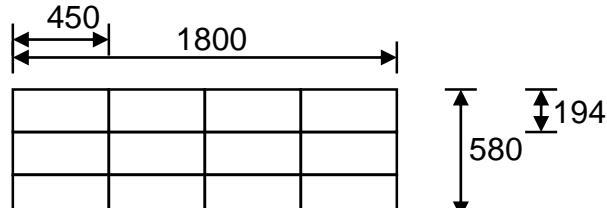
2. The existing MPI-3.1 and MPI-4.0 interfaces are not optimal
 - for cluster of ccNUMA node hardware,
 - We substitute `MPI_Dims_create() + MPI_Cart_create()`
by `MPIX_Cart_weighted_create(... MPIX_WEIGHTS_EQUAL ...)`
 - nor for application specific data mesh sizes
or direction-dependent bandwidth
 - by `MPIX_Cart_weighted_create(... weights)`
3. Caution: The application must be prepared for rank re-numbering
 - All communication through the newly created
Cartesian communicator with re-numbered ranks!
 - One must not load data based on `MPI_COMM_WORLD` ranks!

Examples

- Application topology awareness

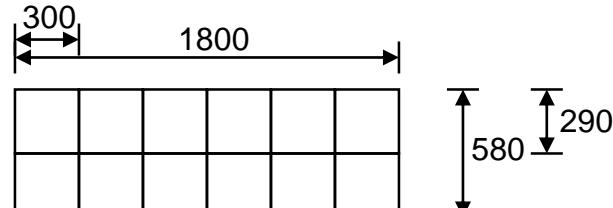
- 2-D example with 12 MPI processes and data mesh size 1800x580

- MPI_Dims_create → 4x3**



$$\text{Boundary of a subdomain} = 2(450+194) = 1288 \text{ ☹}$$

- data mesh aware → 6x2 processes**

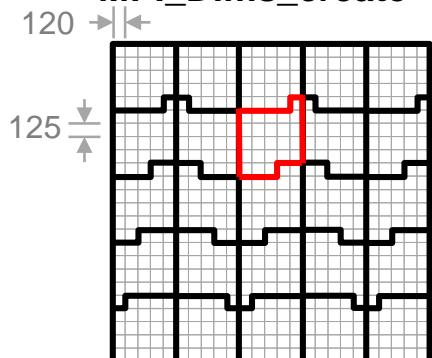


$$\text{Boundary of a subdomain} = 2(300+290) = 1180 \text{ ☺}$$

- Hardware topology awareness

- 2-D example with 25 nodes x 24 cores and data mesh size 3000x3000

- MPI_Dims_create → 25 x 24**

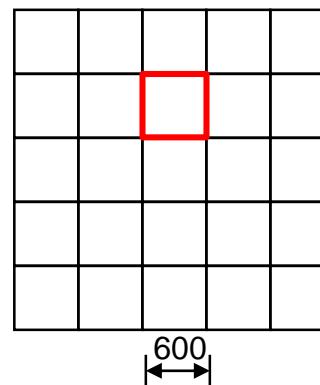


Accumulated communication per node

$$O(10 \times 120 + 12 \times 125) = O(2700) \text{ ☹}$$

- Hardware aware**

→ (5 nodes x 6 cores) X (5 nodes x 4 cores)



Accumulated communication per node

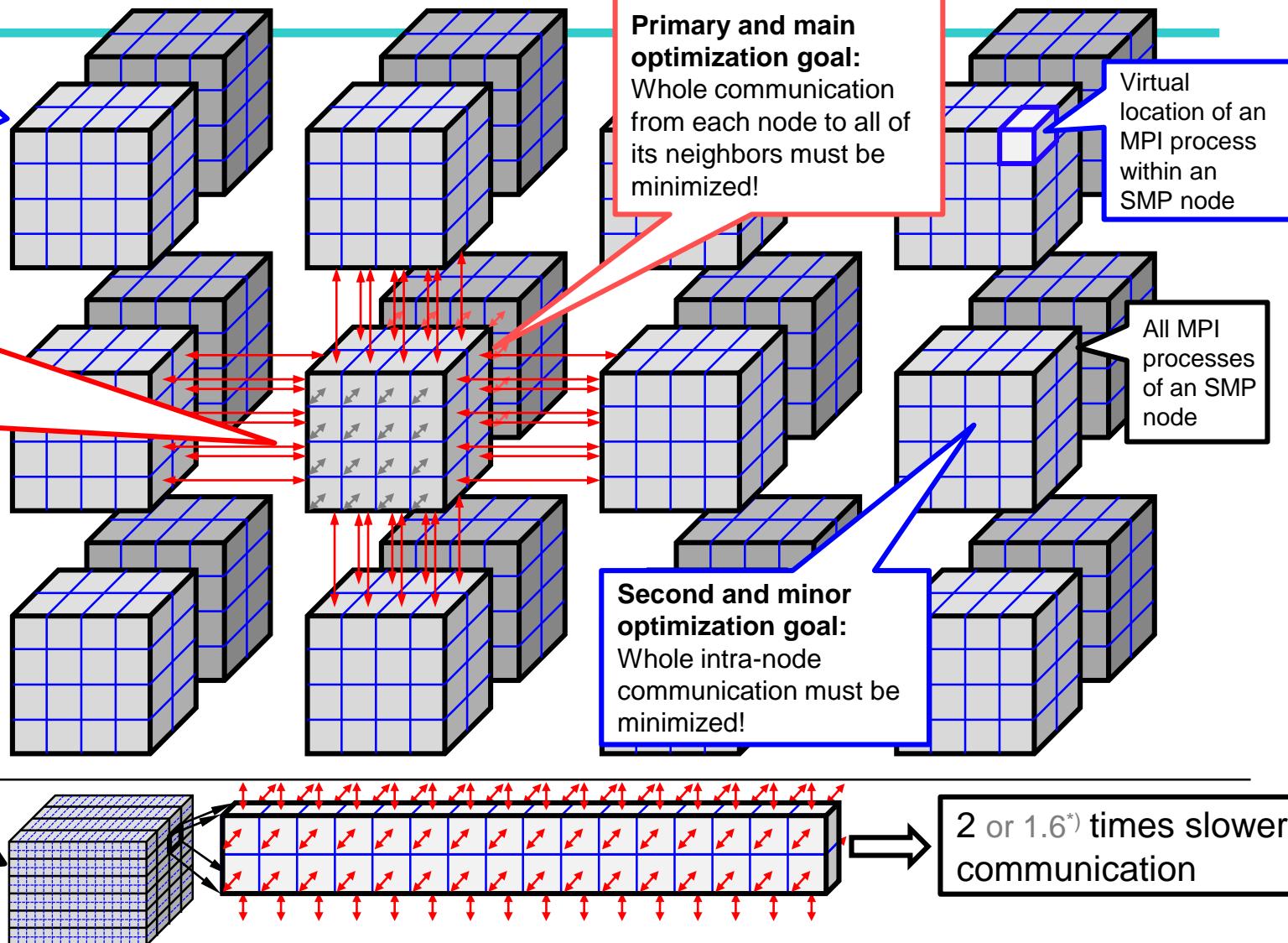
$$O(4 \times 600) = O(2400) \text{ ☺}$$

Hierarchical Cartesian Domain Decomposition

Example:
24 SMP nodes
 \times
32 cores/node

Per node:
maximal
 $8+8+8+8+16+16^{*}=$
48 or 64^{*}
connections
to neighbor
nodes
^{*} with cyclic communication

Without
topology-
optimization:
96 connections
to other nodes



Goals of Cartesian MPI_CartDims_create

- Given: comm_old (e.g., MPI_COMM_WORLD), ndims (e.g., 3 dimensions)
- Provide
 - a **factorization** of #processes (of comm_old) into the dimensions $\text{dims}[i]_{i=1..ndims}$
 - a Cartesian communicator **comm_cart**
 - a **optimized reordering** of the ranks in comm_old into the ranks of comm_cart to minimize the Cartesian communication time, e.g., of
 - MPI_Neighbor_alltoall
 - Equivalent communication pattern implemented with
 - MPI_Sendrecv
 - Nonblocking MPI point-to-point communication

The limits of MPI_Dims_create + MPI_Cart_create

- Not application topology aware
 - MPI_Dims_create can **only map evenly balanced** Cartesian topologies
 - Factorization of 48,000 processes into $20 \times 40 \times 60$ processes (e.g. for a mesh with $200 \times 400 \times 600$ mesh points)
→ no chance with current interface
- Only partially hardware topology aware
 - MPI_Dims_create has no communicator argument → not hardware aware
 - An application mesh with 3000×3000 mesh points on 25 nodes $\times 24$ cores ($=600$ MPI processes)
 - Answer from MPI_Dims_create:
 - » 25×24 MPI processes
 - » Mapped by most libraries to 25×1 nodes with 120×3000 mesh points per node
→ too much node-to-node communication

Major problems:

- No weights, no info
- Two separated interfaces for two common tasks:
 - Factorization of #processes
 - Mapping of the processes to the hardware

Goals of Cartesian MPI_CartDims_create

- Remark: On a hierarchical hardware,
 - **optimized factorization and reordering** typically means **minimal node-to-node** communication,
 - which typically means that the communicating surfaces of the data on each node is as quadratic as possible (or the subdomain as cubic as possible)
- The current API, i.e.,
 - due to the missing weights
 - and the non-hardware aware MPI_Dims_create, does **not** allow such an optimized factorization and reordering in many cases.

The new interface – proposed for MPI-4.1

- **MPI_Dims_create_weighted** (

```
/*IN*/      int      nnodes,  
/*IN*/      int      ndims,  
/*IN*/      int      dim_weights[ndims],  
/*IN*/      int      periods[ndims], /* for future use in  
                                         combination with info */  
/*IN*/      MPI_Info  info, /* for future use, currently MPI_INFO_NULL */  
*INOUT*  int      dims[ndims]);
```

input for application-topology-awareness

- Arguments have same meaning as in **MPI_Dims_create**

- Goal (in absence of an info argument):

- $\text{dims}[i] \cdot \text{dim_weights}[i]$ should be as close as possible,
- i.e., the $\sum_{i=0..(\text{ndims}-1)} \text{dims}[i] \cdot \text{dim_weights}[i]$ as small as possible
(advice to implementors)

A new courtesy function:
Weighted factorization

The new interface – proposed for MPI-4.1, continued

- **MPI_Cart_create_weighted** (

```
/*IN*/      MPI_Comm    comm_old,  
/*IN*/      int         ndims,  
/*IN*/      int         dim_weights[ndims], /*or MPI_UNWEIGHTED*/  
/*IN*/      int         periods[ndims],  
/*IN*/      MPI_Info    info,      /* for future use, currently MPI_INFO_NULL */  
/*INOUT*/     int         dims[ndims],  
/*OUT*/      MPI_Comm   *comm_cart );
```

input for hardware-awareness

input for application-topology-awareness

The new application & hardware topology aware interface

- Arguments have same meaning as in **MPI_Dims_create** & **MPI_Cart_create**
- See next slide for meaning of **dim_weights[ndims]**
- Goal: chooses
 - an **ndims**-dimensional factorization of #processes of **comm_old** (\rightarrow **dims**)
 - and an appropriate reordering of the ranks (\rightarrow **comm_cart**),

such that the execution time of a communication step along the virtual process grid
(e.g., with **MPI_NEIGHBOR_ALLTOALL** or equivalent calls to **MPI_SENDRECV**
as in the example in previous course Chapter 9.2)
is as small as possible.

How to specify the dim_weights?

- Given: comm_old (e.g., MPI_COMM_WORLD), ndims (e.g., 3 dimensions)
- This means, **the domain decomposition has not yet taken place!**
- Goals for dim_weights and the API at all:
 - Easy to understand
 - Easy to calculate
 - Relevant for typical Cartesian communication patterns (MPI_Neighbor_alltoall or alternatives)
 - Rules fit to usual design criteria of MPI
 - E.g., reusing MPI_UNWEIGHTED → integer array
 - Can be enhanced by vendors for their platforms → additional info argument for further specification
 - To provide also the less optimal two stage interface (in addition to the combined routine)

The dim_weights[i], example with 3 dimensions

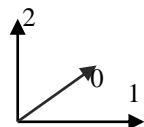
Abbreviations:

$d_i = \text{dims}[i]$

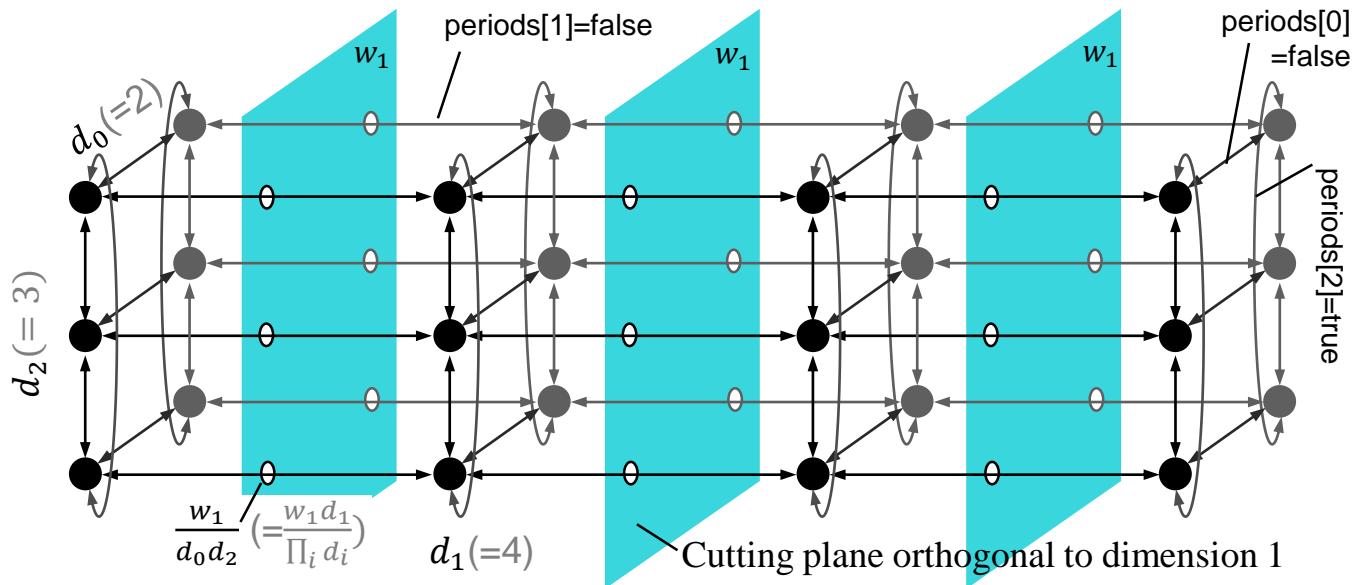
$w_i = \text{dim_weights}[i]$

with

$i = 0..(\text{ndims}-1)$

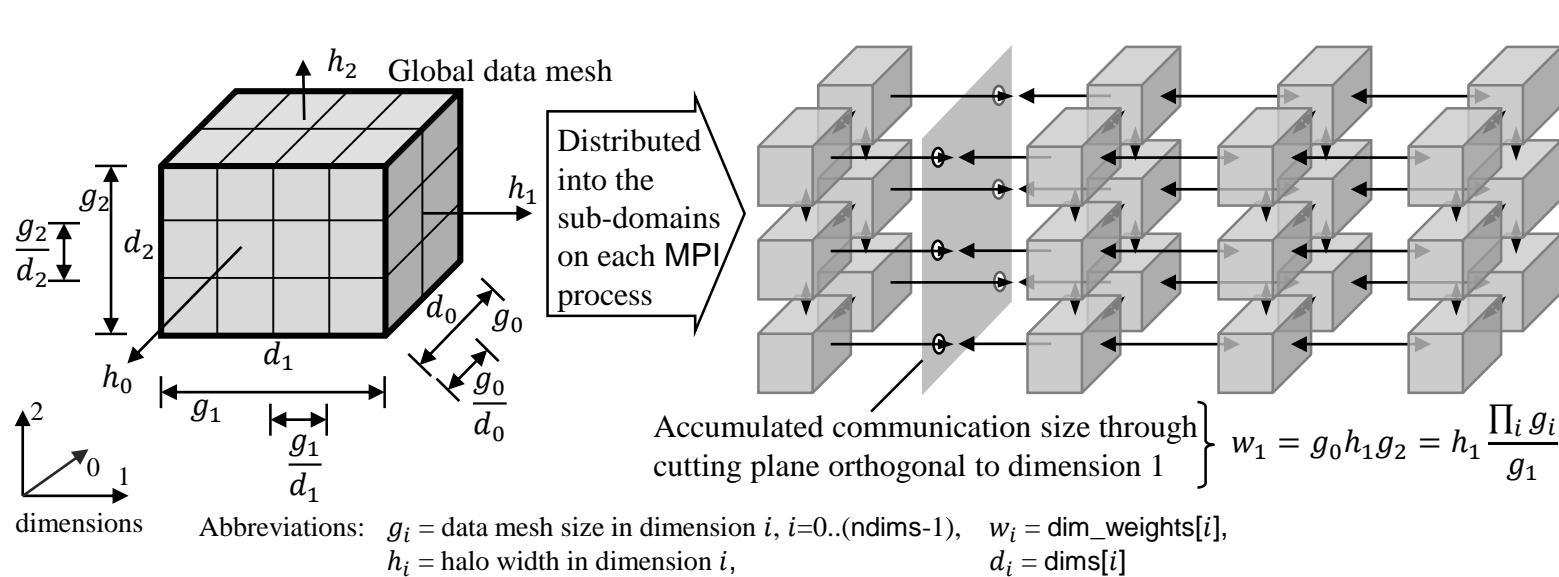


Three dimensions,
i.e., $\text{ndims}=3$



The arguments **dim_weights[i]** $i = 0::(\text{ndims}-1)$, abbreviated with **w_i**, should be specified as the accumulated message size (in bytes) communicated in one communication step through each **cutting plane** orthogonal to dimension d_i and in each of the two directions.

The dim_weights[i], example with 3 dimensions, continued



Important:

- The definition of the `dim_weights` ($= w_i$ in this figure) is independent of the total number of processes and its factorization into the dimensions ($= d_i$ in this figure)

Example for the calculation of the accumulated communication size $w_{i,i=0..2}$ in each dimension.

- g_i – The data mesh sizes $g_{i,i=0..2}$ express the three dimensions of the total application data mesh.
- h_i – The value h_i represents the halo width in a given direction when the 2-dimensional side of a subdomain is communicated to the neighbor process in that direction.

Output from `MPI_Cart/Dims_create_weighted`: The dimensions $d_{i,i=0..2}$

$$w_i = h_i \frac{\prod_j g_j}{g_i}$$

Simple answers to our problems / examples

- Existing API is not application topology aware
 - Factorization of 48,000 processes into $20 \times 40 \times 60$ processes
→ no chance with current API
(e.g. for a mesh with $200 \times 400 \times 600$ mesh points)
 - Use `MPI_Cart_create_weighted` with the `dim_weights=(N/200, N/400, N/600)` with $N=200\cdot400\cdot600$
- Existing API is only partially hardware topology aware
 - An application mesh with 3000×3000 mesh points (i.e., example with `MPI_UNWEIGHTED`) on $25 \text{ nodes} \times 24 \text{ cores} (=600 \text{ MPI processes})$
 - Current API must factorize into 25×24 MPI processes
 - » 25×1 nodes → 120×3000 mesh points → too much node to node communication
 - Optimized answer from `MPI_Cart_create_weighted` may be:
 - » 30×20 MPI processes
 - » Mapped to 5×5 nodes with 600×600 mesh points per node
→ minimal node-to-node communication

The new interfaces – a real implementation

Substitute for / enhancement to existing MPI-1

- `MPI_Dims_create (size_of_comm_old, ndims, dims[ndims]);`
- `MPI_Cart_create (comm_old, ndims, dims[ndims], periods, reorder, *comm_cart);`

New: (in MPI/tasks/C/Ch9/MPIX/)

- **`MPIX_Cart_weighted_create`** (
 - /*IN*/ `MPI_Comm comm_old,`
 - /*IN*/ `int ndims,`
 - /*IN*/ `double dim_weights[ndims], /*or MPIX_WEIGHTS_EQUAL*/`
 - /*IN*/ `int periods[ndims],`
 - /*IN*/ `MPI_Info info, /* for future use, currently MPI_INFO_NULL */`
 - /*INOUT*/ `int dims[ndims],`
 - /*OUT*/ `MPI_Comm *comm_cart);`
 - Arguments have same meaning as in `MPI_Dims_create` & `MPI_Cart_create`
 - See next slide for meaning of `dim_weights[ndims]`
- **`MPIX_Dims_weighted_create`** (`int nnodes, int ndims, double dim_weights[ndims],`
`/*OUT*/ int dims[ndims]);`

Substitute for / enhancement to existing MPI-1

```
MPI_Dims_create ( size_of_comm_old, ndims, dims );
MPI_Cart_create ( comm_old, ndims, dims, periods,
reorder, *comm_cart );
```

Further Interfaces

We proposed the algorithm in

- Christoph Niethammer and Rolf Rabenseifner. 2018.
Topology aware Cartesian grid mapping with MPI.
EuroMPI 2018. <https://eurompi2018.bsc.es/>
→ Program → Poster Session → Abstract+Poster
- <https://fs.hlr.de/projects/par/mpi/EuroMPI2018-Cartesian/>
→ All info + slides + software
- <http://www.hlr.de/training/par-prog-ws/>
→ Practical → MPI31.tar.gz → MPI/tasks/C/eurompi18/

Here, you get the
new **optimized**
interface +
implementation +
documentation

MPIX_Dims_weighted_create() is based on the ideas in:

- Jesper Larsson Träff and Felix Donatus Lübbe. 2015.
Specification Guideline Violations by MPI Dims Create.
In *Proceedings of the 22nd European MPI Users' Group Meeting (EuroMPI '15)*. ACM, New York, NY, USA, Article 19, 2 pages.

Full paper:

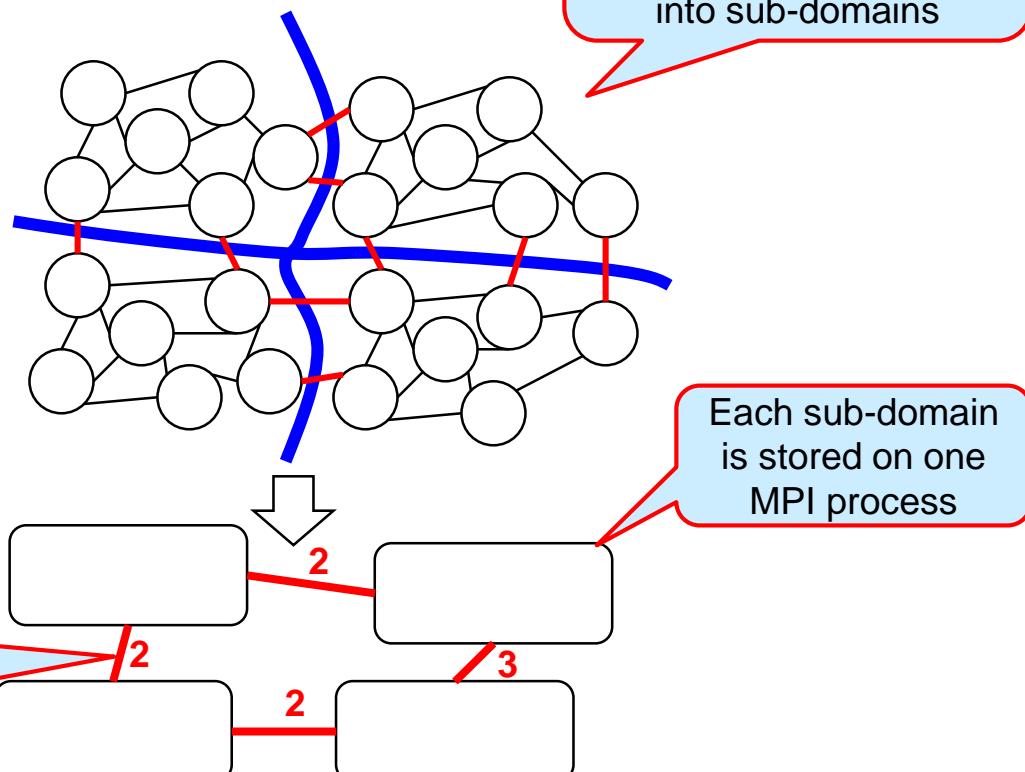
- Christoph Niethammer, Rolf Rabenseifner:
An MPI interface for application and hardware aware cartesian topology optimization.
EuroMPI 2019. Proceedings of the 26th European MPI Users' Group Meeting, September 2019, article No. 6, pages 1-8, <https://doi.org/10.1145/3343211.3343217>

Remarks

- The portable MPIX routines internally use `MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, ...)` to split `comm_old` into ccNUMA nodes,
- plus (may be) additional splitting into NUMA domains.
- With using hyperthreads, it ***may be helpful*** to apply sequential ranking to the hyperthreads,
 - i.e., in `MPI_COMM_WORLD`, ranks 0+1 should be
 - **the first two hyperthreads**
 - of the first core
 - of the first CPU
 - of the first ccNUMA node
- Especially with weights w_i based on $\frac{G}{g_i}$, it is important
 - that the data of the mesh points is **not** read in based on (**old**) ranks in `MPI_COMM_WORLD`,
 - because the domain decomposition must be done based on **comm_cart** and its dimensions and (**new**) ranks

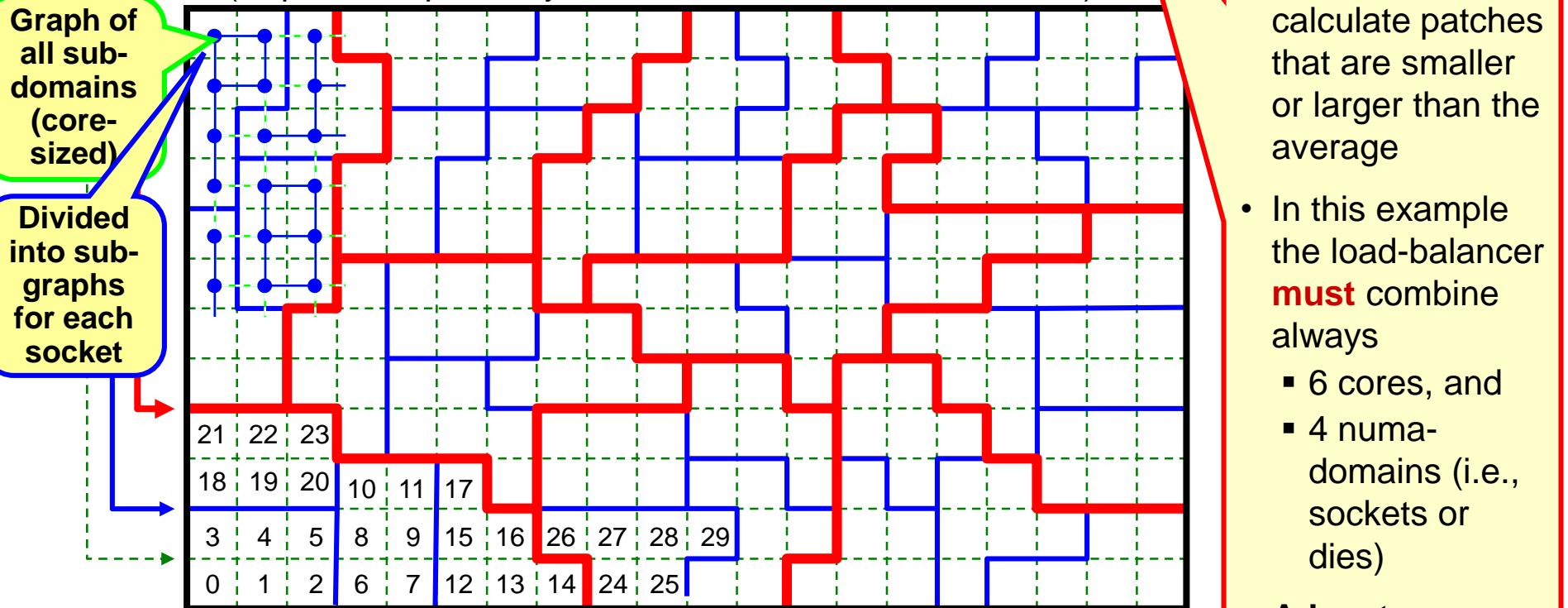
Unstructured Grid / Data Mesh

- Mesh partitioning with special load balancing libraries
 - Metis (George Karypis, University of Minnesota)
 - ParMetis (internally parallel version of Metis)
 - <http://glaros.dtc.umn.edu/gkhome/views/metis/metis.html>
 - Scotch & PT-Scotch (Francois Pellegrini, LaBRI, France)
 - <https://www.labri.fr/perso/pelegrin/scotch/>
 - Goals:
 - Same work load in each sub-domain
 - Minimizing the maximal number of neighbor-connections between sub-domains
 - Minimizing the total number of neighbor sub-domains of each sub-domain



Unstructured Grid / Data Mesh – Multi-level Domain Decomposition through Recombination

1. Core-level DD: partitioning of (large) application's data grid, e.g., } with Metis / Scotch
2. Numa-domain-level DD: recombining of core-domains } with Metis / Scotch
3. SMP node level DD: recombining of socket-domains }
4. Numbering from core to socket to node
(requires sequentially numbered MPI_COMM_WORLD)



Exercise:

Adding a Cartesian Topology

- This exercise is part of our **Hybrid MPI+X** course
- It is **not part of our MPI courses**, but
- we provide it here for you as a self-study exercise / example.

- Given: a 3-D halo communication benchmark using irecv + send
 - cd MPI/tasks/C/Ch9/MPIX/
 - mpicc course/C/Ch9/halo_irecv_send_toggle_3dim_grid_skel.c **MPIX*.c -lm**
 - The application uses a 3-D Cartesian communicator.
 - From this one, it uses 1-D line communicators for communicating in the 3 dimensions
- Overview on the to-do's:
 - “*substituting*” the not reordered Cartesian topology (cart_method==1) through an optimizing algorithm (cart_method==2,3,4)
 - cart_method==2: Add MPIX_Cart... (...MPI_WEIGHTS_EQUAL...)
 - cart_method==3: Calculate the weights based on meshsize_avg_per_proc_startval
Add MPIX_Cart... (...weights...)
 - cart_method==4: same as with cart_method==3, but without weights-calculation
 - Or just use halo_irecv_send_toggle_3dim_grid.c and look at the diff
 - diff halo_irecv_send_toggle_3dim_grid_skel.c halo_irecv_send_toggle_3dim_grid.c
 - Measure the communication bandwidth win
 - For default mesh size 2 / 2 / 2
 - For other mesh sizes, e.g., 1 / 2 / 4

My apologies for missing Fortran

See /* TODO lines

Start a 8 or 12-node batch-job with your own input file:
Report your acceleration factors to the course group

Exercise: Explanations

- Input per measurement, e.g.on 8 nodes x 2 CPUs x 12 cores: Example 2

Column 1

- cart_method:
 - 1=Dims_create+Cart_create,
 - 2=Cart_weighted_create (**MPIX_WEIGHTS_EQUAL**),
 - 3=dito(weights),
 - 4=dito manually,
 - 5=Cart_ml_create(dims_ml),
 - 0=end of input

These base values (per process) are multiplied with $\sqrt[3]{\#processes}$ and then with 1, 2, 4, 8, ... 512, e.g., with 192 processes: $2 \cdot \sqrt[3]{192} \cdot 512 = 5910$ (rounded to a multiple of the dim. of the process grid). See also later the slide explaining the output.
 Recommendation for several experiments: **Use the same initial mesh volume** (here 8), e.g., 1x2x4, 2x2x2, 4x2x1. Note that this application data mesh volume is **completely independent** of the number of hardware nodes, CPUs, cores.

Columns 2-4

- Data mesh sizes, integer start values (= ratio) 0 0 = contiguous 1 2 4

Columns 5-10

- Using MPI_Type_vector, for each dimension a pair of blocklength&stride 0 0 0 0 0 0

Columns 11-13

- weights (double values) (only with cart_method==4) 1.00 0.50 0.25

Column 11

- number of hardware levels (only with cart_method==5)
 - dims_ml: for each of the 3 Cartesian dimensions a list of 3 dimensions from outer to inner hardware level, e.g., 8 nodes x 2 CPUs x 12 cores are split into 1x2x4 nodes x 2x1x1 CPUs x 2x3x2 cores

Columns 12-14

- dims_ml[d=0] = 1 2 2

15-17

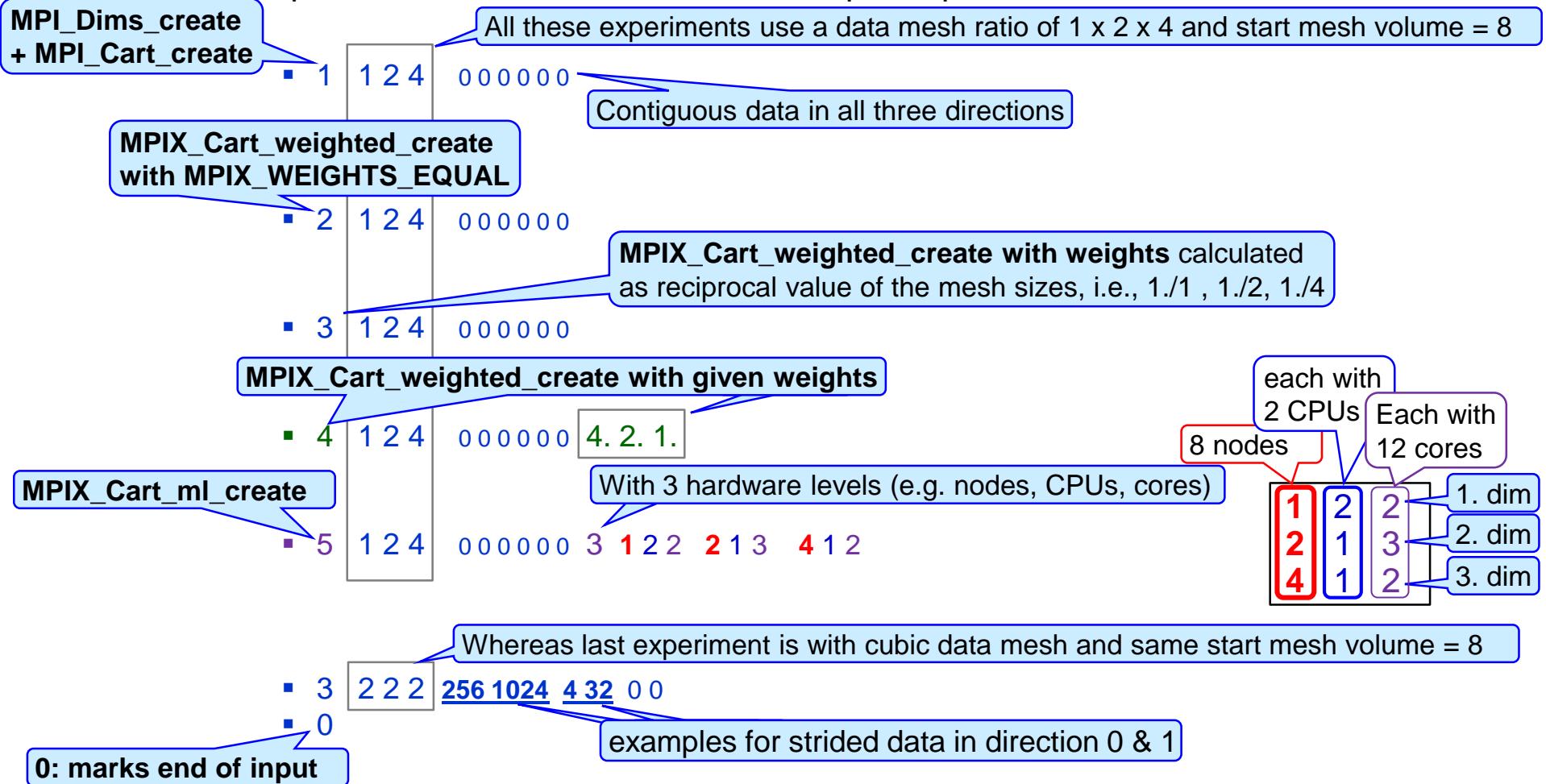
- dims_ml[d=1] = 2 1 3

18-20

- dims_ml[d=2] = 4 1 2

Exercise: Explanations, continued

- Input can be concatenated to one line per experiment:



Exercise: Additional Remarks

- Caution with stdout and stdin when switching I/O from process world_rank==0 to cart_rank==0:
 - **Before** establishing the new comm_cart, all I/O on stdout/stdin is done by world_rank==0 (in MPI_COMM_WORLD)
 - **After** establishing the new comm_cart, all I/O on stdout/stdin is done by cart_rank==0 (in comm_cart)
 - In between, we recommended (although it is not guaranteed that an *output on comm_cart* may overtake an *output on MPI_COMM_WORLD*):
 - MPI_Barrier(MPI_COMM_WORLD);
 - sleep(1); // costs nearly nothing, e.g., 30 Mio € TCO/year / (365 days/year * 24 hours/day * 3600 sec/hour) * 1 sec = 1€
 - MPI_Barrier(comm_cart);
- The following slide shows the win through the re-ranking by the new routines:
 - Less % is better – the communication time reduction factors are:
 - 1.1-1.2 
 - 1.75 
 - 2.75 
 - 4.5-5.0 

Halosize/process ~ = 26 MB	Depend on chosen dims	MPI_Dims_create + MPI_Cart_create	MPIX_Cart_weighted_create(MPI_WEIGHTS_EQUAL)	MPIX_Cart_weighted_create(...weights...)
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	Communication time [ms]	Communication time [ms]
2 x 2 x 2	8x2x12	84.545 = baseline	52.666 = 62%	48.556 = 57%
	64x2x12	194.856 = baseline	73.756 = 38%	72.051 = 37%
	512x2x12	247.631 = baseline	85.530 = 35%	85.491 = 35%
1 x 2 x 4	8x2x12	172.850 = baseline	63.796 = 37%	37.953 = 22%
	64x2x12	360.364 = baseline	91.524 = 25%	74.199 = 21%
	512x2x12	457.858 = baseline	125.468 = 27%	93.615 = 20%
4 x 2 x 1	8x2x12	40.050 = baseline	59.421 = 148%	36.778 = 92%
	64x2x12	78.503 = baseline	100.203 = 128%	69.802 = 89%
	512x2x12	103.002 = baseline	93.189 = 90%	85.044 = 83%

Exercise: To do (1)

- **cd MPI/tasks/C/Ch9/MPIX/**
 - You get the benchmark skeleton `halo_irecv_send_toggle_3dim_grid_skel.c`
 - And all `MPIX_*.c` files and the header `MPIX_interface_proposal.h`
- **mpicc -o halo.exe halo_irecv_send_toggle_3dim_grid_skel.c MPIX_*.c -lm**
- First test with non-optimized `cart_method==1`, i.e., `MPI_Dims_create + MPICart_create`
 - Choose your batch job: `halo_skel_[LRZ|VSC|HLRS].sh` which contains
 - Number of nodes and cores/node
 - and, e.g.,
`mpirun -np 192 ./halo.exe < input-skel.txt`
 - Start your batchjob
 - Try to understand the output:
 - It contains two experiments: a mesh with cubic  and one with non-cubic  ratio
 - The number of MPI processes, e.g. 192, is factorized → domain decomposition into, e.g., 8 x 6 x 4 processes
 - The measurements are done for 10 global meshesizes
 - The domain decomposition implies the local meshesizes
 - The local meshesizes imply the size of the halos in each direction
 - → the sum of the time for the communication into the 3 dimensions x 2 directions (left+right)

1	2	2	2	0	0	0	0	0	0
1	1	2	4	0	0	0	0	0	0
0									

See
next slide

Exercise: To do (2)

`cart_method = 1`

start mesh sizes integer start values for 3 dimensions = **2 2 2**
 blocklength & stride pairs for each of the 3 dimensions = **0 0 0 0 0 0**

Creating the Cartesian communicator and further input arguments:

`cart_method == 1: MPI_Dims_create + MPI_Cart_create`

[MPI_Barrier and switching to output via stdout through rank==0 in comm_cart]

`ndims=3 dims= 8 6 4`

message size transfertime duplex bandwidth per process and neighbor (mesh&halo in #floats)

			meshsizes total=	per process=	halosizes=
128 bytes	34.537 usec	3.706 MB/s	16 12 12	2 2 3	16= 6 + 6 + 4
432 bytes	39.840 usec	10.843 MB/s	24 24 24	3 4 6	54= 24 + 18 + 12
1728 bytes	41.122 usec	42.021 MB/s	48 48 48	6 8 12	216= 96 + 72 + 48
6688 bytes	23.961 usec	96 96 92	12 16 23	836= 368 + 276 + 192	
25576 bytes	93.703 usec	184 186 184	23 31 46	3197= 1426 + 1058 + 713	
104408 bytes	271.721 usec	376 372 372	47 62 93	13051= 5766 + 4371 + 2914	
411192 bytes	1033.001 usec	744 738 740	93 123 185	51399= 22755+17205 + 11439	
1636392 bytes	4398.680 usec	1480 1476 1476	185 246 369	204549= 90774+68265 + 45510	
6561336 bytes	18173.518 usec	2960 2958 2956	370 493 739	820167=364327+273430+182410	
26194104 bytes	76132.216 usec	344.061 MB/s	5912 5910 5908	739 985 1477	3274263=1454845+1091503+727915

`cart_method = 1`

* 2 directions * 4 byte

start mesh sizes integer start values for 3 dimensions = **1 2 4**

blocklength & stride pairs for each of the 3 dimensions = **0 0 0 0 0 0**

`cart_method == 1: MPI_Dims_create + MPI_Cart_create`

`ndims=3 dims= 8 6 4`

message size transfertime duplex bandwidth per process and neighbor (mesh&halo in #floats)

160 bytes	14.720 usec	10.870 MB/s	8 12 24	1 2 6	20= 12 + 6 + 2
... 34936960	156869.278 usec	222.714 MB/s	2960 5910 11816	370 985 2954	4367120=2909690+1092980+364450

Same values, because
`MPI_Dims_create()` factorizes
 the #processes independent
 from the user's meshsizes.

Second value for
 our table

These base values (per process) are multiplied with $\sqrt[3]{\#processes}$ and then with 1, 2, 4, 8, ... 512,
 e.g., $2 \cdot \sqrt[3]{192} \cdot 512 = 5910$
 (rounded to a multiple of the dimension of the process grid)

REC → online

Exercise: To do (3)

- Fill in the table**

Defined in batch job + hardware knowledge

Execution time of **largest** mesh and halo size of both measurements

Given from MPI_Dims_create()

Nodes
CPUs
cores

d=0:	8	= 8 x 1 x 1
d=1:	6	= 1 x 2 x 3
d=2:	4	= 1 x 1 x 4

Total 192 = **8 x 2 x 12**

Have to be calculated by hand:

Fill in maximal factors.

Factorize first the cores and start with d=2.

Then the CPUs & then the nodes.
(All based on sequential ranking of MPI_COMM_WORLD)

Halosize/process ≈ 26 MB		MPI_Dims_create + MPI_Cart_create			MPI_Cart_create(MPIX_WEIGHTS_EQUAL)			MPIX_Cart_weighted_create(...weights...)		
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]			
2 x 2 x 2	_____ x _____	_____	____ = _____ x _____ ____ = _____ x _____ ____ = _____ x _____	= baseline						
1 x 2 x 4	Same as above	_____	Same as above							

Exercise: To do (4)

- `cp halo_irecv_send_toggle_3dim_grid_skel.c halo_optim.c`
- **Edit `halo_optim.c`**
 - On lines 160, 165, and 171, substitute the `/* TODO: ... */` by correct code

```
153 if (cart_method == 1) {  
154     if (my_world_rank==0) printf("cart_method == 1: MPI_Dims_create + MPI_Cart_create\n");  
155     MPI_Dims_create(size, ndims, dims);  
156     MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, 0, &comm_cart);  
157 } else if (cart_method == 2) {  
158     if (my_world_rank==0) printf("cart_method == 2: MPIX_Cart_weighted_create( MPIX_WEIGHTS_EQUAL )\n");  
159     /* TODO: Appropriate call to MPIX_Cart_weighted_create(...) with MPIX_WEIGHTS_EQUAL  
160      instead of calling MPI_Dims_create() and MPI_Cart_create() as in method 1 */  
161  
163 } else if (cart_method == 3) {  
164     /* TODO: Appropriate calculation of weights[ ] based on meshsize_avg_per_proc_startval[ ] */  
165     if (my_world_rank==0) { printf("cart_method == 3: MPIX_Cart_weighted_create( weights := _____ TODO _____)\n");  
166         printf("weights= "); for (d=0; d<ndims; d++) printf(" %lf",weights[d]); printf("\n");  
167     }  
168     /* TODO: Appropriate call to MPIX_Cart_weighted_create(...) with weights  
169      instead of MPIX_WEIGHTS_EQUAL as in method 2 */  
170  
174 } else ...
```

Exercise: To do (5)

- `mpicc -o halo_optim.exe halo_optim.c MPIX_*.c`
- Check: `diff halo_optim.c halo_irrecv_send_toggle_3dim_grid_solution.c`
- Now, use all three `cart_method==1, 2, 3`

– Choose your batch job:

- `halo_optim_[LRZ|VSC|HLRS].sh` which contains, e.g.:
- `mpirun -np 192 ./halo_optim.exe < input-optim.txt`
- Start your batchjob → output file `output_optim.txt`
- Fill in the table

Note, that the optimization changes the dims-array → modified halo sizes!

Although halos may be larger, the optimized communication time should be shorter!

Base meshesizes

Cart_method

1	2	2	2	0	0	0	0	0
2	2	2	2	0	0	0	0	0
1	1	2	4	0	0	0	0	0
2	1	2	4	0	0	0	0	0
3	1	2	4	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Block length +
stride for each
dimension

Halosize/process ~ = 26 MB		cart_method==1			cart_method==2			cart_method==3		
		MPI_Dims_create + MPI_Cart_create			MPIX_Cart_weighted_ create(MPIX_WEIGHTS_EQUAL)			MPIX_Cart_weighted_ create(...weights...)		
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_mi[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_mi[d=0] dims_ml[d=1] dims_ml[d=2]	Communication dims_mi[d=0] Reported by dims_mi[d=1] MPI_Cart_weighted_ dims_mi[d=2] create()	Communication dims_mi[d=0] Same as with dims_mi[d=1] MPI_Cart_weighted_ dims_mi[d=2] create()	Communication dims_mi[d=0] Same as with dims_mi[d=1] MPI_Cart_weighted_ dims_mi[d=2] create()	Communication dims_mi[d=0] Same as with dims_mi[d=1] MPI_Cart_weighted_ dims_mi[d=2] create()	
2 x 2 x 2	— x — x —	————— = baseline	— = x — x — — = x — x — — = x — x —	————— = _____ % of baseline	— = x — x — — = x — x — — = x — x —	————— Reported by MPI_Cart_weighted_ create()	————— Same as with MPIX_WEIGHTS_EQUAL	————— Same as with MPIX_WEIGHTS_EQUAL	————— Same as with MPIX_WEIGHTS_EQUAL	
1 x 2 x 4	— x — x —	————— = baseline	Same as above	————— = _____ % of baseline	Same as above	————— Same as with MPIX_WEIGHTS_EQUAL	————— = _____ % of baseline	————— = _____ % of baseline	————— = _____ % of baseline	

Exercise 1a: Results – HLRS, Stuttgart, hazelhen

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create		MPIX_Cart_weighted_ create(MPIX_WEIGHTS_EQUAL)		MPIX_Cart_weighted_ create(...weights...)	
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]
2 x 2 x 2	8 x 2 x 12	78.748 = baseline	8 = 8 x 1 x 1 6 = 1 x 2 x 3 4 = 1 x 1 x 4	50.971  = 65% of baseline	8 = 2 x 2 x 2 6 = 2 x 1 x 3 4 = 2 x 1 x 2	Same as with MPIX_WEIGHTS_EQUAL	
1 x 2 x 4	8 x 2 x 12	168.891 = baseline	Same as above	64.691  = 38% of baseline	Same as above	38.406  = 23% of baseline	4 = 1 x 2 x 2 6 = 2 x 1 x 3 8 = 4 x 1 x 2

Exercise 1a: Results – LRZ, Garching, ivyMUC

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create		MPIX_Cart_weighted_ create(MPIX_WEIGHTS_EQUAL)		MPIX_Cart_weighted_ create(...weights...)	
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]
2 x 2 x 2	12 x 2 x 8	34.814 = baseline	8 = 12 x 2/3 x 1 6 = 1 x 3 x 2 4 = 1 x 1 x 4	26.675  = 77% of baseline	6 = 3 x 1 x 2 8 = 2 x 2 x 2 4 = 2 x 1 x 2	Same as with MPIX_WEIGHTS_EQUAL	
1 x 2 x 4	12 x 2 x 8	54.344 = baseline	Same as above	35.665  = 66% of baseline	Same as above	22.933  = 42% of baseline	4 = 1 x 2 x 2 6 = 3 x 1 x 2 8 = 4 x 1 x 2

Exercise 1a: Results – VSC, Vienna, vsc3

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create			MPIX_Cart_weighted_ create(MPIX_WEIGHTS_EQUAL)			MPIX_Cart_weighted_ create(...weights...)		
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]			
2 x 2 x 2	12 x 2 x 8	61.803 = baseline	8 = 12 x 2/3 x 1 6 = 1 x 3 x 2 4 = 1 x 1 x 4	49.722 = 80% of baseline	6 = 3 x 1 x 2 8 = 2 x 2 x 2 4 = 2 x 1 x 2			Same as with MPIX_WEIGHTS_EQUAL		
1 x 2 x 4	12 x 2 x 8	97.658 = baseline	Same as above	67.208 = 69% of baseline	Same as above	40.283 = 41% of baseline		4 = 1 x 2 x 2 6 = 3 x 1 x 2 8 = 4 x 1 x 2		

Exercise 1a: Your result: _____

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create			MPIX_Cart_weighted_ create(MPIX_WEIGHTS_EQUAL)			MPIX_Cart_weighted_ create(...weights...)		
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]			
2 x 2 x 2	__ x __ x __	_____	__ = __ x __ x __ __ = __ x __ x __ __ = __ x __ x __	_____	__ = __ x __ x __ __ = __ x __ x __ __ = __ x __ x __	_____	_____	Same as with MPIX_WEIGHTS_EQUAL		
1 x 2 x 4	__ x __ x __	_____	Same as above	_____	_____	Same as above	_____	____ = __ x __ x __ ____ = __ x __ x __ ____ = __ x __ x __	____ = ____ % of baseline	of baseline

Quiz on Chapter 9-(3) – Virtual topologies

- A. Which types of MPI topologies for virtual process grids exist?
- B. And for which use cases?

1. _____

For _____

2. _____

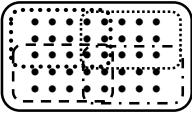
For _____

- C. Where are limits for using virtual topologies, i.e., which use cases do not really fit?

For private notes

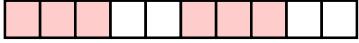
For private notes

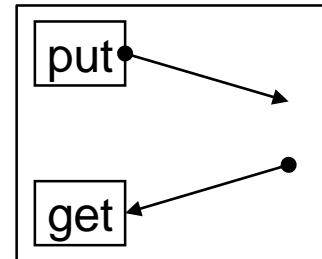
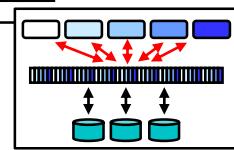
Chap.10 One-sided Communication

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 

10. One-sided communication

– Windows, remote memory access (RMA), synchronization

11. Shared memory one-sided communication
12. Derived datatypes 
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



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



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:

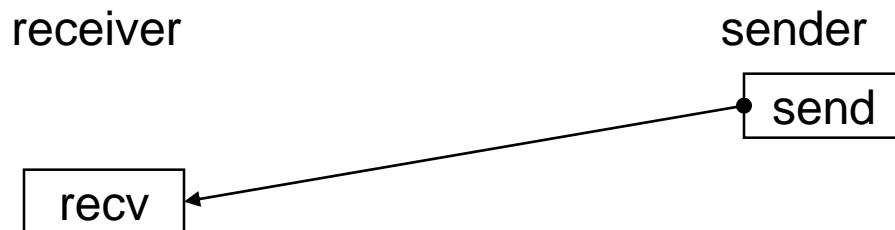
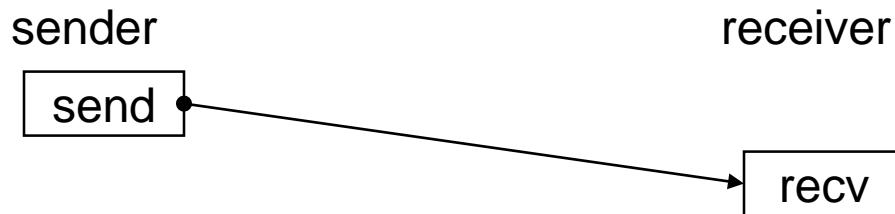
explicit transfer, implicit synchronization,
implicit cache operations

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

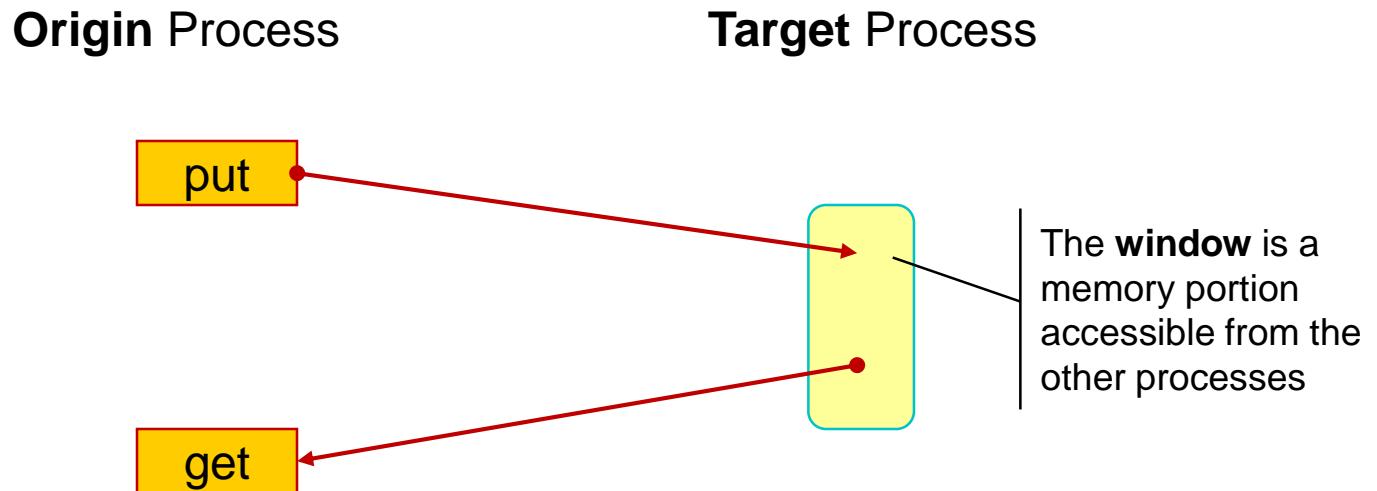
Cooperative Communication

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



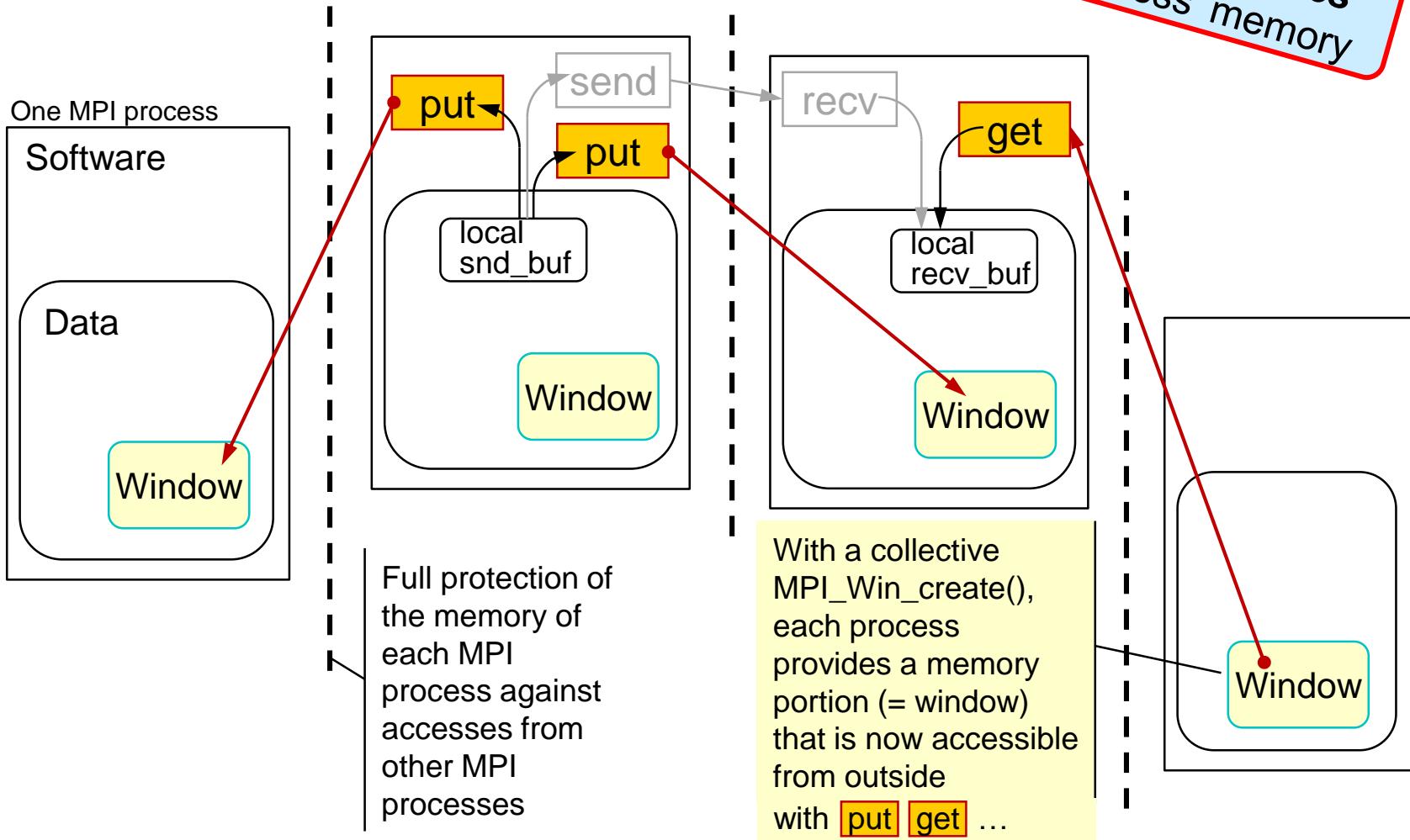
One-sided Communication

- 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



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.
- Remote Memory Access (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/allocation

Synchronization

Remote Memory Accesses
(RMA)

Remote Memory Accesses

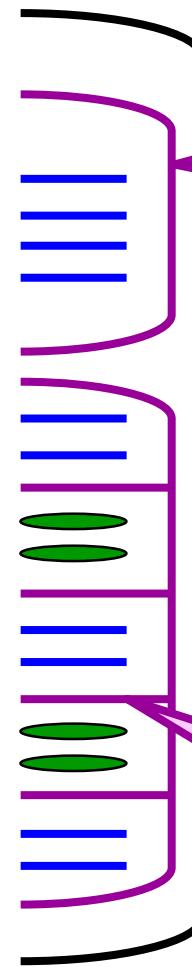
Local load/store

Remote Memory Accesses

Local load/store

Remote Memory Accesses

Window freeing/deallocation



RMA operations must be surrounded by
synchronization calls

RMA epoch

Local load/store epoch

...

Epochs must be separated by
synchronization calls



Window creation or allocation

Four different methods

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

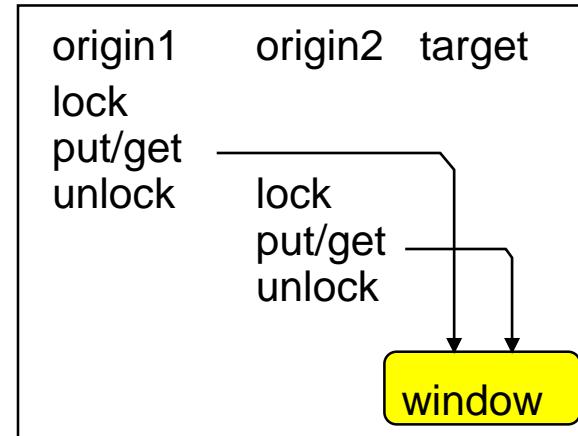
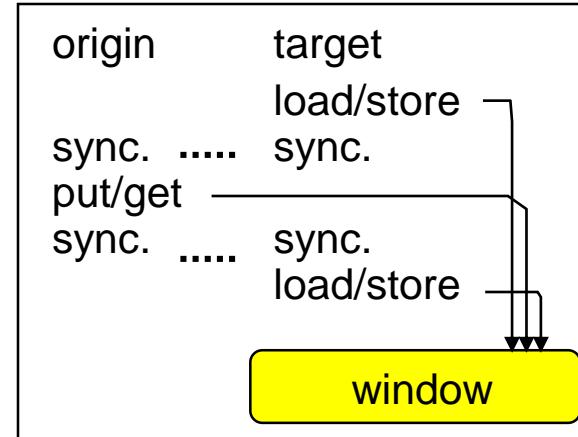
- **MPI_Rget**
- **MPI_Rput**
- **MPI_Raccumulate**
- **MPI_Rget_accumulate**

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

R = request-based

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



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

```
MPI_Win_create( win_base_addrtarget, win_sizetarget,  
                 disp_unittarget, info, comm, win)
```

A normal buffer argument

byte size, `MPI_Aint`

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



language bindings

→ see next slide (skipped)
or MPI Standard

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,
Large count version,
new in MPI-4.0 MPI_Aint disp_unit, MPI_Info info,
MPI_Comm comm, MPI_Win *win)

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*

Python

- Python: *win* = MPI.Win.Create(memory, disp_unit, info, comm)
e.g., a numpy array

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

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)
```

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
```

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

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:

- Substitute the nonblocking communication by one-sided communication.
 - Two choices:

- either **recv_buf = window**
 - **MPI_Win_fence** - the `recv_buf` can be used to receive data
 - **MPI_Put** - to write the content of the local variable `snd_buf` into the remote window (`recv_buf`)
 - **MPI_Win_fence** - the one-sided communication is finished, `recv_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 `recv_buf`
 - **MPI_Win_fence** - the one-sided communication is finished, `recv_buf` is filled

Please use this choice in this exercise!

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

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

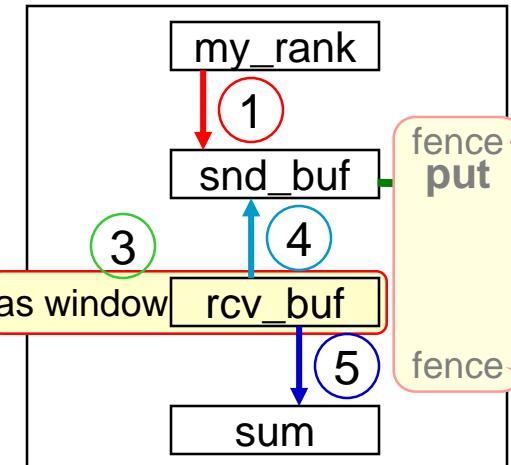
ring.c / .f: Rotating information around a ring

Initialization: 1

Each iteration:



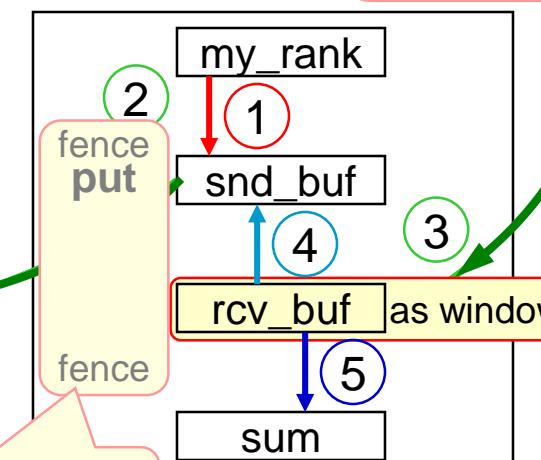
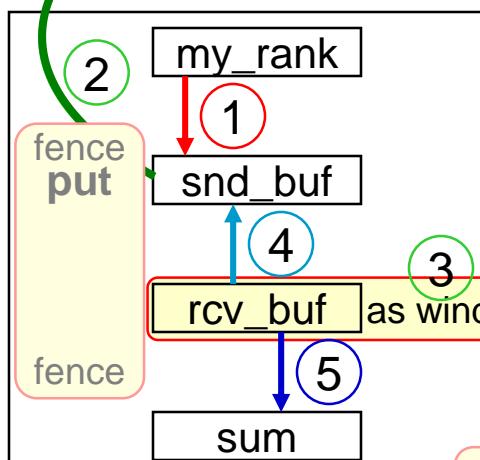
to be substituted
by 1-sided comm.



Solution with
rcv_buf as window

the rcv_buf can be used
to receive data &
want to start RMA

one-sided comm.
is locally and remotely
completed:
snd_buf reusable
rcv_buf is filled



All the rest will come
in Exercise 2



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

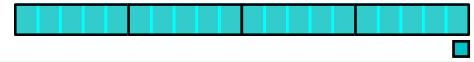
- C/C++: `MPI_Win_create(&rcv_buf, (MPI_Aint) sizeof(int), sizeof(int), MPI_INFO_NULL, ..., &win);`

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/-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)**

During the Exercise (20 min.)



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



And have fun with this a bit longer 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:

Any idea, why I preferred MPI_Put instead of MPI_Get?



Advanced Exercise 1b

- My apologies, there is no advanced exercise at the beginning of this MPI one-sided communication chapter.
- You may look at the previous MPI nonblocking basic and advanced exercises
(before you use the following hyper-refs, please note the current slide number 341, because there aren't any "back" buttons to here)
 - Course Chap. 4 [🔗](#)
 - Exercises 1 [🔗](#), 2 [🔗](#), and advanced exercise 3 [🔗](#)

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 neighbors, 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
Recv or Probe+malloc+Recv
sender: Waitall

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



2nd skip-point: Skip rest of this chapter

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 syncron.

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

- The target buffer is at address

$\text{target_addr} = \text{win_base}_{\text{target_process}}$

+ $\text{target_disp}_{\text{origin_process}} * \text{disp_unit}_{\text{target_process}}$

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,

Like **dest** in MPI_Send

target_rank, target_disp_{origin_process},

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

target_count, target_datatype, win)

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

Overloaded large count
version since MPI-4.0

mpi_f08:
Overloaded large count
version since MPI-4.0 or
or
:: origin_addr
:: origin_count, target_count
:: origin_count, target_count
:: target_rank
:: target_datatype, target_datatype
:: target_disp
:: win
:: ierror

INTEGER, INTENT(IN)
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN)
INTEGER, INTENT(IN)
TYPE(MPI_Datatype), INTENT(IN)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN)
TYPE(MPI_Win), INTENT(IN)
INTEGER, OPTIONAL, INTENT(OUT)

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))`



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

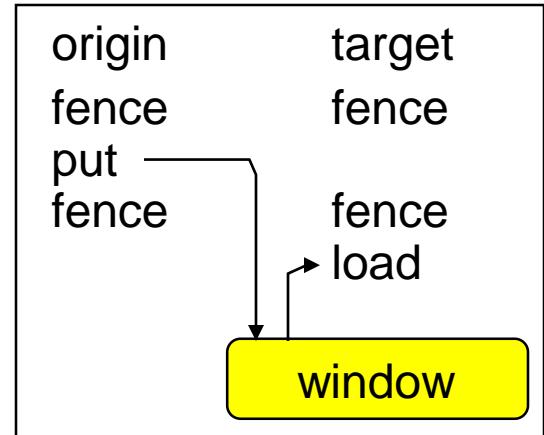
- Accumulates the contents of the origin buffer to the target area specified using the predefined operation `op`
- User-defined operations cannot be used
- Accumulate is **elementwise atomic**:
many accumulates can be done by many origins to one target
-> [*may be expensive*]

```
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/-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_...`

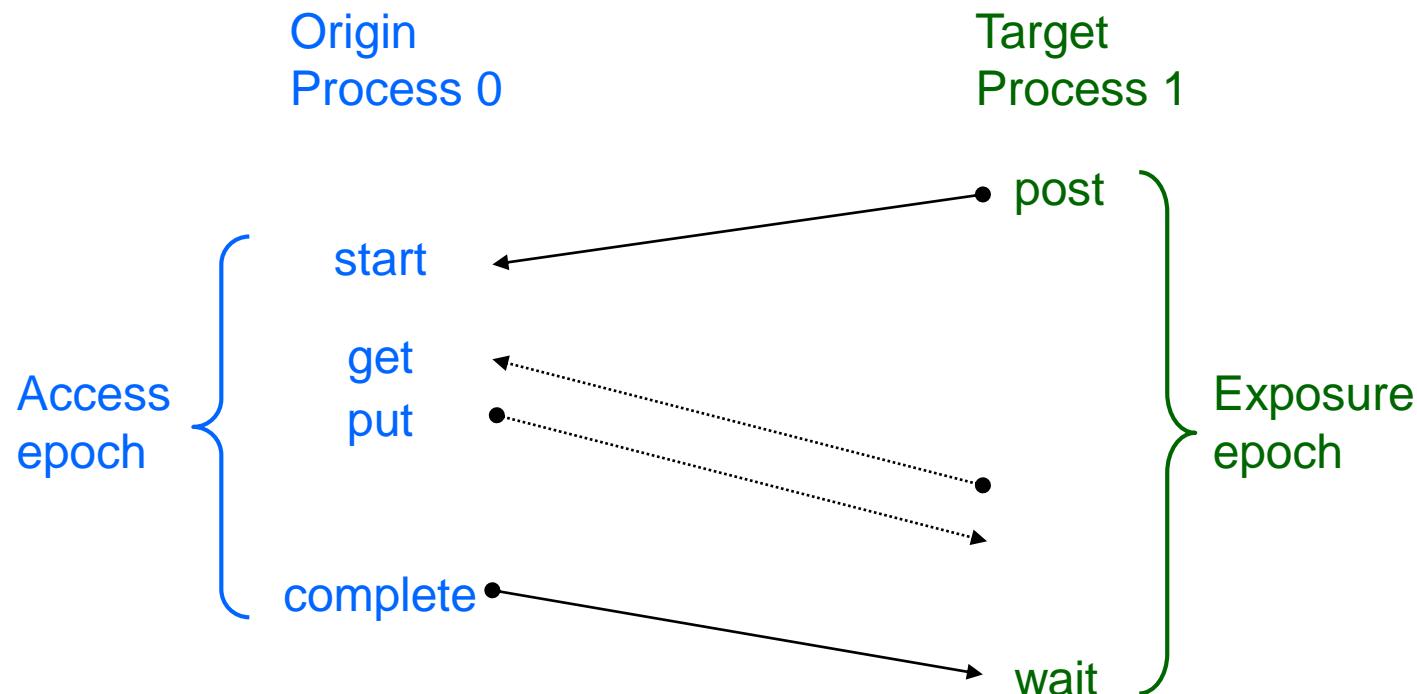
Fortran: `IOR(MPI_MODE_NOSTORE, IOR(MPI_MODE_..., MPI_...))`

Because assertions are bit-vectors, e.g.

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

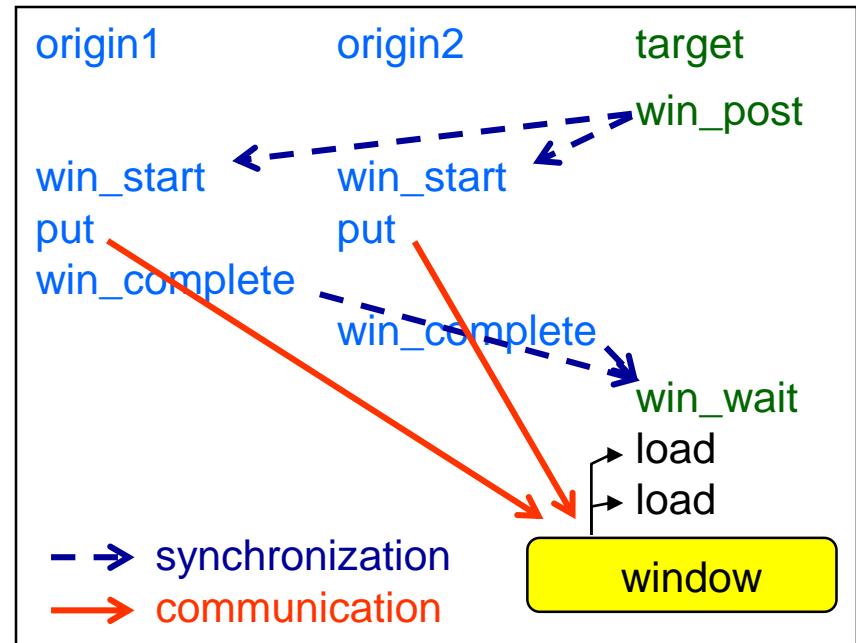
Start/Complete and Post/Wait, I.

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



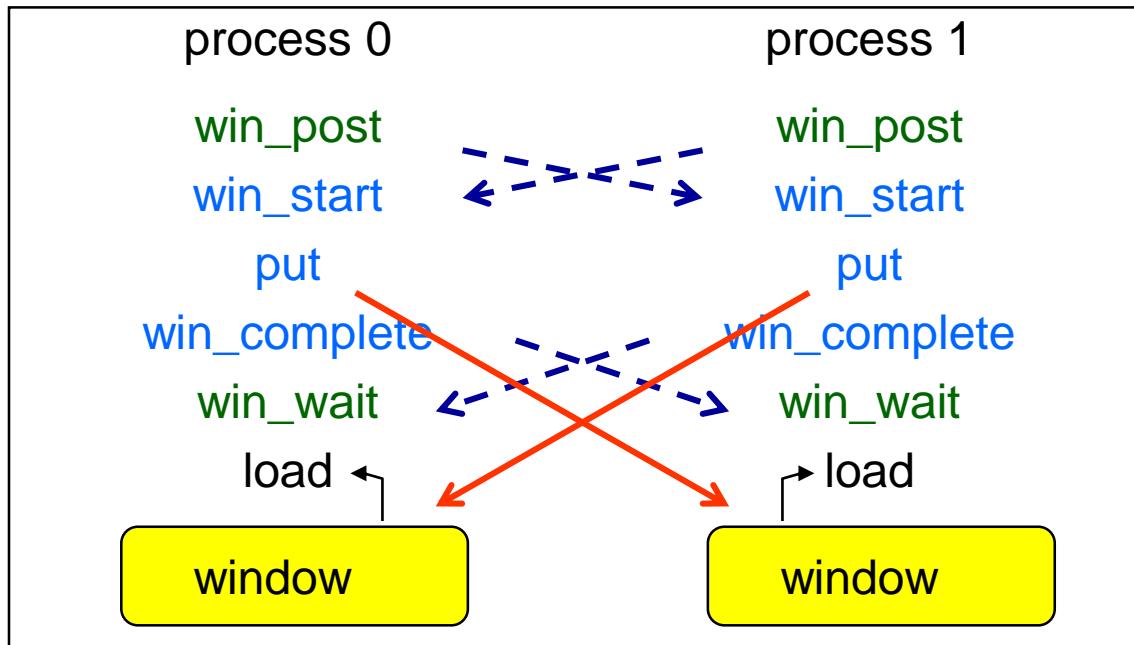
Start/Complete and Post/Wait, II.

- 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.

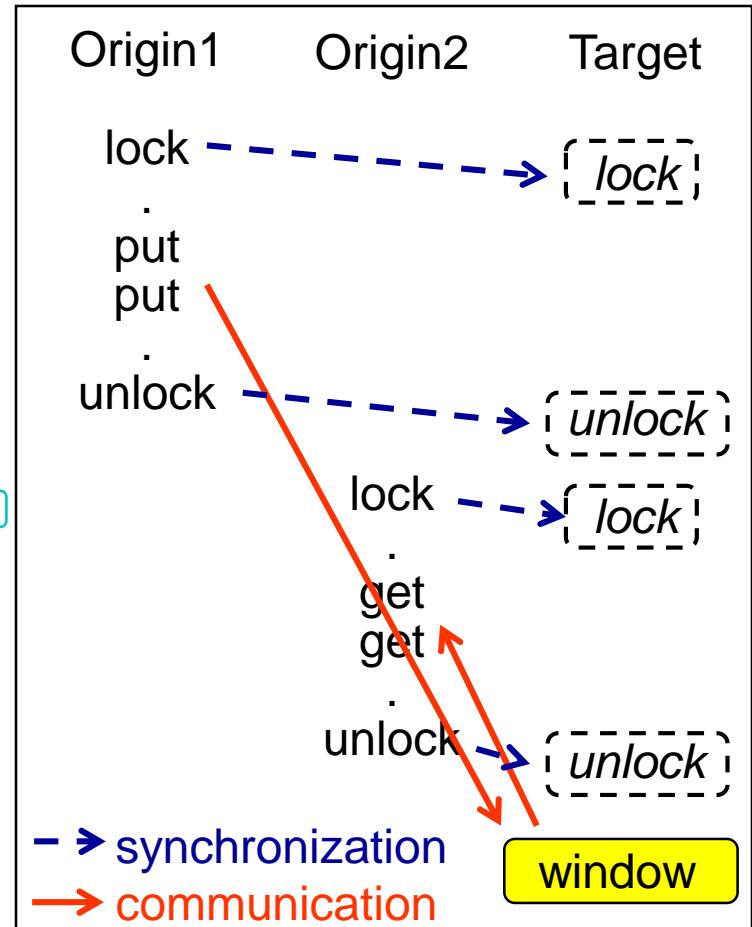
- symmetric communication possible,
only `win_start` and `win_wait` may block



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

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 **New in MPI-4.0** `MPI_Win_allocate_shared`
- RMA completed after `MPI_Unlock` at both origin and target



Fortran Problems with 1-Sided

Source of Process 1
`bbbb = 777`
`call MPI_WIN_FENCE`
`call MPI_PUT(bbbb`
`into buff of process 2)`
`call MPI_WIN_FENCE`

Source of Process 2
~~`buff = 999`~~
`call MPI_WIN_FENCE`
`call MPI_WIN_FENCE`
`print *, buff`

Executed in Process 2
`register_A := 999`

`stop application thread`
`buff := 777 in PUT handler`
`continue application thread`

`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)

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

- 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)** ~~-----~~

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: 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

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

Please use
this choice in
this exercise!

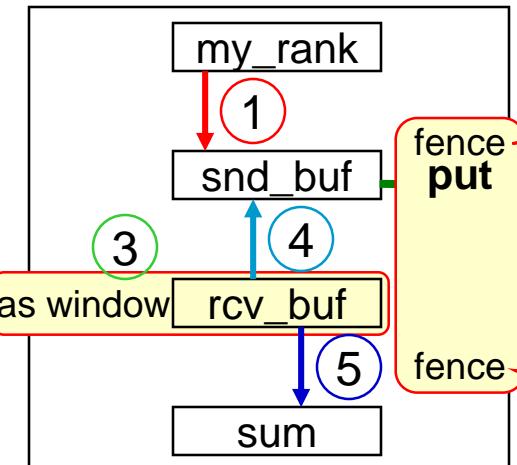
ring.c / .f: Rotating information around a ring

Initialization: 1

Each iteration:



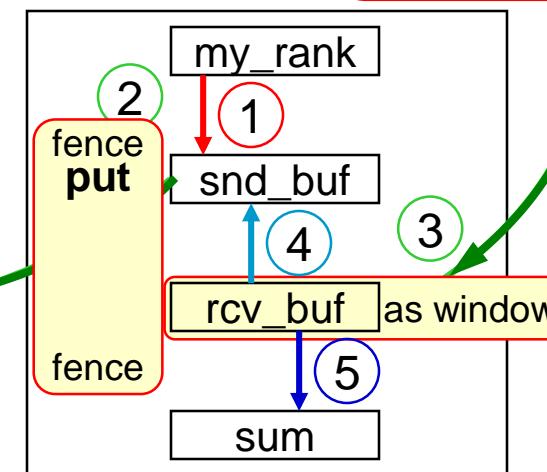
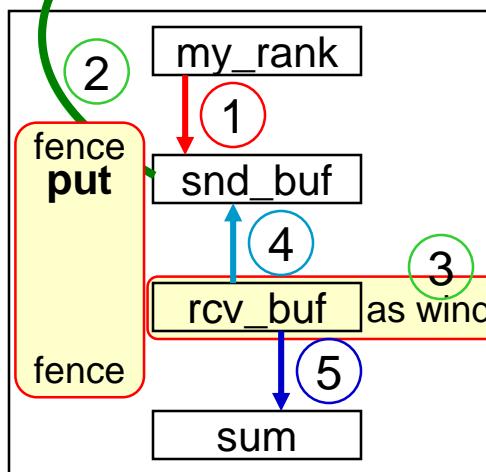
to be substituted
by 1-sided comm.



Solution with
rcv_buf as window

the rcv_buf can be used
to receive data &
want to start RMA

one-sided comm.
is locally and remotely
completed:
snd_buf reusable
rcv_buf is filled



MPI–One-sided Exercise 2: additional hints

- 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 **and** 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](#)

C

Fortran

Fortran

During the Exercise (15 min.)



Please stay here in the main room while you do this 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

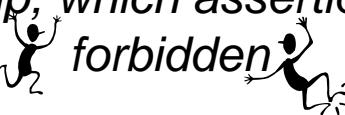
and continue your discussions with your fellow learners:

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 do apply and which ones are forbidden



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

C

Fortran

Python

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`



Quiz on Chapter 10 – One-sided Communication

A. Please describe what is a window?

B. If you execute an MPI_Put, where is the send and where the receive buffer?

C. And same question for MPI_Get?

D. Is the following small example correct?

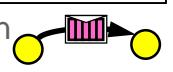
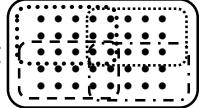
```
rank 0
MPI_Win_create(rcv_buf,...,&win)
MPI_Put(buf,...to rank 1..., win)
MPI_Win_fence(0,win)
```

```
rank 1
MPI_Win_create(rcv_buf,...,&win)
MPI_Win_fence(0,win)
```

E. If all processes are in the role of origin and target process and you want to use the pairs MPI_Win_post & MPI_Win_start before and MPI_Win_complete & MPI_Win_wait after your RMA calls MPI_Put (or ...Get). In what order do you have to call these routines?

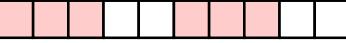
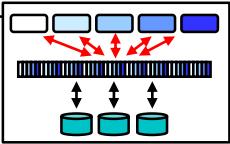
F. And if you call the two calls of such a pair in the contrary order, what could then happen?

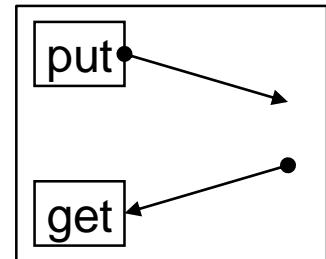
Chap.11 Shared Memory One-sided Communication

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 
10. 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**

12. Derived datatypes 
13. Parallel file I/O 
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



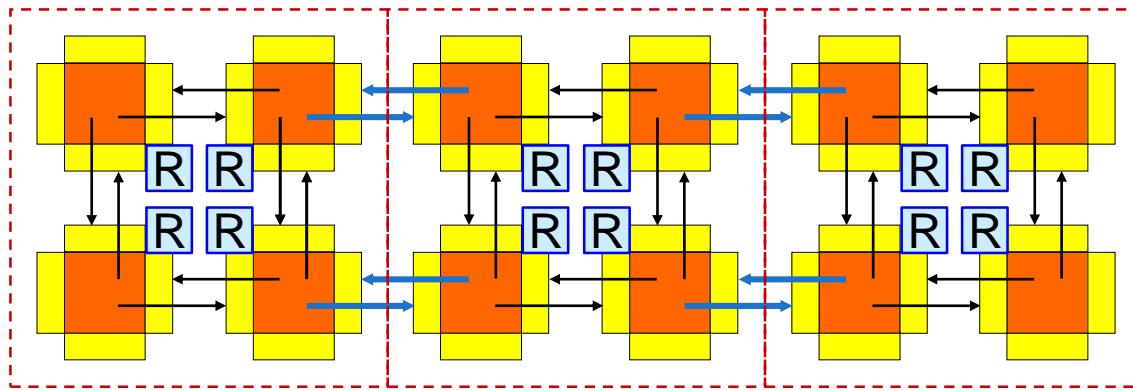
MPI shared memory

- 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)



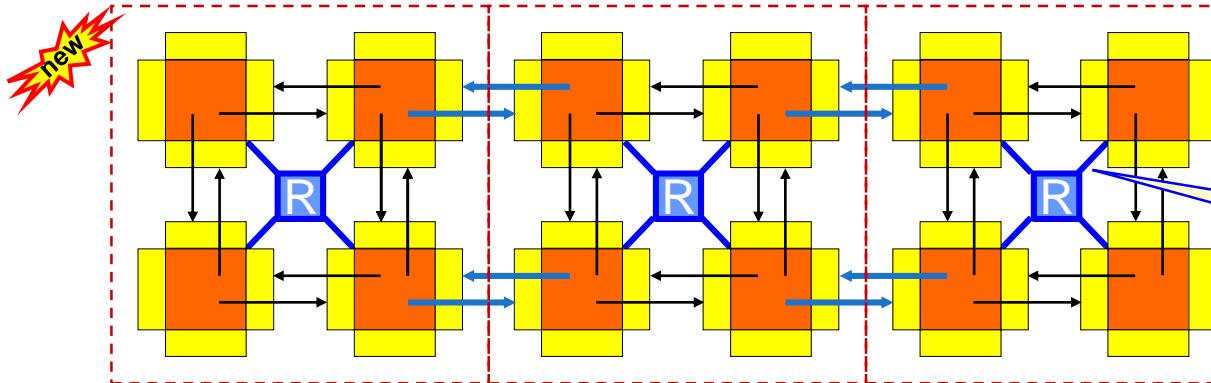
Programming opportunities with MPI shared memory:

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 only once within each SMP node

Direct loads & stores,
no library calls

Using MPI shared memory methods

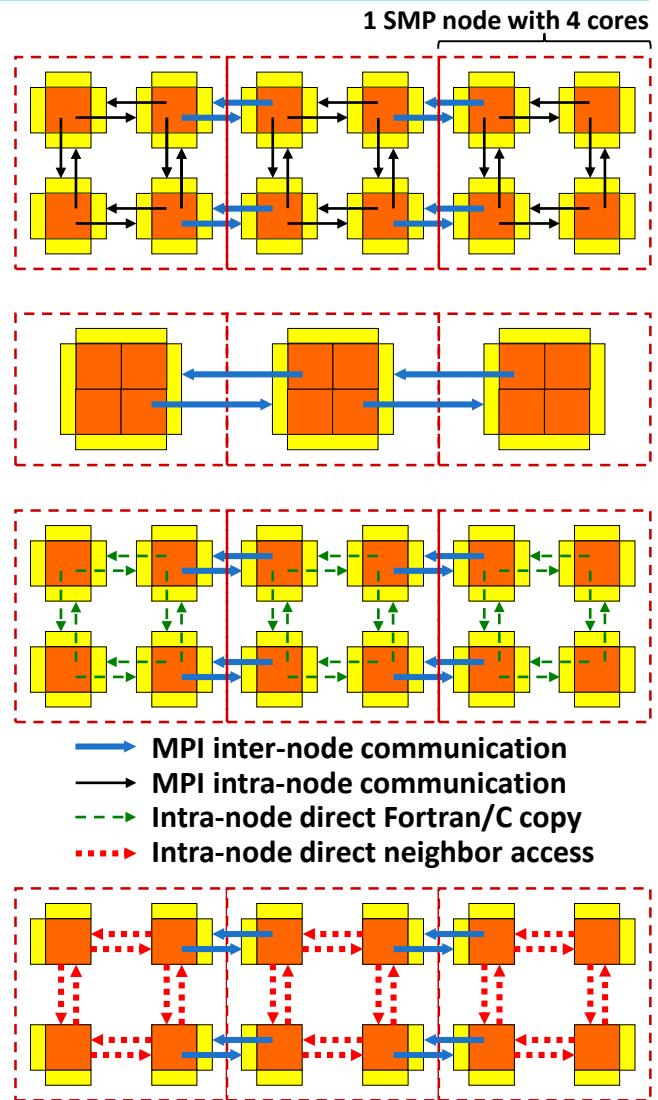
MPI shared memory can be used
to significantly reduce the memory needs for replicated data.



Programming opportunities with MPI shared memory:

2) Hybrid shared/cluster programming models

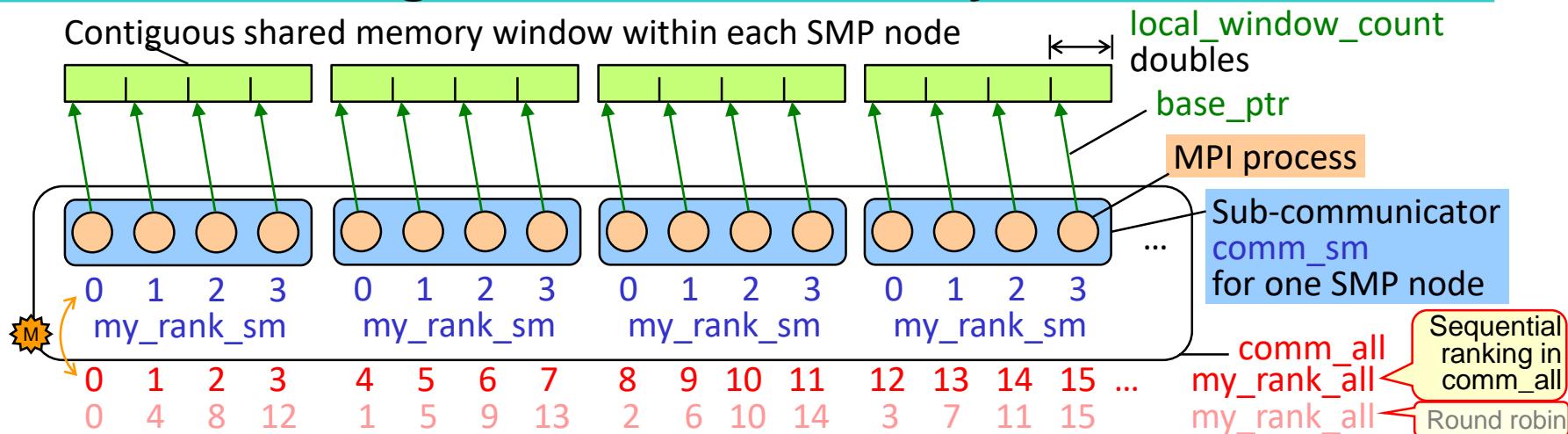
- MPI on each core (not hybrid)
 - Halos between all cores
 - MPI uses internally shared memory and cluster communication protocols
- MPI+OpenMP
 - Multi-threaded MPI processes
 - Halos communica. only between MPI processes
- MPI cluster communication + MPI shared memory communication
 - Same as “MPI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



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,
    collective call      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 */

```

```

F MPI_Win_allocate_shared ((MPI_Aint) local_window_count*disp_unit, disp_unit,
    collective call      MPI_INFO_NULL, comm_sm, &base_ptr, &win_sm);

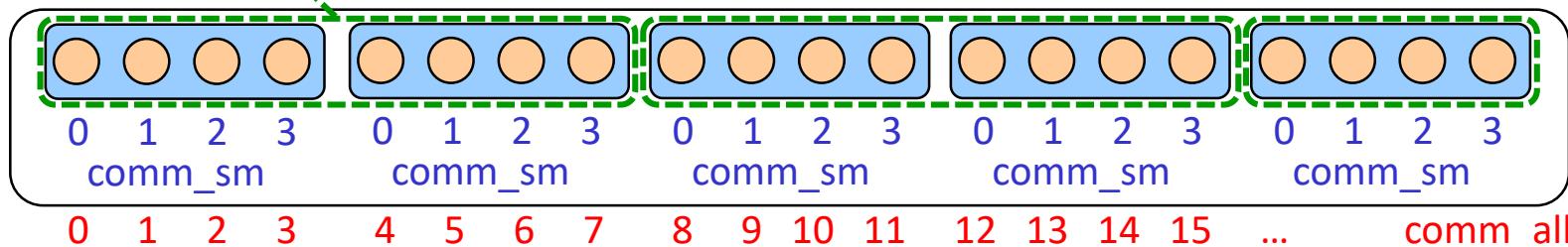
```

- F In Fortran, MPI-3.1/-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.

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.

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"

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.

May not work with Intel-MPI

New in MPI-4.0

Exercise 1: Shared memory ring communication

- 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!
 - 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(&rev_buf, sizeof(int), sizeof(int), MPI_INFO_NULL,
-----MPI_COMM_WORLD, &rcv_buf_ptr, &win);
-----snd_buf = *rcv_buf_ptr;
-----sum += *rcv_buf_ptr;
  
```

- 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(rev_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

- 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

Exercise 1: Shared memory ring communication

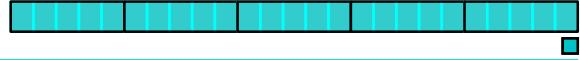
- Tasks:

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**
 - 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** **)**

i.e., in C and Fortran,
each process points to
its own window portion

During the Exercise (25 min.)



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



And have fun with this longer 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:



*You may already discuss with your colleagues,
which is the correct index for `rcv_buf[?]` within an origin process
to access the `rcv_buf` of the “right” (=target) process?*



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

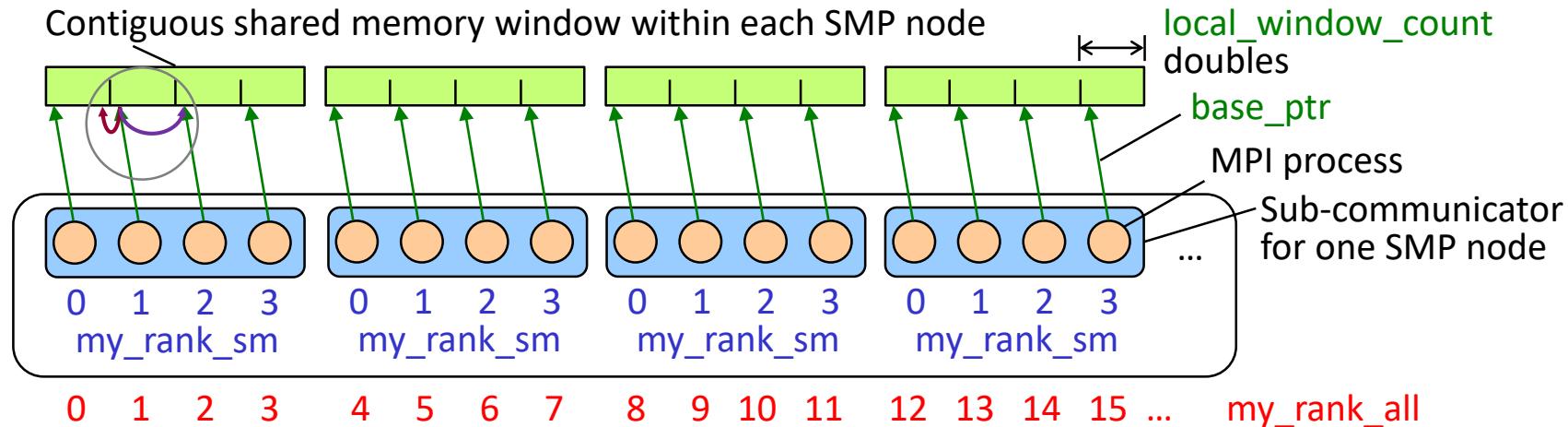
```

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)

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);

Synchronization
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 */

Synchronization
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]);

```

In Fortran, before and after the synchronization, one 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".

Direct load access
to remote window
portion

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>

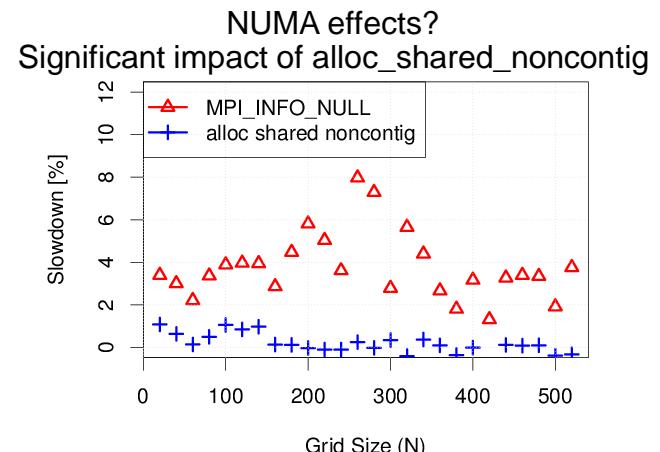
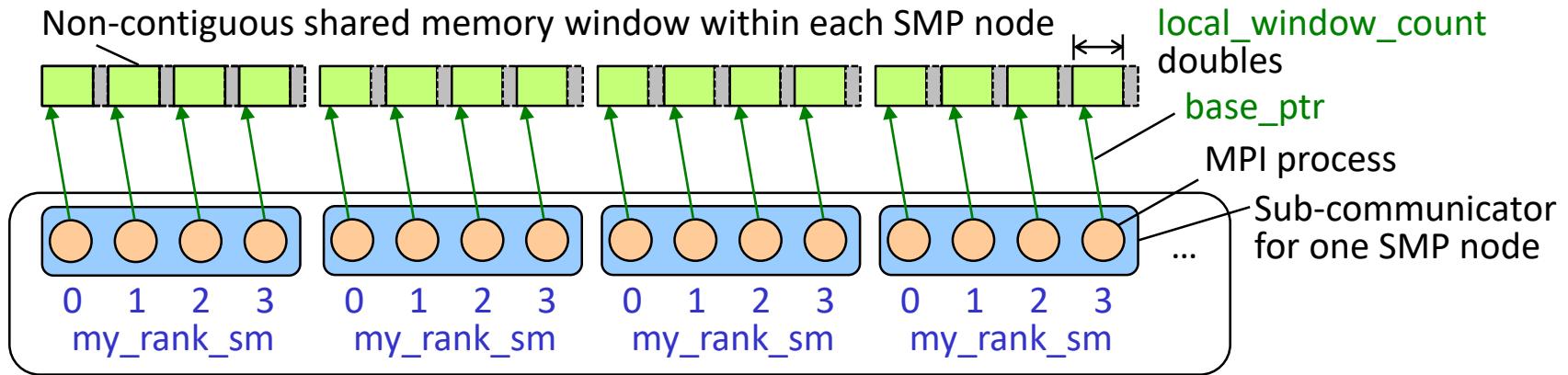


Image: Courtesy of Torsten Hoefler

Non-contiguous shared memory allocation

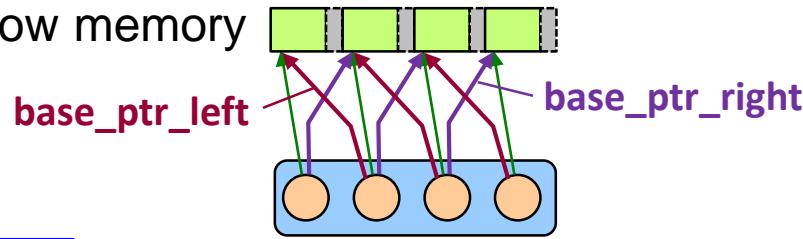


```
MPI_Aint /*IN*/ local_window_count;      double /*OUT*/ *base_ptr;  
disp_unit = sizeof(double); /* shared memory should contain doubles */  
MPI_Info info_noncontig;  
MPI_Info_create (&info_noncontig);  
MPI_Info_set (info_noncontig, "alloc_shared_noncontig", "true");  
MPI_Win_allocate_shared ((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**
- Example: only pointers to the window memory of the left & right neighbor

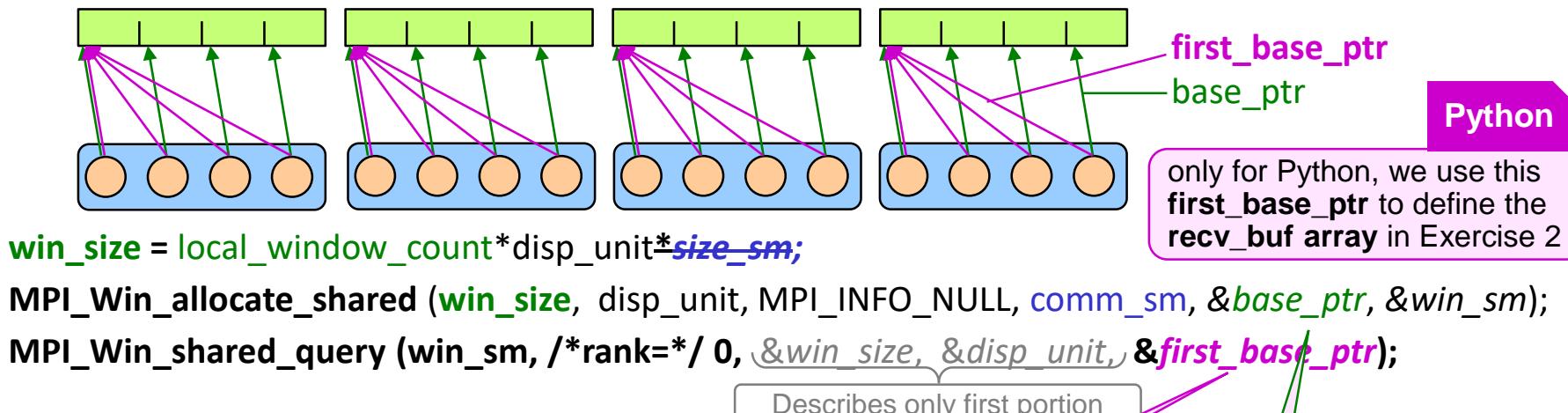
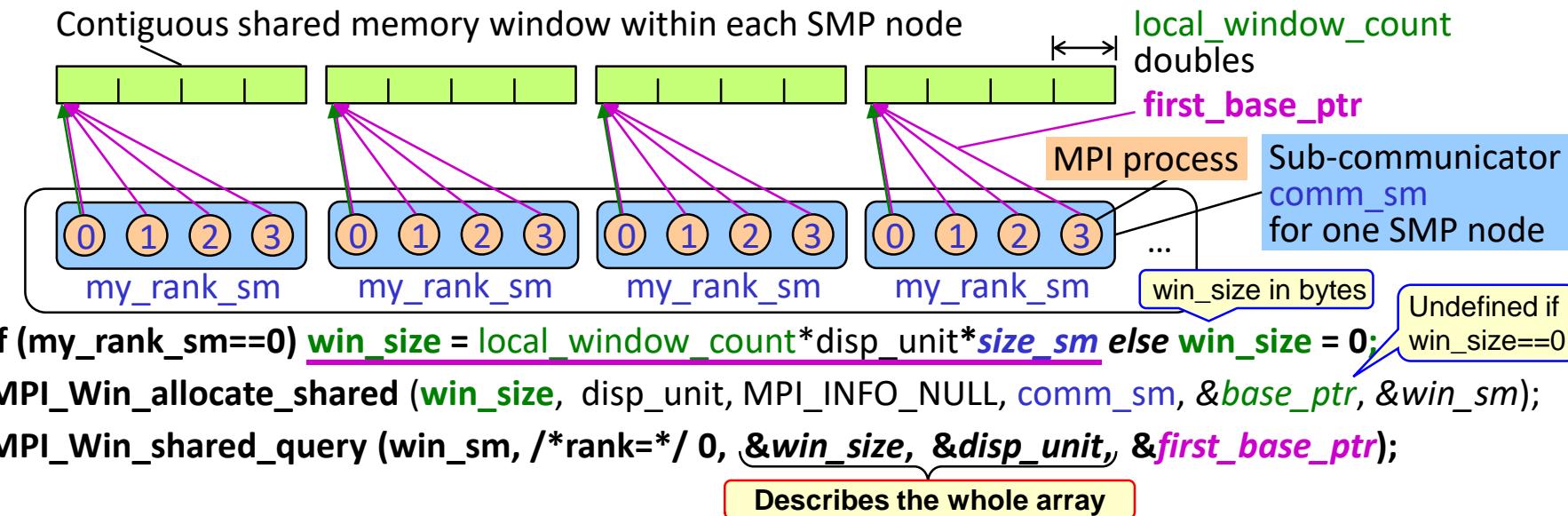
If only one process allocates the whole window
→ to get the base_ptr, all processes call MPI_WIN_SHARED_QUERY



```
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.

Whole shared memory allocation by rank 0 in comm_sm



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 .`

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).

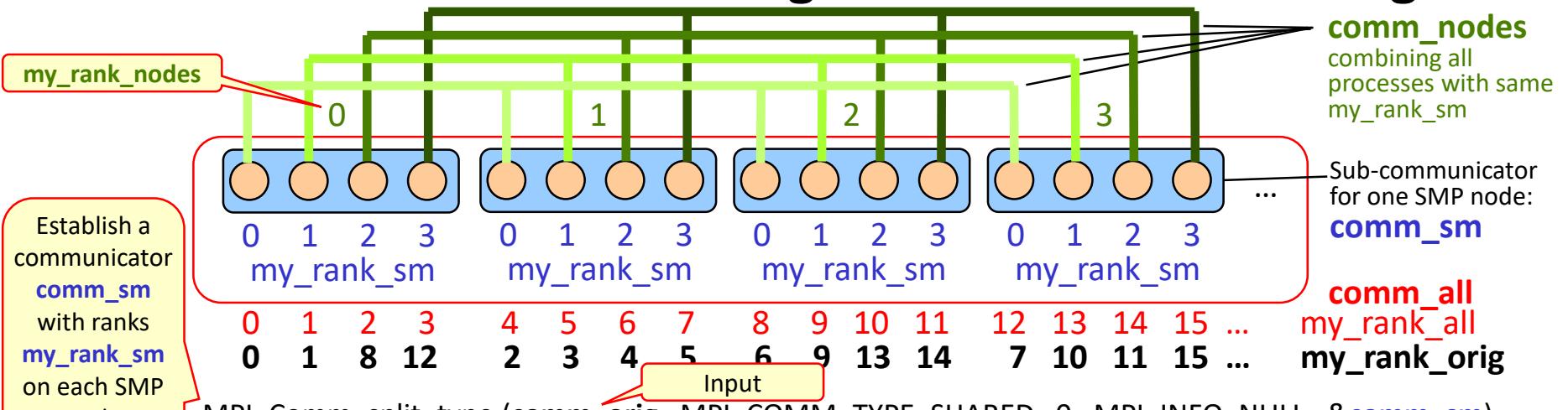
Thanks to Jeff Hammond and Jed Brown (ANL), Brian W Barrett (SANDIA), and Steffen Weise (TU Freiberg), for input and discussion.

Due to default limit of context IDs in mpich

Annex:

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

skipped



```

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); Result: comm_nodes combines all processes with a
MPI_Comm_size (comm_nodes, &size_nodes); given my_rank_sm into a separate communicator.

Exscan does not return value on the first rank, therefore
if (my_rank_sm==0) { On processes with my_rank_sm > 0, this comm_nodes is unused because
    MPI_Comm_rank (comm_nodes, &my_rank_nodes); node-numbering within these comm_nodes may be different.

    MPI_Exscan (&size_sm, &my_rank_all, 1, MPI_INT, MPI_SUM, comm_nodes); Expanding the numbering from
    if (my_rank_nodes == 0) my_rank_all = 0; comm_nodes with my_rank_sm == 0 to all new node-to-node
} my_rank_nodes is not identical to the rank in comm_nodes if node sizes are not identical communicators comm_nodes.

MPI_Comm_free (&comm_nodes); Calculating my_rank_all and
MPI_Bcast (&my_rank_nodes, 1, MPI_INT, 0, comm_sm); establishing global communicator
MPI_Comm_split (comm_orig, my_rank_sm, my_rank_nodes, &comm_nodes); subsets.

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);

```

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 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`!
- 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`



```

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

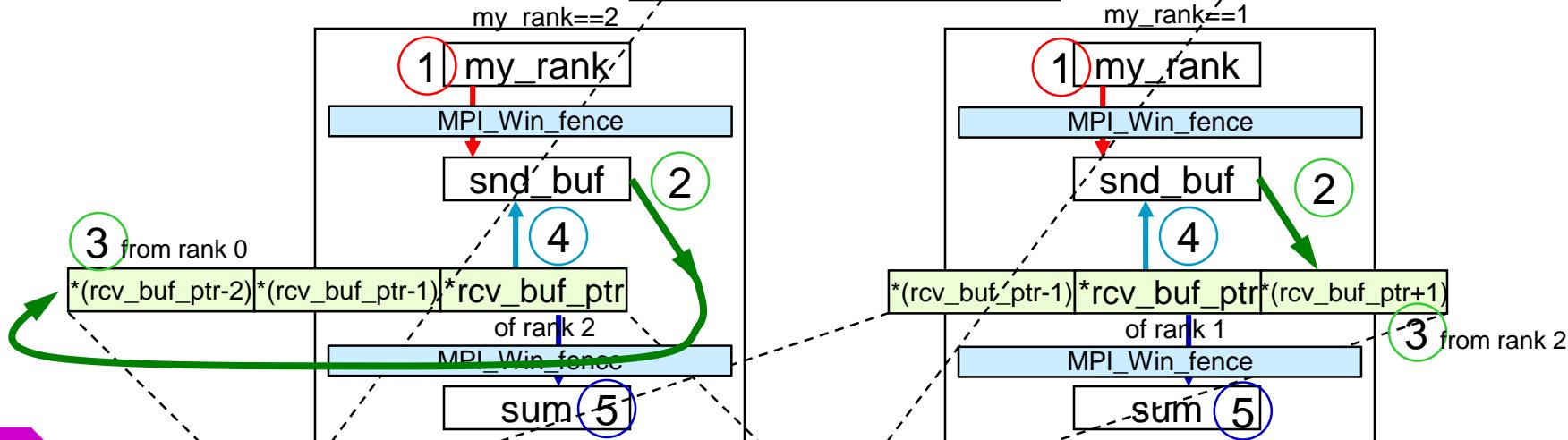
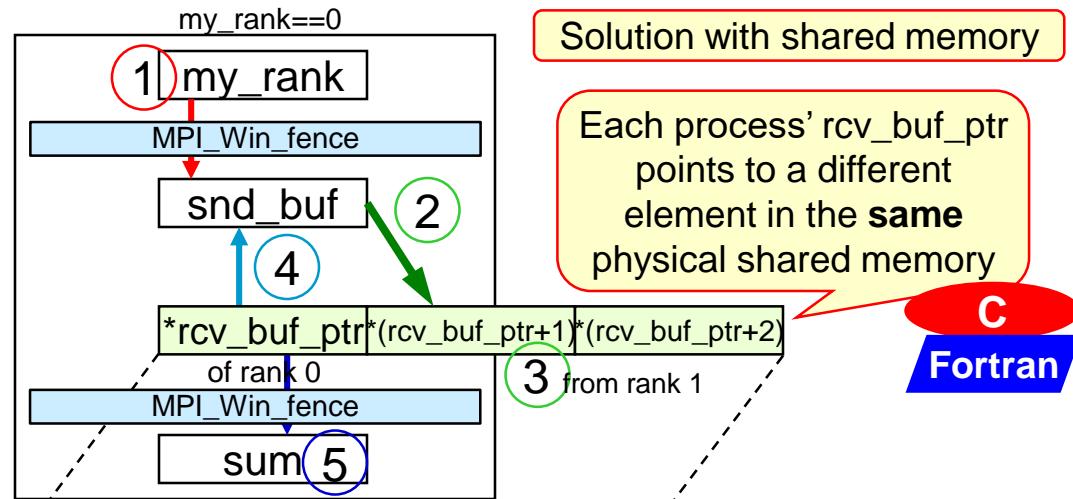
Exercise 2: Shared memory ring communication

Initialization: 1

Each iteration:



to be substituted
by 1-sided shared
memory assignments

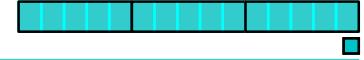


Python

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

*rcv_buf_ptr *rcv_buf_ptr *rcv_buf_ptr all windows in one long shared memory array

During the Exercise (15 min.)



Please stay here in the main room while you do this 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:

You and your colleagues may go to the 2nd next slide: halo benchmarks!

Please, compile & test them on your system.

*For example you may compare **halo_irecv_send** with **halo_1sided_put** and **halo_1sided_store_win_alloc_shared** !*

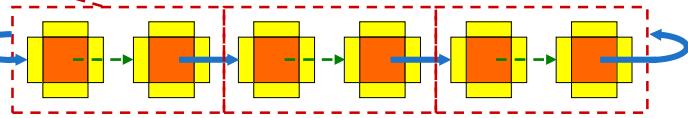
Now, you may compare your systems' result with such of your colleagues



Go to quiz

Exercise 3 (advanced): Mixed ring communication

- Task of this exercise:
 - Use **C** C/Ch11/ring-1sided-mixed-issend-recv-store-skel.c
 - or **Fortran** F_30/Ch11/ring-1sided-mixed-issend-recv-store-skel_30.f90
 - or **Python** PY/Ch11/ring-1sided-mixed-issend-recv-store-skel.py
- The program splits MPI_COMM_WORLD
 - into shared memory islands: comm_sm, and this one
 - into two sub-islands: comm_sm_sub.
- Result: also on a shared memory system, we'll have at least two shared memory islands
- Task:
 - i.e., with world rank and size
 - Implement the original ring communication by using
 - Point-to-point MPI_Issend + MPI_Recv + MPI_Wait
for communication between comm_sm_sub islands
 - - → MPI_Win_fence + shared memory assignments
for communication within comm_sm_sub islands
 - And use 2x MPI_Group_translate_ranks to detect, which communication is needed
- Check for _____ and implement the right decisions
- mpirun -np 8 ./a.out | sort | sed -e 's/_/_g'



Summary of halo benchmark files (and some ring files)

In MPI/tasks/C/halo-benchmarks and MPI/tasks/F_30/halo-benchmarks you find all halo_...



Chapter 11 Shared Mem.,
Part (2), Exercise 5:
Benchmarking all solutions

C unlimit or
ulimit -s 200000
once before calling mpirun

Chapter 11 Shared Mem.,
Exercise 1+2

Chapter 11 Shared Mem.,
Part (2), Exercise 5

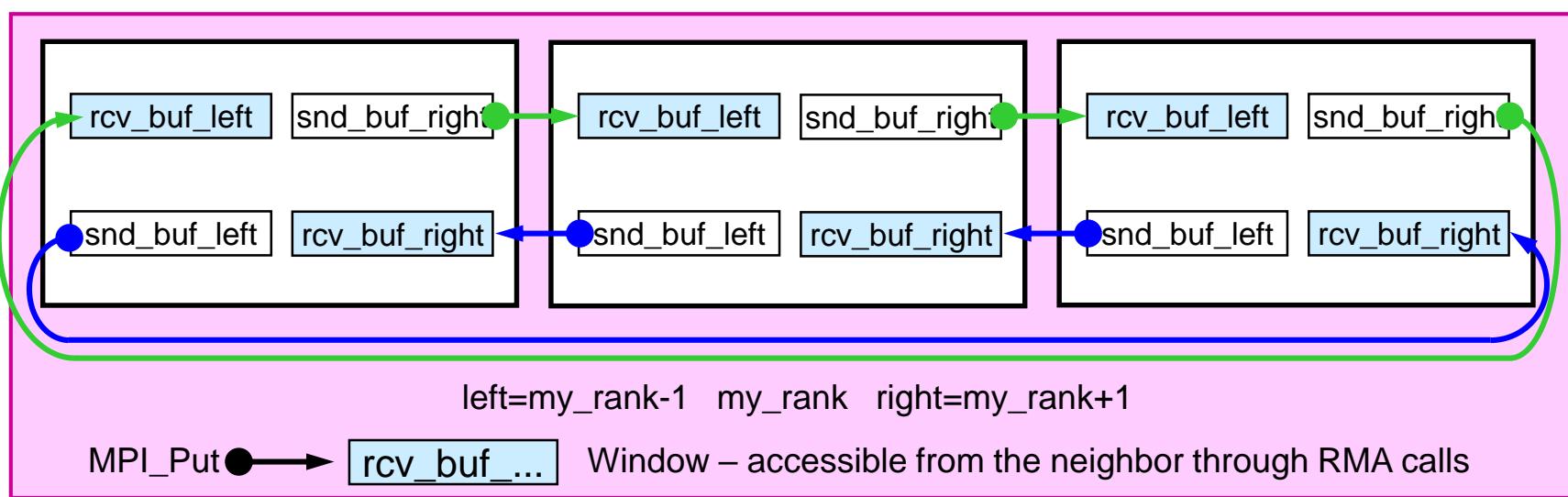
Python Use C and call
unlimit or ulimit -s 200000
once before calling mpirun

Chapter 11 Shared Mem., Part (2), Exercise 6

See HLRS online courses <http://www.hlrs.de/training/par-prog-ws/>
→ Practical → MPI31.tar.gz → MPI/tasks/*/halo-benchmarks/

Exercise 4 (advanced): Halo communication with MPI_Put

- Have a look at this source code:
 - `~/MPI/tasks/C/halo-benchmarks/halo_1sided_put_win_alloc.c`
 - `~/MPI/tasks/F_30/halo-benchmarks/halo_1sided_put_win_alloc_30.f90`
- halo... communicates along the 1-dim ring of processes in both directions
 - ➔ **Into right direction:** Put `snd_buf_right` into the `rec_buf_left` of the right neighbor
 - ⬅ **Into left direction:** Put `snd_buf_left` into the `rec_buf_right` of the left neighbor



- Compile and run the original `halo_1sided_put_win_alloc*.c/f90` program
 - With MPI processes on **4 cores** & **all cores** of a shared memory node

C

unlimit or
ulimit -s 200000
once before calling mpirun

Exercise 4a: Shared memory halo communication

- Have a look at this source code:
 - `~/MPI/tasks/C/halo-benchmarks/halo_1sided_store_win_alloc_shared.c`
 - `~/MPI/tasks/F_30/halo-benchmarks/halo_1sided_store_win_alloc_shared_30.f90`
- Compile and run shared memory program
 - With MPI processes on **4 cores** & **all cores** of a shared memory node
- Compare latency and bandwidth
- Compare the source codes (with “diff”)

C

unlimit or
ulimit -s 200000
once before calling mpirun

skipped

(Exercise 4b: Shared memory **halo** communication)

- Use the given program as your baseline for the following exercise:
 - `cp halo_1sided_put_win_alloc.c my_shared_exa2.c or ..._30.f90`
- Tasks: Substitute the distributed window by a shared window
 - Substitute `MPI_Mem_alloc+MPI_Win_allocate` by `MPI_Win_allocate_shared`
 - Substitute both `MPI_Put` by direct assignments:
 - `rcv_buf_right[i]` of the **left neighbor** can be now accessed directly through own `rcv_buf_right` as `rcv_buf_right[i+offset_left]` with `offset_xxx = (xxx - my_rank) * max_length`.
 - `xxx = left or right`. The formula is correct for any rank.
 - `max_length` is the number of elements in the window of each process.
 - 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...(...+offset_xxx : ...+offset_xxx) = snd_buf...(... : ...)`
- Compile and run shared memory program
 - With MPI processes on **4 cores & all cores** of a shared memory node

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` should be used, but only with `assert = 0`**. (State March 01, 2015)

Exercise 5 (advanced): MPI_Bcast into shared memory

Exercise 5

C
Fortran
Python

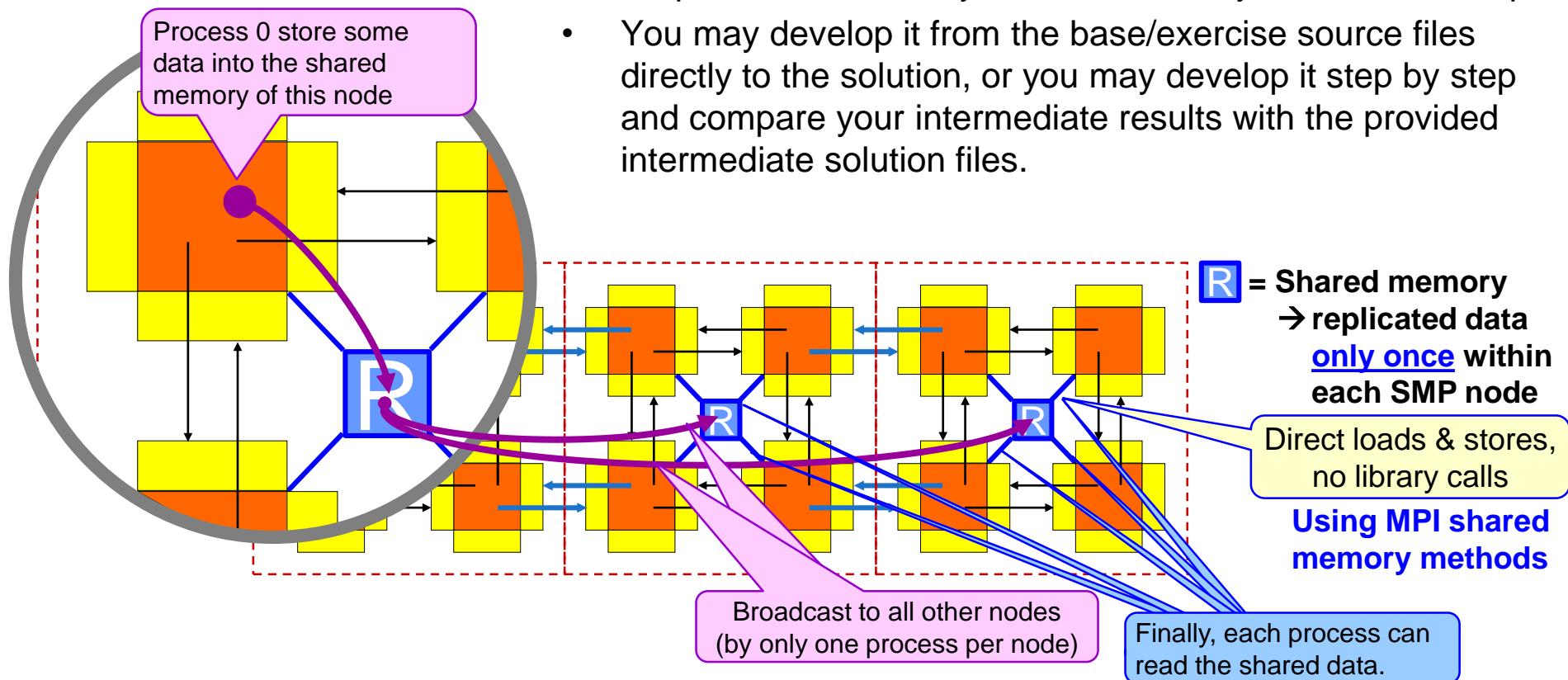
Intro

- This exercise implements the example in the introduction of this chapter: 
- Use ~MPI/tasks/C/Ch11/data-rep/data-rep_base.c & ...data-rep_exercise.c
~MPI/tasks/F_30/Ch11/data-rep/data-rep_base_30.f90 & ..._exercise_30.f90
~MPI/tasks/PY/Ch11/data-rep/data-rep_base.py & ..._exercise.py
as your baseline **my_shared_exa3.c or ..._30.f90 orpy** for the following exercise:
- data-rep_base.c / _30.f90 is the original MPI program:
 - It copies data from the process rank 0 in MPI_COMM_WORLD to all processes.
 - On all processes it uses the data: in this example, just the sum is calculated.
- ..._exercise** is the **skeleton** for using shared memory within a shared memory node.
- Tasks:
 - Copy the original data from rank 0 in MPI_COMM_WORLD into the destination arrays by using only **one** process per shared memory node.
 - By using shared memory, the broadcasted data is everywhere available.
 - Calculate the sum on all processes.
- You may compare your result with our **..._solution.c / ...solution_30.f90 / ...ion.py** file

Exercise developed together with Irene Reichl, VSC, TU Wien, Vienna.

Exercise 5 (advanced): MPI_Bcast into shared memory

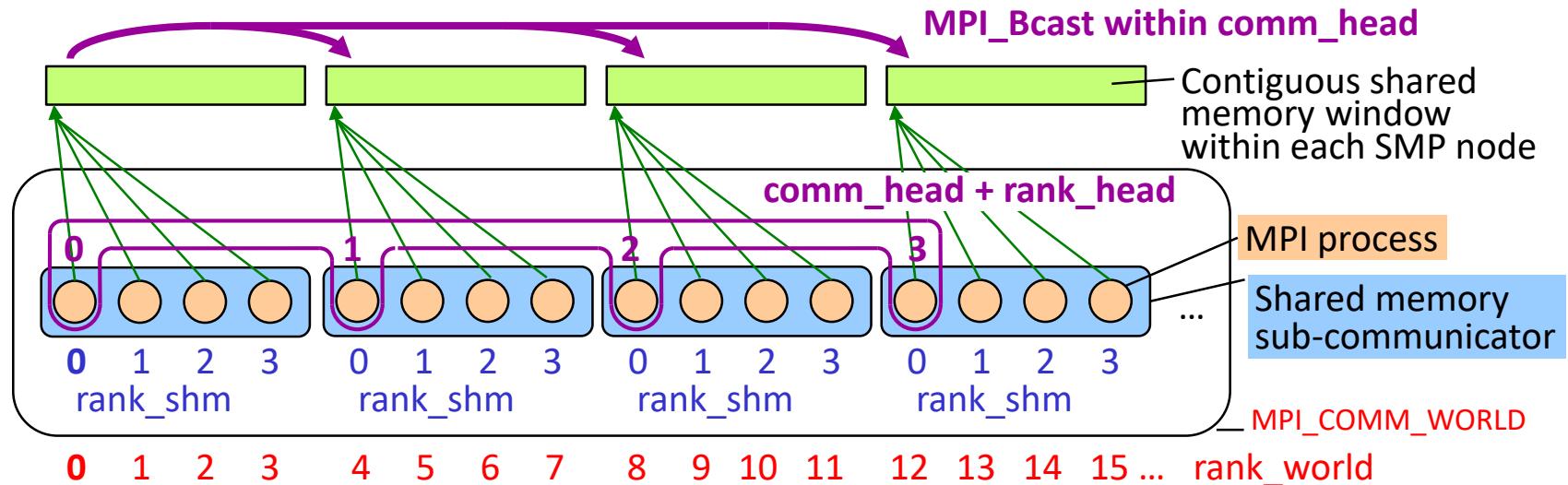
- This exercise is part of our **Hybrid MPI+X** course
- It is **not part of our MPI courses**, but
- we provide it here for you as a self-study exercise / example.
- You may develop it from the base/exercise source files directly to the solution, or you may develop it step by step and compare your intermediate results with the provided intermediate solution files.



Exercise 5 (advanced):

The allocation of the shared memory within each node

- Now illustrated as in the previous slides
- Each  represents such a replicated memory  of an SMP node

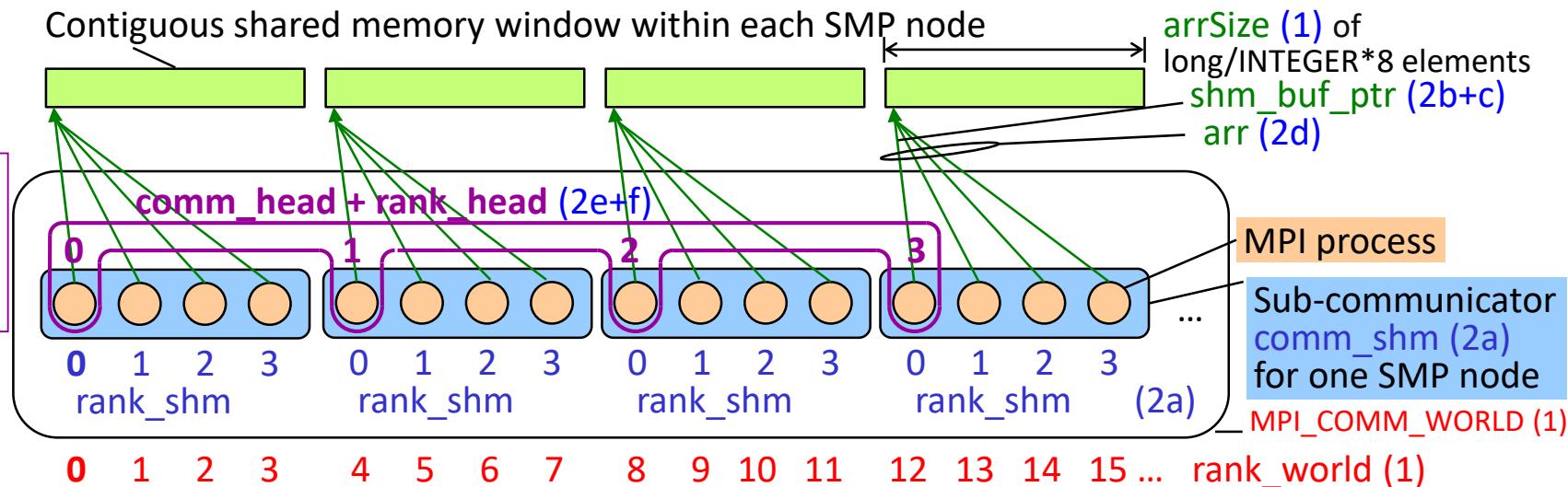


- Application: We'll store numbers 1, 2, ... into the green array by process 0
- And then bcast it to all other shared memories
- At the end, each process calculates the sum of all number within *its* shared memory.

Exercise 5 – the steps 1-2:

The allocation of the shared memory within each node

`rank_world==0
&& key==0
→
rank_head==0
and
rank_shm==0`



1st exercise step
(~5 lines of code
+2 lines printing)

- (1) Given: arrSize, MPI_COMM_WORLD → rank_world
- (2a) MPI_Comm_split_type(key=0) → comm_shm → MPI_Comm_rank() → rank_shm

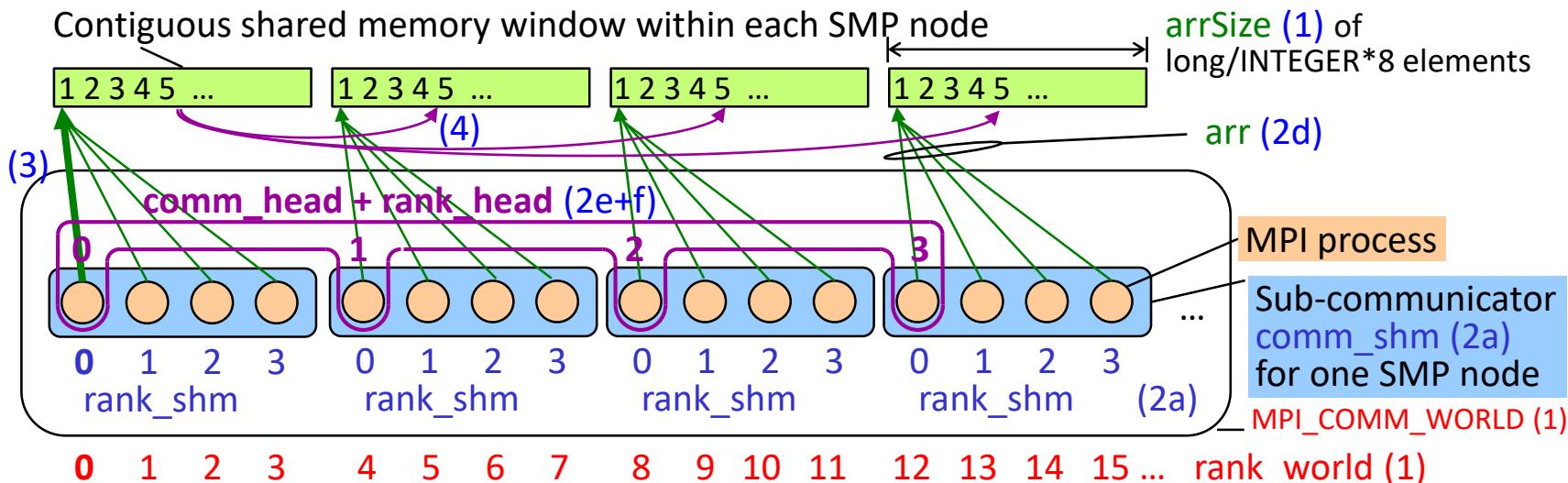
2nd exercise step
(~11 lines of code
+2 lines printing)

- (2b) if (rank_shm == 0) then individualShmSize = arrSize else individualShmSize = 0
- (2c) MPI_Win_allocate_shared (comm_shm → win & shm_base_ptr (but only if rank_shm== 0))
- (2d) MPI_Win_shared_query (win & rank 0 → arr, i.e., the base pointer on all processes);

3rd exercise step
(~12 lines of code
+2 lines printing)

- (2e) if (rank_shm == 0) then color=0 else color=MPI_UNDEFINED
- (2f) MPI_Comm_split(MPI_COMM_WORLD, key=0, color → comm_head) → rank_head
and in all processes with color==MPI_UNDEFINED → MPI_COMM_NULL

Exercise 5 – the steps 3-6: The usage of the shared memory



(3-4) Store epoch: we store the replicated data in all shared memories
(don't forget `MPI_Win_fence()` within all `comm_shm`/win before starting the store epoch for `arr`)

(3) Process with `rank_world==0` stores numbers into 1st green `arr`

(4) All processes in `comm_head` `MPI_Bcast()` the data from `rank_head==0` to all others

(5) Local load epoch: each process reads the data and locally calculates the sum
(don't forget `MPI_Win_fence()` within all `comm_shm` / win before starting the local load epoch)

(6) Print the results

End of time step loop

(7) Finish the local load epoch → `MPI_Win_fence()` // free the window → `MPI_Win_free()`

4th exercise step
(~5 lines of code
+1 lines printing)

5th exercise step
(~1 lines of code)

Exercise 5: MPI_Bcast into shared memory (Preparation 1)

- Directories in your personal account:
 - MPI/tasks/C/Ch11/data-rep/
 - data-rep_base.c
 - data-rep_exercise.c
 - data-rep_base_VSC.sh
 - data-rep_exercise_VSC.sh
 - data-rep_solution_VSC.sh
 - **(already together with all solution files)**

C

Fortran

- MPI/tasks/F_30/Ch11/data-rep/
 - data-rep_base_30.f90
 - data-rep_exercise_30.f90
 - data-rep_base_VSC.sh
 - data-rep_exercise_VSC.sh
 - data-rep_solution_VSC.sh
 - **(already together with all solution files)**

Python

- data-rep_base.c / _30.f90 / .py is the original MPI program
- data-rep_exercise.c / _30.f90 / .py is the basis for this shared memory exercise

MPI/tasks/PY/Ch11/data-rep/

- data-rep_base.py
- data-rep_exercise.py
- data-rep_base_VSC-C-version.sh
- data-rep_exercise_VSC-C-version.sh
- data-rep_solution_VSC-C-version.sh
- **(already together with all solution files)**

Exercise 5: MPI_Bcast into shared memory (Preparation, 10 Minutes)

- data-rep_base.c / _30.f90 is the original MPI program:
 - It copies data from the process rank 0 in MPI_COMM_WORLD to all processes.
 - On all processes it uses the data: in this example, just the sum is calculated.
 - Compile it and run it:
 - mpiicc -o data-rep_base data-rep_base.c
 - mpiifort -o data-rep_base data-rep_base_30.f90
 - sbatch data-rep_base_VSC.sh (will use 4 nodes)
 - sq
 - Output will be written to: **slurm-* .out**
 - Output from only 2 nodes (each with 16 MPI processes) and 3 time steps:

```
it: 0, rank ( world: 31 ): sum(i=0...i=255999999) = 32767999872000000
it: 0, rank ( world: 0 ): sum(i=0...i=255999999) = 32767999872000000
it: 0, rank ( world: 1 ): sum(i=0...i=255999999) = 32767999872000000
it: 1, rank ( world: 0 ): sum(i=1...i=256000000) = 32768000128000000
it: 1, rank ( world: 1 ): sum(i=1...i=256000000) = 32768000128000000
it: 1, rank ( world: 31 ): sum(i=1...i=256000000) = 32768000128000000
it: 2, rank ( world: 0 ): sum(i=2...i=256000001) = 32768000384000000
it: 2, rank ( world: 1 ): sum(i=2...i=256000001) = 32768000384000000
it: 2, rank ( world: 31 ): sum(i=2...i=256000001) = 32768000384000000
```

- 1st time step
- output from 3 processes per communicator:
- ranks 0, 1 & last rank

Exercise 5: MPI_Bcast into shared memory (Step 2a, 15 Minutes)

- ..._exercise is the **skeleton** for using shared memory within a shared memory node
- Step 2a:
 - Declare variables comm_shm, size_shm, rank_shm (2 lines of code)
 - Split MPI_COMM_WORLD into shared memory island communicators comm_shm (use key == 0) (1 line of code)
 - Query size_shm, rank_shm (2 lines of code)
 - After this splitting: print and stop (3 lines of code, copy print statement from end of your source file)

```
/*TEST*// To minimize the output, we print only from 3 process per SMP node
/*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
    printf("\t\t rank ( world: %i, shm: %i)\n", rank_world, rank_shm);
/*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!.\n"); MPI_Finalize(); return 0;
```

- Expected output from 2 SMP nodes:

rank (world: 0, shm: 0)
ALL finalize and return !!!.
rank (world: 16, shm: 0)
rank (world: 1, shm: 1)
rank (world: 17, shm: 1)
rank (world: 15, shm: 15)
rank (world: 31, shm: 15)

- After ~10 Minutes, in the solution directory: data-rep_sol_2a.c / _30.f90
- Q & A & Discussion

Exercise 5: MPI_Bcast into shared memory (Steps 2b-d, 20 Minutes)

- Steps 2b-d:
 - Declare needed variables (5 LOC)
 - (2b) `if (rank_shm == 0) then individualShmSize = arrSize else individualShmSize = 0` (4 LOC)
 - (2c) `MPI_Win_allocate_shared (comm_shm → win & shm_base_ptr` (but only if rank_shm== 0)) (1 LOC)
 - (2d) `MPI_Win_shared_query (win & rank 0 → arr`, i.e., the base pointer on all processes); (1 LOC)
 - After this splitting: print and stop (3 lines of code)
 - Expected output from 2 SMP nodes:

rank (world: 0, shm: 0) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = 0x2b916e509000, arr_ptr = 0x2b916e509000
ALL finalize and return !!!.

rank (world: 16, shm: 0) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = 0x2aacb082c000, arr_ptr = 0x2aacb082c000
rank (world: 1, shm: 1) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr ≠ (nil), arr_ptr = 0x2adafb2ca000
rank (world: 17, shm: 1) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = (nil), arr_ptr = 0x2aca6d8c9000
rank (world: 15, shm: 15) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = (nil), arr_ptr = 0x2b90d94ff000
rank (world: 31, shm: 15) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = (nil), arr_ptr = 0x2b00ed3db000

- After ~10 Minutes, in the solution directory: data-rep_sol_2d.c / _30.f90
- Q & A & Discussion

Output from
1st SMP node
2nd SMP node

Processes with `individualShmSize = 0`,
do not get a buffer pointer from
`MPI_Win_allocate_shared`

Each process within an SMP
node has **different virtual
addresses** for the **same**
shared memory array

Exercise 5: MPI_Bcast into shared memory (Steps 2e-f, 15 Minutes)

- Steps 2e-f:
 - Declare needed variables (3 LOC)
 - (2e) `if (rank_shm == 0) then color=0 else color=MPI_UNDEFINED` (2 LOC)
 - (2f) `MPI_Comm_split(MPI_COMM_WORLD, key=0, color → comm_head) → rank_head` (8 LOC)
and in all processes with color==MPI_UNDEFINED → MPI_COMM_NULL
 - After this splitting: print and stop (3 LOC)
 - Expected output from 2 SMP nodes:

```
rank ( world: 0, shm: 0, head: 0) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = 0x2ba944547000, arr_ptr = 0x2ba944547000  
ALL finalize and return !!!
```

```
rank ( world: 1, shm: 1, head: -1) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = (nil), arr_ptr = 0x2b591467d000
```

```
rank ( world: 16, shm: 0, head: 1) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = 0x2ab7ca10c000, arr_ptr = 0x2ab7ca10c000
```

```
rank ( world: 15, shm: 15, head: -1) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = (nil), arr_ptr = 0x2b78df3fb000
```

```
rank ( world: 17, shm: 1, head: -1) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = (nil), arr_ptr = 0x2ba1d1aef000
```

```
rank ( world: 31, shm: 15, head: -1) arrSize 256000000 arrSize_ 2048000000 shm_buf_ptr = (nil), arr_ptr = 0x2b1ae970e000
```

- After ~10 Minutes, in the solution directory: data-rep_sol_2f.c / _30.f90
- Q & A & Discussion

Exercise 5: MPI_Bcast into shared memory (Steps 3-6, 15 Minutes)

- Steps 3-6 (6 lines of code)

(3-4) **Store epoch:** we store the replicated data in all shared memories

(don't forget **MPI_Win_fence()** within all **comm_shm/win** before starting the store epoch for **arr**)

(3) Process with rank_world==0 **stores numbers** into 1st green **arr**

(4) All processes in comm_head **MPI_Bcast()** the data from rank_head==0 to all others

(5) **Local load epoch:** each process reads the data and locally **calculates the sum**

(don't forget **MPI_Win_fence()** within all **comm_shm / win** before starting the local load epoch)

(6) **Print the results**

- Expected output from 2 SMP nodes:

it: 0, rank (world: 0, shm: 0, head: 0): sum(i=0...i=255999999) = 32767999872000000

it: 0, rank (world: 16, shm: 0, head: 1): sum(i=0...i=255999999) = 32767999872000000

it: 0, rank (world: 17, shm: 1, head: -1): sum(i=0...i=255999999) = 32767999872000000

it: 0, rank (world: 1, shm: 1, head: -1): sum(i=0...i=255999999) = 32767999872000000

it: 0, rank (world: 31, shm: 15, head: -1): sum(i=0...i=255999999) = 32767999872000000

it: 0, rank (world: 15, shm: 15, head: -1): sum(i=0...i=255999999) = 32767999872000000

it: 1, rank (world: 0, shm: 0, head: 0): sum(i=1...i=256000000) = 32768000128000000

it: 1, rank (world: 1, shm: 1, head: -1): sum(i=1...i=256000000) = 32768000128000000

...

it: 2, rank (world: 0, shm: 0, head: 0): sum(i=2...i=256000001) = 32768000384000000

ALL finalize and return !!!.

it: 2, rank (world: 1, shm: 1, head: -1): sum(i=2...i=256000001) = 32768000384000000

...

Same data in
the shared
memory arrays
of both SMP
nodes

Same data also
in 2nd time step

Same on all
processes also
in 3rd time step

- After ~10 Minutes, in the solution directory: data-rep_sol_3-6.c / _30.f90

- Q & A & Discussion

Exercise 5: MPI_Bcast into shared memory (Step 7, 15 Minutes)

- Step 7 (6 lines of code)

(7) Finish the local load epoch → **MPI_Win_fence()** // free the window → **MPI_Win_free()**

- Expected output from 2 SMP nodes (**same as after Step 6, but now without premature stop**):

```
it: 0, rank ( world: 0, shm: 0, head: 0 ):    sum(i=0...i=255999999) = 32767999872000000
it: 0, rank ( world: 16, shm: 0, head: 1 ):    sum(i=0...i=255999999) = 32767999872000000
it: 0, rank ( world: 17, shm: 1, head: -1 ):   sum(i=0...i=255999999) = 32767999872000000
it: 0, rank ( world: 1, shm: 1, head: -1 ):    sum(i=0...i=255999999) = 32767999872000000
it: 0, rank ( world: 31, shm: 15, head: -1 ):  sum(i=0...i=255999999) = 32767999872000000
it: 0, rank ( world: 15, shm: 15, head: -1 ):  sum(i=0...i=255999999) = 32767999872000000
it: 1, rank ( world: 0, shm: 0, head: 0 ):     sum(i=1...i=256000000) = 32768000128000000
it: 1, rank ( world: 1, shm: 1, head: -1 ):    sum(i=1...i=256000000) = 32768000128000000
...
it: 2, rank ( world: 0, shm: 0, head: 0 ):     sum(i=2...i=256000001) = 32768000384000000
ALL finalize and return !!!
it: 2, rank ( world: 1, shm: 1, head: -1 ):    sum(i=2...i=256000001) = 32768000384000000
```

- After ~5 Minutes, in the solution directory: data-rep_sol_7.c / _30.f90
- And add-on: data-rep_solution.c / _30.f90 with additional analysis and output:
 - The number of shared memory islands is: 2 islands
 - The size of all shared memory islands is: 16 processes
- Q & A & Discussion

Quiz on Chapter 11-(1) – Shared Memory

A. Before you call **MPI_Win_allocate_shared**, what should you do?

B. If your communicator within your shared memory island consists of 12 MPI processes, and each process wants to get an own window with 10 doubles (each 8 bytes),

a. which **window size** must you specify in **MPI_Win_allocate_shared**?

b. And how long is the totally allocated shared memory?

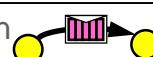
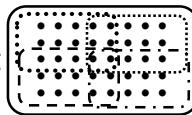
c. The returned base_ptr, will it be identical on all 12 processes?

d. If all 12 processes want to have a pointer that points to the beginning of the totally allocated shared memory, which MPI procedure should you use and with which major argument?

e. If you do this, do these 12 pointers have identical values, i.e., are identical addresses?

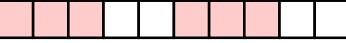
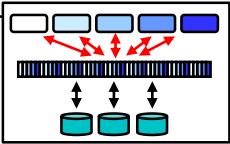
C. Which is the major method to store data from one process into the shared memory window portion of another process?

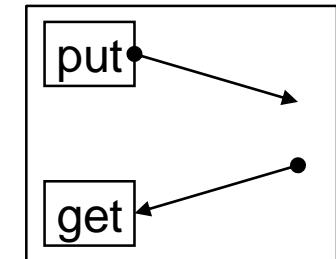
Chap.11 Shared Memory One-sided Communication

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 
10. 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

12. Derived datatypes 
13. Parallel file I/O 
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



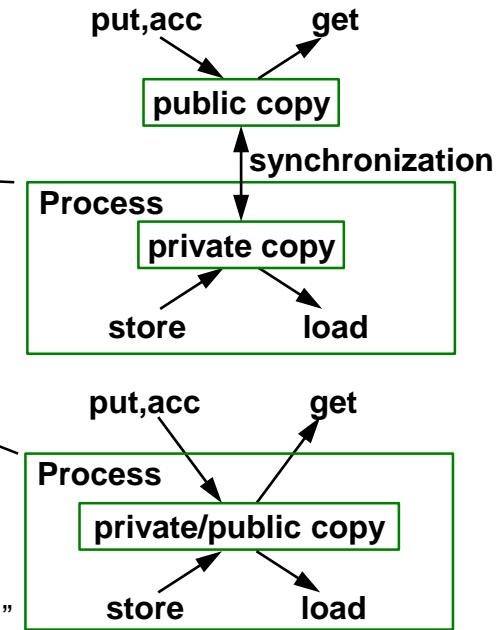
A key feature for
strong scaling?

Lowest latencies

- Usage of MPI shared memory without one-sided synchronization methods
- MPI provides the shared memory, but used
 - only with compiler generated loads & stores
 - together with C++11 memory fences

Two memory models

- Query for new attribute to allow applications to tune for cache-coherent architectures
 - Attribute MPI_WIN_MODEL with values
 - MPI_WIN_SEPARATE model
 - MPI_WIN_UNIFIED model on cache-coherent systems
- Shared memory windows always use the MPI_WIN_UNIFIED model
 - Public and private copies are **eventually** synchronized without additional RMA calls
(MPI-3.1/-4.0, Section 11/12.4, page 435/592 lines 43-46/42-45)
 - For synchronization **without delay**: MPI_WIN_SYNC()
(MPI-3.1/-4.0 Section 11/12.7: "Advice to users. In the unified memory model..." in U5 on page 456/613f, and Section 11/12.8, Example 11/12.21 on pages 468f/626f)
 - or any other RMA synchronization:
"A consistent view can be created in the unified memory model (see Section 11.4) by utilizing the window synchronization functions (see Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH)."
(MPI-3.1/-4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26)



“eventually synchronized“ – the Problem

- The problem with shared memory programming using libraries is:

X is a variable in a shared window initialized with 0.

Process

Rank 0

X = 1

Process

Rank 1

MPI_Send(empty msg to rank 1) → MPI_Recv(from rank 0)

printf ... X

Or any other
process-to-
process
synchronization,
e.g., using also
shared memory
stores and loads

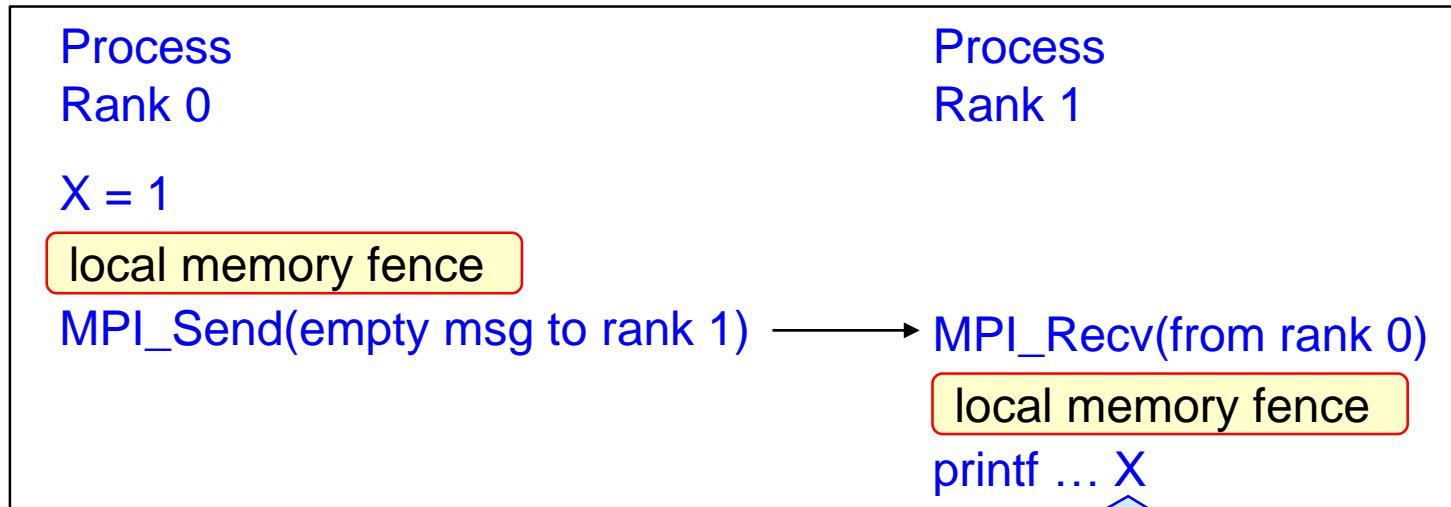
X can be still 0,

because the “1” will eventually be visible to the other process,
i.e., the “1” will be visible but maybe too late ☹ ☹ ☹

“eventually synchronized” – the Solution

- A pair of local memory fences is needed:

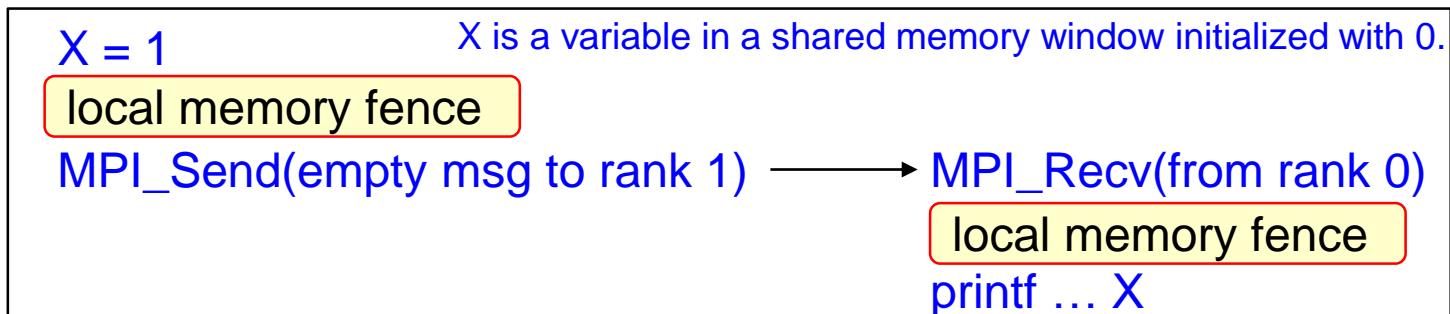
X is a variable in a shared window initialized with 0.



Now, it is guaranteed that
the “1” in X is visible in this process

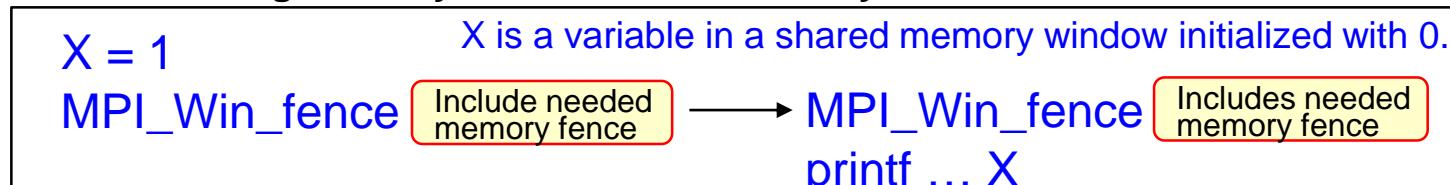


“eventually synchronized“ – Last Question



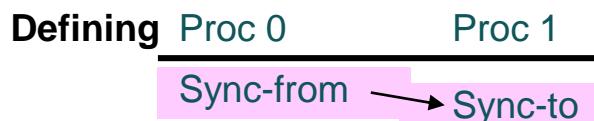
How to make the local memory fence ?

- C11 atomic_thread_fence(order)
 - **Advantage:** one can choose appropriate order = memory_order_acquire, or ..._release to achieve minimal latencies
 - MPI_Win_sync
 - **Advantage:** works also for Fortran
 - **Disadvantage:** may be slower than C11 atomic_thread_fence with appro. order
 - Using RMA synchronization with integrated local memory fence instead of MPI_Send → MPI_Recv
 - **Advantage:** May prevent double fences
 - **Disadvantage:** The synchronization itself may be slower
- } 5 sync methods,
see next slide



General MPI shared memory synchronization rules

(based on MPI-3.1/-4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: “A consistent view ...”)



being MPI_Win_post¹⁾ → MPI_Win_start¹⁾
or MPI_Win_complete¹⁾ → MPI_Win_wait¹⁾
or MPI_Win_fence¹⁾ ←→ MPI_Win_fence¹⁾
or MPI_Win_sync
Any-process-sync²⁾ → Any-process-sync²⁾
MPI_Win_sync
or³⁾ MPI_Win_unlock¹⁾ → MPI_Win_lock¹⁾

and A, B, C are shared variables

and having ...

then it is **guaranteed** that ...

A=val_1
Sync-from → Sync-to
load(A) } ⇒ ... the load(A) in P1 loads val_1
(this is the write-read-rule)

load(B)
Sync-from → Sync-to
B=val_2 } ⇒ ... the load(B) in P0 is not affected by the store of val_2 in P1
(read-write-rule)

C=val_3
Sync-from → Sync-to
C=val_4
load(C) } ⇒ ... that the load(C) in P1 loads val_4
(write-write-rule)

[See next slide]

¹⁾ Must be paired according to the general one-sided synchronization rules.

²⁾ "Any-process-sync" may be done with methods from MPI (e.g. with send-->recv as in MPI-3.1/-4.0 Example 11/12.21, but also with some synchronization through MPI shared memory loads and stores, e.g. with C++11 atomic loads and stores).

³⁾ No rule for MPI_Win_flush (according current forum discuss.)

“Any-process-sync” & MPI_Win_sync on shared memory

- If the shared memory data transfer is done without RMA operation, then the synchronization can be done by other methods.
- This example demonstrates the rules for the unified memory model if the **data transfer** is implemented **only with load and store** (instead of MPI_Get or MPI_Put)
- and the **synchronization** between the processes is done **with MPI communication** (instead of RMA synchronization routines).

X is part of a shared memory window and should be **the same** memory location in **both processes**.

Process A
MPI_WIN_LOCK_ALL(
MPI_MODE_NOCHECK,win)
DO ...
X=...

A new value is **written** in X
MPI_F_SYNC_REG(X)¹⁾
MPI_Win_sync(win)
MPI_Send

Message telling that X is filled

Message telling that X is read out and can be refilled

MPI_Recv
MPI_Win_sync(win)
MPI_F_SYNC_REG(X)¹⁾

END DO

Process B
MPI_WIN_LOCK_ALL(
MPI_MODE_NOCHECK,win)
DO ...

MPI_Recv
MPI_Win_sync(win)
MPI_F_SYNC_REG(X)¹⁾
local_tmp = X
X is **read out**

MPI_F_SYNC_REG(X)¹⁾
MPI_Win_sync(win)
MPI_Send

print local_tmp

END DO

→ See Exercise 3

For MPI_WIN_SYNC, a passive target epoch is established with MPI_WIN_LOCK_ALL.

Data exchange in this direction, therefore **MPI_Win_sync** is needed in both processes:
Write-read-rule

MPI_WIN_SYNC acts only locally as a processor-memory-fence.

2nd pair of MPI_Win_sync is needed to guarantee the **read-write-rule**

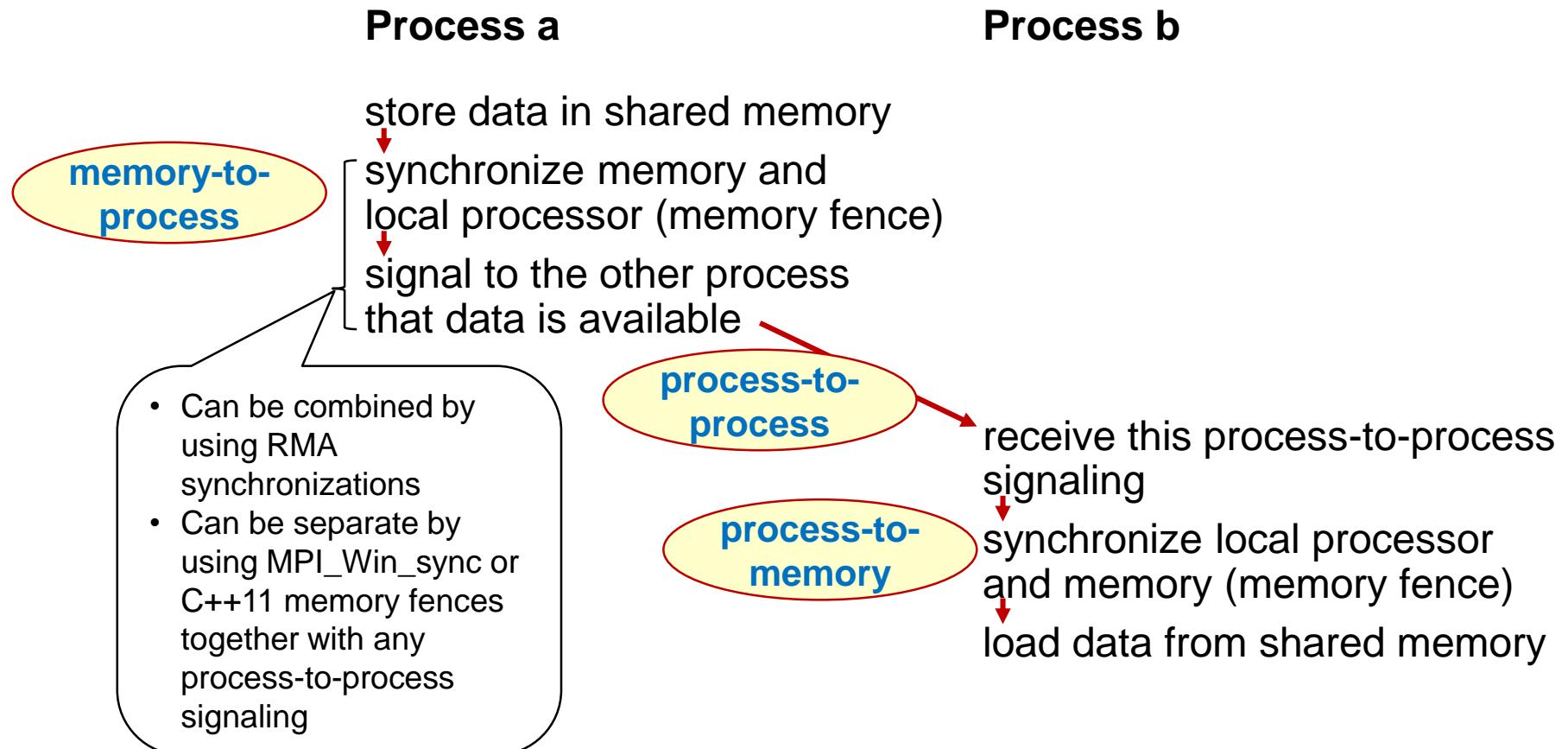
At begin of next iteration:
Next **write** of X

¹⁾ Fortran only.

Is missing in MPI-3.1/-4.0,
pages 468f/626f, Exa. 11/12.21
(i.e., page 469/627, line 31/14)

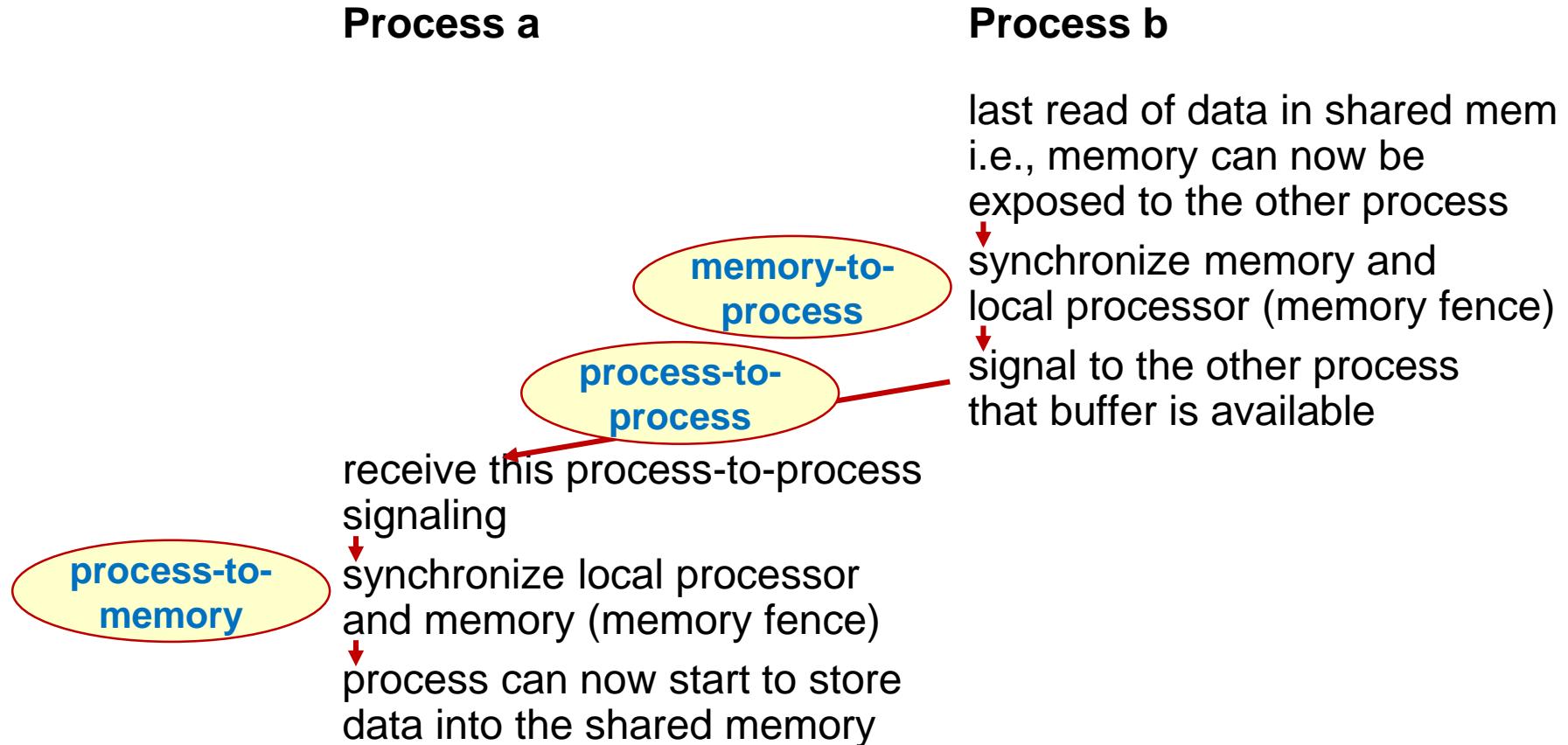
Shared memory synchronization summary (1/2)

- Write-read-rule in general



Shared memory synchronization summary (2/2)

- Read-write-rule: Memory fence is also needed when **exposing a buffer**



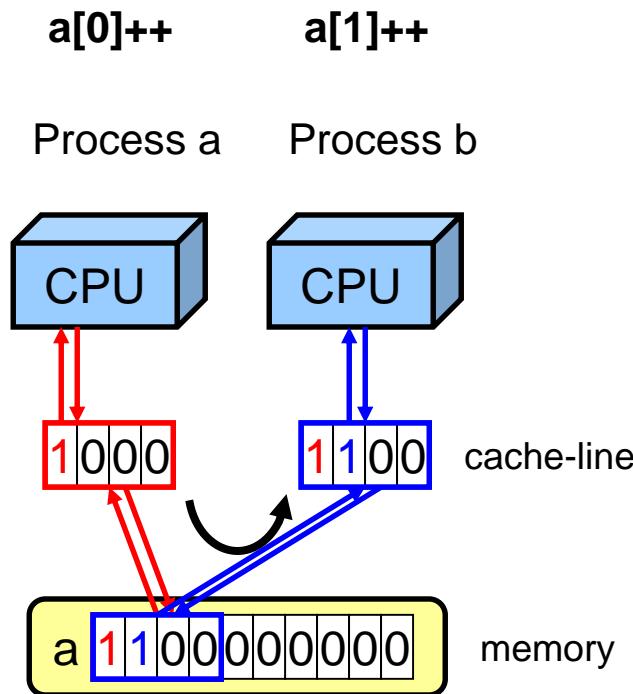
Shared memory problems (1/2)

- **Race conditions**
 - as with OpenMP or any other shared memory programming models
 - Data-Race: *Two processes access the same shared variable and at least one process modifies the variable and the accesses are concurrent, i.e. unsynchronized, i.e., it is not defined which access is first*
 - The outcome of a program depends on the detailed timing of the accesses
 - This is often caused by unintended access to the same variable, or missing memory fences

Shared memory problems (2/2)

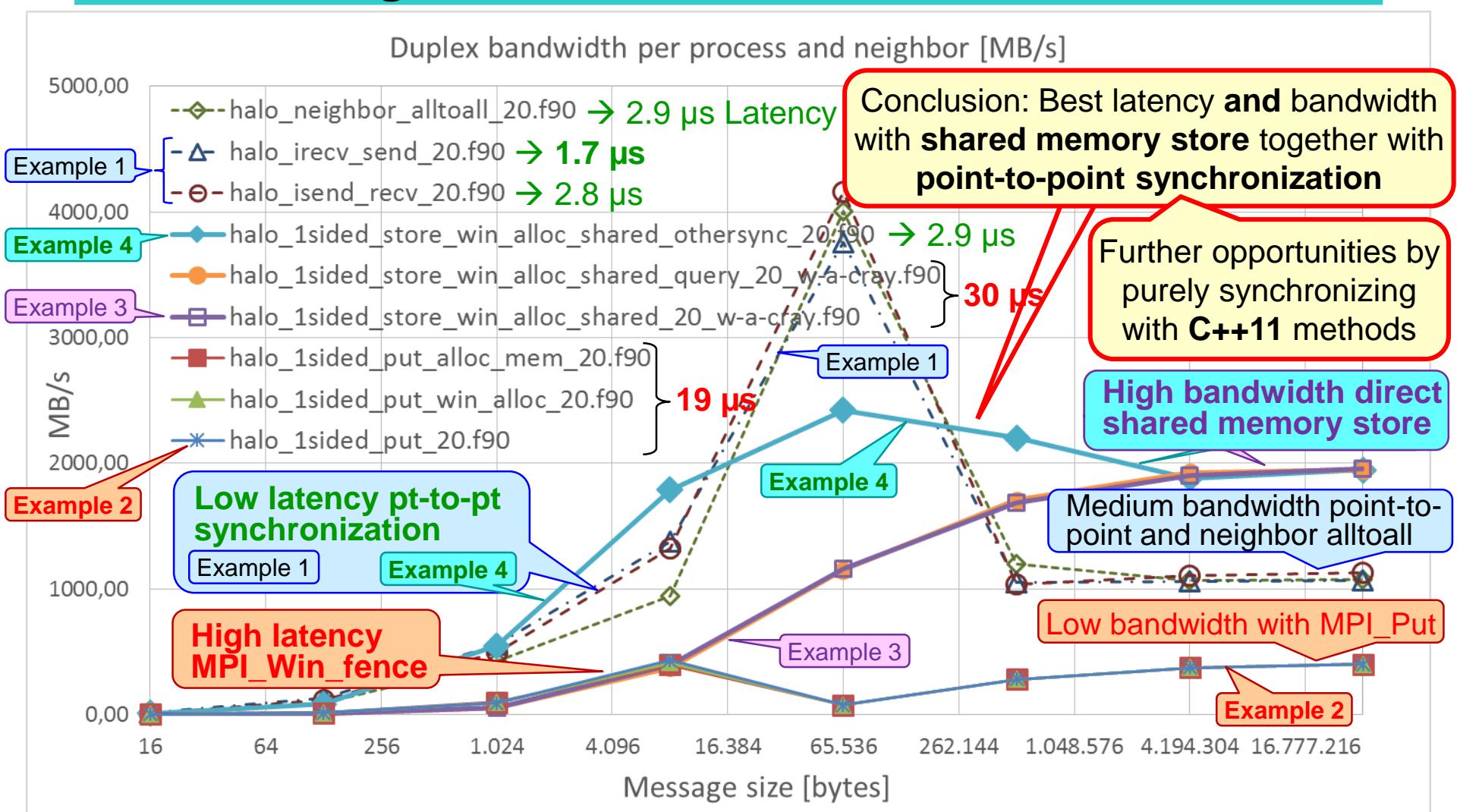
- **Cache-line false-sharing**

- As with OpenMP or any other shared memory programming models
- The cache-line is the smallest entity usually accessible in memory



- Several processes are accessing shared data through the same cache-line.
- This cache-line has to be moved between these processes (cache coherence protocol).
- This is very time-consuming.

MPI Communication inside of the SMP nodes: Benchmark results on a Cray XE6 – 1-dim ring communication on 1 node with 32 cores



On Cray XE6 Hermit at HLRS with aprun -n 32 -d 1 -ss, best values out of 6 repetitions, modules PrgEnv-cray/4.1.40 and cray-mpich2/6.2.1

MPI shared memory – a Summary

- Shared Memory was introduced in MPI-3.0.
- MPI shared memory can be used to **significantly reduce** the memory needs for **replicated data**.
- It is an opportunity to omit unnecessary communication inside of shared memory or ccNUMA nodes.
- Direct memory access may have best bandwidth compared to other MPI communication methods.
- Direct memory access can be combined with fast synchronization methods.
- Which communication option is the fastest?

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming

but

efficiency of MPI application-programming is **not portable!**

Further reading on shared memory synchronization

- Wikipedia: Memory barrier. https://en.wikipedia.org/wiki/Memory_barrier
- Wikipedia: Runtime memory ordering
https://en.wikipedia.org/wiki/Memory_ordering#Runtime_memory_ordering
(and courtesy to Dave Goodell):
- Paul E. McKenney (ed.). Is Parallel Programming Hard, And, If So, What Can You Do About It?
First Edition, Linux Technology Center, IBM Beaverton, March 10, 2014.
<http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e1.pdf>

On compiler optimization problems (courtesy to Bill Gropp):

- Hans-J. Boehm. Threads Cannot be Implemented as a Library.
HP Laboratories Palo Alto, report HPL-2004-2092004, 2004.
<http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf>
- Sarita V. Adve, Hans-J. Boehm:
You don't know Jack About Shared Variables or Memory Models.
<http://queue.acm.org/detail.cfm?id=2088916>

Exercise 6 (advanced): Ring – Using other synchronization

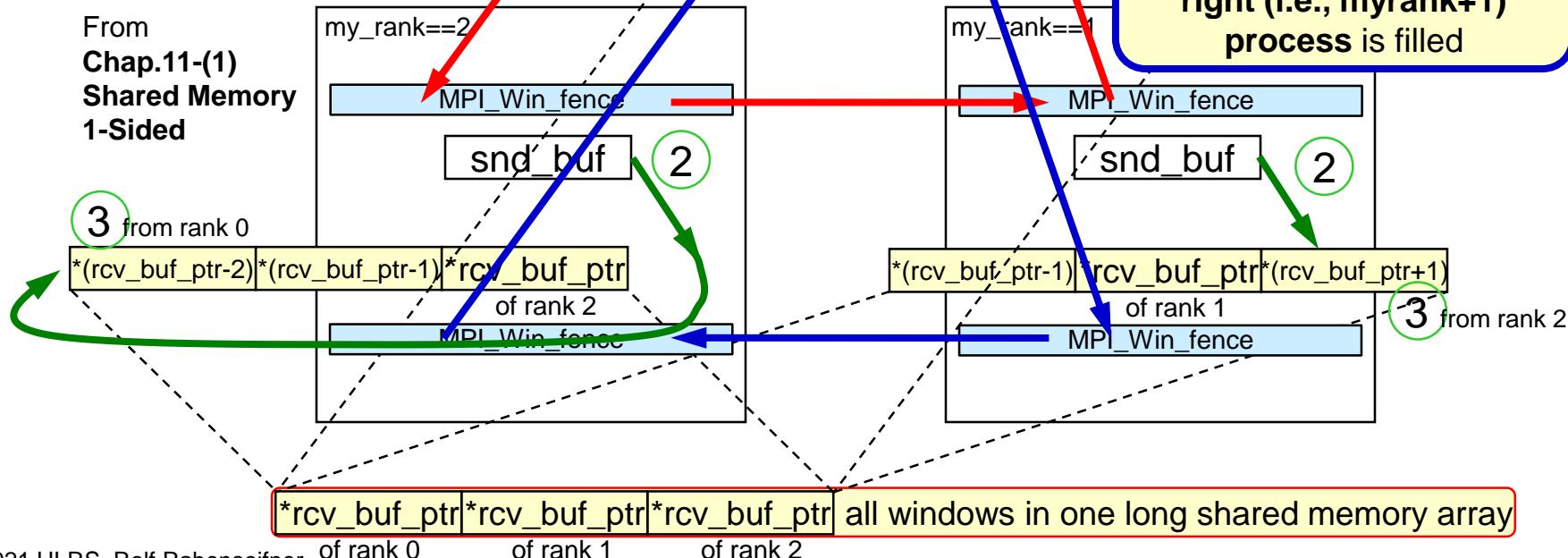
- Use **C** C/Ch11/ring-1sided-store-win-alloc-shared-**othersync**-skel.c
or **Fortran** F_30/Ch11/ring-1sided-store-win-alloc-shared-**othersync**-skel_30.f90
or **Python** PY/Ch11/ring-1sided-store-win-alloc-shared-**othersync**-skel.py
 - which should be identical to your result of 11-(1) Exercises 2.
- Tasks: Substitute the MPI_Win_fence synchronization by pt-to-pt communication
 - Use empty messages for synchronizing
 - Substitute the first MPI_Win_fence by ring-communication to the **left**, because it signals to the **left** neighbor that the local rcv_buf target is exposed for new data
 - `MPI_Irecv(...right,...,&rq); MPI_Send(...left, ...); MPI_Wait(&rq ...);`
 - Substitute the second Win_fence by ring-communication to the **right**, because it signals to the **right** neighbor that data is stored in the rcv_buf of the right neighb.
 - Local **MPI_Win_sync** is needed for write-read and read-write-rule
 - Requires (once, before the loop) `MPI_Win_lock_all(MPI_MODE_NOCHECK, win);`
(and once after the loop) `MPI_Win_unlock_all(win);`
- Compile and run this program on 4 cores & **all cores** of a shared memory node.

Exercise 6 (advanced): Ring – Using other synchronization — shared memory with fast pt-to-pt synchronization

To do:

Substitute both
`MPI_Win_fence()`
 by
`MPI_Irecv(...)`
`MPI_Send(...)`
`MPI_Wait(...)`
 into the **correct** direction:
`my_rank -1` and `+1`

From
Chap.11-(1)
Shared Memory
1-Sided



Exercise 6 (advanced): Ring – Using other synchronization

- Communication pattern between each pair of neighbor processes

Outside of the loop:
MPI_Win_lock_all(...);

Fortran: IF(..) CALL
 MPI_F_sync_reg(rcv_buf)
MPI_Win_sync(win);
MPI_Irecv(&dummy,...,
 right,...);
MPI_Send (&dummy,...,
 left,...);
MPI_Wait(...);
MPI_Win_sync(win);
Fortran: IF(..) CALL
 MPI_F_sync_reg(rcv_buf)

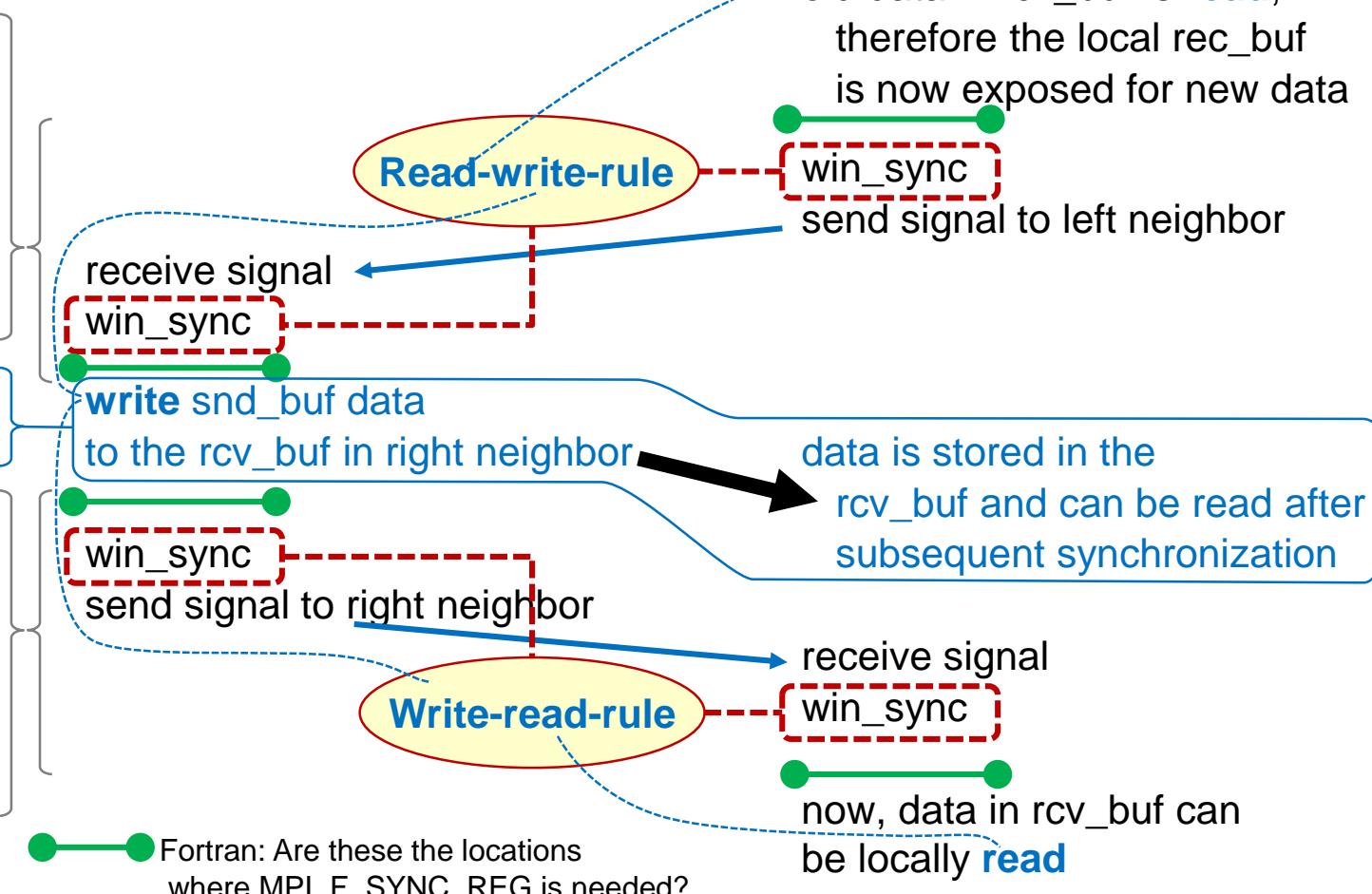
*(rcv_buf_ptr
 + (right-my_rank))
 = snd_buf;

Fortran: IF(..) CALL
 MPI_F_sync_reg(rcv_buf)
MPI_Win_sync(win);
MPI_Irecv(&dummy,...,
 left,...);
MPI_Send (&dummy,...,
 right,...);
MPI_Wait(...);
MPI_Win_sync(win);
Fortran: IF(..) CALL
 MPI_F_sync_reg(rcv_buf)

Outside of the loop:
MPI_Win_unlock_all(...);

Process rank n

Process rank n+1



During the Exercise (20 min.)



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

And have fun with this **long & complex** 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:

This is a very complex exercise:

2 lines of code for MPI_Lock_all/Unlock_all

and 5 lines as substitute for each MPI_Win_fence:

2xMPI_Win_sync + 3 lines Irecv+Send+Wait + 3 lines once for additional declarations therefore

plus 2xMPI_F_SYNC_REG(rcv_buf) with Fortran

$$= 2 + 2x(2+3+2) + 3 = 19 \text{ lines of code}$$



*You may decide to spend some time for understanding the problems
and then looking at the solution
or just to go to Exercise 8*

Exercise 6b (advanced): **Halo** – Using *other* synchronization

- Use your Exercise 3b result or
 - `~/MPI/tasks/C/halo-benchmarks/halo_1sided_store_win_alloc_shared.c`
`~/MPI/tasks/F_30/halo-benchmarks/halo_1sided_store_win_alloc_shared_30.f90` as your baseline for the following exercise:
- Tasks: Substitute the Win_fence synchronization by pt-to-pt communication
 - Use empty messages for synchronizing
 - Substitute each pair of MPI_Win_fence by
 - `MPI_Irecv(...right,...,rq[1] ...); MPI_Irecv(...left, ...,rq[2] ...);`
`MPI_Send(...left, ...); MPI_Send(...right, ...); MPI_Waitall(2,rq ...);`
 - Local MPI_Win_sync is needed for write-read and read-write-rule
- Compile and run shared memory program
 - With MPI processes on **4 cores** & **all cores** of a shared memory node



Exercise 7 (advanced): Ring – with memory signals

- Goal:
 - Substitute the Irecv-Send-Wait communication by two shared memory flags
- Hints:
 - After initializing these shared memory variables with 0, an additional MPI_Win_sync + MPI_Barrier + MPI_Win_sync is needed
 - Normally, from three consecutive MPI_Win_sync, only one call may be needed, because one memory fence is enough
- Recommendation:
 - One may study and run both the solution files and compare the latency
 - `halo_1sided_store_win_alloc_shared_signal.c` (only solution in C)
 - `ring_1sided_store_win_alloc_shared_signal.c / .py` (only solution in C and mpi4py)

Exercise 7 (advanced): Ring – with memory signals

Process rank n

Atomic (or volatile) load

signal_A

while ($A==0$) IDLE

$A = 0$

win_sync(A)

win_sync(B)

→ B is now locally 0

win_sync

write snd_buf data

to the rcv_buf in right neighbor

win_sync

1

Process rank n+1

old data in rcv_buf is read

win_sync

1

Atomic (or volatile) store

data is stored in the rcv_buf and
can be read after subsequent synchronization

signal B

while ($B==0$) IDLE

$B = 0$

win_sync(B)

→ A is now locally 0

win_sync

now, data in rcv_buf can be locally read



Exercise 8 (advanced): Halo communication benchmarking

- Goal:
 - Learn about the communication latency and bandwidth on your system
- Method:
 - **cd MPI/tasks/C/halo-benchmarks**
 - **unlimit** or **ulimit -s 200000** once before calling **mpirun**
 - On a shared or distributed memory, run and compare:
 - halo_irecv_send.c
 - halo_isend_recv.c
 - halo_neighbor_alltoall.c
 - halo_1sided_put.c
 - halo_1sided_put_alloc_mem.c
 - halo_1sided_put_win_alloc.c

} Different communication methods

 - halo_1sided_store_win_alloc_shared.c
 - halo_1sided_store_win_alloc_shared_query.c (with alloc_shared_noncontig)
 - halo_1sided_store_win_alloc_shared_pscw.c
 - halo_1sided_store_win_alloc_shared_othersync.c
 - halo_1sided_store_win_alloc_shared_signal.c

} Different communication methods

Quiz on Chapter 11-(2) – Shared Memory Model & Sync.

- A. Which MPI memory model applies to MPI shared memory?
MPI_WIN_SEPARATE or MPI_WIN_UNIFIED ?
- B. “Public and private copies are ? synchronized without additional RMA calls.”
- C. Which process-to-process synchronization methods can be used that, e.g., a store to a shared memory variable gets visible to another process (within the processes of the shared memory window)?

1. _____
2. _____
3. _____

- D. That such a store gets visible in another process after the synchronization is named here as “*write-read-rule*”.

Which other rules are implied by such synchronizations and what do they mean?

1. _____
2. _____

- E. How can you define a **race-condition** and which problems arise from **cache-line false-sharing**?

1. _____
2. _____

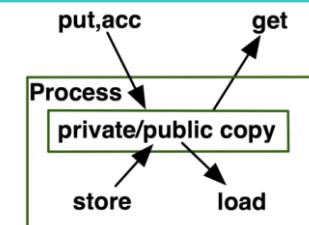


Figure: Courtesy of Torsten Hoefer

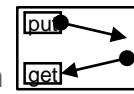
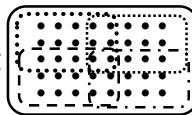
For private notes

Chap.12 Derived Datatypes

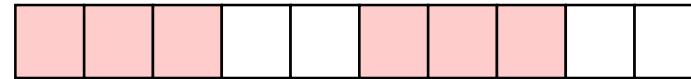
1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. The New Fortran Module `mpi_f08`
6. Collective communication
7. Error Handling
8. Groups & communicators, environment management
9. Virtual topologies
10. One-sided communication
11. Shared memory one-sided communication



`MPI_Init()`
`MPI_Comm_rank()`

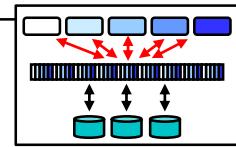


12. Derived datatypes



- (1) transfer of any combination of typed data
- (2) advanced features, alignment, resizing

13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



tour

Short tour – 3 slides



MPI Datatypes

- In the previous chapters:
 - A messages was a contiguous sequence of elements of basic types:
 - `buf, count, datatype_handle`
- New goals in this course chapter:
 - Transfer of any data in memory in one message
 - Strided data (portions of data with holes between the portions)
 - Various basic datatypes within one message
 - No multiple messages → **no multiple latencies**
 - No copying of data into contiguous scratch arrays
→ **no waste of memory bandwidth**
- Method: **Datatype handles**
 - Memory layout of send / receive buffer
 - Basic types / **derived types**:
 - vectors
 - subarrays
 - structs
 - others



Message passing:

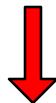
- Goal and reality may differ !!!

Parallel file I/O:

- Derived datatypes are important to express I/O patterns

Data Layout and the Describing Datatype Handle

```
struct buff_layout  
{ int i_val[3];  
  double d_val[5];  
 } buffer;
```



Compiler

```
array_of_types[0]=MPI_INT;  
array_of_blocklengths[0]=3;  
array_of_displacements[0]=0;  
array_of_types[1]=MPI_DOUBLE;  
array_of_blocklengths[1]=5;  
array_of_displacements[1]=...;
```

```
MPI_Type_create_struct(2, array_of_blocklengths,  
                      array_of_displacements, array_of_types,  
                      &buff_datatype);
```

```
MPI_Type_commit(&buff_datatype);
```

MPI_Send(&buffer, 1, **buff_datatype**, ...)

&buffer = the start
address of the data

the datatype handle
describes the data layout



Derived Datatypes — Type Maps

- A derived datatype is logically a pointer to a list of entries:
 - *basic datatype at displacement*

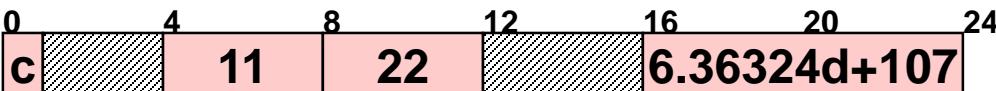
basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1



- Matching datatypes:
 - List of basic datatypes must be identical,
 - (*Displacements irrelevant*)

basic datatype 0	disp 0
basic datatype 1	disp 1
...	...
basic datatype n-1	disp n-1

Derived Datatypes — Type Maps

Example: 

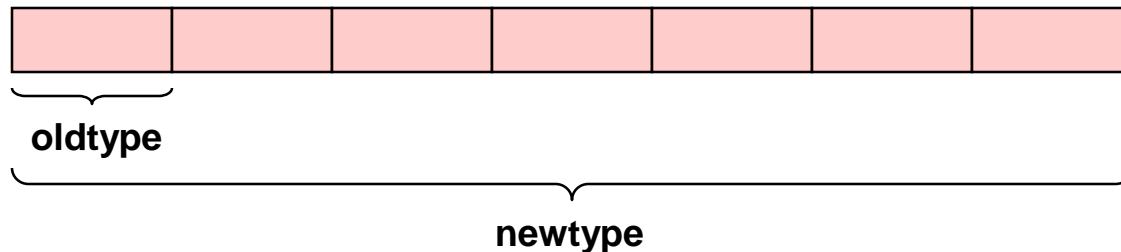
derived datatype handle

basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

A derived datatype describes the memory layout of, e.g., structures, common blocks, subarrays, some variables in the memory

Contiguous Data

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype



C

- C/C++: `int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)`
- Fortran: `MPI_TYPE_CONTIGUOUS(count, oldtype, newtype, ierror)`
mpi_f08: `INTEGER :: count`
`TYPE(MPI_Datatype) :: oldtype, newtype`
`INTEGER, OPTIONAL :: ierror`
mpi & mpif.h: `INTEGER count, oldtype, newtype, ierror`
- Python: `newtype = oldtype.Create_contiguous(int count)`

Python

Committing and Freeing a Datatype

- Before a datatype handle is used in message passing communication, it **needs to be committed with MPI_TYPE_COMMIT**.
- This need be done only once (by each MPI process).
(Using more than once ☺ corresponds to additional no-operations.)

C

Fortran

Python

- C/C++: `int MPI_Type_commit(MPI_Datatype *datatype);`
- Fortran: `MPI_TYPE_COMMIT(datatype, IERROR)`
mpi_f08: `TYPE(MPI_Datatype) :: datatype`
`INTEGER, OPTIONAL :: ierror`
mpi & mpif.h: `INTEGER datatype, ierror`
- Python: `datatype.Commit()`

IN-OUT argument
(although handle
is not modified)

- If usage is over, one may call `MPI_TYPE_FREE()` to free a datatype and its internal resources.

Exercise 1 — Derived Datatypes

In MPI/tasks/...

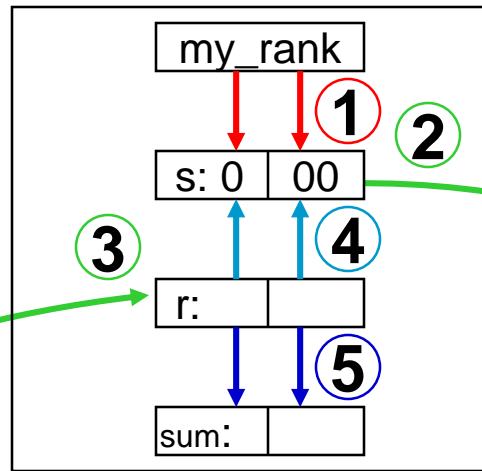
- Use **C** C/Ch12/derived-contiguous-skel.c
or **Fortran** F_30/Ch12/derived-contiguous-skel_30.f90
or **Python** PY/Ch12/derived-contiguous-skel.py
- We use a modified pass-around-the-ring exercise:
It sends a struct with two integers
- They are initialized with **my_rank** and **10*my_rank**
- Therefore we calculate two separate sums.
- Currently, the data is sent with the description
 - “`snd_buf, 2, MPI_INTEGER`”
- Please substitute this by using a
 - derived datatype
 - with a type map of “two integers”
 - Of course produced with the two routines on the previous slides

Exercise 1 — Derived Datatypes

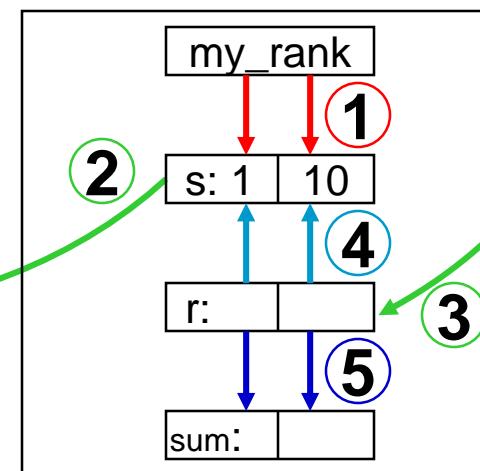
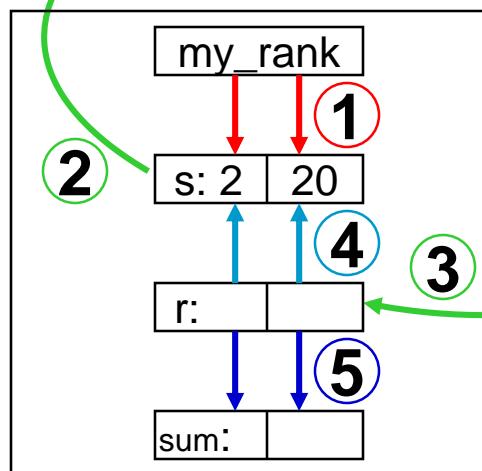
Initialization: 1

Each iteration:

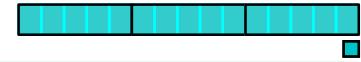
2 3 4 5



Sending both integers
• with **one** instance of an
MPI_Type_contiguous
derived datatype
• containing two integers



During the Exercise (15 min.)



Please stay here in the main room while you do this 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:



It looks easy, isn't it?

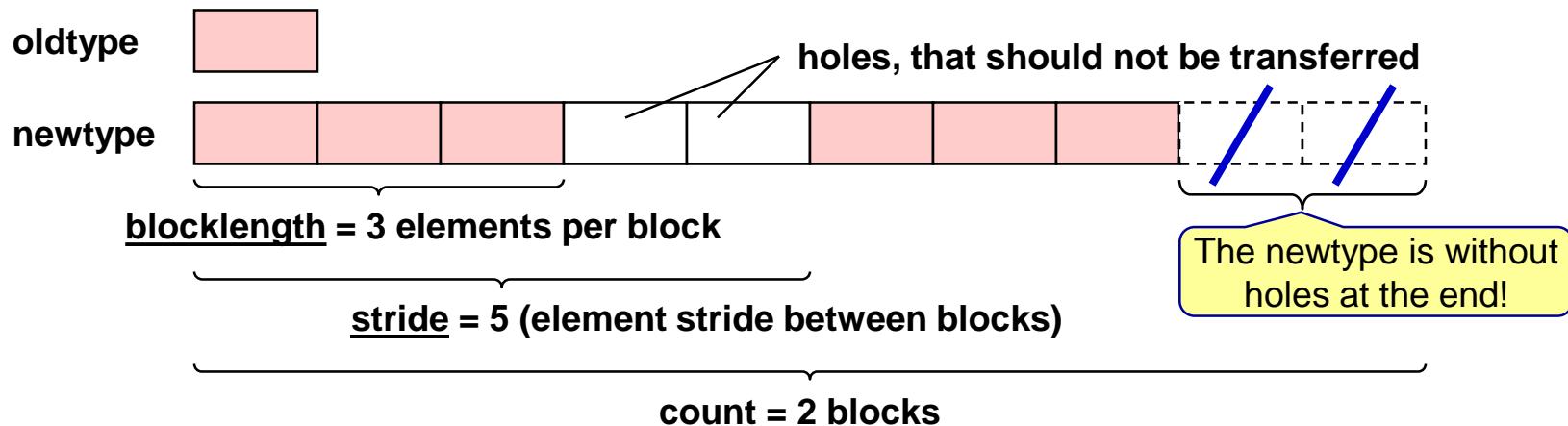


Exercises 1b (advanced) — MPI_Sendrecv

3. Substitute your Issend–Recv–Wait method by **MPI_Sendrecv** in your ring-with-datatype program:
 - MPI_Sendrecv is a *deadlock-free* combination of MPI_Send and MPI_Recv: **2** **3**
 - MPI_Sendrecv is described in the MPI standard.
(You can find MPI_Sendrecv by looking at the function index on the last pages of the standard document.)
 - Start from your solution of Exercise 1
 - Solution: MPI/tasks/C/Ch12/solutions/derived-contiguous-advanced-sendrecv.c
and MPI/tasks/F_30/Ch12/solutions/derived-contiguous-advanced-sendrecv_30.f90
and MPI/tasks/PY/Ch12/solutions/derived-contiguous-advanced-sendrecv.py

Vector Datatype

MPI_Type_create_subarray
is more flexible and usable for any dimensions, see course chapter
12-(2) and example in 13-(2)



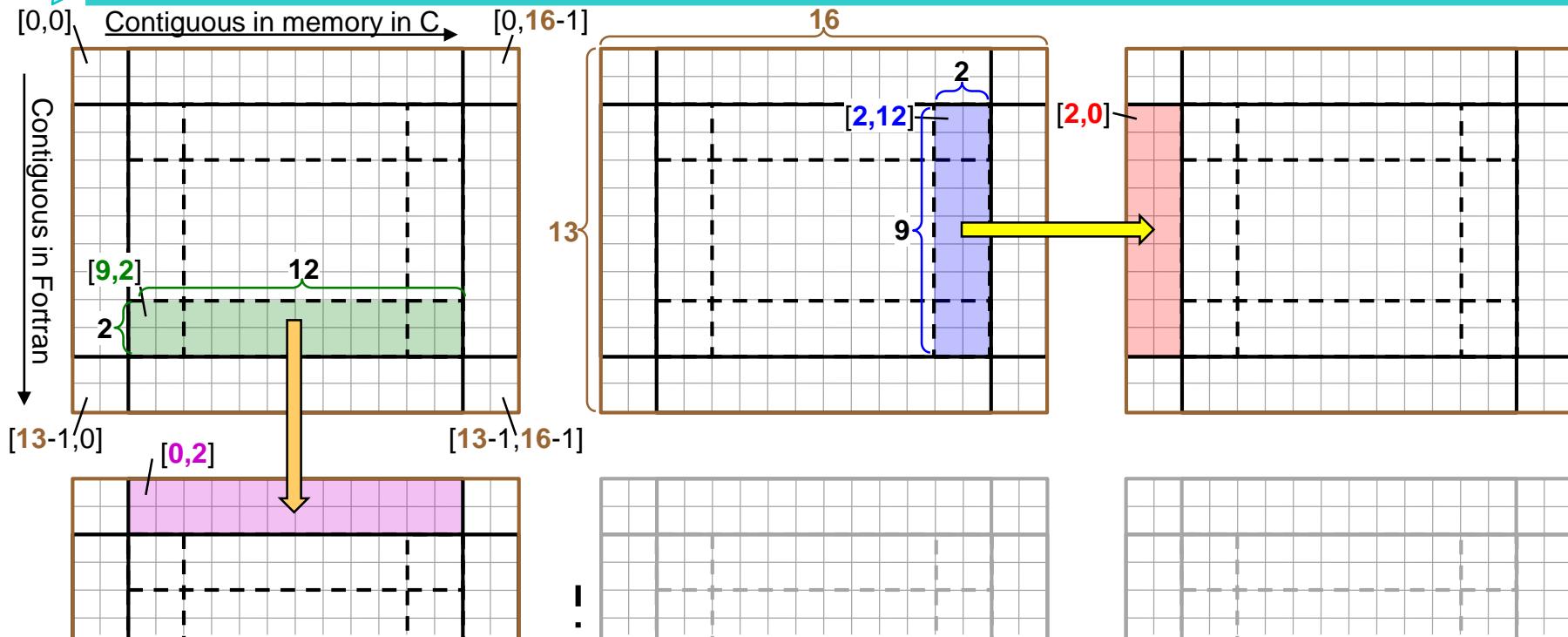
C

- C/C++: `int MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)`
- Fortran: `MPI_TYPE_VECTOR(count, blocklength, stride,
oldtype, newtype, ierror)`
mpi_f08: `INTEGER :: count, blocklength, stride
TYPE(MPI_Datatype) :: oldtype, newtype
INTEGER, OPTIONAL :: ierror`
mpi & mpif.h: `INTEGER count, blocklength, stride, oldtype, newtype, ierror`
- Python: `newtype = oldtype.Create_vector(int count, int blocklength, int stride)`

Python

Same indexes in C and Fortran

Example with MPI_Type_vector



Fortran c

```
MPI_Type_vector(2, 12, 16, etype, &newt);  
MPI_Send(a[9,2], 1, newt, ...);
```

```
CALL MPI_Type_vector(12, 2, 13, ..., newt)  
CALL MPI_Send(a(9,2), 1, newt, ...)
```

```
MPI_Type_vector(2, 12, 16, etype, &newt);  
MPI_Recv(a[0,2], 1, newt, ...);
```

```
CALL MPI_Type_vector(12, 2, 13, ..., &newt)  
CALL MPI_Recv(a(0,2), 1, newt, ...)
```

```
MPI_Type_vector(9, 2, 16, etype, &newt);  
MPI_Send(a[2,12], 1, newt, ...); → MPI_Recv(a[2,0], 1, newt, ...);
```

```
CALL MPI_Type_vector(2, 9, 13, ...)  
CALL MPI_Send(a(2,12), 1, newt, ...) → CALL MPI_Recv(a(2,0), 1, newt, ...)
```

```
MPI_Type_vector(9, 2, 16, etype, &newt);  
MPI_Recv(a[2,0], 1, newt, ...);
```

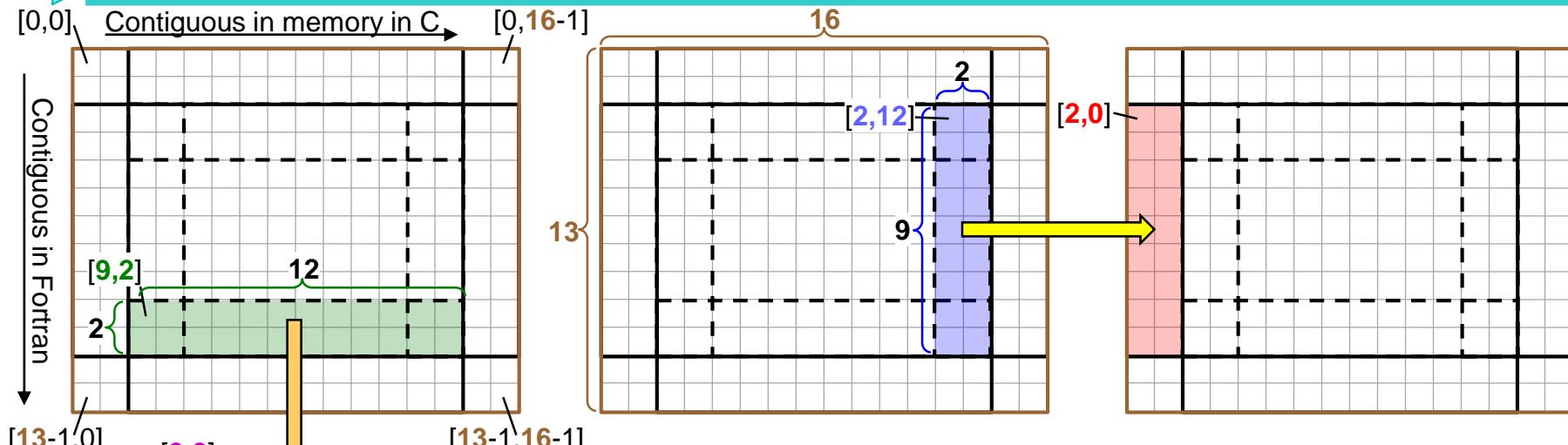
```
CALL MPI_Type_vector(2, 9, 13, ..., newt)  
CALL MPI_Recv(a(2,0), 1, newt, ...)
```

Python

Same numbering as with c

Same indexes in C and Fortran

Same example with MPI_Type_create_subarray



Fortran

```
CALL MPI_Type_create_subarray(  
 2, [13,16], [2,12], [9,2],  
  MPI_ORDER_FORTRAN, etype, newt)  
MPI_Send(a, 1, newt, ...);  
  
CALL MPI_Type_create_subarray(  
 2, [13,16], [2,12], [0,2],  
  MPI_ORDER_FORTRAN, etype, newt)  
MPI_Recv(a, 1, newt, ...);
```

```
CALL MPI_Type_create_subarray(  
  ndims=2, array_of_sizes=[13,16],  
  array_of_subsizes=[9,2],  
  array_of_starts=[2,12],  
  order=MPI_ORDER_FORTRAN,  
  oldtype=etype, newtype=newt)  
CALL MPI_Send(a, 1, newt, ...)
```

C
Python

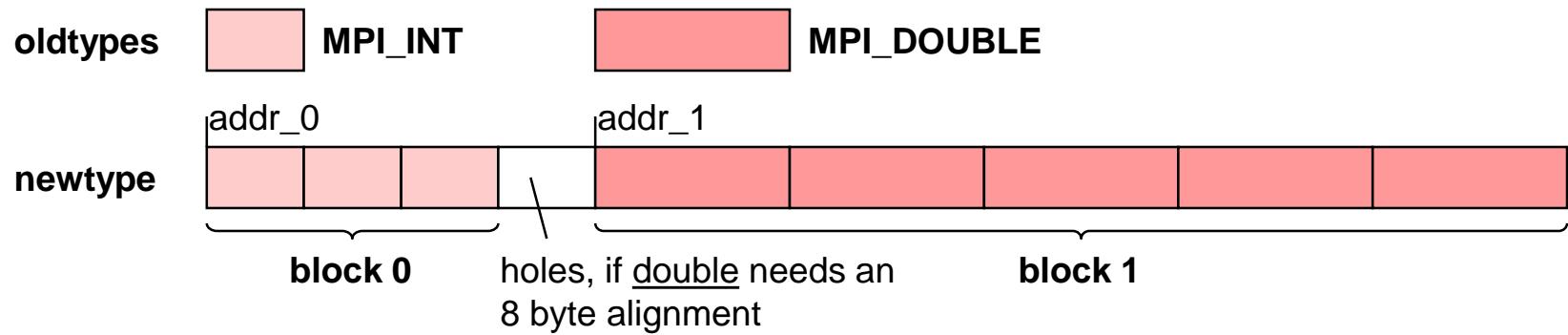
```
CALL MPI_Type_create_subarray(  
  ndims=2, array_of_sizes=[13,16],  
  array_of_subsizes=[9,2],  
  array_of_starts=[2,0],  
  order=MPI_ORDER_FORTRAN,  
  oldtype=etype, newtype=newt)  
MPI_Recv(a, 1, newt, ...)
```

Same numbers in C and Fortran,
only the **order** is different:
MPI_ORDER_C in C and Python

See also
13-(2)
subarray



Struct Datatype



C

- C/C++: `int MPI_Type_create_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)`
- Fortran: `MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements1, array_of_types, newtype, ierror)`
- Python: `newtype = MPI.Datatype.Create_struct(array_of_blocklengths, array_of_displacements, array_of_types)`

```
count = 2
array_of_blocklengths = ( 3,      5 )
array_of_displacements = ( 0,      addr_1 - addr_0 )2
array_of_types = ( MPI_INT, MPI_DOUBLE )
```

¹) INTEGER(KIND=MPI_ADDRESS_KIND) array_of_displacements

²) Via MPI_Get_address and MPI_Aint_diff, see following slides

Memory Layout of Struct Datatypes

buf_datatype



C

Fixed memory layout:

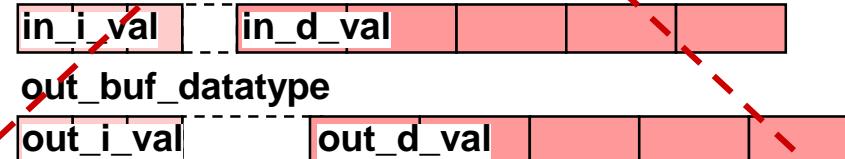
- C
struct buff
{ int i_val[3];
double d_val[5]; }
- Fortran, derived types
 - TYPE buff_type
SEQUENCE !!!
INTEGER, DIMENSION(3):: i_val
DOUBLE PRECISION, &
DIMENSION(5):: d_val
END TYPE buff_type
TYPE (buff_type) :: buff_variable
 - Alternative, in MPI-3.0:
TYPE, BIND(C) :: buff_type
- Fortran, common block
integer i_val(3)
double precision d_val(5)
common /bcomm/ i_val, d_val
- Python – mpi4py with numpy:
buff_type = np.dtype([('i', np.intc, 3), ('d', np.double, 5)], align=True)
buff = np.empty(), dtype=buff_type)

Fortran

Python

Alternatively, arbitrary memory layout:

- Each array is allocated independently.
- Each buffer is a pair of a 3-int-array and a 5-double-array.
- The length of the hole may be any arbitrary positive or negative value!
- For each buffer, one needs a specific datatype handle
- CAUTION – Fortran register optimization:
MPI_Send & _Recv of ...d_val is invisible for the compiler → add MPI_Address
in_buf_datatype



Not portable, because address differences are allowed only inside of structures or arrays

→ MPI-3.1/-4.0, Sect. 4/5.1.12 “Correct Use of Addresses”

True: with hole, as in C.

Default = False: no holes,
problematic, see course Chapter 12-(2).

How to compute the displacement (1)

- `array_of_displacements[i] := address(block_i) – address(block_0)`

Retrieve an absolute address:

- C/C++: `int MPI_Get_address(void* location, MPI_Aint *address)`
- Fortran:
 - mpi_f08: `MPI_GET_ADDRESS(location, address, ierror)`
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location
INTEGER(KIND=MPI_ADDRESS_KIND) :: address
INTEGER, OPTIONAL :: ierror
 - mpi & mpif.h: `<type> location(*)`
INTEGER(KIND=MPI_ADDRESS_KIND) address
INTEGER ierror
- Python: `address = MPI.Get_address(location)`

C

Fortran

Python

How to compute the displacement (2)

New in MPI-3.1

Relative displacement := absolute address 1 – absolute address 2

C

Fortran

Python

- C/C++: `MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)`
- Fortran: `MPI_AINT_DIFF(addr1, addr2)`
mpi_f08: `INTEGER(KIND=MPI_ADDRESS_KIND) :: addr1, addr2`
mpi & mpif.h: `INTEGER(KIND=MPI_ADDRESS_KIND) addr1, addr2`
- Python: `int MPI.Aint_diff(addr1, addr2)`

Python's int allows 64 bit

New in MPI-3.1

C

Fortran

Python

New absolute address := existing absolute address + relative displacement:

- C/C++: `MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)`
- Fortran: `MPI_AINT_ADD(base, disp)`
mpi_f08: `INTEGER(KIND=MPI_ADDRESS_KIND) :: base, disp`
mpi & mpif.h: `INTEGER(KIND=MPI_ADDRESS_KIND) base, disp`
- Python: `int MPI.Aint_add(base, disp)`

Example for array_of_displacements[i] := address(block_i) – address(block_0)

C

```
struct buff
{
    int     i[3];
    double d[5];
} snd_buf;
MPI_Aint iaddr0, iaddr1, disp;
MPI_Get_address( &snd_buf.i[0], &iaddr0); // the address value &snd_buf.i[0] is stored into variable iaddr0
MPI_Get_address(&snd_buf.d[0], &iaddr1);
disp = MPI_Aint_diff(iaddr1, iaddr0);           // MPI-3.0 & former: disp = iaddr1–iaddr0
```

New in MPI-3.1

Fortran

```
TYPE buff_type
  SEQUENCE
    INTEGER,                      DIMENSION(3) :: i
    DOUBLE PRECISION,             DIMENSION(5) :: d
  END TYPE buff_type
  TYPE (buff_type) :: snd_buf
  INTEGER(KIND=MPI_ADDRESS_KIND) iaddr0, iaddr1, disp; INTEGER ierror
  CALL MPI_GET_ADDRESS( snd_buf%i(1), iaddr0, ierror) ! The address of snd_buf%i(1) is stored in iaddr0
  CALL MPI_GET_ADDRESS(snd_buf%d(1), iaddr1, ierror)
  disp = MPI_AINT_DIFF(iaddr1, iaddr0)                   ! MPI-3.0 & former: disp = iaddr2–iaddr1
```

New in MPI-3.1

Python

```
np_dtype = np.dtype([('i', np.intc, 3), ('d', np.double, 4)])
snd_buf = np.empty((), dtype=np_dtype)
addr0 = MPI.Get_address(snd_buf['i'])
addr1 = MPI.Get_address(snd_buf['d'])
disp = MPI.Aint_diff(addr1, addr0)
```

See also MPI-3.1/-4.0, Example 4.8/5.8, page 102/142
and Example 4.17/5.17, pp 125-127/168-171

Scope & Performance options

Scope of MPI derived datatypes:

- Fixed memory layout
 - but not a linked list/tree,
i.e., if the location of data portions depend on data (pointers/indexes) in this list
- C++ data structures often require external libraries for flattening such data
- E.g., Boost serialization methods

Which is the fastest neighbor communication with strided data?

- **Copying** the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer, and copying the recv-buffer back into the strided application array
- Using derived datatype handles
- And which of the communication routines should be used?

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming
but

efficiency of MPI application-programming is **not portable!**

Especially with **hybrid MPI+OpenMP**, multiple threads may be used for such **copying**, whereas an MPI call may internally process **derived types** only with one thread.

Exercise 2 — Derived Datatypes

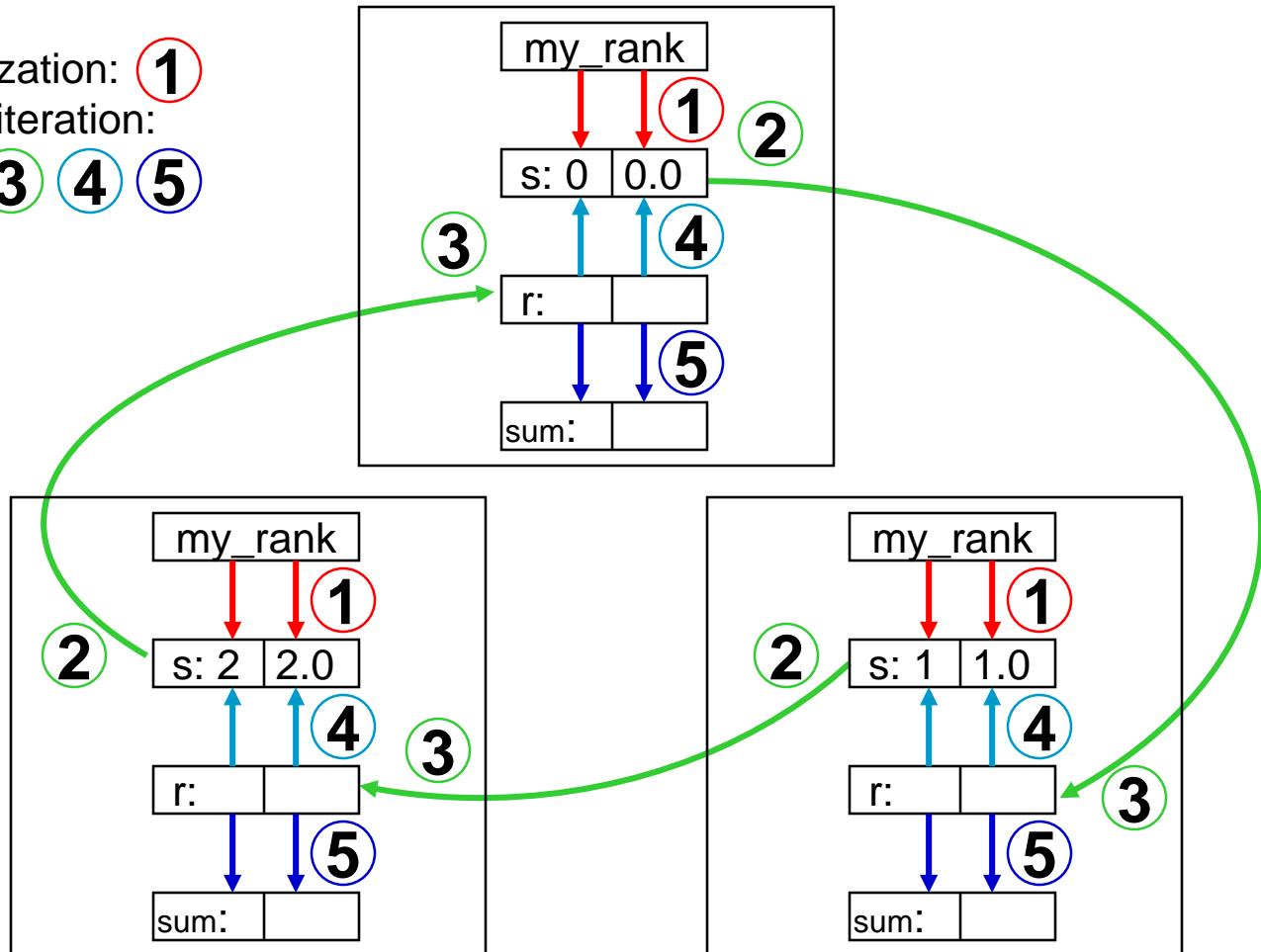
- Modify the pass-around-the-ring exercise.
- Use the following skeletons to reduce software-coding time:
 - C** cd ~/MPI/tasks/C/Ch12/ ; cp -p derived-struct-skel.c derived-struct.c
 - Fortran** cd ~/MPI/tasks/F_30/Ch12/ ; cp -p derived-struct-skel_30.f90 derived-struct_30.f90
 - Python** cd ~/MPI/tasks/PY/Ch12/ ; cp -p derived-struct-skel.py derived-struct.py
- Calculate two separate sums:
 - rank integer sum (as before)
 - rank floating point sum
- Use a *struct* datatype for this
- with same fixed memory layout for send and receive buffer.
- Substitute all within the skeleton
and modify the second part, i.e., steps 1-5 of the ring example

Exercise 2 — Derived Datatypes

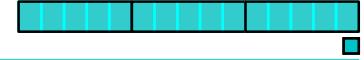
Initialization: 1

Each iteration:

2 3 4 5



During the Exercise (15 min.)



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



And have fun with this a bit longer exercise

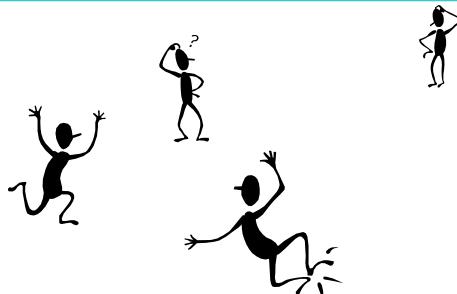
Please do not look at the solution before you finished this exercise,
otherwise,
90% of your learning outcome may be lost



As soon as you finished the exercise,
please go to your breakout room

and continue your discussions with your fellow learners:

*If you want, you can share your thoughts about
whether you would use MPI derived datatypes*



Exercises 3+4 (advanced) — Sendrecv & Sendrecv_replace

3. Substitute your Issend–Recv–Wait method by **MPI_Sendrecv** in your ring-with-datatype program:
 - MPI_Sendrecv is a *deadlock-free* combination of MPI_Send and MPI_Recv: **2** **3**
 - MPI_Sendrecv is described in the MPI standard.
 - You can find MPI_Sendrecv by looking at the function index on the last pages of the standard document.
 - Solution: MPI/tasks/C/Ch12/solutions/derived-struct-advanced-sendrecv.c
and MPI/tasks/F_30/Ch12/solutions/derived-struct-advanced-sendrecv_30.f90
and MPI/tasks/PY/Ch12/solutions/derived-struct-advanced-sendrecv.py

Same Exercise as Advanced Exercise 1b

If you solved already Advanced Exercise 1b then move to Exercise 4

4. Substitute MPI_Sendrecv by **MPI_Sendrecv_replace**:
 - Three steps are now combined: **2** **3** **4**
 - The receive buffer (rcv_buf) must be removed.
 - The iteration is now reduced to three statements:
 - MPI_Sendrecv_replace to pass the ranks around the ring,
 - computing the integer sum,
 - computing the floating point sum.
 - Solution: MPI/tasks/C/Ch12/solutions/derived-struct-advanced-sendrecv-replace.c
and MPI/tasks/F_30/Ch12/solutions/derived-struct-advanced-sendrecv-replace_30.f90
and MPI/tasks/PY/Ch12/solutions/derived-struct-advanced-sendrecv-replace.py

Quiz on Chapter 12-(1) – Derived datatypes

A. Which types of data in your application's memory can you describe with a derived datatype handle?

B. Logically, to which internal structure points a derived datatype handle?

C. Two pairs (count_1 , datatype_1) and (count_2 , datatype_2) match if ...?

D. If you have an array of a structure in your memory, how would you describe this?

E. Which additional MPI procedure call is required, before you can use a newly generated derived datatype handle in an MPI communication procedure?

F. If you have a (noncontiguous) subarray of a multidimensional array, which procedure would you use to generate an appropriate derived datatype handle and which count value would you use in MPI_Send or MPI_Recv?

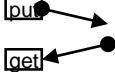
G. Which MPI procedures and functions should you use to calculate a byte displacement?

For private notes

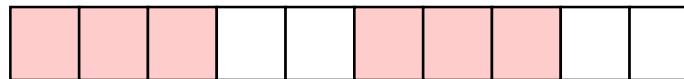
For private notes

For private notes

Chap.12 Derived Datatypes (2nd part)

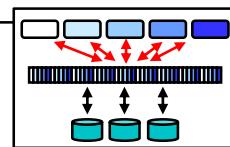
1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management
9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication

12. Derived datatypes



- (1) transfer of any combination of typed data
- (2) alignment, resizing, large counts, other derived types, MPI_Pack, MPI_BOTTOM

13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

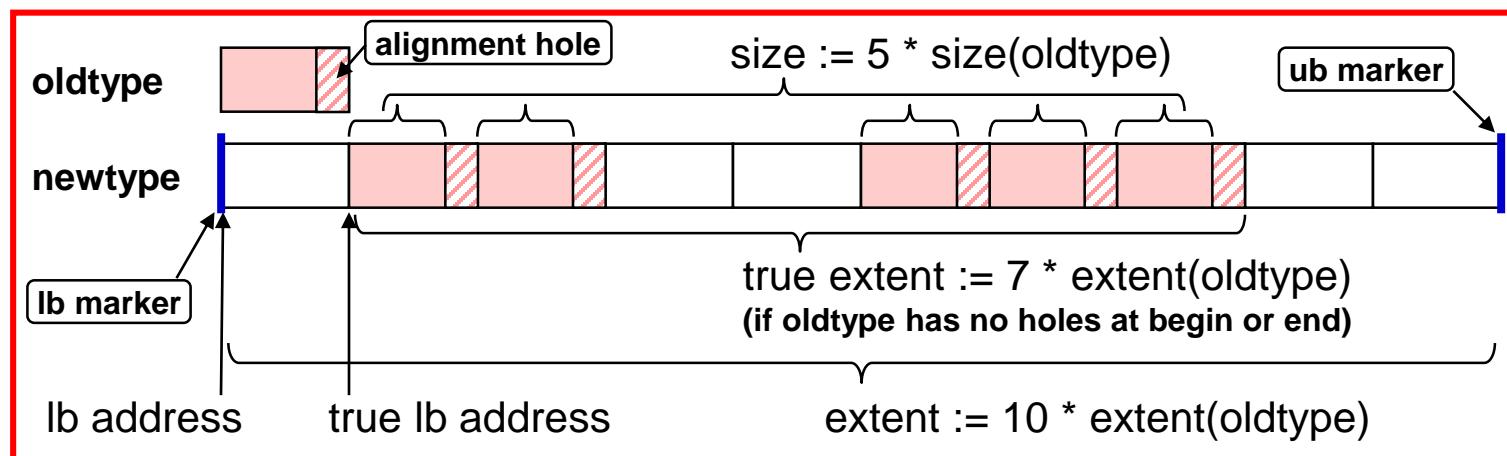


Does your MPI derived datatype really fits to the data in your memory?
→ More complicated than expected!

Size, Extent and True Extent of a Datatype, I.

- Size := number of bytes that have to be transferred.
- Extent := spans from first to last byte (including all holes).
- True extent := spans from first to last true byte (excluding holes at begin+end)
- Automatic holes at the end for necessary alignment purpose
- Additional holes at begin and by lb and ub markers: MPI_TYPE_CREATE_RESIZED
- Basic datatypes: Size = Extent = number of bytes used by the compiler.

Example:



skipped

Size and Extent of a Datatype, II.

C

Fortran

Python

C

Fortran

Python

C

Fortran

Python

- C/C++: int MPI_Type_size(MPI_Datatype datatype, int **size*)
- Fortran: MPI_TYPE_SIZE(datatype, *size*, *ierror*)
 - mpi_f08: TYPE(MPI_Datatype) :: datatype
 - INTEGER :: size
 - INTEGER, OPTIONAL :: ierror
- mpi & mpif.h: INTEGER datatype, size, ierror
- Python: *size* = datatype.Get_size(MPI_Datatype)

- C/C++: int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint **lb*, MPI_Aint **extent*)
- Fortran: MPI_TYPE_GET_EXTENT(datatype, *lb*, *extent*, *ierror*)
 - mpi_f08: TYPE(MPI_Datatype) :: datatype
 - INTEGER(KIND=MPI_ADDRESS_KIND) :: lb, extent
 - INTEGER, OPTIONAL :: ierror
- mpi & mpif.h: INTEGER datatype, ierror
- INTEGER(KIND=MPI_ADDRESS_KIND) lb, extent
- Python: (*lb*, *extent*) = datatype.Get_extent()

- C/C++: int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint **true_lb*, MPI_Aint **true_extent*)
- Fortran, Python: ditto

- Removed MPI-1 interface: MPI_TYPE_EXTENT()

Fortran derived types and MPI_Type_create_struct

- SEQUENCE **and BIND(C)** derived application types can be used as buffers in MPI operations.
- Alignment calculation of basic datatypes:
 - In MPI-2.2, it was undefined in which environment the alignments are taken.
 - There is no sentence in the standard.
 - **It may depend on compilation options!**
 - In MPI-3.0 to MPI-4.0, still undefined, but recommended to use a BIND(C) environment.

Alignment rule, holes and resizing of structures (1)

- The compiler may add additional alignment holes
 - within a structure (e.g., between a float and a double)
 - at the end of a structure (after elements with different sizes)!
 - See MPI-3.1/-4.0, Sect. 4/5.1.6, Advice to users on page 106 / 146.
- Alignment hole at the end is important when using an array of structures!
- Implication (**for C, Fortran(!) and Python**):
 - If an array of structures (in C/C++) or derived types (in Fortran) should be communicated, it is recommended that
 - the user creates a portable datatype handle and
 - should apply additionally **MPI_Type_create_resized** to this datatype handle.
 - See Example in MPI-3.1/-4.0, Sect. 17/19.1.15 on pages 637-638 / 823-825.
- Holes (e.g., due to alignment gaps) may cause significant loss of bandwidth
 - By definition, MPI is not allowed to transfer the holes.
 - Therefore the user should fill holes with dummy elements.
 - See Example MPI-3.1/-4.0, Sect. 4/5.1.6, Advice to users on page 106 / 146.

Alignment rule, holes and resizing of structures (2)

- **Correctness** problem with **array of structures**:
 - Possibility: MPI extent of a structure != real size of the structure
 - Reason: MPI adds at the end an alignment hole because the MPI library has wrong expectations about compiler rules
 - **For a basic datatype within the structure**
 - **For the allowed size of the whole structure (e.g. multiple of 16)**

C

- Solution in C: Call MPI_Type_create_resized with lb=0 and new_extent=sizeof(one structure), or use the following method:

Fortran

```
– & in Fortran: INTEGER(KIND=MPI_ADDRESS_KIND) &
                 :: address1, address2, lb, new_extent
CALL MPI_Get_address( my_struct(1), address1, ierror)
CALL MPI_Get_address( my_struct(2), address2, ierror)
new_extent = MPI_Aint_diff( address2, address1); lb = 0
CALL MPI_Type_create_resized ( &
                               old_struct_type, lb, new_extent, correct_struct_type, ierror)
```

Python

- & in Python: send_recv_resized =

correct type of
one struct

```
send_recv_type.Create_resized(0, snd_buf.itemsize)
```

type of one struct with
possibly wrong extent

array
of struct

length of
one struct

Alignment rule, holes and resizing of structures (3)

- Correctness problem with array of structures (continued):

C

- Example in C with [double+int]-structure:

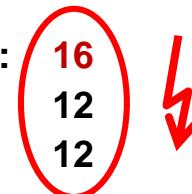
- MPI/tasks/C/Ch12/derived-struct-double+int.c
- Compiled and run on Cray with Intel compiler

With default alignment, all
works on the tested platform
(in Nov. 2015)

```
– module switch PrgEnv-cray PrgEnv-intel
– cc -Zp4 -o a.out ~/MPI/tasks/C/Ch12/derived-struct-double+int.c
– aprun -n 4 ./a.out | sort
```

- Result:

- MPI_Type_get_extent: **16**
- sizeof: **12**
- real size is: **12**



Python

- Similar in Python with default align=False

- MPI/tasks/C/Ch12/derived-struct-double+int.py
- Change align=True on line 34 to align=False → wrong results as above



For portable & correct applications
with arrays of structures,
the datatypes should be always resized!

Alignment rule, holes and resizing of structures (4)

- Correctness problem with array of structures (continued):

Fortran

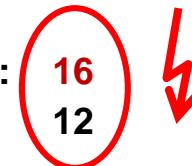
- Example in Fortran with [double precision + integer]-structure:

- MPI/tasks/F_30/Ch12/derived_struct_dp+integer_30.f90
- Compiled and run on Cray with Intel compiler
 - module switch PrgEnv-cray PrgEnv-intel
 - ftn –o a.out ~/MPI/tasks/F_30/Ch12/derived-struct-dp+integer_30.f90
 - aprun –n 4 ./a.out | sort

Fortran struct with SEQUENCE attribute

- Result:

- MPI_Type_get_extent: 16
 - real size is: 12



- Surprise (?):

- ~/MPI/tasks/F_30/Ch12/derived-struct-dp+integer-bindC_30.f90
 - MPI_Type_get_extent: 16
 - real size is: 16

Fortran struct with BIND(C)

- 2nd Surprise: With PrgEnv-cray, all sizes are 16 bytes

Alignment rule, holes and resizing of structures (5)

- **Performance** problem with **holes in structures**:
 - Correct solution for homogeneous and heterogeneous environments:
 - **Add dummy elements to fill the holes
(in the structure and in the datatype)**
 - In a homogeneous environment:
 - **One may use MPI_BYTE**
 - **Transfer whole structure as an array of bytes**
 - **CAUTION: No data conversion of different data representations
(e.g., big and little endian) in heterogeneous environments**

Large Counts with MPI_Count, ...

- MPI uses different integer types
 - int and INTEGER
 - MPI_Aint = INTEGER(KIND=MPI_ADDRESS_KIND)
 - MPI_Offset = INTEGER(KIND=MPI_OFFSET_KIND)
 - MPI_Count = INTEGER(KIND=MPI_COUNT_KIND) New in MPI-3.0

• $\text{sizeof}(\text{int}) \leq \text{sizeof}(\text{MPI_Aint}) \leq \text{sizeof}(\text{MPI_Count})$

- All count arguments are int or INTEGER.
- Real message sizes may be larger due to datatype size.

• MPI_Type_get_extent, MPI_Type_get_true_extent,
 MPI_Type_size, MPI_Type_get_elements
 return **MPI_UNDEFINED** if value is too large New in MPI-3.0

New in
MPI-3.0

• MPI_Type_get_extent_x, MPI_Type_get_true_extent_x,
 MPI_Type_size_x, MPI_Type_get_elements_x
 return values as **MPI_Count**

New in
MPI-4.0

• **MPI_Xxxx_c(...)** in C: additional interfaces with large counts
 MPI_Xxxx(...)_!(_c) in Fortran: overloaded interfaces with large counts

Python with mpi4py:
 Although mpi4py uses **Python's int counts** (mostly inferred from numpy arrays), and Python's int is **not restricted to 32 bit**, the mpi4py interfaces may be mapped to the **32 bit C int count** arguments of the underlying MPI library.

New in MPI-3.0

Two exceptions with explicit _c in Fortran:
MPI_Op_create_c & **MPI_Register_datarep_c**

All Derived Datatype Creation Routines (1)

—skipped—

- **MPI_Type_contiguous()**
→ already discussed
 - **MPI_Type_vector()**
→ already discussed
 - **MPI_Type_indexed()**
→ similar to ..._struct(),
same oldtype for all sub-blocks,
displacements based on 0-based index in “array of oldtype”
 - **MPI_Type_create_indexed_block()**
→ same as MPI_Type_indexed()
but same block length
for each sub-block
 - **MPI_Type_create_struct()**
→ already discussed
- MPI_Type_create_hvector()**
→ stride as byte size
 - MPI_Type_create_hindexed()**
→ with byte displacements
 - MPI_Type_create_hindexed_block()**
→ with byte displacements

All Derived Datatype Creation Routines (2)

- **MPI_Type_create_subarray()**
 - Extracts a subarray of an n-dimensional array
 - All the rest are holes
 - Ideal for halo exchange with n-dimensional Cartesian data-sets
 - Similar to MPI_Type_vector(), which works primarily for 2-dim arrays
 - Example, see course Chapter 13 *Parallel File I/O*
- **MPI_Type_create_darray()**
 - A generalization of **MPI_Type_create_subarray()**
 - Example, see course Chapter 13 *Parallel File I/O*

Removed MPI-1 interfaces

- MPI_Address
- MPI_Type_extent
- MPI_Type_hvector
- MPI_Type_hindexed
- MPI_Type_struct
- MPI_Type_LB / _UB
- Constant MPI_LB / _UB

substituted by

- MPI_Get_address
- MPI_Type_get_extent
- MPI_Type_create_hvector
- MPI_Type_create_hindexed
- MPI_Type_create_struct
- MPI_Type_get_extent
- MPI_Type_create_resized

Subarray and darray:
newtype
may contain holes at
begin and end !!! ☺
Important for filetypes
→ **Parallel File I/O**

New in MPI-2.0 to solve Fortran
problem with small integer:
• Unchanged argument list in C.
• Modified length arguments in
Fortran.

Better usable interface

Other MPI features: Pack/Unpack

- MPI_Pack & MPI_Unpack
 - Pack several data into a message buffer
 - Communicate the buffer with datatype = MPI_PACKED
- Canonical Pack & Unpack
 - Header-free packing in “external32” data representation
 - Only useful for cross-messaging **between different MPI libraries!**
 - Communicate the buffer with datatype = MPI_BYTE

Other MPI features: MPI_BOTTOM and absolute addresses

- MPI_BOTTOM in point-to-point and collective communication:
 - Buffer argument is MPI_BOTTOM
 - Then absolute addresses can be used in
 - Communication routines with byte displacement arguments
 - Derived datatypes with byte displacements
 - Displacements must be retrieved with **MPI_Get_address()**
 - MPI_BOTTOM is an address,
i.e., **cannot be assigned to a Fortran variable!**
 - MPI-3.1/-4.0, Section 2.5.4, p. 15 line 44 – p. 16 line 6 / p. 21 lines 12-22 shows all such address constants
that cannot be used in expressions or assignments **in Fortran**, e.g.,
 - **MPI_STATUS_IGNORE** (→ point-to-point comm.)
 - **MPI_IN_PLACE** (→ collective comm.)
 - Fortran: Using MPI_BOTTOM & absolute displacement of variable X
→ MPI_F_SYNC_REG is needed:
 - **MPI_BOTTOM in a blocking MPI routine** → MPI_F_SYNC_REG before and after this routine
 - **in a nonblocking routine** → MPI_F_SYNC_REG before this routine & after final WAIT/TEST

Fortran

Already discussed in course Chapter 9 **Virtual Topologies** → Exercise with **MPI_Neighbor_alltoallw**

Performance options

[already mentioned at the end of 12-(1)]

Which is the fastest neighbor communication with strided data?

- Copying the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer and copying the recv-buffer back into the strided application array
- Using derived datatype handles
- And which of the communication routines should be used?

Especially with **hybrid MPI+OpenMP**, multiple threads may be used for such **copying**, whereas an MPI call may internally process **derived types** only with one thread.

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming

but

efficiency of MPI application-programming is **not portable!**

Exercise 5+6 — Resizing a Derived Datatypes

Use the following examples for testing and as code-basis:

C

- **MPI/tasks/C/Ch12/derived-struct-double+int.c** or
- **MPI/tasks/F_30/Ch12/derived-struct-dp+integer_30.f90** and
- **MPI/tasks/F_30/Ch12/derived-struct-dp+integer-bindC_30.f90**
- **MPI/tasks/PY/Ch12/derived-struct-double+int.py**

Fortran

Python

5. Compile and test with different compilers and accompanying MPI libraries

- Pipe the stdout to: | sort +0 -1 -n +1 -2
- Example:

```
mpiexec -n 4 ./a.out | sort +0 -1 -n +1 -2
```

6. Implement a new datatype handle by resizing the old one.

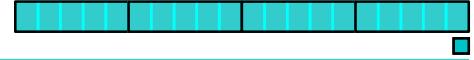
- Don't forget to substitute the datatype handle in all communication calls.

7. Advanced Exercise: Did you solve already the advanced exercises in

(before you use the following hyper-refs, please note the current slide number 468, because there aren't any "back" buttons to here)

- Course chapter 12-(1) Exe.1b : MPI_Sendrecv & MPI_Sendrecv_replace
- Course chapter 6-(2) Adv.Exe. 4 : MPI_IN_PLACE in collective communication

During the Exercise (20 min.)



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



And have fun with this a bit longer 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:

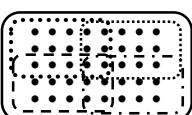
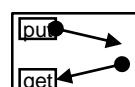
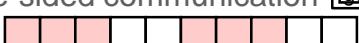
I recommend that you directly go to your breakout room
to exchange your questions, remarks and results with your colleagues.

For private notes

For private notes

For private notes

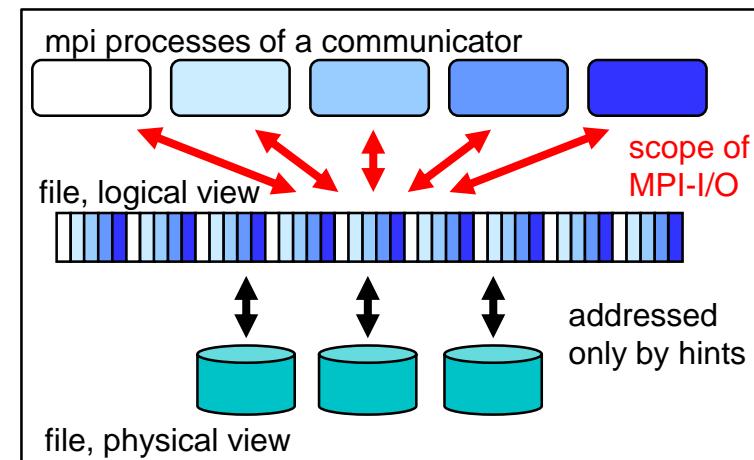
Chap.13 Parallel File I/O

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication
12. Derived datatypes 

13. Parallel file I/O

- Writing and reading a file in parallel by many processes

14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



Outline

- **Block 1**
 - Introduction [323]
 - **Definitions** [328]
 - Open / Close [330]
 - WRITE / **Explicit Offsets** [335]
 - Exercise 1 [336]
- **Block 2**
 - **File Views** [338]
 - **Subarray & Darray** [342]
 - I/O Routines Overview [350]
 - READ / Explicit Offsets [352]
 - **Individual File Pointer** [353]
 - Exercise 2 [355]
- **Block 3**
 - **Shared File Pointer** [358]
 - **Collective** [360]
 - Non-Blocking / Split Collective [364/365]
 - Other Routines [368]
 - Error Handling [369]
 - Implementation Restrictions [370]
 - **Summary** [371]
 - Exercise 3 [372]
 - Exercise 4 [373]

Motivation, I.

- Many parallel applications need
 - coordinated parallel access to a file by a group of processes
 - simultaneous access
 - all processes may read/write many (small) non-contiguous pieces of the file,
i.e. the data may be distributed amongst the processes according to a partitioning scheme
 - all processes may read the same data
- Efficient collective I/O based on
 - fast physical I/O by several processors, e.g. striped
 - distributing (small) pieces by fast message passing

Motivation, II.

- Analogy: writing / reading a file is like sending/receiving a message
- Handling parallel I/O needs
 - handling groups of processes → MPI topologies and groups
 - collective operations → file handle defined like communicators
 - nonblocking operations → MPI_I..., MPI_Wait, ... & new **split** collective interface
 - non-contiguous access → MPI derived datatypes

MPI-I/O Features

- Provides a high-level interface to support
 - data file partitioning among processes
 - transfer global data between memory and files (collective I/O)
 - asynchronous transfers
 - strided access
- MPI derived datatypes used to specify common data access patterns for maximum flexibility and expressiveness

MPI-I/O, Principles

- MPI file contains elements of a single MPI datatype (etype)
- partitioning the file among processes with an access template (filetype)
- all file accesses transfer to/from a contiguous or non-contiguous user buffer (MPI datatype)
- nonblocking / blocking and collective / individual read / write routines
- individual and shared file pointers, explicit offsets
- binary I/O
- automatic data conversion in heterog. systems
- file interoperability with external representation

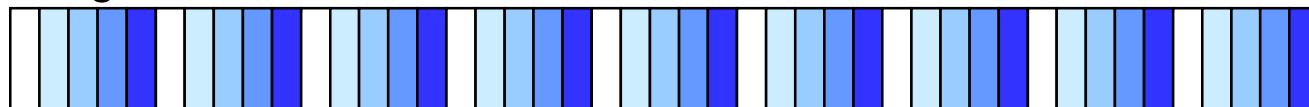


Logical view / Physical view

mpi processes of a communicator

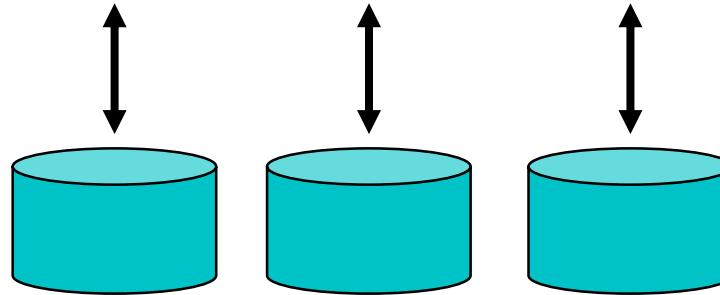


file, logical view



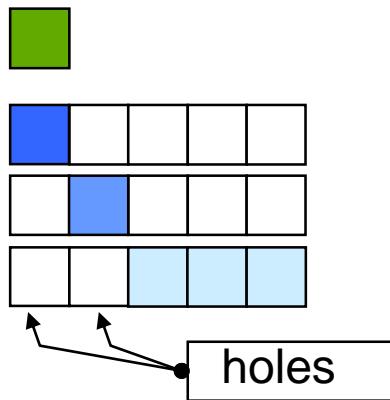
scope of
MPI-I/O

file, physical view



addressed
only by hints

Definitions



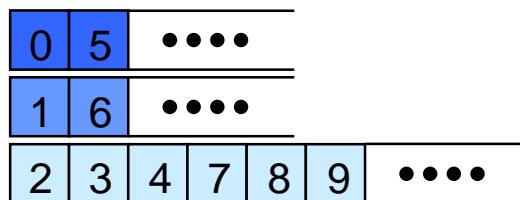
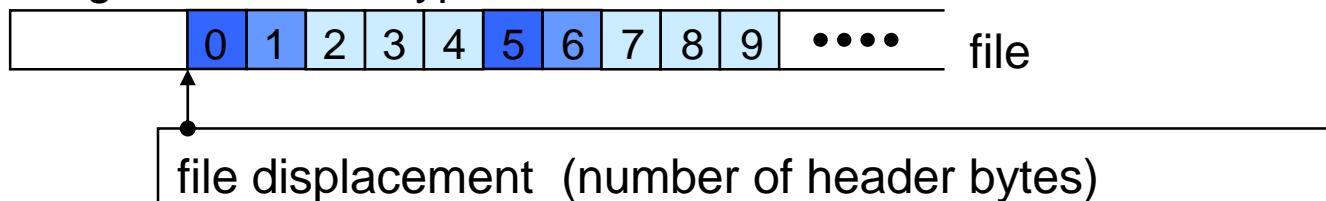
etype (elementary datatype)

filetype process 0

filetype process 1

filetype process 2

tiling a file with filetypes:



view of process 0

view of process 1

view of process 2

Comments on Definitions

- file** - an ordered collection of typed data items
- etypes** - is the unit of data access and positioning / offsets
 - can be any basic or derived datatype
(with non-negative, monotonically non-decreasing, non-absolute displacem.)
 - generally contiguous, but need not be
 - typically same at all processes
- filetypes** - the basis for partitioning a file among processes
 - defines a template for accessing the file
 - different at each process
 - the etype or derived from etype (displacements:
non-negative, monoton. non-decreasing, non-abs., multiples of etype extent)
- view** - each process has its own view, defined by:
a displacement, an etype, and a filetype.
 - The filetype is repeated, starting at **displacement**
- offset** - position relative to current view, in units of etype

Opening an MPI File

- **MPI_File_open** is collective over `comm`
- filename's namespace is implementation-dependent!
- filename must reference the same file on all processes
- process-local files can be opened by passing `MPI_COMM_SELF` as `comm`
- returns a file handle *fh*
[represents the file, the process group of `comm`, and the current view]

```
MPI_File_open(comm, filename, amode, info, fh)
```

Fortran

C/C++

Python

language bindings – see MPI Standard
and mpi4py

Default View

`MPI_File_open(comm, filename, amode, info, fh)`

- Default:

- displacement = 0
 - etype = MPI_BYTE
 - filetype = MPI_BYTE
- } each process has access to the whole file



view of process 0



view of process 1



view of process 2

- Sequence of MPI_BYTE matches with any datatype
(see MPI-3.1/-4.0, Section 13/14.6.6 on page 549/714)
- Binary I/O (no ASCII text I/O)

Access Modes

- same value of `amode` on all processes in **`MPI_File_open`**
- Bit vector OR of integer constants (Fortran 77: +)
 - `MPI_MODE_RDONLY` - read only
 - `MPI_MODE_RDWR` - reading and writing
 - `MPI_MODE_WRONLY` - write only
 - `MPI_MODE_CREATE` - create if file doesn't exist
 - `MPI_MODE_EXCL` - error creating a file that exists
 - `MPI_MODE_DELETE_ON_CLOSE` - delete on close
 - `MPI_MODE_UNIQUE_OPEN` - file not opened concurrently
 - `MPI_MODE_SEQUENTIAL` - file only accessed sequentially:
mandatory for sequential stream files (pipes, tapes, ...)
 - `MPI_MODE_APPEND` - all file pointers set to end of file
[caution: reset to zero by any subsequent `MPI_FILE_SET_VIEW`]

File Info: Reserved Hints

- Argument in MPI_File_open, MPI_File_set_view, MPI_File_set_info
- reserved key values:
 - collective buffering
 - “`collective_buffering`”: specifies whether the application may benefit from collective buffering
 - “`cb_block_size`”: data access in chunks of this size
 - “`cb_buffer_size`”: on each node, usually a multiple of block size
 - “`cb_nodes`”: number of nodes used for collective buffering
 - disk striping (only relevant in MPI_FILE_OPEN)
 - “`striping_factor`”: number of I/O devices used for striping
 - “`striping_unit`”: length of a chunk on a device (in bytes)
- MPI_INFO_NULL may be passed

Closing and Deleting a File

- Close: collective

```
MPI_File_close(fh)
```

- Delete:

- automatically by MPI_FILE_CLOSE
if **amode=MPI_DELETE_ON_CLOSE** | ...
was specified in MPI_FILE_OPEN
- deleting a file that is not currently opened:

```
MPI_File_delete(filename, info)
```

[same implementation-dependent rules as in MPI_FILE_OPEN]



language bindings – see MPI Standard



and mpi4py

Writing with Explicit Offsets

MPI_File_write_at(fh, offset, buf, count, datatype, *status*)

- writes **count** elements of **datatype** from memory **buf** to the file
- starting **offset * units** of **etype** from begin of view
- the elements are stored into the locations of the current view
- the sequence of basic datatypes of **datatype** (= signature of **datatype**) must match contiguous copies of the **etype** of the current view



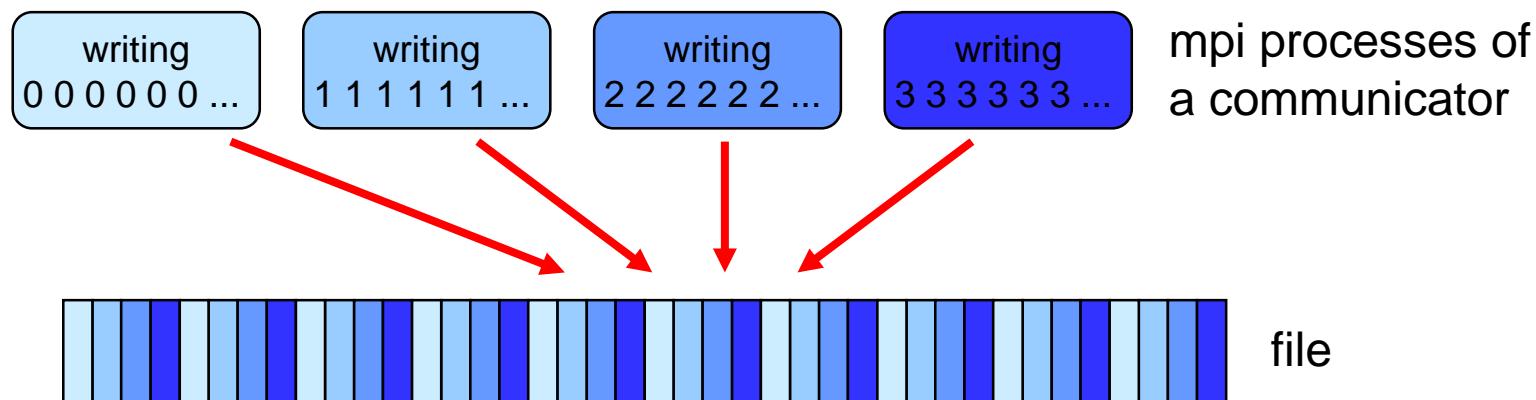
language bindings – see MPI Standard



and mpi4py

MPI-IO Exercise 1: Four processes write a file in parallel

- each process should write its rank (as one character) ten times to the offsets = $\text{my_rank} + i * \text{size_of_MPI_COMM_WORLD}$, $i=0..9$
- Result: “012301230123012301230123012301230123“
- Each process uses the default view



- please, use skeleton:

C

```
cp ~/MPI/tasks/C/Ch13/mpi_io_exa1_skel.c my_exa1.c
```

Fortran

```
cp ~/MPI/tasks/F_30/Ch13/mpi_io_exa1_skel_30.f90 my_exa1_30.f90
```

Python

```
cp ~/MPI/tasks/PY/Ch13/mpi_io_exa1_skel.py my_exa1.py
```

- edit; compile; **`rm -f my_test_file`**; **`mpirun ...`** (always remove `my_test_file` before re-run)
- `cat my_test_file; echo; wc -c my_test_file`** (verifying the result)

During the Exercise (20 min.)



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

And have fun with this a bit longer exercise



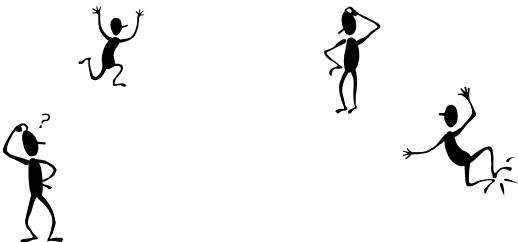
Please do not look at the solution before you finished this exercise,
otherwise,
90% of your learning outcome may be lost



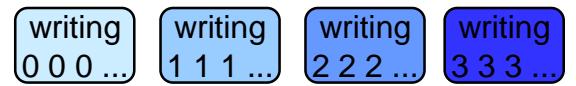
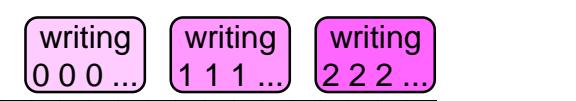
As soon as you finished the exercise,
please go to your breakout room

and continue your discussions with your fellow learners:

Who of you uses NetCDF or HDF5?
As far as I know, both use MPI-I/O.



MPI-IO Advanced Exercise 1b: MPI_File_set_size

- `rm -f my_test_file`
- Run program of Exercise 1 with 4 processes:
Expected result ““
- Do **not** remove `my_test_file`
and run again with **only 2** processes:
Expected result ““
- Please, make a copy of your result: `cp my_exa1.c my_exa1b.c or _30.f90`
- **Set the file size to 0 (zero) directly after the MPI_File_open.**
 - Use `MPI_File_set_size()`
 - For the interface, please look into the MPI standard.
- Compile and run again (**without** removing `my_test_file`),
now with **3** processes:
`cat my_test_file ; echo ; wc -c my_test_file`
Expected result: ““
- Solution: MPI/tasks/C/Ch13/solutions/mpi_io_exa1b.c
and MPI/tasks/F_30/Ch13/solutions/mpi_io_exa1b_30.f90



Outline – Block 2

- **Block 1**
 - Introduction [323]
 - **Definitions** [328]
 - Open / Close [330]
 - WRITE / **Explicit Offsets** [335]
 - Exercise 1 [336]
- **Block 2**
 - **File Views** [338]
 - **Subarray & Darray** [342]
 - I/O Routines Overview [350]
 - READ / Explicit Offsets [352]
 - **Individual File Pointer** [353]
 - Exercise 2 [355]
- **Block 3**
 - **Shared File Pointer** [358]
 - **Collective** [360]
 - Non-Blocking / Split Collective [364/365]
 - Other Routines [368]
 - Error Handling [369]
 - Implementation Restrictions [370]
 - **Summary** [371]
 - Exercise 3 [372]
 - Exercise 4 [373]

File Views

- Provides a visible and accessible set of data from an open file
- A separate view of the file is seen by each process through triple := (displacement, etype, filetype)
- User can change a view during the execution of the program - but collective operation
- A linear byte stream, represented by the triple (0, MPI_BYTE, MPI_BYTE), is the default view

Set/Get File View

- Set view
 - changes the process's view of the data
 - local and shared file pointers are reset to zero
 - collective operation
 - etype and filetype must be committed
 - datarep argument is a string that specifies the format in which data is written to a file:
“native”, “internal”, “external32”, or user-defined
 - same etype extent and same datarep on all processes
- Get view
 - returns the process's view of the data

```
MPI_File_set_view(fh, disp, etype, filetype, datarep, info)
```

```
MPI_File_get_view(fh, disp, etype, filetype, datarep)
```

Fortran C/C++

language bindings – see MPI Standard

Python

and mpi4py

Data Representation, I.

- “native”
 - data stored in file identical to memory
 - on homogeneous systems no loss in precision or I/O performance due to type conversions
 - on heterogeneous systems loss of interoperability
 - no guarantee that MPI files accessible from C/Fortran
- “internal”
 - data stored in implementation specific format
 - can be used with homogeneous or heterogeneous environments
 - implementation will perform type conversions if necessary
 - no guarantee that MPI files accessible from C/Fortran

Data Representation, II.

- “external32”
 - follows standardized representation (IEEE)
 - all input/output operations are converted from/to the “external32” representation
 - files can be exported/imported between different MPI environments
 - due to type conversions from (to) native to (from) “external32” data precision and I/O performance may be lost
 - “internal” may be implemented as equal to “external32”
 - can be read/written also by non-MPI programs
- user-defined

No information about the default,
i.e., datarep without `MPI_File_set_view()` is not defined

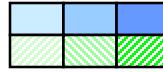
Fileview examples with SUBARRAY and DARRAY

- Task
 - reading a global matrix from a file
 - storing a subarray into a local array on each process
 - according to a given distribution scheme

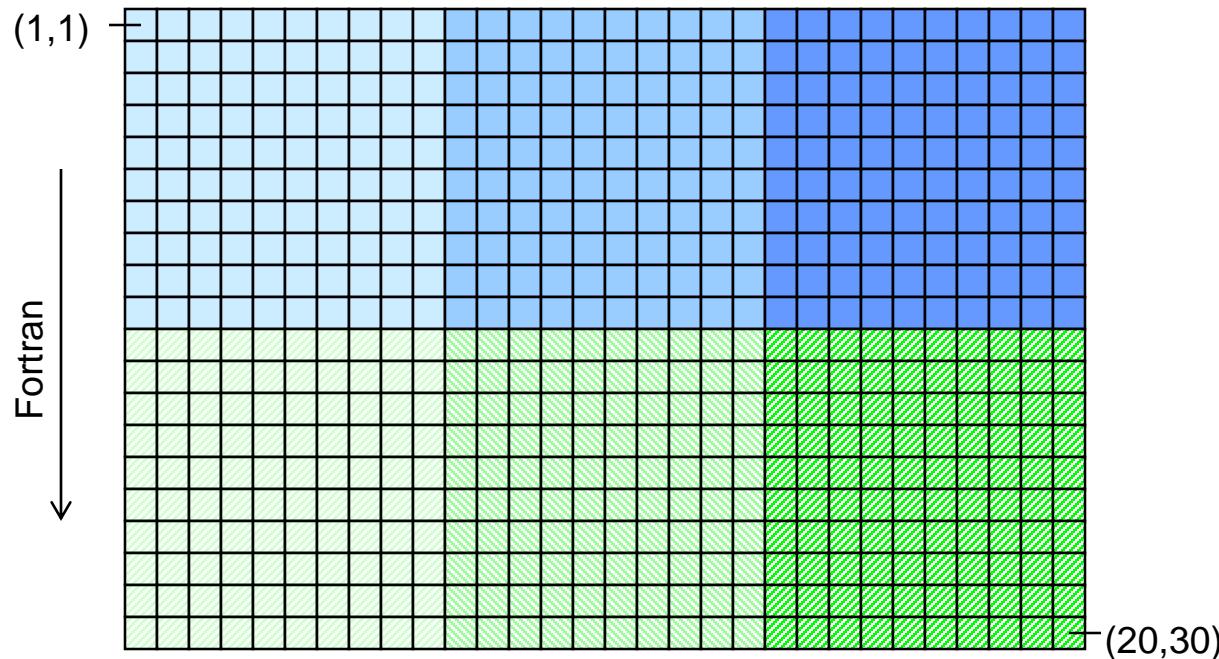
Example with Subarray, I.

- 2-dimensional distribution scheme: (BLOCK,BLOCK)
- garray on the file 20x30:
 - Contiguous indices is language dependent:
 - in Fortran: (1,1), (2,1), (3,1), ..., (1,10), (2,20), (3,10), ..., (20,30)
 - in C/C++: [0][0], [0][1], [0][2], ..., [10][0], [10][1], [10][2], ..., [19][29]
- larray = local array in each MPI process
 - = subarray of the global array
- same ordering on file (garray) and in memory (larray)

Example with Subarray, II. — Distribution

- Process topology: 2x3 
- global array on the file: 20x30
- distributed on local arrays in each process: 10x10

C / C++ (contiguous indices on the file and in the memory) →



Example with Subarray, III. — Reading the file

```
!!!! real garray(20,30)                                ! these HPF-like comment lines !
!!!! PROCESSORS procs(2, 3)                            ! explain the data distribution !
!!!! DISTRIBUTE garray(BLOCK,BLOCK) onto procs ! used in this MPI program !
real larray(10,10) ; integer (kind=MPI_OFFSET_KIND) disp,offset; disp=0; offset=0
ndims=2 ; psizes(1)=2 ; period(1)=.false. ; psizes(2)=3 ; period(2)=.false.
call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, psizes, period,
call MPI_COMM_RANK(comm, rank, ierror)                 .TRUE., comm, ierror)
call MPI_CART_COORDS(comm, rank, ndims, coords, ierror)

gsizes(1)=20 ; lsizes(1)= 10 ; starts(1)=coords(1)*lsizes(1)
gsizes(2)=30 ; lsizes(2)= 10 ; starts(2)=coords(2)*lsizes(2)
call MPI_TYPE_CREATE_SUBARRAY(ndims, gsizes, lsizes, starts,
                           MPI_ORDER_FORTRAN, MPI_REAL, subarray_type, ierror)
call MPI_TYPE_COMMIT(subarray_type , ierror)

call MPI_FILE_OPEN(comm, 'exa_subarray_testfile', MPI_MODE_CREATE +
                  MPI_MODE_RDWR, MPI_INFO_NULL, fh, ierror)
call MPI_FILE_SET_VIEW (fh, disp, MPI_REAL, subarray_type, 'native',
                       MPI_INFO_NULL, ierror)
call MPI_FILE_READ_AT_ALL(fh, offset, larray, lsizes(1)*lsizes(2), MPI_REAL,
                         status, ierror)
```

See also 12-(1)
subarray

tour

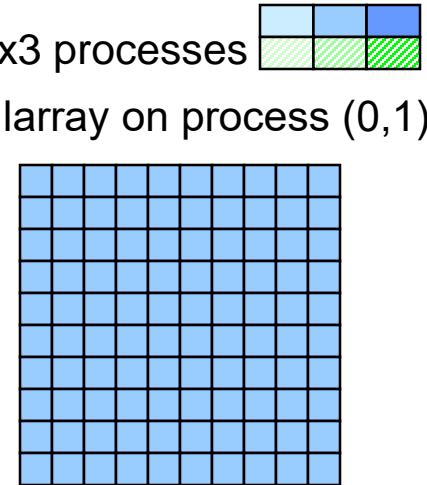
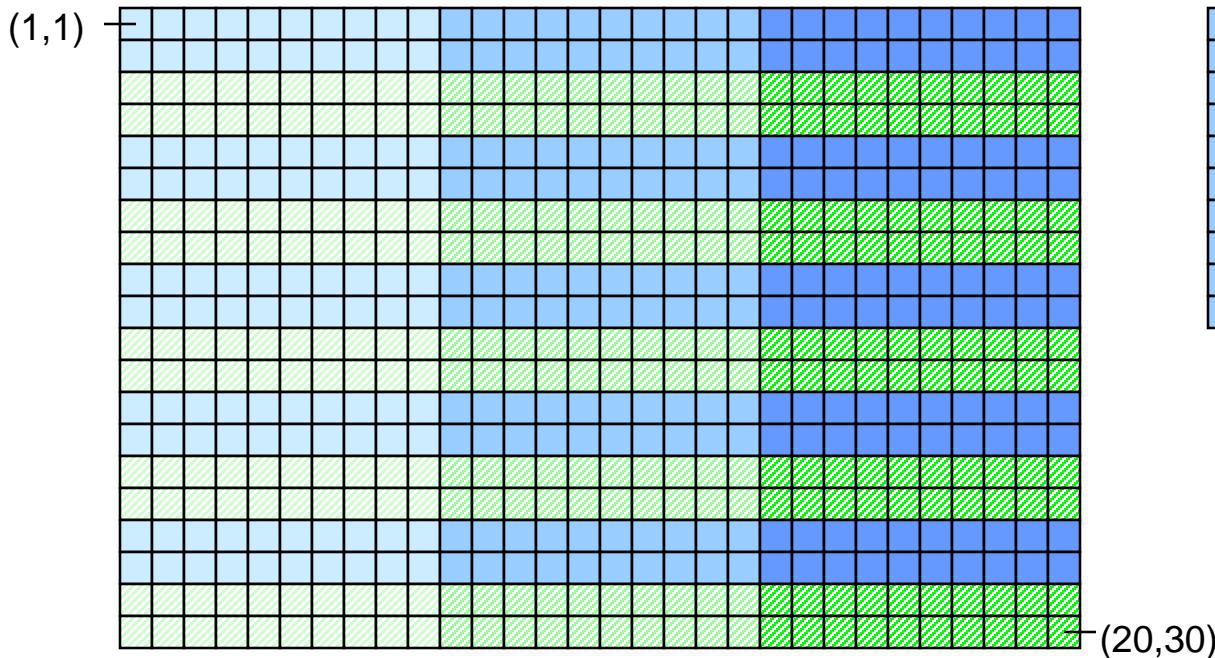
Example with Subarray, IV.

- All MPI coordinates and indices start with 0, even in Fortran, i.e. with MPI_ORDER_FORTRAN
- MPI indices (here **starts**) may differ (~~/~~) from Fortran indices
- Block distribution on 2*3 processes:

rank = 0 coords = (0, 0) starts = (0, 0) garray(1:10, 1:10) = larray (1:10, 1:10)	rank = 1 coords = (0, 1) starts = (0, 10) garray(1:10, 11:20) = larray (1:10, 1:10)	rank = 2 coords = (0, 2) starts = (0, 20) garray(1:10, 21:30) = larray (1:10, 1:10)
rank = 3 coords = (1, 0) starts = (10, 0) garray(11:20, 1:10) = larray (1:10, 1:10)	rank = 4 coords = (1, 1) starts = (10, 10) garray(11:20, 11:20) = larray (1:10, 1:10)	rank = 5 coords = (1, 2) starts = (10, 20) garray(11:20, 21:30) = larray (1:10, 1:10)

Example with Darray, I.

- Distribution scheme: (CYCLIC(2), BLOCK)
- Cyclic distribution in first dimension with strips of length 2
- Block distribution in second dimension
- distribution of global garray onto the larray in each of the 2x3 processes
- garray on the file:
- e.g., larray on process (0,1):



Example with Darray, II.

```
!!!! real garray(20,30) ! these HPF-like comment lines !
!!!! PROCESSORS procs(2, 3) ! explain the data distribution!
!!!! DISTRIBUTE garray(CYCLIC(2),BLOCK) onto procs !used in this MPI program!
real larray(10,10); integer (kind=MPI_OFFSET_KIND) disp, offset; disp=0; offset=0
call MPI_COMM_SIZE(comm, size, ierror)
ndims=2 ; psizes(1)=2 ; period(1)=.false. ; psizes(2)=3 ; period(2)=.false.
call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, psizes, period,
                     .TRUE., comm, ierror)
call MPI_COMM_RANK(comm, rank, ierror)
call MPI_CART_COORDS(comm, rank, ndims, coords, ierror)
gsizes(1)=20 ; distribs(1)= MPI_DISTRIBUTE_CYCLIC; dargs(1)=2
gsizes(2)=30 ; distribs(2)= MPI_DISTRIBUTE_BLOCK; dargs(2)=
               MPI_DISTRIBUTE_DFLT_DARG
call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, gsizes, distribs, dargs,
                            psizes, MPI_ORDER_FORTRAN, MPI_REAL, darray_type, ierror)
call MPI_TYPE_COMMIT(darray_type, ierror)
call MPI_FILE_OPEN(comm, 'exa_subarray_testfile', MPI_MODE_CREATE +
                  MPI_MODE_RDWR, MPI_INFO_NULL, fh, ierror)
call MPI_FILE_SET_VIEW (fh, disp, MPI_REAL, darray_type, 'native',
                       MPI_INFO_NULL, ierror)
call MPI_FILE_READ_AT_ALL(fh, offset, larray, 10*10, MPI_REAL, istatus, ierror)
```

Example with Darray, III.

- Cyclic distribution in first dimension with strips of length 2
- Block distribution in second dimension
- Processes' tasks:

<pre>rank = 0 coords = (0, 0) [1: 2 5: 6 9:10 13:14 17:18] garray(1:10, 1:10) = larray (1:10, 1:10)</pre>	<pre>rank = 1 coords = (0, 1) [1: 2 5: 6 9:10 13:14 17:18] garray(11:20, 1:10) = larray (1:10, 1:10)</pre>	<pre>rank = 2 coords = (0, 2) [1: 2 5: 6 9:10 13:14 17:18] garray(21:30, 1:10) = larray (1:10, 1:10)</pre>
<pre>rank = 3 coords = (1, 0) [3: 4 7: 8 11:12 15:16 19:20] garray(1:10, 1:10) = larray (1:10, 1:10)</pre>	<pre>rank = 4 coords = (1, 1) [3: 4 7: 8 11:12 15:16 19:20] garray(11:20, 1:10) = larray (1:10, 1:10)</pre>	<pre>rank = 5 coords = (1, 2) [3: 4 7: 8 11:12 15:16 19:20] garray(21:30, 1:10) = larray (1:10, 1:10)</pre>

5 Aspects of Data Access

- Direction: Read / Write
- Positioning [realized via routine names]
 - explicit offset (_AT)
 - individual file pointer (no positional qualifier)
 - shared file pointer (_SHARED or _ORDERED)
(different names used depending on whether non-collective or collective)
- Coordination
 - non-collective
 - collective (_ALL)
- Synchronism
 - blocking
 - nonblocking (I) and split collective (_BEGIN, _END)
- Atomicity, [realized with a separate API: MPI_File_set_atomicity]
 - non-atomic (default)
 - atomic: to achieve sequential consistency for conflicting accesses on same fh in different processes

All Data Access Routines

positioning	synchronism	coordination		split collective
		noncollective	collective	
explicit offsets	blocking	READ_AT	READ_AT_ALL	READ_AT_ALL_BEGIN
		WRITE_AT	WRITE_AT_ALL	READ_AT_ALL_END
	nonblocking	IREAD_AT	IREAD_AT_ALL	WRITE_AT_ALL_BEGIN
		IWRITE_AT	IWRITE_AT_ALL	WRITE_AT_ALL_END
individual file pointers	blocking	READ	READ_ALL	READ_ALL_BEGIN
		WRITE	WRITE_ALL	READ_ALL_END
	nonblocking	IREAD	IREAD_ALL	WRITE_ALL_BEGIN
		IWRITE	IWRITE_ALL	WRITE_ALL_END
shared file pointer	blocking	READ_SHARED	READ_ORDERED	READ_ORDERED_BEGIN
		WRITE_SHARED	WRITE_ORDERED	READ_ORDERED_END
	nonblocking	IREAD_SHARED	N/A	WRITE_ORDERED_BEGIN
		IWRITE_SHARED		WRITE_ORDERED_END

Read e.g. **MPI_FILE_READ_AT**

New in MPI-3.1

Explicit Offsets

e.g. MPI_File_read_at(fh, offset, *buf*, count, datatype, *status*)

- attempts to read **count** elements of **datatype**
- starting **offset * units of etype** from begin of view (= **displacement**)
- the sequence of basic datatypes of **datatype** (= signature of **datatype**) must match contiguous copies of the **etype** of the current view
- EOF can be detected by noting that the amount of data read is less than **count**
 - i.e. EOF is no error!
 - use MPI_Get_count(status, datatype, *recv_count*)



language bindings – see MPI Standard



and mpi4py

Individual File Pointer, I.

e.g. `MPI_File_read(fh, buf, count, datatype, status)`

- same as “*Explicit Offsets*”, except:
- the offset is the current value of the **individual file pointer** of the calling process
- the individual file pointer is updated by
$$\text{new_fp} = \text{old_fp} + \frac{\text{elements(datatype)}}{\text{elements(etype)}} * \text{count}$$
i.e. it points to the next etype after the last one that will be accessed
(if EOF is reached, then recv_count is used, see previous slide)



language bindings – see MPI Standard



and mpi4py

Individual File Pointer, II.

`MPI_File_seek(fh, offset, whence)`

- set individual file pointer fp:
 - set fp to offset – if whence=MPI_SEEK_SET
 - advance fp by offset – if whence=MPI_SEEK_CUR
 - set fp to EOF+offset – if whence=MPI_SEEK_END

`MPI_File_get_position(fh, offset)`

`MPI_File_get_byte_offset(fh, offset, disp)`

- to inquire offset
- to convert offset into byte displacement
[e.g. for *disp* argument in a new view]

Fortran C/C++

Python

language bindings – see MPI Standard

and mpi4py

MPI-IO Exercise 2: Using fileviews and individual filepointers

- Copy to your local directory:

C

```
cp ~/MPI/tasks/C/Ch13/mpi_io_exa2_skel.c my_exa2.c
```

Fortran

```
cp ~/MPI/tasks/F_30/Ch13/mpi_io_exa2_skel_30.f90 my_exa2_30.f90
```

Python

```
cp ~/MPI/tasks/PY/Ch13/mpi_io_exa2_skel.py my_exa2.py
```

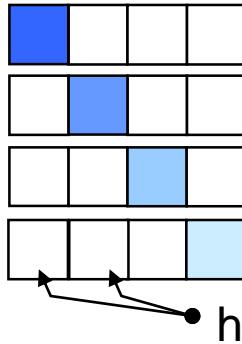
- Tasks:

- Each MPI-process of `my_exa2` should write one character to a file:
 - process “rank=0” should write an ‘a’
 - process “rank=1” should write an ‘b’
 - ...
 - Use a 1-dimensional fileview with `MPI_TYPE_CREATE_SUBARRAY`
 - The pattern should be repeated 3 times, i.e., four processes should write: “abcdabcdabcd”
 - Please, substitute “_____” in your `my_exa2.c` / `_30.f90`
- Edit; compile; `rm -f my_test_file`; `mpirun ...` (always remove `my_test_file` before re-run)
- `cat my_test_file`; `echo`; `wc -c my_test_file` (verifying the result)

MPI-IO Exercise 2: Using fileviews and individual filepointers, continued



etype = MPI_CHARACTER / MPI_CHAR



filetype process 0

filetype process 1

filetype process 2

filetype process 3

tiling a file with filetypes:

a b c d a b c d a b c d file

• file displacement = 0 (number of header bytes), **identical on all processes**

Otherwise optimization may be impossible

a	a	a
b	b	b
c	c	c
d	d	d

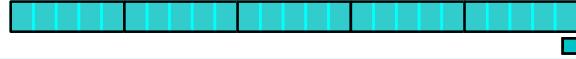
view of process 0

view of process 1

view of process 2

view of process 3

During the Exercise (25 min.)



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



And have fun with this longer 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:



Ask yourself, whether the datatype is a 1- or higher-dimensional array?



And don't forget that counts are normally elements and not bytes!



And to look at the declaration of the buffer is also helpful
to answer the last question

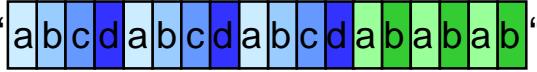


MPI-IO Advanced Exercise 2b+c: Append

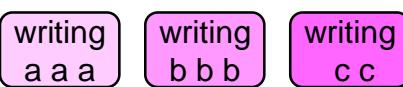
- `rm -f my_test_file`
- Run program of Exercise 1 with 4 processes:
`cat my_test_file ; echo ; wc -c my_test_file`


 Expected result ““ (12 characters)

2b) Please, make a copy of your result: `cp my_exa2.c my_exa2b.c or _30.f90`

- Set the displacement **disp** to the current filesize: Use **`MPI_File_get_size()`**
 (For the interface, please look into the MPI standard)
- Compile and run again (**without** removing `my_test_file`), now with 2 processes:
 Expected result: ““ (18 characters)
 
- Solution: `MPI/tasks/C/Ch13/solutions/mpi_io_exa2b.c` and `MPI/tasks/F_30/Ch13/solutions/mpi_io_exa2b_30.f90`

2c) Please, make a copy of your **original** result: `cp my_exa2.c my_exa2c.c or _30.f90`

- Use **`MPI_File_seek()`** to move all individual file pointers to the **end of the file**
 (For the interface, please look into the MPI standard)
- Again (**without** removing `my_test_file`), now with 3 processes
 Expected result: ““ (27 characters)
 
- **Caution:**– Existing file size should be a multiple of the new filetype size
 – Both OpenMPI and mpich may have a bug.
- Solution: `MPI/tasks/C/Ch13/solutions/mpi_io_exa2c.c` and `MPI/tasks/F_30/Ch13/solutions/mpi_io_exa2c_30.f90`



Outline – Block 3

- **Block 1**
 - Introduction [323]
 - **Definitions** [328]
 - Open / Close [330]
 - WRITE / **Explicit Offsets** [335]
 - Exercise 1 [336]
- **Block 2**
 - **File Views** [338]
 - **Subarray & Darray** [342]
 - I/O Routines Overview [350]
 - READ / Explicit Offsets [352]
 - **Individual File Pointer** [353]
 - Exercise 2 [355]
- **Block 3**
 - **Shared File Pointer** [358]
 - **Collective** [360]
 - Non-Blocking / Split Collective [364/365]
 - Other Routines [368]
 - Error Handling [369]
 - Implementation Restrictions [370]
 - **Summary** [371]
 - Exercise 3 [372]
 - Exercise 4 [373]

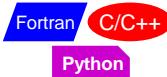
Shared File Pointer, I.

- same view at all processes mandatory!
- the offset is the current, *global* value of the **shared file pointer** of **fh**
- multiple calls [e.g. by *different processes*] **behave as** if the calls were **serialized**
- non-collective, e.g.

```
MPI_File_read_shared(fh, buf, count, datatype, status)
```

- collective calls are *serialized* in the **order** of the processes' ranks, e.g.:

```
MPI_File_read_ordered(fh, buf, count, datatype, status)
```



language bindings – see MPI Standard

and mpi4py

Shared File Pointer, II.

```
MPI_File_seek_shared(fh, offset, whence)
```

```
MPI_File_get_position_shared(fh, offset)
```

```
MPI_File_get_byte_offset(fh, offset, disp)
```

- same rules as with individual file pointers

Fortran C/C++

Python

language bindings – see MPI Standard

and mpi4py

Collective Data Access

- Explicit offsets / individual file pointer:
 - same as non-collective calls by all processes “of `fh`”
 - ***opportunity for best speed!!!***
- shared file pointer:
 - accesses are ordered by the ranks of the processes
 - optimization opportunity:
 - **first, locations within the file for all processes can be computed**
 - **then parallel physical data access by all processes**

Application Scenery, I.

- Scenery A:
 - Task: Each process has to read the whole file
 - Solution: **MPI_File_read_all**
= collective with individual file pointers,
with same view (displacement+etype+filetype)
on all processes
*[internally: striped-reading by several process, only once
from disk, then distributing with bcast]*
- Scenery B:
 - Task: The file contains a list of tasks,
each task requires different compute time
 - Solution: **MPI_File_read_shared**
= non-collective with a shared file pointer
(same view is necessary for shared file p.)

Application Scenery, II.

- Scenery C:

- Task: The file contains a list of tasks,
each task requires **the same** compute time
 - Solution: **MPI_File_read_ordered**
= **collective** with a **shared** file pointer
(same view is necessary for shared file p.)
 - or: **MPI_File_read_all**
= **collective** with **individual** file pointers,
different views: *filetype* with
**MPI_Type_create_subarray(1, nproc,
1, myrank, ..., datatype_of_task, *filetype*)**
[internally: *both may be implemented the same
and equally with following scenery D*]

Application Scenery, III.

- Scenery D:
 - Task: The file contains a matrix, block partitioning, each process should get a block
 - Solution: generate different filetypes with **MPI_Type_create_darray** or ..._subarray, the view on each process represents the block that should be read by this process, **MPI_File_read_at_all** with offset=0 (= collective with explicit offsets) reads the whole matrix collectively
[internally: striped-reading of contiguous blocks by several process, then distributed with “alltoall”]

Nonblocking Data Access

e.g. `MPI_File_iread(fh, buf, count, datatype, request)`
`MPI_Wait(request, status)`
`MPI_Test(request, flag, status)`

- analogous to MPI-1 nonblocking



language bindings – see MPI Standard



and mpi4py

May be deprecated in MPI-4.1
and removed in MPI-5

Split Collective Data Access, I.

- collective operations may be **split** into two parts:
 - start the split collective operation

e.g. `MPI_File_read_all_begin(fh, buf, count, datatype)`

- complete the operation and return the **status**

`MPI_File_read_all_end(fh, buf, status)`



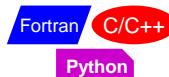
language bindings – see MPI Standard



and mpi4py

Split Collective Data Access, II.

- Rules and Restrictions:
 - the **MPI_..._begin** calls are collective
 - the **MPI_..._end** calls are collective, too
 - only one active (pending) split or regular collective operation per file handle at any time
 - split collective does not match ordinary collective
 - same **buf** argument in MPI_..._begin and MPI_..._end call
- opportunity to overlap file I/O and computation
- but also a valid implementation:
 - does all work within the MPI_..._begin routine, passes status in the MPI_..._end routine
 - passes arguments from MPI_..._begin to MPI_..._end, does all work within the MPI_..._end routine



language bindings – see MPI Standard

and mpi4py

Scenery – Nonblocking or Split Collective

- Scenery A:
 - Task: Each process has to read the whole file
 - Solution:
 - MPI_File_iread_all or MPI_File_read_all_begin
= collective with individual file pointers,
with same view (displacement+etype+filetype)
on all processes
*[internally: starting asynchronous striped-reading
by several process]*
 - then computing some other initialization,
 - MPI_Wait or MPI_File_read_all_end.
*[internally: waiting until striped-reading finished,
then distributing the data with bcast]*

Other File Manipulation Routines

- Pre-allocating space for a file [*collective call, may be expensive*]

`MPI_File_preallocate(fh, size)`

- Resizing a file [*collective call, may speed up first writing on a file*]

`MPI_File_set_size(fh, size)`

`size = 0` → current file content is erased.

Recommended, if the whole file should be overwritten.

- Querying file size

`MPI_File_get_size(filename, size)`

- Querying file parameters

`MPI_File_get_group(fh, group)`

`MPI_File_get_amode(fh, amode)`

- File info object

`MPI_File_set_info (fh, info)` [*collective call*]

`MPI_File_get_info(fh, info_used)`

Returns a new info object that contains the current setting of **all hints** used by the system related to this open file:

- provided by the application, and
- provided by the system

Fortran C/C++

language bindings – see MPI Standard

Python

and mpi4py

MPI I/O Error Handling

- File handles have their own error handler
- Default is MPI_ERRORS_RETURN,
i.e. **non-fatal**
[vs message passing: MPI_ERRORS_ARE_FATAL]
- Default is associated with MPI_FILE_NULL
[vs message passing: with MPI_COMM_WORLD]
- Changing the default, e.g., after MPI_Init:
 - C/C++ `MPI_File_set_errhandler(MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL);`
 - Fortran `CALL MPI_FILE_SET_ERRHANDLER(MPI_FILE_NULL,MPI_ERRORS_ARE_FATAL,ierr)`
 - Python `MPI.FILE_NULL.Set_errhandler(MPI.ERRORS_ARE_FATAL)`
- MPI is *undefined* after first erroneous MPI call
- but a **high quality** implementation
will support I/O error handling facilities

Implementation-Restrictions

- ROMIO based MPI libraries:
 - datarep = “internal” and “external32” is still not implemented
 - User-defined data representations are not supported

MPI-I/O: Summary

- Rich functionality provided to support various data representation and access
- MPI I/O routines provide flexibility as well as portability
- Collective I/O routines can improve I/O performance
- ROMIO from Argonne was an initial implementation of MPI I/O
- Available (nearly) on every MPI implementation
- Parallel MPI I/O also used as basis for important I/O packages:
 - Parallel HDF5
<https://portal.hdfgroup.org/display/HDF5/Introduction+to+Parallel+HDF5>
 - Parallel NetCDF, e.g.,
<https://en.wikipedia.org/wiki/NetCDF#Parallel-NetCDF>

MPI-IO Exercise 3: Collective ordered I/O

- Copy to your local directory:

C `cp ~/MPI/tasks/C/Ch13/mpi_io_exa3_skel.c my_exa3.c`

Fortran `cp ~/MPI/tasks/F_30/Ch13/mpi_io_exa3_skel_30.f90 my_exa3_30.f90`

Python `cp ~/MPI/tasks/PY/Ch13/mpi_io_exa3_skel.py my_exa3.py`

- Tasks:

- Substitute the write call with individual filepointers by a collective write call with shared filepointers
 - Edit your `my_exa3.c` / `_30.f90`

- Compile; `rm -f my_test_file`; `mpirun ...` (always remove `my_test_file` before re-run)
- `cat my_test_file`; `echo`; `wc -c my_test_file` (verifying the result)

During the Exercise (25 min.)



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



And have fun with this longer 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:

This exercise is mainly removing  all about the fileview.



*With the shared file pointer and collective writing, this exercise
is a one-line problem, isn't it?*



Good luck!



MPI–IO Exercise 4: I/O Benchmark

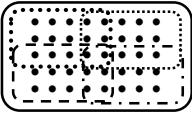
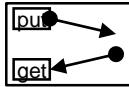
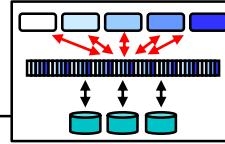
- Use:
`MPI/tasks/F_30/Ch13/mpi_io_exa4_30.f90`
(my apologies that there is only a Fortran version)
- Tasks:
 - Compile and execute `mpi_io_exa4` on 2, 4 and 8 MPI processes.
 - Duplicate “`WRITE_ALL & READ_ALL`” block
and substitute by non-collective “`WRITE & READ`”.
 - Compare collective and non-collective I/O.
 - Double the value of `gsize` and compile and execute again.



For private notes

For private notes

Chap.14 MPI and Threads

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication 
12. Derived datatypes 
13. Parallel file I/O

14. MPI and Threads

– e.g., hybrid MPI and OpenMP

15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice



MPI rules with OpenMP / Automatic SMP-parallelization

- Special MPI init for multi-threaded MPI processes:

New in MPI-2.0

```
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 main thread¹⁾ 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

¹⁾ Main thread = thread that called MPI_Init_thread.

Recommendation:

Start MPI_Init_thread from OpenMP master thread → OpenMP master thread = MPI main thread

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!

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

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

}
```

No barrier inside

Barriers needed to prevent data races

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 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 use-case:
 - Replicated application data in one shared memory window per CPU or per ccNUMA node
- General
 - Efficient placement of cores and processes and threads on ccNUMA nodes

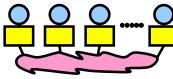
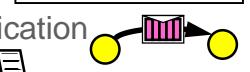
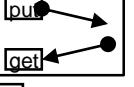
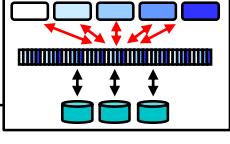
Partitioned Point-to-Point Communication

- MPI-4.0:
Partitioned communication is “partitioned“ because it allows for multiple contributions of data to be made, potentially, from multiple actors (e.g., threads or tasks) in an MPI process to a single communication operation.
- A point-to-point operation (i.e., send or receive)
 - can be split into partitions,
 - and each partition is filled and then “send” with MPI_Pready by a thread;
 - And same for receiving.
- Technically provided as a new form of persistent communication.

For private notes

For private notes

Chap.15 Probe, Persistent Requests, Cancel

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication 
12. Derived datatypes 
13. Parallel file I/O
14. MPI and threads

15. Probe, Persistent Requests, Request_free, Cancel

16. Process creation and management
17. Other MPI features
18. Best Practice

Two skip-points:
 After probe-slide (skip 5 slides)
 Here, skip this section (skip 6 slides) → 

Short tour – 3 slides


Probing a message

- Question: The receiver wants to allocate the receive buffer prior to MPI_Recv, but does not know the sent message size
- Solution: Look at the message envelop (status) to examine the message count prior to MPI_Recv and then, allocate an appropriate receive buffer
- Two methods:
 - MPI_Probe or MPI_Iprobe
 - → status of next unreceived message
 - After buffer allocation: Normal MPI_Recv
 - May cause problems in a multi-threaded MPI process

Within one MPI process, thread A may call MPI_PROBE. Another thread B may steal the probed message. Thread A calls MPI_RECV, but may not receive the probed message.

New in MPI-3.0

Matching Probe

- MPI_Iprobe(source, tag, comm, flag, message, status) or
- MPI_Mprobe(source, tag, comm, message, status)

together with Matched Receive

- MPI_Mrecv(buf, count, datatype, message, status) or
- MPI_Imrecv(buf, count, datatype, message, request)

MPI_Message handle,

e.g., stored in a thread-local variable

- After buffer allocation: MPI_(I)mrecv exactly receives the probed message
- Multiple threads within one MPI process can probe and receive several messages in parallel
- See also MPI_Ibarrier example in course chapter 6-(2) 



Operations and Procedures, (non)blocking / (non-)local

- **MPI operations** consist of four stages:
 - Initialization, starting, completion, freeing
- **MPI operations** can be
 - **Blocking**: all four stages are combined in a single complete/blocking procedure.
→ which returns when operation has completed.
 - **Nonblocking**: → next slide & Chapter 4
 - **Persistent**: → 2nd next slide

- **MPI procedures** can be
 - **Non-local**: returning may require, during its execution, some specific semantically-related MPI procedure to be called on another MPI process.
 - **Local**: is not non-local. (See also discussion of “weak local” in course chapter 18 

- **MPI procedures** (if they implement an operation or parts of it) can be
 - **Completing**: on return, all resources (e.g., buffers or array arg.s) can be reused.
 - **Incomplete**: return before resources can be reused.
 - **Nonblocking**: incomplete AND local / **Blocking**: Completing OR non-local.

- **Examples:**
 - Nonblocking:
 - Incomplete & local: MPI_Isend, MPI_Irecv, MPI_Ibcast, MPI_Send_init
 - Blocking:
 - Completing & non-local: MPI_Send, MPI_Recv, MPI_Bcast
 - Incomplete & non-local: MPI_Mprobe, MPI_Bcast_init
 - Completing & local: MPI_Bsend, MPI_Rsend, MPI_Mrecv

Orthogonal concept,
although in most cases:

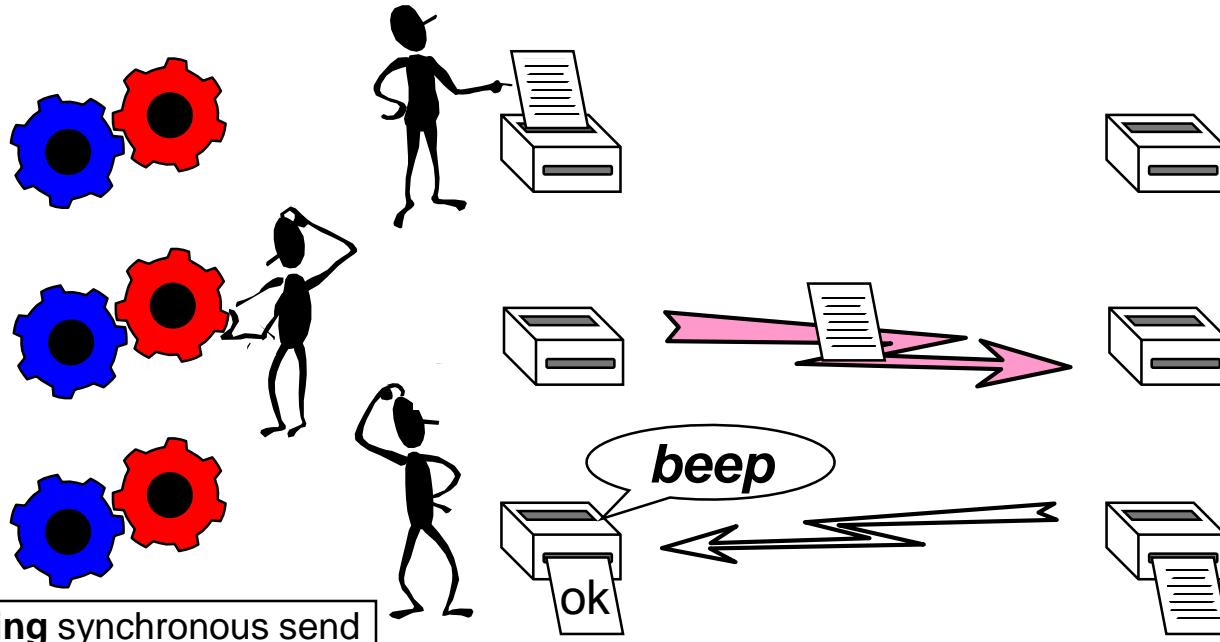
- Incomplete/nonblocking communication proc.
→ local
- Complete/blocking communication proc.
→ non-local
(with some exceptions)

Nonblocking Operations

Nonblocking operations consist of:

New in MPI-4.0

- A **incomplete**/nonblocking procedure call: it returns **immediately** and allows the sub-program to perform other work → stages initialization + starting
- At some later time the sub-program must **test** or **wait** for the completion of the nonblocking operation → stages completion + freeing



New in MPI-4.0

© 2000-2021 HLRS, Rolf Rabenseifner • REC → [online](#)

MPI course → Chap. 15 Probe, Persistent Requests, Cancel

Goal:
Enables additional optimizations
within the MPI library

Persistent Requests

For communication calls with identical argument lists in each loop iteration (only buffer content changes):

New in MPI-4.0

Stage

initialization

- **MPI_(,B,S,R)Send_init** and **MPI_Recv_init**

- Creates a persistent MPI_Request handle
 - Status of the handle is initiated as *inactive*
 - Local calls (does not communicate)
 - It only setups the argument list

New in MPI-4.0

- **MPI_Bcast_init** ..., also for collective operations

- Blocking & collective calls (may communicate)

Recommendation:
Never free an **active** request handle.
Active request handles should be completed with WAIT or TEST

starting

- **MPI_Start(request [,ierror]) / MPI_Startall(cnt, requests [,ierror])**

- Starts the communication call(s) as nonblocking call(s), i.e., handle gets *active*

completion

- To be completed with regular MPI_Wait... / MPI_Test... calls → *inactive*

- **MPI_Request_free** to finally free such a handle

freeing

- Usage sequence: init Loop(Start Wait/Test) Request_free

Persistent inactive request → **active**

Completes an active request handle
→ **inactive**

Free the **inactive**
persistent request
handle



New in MPI-4.0

© 2000-2021 HLRS, Rolf Rabenseifner REC → [online](#)

MPI course → Chap. 15 Probe, Persistent Requests, Cancel

MPI_Request_free

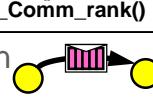
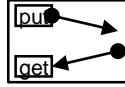
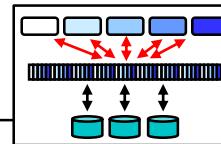
- MPI_Request_free really useful only for *inactive* persistent requests i.e., after such Loop(Start Wait/Test),
i.e., not after Start
- MPI_Request_free for *active* communication request:
 - Marks a request handle for deallocation
 - Deallocation will be done after *active* communication completion
 - May be used only for *active* send-request to substitute MPI_Wait, but highly dangerous when there is no other 100% guarantee that the send-buffer can be reused.
 - Active send handle is produced with **MPI_I(,s,b,r)send** or **MPI_(,S,B,R)send_init + MPI_Start**
 - Should never be used for *active receive* requests.

MPI_Cancel

- Marks a active nonblocking communication handle for cancellation.
- MPI_Cancel is a local call, i.e., returns immediately.
- **Subsequent call to MPI_Wait must return irrespective of the activities of other processes.**
- **Either the cancellation or the communication succeeds, but not both.**
- MPI_Test_cancelled(wait_status, flag [,ierror])
 - flag = true → cancellation succeeded, communication failed
 - flag = false → cancellation failed, communication succeeded
- **Comment: Do not use it – may be reason for worse performance**
- **Since MPI-4.0: MPI_Cancel of send requests is deprecated**

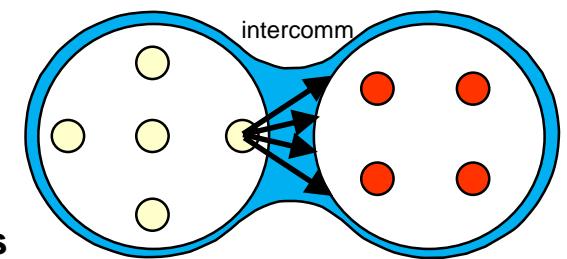
For private notes

Chap.16 Process Creation and Management

1. MPI Overview 
2. Process model and language bindings
MPI_Init()
MPI_Comm_rank()
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication 
12. Derived datatypes 
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel

16. Process creation and management

- Spawning additional processes
 - Singleton MPI_INIT
 - Connecting two independent sets of MPI processes
17. Other MPI features
 18. Best Practice



Skip this section
(4 slides + 4 hidden slides) →  

Short tour – 3 slides 

Dynamic Process Management

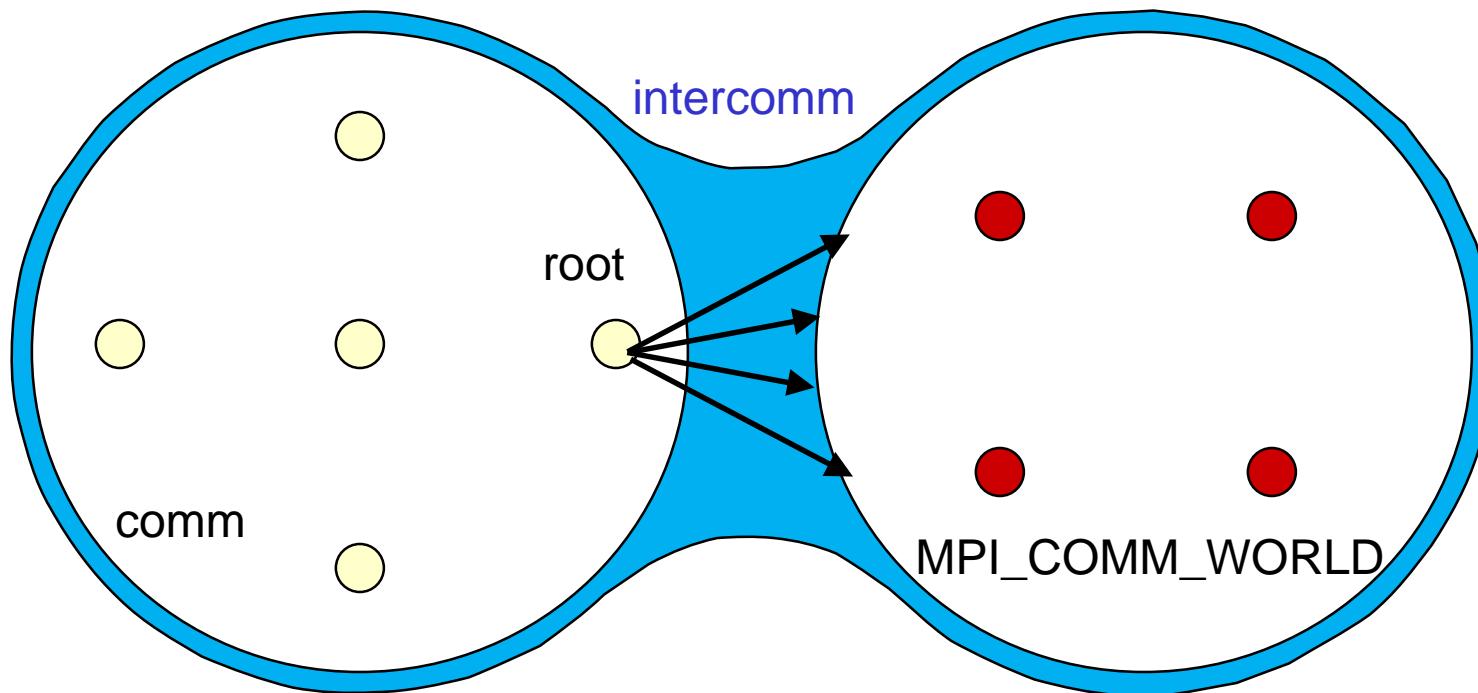
- Two independent goals
 1. starting additional MPI processes
 2. connecting independently started MPI processes
- Issues
 - maintaining simplicity, flexibility, and correctness
 - interaction with operating systems, resource manager, and process manager
- Starting additional MPI processes with the **spawn interfaces**:
 - at initiators (parents):
 - Spawning new processes is **collective**, returning an intercommunicator.
 - Local group is **group of spawning processes**.
 - Remote group is **group of spawned processes**.
 - at spawned processes (children):
 - New processes have own **MPI_COMM_WORLD**
 - **MPI_Comm_get_parent()** returns intercommunicator to parent processes



tour

Dynamic Process Management — Get the **intercomm**,

I.



Parents:

`MPI_Comm_spawn (████, root, comm, intercomm, ...)`

Children:

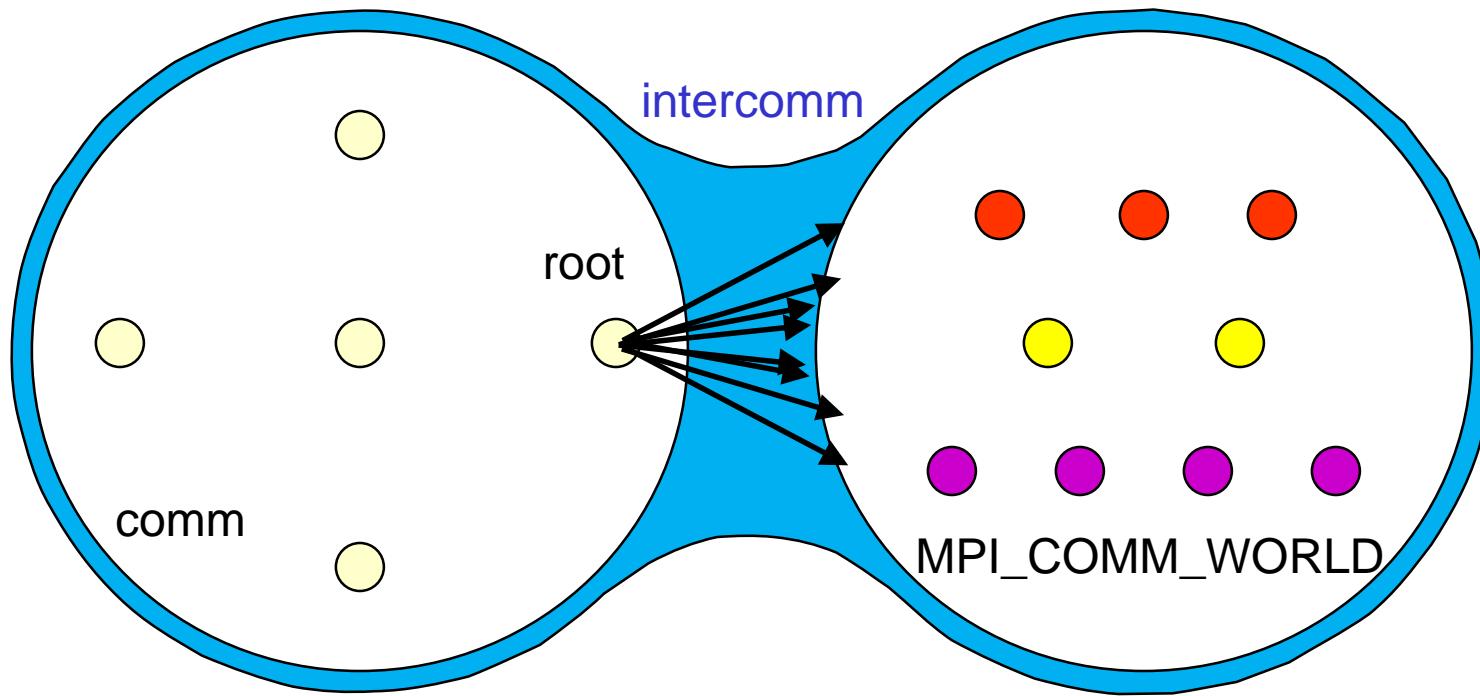
`MPI_Init(...)`
`MPI_Comm_get_parent(intercomm)`

Step to next **hidden** slide on
MPI_Comm_spawn_multiple



skipped

Dynamic Process Management — Get the *intercomm*, II.



Parents:

`MPI_Comm_spawn
_multiple (3, [color bars],
root, comm, intercomm,...)`

Children:

`MPI_Init(...)
MPI_Comm_get_parent(intercomm)`



Major Problem with Spawning of Dynamic Processes

- **Typical static environment**
 - MPI processes within a batch job (\rightarrow qsub)
 - Dedicated cores/CPUs for each MPI process
 - Why?
 - Communication and load balancing requires:
 - All communication partners must be available
 - Otherwise idle time due to polling strategy within MPI_Recv
 - Alternative: Gang-scheduling
 - i.e., all MPI processes are running or all are sleeping
 - In most cases: Not available or does not work correctly
- **Dynamic spawning of additional processes**
 - CPUs not available within current batch job, or
 - CPUs are available,
but wasted cycles between MPI_Init and MPI_Spawn

→ **In most cases not useful**

Step through all slides:
Singleton Init
MPI_Intercomm_merge
MPI_Comm_connect
MPI_Comm_join



End of short tour



Dynamic Process Management — Singleton INIT

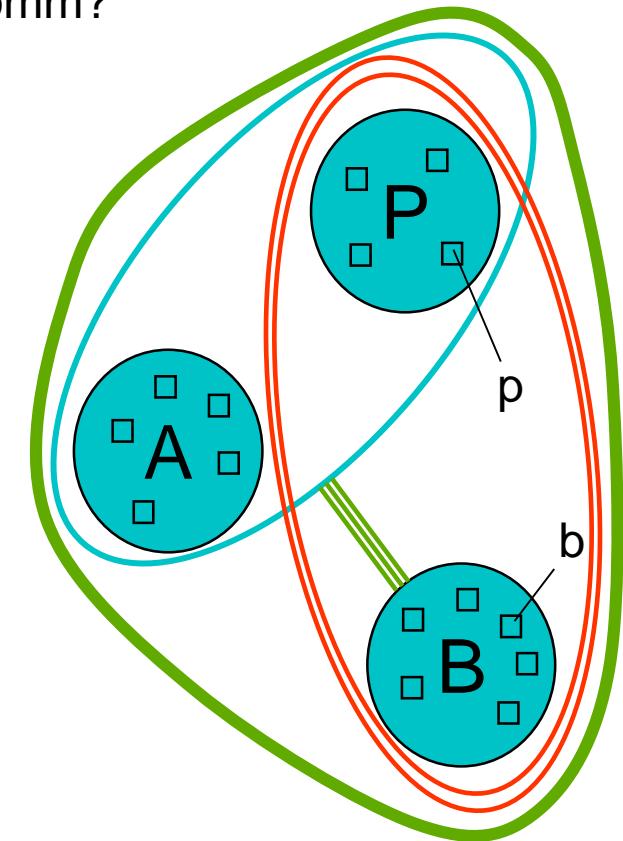
- High quality MPI's will allow single processes to start, call `MPI_Init()`, and later spawn other MPI processes
- This approach supports
 - parallel plug-ins to sequential APPs
 - other transparent uses of MPI
- Provides a means for using MPI without having to have the “main” program be MPI specific.



skipped —

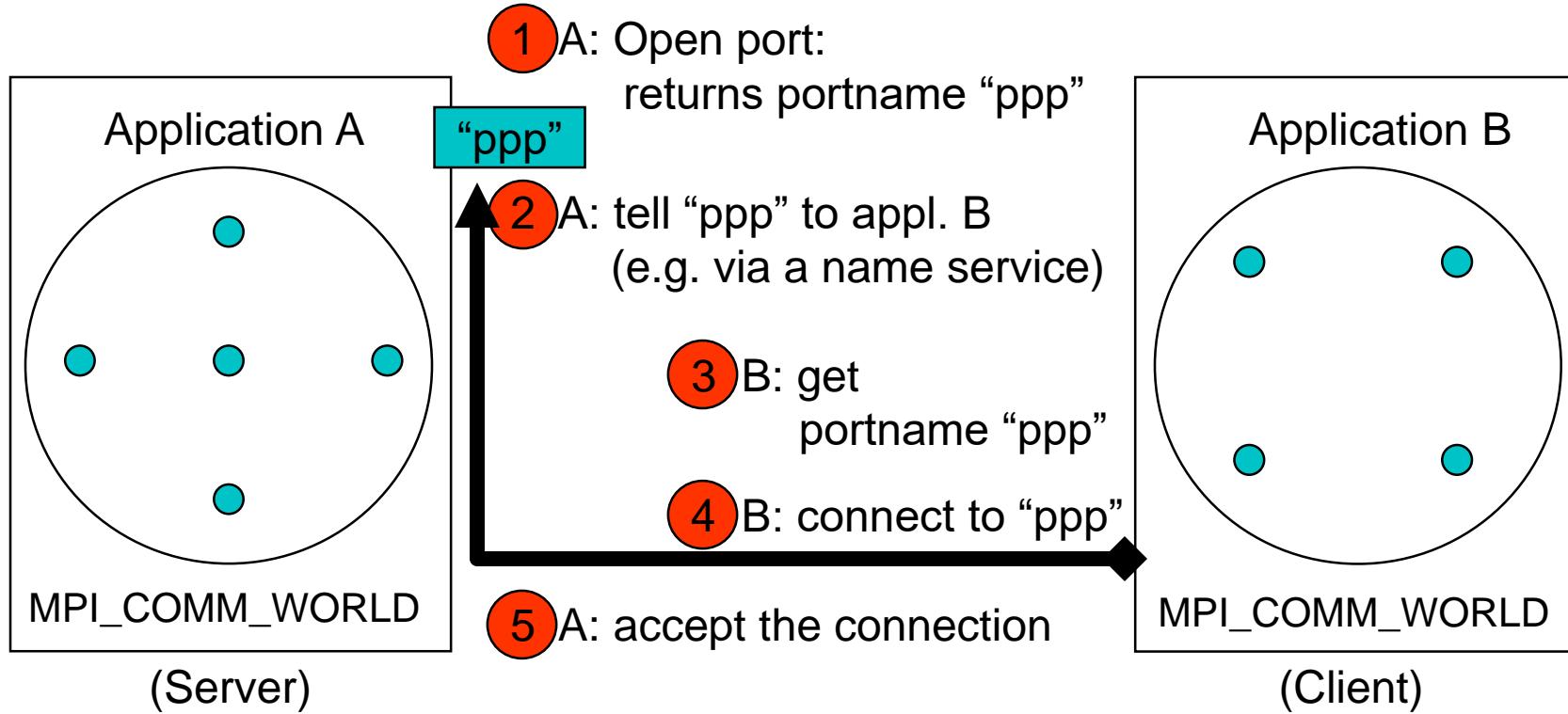
Dynamic Process Management — Multi-merging, a Challenge

- If a communicator P spawns A and B sequentially, how can P, A and B communicate in a single intracomm?
- The following sequence supports this:
 - P+A merge to form intracomm PA
 - P+B merge to form intracomm PB
 - PA and B create intercomm PA+B
[using PB as peer, with **p**, **b** as leaders]
 - PA+B merge to form intracomm PAB
- Routine: **MPI_Intercomm_merge**
- This is not very easy,
but does work



skipped

Dynamic Process Management — Establishing Communication



Result: An intercommunicator between both original communicators



skipped —

Dynamic Process Management — Another way

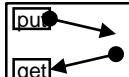
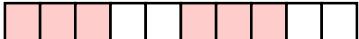
- Another way to establish MPI communication
- `MPI_Comm_join(fd, intercomm)`
- joins by an intercommunicator
- two independent MPI processes
- that are connected with Berkley Sockets
of type `SOCK_STREAM`

end

For private notes

Chap.17 Other MPI Features

Together with this chapter,
you'll have received a (nearly) complete
overview on all features of MPI-3.1
+ an overview on MPI-4.0

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication
12. Derived datatypes 
13. Parallel file I/O
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features
18. Best Practice

Further MPI-3.1/-4.0 chapters and sections:

- | | |
|--|--|
| [1, 2] | – Introduction, Terms & Conventions |
| [12/13.1-3] | – Generalized requests |
| [14/15] | – Profiling & Tools support |
| [15+16/16+17] | – Deprecated & removed interfaces |
| [--/18] New in MPI-4.0 | – Semantic changes & warnings |
| [17.2/19.3] | – Language interoperability |
| [A] | – Language bindings summary |
| [--/A.2] New in MPI-4.0 | – Semantics of op-related procedures |
| [B] | – Change-log |

Other MPI Features

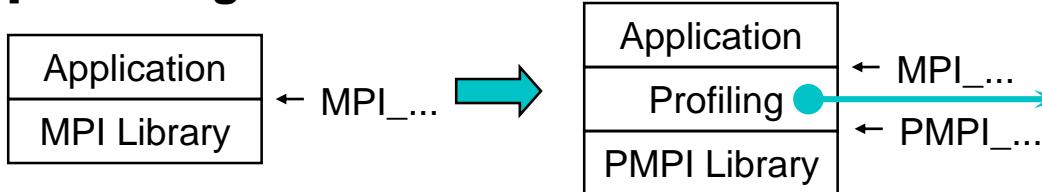
Further MPI-3.1/-4.0 chapters and sections:

- [1, 2] **Introduction, Terms & Conventions**
 - This course is an introduction to MPI
 - MPI Chap. 1+2 gives a good overview of
 - the MPI standard, and
 - all major terms used within this standard
- [12/13.1-3] **Generalized requests**
 - If you want to use the MPI request handling for an own interface.
 - Needed, e.g., if you want to implement a part of the MPI standard (e.g. I/O) as a portable software for all MPI libraries.

Other MPI Features

Further MPI-3.1/-4.0 chapters and sections:

- [14/15.1-2] **Profiling Interface**



e.g. with Vampir

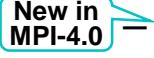
- [14/15.3] **The MPI Tool Information Interface**

- Access to **internal performance and control data** of an MPI library
- Query API for all MPI_T variables / 2 phase approach
 - Setup: Query all variables and select from them
 - Measurement: allocate handles and read variables



Other MPI Features

Further MPI-3.1/-4.0 chapters and sections:

- [2.6.1] **Deprecated and Removed Names and Functions**
 - Table 2.1 on page 18/[25](#) presents a good overview
- [15+16/16+17] **Deprecated & Removed Interfaces**
-  [--/18] **Semantic changes & warnings**
- [17.2/19.3] **Language interoperability**
 - C / C++ / Fortran language interoperability
 - between languages in same processes
 - messages transferred from one language to another
- [A] **Language bindings summary**
 - [A.1] All constants, predefined handles, type and callback prototypes
 -  [A.2] Summary of the semantics of all operation-related MPI procedures
- [B] **Change-log**
 - What is new in MPI-4.0 / 3.1 / 3.0 / 2.2 / 2.1

MPI provider

- The vendor of your computers
- MPICH – the public domain MPI library from Argonne
 - for all UNIX platforms
 - for Windows NT, ...
 - www.mcs.anl.gov/mpi/mpich/
- OpenMPI www.open-mpi.org
- see also at http://en.wikipedia.org/wiki/Message_Passing_Interface
- Standard, errata, printed books at [www.mpi-forum.org](http://www mpi-forum.org)

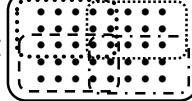
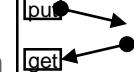
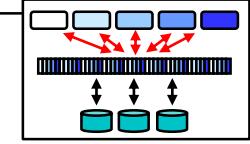
Caution: Open**MPI** is an MPI library, whereas Open**MP** is a shared memory parallel programming model

For private notes

For private notes

For private notes

Chap.18 Best Practice

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication 
4. Nonblocking communication 
5. The New Fortran Module mpi_f08
6. Collective communication 
7. Error Handling
8. Groups & communicators, environment management 
9. Virtual topologies 
10. One-sided communication 
11. Shared memory one-sided communication 
12. Derived datatypes
13. Parallel file I/O 
14. MPI and threads
15. Probe, Persistent Requests, Cancel
16. Process creation and management
17. Other MPI features

18. Best Practice

- Parallelization strategies (e.g. Foster's Design Methodology) 
- Performance considerations 
- Pitfalls 

Foster's design methodology

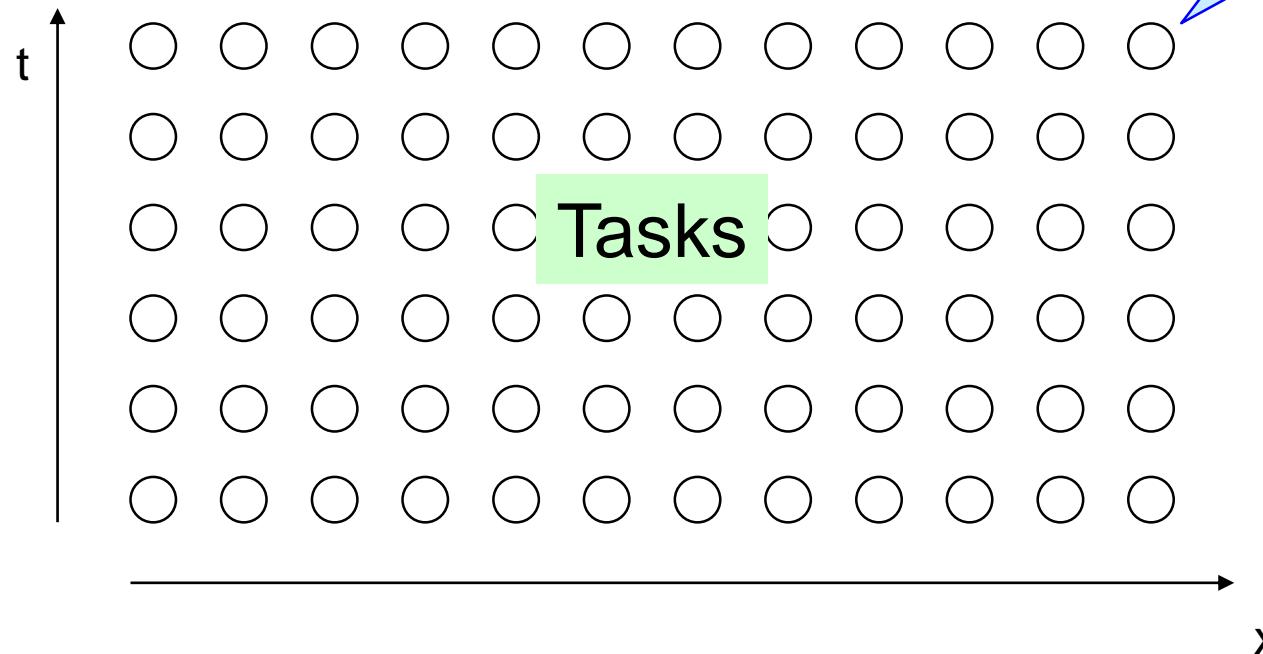
- Partitioning
 - Dividing the computation and the data into primitive “tasks”
- Communication
 - Determining how tasks will communicate with each other
- Agglomeration
 - Grouping tasks into larger tasks to improve performance or simplify programming
- Mapping
 - Assigning tasks to physical processors (here MPI processes)

Foster's design methodology – Example

- Example:
 - $T(x,t) = \text{function}(T(x, t-1), T(x-1, t-1), T(x+1, t-1))$
(e.g., 1-dim heat equation with time integration)

Calcu-
lation
of $T(x,t)$

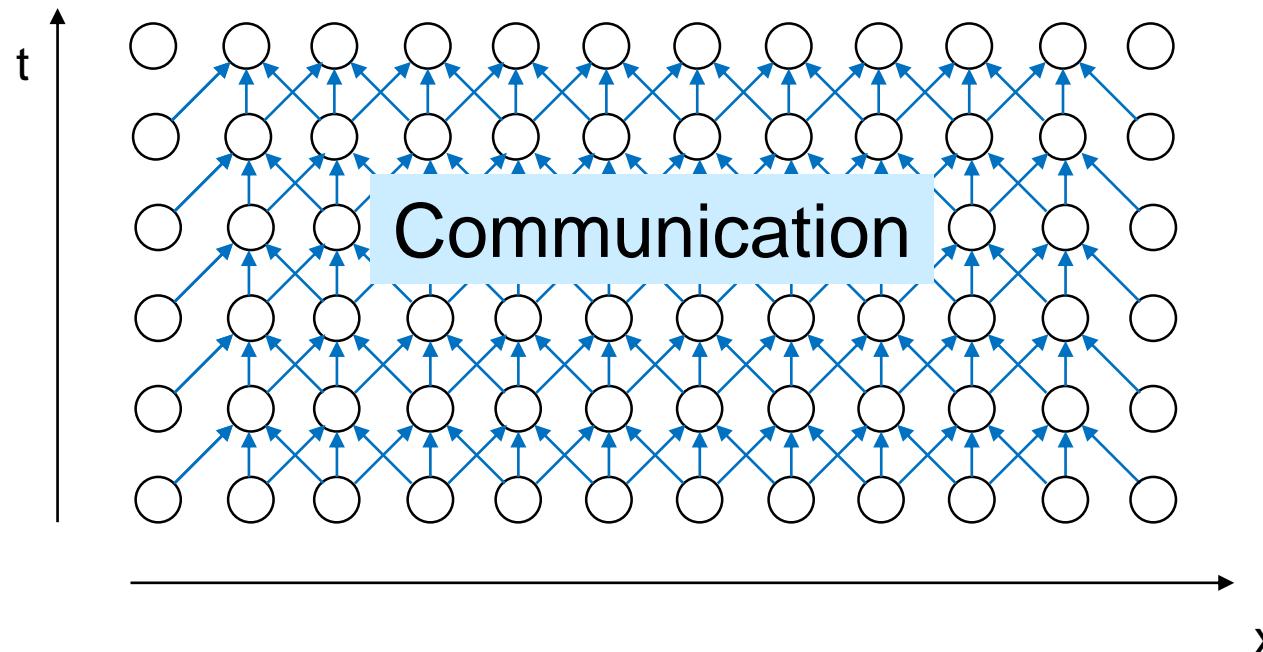
Dividing the computation and the data into primitive “tasks”



Foster's design methodology – Example

- Example:
 - $T(x,t) = \text{function}(T(x, t-1), T(x-1, t-1), T(x+1, t-1))$
(e.g., 1-dim heat equation with time integration)

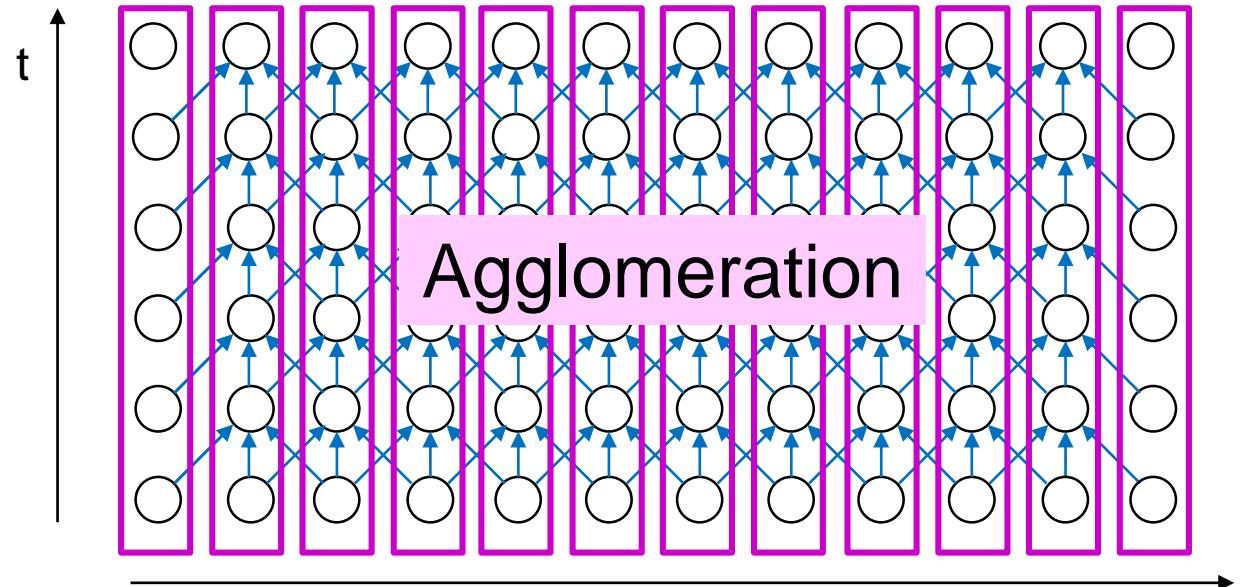
Determining how tasks will communicate with each other



Foster's design methodology – Example

- Example:
 - $T(x,t) = \text{function}(T(x, t-1), T(x-1, t-1), T(x+1, t-1))$
(e.g., 1-dim heat equation with time integration)

Grouping tasks
into larger
tasks to
improve
performance or
simplify
programming



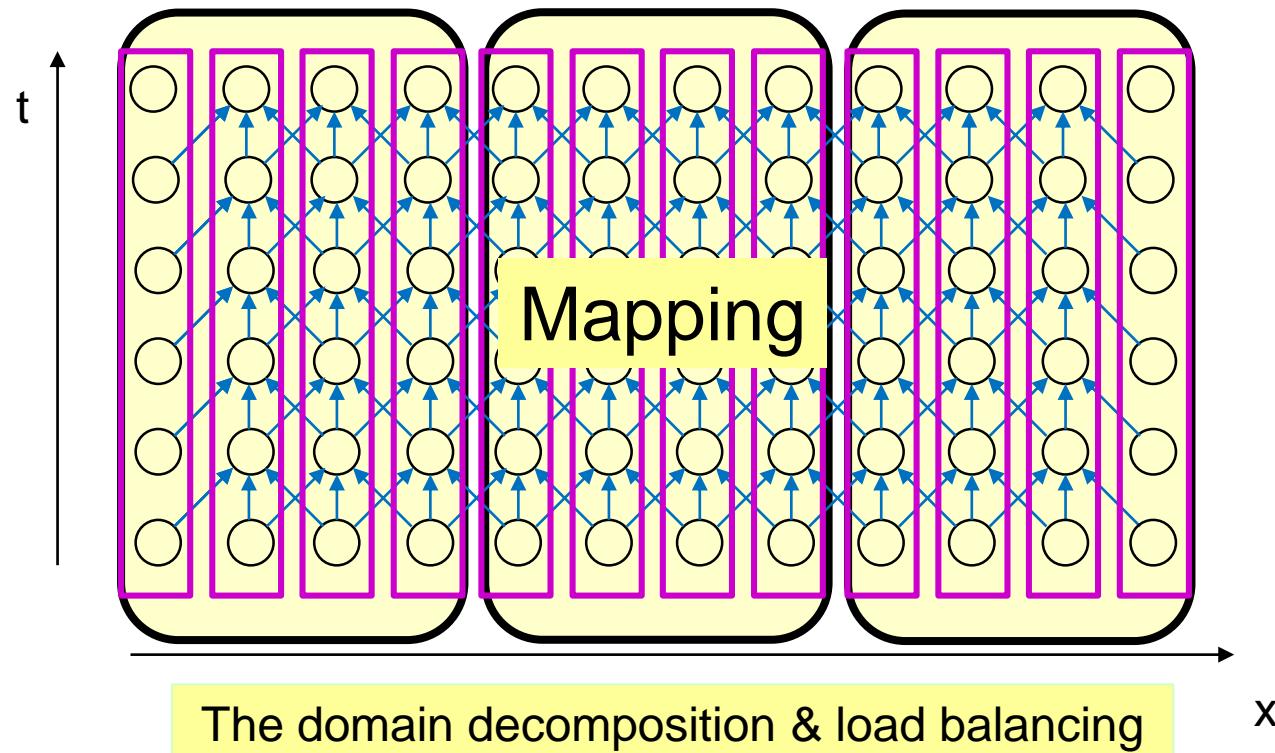
e.g. all timesteps of a given x are grouped together
but there is more freedom!

x

Foster's design methodology – Example

- Example:
 - $T(x,t) = \text{function}(T(x, t-1), T(x-1, t-1), T(x+1, t-1))$
(e.g., 1-dim heat equation with time integration)

Assigning tasks to physical processors (here MPI processes)



Performance considerations

MPI targets portable and efficient message-passing programming
but
efficiency of MPI application-programming is not portable!

- We saw during all chapters many options to implement a parallelization
 - Which point-to-point protocol (buffered, synchronous, standard, ready)
 - Blocking / nonblocking / persistent communication
 - Using collective operations
 - One-sided communication
 - MPI's new shared memory programming interface
 - Derived datatypes

Performance considerations

- Internally used protocols (short / eager /rendezvous)
- Specific options with each MPI library:
 - E.g., to define the limit for automatic buffering with MPI_Send
- Scalability
 - Some MPI routines' interfaces contain arrays with length $O(\#process)$
→ non-scalable interfaces
 - E.g., all collective ...v and ...w routines –
with exception: MPI_Neighbor_... routines
- Message sizes
 - Latencies $\sim 0.x - x.y \mu\text{sec}$
 - Bandwidth $\sim 0.1 - 50 \text{ Gbytes/sec}$
 - Latency * Bandwidth $\sim 10 - 100 \text{ kBytes}$
 - smaller messages are dominated by the latency
 - try to aggregate neighbor communication in long messages

If communication time =
latency + message size / bandwidth
and latency = $2\mu\text{s}$, bandwidth = 10 GB/s
then latency * bandwidth = 20 kB
→ For a message with such 20 kB,
communication time = $2 \mu\text{s} + 2\mu\text{s}$,
i.e., 50% is latency-based
and 50% is bandwidth-based

Rules for the communication modes

- Standard send (**MPI_Send**)
 - minimal transfer time
 - may block due to synchronous mode
 - → risks with synchronous send
- Synchronous send (**MPI_Ssend**)
 - risk of deadlock
 - risk of serialization
 - risk of waiting → idle time
 - high latency / best bandwidth
- Buffered send (**MPI_Bsend**)
 - low latency / bad bandwidth
- Ready send (**MPI_Rsend**)
 - use **never**, except you have a *200% guarantee* that Recv is already called in the current version and all future versions of your code
 - may be the fastest

Repeated from

course Chapter 4 *Nonblocking Communication*
and course Chapter 12 *Derived Datatypes*

Performance options

Which is the fastest neighbor communication with strided data?

- Using derived datatype handles, versus:
- **Copying** the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer, and copying the recv-buffer back into the strided application array
 - Often, a compiled application program can use the hardware more efficiently than the MPI library by running through a derived datatype
 - In principle, the MPI library may prohibit intermediate memory copies of the data
 - Especially with hybrid MPI+OpenMP, multiple threads may be used for such **copying**, whereas an MPI call may internally process derived types only with one thread
- And which of the communication routines should be used?
 - MPI_Irecv + MPI_Send
 - MPI_Irecv + MPI_Isend
 - MPI_Isend + MPI_Recv
 - MPI_Isend + MPI_Irecv
 - MPI_Sendrecv
 - MPI_Neighbor_alltoall → see course Chapter 9-(2) *Virtual Topologies*
 - Persistent pt-to-pt communication

MPI shared memory – a Summary

- Shared Memory was introduced in MPI-3.0.
- MPI shared memory can be used to **significantly reduce** the memory needs for **replicated data**.
- It is an opportunity to omit unnecessary communication inside of shared memory or ccNUMA nodes.
- Direct memory access may have best bandwidth compared to other MPI communication methods.
- Direct memory access can be combined with low latency synchronization.
- Which communication option is the fastest?

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming
but

efficiency of MPI application-programming is **not portable!**

Common pitfalls

- Deadlocks and **serialization**
→ see discussion and use of nonblocking communication
- Late sender
 - Receiver idles until the sender send the message
- Late receiver
 - The sender may be blocked due to synchronous protocol until the receive is started
- Tools:
see VI-HPS (Virtual Institute – High Productivity Supercomputing) <https://www.vi-hps.org/>
Tools Guide: <https://www.vi-hps.org/cms/upload/material/general/ToolsGuide.pdf>
and [training events](#)

Progress / weak local

An MPI procedure is **non-local** if returning may require, during its execution, some **specific** semantically-related MPI procedure to be called on another MPI process.

An MPI procedure is **local** if it is not *non-local*.

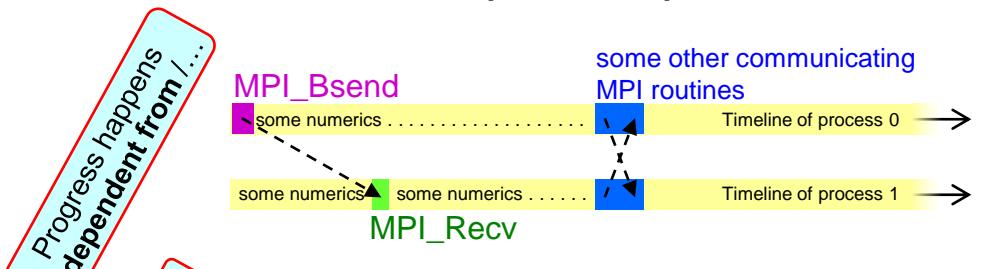
See course chapter 15



- Local MPI procedures may be implemented as “*weak local*”:
 - To complete its work locally,
it may require an *unspecific* MPI call on another process

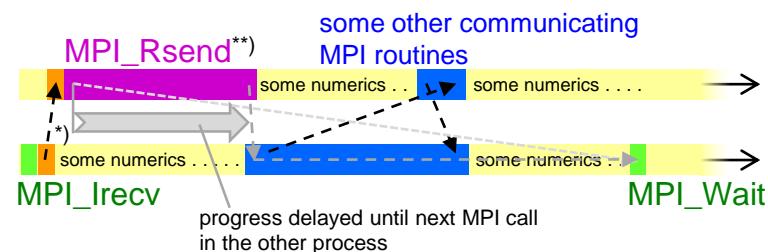
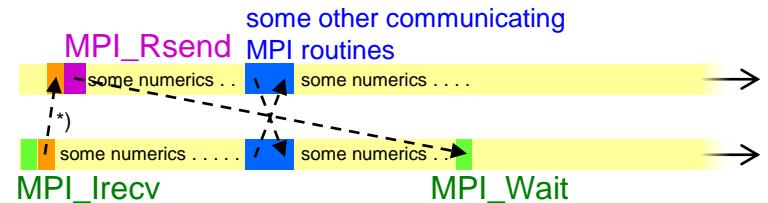
- Examples (always tested with **large** messages):

- Bsend is local.
 - Corresponding MPI_Recv may require progress in the sending process → may be blocked until the sending process calls another unspecific MPI procedure



Normally perfect ☺
Always correct ☺
But may also lead to negative surprises ☹

- Rsend is local, since the corresponding MPI_(I)Recv must already be called.
 - But the MPI_Rsend may require progress in the receiving process → may be blocked until the receiving process calls another unspecific MPI routine



MPI/tasks/C/Ch18/progress-test-bsend.c + progress-test-bsend-output.txt
progress-test-rsend.c + progress-test-rsend-output.txt

*) Additional communication that guarantees that MPI_Rsend is called after the corresponding MPI_Irecv is already started.

**) Same for MPI_Ssend and MPI_Send.

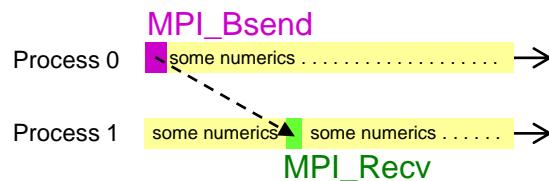


→ Next slide

Slide 589 / 593

What is progress

- To internally finish a started operation
 - the process that started the operation, and/or other related processes may need to make **progress** from the viewpoint of the underlying MPI system.
 - Example:
 - **Process 1:** Operation MPI receive, e.g., started with MPI_Recv or MPI_Irecv
 - **Process 0:** Is other related process
 - Called MPI_Bsend, already returned,
 - but data still buffered (from the viewpoint of the underlying MPI system)
 - **That process 1 can internally finish the receive operation, process 0 needs to make progress, i.e., to really send the buffered data**



- Which rules apply that process 0 provides progress? → See next slide

MPI Progress Rule

- MPI library must provide the following **minimal** progress

1. **Blocked MPI procedure calls** must provide progress on **all** enabled MPI operations.
 2. Test procedures will eventually return flag=true once the matching operation has been started:
 - **MPI_Test**, **MPI_Iprobe**, **MPI_Improbe**,
 - **MPI_Request_get_status**, **MPI_Win_test** (specification is missing in MPI-3.1/-4.0, will be clarified in MPI-4.1)
 - **MPI_Parrived** (new procedure in MPI-4.0)
 3. MPI finalization must guarantee that all required progress will be provided before the process exits.
 4. Further rules, e.g., on collectives, I/O, ...

- A **blocked MPI procedure call** can be:

- ### – Non-local MPI procedure

(e.g., `MPI_Send`, `MPI_Recv`, `MPI_Wait` for a receive/send request handle)

waits for a specific semantically-related MPI call on another MPI process

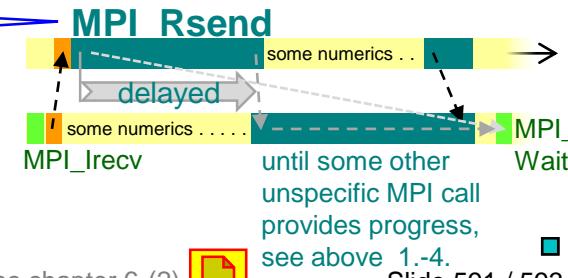
(e.g., MPI_(I)Recv, MPI_(I)Send, MPI_(I)Send / MPI_(I)Recv)

- **Local MPI procedure** (see also references 3.)

(e.g., `MPI_Rsend`)
waits for some unspecific MPI call on another MPI process

(e.g., any other MPI call that must do progress → see above 1. or 2. or 3 but it may be also a related routine, e.g., the `MPI_Wait` in the example).

locked call } MPI_Rsend



Of course,
more progress is always allowed!
E.g., through a progress thread
→ Course chapter 6-(2)  ☺

References in MPI-3.1:

1. Sect. 3.5, page 41, and 3.7.4, page 56.
Paragraphs “Progress”.
Sect. 12.4: MPI and Threads.
Sect. 11.7.3: Progress with one-sided communication, especially the rationale at the end.
 2. Sect. 3.7.4 on MPI_Test.
Sect. 3.8.1 & 3.8.2: MPI_(I)(M)PROBE.
 3. Sect. 3.8.4 Cancel, p. 73 lines 17-25.
MPI_Finalize Example 8.9, p. 358-359.
 4. Sect. 5.12: Nonblocking Collectives.
Sect. 13.6.3: MPI-I/O

For private notes

MPI – Summary

- Parallel MPI process model
- Message passing
 - blocking → several modes (**standard, buffered, synchronous, ready**)
 - nonblocking
 - to allow message passing from all processes in parallel
 - to avoid deadlocks and serializations
 - derived datatypes
 - to transfer any combination of data in one message
- Virtual topologies → a convenient processes naming scheme
- Collective communications → a major opportunity for optimization
- One-sided communication and shared memory → functionality & perform.
- Parallel file I/O → important option on large systems / part of NetCDF, HDF5, ...
- Overview on other MPI features (probe, groups&comms, spawn, MPI+threads)

MPI targets portable and efficient message-passing programming

but

efficiency of MPI application-programming is **not portable!**



For private notes

For private notes

For private notes

APPENDIX

Solutions (first the quiz-answers and then all exercises)



Chapter 1: (2) sub-domain calculation – versions A and B



Chapter 2: (1) Hello world / (2) I am *my_rank* of size



Chapter 3: (1) Ping with pt-to-pt, (2) ping-pong, (3) ping-pong benchmark



Chapter 4: (1) Wrong halo-copy in a ring, (2) nonblocking halo-copy solution



Chapter 6: Collective comm. (1) MPI_Gather, (2) MPI_Allreduce, (2) barrier



Chapter 8: (1) Split into two rings, (2) inter-communicator



Chap. 9-(1): Ring with virtual Cartesian topology: (1) Cart_create, (2) Cart_shift



Chap. 9-(2): (5) Neighbor_alltoall, (6) Neighbor_alltoallw



Chap. 9-(3): MPI_Cart_create_wighted / MPIX_Cart_weighted_create



Chapter 10: Ring with one-sided communication



Chap. 11-(1): Ring with MPI shared mem. (1) MPI_Comm_split_type, (2) Assignm.
(3) Mixed pt-to-pt and shared memory ring communication,



(4) Halo-copy (bidirectional) benchmark, (5) Bcast to shared mem



Chap. 11-(2): (6) Ring with shared memory & other synchroniz. + MPI_Win_sync



Chap. 12-(1): Derived types: (1) MPI_Type_contiguous (2) MPI_Type_create_struct



Chap. 12-(2): (5+6) Resizing of derived types



Chapter 13: Parallel file I/O exercises, Exa.2 Filetypes, Exa.3 Ordered

Quiz on Chapter 1 – Overview

Two developers report about their limited success when parallelizing an application:

- A. “My application is now running in parallel with 1000 MPI processes and my major limiting factor for scaling is
 - that I need about 50% of the whole compute time for MPI communication.”
- B. “My application is now running in parallel with 1000 MPI processes and my major limiting factor for scaling is
 - that I could not parallelize about 10% of the execution time of my sequential program.”

What are your answers for

- In your opinion, who of them was **more successful, A or B?**
 - **In my opinion: A**
- Can you calculate an estimate for the parallel efficiency of the parallel run reported by A and B?
 - Parallel Speedup: $S(p,N) = T(1,N)/T(p,N)$ → Parallel Efficiency: $E(p,N) = S(p,N)/p$
 - **A:** According to A, $p=1000$ and $T(1,N) = 50\% \cdot T(p,N) \cdot p$
→ $S(p,N) = 50\% \cdot T(p,N) \cdot p / T(p,N) = 500$ → $E(p,N) = 500/1000 = 50\%$
 - **B:** $T(p,N) = 10\% \cdot T(1,N) + 90\% \cdot T(1,N)/p$ with $p=1000$ → $T(p,N) = 10.09\% \cdot T(1,N)$
→ $S(p,N) = 9.91$ → $E(p,N) = 9.91/1000 = 0.991\% \approx 1\%$
 - **In other words: A is 50 times faster than B** (if A and B parallelized the same program)

Quiz on Chapter 3 – Point-to-point communication

- A. How many different MPI point-to-point send protocols (=blocking APIs) exist?
 - 4 (standard MPI_Send, synchronous MPI_Ssend, buffered MPI_Bsend, ready send MPI_Rsend)
- B. Which one requires that you first use MPI_Buffer_attach?
 - MPI_Bsend
- C. Which one is recommended for smallest latency and highest bandwidth both together?
 - MPI_Send (or MPI_Rsend for very special cases due to its additional requirement)

- D. If your buffer is an array buf with 5 double precision values that you want to send?
How do you describe your message in the call to MPI_Send in C and Fortran?
 - in C: buf, 5, MPI_DOUBLE
 - in Fortran: buf, 5, MPI_DOUBLE_PRECISION
- E. When calling MPI_Recv to receive this message which count values would be correct?
 - 5 and larger (but not smaller, i.e., truncation of messages is strictly forbidden)
- F. When I use one of the MPI send routines, how many messages do I send?
 - 2 (the message described by buf, count, datatype, and the tag, an additional piggyback message)

- G. Which is the predefined communicator that can be used to exchange a message from process rank 3 to process rank 5?
 - MPI_COMM_WORLD (in mpi4py: MPI.COMM_WORLD)
- H. If you send two messages msg1 and msg2 from rank 3 to rank 5, is it possible that the second one can overtake, i.e., be received before the first one?
 - Yes, e.g., if 1st message msg1 is send with MPI_Bsend and tag=1 and 2nd message with tag=2, and process 5 1st calls MPI_Recv with tag=2 and 2nd MPI_Recv with tag=1 then they overtake.

- I. Do you remember the major risks of synchronous send?
 - Risk of deadlocks and risk of serializations.
- J. Has standard send the same risks?
 - Yes, because standard send is allowed to be implemented with a synchronous send.
- K. What is the major use case for tags?
 - For security aspects that, e.g., a message with velocity data cannot be received into a temperature array

Quiz A+B on Chapter 4 – Nonblocking communication

- A. MPI nonblocking communication: Which are the three major use-cases and please sort from most important to less important:
1. Prevent serializations
 2. Prevent deadlocks
 3. For overlapping communication with computation or other communication

- B. [MPI/participant-test/ring-test.c](#) or [MPI/participant-test/ring-test_30.f90](#)

Is this **loop body** correctly programmed with MPI

- No

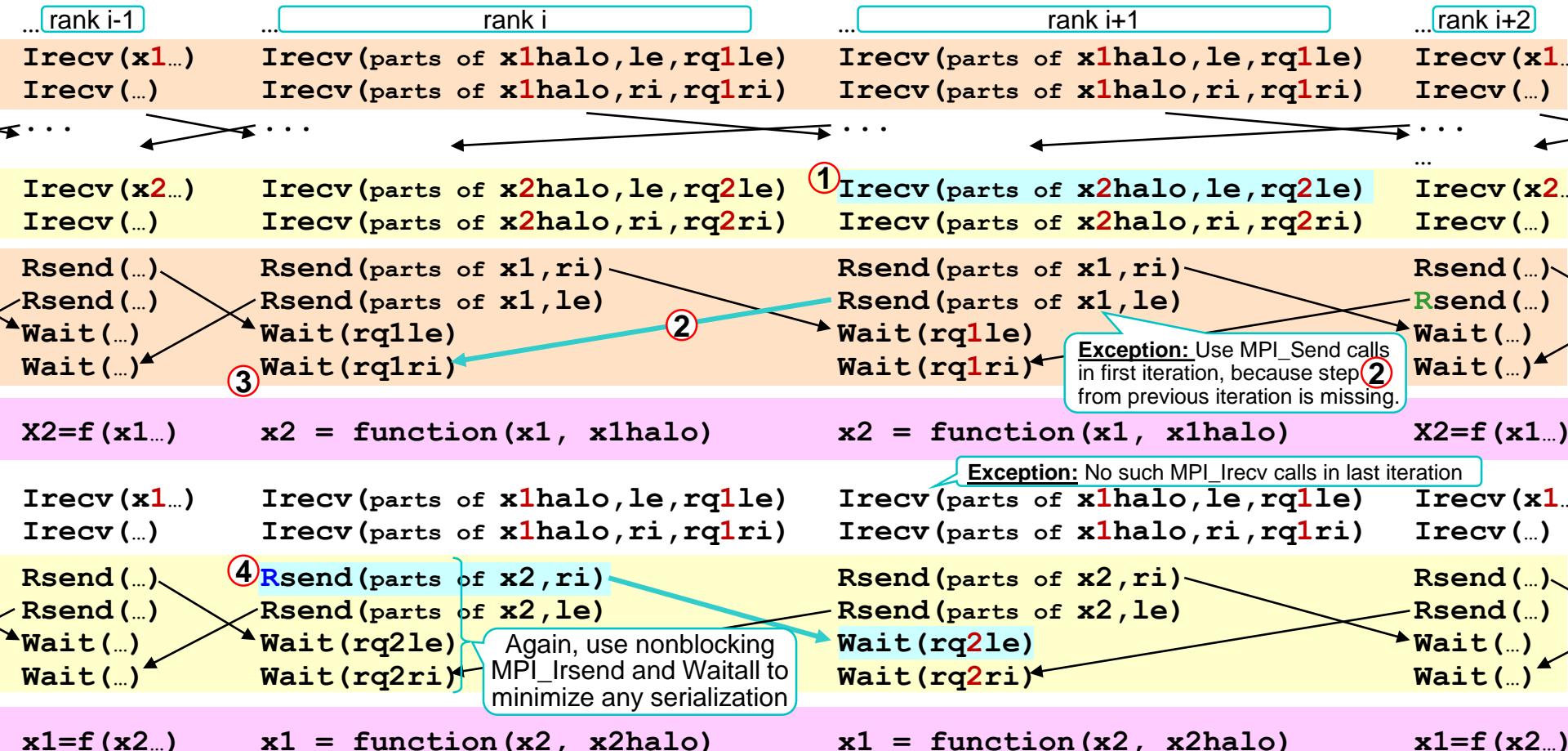
For the case that you answered with "No":

- Isend must be completed by **MPI_Wait**
- It must be added **after MPI_Recv**
- **Between MPI_Isend and MPI_Wait, the buffer must not be modified.**
- Therefore a **second buffer is needed** – `recv_buffer` – and
- after the `MPI_Wait` one must **add `buffer = recv_buffer`**
- As Fortran programmer, one must declare the
buffer in the nonblocking call as `ASYNCHRONOUS` and
one must add the `CALL MPI_F_sync_reg(buffer)` after `MPI_Wait`

```
MPI_Isend(&buffer,1,MPI_INT,right,  
          17, MPI_COMM_WORLD, &request);  
MPI_Recv (&buffer,1,MPI_INT, left,  
          17, MPI_COMM_WORLD, &status);  
sum += buffer;
```

Quiz C on Chapter 4 – Double buffering and MPI_Rsend

C. Who can see, what must we change that we can use **`MPI_Rsend`** instead of **`MPI_Send`**?



Proof. ① The receive is called. Then after ①, the message ② guarantees that all code in rank i after ③ is executed after ① and therefore, **`MPI_Rsend`** ④ is called after the corresponding receive ①.

For this, the content of the message is **not** relevant.



Quiz on Chapter 6-(1) – Collective communication

- MPI Collective communication: Which are the **major rules when using collective communication** routines and that **do not apply** to point to point communication? Please try to find at least two or three:
 1. All processes of a communicator must call this routine.
 2. Several collectives must be called in the same sequence on all processes of the given communicator.
 3. Each collective routine is allowed to synchronize with a barrier (although the communication pattern, e.g., broadcast, would not require this).
 4. The message size argument on the receive side must match the message size argument on the sender side,
i.e. the receive buffers must not be larger than the message
(i.e., larger receive buffers are allowed for point-to-point and one-sided communication, but not for collective ones).
 5. Nonblocking collectives do not match with blocking collectives.
 6. (There are no tags, but this is obvious when programming the argument list, and therefore not really a major rule that you should observe during programming.)

Quiz on Chapter 8-(1) – Groups & Communicators

- A. Which is the easiest way to build a set of disjoint subcommunicators?
 - **MPI_Comm_split**
- B. What are the major differences between
 - a group of processes referenced by a group handle, and
 - a communicator referenced by a communicator handle?
 - A group handle can **not** be used for any communication.
 - All MPI procedures to produce **group handles** are **non-collective local procedures** whereas all creating of **communicator handles** are **collective procedures**.
 - No difference is for the fact that both handles contain a list of processes sorted by ranks from 0 to #numprocs-1
- C. Can you produce with one call to MPI_Comm_create
 - Only one subcommunicator? **WRONG**
 - One or more disjoint subcommunicators? **CORRECT**
 - One or more overlapping subcommunicators? **WRONG**
- D. If you split a communicator in five subcommunicators,
must you then use an array for 5 handles as *newcomm* output argument
instead of only a single *newcomm* handle variable?
 - **No.** The output is always only one subcommunicator handle:
the one that contains the calling process
(or MPI_COMM_NULL if the calling is not part of any of the generated subcommunicators)

Quiz on Chapter 8-(2) – Groups & Communicators

A. Which data can be stored in an info handle?

Pairs of keyword + value

B. Which rules apply to such content?

1. Both, keywords and values are **strings**, e.g., "true"
2. The same keyword cannot be stored twice in the same info handle, i.e., with a second store (with MPI_Info_set), the original value is **overwritten**.
3. If an info handle is cached on a communicator, file, or window handle, some pairs may be removed, added or modified reflecting system specific properties

Quiz on Chapter 9-(1) – Virtual topologies

- A. What are the two major benefits from virtual topology process grids?
 1. **Minimizing communication time**
 2. **Better readable programs / simplifying code writing**
- B. What must the application programmer do to enable these opportunities?
 1. **Setting up a topology communicator** with a virtual process grid **that fits** to the application's communication pattern.
 2. **Allow reordering** of the process ranks (this is the base for faster communication).
 3. **From then**, communicating **only** through the new virtual grid communicator.
- C. Which MPI procedure can be used to create a set of non-overlapping subcommunicators? (do not forget to also mention the routines from course chapter 8!)
And are their specific prerequisites?
 1. **MPI_Comm_split** (see course chapter 8)
 - **Requires that all processes provide the same color value if they should be together in one subcommunicator**
 2. **MPI_Comm_create** (see course chapter 8)
 - **All processes of a new group must provide the same group, i.e. same processes and same ranks**
 3. **MPI_Cart_sub**
 - **The old communicator Cartesian virtual grid communicator**

Quiz on Chapter 9-(3) – Virtual topologies

A. Which types of MPI topologies for virtual process grids exist?

B. And for which **use cases**?

1. Cartesian topologies

- For Cartesian data meshes with identical compute time per mesh element
- For any Cartesian process grid with identical compute time per process and numerical epoch, and its communication mainly on the virtual Cartesian grid between the processes

2. Distributed graph topologies and graph topologies

- For applications with unstructured grids

C. Where are **limits** for using virtual topologies, i.e., which use cases do not really fit?

- Applications with mesh refinements, dynamic load balancing and diffusion of mesh elements to other processes
→ all cases with **changing virtual process grids over time**;
- Communication pattern not known in advance.

Quiz on Chapter 10 – One-sided Communication

A. Please describe what is a window?

A **window** is a memory portion accessible from the other processes, e.g. to store data there using MPI_Put, or to fetch data from there using MPI_Get.

B. If you execute an MPI_Put, where is the send and where the receive buffer?

The **send_buffer** is the **local buffer in the MPI_Put call**.

The **receive buffer** is the **window on the target process**, which is defined through the rank argument in the MPI_Put call.

C. And same question for MPI_Get?

The **send_buffer** is the **window on the target process**, which is defined through the rank argument in the MPI_Get call. The **receive buffer** is the **local buffer in the MPI_Get call**.

D. Is this example correct?

No, RMA calls (here Put) must be surrounded by synchronization calls.	rank 0 <code>MPI_Win_create(rcv_buf,...,&win)</code> <code>MPI_Win_fence(0,win)</code> <code>MPI_Put(buf,...to rank 1..., win)</code> <code>MPI_Win_fence(0,win)</code>	rank 1 <code>MPI_Win_create(rcv_buf,...,&win)</code> <code>MPI_Win_fence(0,win)</code> <code>MPI_Win_fence(0,win)</code>
---	---	---

E. If all processes are in the role of origin and target process and you want to use the pairs MPI_Win_post & MPI_Win_start before and MPI_Win_complete & MPI_Win_wait after your RMA calls MPI_Put (or ...Get). In which sequence, you must call these routines?

Win_post / Win_start / Put/Get... / Win_complete / Win_wait

F. And if you call the two calls of such a pair in the contrary order, what could then happen?

The MPI library may deadlock.

Quiz on Chapter 11-(1) – Shared Memory

- A. Before you call **MPI_Win_allocate_shared**, what should you do?

MPI_Comm_split_type(comm_old, MPI_COMM_TYPE_SHARED, ..., &comm_sm)

will guarantee that `comm_sm` contains only processes **of the same shared memory island**.

- B. If your communicator within your shared memory island consists of 12 MPI processes, and each process wants to get an own window with 10 doubles (each 8 bytes),

- a. which **window size** must you specify in **MPI_Win_allocate_shared**?

10 * 8 = 80 bytes

- b. And how long is the totally allocated shared memory?

80 * 12 = 960 bytes

- c. The returned `base_ptr`, will it be identical on all 12 processes?

No, within each process, the `base_ptr` points to its own portion of the totally allocated shared mem.

- d. If all 12 processes want to have a pointer that points to the beginning of the totally allocated shared memory, which MPI procedure should you use and with which major argument?

`MPI_Win_shared_query` with rank = 0

- e. If you do this, do these 12 pointers (to the window-start) point to identical addresses?

No, they point to the same physical address, but each MPI process may use different virtual addresses for this.

- C. Which is the major method to store data from one process into the shared memory window portion of another process?

Normal assignments (with C/C++ or Fortran) to the correct location, i.e., **no calls to MPI_Put/Get**.

Quiz on Chapter 11-(2) – Shared Memory Model & Sync.

- A. Which MPI memory model applies to MPI shared memory?
MPI_WIN_SEPARATE or **MPI_WIN_UNIFIED**?
- B. “Public and private copies are **eventually** synchronized without additional RMA calls.”
- C. Which process-to-process synchronization methods can be used that, e.g., a store to a shared memory variable gets visible to another process (within the processes of the shared memory window)?
 1. Any **MPI one-sided synchronization** (e.g., MPI_Win_fence, ..._post/start, ..., ..._lock/unlock)
 2. Any **(MPI) synchronization** together with a **pair of MPI_Win_sync**
 3. Any **(MPI) synchronization** together with a **pair of C11 atomic_thread_fence(order)**
- D. That such a store gets visible in another process after the synchronization is named here as **“write-read-rule”**.
Which other rules are implied by such synchronizations and what do they mean?
 1. **Read-write-rule**: a **load** (=**read**) in one process before the synchronization cannot be affected by a **store** (=**write**) in another process after the synchronization.
 2. **Write-write-rule**: a **store** (=**write**) in one process before the synchronization cannot overwrite a **store** (=**write**) in another process after the synchronization.
- E. How can you define a **race-condition** and which problems arise from **cache-line false-sharing**?
 1. Two processes access the **same shared variable** and at **least one process modifies** the variable **and the accesses are concurrent**.
 2. Significant performance problems if two or more processes **often access different portions of the same cache-line**.

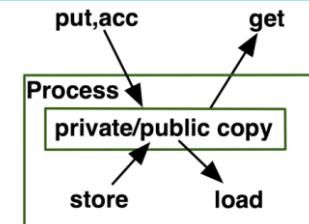


Figure: Courtesy of Torsten Hoefer

Quiz on Chapter 12-(1) – Derived datatypes

- A. Which types of data in your application's memory can you describe with a derived datatype handle?

Any combination of data (to be transferred) and holes (no transfer) in the memory,

e.g., a structure with elements of different types, or a noncontiguous subarray,

but **not** a linked list/tree, i.e., if the location of data portions depend on data (pointers/indexes) in this list

- B. Logically, to which internal structure points a derived datatype handle?

To a **list of pairs of basic datatype & its byte displacement.**

- C. Two pairs (count_1 , datatype_1) and (count_2 , datatype_2) match if ...?

($\text{count}_1 \times \text{list of basic datatypes of datatype}_1$) == ($\text{count}_2 \times \text{list of basic datatypes of datatype}_2$)

- D. If you have an array of a structure in your memory, how would you describe this?

Using **MPI_Type_create_struct** for the structure

and the **count** argument in the MPI communication procedure for the size of the array.

- E. Which additional MPI procedure call is required, before you can use a newly generated derived datatype handle in an MPI communication procedure?

MPI_Type_commit

- F. If you have a (noncontiguous) subarray of a multidimensional array, which procedure would you use to generate an appropriate derived datatype handle and which count value would you use in MPI_Send or MPI_Recv?

MPI_Type_create_subarray together with **count=1** in MPI_Send/Recv

- G. Which MPI procedures and functions should you use to calculate a byte displacement?

MPI_Get_address(var1,&addr1); MPI_Get_address(var2,&addr2); disp = MPI_Aint_diff(addr2, addr1);

Chapter 1 – Exercise 2: Solution – C – Version A

- Example output for 4 processes and 5 elements with 3 different sub_n:

I am process 0 out of 4, responsible for the 2 elements with indexes 0 .. 1
I am process 1 out of 4, responsible for the 2 elements with indexes 2 .. 3
I am process 2 out of 4, responsible for the 1 elements with indexes 4 .. 4
I am process 3 out of 4, responsible for the 0 elements with indexes -1 .. -2

- MPI/tasks/C/Ch1/first-dd-a.c

```
// Calculating the number of elements of my subdomain: sub_n
// Calculating the start index sub_start within 0..n-1
// or sub_start = -1 and sub_n = 0 if there is no element
// The following algorithm divided 5 into 2 + 2 + 1 + 0
sub_n = (n-1) / num_procs +1; // = rounding_up(n/num_procs)
sub_start = 0 + my_rank * sub_n;
if (sub_start < n)
{ // this process has a real element
    if (sub_start+sub_n-1 > n-1)
        { // but #elements must be smaller than sub_n
            sub_n = n - sub_start;
        } // else sub_n is already correct
} else
{ // this process has only zero elements
    sub_start = -1;
    sub_n = 0;
}
```

Or -1 if 0 elements

C

Major principle:

Filling the sub-domains with rounded-up sub_n until all elements are sorted into sub-domains.

Chapter 1 – Exercise 2: Solution – Fortran – Version A

- Example output for 4 processes and 5 elements with 3 different sub_n:

I am process 0 out of 4, responsible for the
I am process 1 out of 4, responsible for the
I am process 2 out of 4, responsible for the
I am process 3 out of 4, responsible for the

2 elements with indexes	0 ..	1
2 elements with indexes	2 ..	3
1 elements with indexes	4 ..	4
0 elements with indexes	-1 ..	-2

Or -1 if 0 elements

- MPI/tasks/F_30/Ch1/first-dd-a_30.f90

Fortran

```
! Calculating the number of elements of my subdomain: sub_n
! Calculating the start index sub_start within 0..n-1
! or sub_start = -1 and sub_n = 0 if there is no element
!The following algorithm divided 5 into 2 + 2 + 1 + 0
sub_n = (n-1) / num_procs +1 ! = rounding_up(n/num_procs)
sub_start = 0 + my_rank * sub_n;
IF (sub_start < n) THEN
    ! this process has a real element
    IF (sub_start+sub_n-1 > n-1) THEN
        ! but #elements must be smaller than sub_n
        sub_n = n - sub_start;
    ENDIF ! ELSE sub_n is already correct
ELSE
    ! this process has only zero elements
    sub_start = -1;
    sub_n = 0;
ENDIF
```

Major principle:

Filling the sub-domains with rounded-up sub_n until all elements are sorted into sub-domains.

Chapter 1 – Exercise 2: Solution – Python – Version A

- Example output for 4 processes and 5 elements with 3 different sub_n:

I am process 0 out of 4, responsible for the 2 elements with indexes 0 .. 1
I am process 1 out of 4, responsible for the 2 elements with indexes 2 .. 3
I am process 2 out of 4, responsible for the 1 elements with indexes 4 .. 4
I am process 3 out of 4, responsible for the 0 elements with indexes -1 .. -2

- MPI/tasks/C/Ch1/first-dd-a.py

Or -1 if 0 elements

Python

```
# Calculating the number of elements of my subdomain: sub_n
# Calculating the start index sub_start within 0..n-1
# or sub_start = -1 and sub_n = 0 if there is no element

# The following algorithm divided 5 into 2 + 2 + 1 + 0
sub_n = (n-1) // num_procs +1 # = ceil(n/num_procs), i.e., rounding up
sub_start = 0 + my_rank * sub_n
if (sub_start < n):
    # this process has a real element
    if (sub_start+sub_n-1 > n-1):
        # but #elements must be smaller than sub_n
        sub_n = n - sub_start;
    # else sub_n is already correct
else:
    # this process has only zero elements
    sub_start = -1
    sub_n = 0
```

Major principle:

Filling the sub-domains with rounded-up sub_n until all elements are sorted into sub-domains.

Chapter 1 – Exercise 2: Solution – C – Version B

- Example output for 4 processes and 5 elements with 2 different sub_n:

I am process 0 out of 4, responsible for the 2 elements with indexes 0 .. 1
I am process 1 out of 4, responsible for the 1 elements with indexes 2 .. 2
I am process 2 out of 4, responsible for the 1 elements with indexes 3 .. 3
I am process 3 out of 4, responsible for the 1 elements with indexes 4 .. 4

- MPI/tasks/C/Ch1/first-dd-b.c

Or -1 if 0 elements

C

```
// Calculating the number of elements of my subdomain: sub_n
// Calculating the start index sub_start within 0..n-1
// or sub_start = -1 and sub_n = 0 if there is no element

// The following algorithm divided 5 into 2 + 1 + 1 + 1
sub_n = n / num_procs; // = rounding_off(n/num_procs)
int num_larger_procs = n - num_procs*sub_n;
if (my_rank < num_larger_procs)
{ sub_n = sub_n + 1;
  sub_start = 0 + my_rank * sub_n;
} else if (sub_n > 0)
{ sub_start = 0 + num_larger_procs + my_rank * sub_n;
} else
{ // this process has only zero elements
  sub_start = -1;
  sub_n = 0;
}
```

= #procs with
sub_n+1 elements

Major principle:

Use rounded-off divide for sub_n
and calculate, how many elements
do not yet have a subdomain.

→ Take this number as the number
of processes that must have exactly
1 element more than the rounded-off
value.

Chapter 1 – Exercise 2: Solution – Fortran – Version B

- Example output for 4 processes and 5 elements with 2 different sub_n:

I am process 0 out of 4, responsible for the
I am process 1 out of 4, responsible for the
I am process 2 out of 4, responsible for the
I am process 3 out of 4, responsible for the

2 elements with indexes	0 ..	1
1 elements with indexes	2 ..	2
1 elements with indexes	3 ..	3
1 elements with indexes	4 ..	4

Or -1 if 0 elements

- MPI/tasks/F_30/Ch1/first-dd-b_30.f90

Fortran

```
! Calculating the number of elements of my subdomain: sub_n
! Calculating the start index sub_start within 0..n-1
! or sub_start = -1 and sub_n = 0 if there is no element

! The following algorithm divided 5 into 2 + 1 + 1 + 1
sub_n = n / num_procs ! = rounding_off(n/num_procs)
num_larger_procs = n - num_procs*sub_n
IF (my_rank < num_larger_procs) THEN
    sub_n = sub_n + 1
    sub_start = 0 + my_rank * sub_n
ELSE IF (sub_n > 0) THEN
    sub_start = 0 + num_larger_procs + my_rank * sub_n
ELSE
    ! this process has only zero elements
    sub_start = -1;
    sub_n = 0;
ENDIF
```

= #procs with
sub_n+1 elements

Major principle:

Use rounded-off divide for sub_n and calculate, how many elements do not yet have a subdomain.
→ Take this number as the number of processes that must have exactly 1 element more than the rounded-off value.

Chapter 1 – Exercise 2: Solution – Python – Version B

- Example output for 4 processes and 5 elements with 2 different sub_n:

I am process 0 out of 4, responsible for the 2 elements with indexes 0 .. 1
I am process 1 out of 4, responsible for the 1 elements with indexes 2 .. 2
I am process 2 out of 4, responsible for the 1 elements with indexes 3 .. 3
I am process 3 out of 4, responsible for the 1 elements with indexes 4 .. 4

- MPI/tasks/C/Ch1/first-dd-b.c

Or -1 if 0 elements

Python

```
# Calculating the number of elements of my subdomain: sub_n
# Calculating the start index sub_start within 0..n-1
# or sub_start = -1 and sub_n = 0 if there is no element

# The following algorithm divided 5 into 2 + 1 + 1 + 1
sub_n = n // num_procs # = floor(n/num_procs), i.e., rounding off
num_larger_procs = n - num_procs*sub_n
if (my_rank < num_larger_procs):
    sub_n = sub_n + 1
    sub_start = 0 + my_rank * sub_n
elif (sub_n > 0):
    sub_start = 0 + num_larger_procs + my_rank * sub_n
else:
    # this process has only zero elements
    sub_start = -1
    sub_n = 0
```

= #procs with
sub_n+1 elements

Major principle:

Use rounded-off divide for sub_n and calculate, how many elements do not yet have a subdomain.
→ Take this number as the number of processes that must have exactly 1 element more than the rounded-off value.

Do you prefer A or B, and why?

B: because for $n \geq \text{num_procs}$, all processes have at least one mesh element!



Chapter 2 – Exercise 1: Hello world

C

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv); // or MPI_Init(NULL, NULL);
    printf ("Hello world!\n");
    MPI_Finalize();
    return 0;
}
```

MPI/tasks/C/Ch2/solutions/hello.c

Fortran

```
PROGRAM hello
USE mpi_f08
IMPLICIT NONE
CALL MPI_Init()
WRITE(*,*) 'Hello world!'
CALL MPI_Finalize()
END PROGRAM
```

MPI/tasks/F_30/Ch2/solutions/hello_30.f90

Python

```
from mpi4py import MPI
print("Hello World!")
```

MPI/tasks/PY/Ch2/solutions/hello.py



Chapter 2 – Exercise 2: I am my_rank of size

C

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{   int my_rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (my_rank == 0) { printf ("Hello world!\n");
    printf("I am process %i out of %i.\n", my_rank, size);
    MPI_Finalize();
    return 0;
}
```

MPI/tasks/C/Ch2/solutions/myrank.c

Fortran

```
PROGRAM hello
USE mpi_f08
IMPLICIT NONE
INTEGER my_rank, size
CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
CALL MPI_Comm_size(MPI_COMM_WORLD, size)
IF (my_rank .EQ. 0) THEN ; WRITE(*,*) 'Hello world!' ; END IF
WRITE(*,*) 'I am process', my_rank, ' out of', size
CALL MPI_Finalize()
END PROGRAM
```

MPI/tasks/F_30/Ch2/solutions/myrank_30.f90

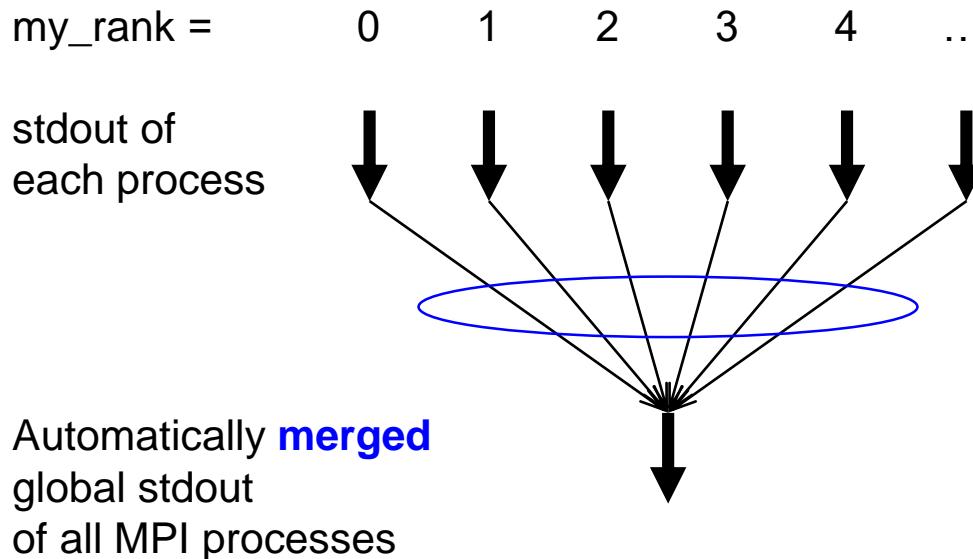
Python

```
from mpi4py import MPI
comm_world = MPI.COMM_WORLD
# MPI-related data
my_rank = comm_world.Get_rank()
size = comm_world.Get_size()
if (my_rank == 0):
    print("Hello world!")
print(f"I am process {my_rank} out of {size}")
```

MPI/tasks/PY/Ch2/solutions/myrank.py



Chapter 2 – Exercise 3: Advanced exercise



- The **merge** of the stdout pipes of all MPI processes to the global stdout is undefined,
- i.e., no sequence rules,
- i.e., a sequence can be defined only if all output on stdout is done only by one MPI process (e.g., with `my_rank == 0`)

Chapter 3 – Exercise 1: Ping with point-to-point communication

C

```
float buffer[1];  
...  
if (my_rank == 0)  
{  printf("I am %i before send ping \n", my_rank);  
    MPI_Send( buffer, 1, MPI_FLOAT, 1, 17, MPI_COMM_WORLD);  
}else if (my_rank == 1)  
{  MPI_Recv( buffer, 1, MPI_FLOAT, 0, 17, MPI_COMM_WORLD, &status);  
    printf("I am %i after  recv ping \n", my_rank);  
}
```

MPI/tasks/C/Ch3/solutions/ping.c

Fortran

```
REAL :: buffer(1)  
...  
IF (my_rank .EQ. 0) THEN  
    WRITE(*,*) 'I am ', my_rank, ' before send ping'  
    CALL MPI_Send(buffer, 1, MPI_REAL, 1, 17, MPI_COMM_WORLD)  
ELSE IF (my_rank .EQ. 1) THEN  
    CALL MPI_Recv(buffer, 1, MPI_REAL, 0, 17, MPI_COMM_WORLD, status)  
    WRITE(*,*) 'I am ', my_rank, ' after  recv ping'  
END IF
```

MPI/tasks/F_30/Ch3/solutions/ping_30.f90

Python

```
buffer = [ None ]  
if (my_rank == 0):  
    print(f"I am {my_rank} before send ping")  
    comm_world.send(buffer, dest=1, tag=17);  
elif (my_rank == 1):  
    buffer = comm_world.recv(source=0, tag=17);  
    print(f"I am {my_rank} after  recv ping ")
```

MPI/tasks/PY/Ch3/solutions/ping.py

Expected output

I am 0 before send ping
I am 1 after recv ping

possible,
but
unexpected

I am 1 after recv ping
I am 0 before send ping



Chapter 3 – Exercise 2: Ping-pong with point-to-point communication

C

```
if (my_rank == 0)
{
    printf("I am %i before send ping \n", my_rank);
    MPI_Send(buffer, 1, MPI_FLOAT, 1, 17, MPI_COMM_WORLD);
    MPI_Recv(buffer, 1, MPI_FLOAT, 1, 23, MPI_COMM_WORLD, &status);
    printf("I am %i after recv pong \n", my_rank);
}
else if (my_rank == 1)
{
    MPI_Recv(buffer, 1, MPI_FLOAT, 0, 17, MPI_COMM_WORLD, &status);
    printf("I am %i after recv ping \n", my_rank);
    printf("I am %i before send pong \n", my_rank);
    MPI_Send(buffer, 1, MPI_FLOAT, 0, 23, MPI_COMM_WORLD);
}
```

MPI/tasks/C/Ch3/solutions/pingpong.c

Fortran

```
IF (my_rank .EQ. 0) THEN
    WRITE(*,*) 'I am ', my_rank, ' before send ping'
    CALL MPI_Send(buffer, 1, MPI_REAL, 1, 17, MPI_COMM_WORLD)
    CALL MPI_Recv(buffer, 1, MPI_REAL, 1, 23, MPI_COMM_WORLD, status)
    WRITE(*,*) 'I am ', my_rank, ' after recv pong'
ELSE IF (my_rank .EQ. 1) THEN
    CALL MPI_Recv(buffer, 1, MPI_REAL, 0, 17, MPI_COMM_WORLD, status)
    WRITE(*,*) 'I am ', my_rank, ' after recv ping'
    WRITE(*,*) 'I am ', my_rank, ' before send pong'
    CALL MPI_Send(buffer, 1, MPI_REAL, 0, 23, MPI_COMM_WORLD)
END IF
```

MPI/tasks/F_30/Ch3/solutions/pingpong_30.f90

Python

Same solution in Python

MPI/tasks/PY/Ch3/solutions/pingpong.py

sequence
of the output
reads well

```
I am 0 before send ping
I am 1 after recv ping
I am 1 before send pong
I am 0 after recv pong
```

sequence
of the output
reads badly

```
I am 0 before send ping
I am 0 after recv pong
I am 1 after recv ping
I am 1 before send pong
```

Chapter 3 – Exercise 3: Ping-pong benchmark with point-to-point communication

C

```
double start, finish, ...
start = MPI_Wtime();
for (i = 1; i <= 50; i++)
{ if (my_rank == 0)
    { MPI_Send(buffer, 1, MPI_FLOAT, 1, 17, MPI_COMM_WORLD);
      MPI_Recv(buffer, 1, MPI_FLOAT, 1, 23, MPI_COMM_WORLD, &status);
    }else if (my_rank == 1)
    { MPI_Recv(buffer, 1, MPI_FLOAT, 0, 17, MPI_COMM_WORLD, &status);
      MPI_Send(buffer, 1, MPI_FLOAT, 0, 23, MPI_COMM_WORLD);
    }
}
finish = MPI_Wtime();
msg_transfer_time = ((finish - start) / (2 * 50)) * 1e6; // in microsec
if (my_rank == 0)
    printf("Time for one message: %f micro seconds.\n", msg_transfer_time);
```

MPI/tasks/C/Ch3/solutions/pingpong-bench.c

Fortran

```
double precision :: start ...
start = MPI_Wtime()
DO i = 1, 50
    IF (my_rank .EQ. 0) THEN
        ...
    ELSE IF (my_rank .EQ. 1) THEN
        ...
    END IF
END DO
finish = MPI_Wtime()
msg_transfer_time = ((finish - start) / (2 * 50)) * 1e6 ! in microsec
IF (my_rank .EQ. 0) THEN
    WRITE(*,*) 'One message:', msg_transfer_time, ' micro seconds'
ENDIF
```

MPI/tasks/F_30/Ch3/solutions/pingpong-bench_30.f90

Python

```
start = MPI.Wtime()
for i in range(1, number_of_messages+1):
    if ...
    elif ...
finish = MPI.Wtime()
if (my_rank == 0):
    msg_transfer_time = ((finish - start) / (2 * number_of_messages)) * 1e6 # in microsec
    print(f"Time for one message: {msg_transfer_time:f} micro seconds.")
```

MPI/tasks/PY/Ch3/solutions/pingpong-bench.py



Chapter 4 – Exercise 1: Nonblocking halo-copy in a ring

C

MPI/tasks/C/Ch4/ring-WRONC1.c

```
int buffer, sum;
int right, left;
int sum, i, my_rank, size;
MPI_Status status;
MPI_Request request;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

right = (my_rank+1) % size;
left = (my_rank-1+size) % size;
sum = 0;

1 buffer = my_rank;
for( i = 0; i < size; i++) Wrong, because it may cause a deadlock
{
2   MPI_Send(&buffer, 1, MPI_INT, right, 17, MPI_COMM_WORLD);
3   MPI_Recv(&buffer, 1, MPI_INT, left, 17, MPI_COMM_WORLD, &status);
4   // buffer = buffer; // Using only one buffer, then no need for step 4
5   sum += buffer;
}
printf ("PE%i:\tSum = %i\n", my_rank, sum);
MPI_Finalize();
```

Chapter 4 – Exercise 1: Nonblocking halo-copy in a ring

Fortran

MPI/tasks/F_30/Ch4/ring-WRONG1_30.f90

```
INTEGER :: buffer
INTEGER :: sum, i, my_rank, size
TYPE(MPI_Status) :: status
TYPE(MPI_Request) :: request
INTEGER(KIND=MPI_ADDRESS_KIND) :: iadummy

CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
CALL MPI_Comm_size(MPI_COMM_WORLD, size)
right = mod(my_rank+1, size)
left = mod(my_rank-1+size, size)
sum = 0
1 buffer = my_rank
DO i = 1, size
    CALL MPI_Send(buffer, 1, MPI_INTEGER, right, 17, MPI_COMM_WORLD)
2 CALL MPI_Recv(buffer, 1, MPI_INTEGER, left, 17, MPI_COMM_WORLD, status)
3 CALL MPI_Recv(buffer, 1, MPI_INTEGER, left, 17, MPI_COMM_WORLD, status)
4 ! buffer = buffer ! Using only one buffer, then no need for step 4
5 sum = sum + buffer
END DO
WRITE(*,*) 'PE', my_rank, ': Sum =', sum
CALL MPI_Finalize()
```

Wrong, because it may cause a deadlock

1

2

3

4

5

Chapter 4 – Exercise 1: Nonblocking halo-copy in a ring

Python

```
#!/usr/bin/env python3
```

MPI/tasks/PY/Ch4/ring-WRONG1.py

```
from mpi4py import MPI
import numpy as np

status = MPI.Status()

comm_world = MPI.COMM_WORLD
my_rank = comm_world.Get_rank()
size = comm_world.Get_size()
right = (my_rank+1)      % size;
left   = (my_rank-1+size) % size;
sum = 0
1    snd_buf = np.array(my_rank)
```

```
for i in range(size):
```

Wrong, because it may cause a deadlock

```
    comm_world.Send(buffer, 1, MPI.INT), dest=right, tag=17)
```

```
2      comm_world.Recv(buffer, 1, MPI.INT), source=left, tag=17, status=status)
```

```
3      # buffer = buffer # With only one buffer, one does not need this step 4
```

```
4      sum += buffer
```

```
5      print(f"PE{my_rank}: \tSum = {sum}")
```

Chapter 4 – Exercise 1: Nonblocking halo-copy in a ring

C

MPI/tasks/C/Ch4/ring-WRON2.c

```
int snd_buf, rcv_buf, sum;
int right, left;
int sum, i, my_rank, size;
MPI_Status status;
MPI_Request request;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

right = (my_rank+1) % size;
left = (my_rank-1+size) % size;
sum = 0;

1 snd_buf = my_rank;
for( i = 0; i < size; i++) Wrong, because it may cause a deadlock
{
2   MPI_Send(&snd_buf, 1, MPI_INT, right, 17, MPI_COMM_WORLD);
3   MPI_Recv(&rcv_buf, 1, MPI_INT, left, 17, MPI_COMM_WORLD, &status);
4   snd_buf = rcv_buf;
5   sum += rcv_buf;
}
printf ("PE%i:\tSum = %i\n", my_rank, sum);
MPI_Finalize();
```

Chapter 4 – Exercise 1: Nonblocking halo-copy in a ring

Fortran

MPI/tasks/F_30/Ch4/ring-WRONG2_30.f90

```
INTEGER :: snd_buf
INTEGER :: rcv_buf, sum, i, my_rank, size
TYPE(MPI_Status) :: status
TYPE(MPI_Request) :: request
INTEGER(KIND=MPI_ADDRESS_KIND) :: iadummy

CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
CALL MPI_Comm_size(MPI_COMM_WORLD, size)
right = mod(my_rank+1, size)
left = mod(my_rank-1+size, size)
sum = 0
1  snd_buf = my_rank
DO i = 1, size
    CALL MPI_Send(snd_buf, 1, MPI_INTEGER, right, 17, MPI_COMM_WORLD)
2  CALL MPI_Recv(rcv_buf, 1, MPI_INTEGER, left, 17, MPI_COMM_WORLD, status)
3  CALL MPI_Recv(rcv_buf, 1, MPI_INTEGER, left, 17, MPI_COMM_WORLD, status)
4  snd_buf = rcv_buf
5  sum = sum + rcv_buf
END DO
WRITE(*,*) 'PE', my_rank, ': Sum =', sum
CALL MPI_Finalize()
```

Wrong, because it may cause a deadlock

1

2

3

4

5

Chapter 4 – Exercise 1: Nonblocking halo-copy in a ring

Python

```
#!/usr/bin/env python3
from mpi4py import MPI
import numpy as np

recv_buf = np.empty((), dtype=np.intc)
status = MPI.Status()

comm_world = MPI.COMM_WORLD
my_rank = comm_world.Get_rank()
size = comm_world.Get_size()
right = (my_rank+1) % size;
left = (my_rank-1+size) % size;
sum = 0
1 snd_buf = np.array(my_rank, dtype=np.intc)

for i in range(size):
    comm_world.Send((snd_buf, 1, MPI.INT), dest=right, tag=17)
    comm_world.Recv((recv_buf, 1, MPI.INT), source=left, tag=17, status=status)
    np.copyto(snd_buf, recv_buf)
    sum += recv_buf

print(f"PE{my_rank}: \tSum = {sum}")
```

MPI/tasks/PY/Ch4/ring-WRON2.py

In course chapters 1-3,
we used the object-serializing interface with
`comm_world.send(obj, ...)`.

From now on, we'll use the direct communication with
`comm_world.Send(numpy arrays, ...)`.

Integer array with one element
initialized with the value of my_rank

Wrong, because it may cause a deadlock

2 `comm_world.Send((snd_buf, 1, MPI.INT), dest=right, tag=17)`

3 `comm_world.Recv((recv_buf, 1, MPI.INT), source=left, tag=17, status=status)`

4 `np.copyto(snd_buf, recv_buf)`

We make a copy here.
Remember that `snd_buf = recv_buf` binds them to the same object
(i.e. `snd_buf` and `recv_buf` are then two different names pointing to the
same object), which is unintended and can lead to incorrect code.

5 `sum += recv_buf`

Chapter 4 – Exercise 1: Nonblocking halo-copy in a ring

C

```
    snd_buf = my_rank;
    for( i = 0; i < size; i++)
    { if (my_rank == 0)
        { MPI_Recv(&rcv_buf, 1, MPI_INT, left, 17, MPI_COMM_WORLD, &status);
          MPI_Ssend(&snd_buf, 1, MPI_INT, right, 17, MPI_COMM_WORLD);
        } else
        { MPI_Ssend(&snd_buf, 1, MPI_INT, right, 17, MPI_COMM_WORLD);
          MPI_Recv(&rcv_buf, 1, MPI_INT, left, 17, MPI_COMM_WORLD, &status);
        }
        snd_buf = rcv_buf;
        sum += rcv_buf;
    }
    printf ("PE%i:\tSum = %i\n", my_rank, sum);
```

WRONG program, because the deadlock is resolved by a serialization, which is for performance reasons wrong!
And this program will still deadlock when running with only one process.

Fortran

```
    snd_buf = my_rank      MPI/tasks/F_30/Ch4/solutions/ring-WRONG2-serialized_30.f90
    DO i = 1, size
      IF (my_rank == 0) THEN
        CALL MPI_Recv(rcv_buf, 1, MPI_INTEGER, left, 17, MPI_COMM_WORLD, status)
        CALL MPI_Ssend(snd_buf,1, MPI_INTEGER, right, 17, MPI_COMM_WORLD)
      ELSE
        CALL MPI_Ssend(snd_buf,1, MPI_INTEGER, right, 17, MPI_COMM_WORLD)
        CALL MPI_Recv(rcv_buf, 1, MPI_INTEGER, left, 17, MPI_COMM_WORLD, status)
      ENDIF
      snd_buf = rcv_buf
      sum = sum + rcv_buf
    END DO
    WRITE(*,*) 'PE', my_rank, ': Sum =', sum
```

WRONG program, because the deadlock is resolved by a serialization, which is for performance reasons wrong!
And this program will still deadlock when running with only one process.

Python

```
# same solution with Python
```

```
MPI/tasks/PY/Ch4/solutions/ring-WRONG2-serialized.py
```



Chapter 4 – Exercise 2: Nonblocking halo-copy in a ring

C

```
int snd_buf, rcv_buf, sum;
int right, left;
int sum, my_rank, size, i;
MPI_Status status;
MPI_Request request;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

right = (my_rank+1) % size;
left = (my_rank-1+size) % size;
sum = 0;
snd_buf = my_rank;
for( i = 0; i < size; i++)
{
    MPI_Issend(&snd_buf, 1, MPI_INT, right, 17, MPI_COMM_WORLD, &request);
    MPI_Recv ( &rcv_buf, 1, MPI_INT, left, 17, MPI_COMM_WORLD, &status);
    MPI_Wait(&request, &status);
    snd_buf = rcv_buf;
    sum += rcv_buf;
}
printf ("PE%i:\tSum = %i\n", my_rank, sum);
MPI_Finalize();
```

MPI/tasks/C/Ch4/solutions/ring.c

In C, normally such helper variables should be declared only within the scope where needed, here the loop. For our exercises, they are all declared at the beginning, mainly to keep C and Fortran solutions identical.

Synchronous send (**Issend**) instead of standard send (**Isend**) is used only to demonstrate the use of the nonblocking routine resolves the deadlock (or serialization) problem.
A real application would use standard **Isend()**.

1

2

3

4

5

Chapter 4 – Exercise 2: Nonblocking halo-copy in a ring

MPI/tasks/F_30/Ch4/solutions/ring_30.f90

MPI/tasks/F_30/Ch4/solutions/ring-WRON-S_30.f90

Fortran

```
INTEGER, 2 ASYNCHRONOUS :: snd_buf
INTEGER :: rcv_buf, sum, i, my_rank, size
TYPE(MPI_Status) :: status
TYPE(MPI_Request) :: request
INTEGER(KIND=MPI_ADDRESS_KIND) :: iadummy

CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
CALL MPI_Comm_size(MPI_COMM_WORLD, size)
right = mod(my_rank+1, size)
left = mod(my_rank-1+size, size)
sum = 0
snd_buf = my_rank
DO i = 1, size
    CALL MPI_Issend(snd_buf, 1, MPI_INTEGER, right, 17, MPI_COMM_WORLD, request)
    CALL MPI_Recv ( rcv_buf, 1, MPI_INTEGER, left, 17, MPI_COMM_WORLD, status)
    CALL MPI_Wait(request, status)
    IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
    snd_buf = rcv_buf
    sum = sum + rcv_buf
END DO
WRITE(*,*) 'PE', my_rank, ': Sum =', sum
CALL MPI_Finalize()
```

Synchronous send (**Issend**) instead of standard send (**Isend**) is used only to demonstrate the use of the nonblocking routine resolves the deadlock (or serialization) problem. A real appl. would use Isend.

1

2

3

4

5

In **ring-WRON-S_30.f90**, this line is commented out:
For example using compiler with -O4, you may get completely wrong results.
In Chap. 10, you can see such wrong results, e.g., with the gfortran compiler.

Chapter 4 – Exercise 2: Nonblocking halo-copy in a ring

MPI/tasks/PY/Ch4/solutions/ring.py

Python

```
#!/usr/bin/env python3
from mpi4py import MPI
import numpy as np

rcv_buf = np.empty((), dtype=np.intc)
status = MPI.Status()

comm_world = MPI.COMM_WORLD
my_rank = comm_world.Get_rank()
size = comm_world.Get_size()
right = (my_rank+1) % size
left = (my_rank-1+size) % size
sum = 0

1  snd_buf = np.array(my_rank, dtype=np.intc)
for i in range(size):
    request = comm_world.Issend((snd_buf, 1, MPI.INT), dest=right, tag=17)
    comm_world.Recv((rcv_buf, 1, MPI.INT), source=left, tag=17, status=status)
    request.Wait(status)

2  np.copyto(snd_buf, rcv_buf) # We make a copy here.
3  sum += rcv_buf

4  print(f"PE{my_rank}: \tSum = {sum}")

5 
```

Synchronous **send (Issend)** instead of standard send (**Isend**) is used only to demonstrate the use of the nonblocking routine resolves the deadlock (or serialization) problem. A real appl. would use Isend.

2

3

4

5

Chapter 6-(1) – Exercise 1: Collective communication with MPI_Gather – skeleton

C

```
result = 100.0 + 1.0 * my_rank;
printf("I am process %i out of %i, result=%f \n",
       my_rank, num_procs, result);
if (my_rank == 0)
{ result_array = malloc(sizeof(double) * num_procs);
}

// ----- the following gathering of the results should -----
// ----- be substituted by one call to MPI_Gather -----
if (my_rank != 0) // in all processes, except "root" process 0
{ // sending some results from all processes (except 0) to process 0:
    MPI_Send(&result, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
} else // only in "root" process 0
{ result_array[0] = result; // process 0's own result
    // receiving all these messages
    for (rank=1; rank<num_procs; rank++)
    { // result of processes 1, 2, ...
        MPI_Recv(&result_array[rank], 1, MPI_DOUBLE, rank, 99, MPI_COMM_WORLD,
                  MPI_STATUS_IGNORE);
    }
}
// ----- end of the gathering algorithm -----
```

should be substituted by one call to MPI_Gather

Receive buffer variable is declared on all processes, but memory allocation is needed only on the root process

Fortran

&

Python

→ in principle, no difference to C

Chapter 6-(1) – Exercise 1: Collective communication with MPI_Gather

C

```
result = 100.0 + 1.0 * my_rank;                                MPI/tasks/C/Ch6/solutions/gather.c
printf("I am process %i out of %i, result=%f \n",
       my_rank, num_procs, result);
if (my_rank == 0)
{ result_array = malloc(sizeof(double) * num_procs);
}
MPI_Gather(&result,1,MPI_DOUBLE,                                     Rank of root
            result_array,1,MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (my_rank == 0)
{ for (rank=0; rank<num_procs; rank++)
    printf("I'm proc 0: result of process %i is %f \n",
           rank, result_array[rank]);
}
```

Receive buffer variable
must be declared on all
processes, but memory allocation
is needed only on the root process

Fortran

```
result = 100.0 + 1.0 * my_rank                                MPI/tasks/F_30/Ch6/solutions/gather_30.f90
WRITE(*,'(A,I3,A,I3,A,I2,A,I5,A,F9.2)') &
  & 'I am process ', my_rank, ' out of ', num_procs, &
  & ' handling the ', my_rank, 'th part of n=', n, ' elements, result=' , result
IF (my_rank == 0) THEN
  ALLOCATE(result_array(0:num_procs-1))
ENDIF
CALL MPI_Gather(result,1,MPI_DOUBLE PRECISION, &                                     Rank of root
      & result_array,1,MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD)
IF (my_rank == 0) THEN
  DO rank=0, num_procs-1
    WRITE(*,'(A,I3,A,F9.2)') &
      & 'I ''m proc 0: result of process ', rank, ' is ', result_array(rank)
  END DO
ENDIF
```

Python

```
comm_world.Gather((result,1,MPI.DOUBLE),          MPI/tasks/PY/Ch6/solutions/gather.py
                  (result_array,1,MPI.DOUBLE), root=0)
```

root=0 is default,
could be omitted



Chapter 6-(1) – Exercise 2: Collective communication with MPI_Allreduce

C

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
    int my_rank, size, sum;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* Compute sum of all ranks. */
    MPI_Allreduce (&my_rank, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    printf ("PE%i:\tSum = %i\n", my_rank, sum);
    MPI_Finalize();
}
```

MPI/tasks/C/Ch6/solutions/allreduce.c

Fortran

```
PROGRAM ring
USE mpi_f08
IMPLICIT NONE
INTEGER :: my_rank, size, sum
CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
CALL MPI_Comm_size(MPI_COMM_WORLD, size)
! ... Compute sum of all ranks:
CALL MPI_Allreduce(my_rank, sum, 1,MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD)
WRITE(*,*) 'PE', my_rank, ': Sum =', sum
CALL MPI_Finalize()
END PROGRAM
```

MPI/tasks/F_30/Ch6/solutions/allreduce_30.f90

Python

```
snd_buf = np.array(my_rank, dtype=np.intc)
sum     = np.empty((), dtype=np.intc)
comm_world.Allreduce(snd_buf, (sum,1,MPI.INT), op=MPI.SUM )
# or comm_world.Allreduce((snd_buf,1,MPI.INT), (sum,1,MPI.INT), op=MPI.SUM)
# or comm_world.Allreduce(snd_buf,sum) op=MPI.SUM is default
```

MPI/tasks/PY/Ch6/solutions/allreduce.py



Chapter 6-(2): Collective communication with MPI_Ibarrier

C

```
int snd_finished=0, ib_finished=0;
while(! ib_finished) {
    /* in the role as receiving process */
    rcv_flag = 1;
    while(rcv_flag){ /* no problem to receive as many messages as possible*/
        MPI_Iprobe(MPI_ANY_SOURCE,222,MPI_COMM_WORLD,&rcv_flag,MPI_STATUS_IGNORE);
        if(rcv_flag) {
            MPI_Recv(&rcv_buf,1,MPI_INT,MPI_ANY_SOURCE,222,MPI_COMM_WORLD,&rcv_sts);
            printf(...);
        }
    }
    /* in the role as sending process */
    if(!snd_finished) {
        /* Check whether all MPI_ISSENs are finished */
        MPI_Testall(number_of_dests, snd_rq, &snd_finished, MPI_STATUSES_IGNORE);
        /* if all MPI_ISSENs are finished then call MPI_IBARRIER */
        if(snd_finished) { /*i.e., the first time, i.e., only once*/
            MPI_Ibarrier(MPI_COMM_WORLD, &ib_rq);
        }
    }
    /* loop until MPI_IBARRIER finished */
    if(snd_finished) { /* the test whether the MPI_IBARRIER is finished
                        can be done only if MPI_IBARRIER is already started.
                        This is true as soon snd_finished is true */
        MPI_Test(&ib_rq, &ib_finished, MPI_STATUS_IGNORE);
    }
}
```

MPI/tasks/C/Ch6/solutions/ibarrier.c

Fortran

& Python

→ in principle, no difference to C

MPI/tasks/F_30/Ch6/solutions/ibarrier_30.f90

MPI/tasks/PY/Ch6/solutions/ibarrier.py



Chapter 8-(1), Exercise1: Two independent sub-communicators

MPI/tasks/C/Ch8/solutions/comm-split.c

C

```
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_world_rank);
mycolor = (my_world_rank > (world_size-1)/3);
/* This definition of mycolor implies that the first color is 0 */
MPI_Comm_split(MPI_COMM_WORLD, mycolor, 0, &sub_comm);
MPI_Comm_size(sub_comm, &sub_size);
MPI_Comm_rank(sub_comm, &my_sub_rank);
MPI_Allreduce (&my_world_rank, &sumA, 1, MPI_INT, MPI_SUM, sub_comm);
MPI_Allreduce (&my_sub_rank, &sumB, 1, MPI_INT, MPI_SUM, sub_comm);
printf ("PE world:%3i, color=%i sub:%3i, SumA=%3i, SumB=%3i in sub_comm\n",
       my_world_rank, mycolor, my_sub_rank, sumA, sumB);
```

MPI/tasks/F_30/Ch8/solutions/comm-split_30.f90

Fortran

```
CALL MPI_Comm_size(MPI_COMM_WORLD, world_size)
CALL MPI_Comm_rank(MPI_COMM_WORLD, my_world_rank)
IF (my_world_rank>(world_size-1)/3) THEN; mycolor=1; ELSE; mycolor=0; ENDIF
! This definition of mycolor implies that the first color is 0
CALL MPI_Comm_split(MPI_COMM_WORLD, mycolor, 0, sub_comm)
CALL MPI_Comm_size(sub_comm, sub_size)
CALL MPI_Comm_rank(sub_comm, my_sub_rank)
CALL MPI_Allreduce (my_world_rank,sumA, 1, MPI_INTEGER, MPI_SUM, sub_comm);
CALL MPI_Allreduce (my_sub_rank, sumB, 1, MPI_INTEGER, MPI_SUM, sub_comm);
write(*,'("PE world:',I3,', color=',I3,', sub:',I3,', SumA=',I5,', SumB=',I5,', in subcomm')') &
     &           my_world_rank,      mycolor,   my_sub_rank,    sumA,        sumB
```

Python

```
sub_comm = comm_world.Split(mycolor, 0)          MPI/tasks/PY/Ch8/solutions/comm-split.py
```



Chapter 8-(2), Exercise 3: Create an inter-communicator

C

MPI/tasks/C/Ch8/solutions/intercomm.c

```
/* local leader in the lower group is locally rank 0
   (to be provided in the lower group),
   and within MPI_COMM_WORLD (i.e., in peer_comm) rank 0
   (to be provided in the upper group) */
/* local leader in the upper group is locally rank 0
   (to be provided in the upper group),
   and within MPI_COMM_WORLD (i.e., in peer_comm)
   rank 0+(size of lower group)
   (to be provided in the lower group) */
if (mycolor==0) /* This "if(...)" requires that mycolor==0 is the lower group!*/
{ /*lower group*/
    remote_leader = 0 + sub_size;
} else{ /*upper group*/
    remote_leader = 0; Ranks within MPI_COMM_WORLD
}
MPI_Intercomm_create(sub_comm,0,MPI_COMM_WORLD,remote_leader,0,
                     &inter_comm);
```

Python

```
inter_comm=sub_comm.Create_intercomm(0,comm_world,remote_leader,0)

MPI_Comm_rank(inter_comm, &my_inter_rank);

MPI_Allreduce(&my_inter_rank, &sumC, 1, MPI_INT, MPI_SUM, sub_comm);

MPI_Allreduce(&my_inter_rank, &sumD, 1, MPI_INT, MPI_SUM, inter_comm);
```

Fortran

&

Python

→ in principle, no difference to C

MPI/tasks/F_30/Ch8/solutions/intercomm_30.f90

MPI/tasks/PY/Ch8/solutions/intercomm.py



Chapter 9 – Exercise 1: Ring with virtual Cartesian topology

C

```
MPI_Comm    comm_cart;
int         dims[1], periods[1], reorder;
dims[0] = size;
periods[0] = 1;
reorder = 1;
MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, reorder, &comm_cart);
MPI_Comm_rank(comm_cart, &my_rank);
right = (my_rank+1)      % size;
left  = (my_rank-1+size) % size;
-----
MPI_Issend(&snd_buf, 1, MPI_INT, right, 17, comm_cart, &request);
MPI_Recv ( &rcv_buf, 1, MPI_INT, left, 17, comm_cart, &status);
```

MPI/tasks/C/Ch9/solutions/cart-create.c

Fortran

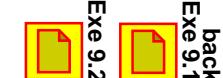
MPI/tasks/F_30/Ch9/solutions/cart-create_30.f90

```
TYPE(MPI_Comm) :: comm_cart
INTEGER :: dims(1)
LOGICAL :: periods(1), reorder
-----
dims(1) = size
periods(1) = .TRUE.
reorder = .TRUE.
CALL MPI_Cart_create(MPI_COMM_WORLD,&
& 1, dims, periods, reorder, comm_cart)
CALL MPI_Comm_rank(comm_cart, my_rank)
right = mod(my_rank+1, size)
left  = mod(my_rank-1+size, size)
-----
CALL MPI_Issend(snd_buf,...,comm_cart,...)
CALL MPI_Recv ( rcv_buf,...,comm_cart,...)
```

MPI/tasks/PY/Ch9/solutions/cart-create.py

```
dims[0] = size
periods[0] = True
reorder = True
-----
comm_cart =
  comm_world.Create_cart(dims=dims,
  periods=periods, reorder=reorder)
my_rank = comm_cart.Get_rank()
right = (my_rank+1)      % size
left  = (my_rank-1+size) % size
-----
request = comm_cart.Issend(
  (snd_buf,1,MPI.INT),right,17)
comm_cart.Recv(
  (rcv_buf,1,MPI.INT),left, 17,
  status)
```

Python



Chapter 9 – Exercise 2: Ring with virtual Cartesian topology

C

```
MPI_Comm    comm_cart;
int         dims[1], periods[1], reorder;
-----  
dims[0] = size;  periods[0] = 1;  reorder = 1;
MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, reorder, &comm_cart);
MPI_Comm_rank(comm_cart, &my_rank);
right = (my_rank+1) % size;
left = (my_rank-1+size) % size;
MPI_Cart_shift(comm_cart, 0, 1, &left, &right);
-----  
MPI_Issend(&snd_buf, 1, MPI_INT, right, 17, comm_cart, &request);
MPI_Recv (&rcv_buf, 1, MPI_INT, left, 17, comm_cart, &status);
```

Fortran

```
TYPE(MPI_Comm) :: comm_cart          MPI/tasks/F_30/Ch9/solutions/cart-shift_30.f90
INTEGER :: dims(1)
LOGICAL :: periods(1), reorder
-----  
dims(1) = size ; periods(1) = .TRUE. ; reorder = .TRUE.
CALL MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, reorder, comm_cart)
CALL MPI_Comm_rank(comm_cart, my_rank)
right = mod(my_rank+1, size)
left = mod(my_rank-1+size, size)
CALL MPI_Cart_shift(comm_cart, 0, 1, left, right)
-----  
CALL MPI_Issend(snd_buf,1,MPI_INTEGER, right, 17, comm_cart, request)
CALL MPI_Recv ( rcv_buf,1,MPI_INTEGER, left, 17, comm_cart, status)
```

Python

```
right = (my_rank+1) % size          MPI/tasks/PY/Ch9/solutions/cart-shift.py
left = (my_rank-1+size) % size
(left,right) = comm_cart.Shift(0, 1)
```

MPI/tasks/C/Ch9/solutions/cart-shift.c

MPI/tasks/F_30/Ch9/solutions/cart-shift_30.f90

MPI/tasks/PY/Ch9/solutions/cart-shift.py



Chapter 9-(2) – Exe. 5: Ring with MPI_NEIGHBOR_ALLTOALL

(major changes compared to ring with MPI_CART_CREATE+MPI_CART_SHIFT and nonblocking communication)

C

```
MPI/tasks/C/Ch9/solutions/ring_neighbor_alltoall.c
int snd_buf_arr[2], rcv_buf_arr[2];
/* snd_buf = snd_buf_arr[1], rcv_buf = rcv_buf_arr[0] */
snd_buf_arr[1] = my_rank;
arr = skeleton / red/black = solution
snd_buf_arr[0] = -1000-my_rank; /* should be never used, only for test purpose */
/* MPI_Issend(&snd_buf_arr[1],1,MPI_INT,right,to_right,new_comm,&request);
   MPI_Recv(&rcv_buf_arr[0], 1, MPI_INT, left, to_right,new_comm,&status);
   MPI_Wait(&request, &status);
*/
MPI_Neighbor_alltoall(snd_buf_arr,1,MPI_INT,rcv_buf_arr,1,MPI_INT,new_com);
new_comm.Neighbor_alltoall((snd_buf_arr,1, MPI.INT), (rcv_buf_arr,1, MPI.INT))
snd_buf_arr[1] = rcv_buf_arr[0]; If count is omitted, then correctly calculated by
sum+= rcv_buf_arr[0]; mpy4py: size(buffer)/size(datatype)/(2*cart_ndims)
```

Python

```
MPI/tasks/F_30/Ch9/solutions/ring_neighbor_alltoall_30.f90
INTEGER :: snd_buf_arr(0:1) ! snd_buf = snd_buf_arr(1)
INTEGER :: rcv_buf_arr(0:1) ! rcv_buf = rcv_buf_arr(0)
snd_buf_arr(1) = my_rank
snd_buf_arr(0) = -1000-my_rank; /* should be never used, only for test purpose */
! CALL MPI_Issend(snd_buf_arr(1),1,MPI_INTEGER,right,to_right,new_comm,request)
! CALL MPI_Recv(rcv_buf_arr(0), 1, MPI_INTEGER, left,to_right,new_comm,status)
! CALL MPI_Wait(request, status)
CALL MPI_Neighbor_alltoall(snd_buf_arr, 1, MPI_INTEGER, &
rcv_buf_arr, 1, MPI_INTEGER, new_comm)
! IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(snd_buf_arr)
snd_buf_arr(1) = rcv_buf_arr(0)
sum = sum + rcv_buf_arr(0)
```



Chapter 9-(2) – Exercise 6 (advanced)

Ring with MPI_NEIGHBOR_ALLTOALLW

MPI/tasks/C/Ch9/solutions/ring_neighbor_alltoallw.c

C

```
MPI_Aint      snd_displs[2], rcv_displs[2];
int          snd_counts[2], rcv_counts[2];
MPI_Datatype  snd_types[2], rcv_types[2];

sum = 0;
snd_buf = my_rank;

rcv_counts[0]=1; MPI_Get_address(&rcv_buf,&rcv_displs[0]); rcv_types[0]=MPI_INT;
rcv_counts[1]=0; rcv_displs[1] = 0 /*unused*/; rcv_types[1]=MPI_INT;
snd_counts[0]=0; snd_displs[0] = 0 /*unused*/; snd_types[0]=MPI_INT;
snd_counts[1]=1; MPI_Get_address(&snd_buf,&snd_displs[1]); snd_types[1]=MPI_INT;

for( i = 0; i < size; i++)
{
    /*Substituted by MPI_Neighbor_alltoallw() :
    MPI_Issend(&snd_buf, 1, MPI_INT, right, to_right, new_comm, &request);
    MPI_Recv(&rcv_buf, 1, MPI_INT, left, to_right, new_comm, &status);
    MPI_Wait(&request, &status);
    */
    MPI_Neighbor_alltoallw(MPI_BOTTOM, snd_counts, snd_displs, snd_types,
                           MPI_BOTTOM, rcv_counts, rcv_displs, rcv_types,new_comm);

    snd_buf = rcv_buf;
    sum+= rcv_buf;
}
```

Chapter 9-(2) – Exercise 6 (advanced)

Ring with MPI_NEIGHBOR_ALLTOALLW

Fortran

MPI/tasks/F_30/Ch9/solutions/ring_neighbor_alltoallw_30.f90

```
INTEGER(KIND=MPI_ADDRESS_KIND) :: snd_displs(2), rcv_displs(2)
INTEGER :: snd_counts(2), rcv_counts(2)
TYPE(MPI_Datatype) :: snd_types(2), rcv_types(2)

sum = 0
snd_buf = my_rank

rcv_counts(1) = 1; CALL MPI_Get_address(rcv_buf, rcv_displs(1));
                     rcv_types(1) = MPI_INTEGER
rcv_counts(2) = 0; rcv_displs(2) = 0; rcv_types(2) = MPI_INTEGER ! unused
snd_counts(1) = 0; snd_displs(1) = 0; snd_types(1) = MPI_INTEGER ! unused
snd_counts(2) = 1; CALL MPI_Get_address(snd_buf, snd_displs(2));
                     snd_types(2) = MPI_INTEGER

DO i = 1, size
! MPI_Issend + MPI_Recv + MPI_Wait: Substituted by:
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
  CALL MPI_Neighbor_alltoallw(MPI_BOTTOM,snd_counts,snd_displs,snd_types,&
                           MPI_BOTTOM,rcv_counts,rcv_displs,rcv_types, new_comm)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
  snd_buf = rcv_buf
  sum = sum + rcv_buf
END DO
```

Chapter 9-(2) – Exercise 6 (advanced)

Ring with MPI_NEIGHBOR_ALLTOALLW

MPI/tasks/PY/Ch9/solutions/ring_neighbor_alltoallw.py

Python

```
snd_displs = [None]*2; rcv_displs = [None]*2
snd_counts = [None]*2; rcv_counts = [None]*2
snd_types = [None]*2; rcv_types = [None]*2

snd_buf = np.array(my_rank, dtype=np.intc)
rcv_buf = np.array(-1000, dtype=np.intc)

mem_from_bottom = MPI.memory.fromaddress(MPI.BOTTOM,0,0)

rcv_counts[0] = 1
rcv_displs[0] = MPI.Get_address(rcv_buf)
rcv_types[0] = MPI.INT
rcv_counts[1] = 0
rcv_displs[1] = 0 # unused
rcv_types[1] = MPI.INT

snd_counts[0] = 0
snd_displs[0] = 0 # unused
snd_types[0] = MPI.INT
snd_counts[1] = 1
snd_displs[1] = MPI.Get_address(snd_buf)
snd_types[1] = MPI.INT

for i in range(size):
    # Substituted by MPI_Neighbor_alltoallw() :
    # MPI_Issend(&snd_buf, 1, MPI_INT, right, 17, new_comm, &request);
    # MPI_Recv(&rcv_buf, 1, MPI_INT, left, 17, new_comm, &status);
    # MPI_Wait(&request, &status);
    new_comm.Neighbor_alltoallw(
        (mem_from_bottom, snd_counts, snd_displs, snd_types),
        (mem_from_bottom, rcv_counts, rcv_displs, rcv_types))
```

Chapter 9-(3) – Exercise

MPIX_Cart_weighted_create

MPI/tasks/C/Ch9/MPIX/halo_irecv_send_toggle_3dim_grid_solution.c

```
if (cart_method == 1) {
    if (my_world_rank==0)
        printf("cart_method==1: MPI_Dims_create + MPI_Cart_create\n");
    MPI_Dims_create(size, ndims, dims);
    MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, 0, &comm_cart);
} else if (cart_method == 2) {
    if (my_world_rank==0)
        printf("cart_method==2: MPIX_Cart_weighted_create(MPIX_WEIGHTS_EQUAL)\n");
    /* TODO: Appropriate call to MPIX_Cart_weighted_create(...) with MPIX_WEIGHTS_EQUAL
       instead of calling MPI_Dims_create() and MPI_Cart_create() as in method 1 */
    MPIX_Cart_weighted_create(MPI_COMM_WORLD, ndims, MPIX_WEIGHTS_EQUAL, periods,
                             MPI_INFO_NULL, dims, &comm_cart);
} else if (cart_method == 3) {
    /* TODO: Appropriate calculation of weights[] based on meshsize_avg_per_proc_startval[] */
    for (d=0; d<ndims; d++) weights[d] = 4.0 / meshsize_avg_per_proc_startval[d];
    if (my_world_rank==0)
    { printf("cart_method==3: MPIX_Cart_weighted_create( weights := 4.0 / meshsize_avg_per_proc_startval )\n");
      printf("weights= ");
      for (d=0; d<ndims; d++) printf(" %lf",weights[d]); printf("\n");
    }
    /* TODO: Appropriate call to MPIX_Cart_weighted_create(...) with weights
       instead of MPIX_WEIGHTS_EQUAL as in method 2 */
    MPIX_Cart_weighted_create(MPI_COMM_WORLD, ndims, weights, periods,
                             MPI_INFO_NULL, dims, &comm_cart);
} else ...
```

Take the arguments

Chapter 10: Ring with one-sided communication

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

Fortran

```
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

```
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)
```

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



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

```
INTEGER, ASYNCHRONOUS::snd_buf, rcv_buf MPI/tasks/F_30/Ch10/solutions/ring-1sided-put_30.f90
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)
```

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

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.

Python

```
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

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

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);  
B 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. 26
27
- 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. 28
29
- 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. 30
31
- 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. 32
33
34
35
36
37
38

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>

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

C

MPI/tasks/C/Ch11/solutions/ring_1sided_put_win_alloc_shared.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;
}
```

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

Fortran

```
USE mpi_f08      MPI/tasks/F_30/Ch11/solutions/ring_1sided_put_win_alloc_shared_30.f90
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

```
from mpi4py import MPI  MPI/tasks/PY/Ch11/solutions/ring_1sided_put_win_alloc_shared.py
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.

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 neighbor'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.

Fortran

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

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 :: recv_buf(:)
TYPE(C_PTR) :: ptr_recv_buf

sum = 0
snd_buf = my_rank_sm
DO i = 1, size_sm
    IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(recv_buf)
    CALL MPI_WIN_FENCE(0, win) ! Workaround: no assertions
    IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(recv_buf)
    recv_buf(0+(right-my_rank_sm)) = snd_buf
    IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(recv_buf)
    CALL MPI_WIN_FENCE(0, win) ! Workaround: no assertions
    IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(recv_buf)
    IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
    {
        snd_buf = recv_buf(0)
        sum = sum + recv_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 recv_buf across the fences in current and next loop iteration.

New:
Needed to prevent movement of recv_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

```
np_dtype = np.intc                                     MPI/tasks/PY/Ch11/solutions/ring_1sided_store_win_alloc_shared.py
# 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 substituted 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

Chapter 11-(1) Exercise 3: Mixed pt-to-pt and shared memory ring communication

MPI/tasks/C/Ch11/solutions/ring-1sided-mixed-issend-recv-store.c

C

```
for( i = 0; i < size_sm; i++)
{
    if(right_sm_sub == MPI_UNDEFINED)
        MPI_Issend(&snd_buf, 1, MPI_INT, right, 17, MPI_COMM_WORLD, &request);
    if(left_sm_sub == MPI_UNDEFINED)
        MPI_Recv(rcv_buf_ptr, 1, MPI_INT, left, 17, MPI_COMM_WORLD, &status);
    if(right_sm_sub == MPI_UNDEFINED)
        MPI_Wait(&request, &status);

    MPI_Win_fence(/*workaround: no assertions:*/ 0, win);
    // collective, therefore all processes in comm_sm_sub must call

    // 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 neighbor's rcv_buf):
    if(right_sm_sub != MPI_UNDEFINED)
        *(rcv_buf_ptr+(right_sm_sub-my_rank_sm_sub)) = snd_buf;

    MPI_Win_fence(/*workaround: no assertions:*/ 0, win);
    // collective, therefore all processes in comm_sm_sub must call

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

Fortran

→ in principle, no difference to C

MPI/tasks/F_30/Ch11/solutions/ring-1sided-mixed-issend-recv-store_30.f90

Chapter 11-(1) Exercise 4 (advanced)

Halo-copy shared memory one-sided comm.

C

```
int k;                                MPI/tasks/C/halo-benchmarks/halo_1sided_store_win_alloc_shared.c
int offset_left, offset_right;
-----
MPI_Win_allocate_shared((MPI_Aint)(max_length*sizeof(float)),
    sizeof(float), MPI_INFO_NULL, MPI_COMM_WORLD, &rcv_buf_left,
    &win_rcv_buf_left);
MPI_Win_allocate_shared((MPI_Aint)(max_length*sizeof(float)),
    sizeof(float), MPI_INFO_NULL, MPI_COMM_WORLD, &rcv_buf_right,
    &win_rcv_buf_right);
/*offset_left is defined so that rcv_buf_left(xxx+offset_left) in
process 'my_rank' is the same location as rcv_buf_left(xxx) in
process 'left': */
offset_left = (left-my_rank)*max_length;
/*offset_right is defined so that rcv_buf_right(xxx+offset_right) in
process 'my_rank' is the same location as rcv_buf_right(xxx) in
process 'right': */
offset_right = (right-my_rank)*max_length;
-----
/* MPI_Put(snd_buf_left, length, MPI_FLOAT, left, (MPI_Aint)0, length,
   MPI_FLOAT, win_rcv_buf_right); */
/* MPI_Put(snd_buf_right, length, MPI_FLOAT, right, (MPI_Aint)0, length,
   MPI_FLOAT, win_rcv_buf_left); ... is substituted by: */
for(k=0; k<length; k++) rcv_buf_right[k+offset_left] = snd_buf_left [k];
for(k=0; k<length; k++) rcv_buf_left [k+offset_right]= snd_buf_right[k];
And all fences without assertions (as long as not otherwise standardized):
MPI_Win_fence( /*workaround: no assertions:*/ 0, ...);
```

Chapter 11-(1) Exercise 4 (advanced)

Halo-copy shared memory one-sided comm.

MPI/tasks/F_30/halo-benchmarks/halo_1sided_store_win_alloc_shared_30.f90

Fortran

```
INTEGER :: offset_left, offset_right

-----  
CALL MPI_Win_allocate_shared(buf_size, disp_unit, MPI_INFO_NULL,  
                           MPI_COMM_WORLD, ptr_rcv_buf_left, win_rcv_buf_left)  
CALL C_F_POINTER(ptr_rcv_buf_left, rcv_buf_left, (/max_length/))  
! offset_left is defined so that rcv_buf_left(xxx+offset_left) in process  
! 'my_rank' is the same location as rcv_buf_left(xxx) in process 'left':  
offset_left = (left-my_rank)*max_length  
-----  
CALL MPI_Win_allocate_shared(buf_size, disp_unit, MPI_INFO_NULL,  
                           MPI_COMM_WORLD, ptr_rcv_buf_right, win_rcv_buf_right)  
CALL C_F_POINTER(ptr_rcv_buf_right, rcv_buf_right, (/max_length/))  
! offset_right is defined so that rcv_buf_right(xxx+offset_right) in proc.  
! 'my_rank' is the same location as rcv_buf_right(xxx) in process 'right':  
offset_right = (right-my_rank)*max_length
```

Substitution of MPI_Put → see next slide

Chapter 11-(1) Exercise 4 (advanced)

Halo-copy shared memory one-sided comm.

Fortran

```
CALL MPI_Win_fence( 0, win_rcv_buf_left) ! Workaround: no assertions
CALL MPI_Win_fence( 0, win_rcv_buf_right) ! Workaround: no assertions

! CALL MPI_Get_address(rcv_buf_right, iadummy)
! CALL MPI_Get_address(rcv_buf_left, iadummy)
! ... or with MPI-3.0 and later:
IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(rcv_buf_right)
IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(rcv_buf_left)

! CALL MPI_Put(snd_buf_left, length, MPI_REAL, left, target_disp, &
!             length, MPI_REAL, win_rcv_buf_right)
! CALL MPI_Put(snd_buf_right, length, MPI_REAL, right, target_disp, &
!             length, MPI_REAL, win_rcv_buf_left)
! ... is substituted by:
rcv_buf_right(1+offset_left:length+offset_left) = snd_buf_left(1:length)
rcv_buf_left(1+offset_right:length+offset_right) = snd_buf_right(1:length)

! CALL MPI_Get_address(rcv_buf_right, iadummy)
! CALL MPI_Get_address(rcv_buf_left, iadummy)
! ... or with MPI-3.0 and later:
IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(rcv_buf_right)
IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(rcv_buf_left)

CALL MPI_Win_fence( 0, win_rcv_buf_left ) ! Workaround: no assertions
CALL MPI_Win_fence( 0, win_rcv_buf_right ) ! Workaround: no assertions
```

MPI_F_SYNC_REG(rcv_buf_right/left) guarantees
that the assignments rcv_buf_right/left = ...
must not be moved across both MPI_Win_fence

Chapter 11-(1) Exercise 5 (advanced)

MPI_Bcast into shared memory (a one-slide-solution)

- Solution files:
 - data-rep_sol_2a.c
 - data-rep_sol_2d.c
 - data-rep_sol_2f.c
 - data-rep_sol_3-6.c
 - data-rep_sol_7.c
 - data-rep_solution.c

Chapter 11-(1) Exercise 5 (advanced)

MPI_Bcast into shared memory (a 1-slide-solution)

MPI/tasks/C/Ch11/data-rep/data-rep_solution.c

C

```
arr = (arrType *) malloc(arrSize * sizeof(arrType)), grey = original code
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, /*key=*/ 0,
                     MPI_INFO_NULL, &comm_shm);
MPI_Comm_size(comm_shm, &size_shm);
MPI_Comm_rank(comm_shm, &rank_shm);
if (rank_shm == 0) { individualShmSize = arrSize; }
else { individualShmSize = 0; }
MPI_Win_allocate_shared(
    (MPI_Aint)(individualShmSize) * (MPI_Aint)(sizeof(arrType)),
    sizeof(arrType), MPI_INFO_NULL, comm_shm, &shm_buf_ptr, &win );
MPI_Win_shared_query( win, 0, &arrSize_, &disp_unit, &arr );
color=MPI_UNDEFINED; if (rank_shm==0) { color = 0; }
MPI_Comm_split(MPI_COMM_WORLD, color, /*key=*/ 0, &comm_head);
if( comm_head != MPI_COMM_NULL )
{MPI_Comm_size(comm_head, &size_head);MPI_Comm_rank(comm_head, &rank_head);}
MPI_Win_fence(/*workaround: no assertions:*/ 0, win); Starting write epoch
if(rank_world==0) for( i=0; i<arrSize; i++) arr[i]=i+it; Filling arr
if( comm_head != MPI_COMM_NULL ) Only the heads of the shared memory islands fill arr by ...
    MPI_Bcast(arr, arrSize, arrDataType, 0, comm_head); ... broadcasting to all heads
}
instead of MPI_COMM_WORLD
MPI_Win_fence(/*workaround: no assertions:*/ 0, win); Starting read epoch by all proc's
sum=0; for( i=0; i<arrSize; i++) sum+= arr[i]; Reading arr
```

The following slides show a step-by-step solving of this exercise

Solutions of MPI shared memory exercise: datarep

data-rep_base.c

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

typedef long arrType ;
#define arrDataType MPI_LONG /* !!!!! A C H T U N G : MPI_Type an arrType anpassen !!!!!
*/
static const int arrSize=16*1.6E7 ;

int main (int argc, char *argv[])
{
    int it ;
    int rank_world, size_world;
    arrType *arr ;
    int i;
    long long sum ;

/* ===> 1 <== */
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank_world);
    MPI_Comm_size(MPI_COMM_WORLD, &size_world);

/* ===> 2 <== */
    arr = (arrType *) malloc(arrSize * sizeof(arrType));
    if(arr == NULL)
    {
        printf("arr NOT allocated, not enough memory\n");
        MPI_Abort(MPI_COMM_WORLD, 0);
    }
    ...

    int it ;
```

In each process, allocating an array for the replicated
TODO: Allocating only once per shared memory node!
This will be done in 3 steps: 2a, 2b-d, 2e-f

Solutions of MPI shared memory exercise: datarep

datarep_base.c (continued)

```
...
/* ==> 3 <== */
for( it = 0; it < 3; it++)
{
    /* only rank_world=0 initializes the array arr */
    if( rank_world == 0 )
    {
        for( i = 0; i < arrSize; i++)
            { arr[i] = i + it ; }
    }
/* ==> 4 <== */
MPI_Bcast( arr, arrSize, arrDataType, 0, MPI_COMM_WORLD );
/* Now, all arrays are filled with the same content. */

/* ==> 5 <== */
sum = 0;
for( i = 0; i < arrSize; i++)
{
    sum+= arr [ i ] ;
}

/* ==> 6 <== */
/*TEST*/ // To minimize the output, we print only from 3 process per SMP node
/*TEST*/ if ( rank_world == 0 || rank_world == 1 || rank_world == size_world - 1 )
    printf ("it: %i, rank ( world: %i ): tsum(i=%i...i=%i) = %lld \n",
           it, rank_world, it, arrSize-1+it, sum);

/* ==> 7 <== */
free(arr);
MPI_Finalize();
}
```

Time step loop

Together in 1 step!

Filling the array by one process.
Will be unchanged.

Broadcasting it to all other processes.

TODO: Only one process per SMP node should broadcast!

Calculating some numerical result in each process.
Will be unchanged.

And printing it out
Will be unchanged.

Freeing the allocated array.
TODO: We must free the window instead.

Last step!

Solutions of MPI shared memory exercise: datarep

data-rep_sol_2a.c

```
...
MPI_Comm comm_shm;
int size_shm, rank_shm;
...

/* ===> 2 <== */
/* Create --> shared memory islands and --> shared memory window inside */
/*          -->      comm_shm           and           -->      win           */

MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, /*key=*/ 0, MPI_INFO_NULL, &comm_shm);
MPI_Comm_size(comm_shm, &size_shm);
MPI_Comm_rank(comm_shm, &rank_shm);
/*TEST*/ // To minimize the output, we print only from 3 process per SMP node
/*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
    printf("\t\trank ( world: %i, shm: %i)\n", rank_world, rank_shm);
/*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!!\n"); MPI_Finalize(); return 0;

/* TO DO:
 * substitute the following malloc
*/
...
```

Solutions of MPI shared memory exercise: datarep

C

data-rep_sol_2d.c

```
...
MPI_Win win;
int individualShmSize ;
arrType *shm_buf_ptr;

/* output MPI_Win_shared_query */
MPI_Aint arrSize_ ;
int disp_unit ;

/* ===> 2 <==== */

/* instead of: arr = (arrType *) malloc(arrSize * sizeof(arrType)); */
if ( rank_shm == 0 )
{ individualShmSize = arrSize_ ; }
else
{ individualShmSize = 0 ; }
MPI_Win_allocate_shared( (MPI_Aint)(individualShmSize) * (MPI_Aint)(sizeof(arrType)),
                        sizeof(arrType), MPI_INFO_NULL, comm_shm, &shm_buf_ptr, &win );
/* shm_buf_ptr is not used because it is only available in process rank_shm==0 */
MPI_Win_shared_query( win, 0, &arrSize_, &disp_unit, &arr );

/*TEST*/ // To minimize the output, we print only from 3 process per SMP node
/*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
printf("\t\trank ( world: %i, shm: %i) arrSize %i arrSize_ %i shm_buf_ptr = %p, arr_ptr = %p \n",
       rank_world, rank_shm, arrSize, (int) (arrSize_), shm_buf_ptr, arr );
/*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!.\n"); MPI_Finalize(); return 0;

/* TO DO: Create communicator comm_head with MPI_Comm_split --> including all the rank_shm == 0
processes.

...
```

data-rep_sol_2d_30.f90

```
...
! INTEGER*8, DIMENSION(:), ALLOCATABLE :: arr
INTEGER*8, DIMENSION(:), POINTER :: arr
...
/* ===> 1 <==== */ ] similar to C
...
/* ===> 2 <==== */
...
```

→ See next slide

Fortran

Solutions of MPI shared memory exercise: datarep

Fortran

data-rep_sol_2d_f90.c

```
! INTEGER*8, DIMENSION(:), ALLOCATABLE :: arr
INTEGER*8, DIMENSION(:), POINTER :: arr
/* ===> 1 <==== */
TYPE(MPI_Win) :: win
INTEGER :: individualShmSize
TYPE(C_PTR) :: arr_ptr, shm_buf_ptr
INTEGER(KIND=MPI_ADDRESS_KIND) :: arrDataTypeSize, lb, ShmByteSize
! /* output MPI_Win_shared_query */
INTEGER(kind=MPI_ADDRESS_KIND) :: arrSize_
INTEGER :: disp_unit
/* ===> 2 <==== */
! instead of: ALLOCATE(arr(1:arrSize))
IF ( rank_shm == 0 ) THEN
    individualShmSize = arrSize_
ELSE
    individualShmSize = 0
ENDIF
CALL MPI_Type_get_extent(arrDataType, lb, arrDataTypeSize)
ShmByteSize = individualShmSize * arrDataTypeSize
disp_unit = arrDataTypeSize
CALL MPI_Win_allocate_shared( ShmByteSize, disp_unit, MPI_INFO_NULL, comm_shm, shm_buf_ptr, win )
! /* shm_buf_ptr is not used because it is only available in process rank_shm==0 */
CALL MPI_Win_shared_query( win, 0, arrSize_, disp_unit, arr_ptr )
CALL C_F_POINTER(arr_ptr, arr, (/arrSize/) )
! TEST: To minimize the output, we print only from 3 process per SMP node
IF ( (rank_shm == 0) .OR. (rank_shm == 1) .OR. (rank_shm == size_shm - 1) ) THEN
    WRITE(*,*) 'rank( world=',rank_world,' shm=',rank_shm,')',' arrSize=',arrSize,' arrSize_=',arrSize_
ENDIF
IF (rank_world == 0) WRITE(*,*) 'ALL finalize and return!!!'; CALL MPI_Finalize(); STOP
...
```

Solutions of MPI shared memory exercise: datarep

Python

data-rep_sol_2d.py

```
arrType = np.int_
arrDataType = MPI.LONG
arrSize=int(16*1.6E7)
# ===> 1 <===
comm_world=MPI.COMM_WORLD; rank_world=comm_world.Get_rank(); size_world=comm_world.Get_size()
# ===> 2 <===
#Create → shared memory islands and → shared memory window inside
# → comm_shm and → win
comm_shm = comm_world.Split_type(MPI.COMM_TYPE_SHARED, key=0, info=MPI.INFO_NULL)
size_shm = comm_shm.Get_size(); rank_shm = comm_shm.Get_rank()
# instead of: arr = np.empty(arrSize, dtype=arrType)
if (rank_shm == 0):
    individualShmSize = arrSize
else:
    individualShmSize = 0
win = MPI.Win.Allocate_shared( individualShmSize * arrType(0).itemsize, arrType(0).itemsize,
    MPI.INFO_NULL, comm_shm )
shm_buf_ptr = win.tomemory().address # This is actually only the address in memory, not a real pointer.
# pointer/memory from win object is not used because it is only available in process rank_shm==0
(buf, disp_unit) = win.Shared_query(0)
# This is only an assertion that all types match, not necessary.
assert disp_unit == arrDataType.Get_size() ; assert disp_unit == arrType(0).itemsize
# To use the raw memory, we consider it as an numpy array
arr = np.frombuffer(buf, dtype=arrType); arrSize_ = arr.nbytes
# To minimize the output, we print only from 3 process per SMP node # TEST #
if (rank_shm == 0 or rank_shm == 1 or rank_shm == size_shm - 1): # TEST #
    print("\t\trank (world:{}, shm:{}), arrSize_{} arrSize_{} shm_buf_ptr= 0x{:x}, arr_ptr= 0x{:x}.".format(
        rank_world, rank_shm, arrSize_, arrSize_, shm_buf_ptr, buf.address ))
if(rank_world==0): # TEST #
    print(f"ALL finalize and return !!!")
sys.exit(0) # Remember sys.exit() still calls all exit handlers, i.e. also MPI.Finalize()
```

See Data types — NumPy v1.22.dev0 Manual → C long

corresponding to C basic datatype handle MPI_LONG



Solutions of MPI shared memory exercise: datarep

data-rep_sol_2f.c

```
...
int color ;
MPI_Comm comm_head;
int size_head, rank_head;
...
/* ==> 2 <== */
...
/* Create communicator including all the rank_shm = 0 */  

/* with the MPI_Comm_split: in color 0 all the rank_shm = 0 , */  

/* all other ranks are color = 1 */  

color=MPI_UNDEFINED ;
if (rank_shm==0) color = 0 ;

MPI_Comm_split(MPI_COMM_WORLD, color, /*key=*/ 0, &comm_head);
rank_head = -1; // only used in the print statements to differentiate unused rank== -1 from used rank==0
if( comm_head != MPI_COMM_NULL ) // if( color == 0 ) // rank is element of comm_head, i.e., it is head  

of one of the islands in comm_shm
{
    MPI_Comm_size(comm_head, &size_head);
    MPI_Comm_rank(comm_head, &rank_head);
}

/*TEST*/ // To minimize the output, we print only from 3 process per SMP node
/*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
    printf("\t\trank ( world: %i, shm: %i, head: %i) arrSize %i arrSize_ %i  shm_buf_ptr = %p, arr_ptr = %p \n",
           rank_world, rank_shm, rank_head, arrSize, (int) (arrSize_), shm_buf_ptr, arr );
/*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!!\n"); MPI_Finalize(); return 0;...
```

Solutions of MPI shared memory exercise: datarep

data-rep_sol_3-6.c (on this slide steps 3-4)

```
...
/* ===> 3 <== */
for( it = 0; it < 3; it++)
{
/* only rank_world=0 initializes the array arr */
/* all rank_shm=0 start the write epoch: writing arr to their shm */
MPI_Win_fence(/*workaround: no assertions:*/ 0, win);
if( rank_world == 0 ) /* from those rank_shm=0 processes, only rank_world==0 fills arr */
{
    for( i = 0; i < arrSize; i++)
    { arr[i] = i + it ; }
}

/* ===> 4 <== */
/* Instead of all processes in MPI_COMM_WORLD, now only the heads of the
 * shared memory islands communicate (using comm_head).
 * Since we used key=0 in both MPI_Comm_split(...), process rank_world = 0
 * - is also rank 0 in comm_head
 * - and rank 0 in comm_shm in the color it belongs to. */

if( comm_head != MPI_COMM_NULL ) // if( color == 0 )
{
    MPI_Bcast(arr, arrSize, arrDataType, 0, comm_head);
    /* with this Bcast, all other rank_shm=0 processes write the data into their arr */
}
...
...
```

Solutions of MPI shared memory exercise: datarep

data-rep_sol_3-6.c (on this slide steps 5-6)

```
...
/* ===> 5 <== */
MPI_Win_fence(/*workaround: no assertions:*/ 0, win);
              // after the fence all processes start a read epoch

/* Now, all other ranks in the comm_sm shared memory islands are allowed to access their shared memory
array. */
/* And all ranks rank_sm access the shared mem in order to compute sum */
sum = 0;
for( i = 0; i < arrSize; i++)
{
    //sum+= *( shm_buf_ptr - rank_shm * shmSize + i ) ;
    sum+= arr [ i ] ;
}

/* ===> 6 <== */
/*TEST*// To minimize the output, we print only from 3 process per SMP node
/*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
    printf ("it: %i, rank ( world: %i, shm: %i, head: %i ):tsum(i=%d...i=%d) = %lld \n",
           it, rank_world, rank_shm, rank_head, it, arrSize-1+it, sum );
}
/*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!!\n"); MPI_Finalize(); return 0;
...

```

Solutions of MPI shared memory exercise: datarep

data-rep_sol_7.c

```
...  
/* ===> 7 <==== */  
MPI_Win_fence(/*workaround: no assertions:*/ 0, win);  
    // free destroys the shm. fence to guarantee that read epoch has been finished  
MPI_Win_free(&win);  
...
```

data-rep_solution.c

```
...  
/* ===> 2 <==== */  
...  
// ADD ON: calculates the minimum and maximum size of size_shm  
int mm[2], minmax[2]; mm[0] = -size_shm ; mm[1] = size_shm ;  
  
if( comm_head != MPI_COMM_NULL )  
{  
    MPI_Reduce( mm, minmax, 2, MPI_INT, MPI_MAX, 0, comm_head) ;  
}  
if( rank_world == 0 )  
{  
    printf("\n\tThe number of shared memory islands is: %i islands \n", size_head ) ;  
    if ( minmax[0] + minmax[1] == 0 )  
        printf("\tThe size of all shared memory islands is: %i processes\n", -minmax[0] ) ;  
    else  
        printf("\tThe size of the shared memory islands is between min = %i and max = %i processes \n",  
              -minmax[0], minmax[1]);  
}  
// End of ADD ON. Note that the following algorithm does not require same sizes of the shared memory  
islands  
  
/* ===> 3 <==== */  
...
```

Trick:

Calculate the minimum through
calculating the maximum for the negative values

Chapter 11-(2) Exercise 6: Ring with shared memory and MPI_Win_sync

C

MPI/tasks/C/Ch11/solutions/ring-1sided-store-win-alloc-shared-othersync.c

```
MPI_Request rq;
MPI_Status status;
int snd_dummy, rcv_dummy;

MPI_Win_allocate_shared(...);
MPI_Win_lock_all(MPI_MODE_NOCHECK, win);

/* In Fortran, a register-sync would be here needed:
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf) */
MPI_Win_sync(win);
MPI_Irecv(&rcv_dummy, 0, MPI_INT, right, 17, MPI_COMM_WORLD, &rq);
MPI_Send(&snd_dummy, 0, MPI_INT, left, 17, MPI_COMM_WORLD);
MPI_Wait(&rq, &status);
MPI_Win_sync(win);
/* In Fortran ... IF (...) CALL MPI_F_sync_reg(rcv_buf) */

*(rcv_buf_ptr+(right-my_rank)) = snd_buf;

/* In Fortran ... IF (...) CALL MPI_F_sync_reg(rcv_buf) */
MPI_Win_sync(win);
MPI_Irecv(&rcv_dummy, 0, MPI_INT, left, 18, MPI_COMM_WORLD, &rq);
MPI_Send(&snd_dummy, 0, MPI_INT, right, 18, MPI_COMM_WORLD);
MPI_Wait(&rq, &status);
MPI_Win_sync(win);
/* In Fortran ... IF (...) CALL MPI_F_sync_reg(rcv_buf) */

MPI_Win_unlock_all(win);
MPI_Win_free(&win);
```

To my_rank

-1

Instead of
MPI_Win_fence(...)

To my_rank

+1

Instead of
MPI_Win_fence(...)

Fortran

Python

© 2000-2021 HLRS, Rolf Rabenseifner

REC → [online](#)

MPI/tasks/F_30/Ch11/solutions/ring-1sided-store-win-alloc-shared-othersync_30.f90

MPI/tasks/PY/Ch11/solutions/ring-1sided-store-win-alloc-shared-othersync.py



Chapter 12-(1), Exercise 1: MPI_TYPE_CONTIGUOUS

C

MPI/tasks/C/Ch12/solutions/derived-contiguous.c

```
struct buff{
    int i;
    int j;
} snd_buf, rcv_buf, sum; }  
Provided in  
the skeleton  
  
MPI_Datatype send_recv_type;  
  
MPI_Type_contiguous(2, MPI_INT, &send_recv_type);  
MPI_Type_commit(&send_recv_type);  
  
sum.i = 0; sum.f = 0;  
snd_buf.i = my_rank; snd_buf.j = 10*my_rank;  
for( i = 0; i < size; i++)  
{ MPI_Issend(&snd_buf, 1, send_recv_type, right, 17, MPI_COMM_WORLD, &request);  
    MPI_Recv (&rcv_buf, 1, send_recv_type, left, 17, MPI_COMM_WORLD, &status);  
    MPI_Wait(&request, &status);  
    snd_buf = rcv_buf;  
    sum.i += rcv_buf.i; sum.j += rcv_buf.j;  
}  
  
printf ("PE %i: Sum = %i and %i \n", my_rank, sum.i, sum.j);
```

Chapter 12-(1), Exercise 1: MPI_TYPE_CONTIGUOUS

Fortran

```
MPI/tasks/F_30/Ch12/solutions/derived_contiguous_30.f90

TYPE t
  SEQUENCE
    INTEGER :: i
    INTEGER :: j
  END TYPE t

  } Provided in
  the skeleton

TYPE(MPI_Datatype) :: send_recv_type
CALL MPI_Type_contiguous(2, MPI_INT, send_recv_type)
CALL MPI_Type_commit(send_recv_type)
-----
sum%i = 0          ; sum%r = 0 ;
snd_buf%i = my_rank ; snd_buf%j = my_rank
DO i = 1, size
  CALL MPI_Issend(snd_buf, 1, send_recv_type, right, 17, MPI_COMM_WORLD, request)
  CALL MPI_Recv ( rcv_buf, 1, send_recv_type, left, 17, MPI_COMM_WORLD, status)
  CALL MPI_Wait(request, status)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
  snd_buf = rcv_buf
  sum%i = sum%i + rcv_buf%i ; sum%j = sum%j + rcv_buf%j
END DO
WRITE(*,*) 'PE', my_rank, ': Sum%i =', sum%i, ' Sum%j =', sum%j
```

Python

```
MPI/tasks/PY/Ch12/solutions/derived_contiguous.py

np_dtype = np.dtype([('i', np.intc), ('j', np.intc)])
snd_buf = np.empty((), dtype=np_dtype)
rcv_buf = np.empty_like(snd_buf)
  } Provided in
  the skeleton

send_recv_type = MPI.INT.Create_contiguous(2)
send_recv_type.Commit()
-----
request = comm_world.Issend((snd_buf, 1, send_recv_type), right, 17)
comm_world.Recv((rcv_buf, 1, send_recv_type), left, 17, status)
```



Chapter 12-(1), Exercise 2: Halo-copy with derived types

C

```
struct buff{
    int i;
    float f;
} snd_buf, rcv_buf, sum;

int      array_of_blocklengths[2];
MPI_Aint array_of_displacements[2], first_var_address, second_var_address;
MPI_Datatype array_of_types[2], send_recv_type;

array_of_types[0] = MPI_INT;  array_of_types[1] = MPI_FLOAT;
array_of_blocklengths[0] = 1;  array_of_blocklengths[1] = 1;
MPI_Get_address(&snd_buf.i, &first_var_address);
MPI_Get_address(&snd_buf.f, &second_var_address);
array_of_displacements[0] = (MPI_Aint) 0;
array_of_displacements[1] = MPI_Aint_diff(second_var_address, first_var_address);
MPI_Type_create_struct(2, array_of_blocklengths, array_of_displacements,
                      array_of_types, &send_recv_type);
MPI_Type_commit(&send_recv_type);

sum.i = 0;                  sum.f = 0;
snd_buf.i = my_rank;        snd_buf.f = 10*my_rank;
for( i = 0; i < size; i++)
{ MPI_Issend(&snd_buf, 1, send_recv_type, right, 17, MPI_COMM_WORLD, &request);
  MPI_Recv (&rcv_buf, 1, send_recv_type, left, 17, MPI_COMM_WORLD, &status);
  MPI_Wait(&request, &status);
  snd_buf = rcv_buf;
  sum.i += rcv_buf.i;  sum.f += rcv_buf.f;
}
printf ("PE %i: Sum = %i and %f \n", my_rank, sum.i, sum.f);
```

MPI/tasks/C/Ch12/solutions/derived-struct.c

Provided in
the skeleton

Chapter 12-(1), Exercise 2: Halo-copy with derived types

Fortran

```
TYPE t
  SEQUENCE
    INTEGER :: i
    REAL   :: r
  END TYPE t
  TYPE(t), ASYNCHRONOUS :: snd_buf
  TYPE(t) :: rcv_buf, sum
  TYPE(MPI_Datatype) :: send_recv_type
  INTEGER(KIND=MPI_ADDRESS_KIND) :: array_of_displacements(2)
  INTEGER(KIND=MPI_ADDRESS_KIND) :: first_var_address, second_var_address
  -----
  CALL MPI_Get_address(snd_buf%i, first_var_address)
  CALL MPI_Get_address(snd_buf%r, second_var_address)
  array_of_displacements(1) = 0
  array_of_displacements(2)=MPI_Aint_diff(second_var_address,first_var_address)
  CALL MPI_Type_create_struct(2, (/1,1/), &
    & array_of_displacements, (/MPI_INTEGER,MPI_REAL/), send_recv_type)
  CALL MPI_Type_commit(send_recv_type)
  -----
  sum%i = 0           ; sum%r = 0 ;
  snd_buf%i = my_rank ; snd_buf%r = REAL(10*my_rank)
  DO i = 1, size
    CALL MPI_Issend(snd_buf,1,send_recv_type,right,17,MPI_COMM_WORLD,request)
    CALL MPI_Recv ( rcv_buf,1,send_recv_type, left, 17,MPI_COMM_WORLD,status)
    CALL MPI_Wait(request, status)
    IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
    snd_buf = rcv_buf
    sum%i = sum%i + rcv_buf%i ; sum%r = sum%r + rcv_buf%r
  END DO
  WRITE(*,*) 'PE', my_rank, ': Sum%i =', sum%i, ' Sum%r =', sum%r

```

MPI/tasks/F_30/Ch12/solutions/derived_struct_30.f90

Provided in
the skeleton

Chapter 12-(1), Exercise 2: Halo-copy with derived types

Python

MPI/tasks/PY/Ch12/solutions/derived_struct.py

```
np_dtype = np.dtype([('i', np.intc), ('f', np.single)])
snd_buf = np.empty((), dtype=np_dtype)
rcv_buf = np.empty_like(snd_buf); sum = np.empty_like(snd_buf)
array_of_blocklengths = [None]*2
array_of_displacements = [None]*2
array_of_types = [None]*2
array_of_blocklengths[0] = 1; array_of_types[0] = MPI.INT
array_of_blocklengths[1] = 1; array_of_types[1] = MPI.FLOAT
first_var_address = MPI.Get_address(snd_buf['i'])
second_var_address = MPI.Get_address(snd_buf['f'])
array_of_displacements[0]= 0
array_of_displacements[1]=MPI.Aint_diff(second_var_address,first_var_address)
send_recv_type = MPI.Datatype.Create_struct(array_of_blocklengths,
                                             array_of_displacements, array_of_types)
send_recv_type.Commit()
sum['i'] = 0; sum['f'] = 0
snd_buf['i'] = my_rank; snd_buf['f'] = 10*my_rank # Step 1 = init
for i in range(size):
    request = comm_world.Issend((snd_buf, 1, send_recv_type), right, 17) # St. 2a
    comm_world.Recv((rcv_buf, 1, send_recv_type), left, 17, status) # Step 3
    request.Wait(status)
    np.copyto(snd_buf, rcv_buf)
    sum['i'] += rcv_buf['i']; sum['f'] += rcv_buf['f'] # Step 4
# Step 5
print(f"PE{my_rank}:\tSum = {sum['i']}\t{sum['f']}")
```

Provided in
the skeleton



Chapter 12-(2), Exercises 5+6: Resizing of derived types (major changes)

C

MPI/tasks/C/Ch12/solutions/derived-struct-double+int-resized.c

```
MPI_Datatype ... send_recv_type, send_recv_resized;  
-----  
MPI_Type_create_struct(COUNT, ..., &send_recv_type);  
MPI_Type_create_resized(send_recv_type,  
    (MPI_Aint) 0, (MPI_Aint) sizeof(snd_buf[0]), &send_recv_resized);  
MPI_Type_commit(&send_recv_resized);  
-----  
MPI_Issend(&snd_buf, arr_lng-1, send_recv_resized, ...)
```

Fortran

MPI/tasks/F_30/Ch12/solutions/derived-struct-dp+integer-resized_30.f90

MPI/tasks/F_30/Ch12/solutions/derived-struct-dp+integer-bindC-resized_30.f90

```
TYPE(MPI_Datatype) :: send_recv_type, send_recv_resized  
-----  
CALL MPI_Type_create_struct(2, ..., send_recv_type)  
CALL MPI_Get_address(snd_buf(1), first_var_address)  
CALL MPI_Get_address(snd_buf(2), second_var_address)  
lb = 0; extent = MPI_Aint_diff(second_var_address, first_var_address)  
CALL MPI_Type_create_resized(send_recv_type, lb, extent, send_recv_resized)  
CALL MPI_Type_commit(send_recv_resized)  
-----  
CALL MPI_Issend(snd_buf, arr_lng-1, send_recv_resized, ...)
```

Python

MPI/tasks/PY/Ch12/solutions/derived-struct-double+int-resized.py

```
np_dtype = np.dtype([('f', np.double), ('i', np.intc)], align=True)  
snd_buf = np.empty(arr_lng, dtype=np_dtype); rcv_buf=np.empty_like(snd_buf)  
-----  
send_recv_type = MPI.Datatype.Create_struct(array_of_blocklengths, ...)  
send_recv_resized = send_recv_type.Create_resized(0, snd_buf.itemsize)  
send_recv_resized.Commit()  
-----  
request = comm_world.Issend((snd_buf, arr_lng-1, send_recv_resized), ...)
```



Chapter 13-(1): Parallel file I/O exercise 1 – explicit file-pointer

C

```
MPI_Offset offset;  
...  
MPI_File_open(MPI_COMM_WORLD, "my_test_file",  
               MPI_MODE_RDWR | MPI_MODE_CREATE,  
               MPI_INFO_NULL, &fh);  
  
for (i=0; i<10; i++) {  
    buf = '0' + (char)my_rank;  
    offset = my_rank + size*i;  
    MPI_File_write_at(fh, offset, &buf, 1, MPI_CHAR, &status);  
}
```

MPI/tasks/C/Ch13/solutions/mpi_io_exa1.c

or better for performance: MPI_MODE_WRONLY

Fortran

```
INTEGER (KIND=MPI_OFFSET_KIND) offset  
...  
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'my test file',  
                   IOR(MPI_MODE_RDWR, MPI_MODE_CREATE),  
                   MPI_INFO_NULL, fh, ierror)  
  
DO I=1,10  
    buf = CHAR( ICHAR('0') + my_rank )  
    offset = my_rank + size*(i-1)  
    CALL MPI_FILE_WRITE_AT(fh, offset, buf, 1, MPI_CHARACTER,  
                           status, ierror)  
END DO
```

or better MPI_MODE_WRONLY

Python

```
fh = MPI.File.Open(comm_world, "my_test_file", MPI.MODE_RDWR | MPI.MODE_CREATE, MPI.INFO_NULL)  
for i in range(10):  
    buf = np.array(ord('0')+my_rank, dtype=np.byte) # numpy.byte should be of integer type and  
    offset = my_rank + size*i; fh.Write_at(offset, (buf, 1, MPI.CHAR), status) # compatible  
                                                # with C char.
```

Chapter 13-(2): Parallel file I/O exercise 2 – with fileview

C

Python

```
MPI_Offset disp;
...
ndims = 1;
array_of_sizes[0] = size;
array_of_subsizes[0] = 1;
array_of_starts[0] = my_rank;
...
MPI_Type_create_subarray(...);
MPI_Type_commit(&filetype);
MPI_File_open(..., MPI_MODE_RDWR | MPI_MODE_CREATE, ...);
disp = 0;
MPI_File_set_view(...);
for(i=0; i<3; i++) {
    buf = 'a' + (char)my_rank;
    MPI_File_write(fh, &buf, 1, etype, &status);
}
```

Python: `filetype.Commit()`

or better for performance: `MPI_MODE_WRONLY`

or `MPI_CHAR`

Fortran

```
INTEGER (KIND=MPI_OFFSET_KIND) disp
...
ndims = 1
array_of_sizes(1) = size
array_of_subsizes(1) = 1
array_of_starts(1) = my_rank
...
CALL MPI_TYPE_CREATE_SUBARRAY(...)
CALL MPI_TYPE_COMMIT(filetype, ierror)
CALL MPI_FILE_OPEN( ..., IOR(MPI_MODE_RDWR, MPI_MODE_CREATE), ...)
disp = 0
CALL MPI_FILE_SET_VIEW(...)
DO I=1,3
    buf = CHAR( ICHAR('a') + my_rank )
    CALL MPI_FILE_WRITE(fh, buf, 1, etype, status, ierror)
END DO
```

or better `MPI_MODE_WRONLY`

or `MPI_CHARACTER`

Chapter 13-(3): Parallel file I/O exercise 3 – shared filepointer

C

Python

```
    MPI_Datatype etype;
    MPI_Datatype filetype;
    MPI_Offset disp;
    etype = MPI_CHAR;
    ndims = 1;
    array_of_sizes[0] = size;
    array_of_subsizes[0] = 1;
    array_of_starts[0] = my_rank;
    order = MPI_ORDER_C;
    MPI_Type_create_subarray(ndims, array_of_sizes,
                           array_of_subsizes, array_of_starts, order, etype, &filetype);
    MPI_Type_commit(&filetype);
    disp = 0;
    MPI_File_set_view(fh, disp, etype, filetype, "native", MPI_INFO_NULL);
    MPI_File_open(MPI_COMM_WORLD, "my_test_file",
                  MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh);
    for (i=0; i<3; i++) {
        buf = 'a' + (char)my_rank;
        MPI_File_write_ordered(fh, &buf, 1, MPI_CHAR, &status);
    }
    MPI_File_close(&fh);
```

Python `fh.write_ordered((buf, 1, MPI.CHAR), status)`

Fortran

```
TYPE(MPI_Datatype) :: etype
                &
                &
    CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'my_test_file', &
                      &IOR(MPI_MODE_RDWR, MPI_MODE_CREATE), &
                      &MPI_INFO_NULL, fh, ierror)
    DO I=1,3
        buf = CHAR( ICHAR('a') + my_rank )
        CALL MPI_FILE_WRITE_ORDERED(fh, buf, 1, MPI_CHARACTER, status, ierror)
    END DO
    CALL MPI_FILE_CLOSE(fh, ierror)
```

MPI/tasks/C/Ch13/solutions/mpi_io_exa3.c

MPI/tasks/F_30/Ch13/solutions/mpi_io_exa3.f



For private notes

For private notes

For private notes

For private notes