Introduction to the Message Passing Interface (MPI)

Rolf Rabenseifner
rabenseifner@hlrs.de

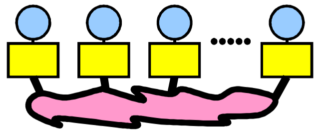
University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

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

Höchstleistungsrechenzentrum Stuttgart

Outline

1. MPI Overview
   – **one program on several processors**
   – **work and data distribution**



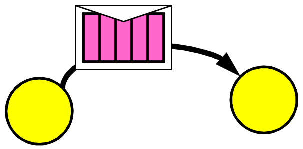
[2.4, 2.7]
slides 9–…

2. Process model and language bindings
   – **starting several MPI processes**

MPI_Init()
MPI_Comm_rank()

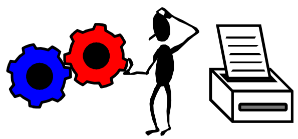
[2.6, 17.1, 6.4.1, 8.7-8]
slides 37–…

3. Messages and point-to-point communication
   – **the MPI processes can communicate**



[3.1-3.6, 8.6]
slides 57–…

4. Nonblocking communication
   – **to avoid idle time and deadlocks**



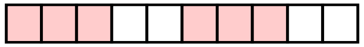
[3.7, 3.10]
slides 81–…

[…] = **MPI-3.0 chapter**

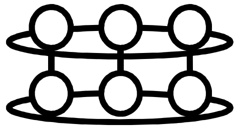
Outline

5. Probe, Persistent Requests, Cancel — [3.8, 3.9] slides105–…

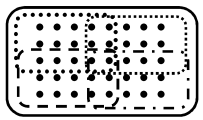
6. Derived datatypes — [4, 3.10, 2.5.8] slides 113–…
   – **transfer any combination of typed data**



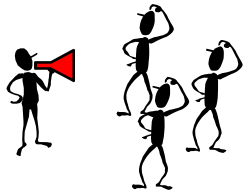
7. Virtual topologies — [7, 3.11] slides 141–…
   – **a multi-dimensional process naming scheme**



8. Groups & Communicators, Environment Management — [6.1-7, 8.1+3-5, 9] slides 161–…
   – **MPI_Comm_split, caching, implementation-information, naming, info, error handling**



9. Collective communication — [5] slides 177–…
   – **e.g., broadcast**
   – **neighborhood communication**



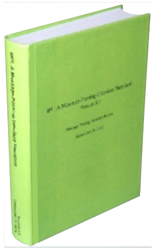
[…] = **MPI-3.0 chapter**

Outline

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)

Information about MPI



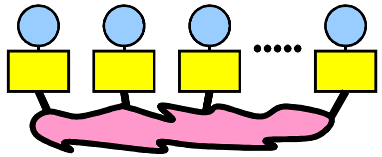
- **MPI: A Message-Passing Interface Standard,** Version 3.0 (September 21, 2012)
(printed hardcover book & online via www.mpi-forum.org)
- Marc Snir and William Gropp et al.:
MPI: The Complete Reference. (2-volume set).The MIT Press, 1998.
(outdated due to new MPI-2.1, MPI-2.2, and MPI-3.0)
- William Gropp, Ewing Lusk and Rajeev Thakur:
Using MPI: Portable Parallel Programming With the Message-Passing Interface.
MIT Press, Nov. 1999, And
Using MPI-2: Advanced Features of the Message-Passing Interface.
MIT Press, Aug. 1999 *(or both in one volume, 725 pages, ISBN 026257134X).*
- Peter S. Pacheco: **Parallel Programming with MPI**. Morgen 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**. Handbook from EPCC.
http://www2.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.pdf
- All MPI standard documents and errata via **www.mpi-forum.org**
- http://en.wikipedia.org/wiki/Message_Passing_Interface (English)
http://de.wikipedia.org/wiki/Message_Passing_Interface (German)

For private notes

For private notes

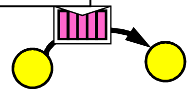
Chap.1 MPI Overview

1. MPI Overview

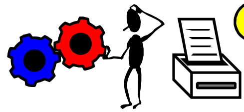


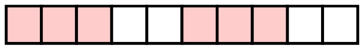
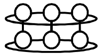
- – **one program on several processors**
- – **work and data distribution**
- – **the communication network**

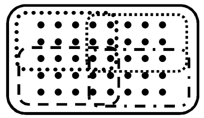
2. Process model and language bindings **MPI_Init()** **MPI_Comm_rank()**



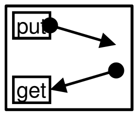
3. Messages and point-to-point communication



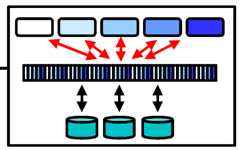
4. Nonblocking communication
5. Probe, Persistent Requests, Cancel
6. Derived datatypes 
7. Virtual topologies 
8. Groups & communicators, environment management



9. Collective communication
10. Process creation and management
11. One-sided communication



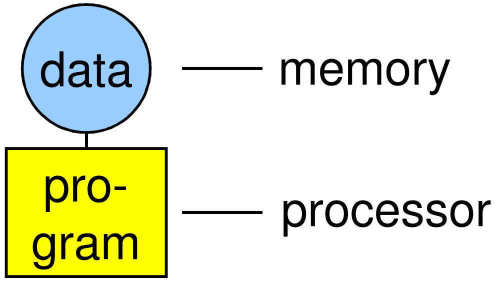
12. Shared memory one-sided communication
13. MPI and threads
14. Parallel file I/O



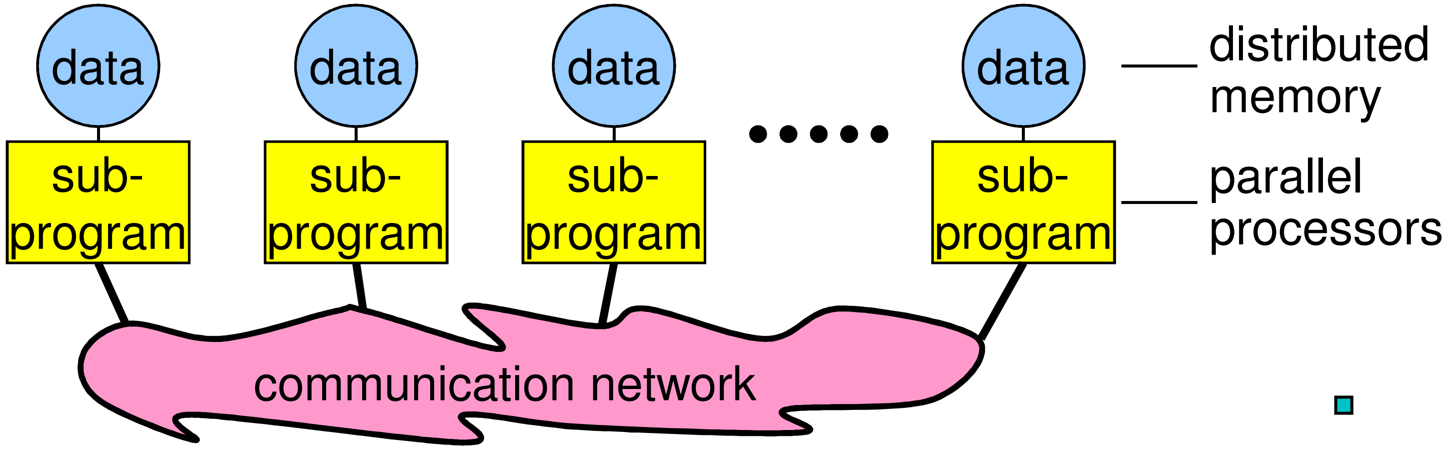
15. Other MPI features

The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**

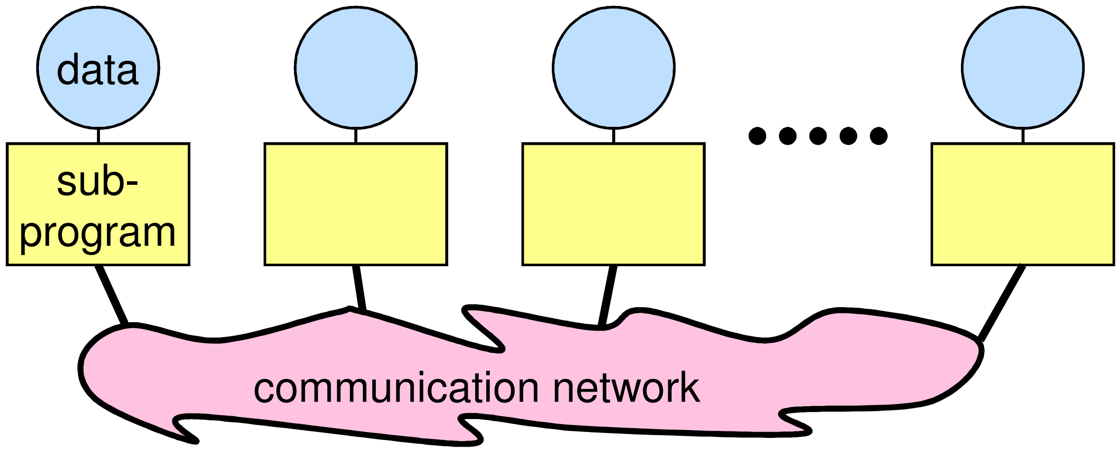


- **Message-Passing Programming Paradigm**



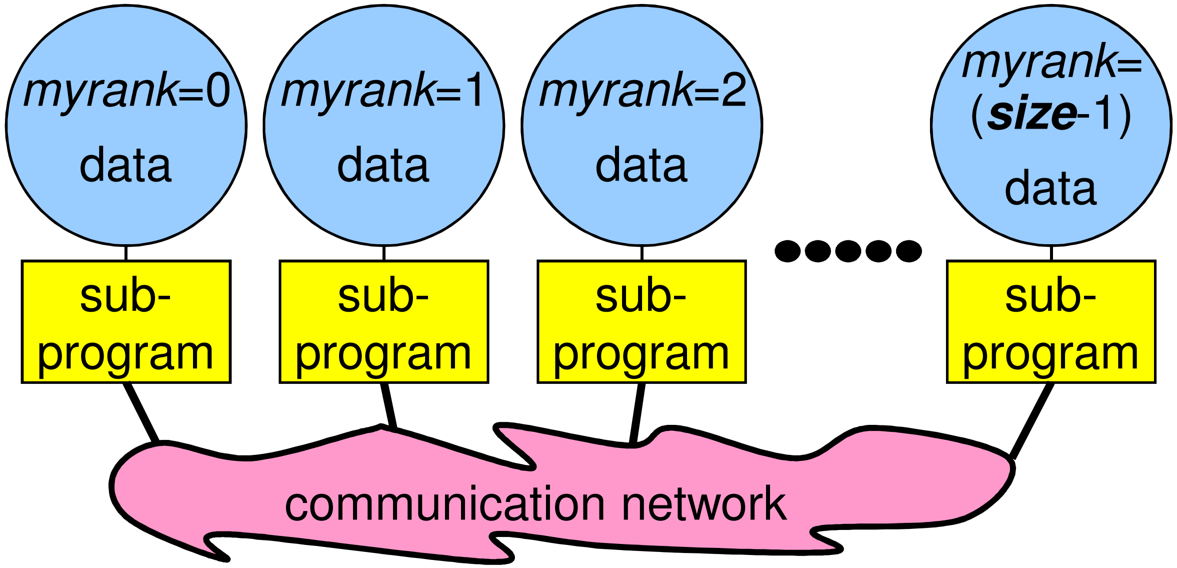
The Message-Passing Programming Paradigm

- Each processor in a message passing program runs a ***sub-program:***
  - written in a conventional sequential language, e.g., C or Fortran,
  - 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

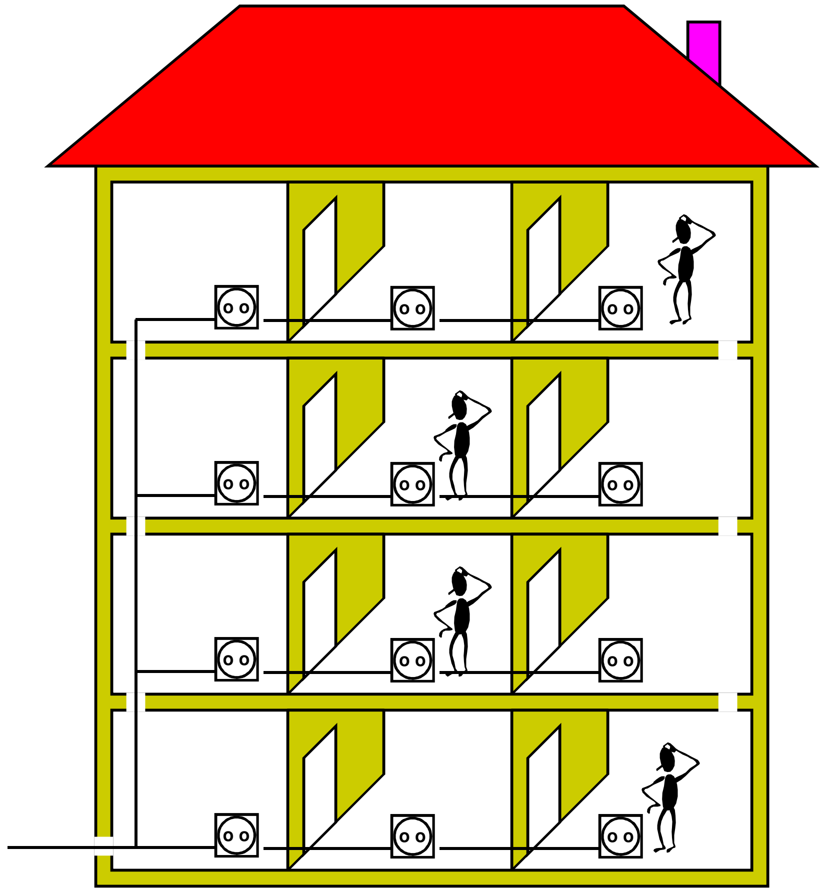


Analogy: Electric Installations in Parallel

- 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



What is SPMD?

- **S**ingle **P**rogram, **M**ultiple **D**ata
- Same (sub-)program runs on each processor

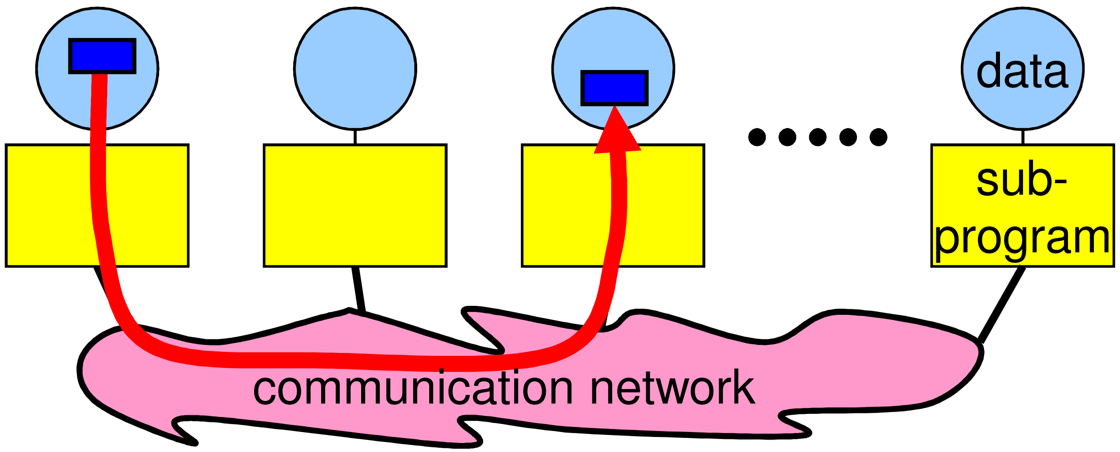
- MPI allows also MPMD, i.e., **Multiple** Program, ...
- but some vendors may be restricted to SPMD
- MPMD can be emulated with SPMD

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
  ```

Messages



- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:

– sending process	– receiving process	i.e., the ranks
– source location	– destination location	
– source data type	– destination data type	
– source data size	– destination buffer size	

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

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)

Reception

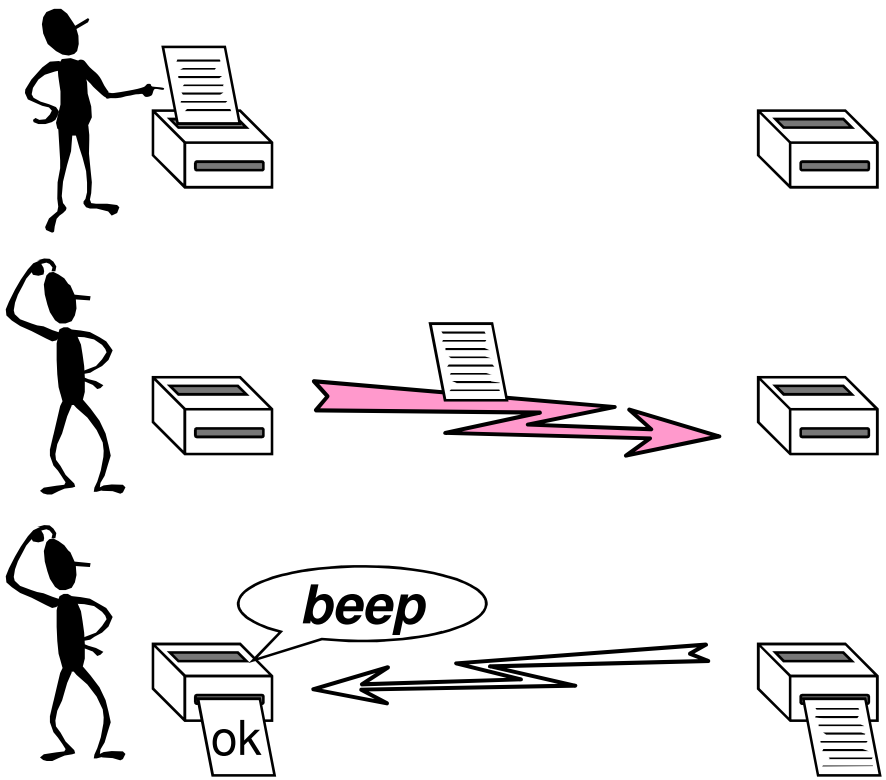
- All messages must be received.

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

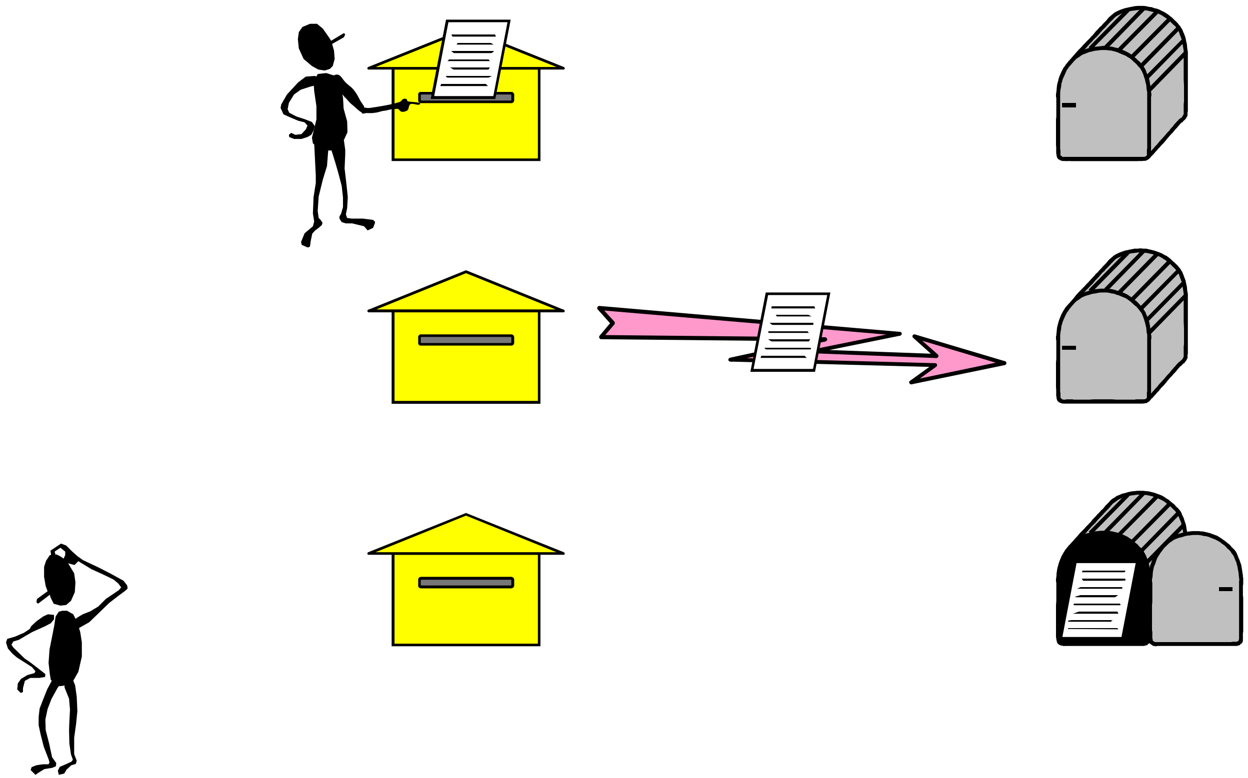
Synchronous Sends

- 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.

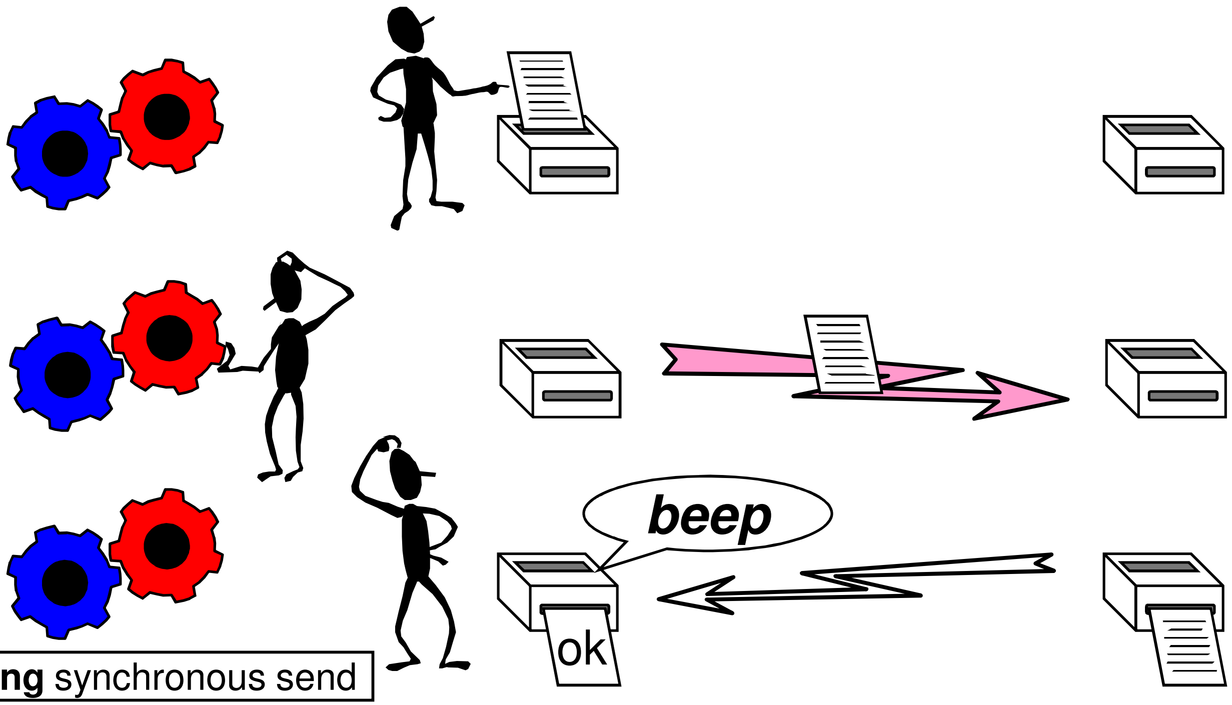


Blocking Operations

- Operations are local activities, e.g.,
  - 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.

Non-Blocking Operations

- Nonblocking operation: returns immediately and allow 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.



nonblocking synchronous send

Non-Blocking Operations (cont‘d)

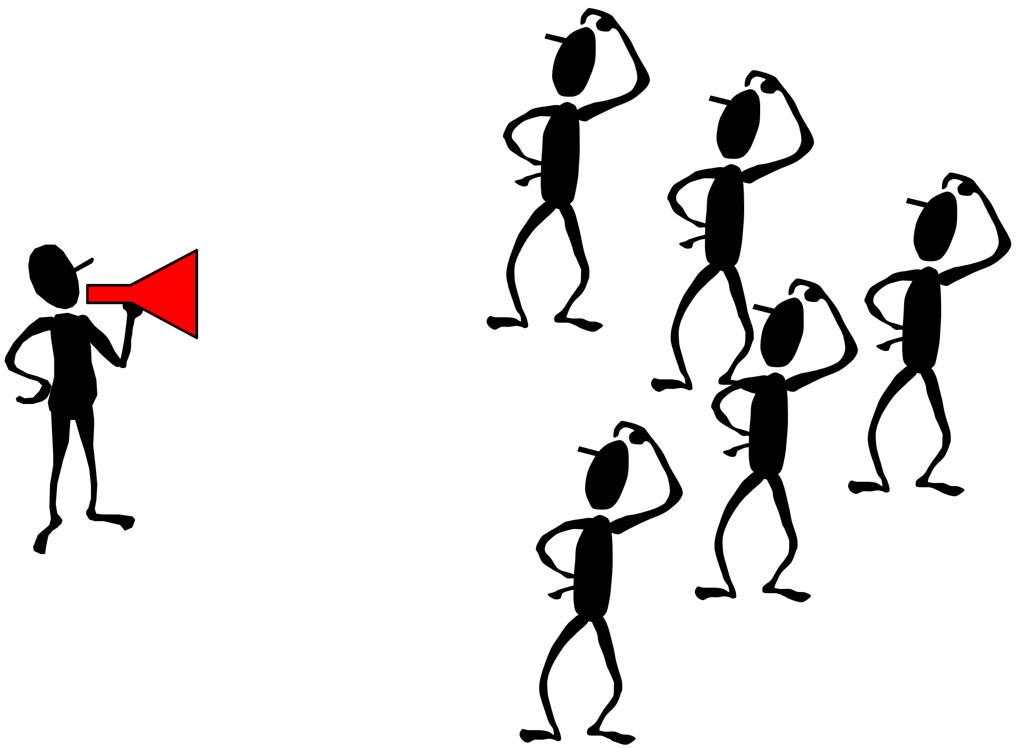
- All nonblocking operations must have matching wait (or test) operations. (Some system or application resources can be freed only when the nonblocking operation is completed.)
- A nonblocking operation immediately followed by a matching wait is equivalent to a blocking operation.
- Nonblocking operations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

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.

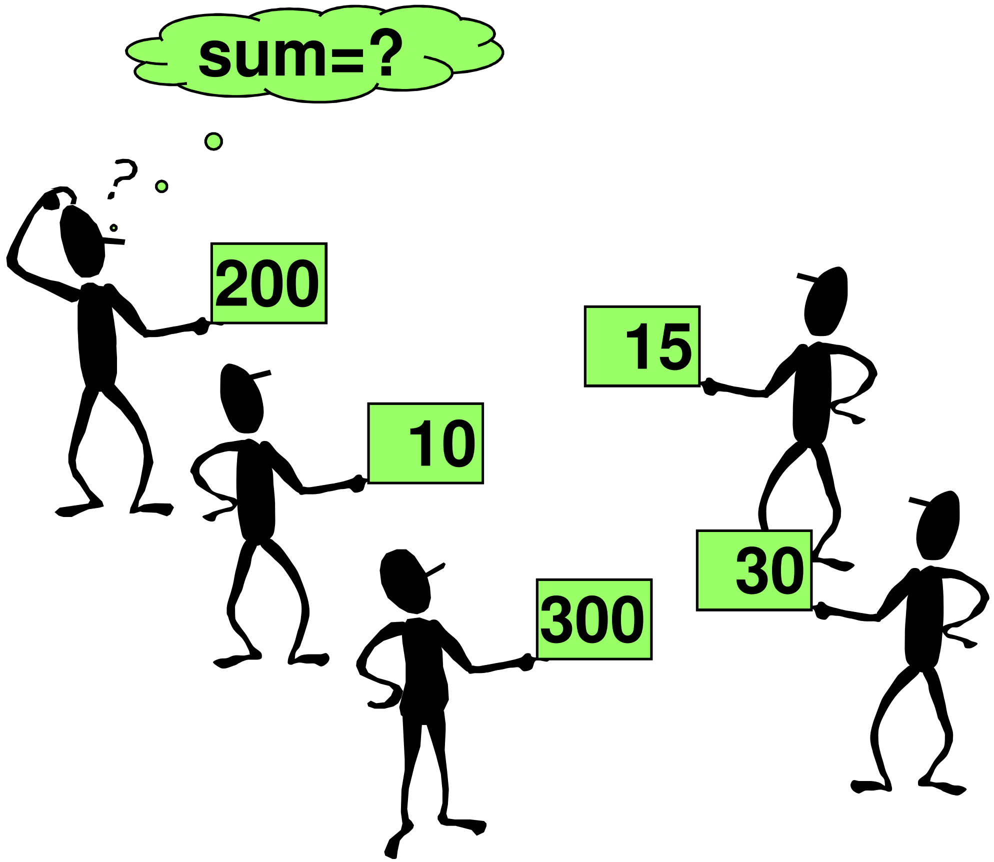
Broadcast

- A one-to-many communication.



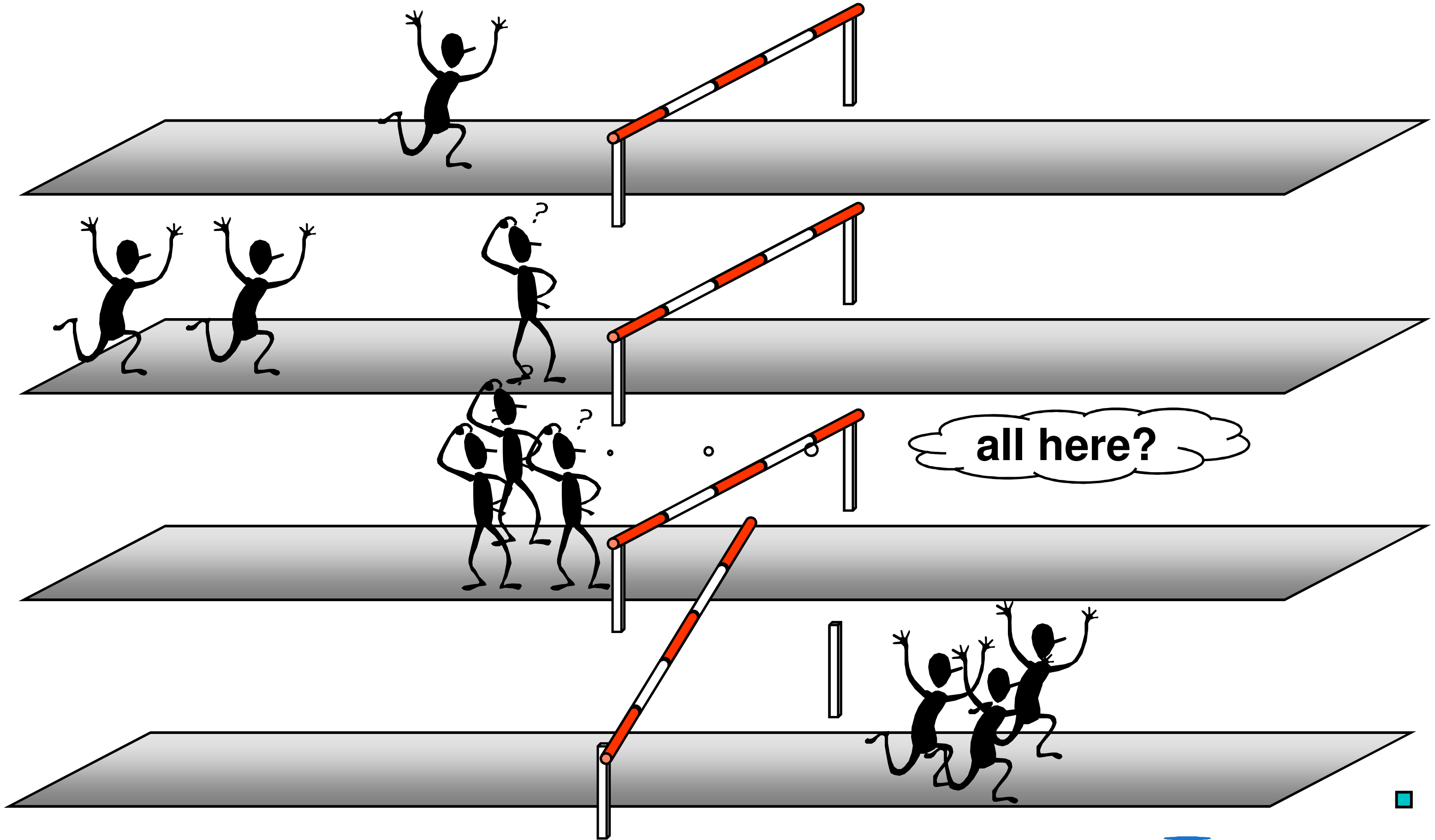
Reduction Operations

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



Barriers

- Synchronize processes.



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-2 and MPI-3 Forum

- **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**
  - 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

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:
  - 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 was developed for better **platform** and **application** support.
- MPI for HPC: Better support of clusters of SMP nodes

→ This is an MPI-3.0 course
  - includes most (performance) features of MPI

- Only overview-information for less important features of MPI

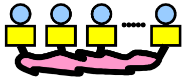
→ This course is for applications on systems ranging
  - from small cluster
  - to large HPC systems

For private notes

- private notes

For private notes

Chap.2 Process Model and Language Bindings

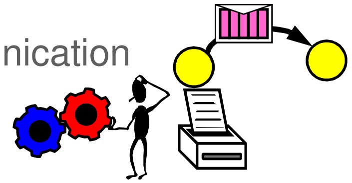
1. MPI Overview 

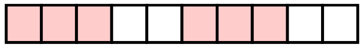
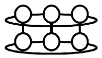
2. Process model and language bindings

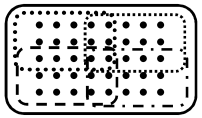
– **starting several MPI processes**

MPI_Init()
MPI_Comm_rank()

3. Messages and point-to-point communication
4. Nonblocking communication
5. Probe, Persistent Requests, Cancel

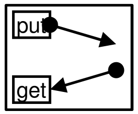


6. Derived datatypes 
7. Virtual topologies 
8. Groups & communicators, environment management

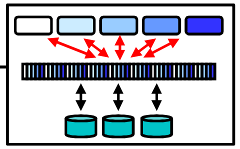


9. Collective communication
10. Process creation and management

11. One-sided communication
12. Shared memory one-sided communication



13. MPI and threads
14. Parallel file I/O
15. Other MPI features



Header files

C

- C / C++

  #include <mpi.h>

Fortran

- Fortran

  use mpi (or: include ´mpif.h´)

  or since MPI-3.0: use mpi_f08

Compile-time argument checking:
MPI-2.0 – 2.2: may be
MPI-3.0: mandatory

Normally without any compile-time argument checking

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

MPI Function Format

C

Fortran

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

- Fortran: CALL MPI_XXXXXX(parameter, ..., *ierror*)

With mpi module or mpif.h:
absolutely never forget!

MPI-3.0, with mpi_f08 module:
ierror is optional

New in MPI-3.0

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

- With the new module

  USE mpi_f08

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

  CALL MPI_SEND(…., MPI_COMM_WORLD)

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

MPI Function Format Details

C

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

New in MPI-3.0

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);	

- Several index sections at the end: 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_ .

Initializing MPI

C

- C/C++: int MPI_Init(int *argc, char ***argv)

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

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

Fortran

- Fortran: MPI_INIT(IERROR)
  INTEGER IERROR

```
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)
....
```

Alternative with MPI-3.0:

use mpi_f08

- Must be first MPI routine that is called
  (only a few exceptions, e.g., MPI_Initialized)

The Fortran support methods

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

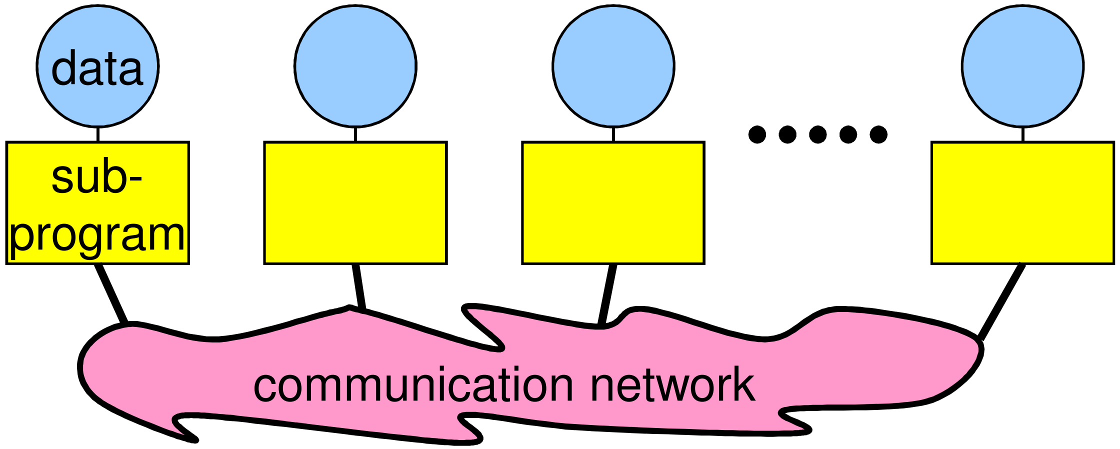
Today (MPI-3) — Maybe in the future (MPI-next, MPI-…, far future)

Level of Quality:

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

Starting the MPI Program

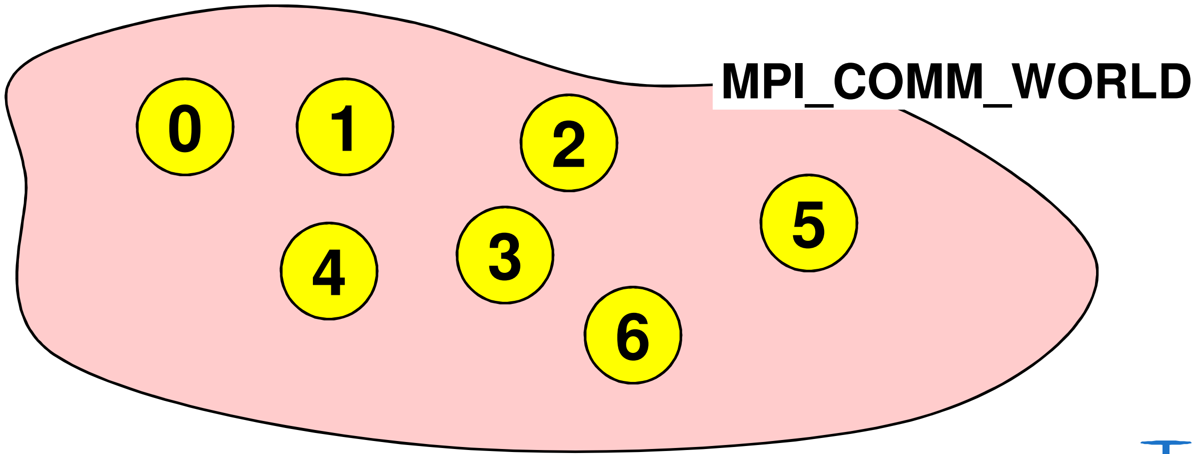
- Start mechanism is implementation dependent
- mpirun –np ***number_of_processes*** ./***executable*** (most implementations)
- mpiexec –n ***number_of_processes*** ./***executable*** (with MPI-2 and later)



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

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

- Handles identify MPI objects.
- For the programmer, handles are
  - predefined constants in mpi.h or mpif.h
    - Example: MPI_COMM_WORLD
    - Can be used in initialization expressions or assignments.
    - The object accessed by the predefined constant handle exists and does not change only between **MPI_Init** and **MPI_Finalize.**
  - values returned by some MPI routines, to be stored in variables, that are defined as
    - in Fortran: **New in MPI-3.0**
      - mpi_f08 module: TYPE(MPI_Comm), etc.
      - mpi module and mpif.h: INTEGER
    - in C: special MPI typedefs, e.g., MPI_Comm
- Handles refer to internal MPI data structures

Fortran

C

Rank

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

C

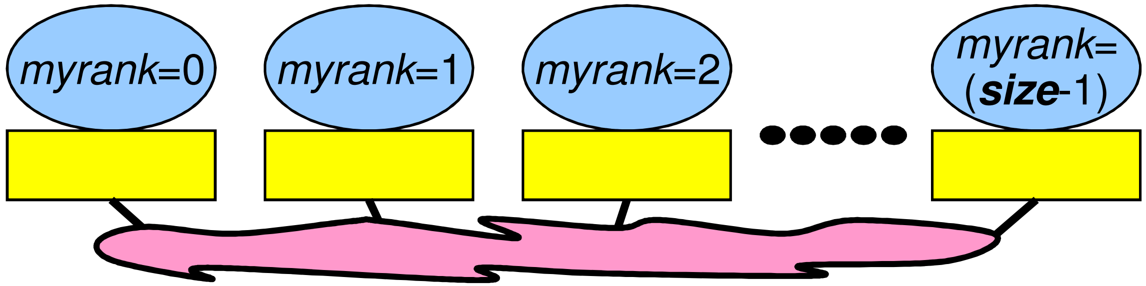
Fortran

- 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



CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierror)

Size

- How many processes are contained within a communicator?

C

Fortran

- 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
```

Exiting MPI

C

Fortran

- C/C++: int MPI_Finalize()
- Fortran: MPI_FINALIZE(*ierror*)

  mpi_f08: INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: INTEGER ierror

- **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 all processes except “rank==0” in MPI_COMM_WORLD

Compilation and Parallel Start

• Your working directory:	~/MPI/<u>#nr</u> *with <u>#nr</u> = number of your PC*	
• Initialization:	module use mpi	(or other setup)
• Compilation in C:	**mpicc –o** <u>*prg*</u> <u>*prg*</u>**.c**	(usual)
	cc –o <u>*prg*</u> <u>*prg*</u>**.c**	(on Cray)
	cc –o <u>*prg*</u> <u>*prg*</u>**.c** **–lmpi**	(on …)
	mpcc_r –o <u>*prg*</u> <u>*prg*</u>**.c**	(on IBM)
• Compilation in Fortran:	**mpif90 –o** <u>*prg*</u> <u>*prg*</u>**.f**	(usual)
	ftn –o <u>prg</u> <u>prg</u>**.f**	(on Cray)
	f90 –o <u>prg</u> <u>prg</u>**.f** **–lmpi**	(on …)
	mpxlf_r –o <u>prg</u> <u>prg</u>**.f**	(on IBM)
• Program start on <u>*num*</u> PEs:	**mpirun –np** <u>*num*</u> **./**<u>*prg*</u>	(all, except ...:)
	mpiexec –n <u>*num*</u> **./**<u>*prg*</u>	(standard MPI-2)
	poe –procs <u>*num*</u> **./**<u>*prg*</u>	(IBM)
• C examples	~/MPI/course/**C**/Ch[2-14]/*.c	
• Fortran examples	~/MPI/course/**F_[123]***/Ch[2-14]/*.f*	

C

Fortran

F_11 (*.f)
- MPI-1.1
- with mpif.h

F_20 (*.f90)
- MPI-2.x
- mpi module

F_30 (*.f90)
- MPI-3.0
- mpi_f08

- 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.

Exercise: Hello World

- Write a minimal MPI program which prints „hello world" by each MPI process.

hint for C: #include <stdio.h>

C

- Compile and run it on a single processor.
- Run it on several processors in parallel.
- Modify your 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
```

see also login-slides

Sol.

Advanced Exercises: Hello World with deterministic output

- Discuss with your neighbor, 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?

Sol.

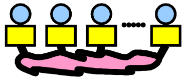
For private notes

For private notes

For private notes

For private notes

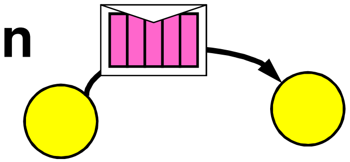
Chap.3 Messages and Point-to-Point Communication

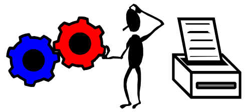
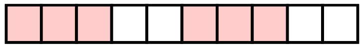
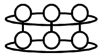
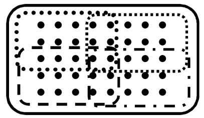
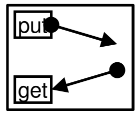
1. MPI Overview 
2. Process model and language bindings

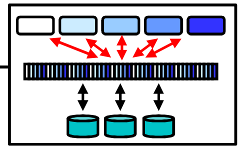
```
MPI_Init()
MPI_Comm_rank()
```

3. Messages and point-to-point communication

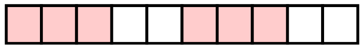
– **the MPI processes can communicate**



4. Nonblocking communication 
5. Probe, Persistent Requests, Cancel
6. Derived datatypes 
7. Virtual topologies 
8. Groups & communicators, environment management 
9. Collective communication 
10. Process creation and management
11. One-sided communication
12. Shared memory one-sided communication 

13. MPI and threads
14. Parallel file I/O 
15. Other MPI features

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**

C

MPI Basic Datatypes — C / C++

MPI Datatype	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-3.0, Annex A.1

Includes also special C++ types, e.g., bool, see page 666

Fortran

MPI Basic Datatypes — Fortran

MPI Datatype	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-3.0, Annex A.1

2345	654	96574	-12	7676

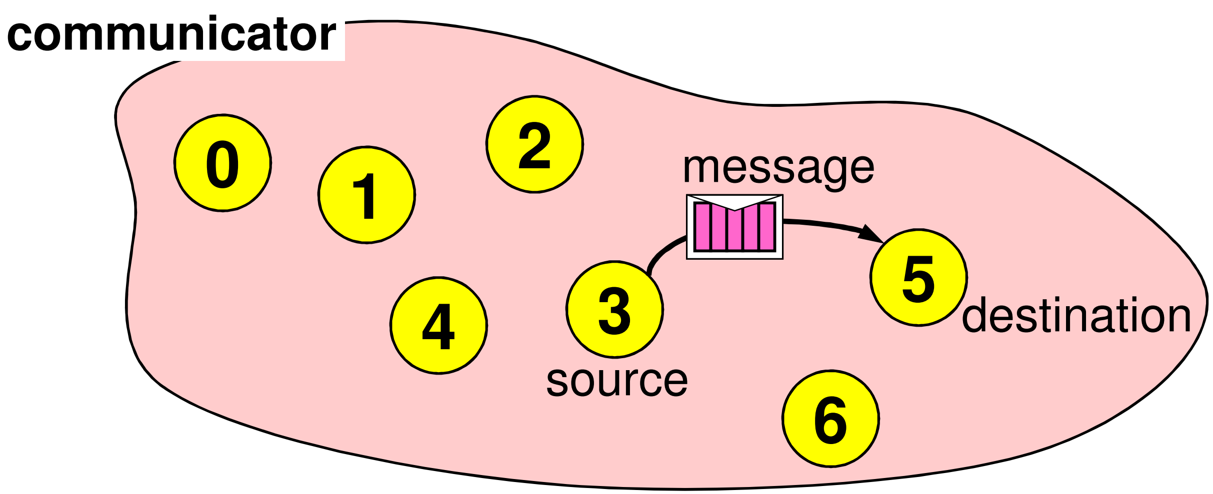
count=5
datatype=MPI_INTEGER

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

Fortran

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

- 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.

H L R S

Receiving a Message

C

- C/C++: int MPI_Recv(void **buf*, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
  MPI_Status **status*)

Fortran

- 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)

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in *status*.
- On can pass MPI_STATUS_IGNORE instead of a status argument.
- Output arguments are printed *blue-cursive*.
- Only messages with matching tag are received.

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.
- Tags must match.
- Buffer’s (C or Fortran) type must match with the datatype handle (in the send and receive call)
- Message datatypes must match.
- Receiver’s buffer must be large enough.

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.

Communication Envelope

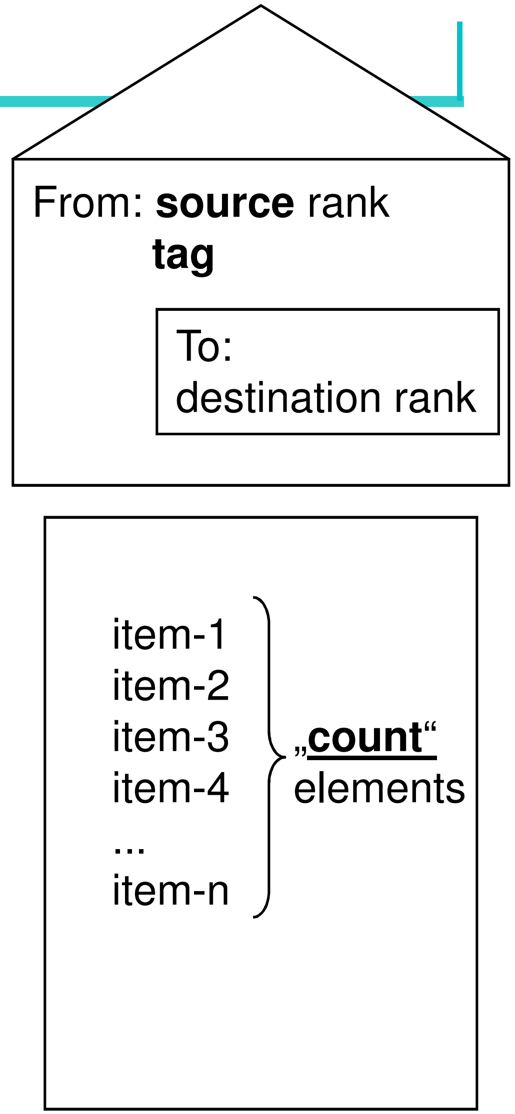
C

Fortran

- Envelope information is returned from MPI_RECV in *status*.
- C/C++: MPI_Status status;
  status.MPI_SOURCE
  status.MPI_TAG
  status.MPI_ERROR *)
  count via MPI_Get_count()
- 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) *)

special indexes

- count via MPI_GET_COUNT()



*) See slide 76 on MPI_Waitall, …

Receive Message Count

C

Fortran

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

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

mpi_f08:

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

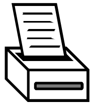
mpi & mpif.h: INTEGER status(MPI_STATUS_SIZE), datatype, count, ierror

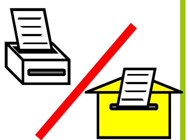
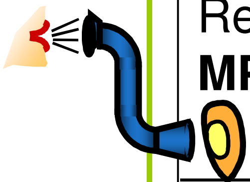
Communication Modes

- Send communication modes:
  - synchronous send → MPI_**S**SEND
  - buffered [asynchronous] send → MPI_**B**SEND
  - standard send → MPI_**SEND**
  - Ready send → MPI_**R**SEND

- Receiving all modes → MPI_**RECV**

Communication Modes — Definitions



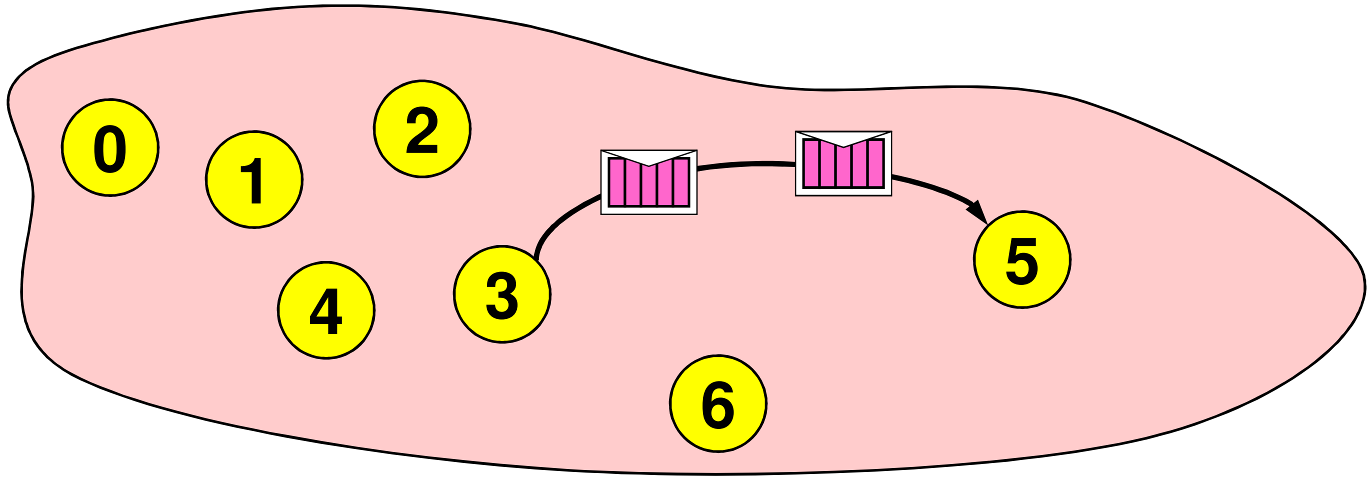
Sender mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Standard send **MPI_SEND**	Either 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
  - —> 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

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.

New in MPI-3.0

Fortran

Routine declarations within the mpi_f08 module

Mainly for implementer's reasons.
Not relevant for users.

Removed, see MPI-3.0 errata Sep. 24, 2013 and later

MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) ~~BIND(C)~~

- TYPE(*), DIMENSION(..), ASYNCHRONOUS [1] :: buf
- INTEGER, INTENT(IN) :: count, source, tag
- TYPE(MPI_Datatype), INTENT(IN) :: datatype
- TYPE(MPI_Comm), INTENT(IN) :: comm
- TYPE(MPI_Status) :: status
- 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

OPTIONAL ierror:
MPI routine can be called without ierror argument

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

[1] ASYNCHRONOUS: only in nonblocking routines, not in MPI_Recv

Fortran

New in MPI-3.0

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-3.0, Section 17.2.5 pages 648-650

Fortran

New in MPI-3.0

Handles in mpi_f08

- Unique handle types, e.g.,
  - INTEGER comm → **TYPE(MPI_Comm) :: comm**

Same names as in C

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

- Handle comparisons, e.g.,
  - comm .EQ. MPI_COMM_NULL → comm .EQ. MPI_COMM_NULL

No change through overloaded operator

- Conversion in mixed applications:
  - Both modules (mpi & mpi_f08) contain the declarations for all handles.

```
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
```

New in MPI-3.0

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.

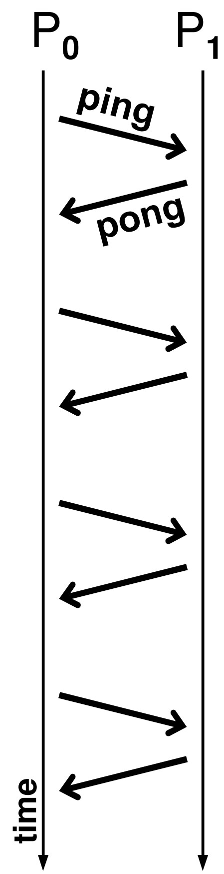
Remark: Some keywords are changed since MPI-2.2

Do not use outdated documents!

- **For consistency reasons, or**
- **To prohibit conflicts with Fortran keywords, e.g., *type*, *function*.**

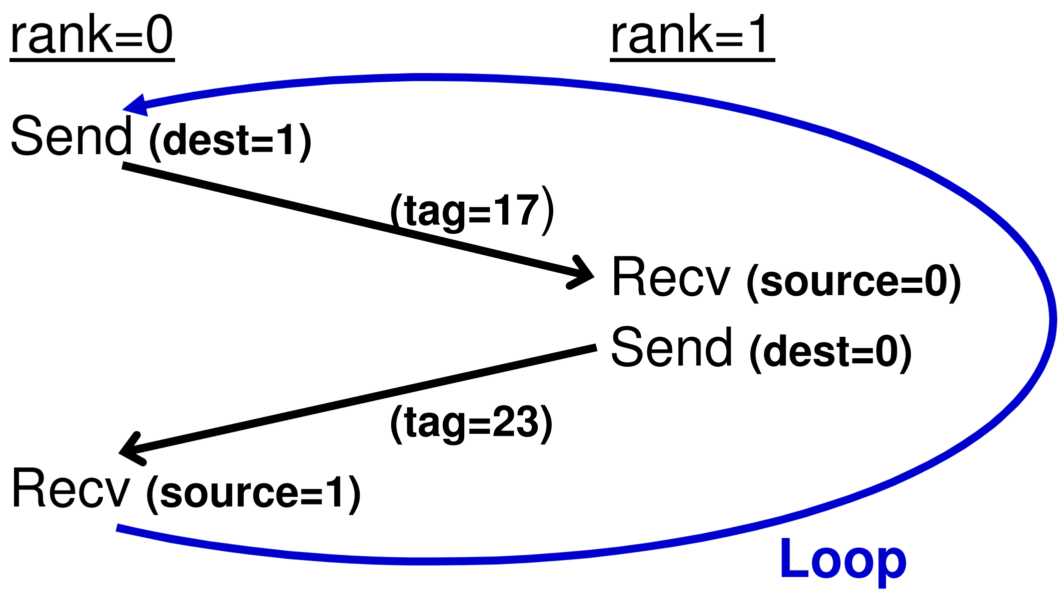
Exercise — Ping pong

- 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*()
- 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., delta_time / (2*50) * 1e6



[1)] One of the rare routines that can be implemented as macros in C, see MPI-3.0, Sect.2.6.4, page 19

Exercise — Ping pong

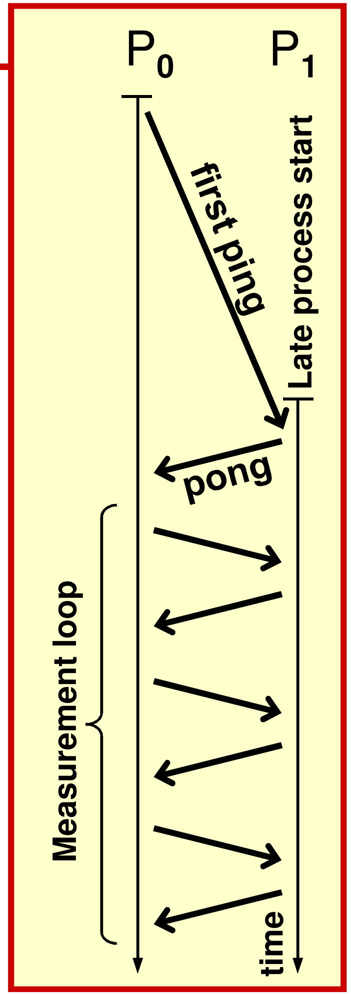



```
if (my_rank==0)              /* i.e., emulated multiple program */
     MPI_Send( ... dest=1 ...)
     MPI_Recv( ... source=1 ...)
else
     MPI_Recv( ... source=0 ...)
     MPI_Send( ... dest=0 ...)
fi
```

see also login-slides

Advanced Exercises — Ping pong latency and bandwidth

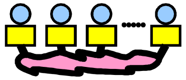
- Exclude startup time problems from measurements:
  - Execute a first ping-pong outside of the measurement loop
- latency = transfer time for short messages
- bandwidth = message size (in bytes) / transfer time
- Print out message transfer time and bandwidth
  - for following send modes:
    - **for 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)**



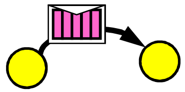
For private notes

For private notes

Chap.4 Nonblocking Communication

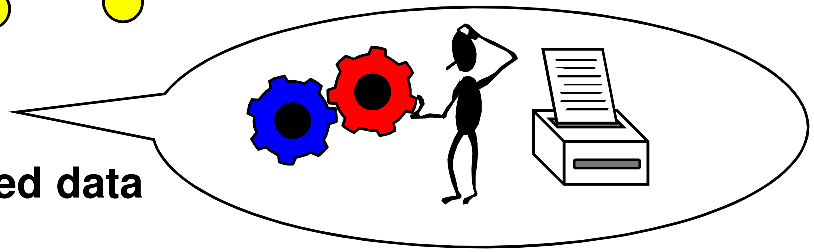
1. MPI Overview 

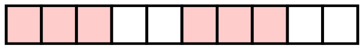
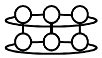
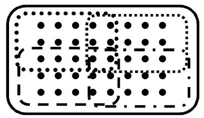
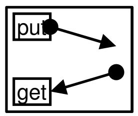
MPI_Init()
MPI_Comm_rank()

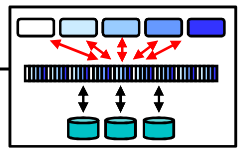
2. Process model and language bindings
3. Messages and point-to-point communication 

4. Nonblocking communication

- **transfer of any combination of typed data**



5. Probe, Persistent Requests, Cancel
6. Derived datatypes 
7. Virtual topologies 
8. Groups & communicators, environment management 
9. Collective communication 
10. Process creation and management
11. One-sided communication
12. Shared memory one-sided communication 

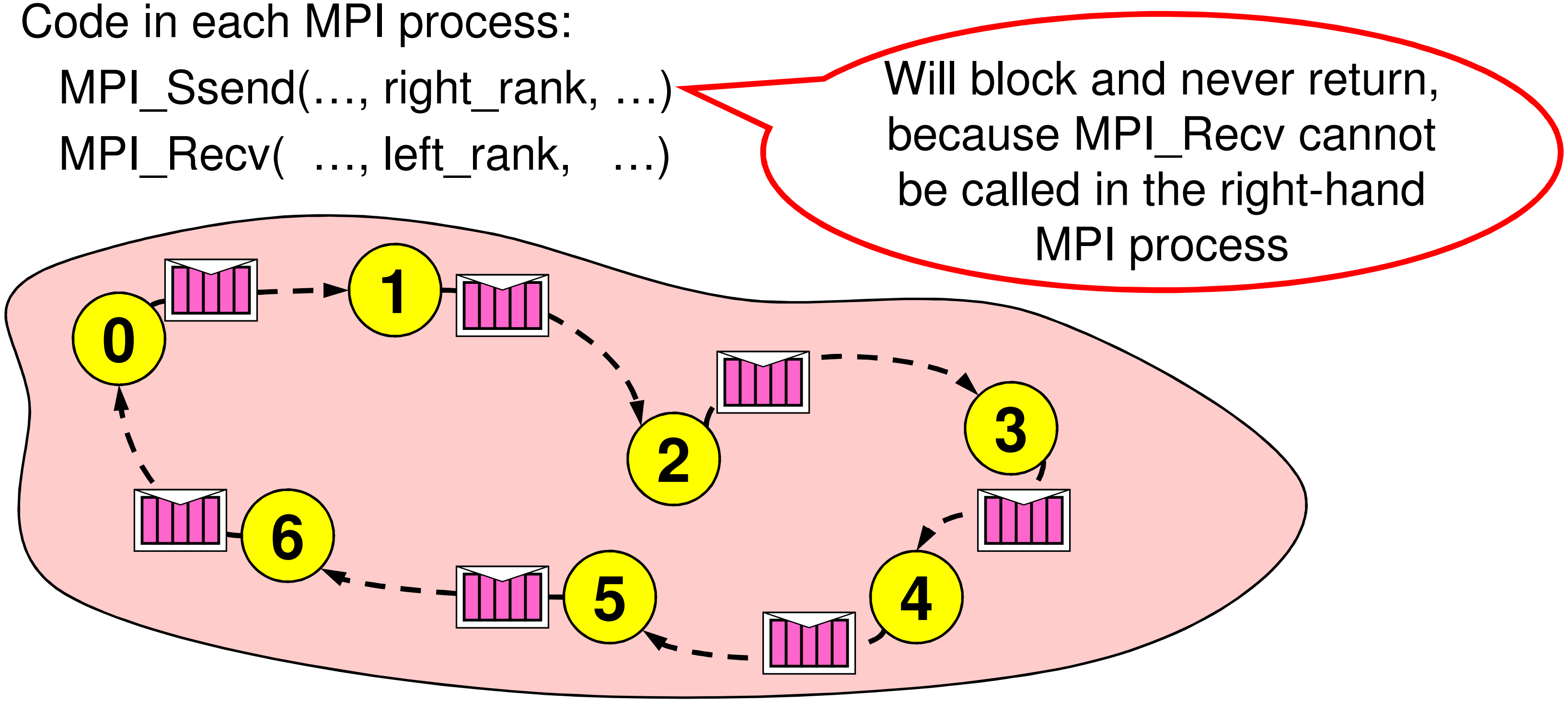
13. MPI and threads
14. Parallel file I/O 
15. Other MPI features

Deadlock

- Code in each MPI process:

  MPI_Ssend(…, right_rank, …)

  MPI_Recv(…, left_rank, …)



- Same problem with standard send mode (MPI_Send), if MPI implementation chooses synchronous protocol

Non-Blocking Communications

- Separate communication into three phases:
- Initiate nonblocking communication
  - returns **I**mmediately
  - routine name starting with MPI_**I**…
- Do some work (perhaps involving other communications?)
- Wait for nonblocking communication to complete

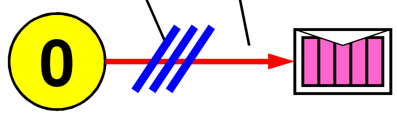
Non-Blocking Examples

- Nonblocking **send**

MPI_Isend(...)

doing some other work

MPI_Wait(...)

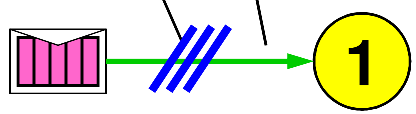


- Nonblocking **receive**

MPI_Irecv(...)

doing some other work

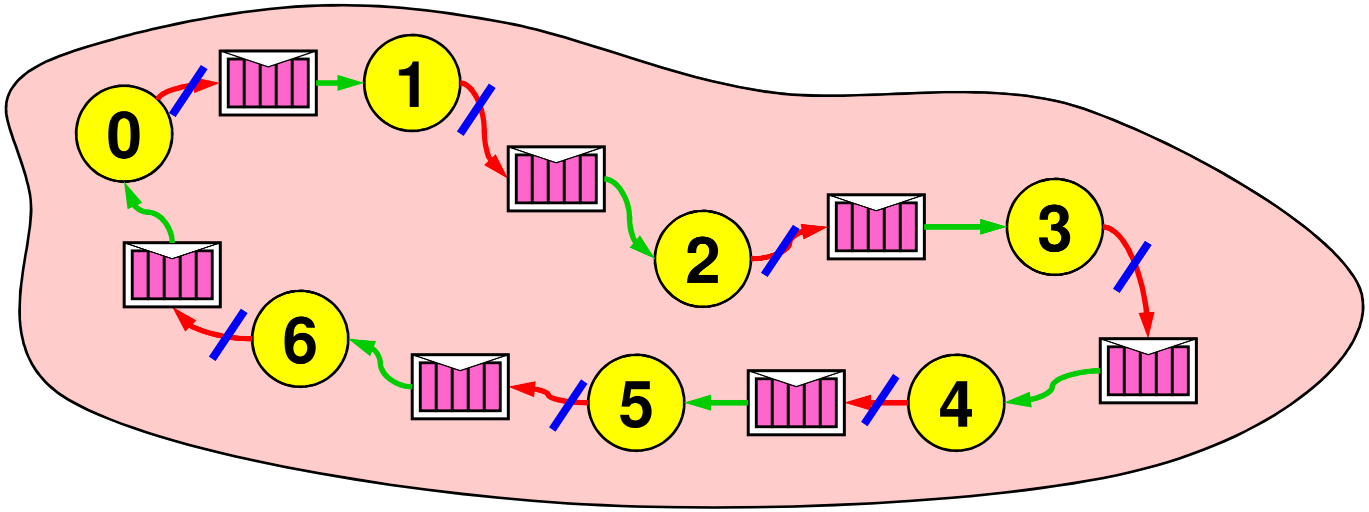
MPI_Wait(...)



/// = waiting until operation locally completed

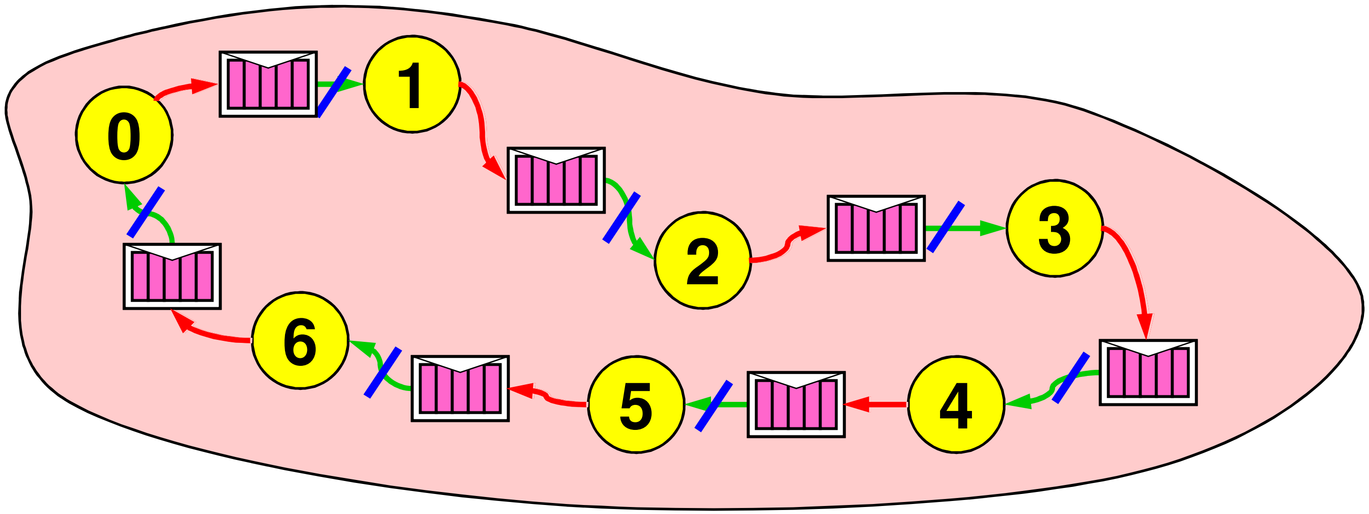
Non-Blocking Send

- 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



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, …

C Fortran

- Handles **can** also be stored in local variables
  - memory for datatype handles
    - in C/C++: MPI_Datatype
    - in Fortran:
      - mpi_f08: TYPE(MPI_Datatype)
      - mpi & mpif.h: INTEGER

C Fortran

  - memory for communicator handles
    - in C/C++: MPI_Comm
    - in Fortran:
      - mpi_f08: TYPE(MPI_Comm)
      - mpi & mpif.h: INTEGER

Request Handles

C

Fortran

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

the value

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

Nonblocking Synchronous Send

C

Fortran

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

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

- Fortran: ....., ASYNCHRONOUS :: buf **New in MPI-3.0**
  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) **New in MPI-3.0**
  & CALL MPI_F_SYNC_REG(buf)

- 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)
- status is not used in Issend, but in Wait (with send: nothing returned)

Fortran

- Fortran problems, see MPI-3.0, Chap. 17.1.2-17.1.19, pp 624-642, and next slides

Nonblocking Receive

C

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

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

Fortran

- Fortran: ....., ASYNCHRONOUS :: buf
  CALL MPI_IRECV (*buf*, count, datatype, source, 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

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

Fortran

- Message status is returned in Wait
- Fortran problems, see MPI-3.0, Chap. 17.1.2-17.1.19, pp 624-642, and next slides

Fortran

Nonblocking Receive and Register Optimization / Code Movement in Fortran

- **Fortran source code:**

  MPI_IRECV (*buf*, ..., *request_handle*, *ierror*)
  MPI_WAIT(request_handle, *status*, *ierror*)
  write (*,*) buf

  buf is not part of the argument list

- **may be compiled as**

  MPI_IRECV (*buf*, ..., *request_handle*, *ierror*)
  registerA = buf
  MPI_WAIT(request_handle, *status*, *ierror*)
  write (*,*) **registerA**

  Data may be received in buf during MPI_Wait

  Therefore old data may be printed instead of received data

- **Solution:**
  - ASCHRONOUS :: buf
  - *buf* may be allocated in a common block or module data, or
  - IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) &
    & CALL MPI_F_SYNC_REG(buf)

  In the scope including nonblocking call and MPI_Wait

  Directly after CALL MPI_Wait

- **Work-around in older MPI versions:**
  - call **MPI_GET_ADDRESS**(buf, *iaddrdummy*, *ierror*)
    with INTEGER(KIND=MPI_ADDRESS_KIND) iaddrdummy

  If MPI_F_SYNC_REG is not yet availalable

MPI_ASYNC_PROTECTS_NONBLOCKING == .TRUE. requires a TS29113 compiler

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

Fortran

Nonblocking MPI routines and strided sub-arrays in Fortran

- **Fortran:**

  MPI_ISEND (*buf(7,:,:)*, ..., *request_handle*, *ierror*)

  - 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**.

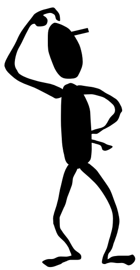
  other work

  - 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**!

  MPI_WAIT(request_handle, *status*, *ierror*)

- **Since MPI-3.0: Works if MPI_SUBARRAYS_SUPPORTED == .TRUE.**

  (requires TS29113 compiler)



- **MPI-1.0 – MPI-2.2:**
  Do not use non-contiguous sub-arrays in nonblocking calls!!!
  - Use first sub-array element (**buf(1,1,9)**) instead of whole sub-array (**buf(:,:,9:13)**)
  - *Call by reference* necessary ➔ *Call by in-and-out-copy* forbidden
    ➔ **use the correct compiler flags!**

New in MPI-3.0

Fortran

Major enhancement with a full MPI-3.0 implementation

- The following features require Fortran 2003 + TS 29113
  - 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 Fortran 2003+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:

- 17.1.8 Additional Support for Fortran Register-Memory-Synchronization
- 17.1.10 Problems With Fortran Bindings for MPI
- 17.1.11 Problems Due to Strong Typing
- 17.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets
- 17.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts
- 17.1.14 Special Constants
- 17.1.15 Fortran Derived Types
- 17.1.16 Optimization Problems, an Overview
- 17.1.17 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 17.1.7)**
  - **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**
- 17.1.18 Temporary Data Movement and Temporary Memory Modication
- 17.1.19 Permanent Data Movement
- 17.1.20 Comparison with C

New in MPI-3.0

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 <u>nonblocking operation immediately followed by a matching wait</u> is equivalent to the <u>blocking operation,</u> except the Fortran problems.

Completion

C

Fortran

- C/C++:

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

- Fortran:

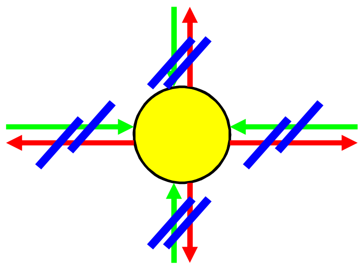
  ```
  CALL MPI_WAIT( request_handle, status, ierror)
  CALL MPI_TEST( request_handle, flag, status, ierror)
  ```

- one must
  - WAIT or
  - loop with TEST until request is completed, i.e., flag == 1 or .TRUE.

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 **as many** messages as possible
  - **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 6. Derived Datatypes

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 Chap. 9 *Collective Communication*

No answer by the MPI standard, because:

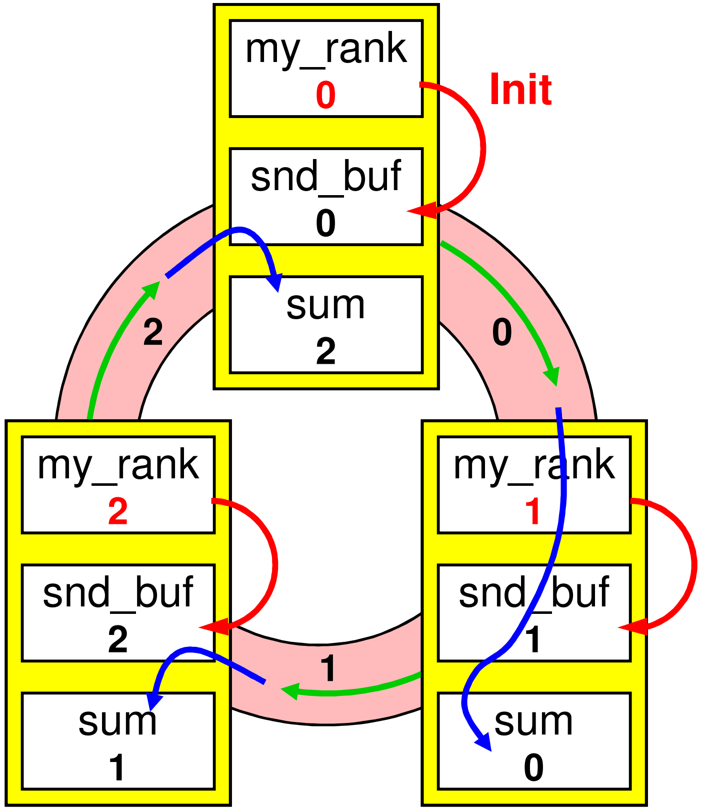
MPI targets portable and efficient message-passing programming
but
efficiency of MPI application-programming is not portable!

Exercise — Rotating information around a ring

Numbers used on next slide

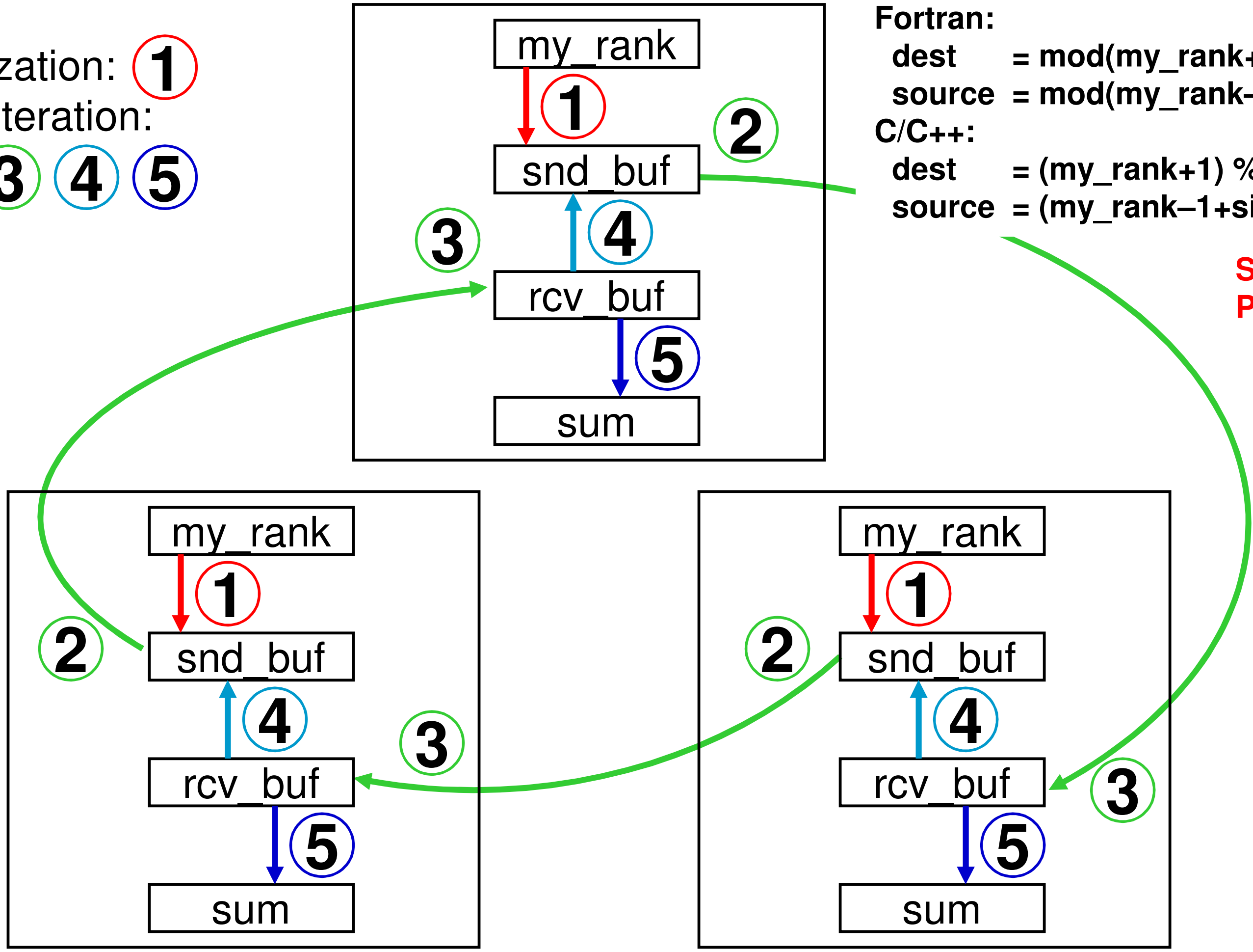
1
2
3
4
5

- A set of processes are arranged in a ring.
- Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.
- Each process passes this on to its neighbor on the right.
- Each processor 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.
- Use nonblocking MPI_Issend
  - to avoid deadlocks
  - to verify the correctness, because blocking synchronous send will cause a deadlock



Exercise — 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 !!!

see also login-slides

Fortran: Do not forget MPI-3.0 → **..., ASYNCHRONOUS :: ..._buf** and **IF(.NOT.MPI_...) CALL MPI_F_SYNC_REG(...)**
or MPI-2.2 . . . . → **CALL MPI_GET_ADDRESS(..., iadummy, ierror)**

Sol.

Advanced Exercises — Irecv instead of Issend

- Substitute the Issend–Recv–Wait method by the Irecv–Ssend–Wait method in your ring program.

- Or

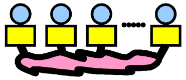
- Substitute the Issend–Recv–Wait method by the Irecv–Issend–Waitall method in your ring program.

For private notes

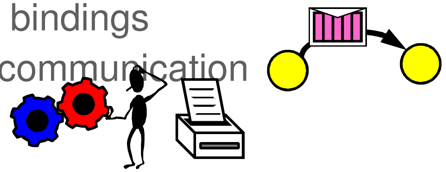
For private notes

Chap.5 Probe, Persistent Requests, Cancel

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

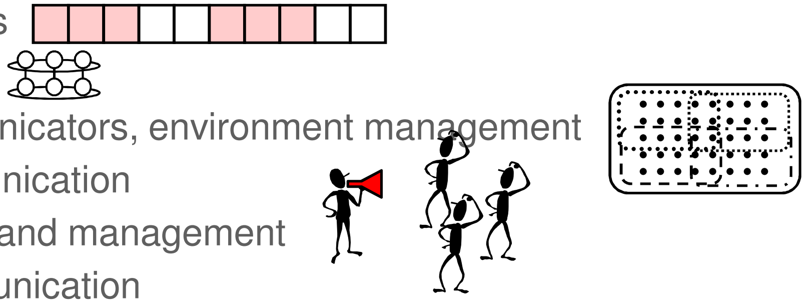


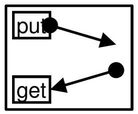
MPI_Init()
MPI_Comm_rank()

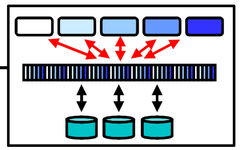


5. Probe, Persistent Requests, Request_free, Cancel

6. Derived datatypes
7. Virtual topologies
8. Groups & communicators, environment management
9. Collective communication
10. Process creation and management
11. One-sided communication
12. Shared memory one-sided communication
13. MPI and threads
14. Parallel file I/O
15. Other MPI features







Skip this section (4 slides)

Probing a message

- Goal – before receiving the message:
  - **Look at the message envelop (status) to examine the message count**
  - **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**
  - Matching Probe
    - **MPI_Improbe(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)**

  → 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

Within one MPI process, thread A may call MPI_PROBE.
Another tread B may steal the probed message.
Thread A calls MPI_RECV, but may not receive the probed message.

New in MPI-3.0

MPI_Message handle,
e.g., stored in a thread-local variable

Persistent Requests

For communication calls with identical argument lists in each loop iteration (only buffer content changes):

- MPI_(,B,S,R)SEND_INIT and MPI_RECV_INIT
  - Creates a persistent MPI_Request handle
  - Status of the handle is initiated as *inactive*
  - Does not communicate
  - It only setups the argument list
- MPI_START(request [,ierrror]) / MPI_STARTALL(cnt, requests [,ierrror])
  - Starts the communication call(s) as nonblocking call(s), i.e., handle gets *active*
  - To be completed with regular MPI_WAIT… / MPI_TEST… calls → *inactive*
- MPI_REQUEST_FREE to finally free such a handle
- Usage sequence: INIT Loop(START WAIT/TEST) FREE
- Enables additional optimizations within the MPI library

MPI_Request_free

- 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 <u>never</u> be used for <u>*active* receive</u> requests
  - Really useful only for <u>*in*</u>*active* persistent requests i.e., after such Loop(START WAIT/TEST), i.e., not after START

MPI_Cancel

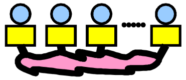
- Marks a active nonblocking communication handle for cancellation.
- MPI_CANCEL is a local call, i.e., returns immediately.
- **Subsequent call to MPI_Wait must return irrespective of the activities of other processes.**
- **Either** the **cancellation** or the **communication** succeeds, but not both.
- MPI_TEST_CANCELLED(wait_status, flag [,ierror])
  - flag = true → cancellation succeeded, communication failed
  - flag = false → cancellation failed, communication succeeded

For private notes

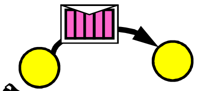
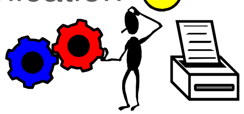
For private notes

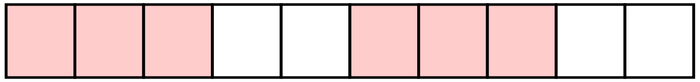
For private notes

Chap.6 Derived Datatypes

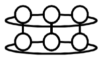
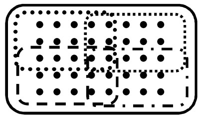
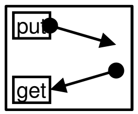
1. MPI Overview 

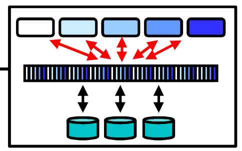
MPI_Init()
MPI_Comm_rank()

2. Process model and language bindings
3. Messages and point-to-point communication 
4. Nonblocking communication
5. Probe, Persistent Requests, Cancel 

6. Derived datatypes 

– **transfer of any combination of typed data**

7. Virtual topologies 
8. Groups & communicators, environment management 
9. Collective communication 
10. Process creation and management
11. One-sided communication
12. Shared memory one-sided communication 

13. MPI and threads
14. Parallel file I/O 
15. Other MPI features

MPI Datatypes

- Description of the memory layout of the buffer
  - for sending
  - for receiving
- Basic types
- Derived types
  - vectors
  - structs
  - others

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:

0		4	8	12	16 20 24
c		11	22		6.36324d+107

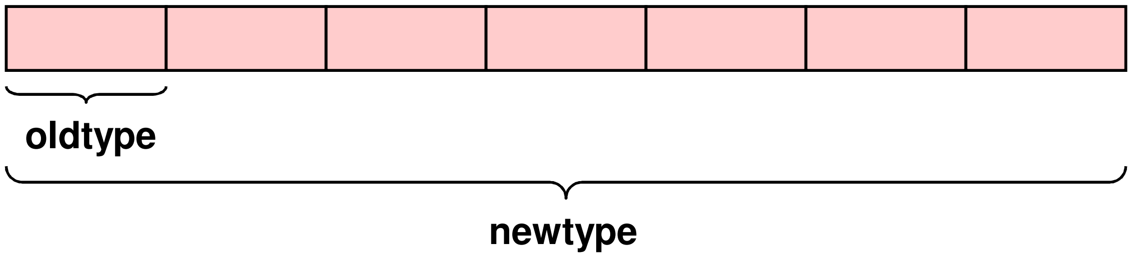
derived datatype handle

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

Fortran

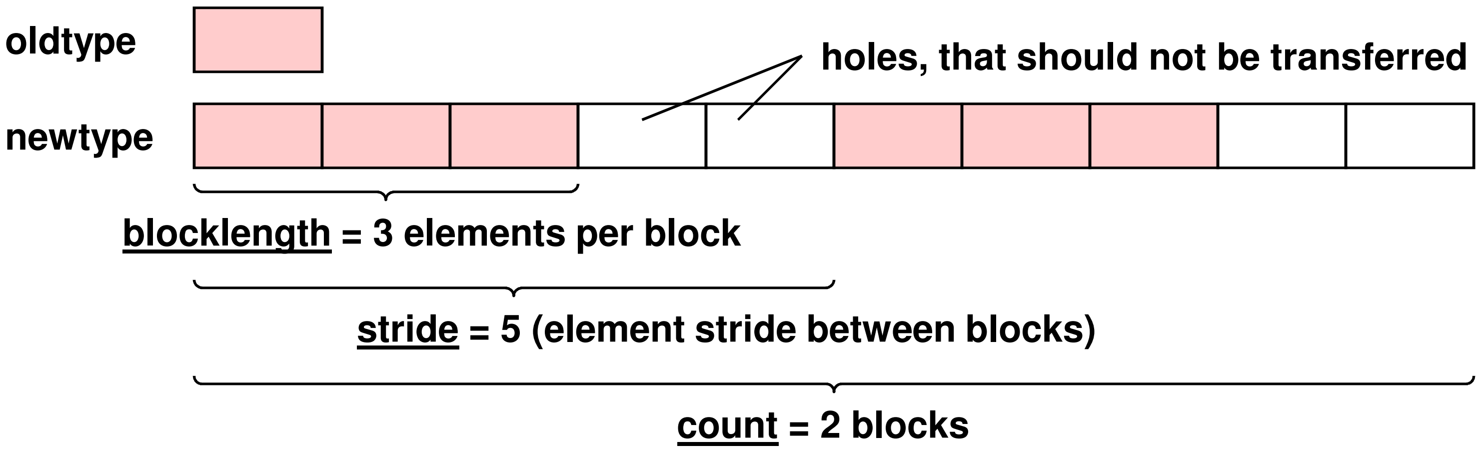
- C/C++: int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
- Fortran: MPI_TYPE_CONTIGUOUS(count, oldtype, *newtype, ierror*)

mpi_f08: INTEGER :: count
TYPE(MPI_Datatype) :: oldtype, newtype
INTEGER, OPTIONAL :: ierror

mpi & mpif.h: INTEGER count, oldtype, newtype, ierror

Handout only contains old style interface

Vector Datatype



C

- C/C++: int MPI_Type_vector(int count, int blocklength, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)

Fortran

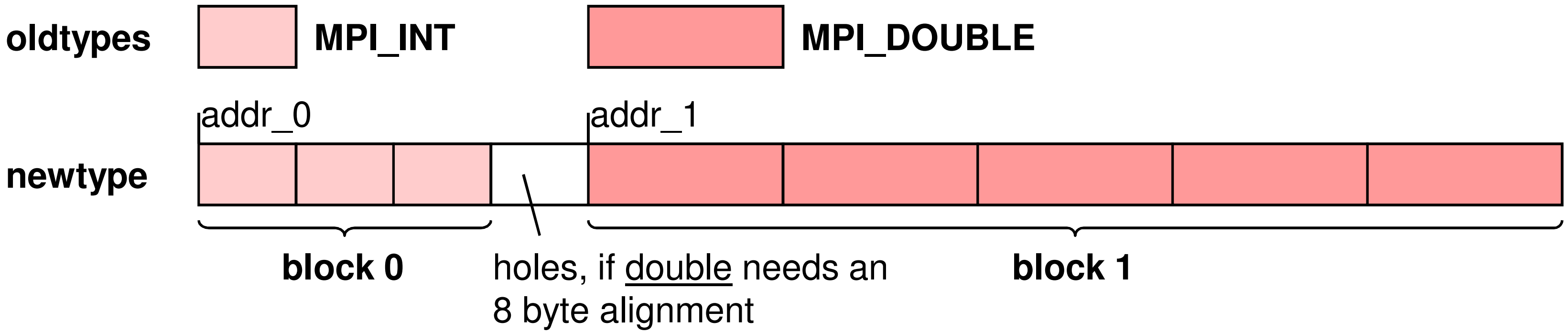
- Fortran: MPI_TYPE_VECTOR(count, blocklength, stride,
  oldtype, *newtype*, *ierror*)

mpi_f08:

```
INTEGER                 :: count, blocklength, stride
TYPE(MPI_Datatype)      :: oldtype, newtype
INTEGER, OPTIONAL       :: ierror
```

mpi & mpif.h: INTEGER count, blocklength, stride, oldtype, newtype, ierror

Struct Datatype



C

Fortran

- 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_displacements [1)] ,
  array_of_types, _newtype_, _ierror_)

count	**=**	**2**	
array_of_blocklengths	**= (3,**	**5**	**)**
array_of_displacements	**= (0,**	**addr_1 – addr_0**	**)**
array_of_types	**= (MPI_INT,**	**MPI_DOUBLE**	**)**

[1)] INTEGER**(KIND=MPI_ADDRESS_KIND)** array_of_displacements

Memory Layout of Struct Datatypes

buf_datatype	int				double				

Fixed memory layout:

- C

```
struct buff
   {  int          i_val[3];
      double    d_val[5];
   }
```

- Fortran, common block

```
integer i_val(3)
double precision d_val(5)
common /bcomm/ i_val, d_val
```

- 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
```

!!!

Recommended in MPI-3.0:
TYPE, **BIND(C)** :: buff_type

~~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 optimi.:** MPI_Send & _Recv of **…d_val** is invisible for the compiler → add MPI_Address~~

~~**in_buf_datatype**~~

in_i_val		in_d_val			

~~**out_buf_datatype**~~

out_i_val		out_d_val			

Not portable, because address differences are allowed only inside of structures or arrays → MPI-3.0, 4.1.12

How to compute the displacement

- array_of_displacements[i] := address(block_i) – address(block_0)

C

Fortran

- C/C++: int MPI_**Get**_address(void* location, MPI_Aint **address*)
- Fortran: MPI_**GET**_ADDRESS(LOCATION, *ADDRESS*, *IERROR*)

  mpi_f08: 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

C

Fortran

- Examples: MPI-3.0, Example 4.17, pp 125-128

Committing ad Freeing a Datatype

- Before a dataytype 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).
(More than once use equivalent to additional no-operations.)

C

Fortran

- 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

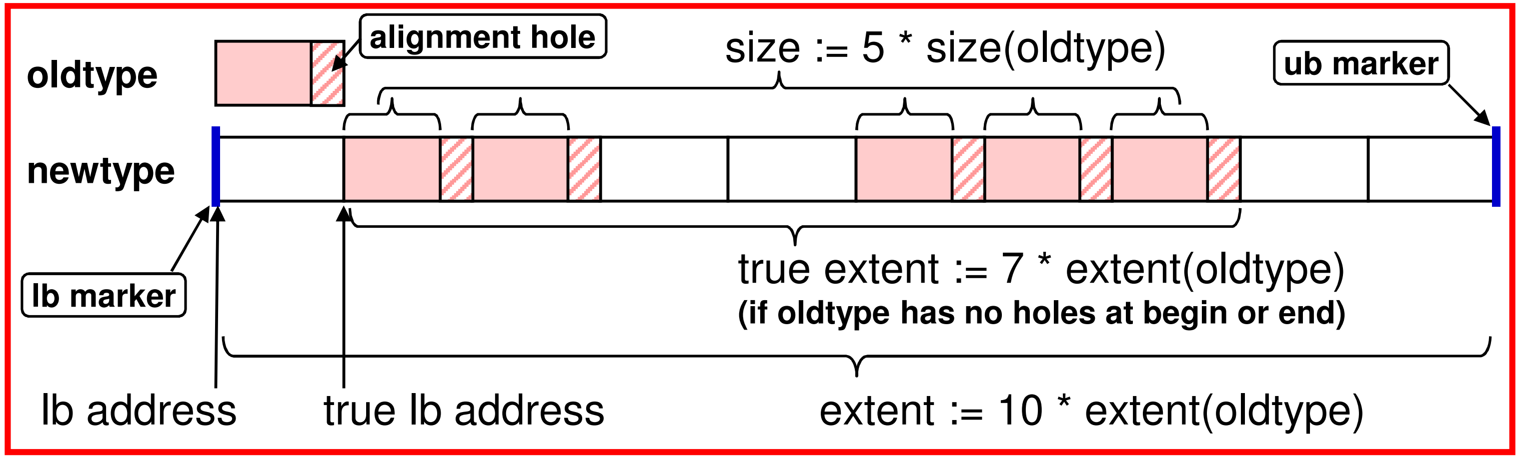
IN-OUT argument

- If usage is over, one may call MPI_TYPE_FREE()
to free a datatype and its internal resources.

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:



Size and Extent of a Datatype, II.

C

Fortran

– 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

C

Fortran

– 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

– C/C++: int MPI_Type_get_true_extent(MPI_Datatype datatype,
MPI_Aint **true_lb*, MPI_Aint **true_extent*)

– Fortran: dito

- 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, still undefined, but recommended to use a BIND(C) environment.

Alignment rule, holes and resizing of structures

- Never trust the compiler that it correctly computes the alignment hole at the end of a structure!
  - See MPI-3.0, Sect. 4.1.6, Advice to users on page 106
- This alignment hole is only important when using an array of structures!
- Implication **(for C and Fortran!)**:
  - 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
  - applies additionally MPI_TYPE_CREATE_RESIZED to this datatype handle.
  - See Example in MPI-3.0, Sect. 17.1.15 on pages 629-630.
- 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-3.0, Sect. 4.1.6, Advice to users on page 106

Large Counts with MPI_Count, …

New in MPI-3.0

- 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
- sizeof(int) ≤ sizeof(MPI_Aint) / sizeof(MPI_Offset) ≤ sizeof(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**

All Derived Datatype Creation Routines (1)

- **MPI_Type_contiguous()**
  → already discussed
- **MPI_Type_vector()**
  → already discussed

  MPI_Type_create_hvector()
  → stride as byte size
- **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_hindexed()
  → with byte displacements
- **MPI_Type_create_indexed_block()**
  → same as MPI_Type_indexed()
  but same block length
  for each sub-block

  MPI_Type_create_hindexed_block()
  → with byte displacements
- **MPI_Type_create_struct()**
  → already discussed

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 Chap.14 *Parallel File I/O*
- **MPI_Type_create_darray()**
  - A generalization of **MPI_Type_create_subarray()**
  - Example, see course Chap.14 *Parallel File I/O*

Removed MPI-1 interfaces	substituted by
MPI_Address	MPI_Get_address
MPI_Type_extent	MPI_Type_get_extent
MPI_Type_hvector	MPI_Type_create_hvector
MPI_Type_hindexed	MPI_Type_create_hindexed
MPI_Type_struct	MPI_Type_create_struct
MPI_Type_LB / _UB	MPI_Type_get_extent
Constant MPI_LB / _UB	MPI_Type_resized

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

Fortran

- MPI_BOTTOM (in point-to-point and collective communication)
  - For messages with derived datatypes with absolute displacements
  - Displacements must be retrieved with MPI_GET_ADDRESS()
  - Buffer argument is then MPI_BOTTOM
  - MPI_BOTTOM is an address,
    i.e., **cannot be assigned to a Fortran variable**!
  - MPI-3.0, Section 2.5.4, page 15 line 42 – page 16 line 3
    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 & derived datatype with absolute displacement
    of variable X → MPI_F_SYNC_REG is needed:
    - **MPI_BOTTOM in a <u>blocking</u> MPI routine → MPI_F_SYNC_REG before and after this routine**
    - **in a <u>nonblocking</u> routine → MPI_F_SYNC_REG before this routine & after final WAIT/TEST**

Performance options

Which is the fastest neighbor communication with strided data?

- Using derived datatype handles
- Copying the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer, and copying the rcv-buffer back into the strided application array
- 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!

Exercise — Derived Datatypes

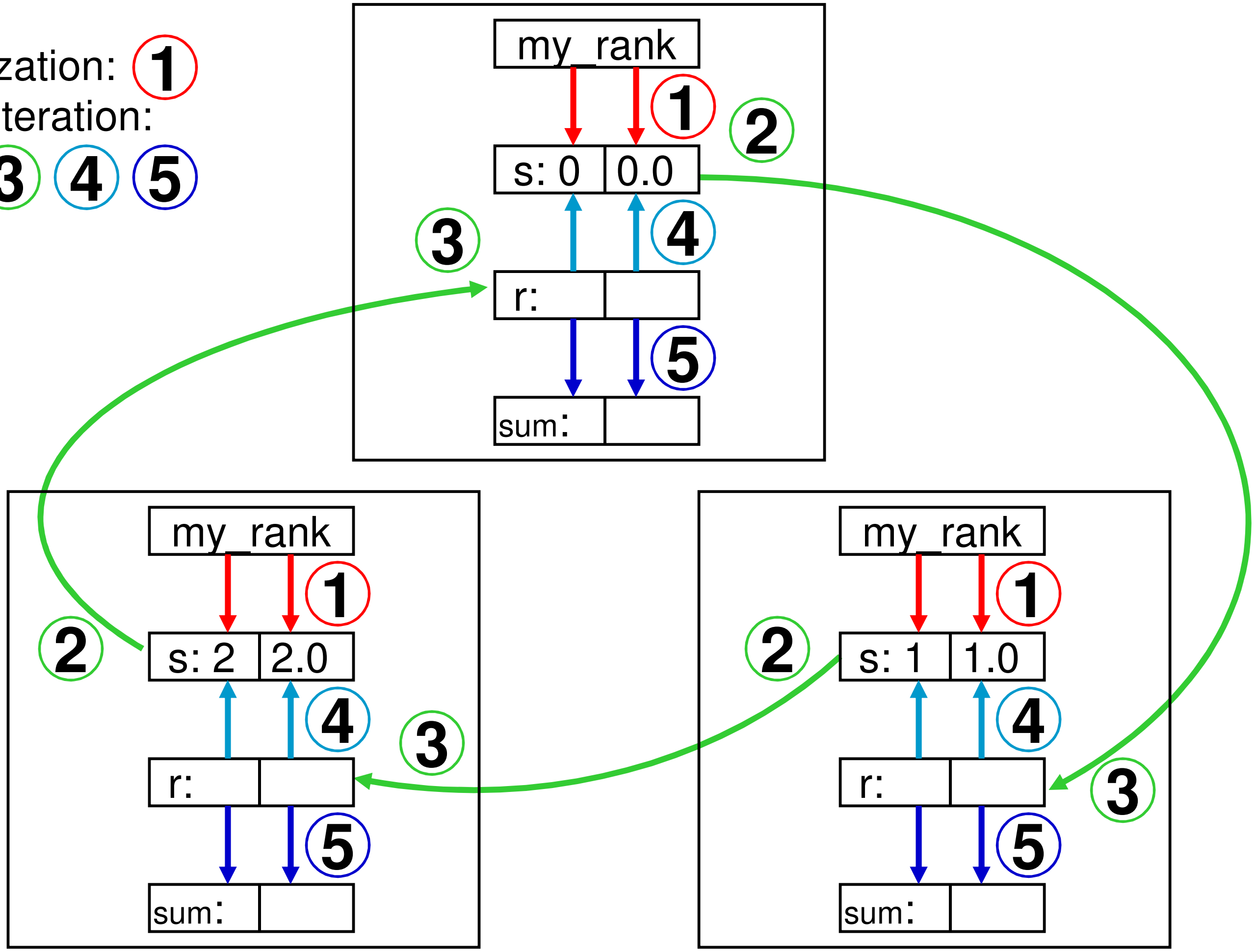
C

Fortran

- Modify the pass-around-the-ring exercise.
- Use the following skeletons to reduce software-coding time:
  cp ~/MPI/course/**C**/Ch6/derived_skeleton_20.c derived_20.c
  cp ~/MPI/course/**F_30**/Ch6/derived_struct_skeleton_30.f90 derived_struct_30.f90
  or
  cp ~/MPI/course/**F_20**/Ch6/derived_struct_skeleton_20.f90 derived_struct_20.f90
- 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 — Derived Datatypes

Initialization: 1
Each iteration:
2 3 4 5



H L R I S

see also login-slides

Sol.

Advanced Exercises — Sendrecv & Sendrecv_replace

- 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.)
- 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.**

For private notes

For private notes

- private notes

For private notes

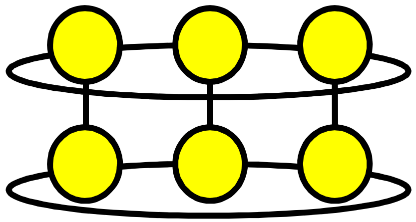
Chap.7 Virtual Topologies

MPI_Init()
MPI_Comm_rank()

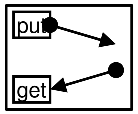
1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. Probe, Persistent Requests, Cancel
6. Derived datatypes

7. Virtual topologies

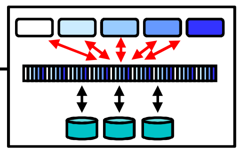
– **a multi-dimensional process naming scheme**



8. Groups & communicators, environment management
9. Collective communication
10. Process creation and management
11. One-sided communication
12. Shared memory one-sided communication



13. MPI and threads
14. Parallel file I/O
15. Other MPI features



Example

• Global array	A(1:3000,	1:4000,	1:500)	=	$6 \cdot 10^9$ words
• on	**3** x	**4** x	**5**	**=**	**60 processors**
• process coordinates	**0..2,**	**0..3,**	**0..4**		

- example:
  on process **ic_0=2,** **ic_1=0,** **ic_2=3** **(rank=43)**
  decomposition, e.g., A(2001:3000, 1:1000, 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

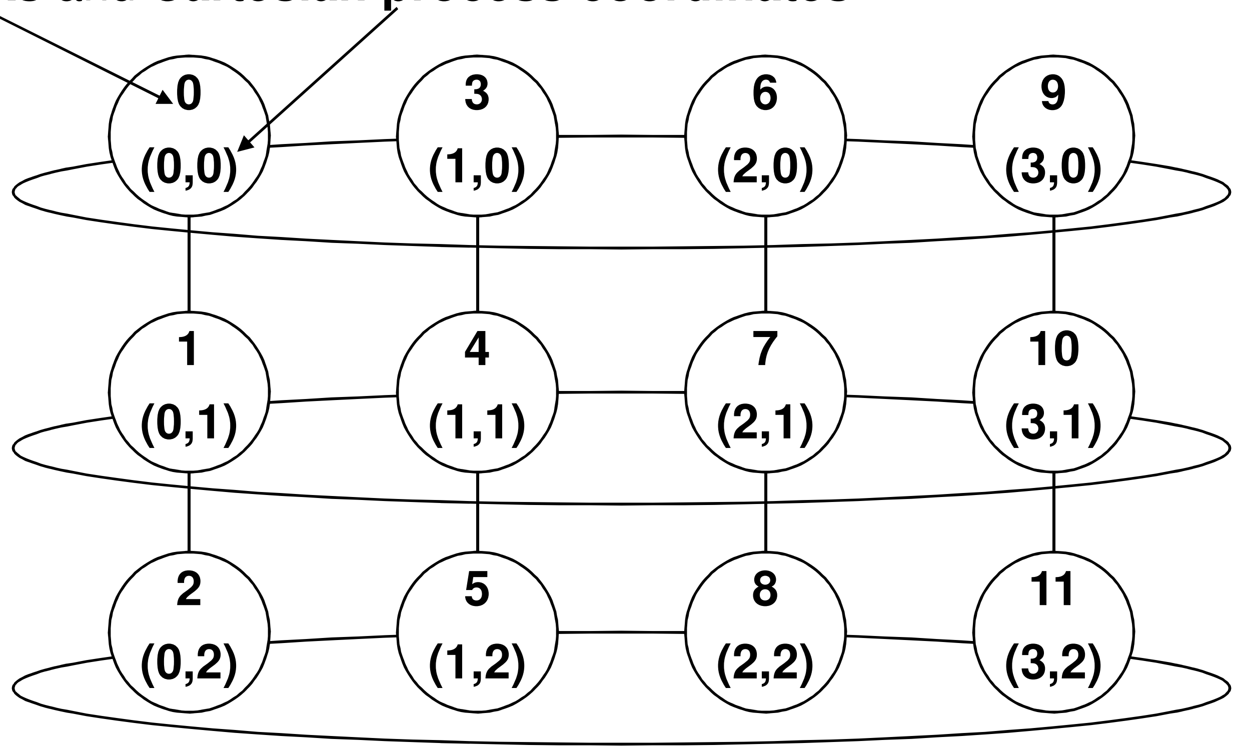
- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimize communications.

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 – A 2-dimensional Cylinder

- **Ranks** and **Cartesian process coordinates**



Topology Types

- Cartesian Topologies
  - each process is *connected* to its neighbor in a virtual 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.

Creating a Cartesian Virtual Topology

C

- C/C++: int MPI_Cart_create(MPI_Comm comm_old, int ndims,
  int *dims, int *periods, int reorder,
  MPI_Comm *comm_cart)

Fortran

- Fortran: MPI_CART_CREATE(comm_old, ndims, dims, periods,
  reorder, comm_cart, ierror)

mpi_f08:

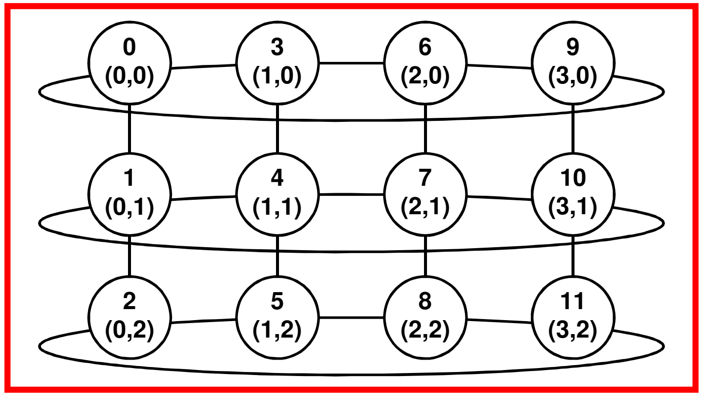
```
TYPE(MPI_Comm)        :: comm_old, comm_cart
INTEGER               :: ndims, dims(*)
LOGICAL               :: periods(*), reorder
INTEGER, OPTIONAL     :: ierror
```

mpi & mpif.h:

```
INTEGER comm_old, ndims, dims(*), comm_cart, ierror
LOGICAL  periods(*), reorder
```

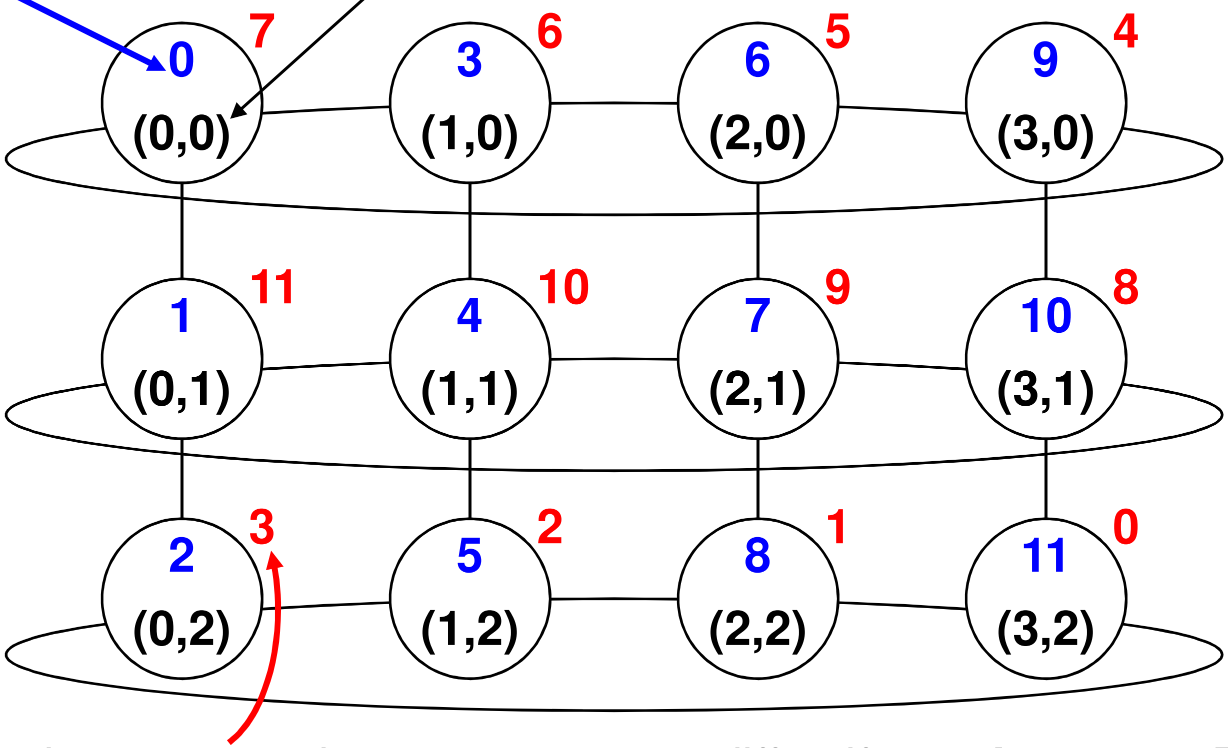
comm_old	**=**	**MPI_COMM_WORLD**	
ndims	**=**	**2**	
dims	**=**	**(4, 3)**	
periods	**=**	**(1, 0)**	(in C)
periods	**=**	**(.true., .false.)**	(in Fortran)
reorder	**=**	***see next slide***	

e.g., size==12 factorized with MPI_Dims_create(), see advanced exercise



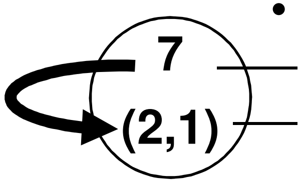
Example – A 2-dimensional Cylinder

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



- Ranks in **comm** and **comm_cart** may differ, if **reorder = 1 or .TRUE.**
- This reordering can allow MPI to optimize communications

Cartesian Mapping Functions



- Mapping
  ranks to
  process grid coordinates

C

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

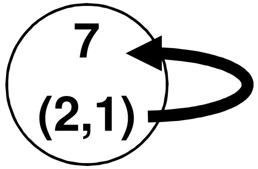
Fortran

- 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	

Cartesian Mapping Functions

- Mapping process grid coordinates to ranks



C

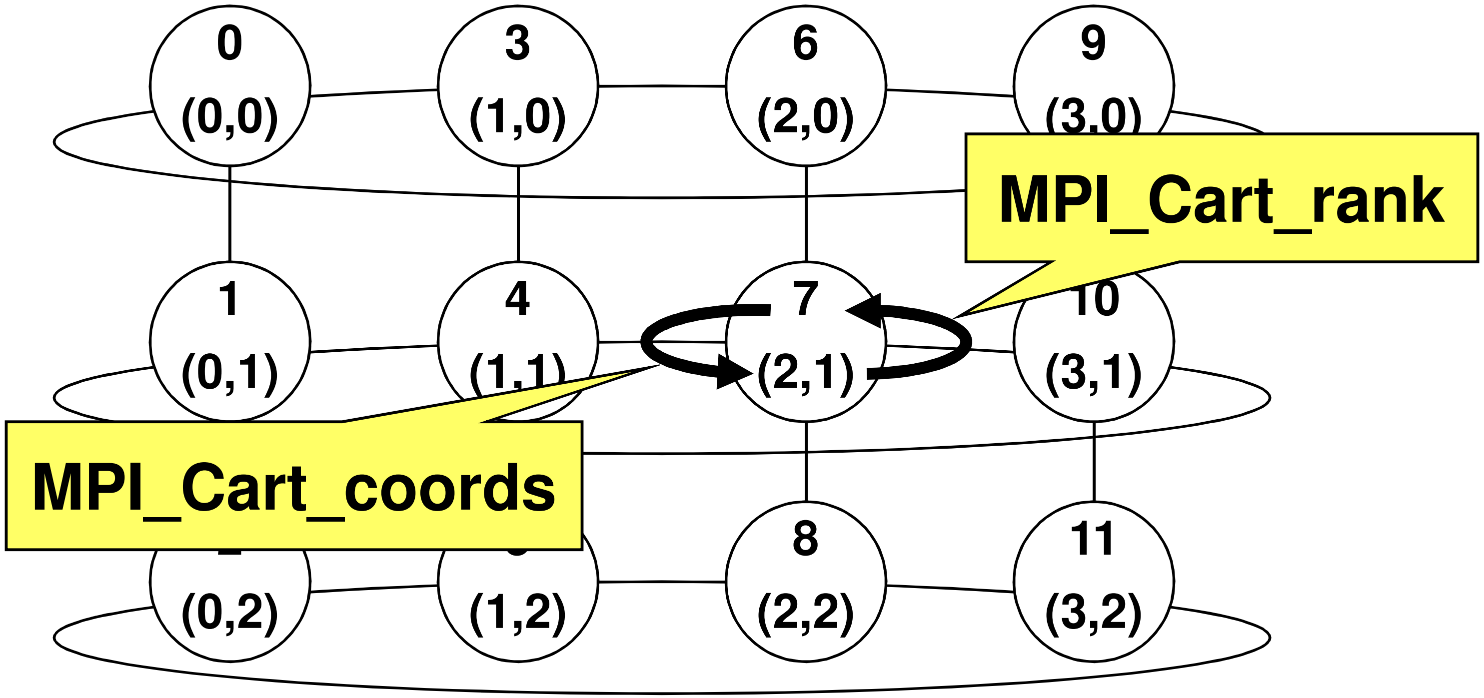
- C/C++: int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int **rank*)

Fortran

- 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	

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*)

Cartesian Mapping Functions

- Computing ranks of neighboring processes

C

Fortran

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

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

mpi_f08:

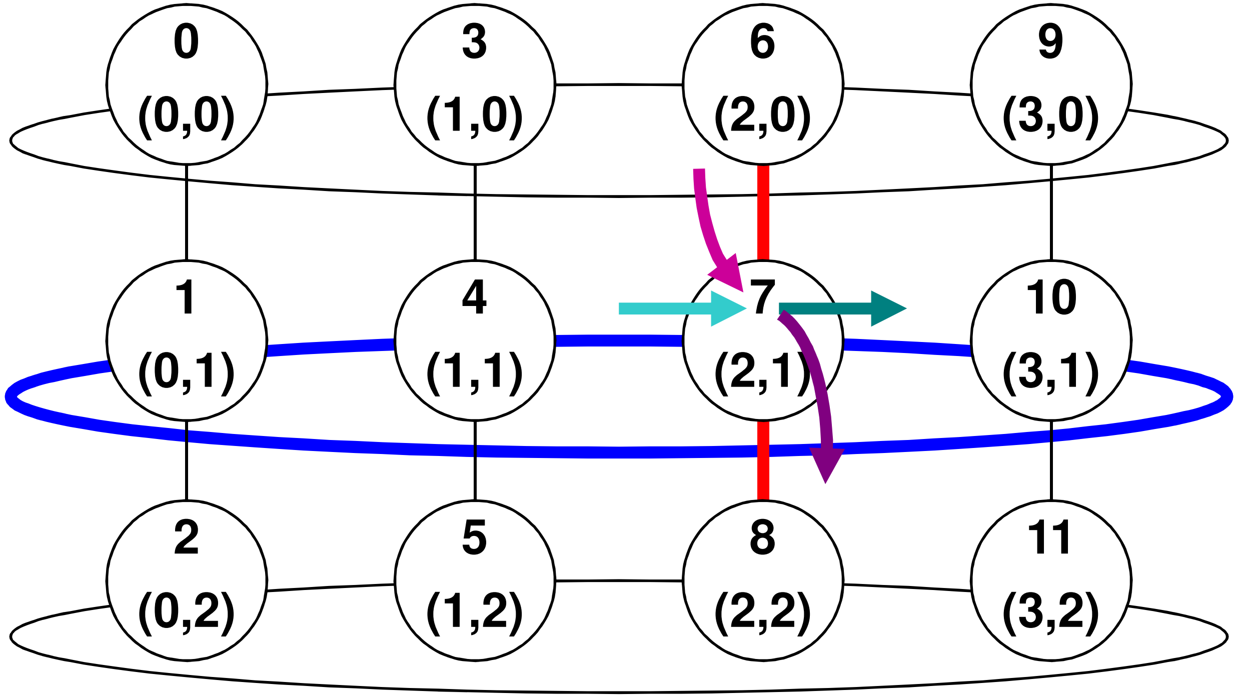
```
TYPE(MPI_Comm)          :: comm_cart
INTEGER                 :: direction, disp, rank_source, rank_dest
INTEGER, OPTIONAL       :: ierror
```

mpi & mpif.h:

```
INTEGER  comm_cart, direction, disp, rank_source, rank_dest, ierror
```

- 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 cart

- MPI_Cart_shift(cart, direction, displace, *rank_source*, *rank_dest*, *ierror*)

	direction	displace	rank_source	rank_dest
example on	0 or	+1	4	10
process rank=**7**	1	+1	6	8

H L R S

Cartesian Partitioning

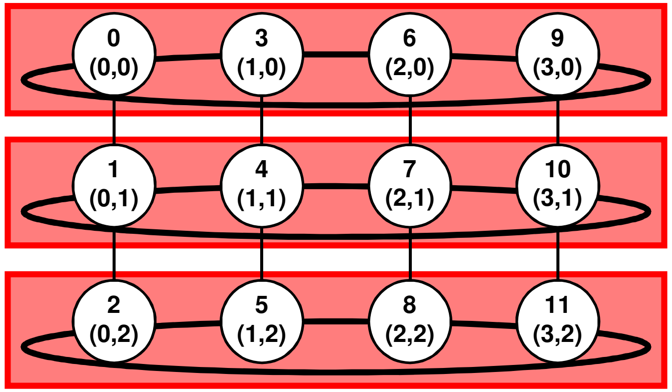
- Cut a grid up into *slices*.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.

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*)

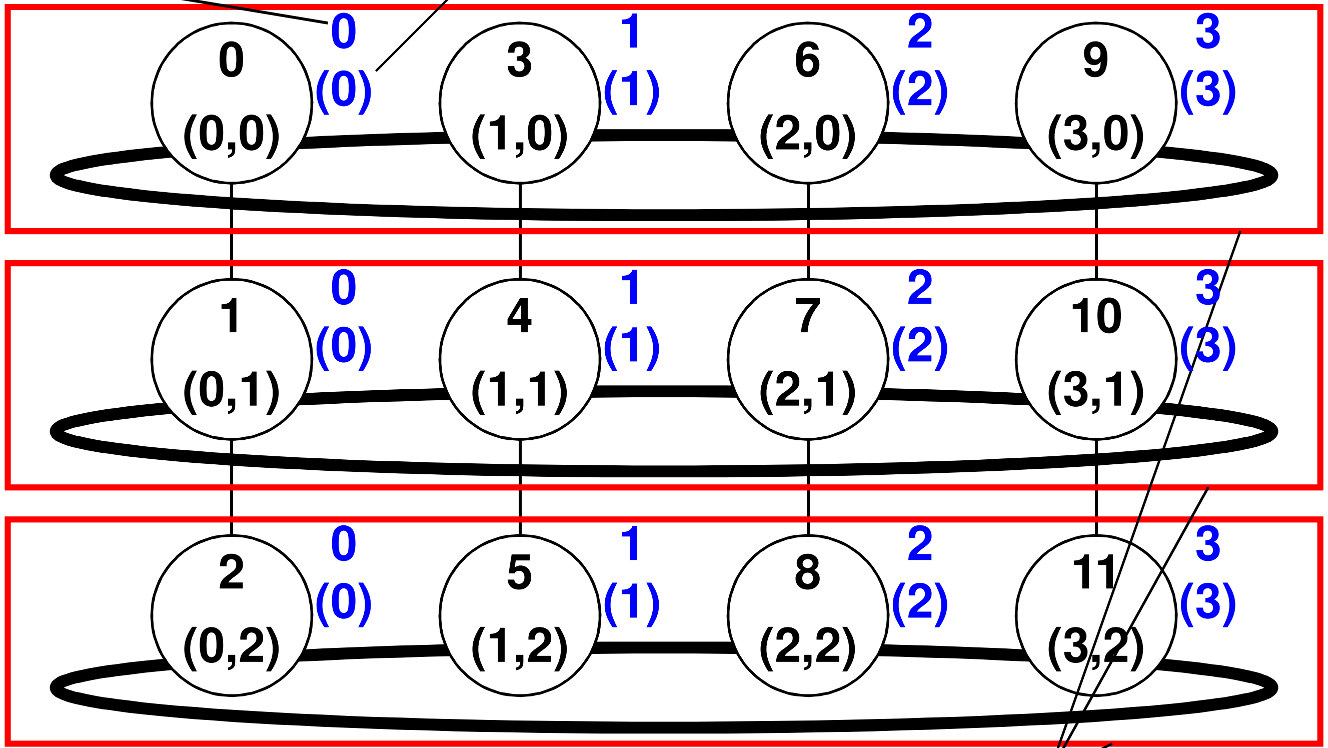



```
mpi_f08:        TYPE(MPI_Comm)       :: comm_cart
                LOGICAL              :: remain_dims(*)
                TYPE(MPI_Comm)       :: comm_slice
                INTEGER, OPTIONAL    :: ierror

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

MPI_Cart_sub – Example

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



- MPI_Cart_sub(comm_cart, remain_dims, ***comm_sub***, *ierror*)

(true, false)

Exercise — One-dimensional ring topology

Fortran

C

- Rewrite the pass-around-the-ring program using a one-dimensional ring topology.
- Use the results from Chap. 4 (nonblocking, without derived datatype):
  ~/MPI/course/**F_30**/Ch4/ring_30.f90 (with mpi_f08 module)
  ~/MPI/course/**F_20**/Ch4/ring_20.f90 (with mpi module)
  ~/MPI/course/**C**/Ch4/ring.c
- 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.
  - 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.
  - Only **one**-dimensional:
    - → only **`direction`**=0
    - → 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)
    - → coordinates are not necessary, because coord==rank

H L R I S

see also login-slides

Sol.

Advanced Exercises — Two-dimensional topology

- 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().

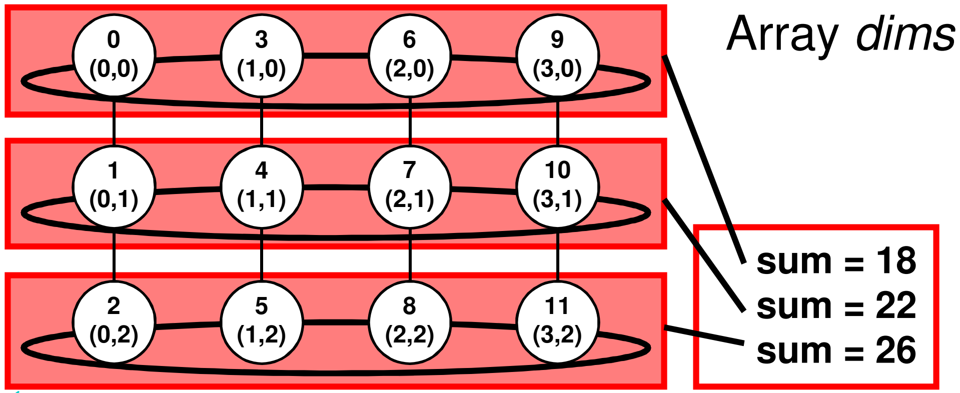
C

Fortran

- 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 **(0,0)**

For private notes

For private notes

For private notes

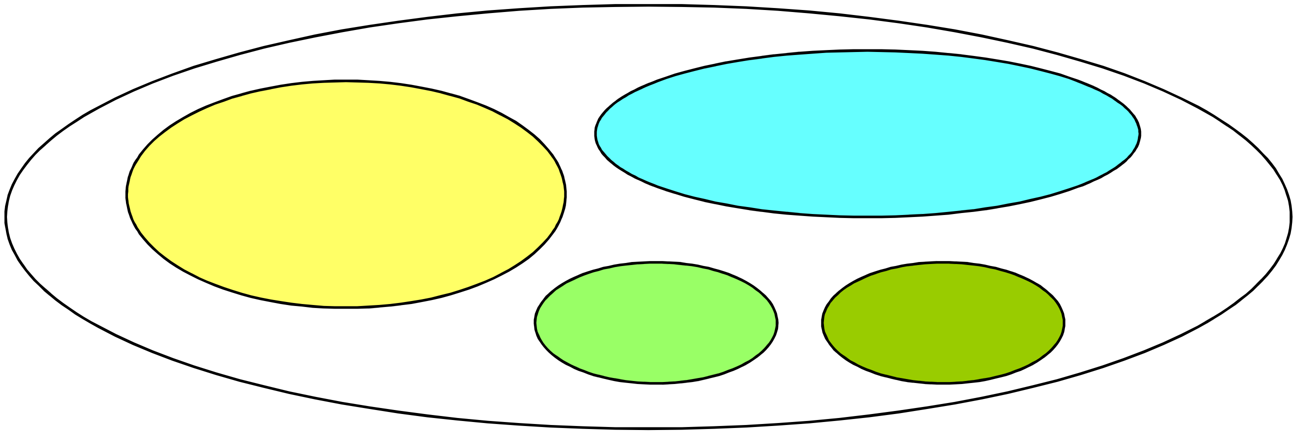
Chap.8 Groups & Communicators, Environment managem.

2nd skip-point after 4 slides / Move to error handling, i.e., skip details in this section (13 slides)

Goals

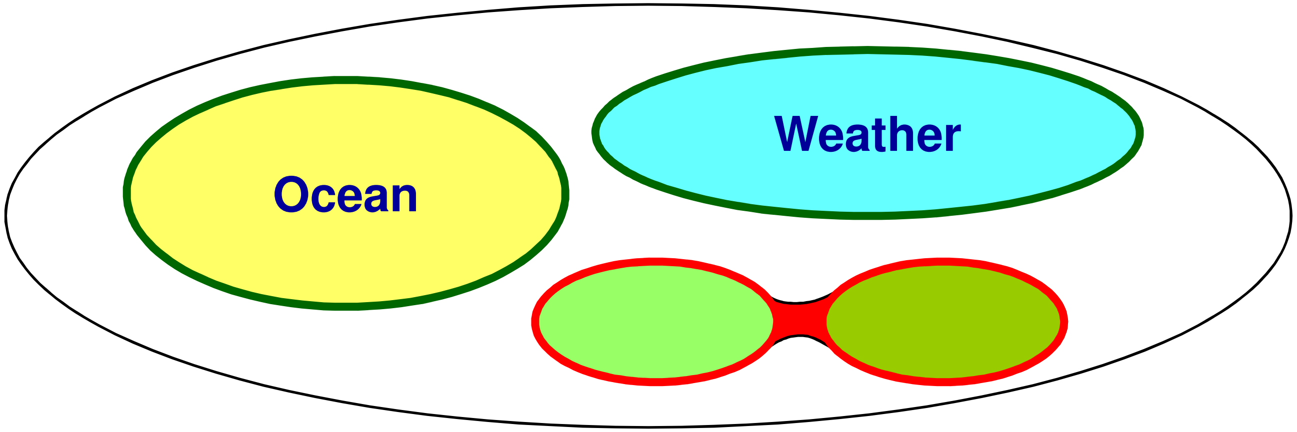
Support for libraries or application sub-spaces

- Safe communication context spaces
- Group scope for collective operations (→ next course chapter)
- Naming of context spaces
- Add additional user-defined attributes to a communication context



Methods

- **Sub-communicators**
  - Collectively defining communication sub-spaces
- Groups and sub-groups
  - Locally defining ordered sets of MPI processes
- **Intra-** and **inter-communicators**
- Attribute caching and **object naming**
  - On communicators, datatypes, windows (see course Chap.11)



Sub-groups and sub-communicators

- Two levels:
  - Group of processes
    - **Without the ability to communicate**
    - **Local routines to build sub-sets**
  - Communicators
    - **Group of processes with additional ability to communicate**

Scalability problems when handling many processes in each process

- Sub-communicators
  - Several ways of establishing:
    - **Communicator → group → sub-group**
      original communicator + sub_group → sub-communicator
    - **Communicator → group → sub-group → sub-communicator**
    - **Communicator → many sub-communicators**
      - e.g., MPI_Comm_split
    - **Communicator → one sub-communicator**

Scaling interfaces

Scalability depends

Example: MPI_Comm_split()

C

Fortran

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

mpi_f08:
TYPE(MPI_Comm) :: comm, newcomm
INTEGER :: color, key;
INTEGER, OPTIONAL :: ierror

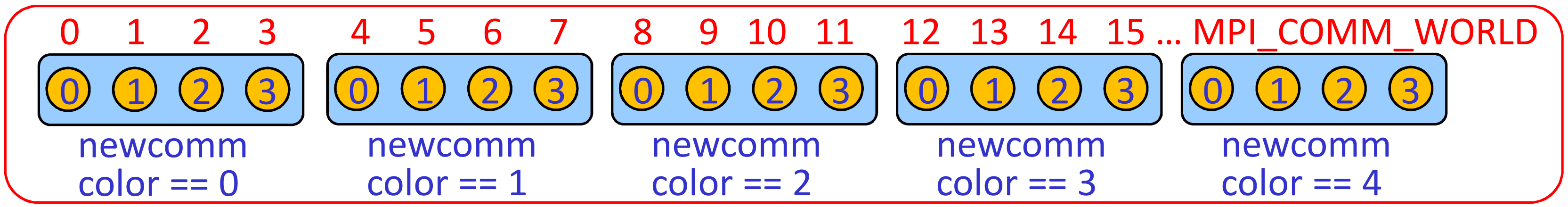
mpi & mpif.h: INTEGER comm, color, key, newcomm, ierror

Example:

```
int my_rank, color, key, my_new_rank;
MPI_Comm newcomm;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
color = my_rank/4;
key = 0;
MPI_Comm_split(MPI_COMM_WORLD, color, key, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

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

Key==0 → ranking in newcomm is sorted as in old comm
Key ≠ 0 → ranking in newcomm is sorted according key values



Move to error handling, i.e., skip details in this section (9 slides)

Sub-groups – non-collective routines

(Sub-)group creation routines:

- MPI_Comm_group (comm, *group*)
- MPI_Group_union (group1, group2, *newgroup*)
- MPI_Group_intersection (group1, group2, *newgroup*)
- MPI_Group_difference (group1, group2, *newgroup*)
- MPI_Group_incl (group, n, ranks, *newgroup*)
- MPI_Group_excl (group, n, ranks, *newgroup*)
- MPI_Group_range_incl (group, n, ranges, *newgroup*)
- MPI_Group_range_excl (group, n, ranges, *newgroup*)

Other routines:

- MPI_Group_rank (group, rank) // my_rank
- MPI_Group_translate_ranks (group1, n, ranks1, group2, *ranks2*)
- MPI_Group_compare (group1, group2, *result*)
- MPI_Group_free ([INOUT] *group*)

Sub-communicators – collective creation

Scaling interfaces

Partially scaling, e.g., when used for splitting in many sub-comms

To be collectively called by all processes of **comm**:

- MPI_Comm_dup (comm, *newcomm*) — New in MPI-3.0
- MPI_Comm_dup_with_info (comm, info, *newcomm*) — New in MPI-3.0
- MPI_Comm_idup (comm, [ASYNCHRONOUS[1]] *newcomm*, *request*)
- MPI_Comm_create (comm, group, *newcomm*) — Several sub-comms in one call since MPI-2.2
- MPI_Comm_split (comm, color, key, *newcomm*) → course Chap. 11
  – All processes with same color are grouped into separate sub-communicators
- MPI_Comm_split_type (comm, split_type, key, info, *newcomm*) — New in MPI-3.0
  – split_type = MPI_COMM_TYPE_SHARED
  – All processes on a shared memory node (= symmetric multi-processing, SMP) are grouped into separate sub-communicators

To be collectively called by all processes of **group**:

- MPI_Comm_create_group (comm, group, tag, *newcomm*) // pt-to-pt tag — New in MPI-3.0

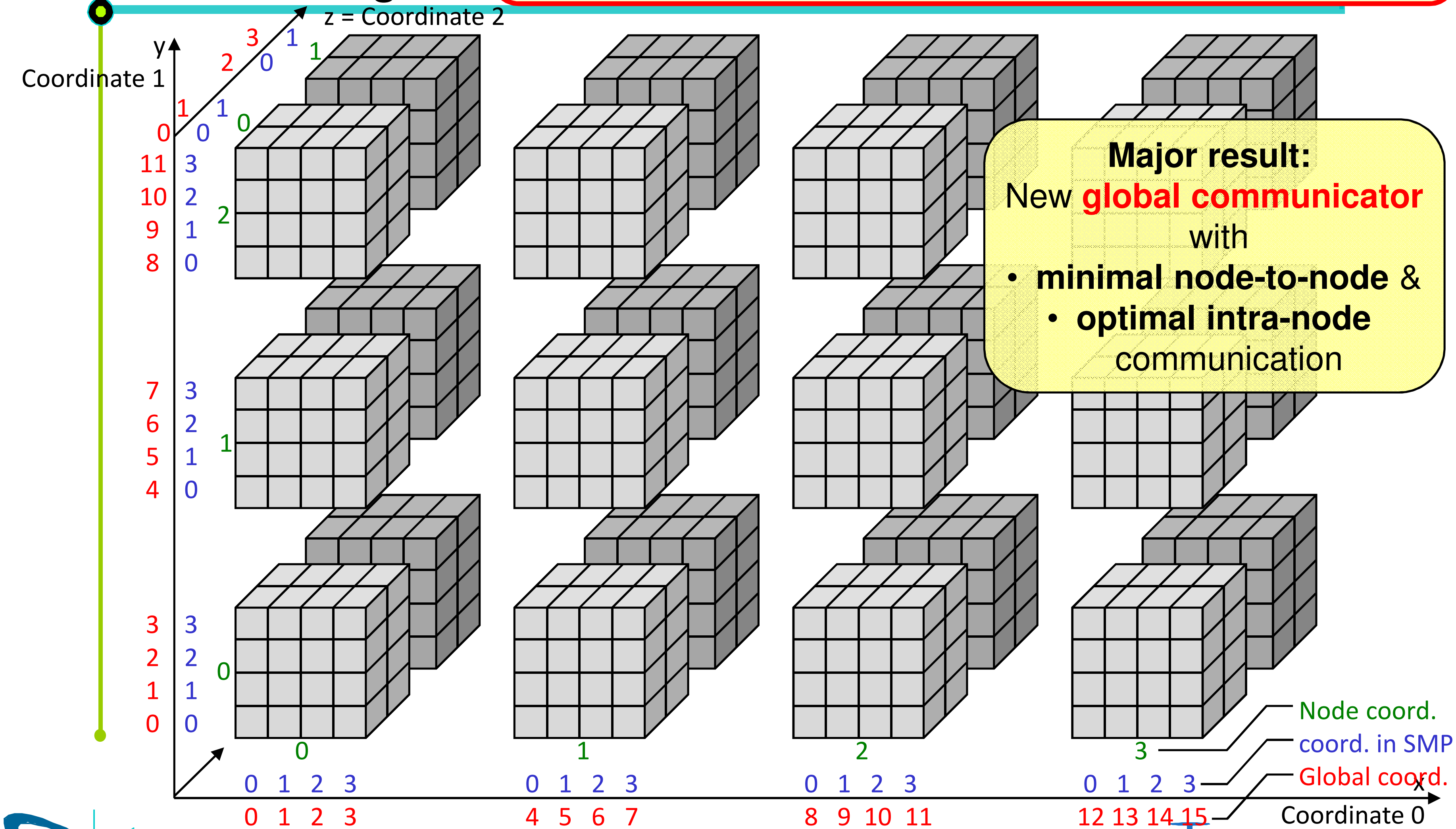
Other routines:

- MPI_Comm_free([INOUT] *comm*) MPI_Comm_compare(comm1, comm2, *result*)
- MPI_Comm_set_info(comm, info) MPI_Comm_get_info(comm, *info_used*)

[1] See MPI-3.0, errata Jan. 4, 2013 and later

Cartesian Grids – Renumbering

If MPI_Cart_create does not optimize:
- Renumbering of MPI_COMM_WORLD based on external knowledge or MPI_Comm_split_type(…MPI_COMM_TYPE_SHARED…)
- MPI_Cart_create with reorder=false

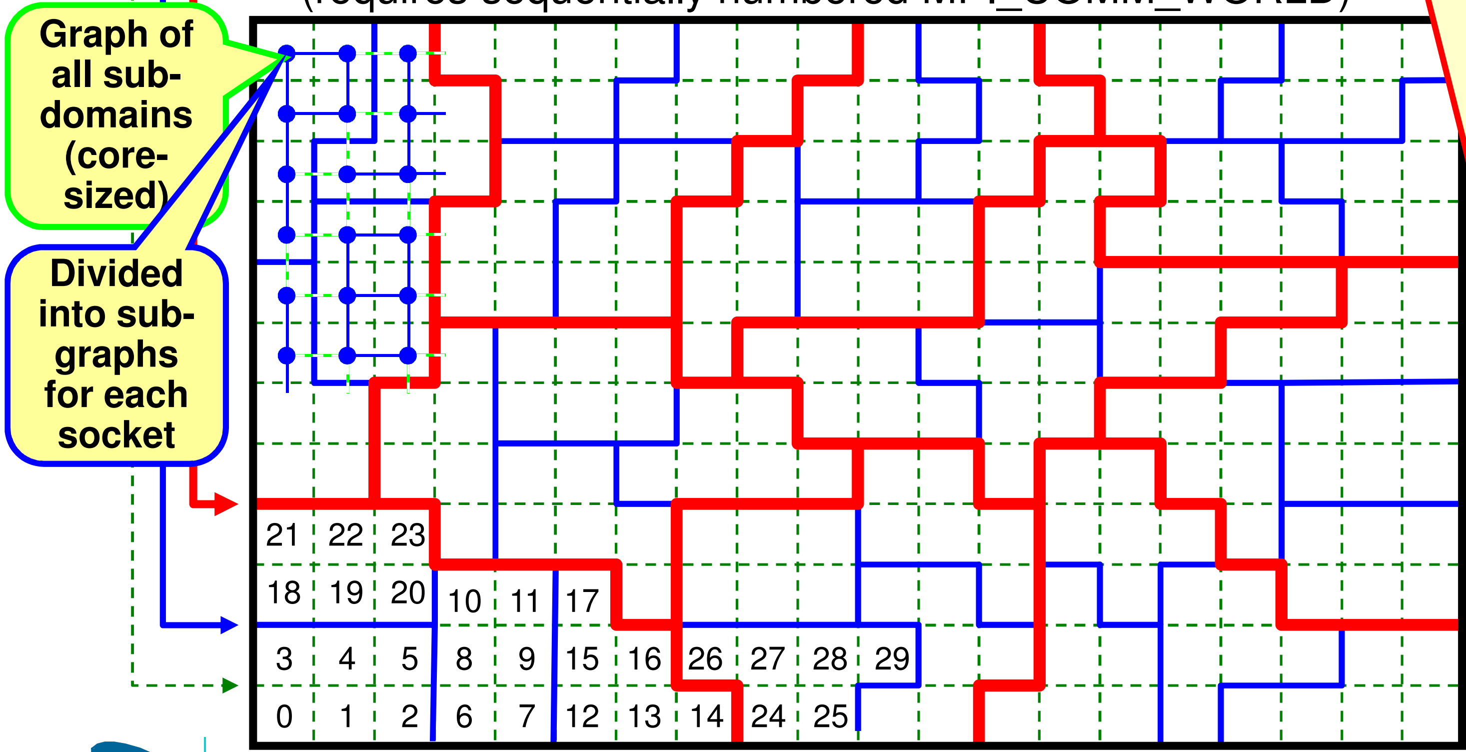


Major result:
New **global communicator**
with
- **minimal node-to-node** &
- **optimal intra-node** communication

See MPI/course/C/1sided/cart_renumber_shared+distr.c
- Compiled with **–D seq** → uses fixed SMP size 16 / otherwise automatic SMP detection

Unstructured Grids –
Multi-level Domain Decomposition through Recombination

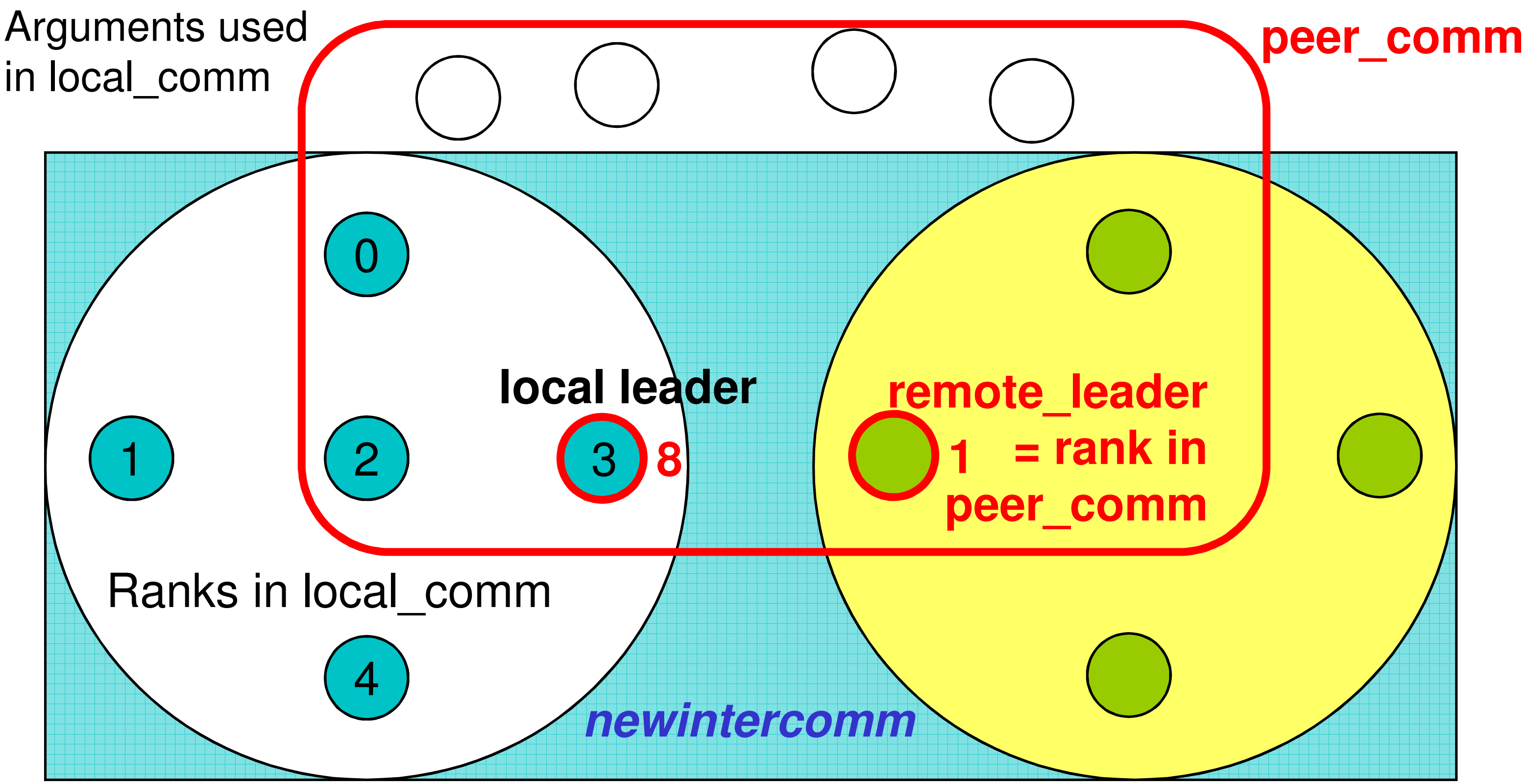
1. **Core-level DD:** partitioning of (large) application's data grid, e.g., with Metis / Scotch
2. **Numa-domain-level DD:** recombining of core-domains
3. **SMP node level DD:** recombining of socket-domains
4. **Numbering** from core to socket to node (requires sequentially numbered MPI_COMM_WORLD)



- **Problem:** Recombination must **not** calculate patches that are smaller or larger than the average
- In this example the load-balancer **must** combine always
  - 6 cores, and
  - 4 numa-domains (i.e., sockets or dies)
- **Advantage:** Communication is balanced!

Inter-communicator – combines a local and a remote comm

- MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, *newintercomm*)



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

MPI_Info Object

- 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 value
  - several keys are reserved by standard / implementation
  - portable programs may use **`MPI_INFO_NULL`** as the info argument, or sets of vendor keys
  - Several sets of vendor-specific keys may be used
- Allows applications to pass environment-specific information
- Several functions provided to manipulate the info objects
- Used in: *Process Creation*,
  Window Creation,
  MPI-I/O,
  MPI_Comm_dup_with_info,
  MPI_INFO_ENV

Naming & attribute caching

Name an object:

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

Caching attributes in two steps:

- First step – generating a keyval:
  - MPI_Comm_create_keyval (comm_copy_attr_fn, comm_delete_attr_fn, *comm_keyval*, extra_state)
- Second step – storing & retrieving an attribute on/from a 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 (comm, comm_keyval)
  - MPI_Comm_free_keyval ([INOUT] *comm_keyval*)

Other objects: Same method for datatypes and windows

Examples: See MPI-3.0 Sect.17.2.7 *Attributes* on pages 653-657

Environment inquiry – implementation information (1)

New in MPI-3.0

Version of MPI

- Compile time information
  - integer MPI_VERSION=3, MPI_SUBVERSION=0
  - Valid pairs: (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

Inquire start environment

- Predefined info object **MPI_INFO_ENV** holds arguments from
  - mpiexec, or
  - MPI_COMM_SPAWN

Inquire processor name

- MPI_Get_processor_name(*name*, *resultlen*)

Environment inquiry – implementation information (2)

<u>Environmental inquiries</u>

- 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*)
- with keyval =
  - **MPI_TAG_UB**

    → returns upper bound for tag values in *attribute_val*

    → must be at least 32767
  - **MPI_HOST**

    → returns host-rank (if exists) or MPI_PROC_NULL (if there is no host)
  - **MPI_IO**

    → returns MPI_ANY_SOUCE 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

<u>Examples:</u> see MPI-3.0, Sect. 17.2.7, page 656, line 43 – page 657, line 13

Error Handling

- 2-level-concept with error codes and error classes, see MPI-3.0 Sect. 8.3-8.5

Most important aspects:

- The communication should be reliable
- If the MPI program is erroneous:
  - by default: abort, if error detected by MPI library
    otherwise, unpredictable behavior
  - Fortran: call MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN, ierr)
    C/C++: MPI_Comm_set_errhandler (comm, MPI_ERRORS_RETURN);
  - e.g., directly after MPI_INIT with comm = MPI_COMM_WORLD, 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)
    - See MPI-3.0 Sect. 13.7, page 550, and course Chap. 14

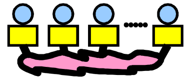
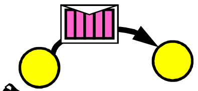
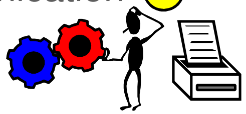
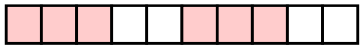
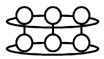
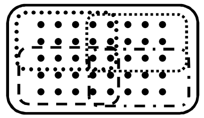
Skip conclusion-slide (1 slide)

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 Chap. 10 *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
- Error Handling
  - Quality as expected for an “assembler for parallel computing”

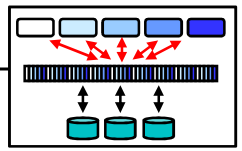
Chap.9 Collective Communication

MPI_Init()
MPI_Comm_rank()

1. MPI Overview 
2. Process model and language bindings 
3. Messages and point-to-point communication
4. Nonblocking communication 
5. Probe, Persistent Requests, Cancel
6. Derived datatypes 
7. Virtual topologies 
8. Groups & communicators, environment management 

9. Collective communication

- **e.g. broadcast**
- **neighborhood communication**

10. Process creation and management
11. One-sided communication
12. Shared memory one-sided communication
13. MPI and threads
14. Parallel file I/O 
15. Other MPI features

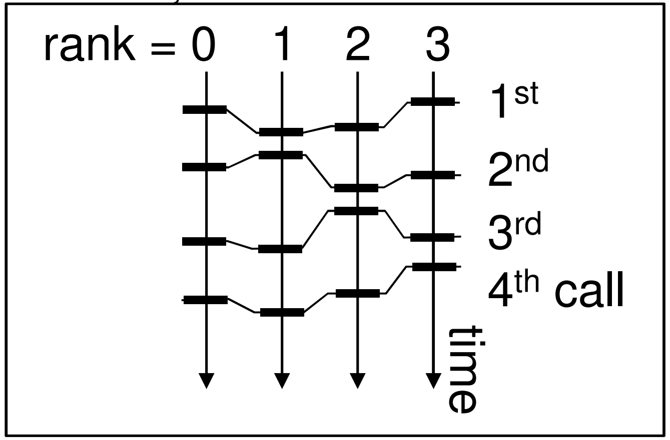
Collective Communication

- 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 grid

New in MPI-3.0

Characteristics of Collective Communication

- Collective action over a communicator.
- All process 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.
- Receive buffers must have exactly the same size as send buffers.



Barrier Synchronization

C

Fortran

- 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

- 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 chapter.**~~

Broadcast

C

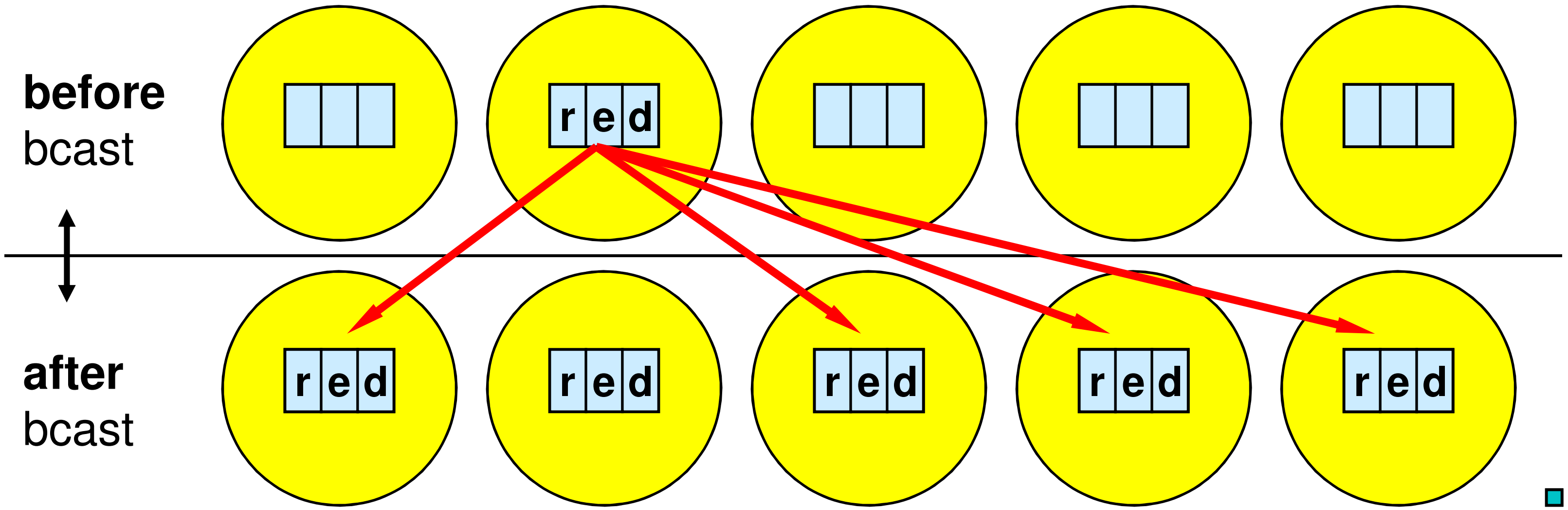
Fortran

- C/C++: int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
  int root, MPI_Comm comm)

- Fortran: MPI_Bcast(buf, count, datatype, root, comm, *ierror*)

  mpi_f08: TYPE(*), DIMENSION(..) :: buf
  TYPE(MPI_Datatype) :: datatype; TYPE(MPI_Comm) :: comm
  INTEGER :: count, root; INTEGER, OPTIONAL :: ierror

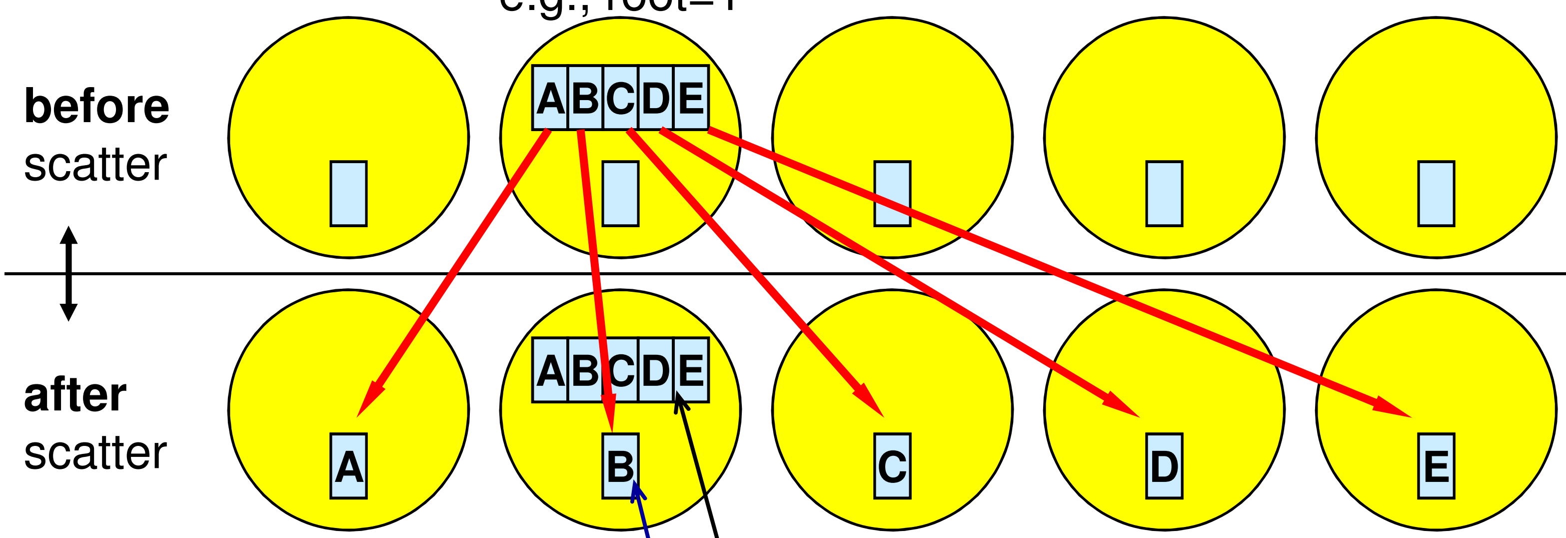
  mpi & mpif.h: <type> buf(*); INTEGER count, datatype, root, comm, ierror



e.g., root=1

- **rank of the sending process (i.e., root process)**
- **must be given identically by all processes**

Scatter



- C/C++: int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void ****recvbuf***, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)

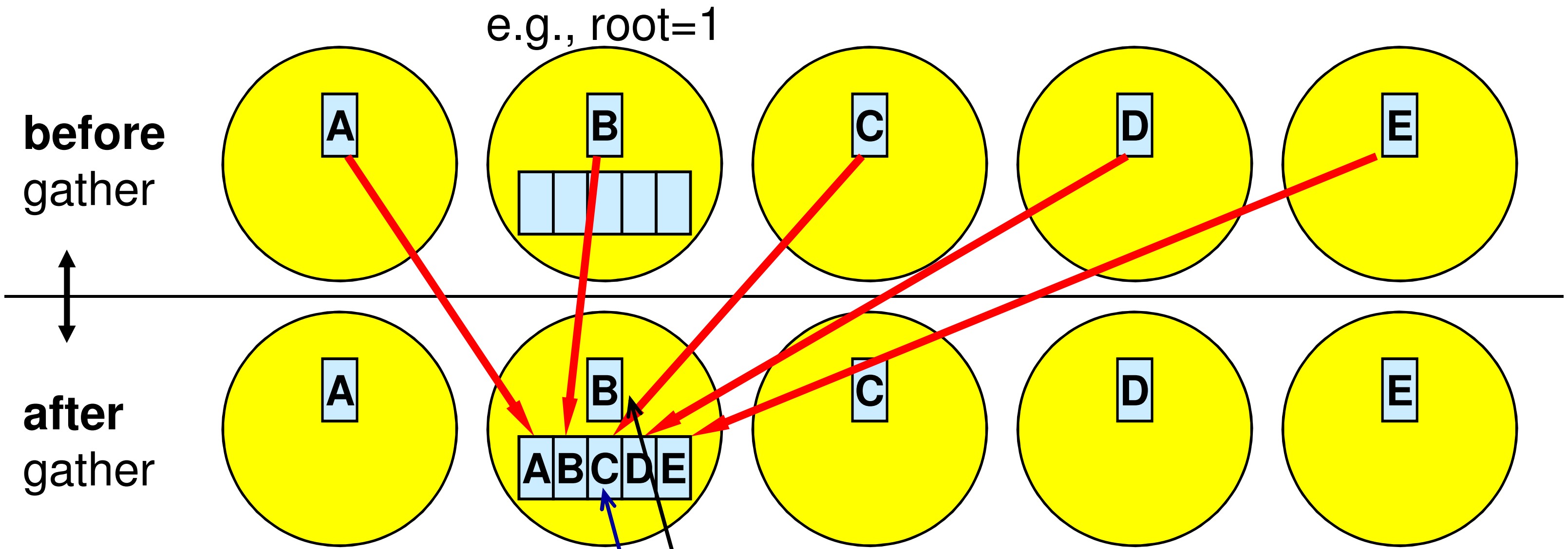
- Fortran: MPI_SCATTER(sendbuf, sendcount, sendtype,
  recvbuf, recvcount, recvtype, root, comm, ***ierror***)

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

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

Example:
MPI_Scatter(sbuf, 1, MPI_CHAR, *rbuf*, 1, MPI_CHAR, 1, MPI_COMM_WORLD)

Gather



- C/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_Comm) :: comm
TYPE(MPI_Datatype) :: sendtype, recvtype; INTEGER, OPTIONAL :: ierror

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

Example:
MPI_Gather(sbuf, 1, MPI_CHAR, *rbuf*, 1, MPI_CHAR, 1, MPI_COMM_WORLD)

Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- d_0 **o** d_1 **o** d_2 **o** d_3 **o** … **o** d_{s-2} **o** d_{s-1}
  - d_i = data in process rank i
    - **single variable, or**
    - **vector**
  - **o** = 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:
  - $[(d_0 \circ d_1) \circ (d_2 \circ d_3)] \circ [\ldots \circ (d_{s-2} \circ d_{s-1})]$
  - $((((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \ldots) \circ d_{s-2}) \circ d_{s-1})$
  - May be even worse through partial sums in each process:
    $\sum_{i=0}^{n-1} x_i \rightarrow [[[(\sum_{i=0}^{n/s-1} x_i \circ \sum_{i=n/s}^{2n/s-1} x_i) \circ (\ldots \circ \ldots)] \circ [\ldots \circ (\ldots \circ \ldots)]]]$

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);

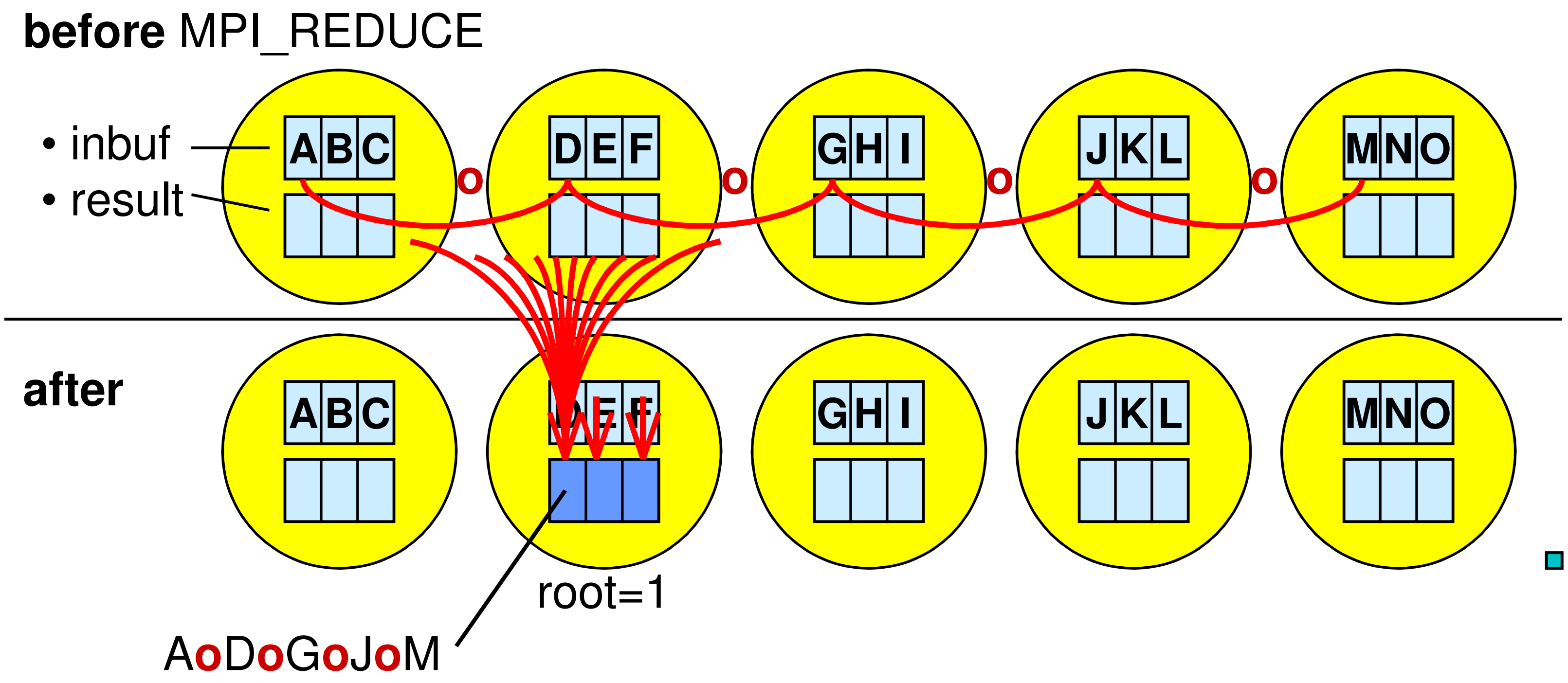
Fortran

- Fortran: root=0
  MPI_REDUCE(inbuf, *resultbuf*, 1, MPI_INTEGER, MPI_SUM,
  root, MPI_COMM_WORLD, *IERROR*)
- The result is only placed in *resultbuf* at the root process.

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
A B C
D E F
G H I
J K L
M N O
o
o
o
o
after
A B C
D E F
G H I
J K L
M N O
root=1
AoDoGoJoM

User-Defined Reduction Operations

- Operator handles
  - predefined – see table above
  - user-defined
- User-defined operation ■:
  - associative
  - user-defined function must perform the operation vector_A ■ vector_B
  - syntax of the user-defined function → MPI standard
- Registering a user-defined reduction function:

C

  - C/C++: MPI_Op_create(MPI_User_function *func, int commute, MPI_Op **op*)

Fortran

  - Fortran: MPI_OP_CREATE(FUNC, COMMUTE, *OP*, *IERROR*)
- COMMUTE tells the MPI library whether FUNC is commutative.

Variants of Reduction Operations

- MPI_ALLREDUCE
  - no root,
  - returns the result in all processes

New in MPI-2.2

- MPI_REDUCE_SCATTER_BLOCK and MPI_REDUCE_SCATTER
  - result vector of the reduction operation is scattered to the processes into the real result buffers

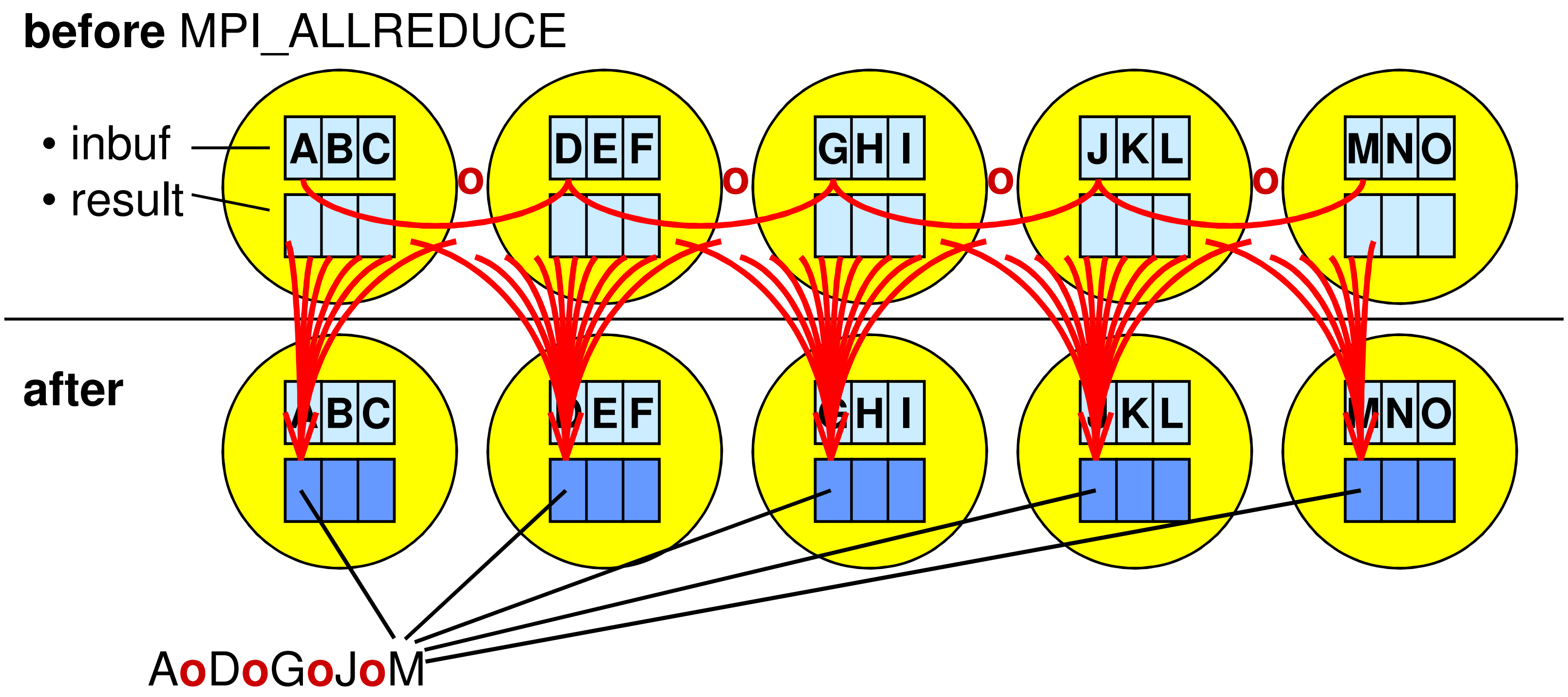
A1	B1	C1
A2	B2	C2
A3	B3	C3

$\Longrightarrow$

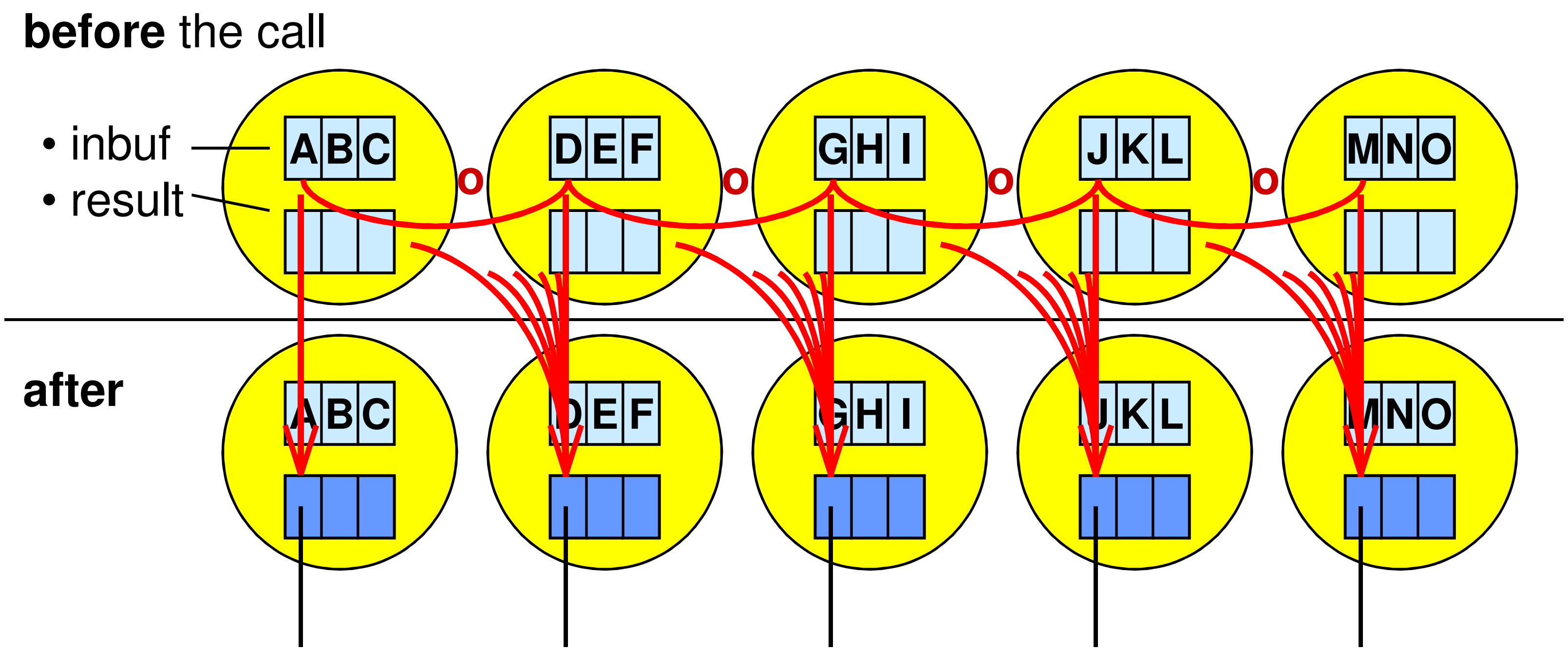
Σ A
Σ B
Σ C

- 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



MPI_SCAN and MPI_EXSCAN

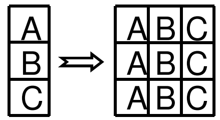


MPI_SCAN:	A	AoD	AoDoG	AoDoGoJ	AoDoGoJoM
MPI_EXSCAN:	-	A	AoD	AoDoG	AoDoGoJ

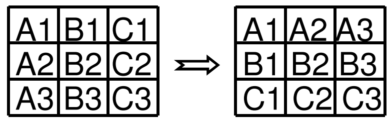
done in parallel

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)

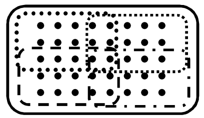
  → interface does **not scale** to thousands of MPI processes!

  → Recommendation: One should try to use data structures with same communication size on all ranks.

Nonblocking Collective Communication Routines

New in MPI-3.0

- MPI_I......... **Nonblocking** variants of all collective communication:
  MPI_Ibarrier, MPI_Ibcast, …
- Collective initiation and completion are separated
- May have multiple outstanding collective communications on same communicator
- Ordered initialization on each communicator
- Offers opportunity to overlap
  - several collective communications,
    e.g., on several overlapping communicators
    - **Without deadlocks or serializations!**



  - computation and communication

    → Often a background MPI progress engine is missing or not efficient

    → Alternative:
    - Several calls to MPI_Test(), which enables progress
    - Use non-standard extensions to switch on asynchronous progress
      - export MPICH_ASYNC_PROGRESS=1

Implies a helper thread and MPI_THREAD_MULTIPLE, see Chapter 13. MPI and Threads

- Parallel MPI I/O:
  The split collective interface may be substituted in the next version of MPI

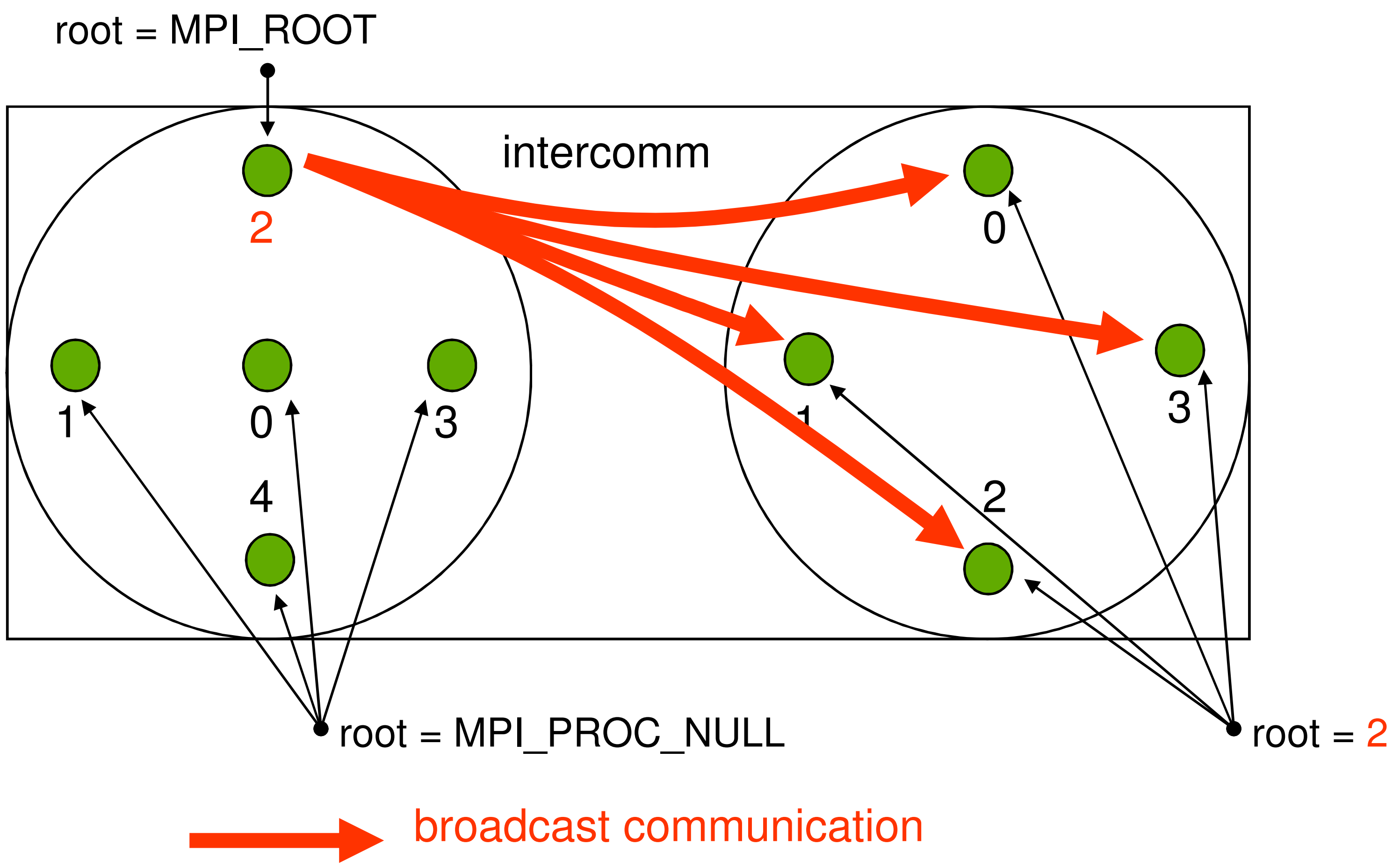
Collective Operations for Intercommunicators

- 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)

2 slides on intercomm-details are skipped / step to these slides: step

skipped

Extended Collective Operations — MPI_Bcast on *inter*comm.

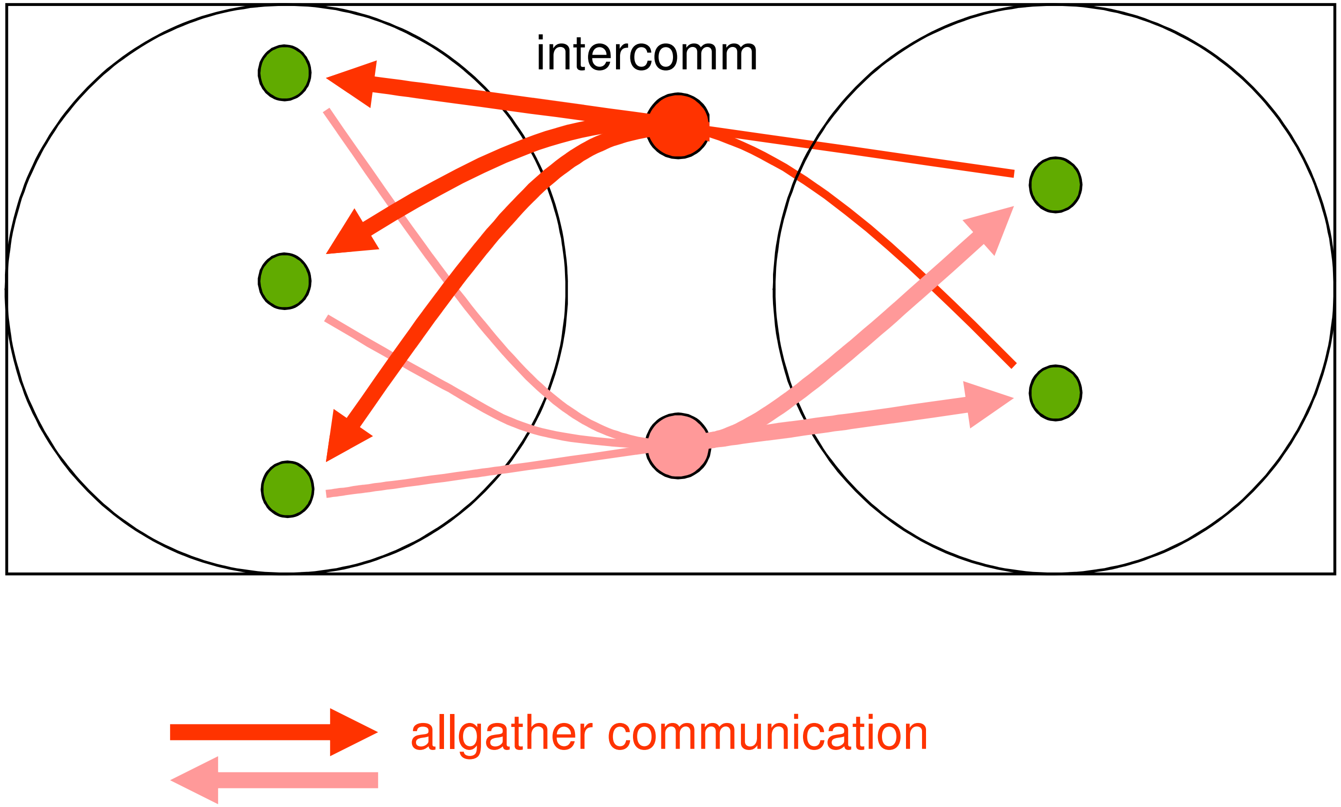


Rolf Rabenseifner

— skipped —

Extended Collective Operations — MPI_Allgather on *inter*comm.

remember: allgather = gather+broadcast



Extended Collective Operations — "In place" Buffer Specification

The **`MPI_IN_PLACE`** has two meanings:

- to **prohibit the local copy** with
  → sendbuf=**`MPI_IN_PLACE`**:
  - (I)GATHER(V) at root process
  - (I)ALLGATHER(V) at all processes

  → recvbuf=**`MPI_IN_PLACE`**:
  - (I)SCATTER(V) at root process
- to **overwrite input buffer** with the result:
  (sendbuf=**`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)

New in MPI-3.0

Sparse Collective Operations on Process Topologies

- 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:
    - **dim=0 source, dim=0 dest, dim=1 source, dim=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 advanced exercise No. 3

Exercise — 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.

- Use the results from Chap. 4:
  - ~/MPI/course/**F_30**/Ch4/ring_30.f90 (with mpi_f08 module)
  - ~/MPI/course/**F_20**/Ch4/ring_20.f90 (with mpi module)
  - ~/MPI/course/**C**/Ch4/ring.c

- I.e., the pass-around-the-ring communication loop must be totally substituted by one call to the MPI collective reduction routine.

Fortran

C

see also login-slides

Sol.

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().
   – Pipe the stdout through **sort –n** to get the output sorted by the ranks:

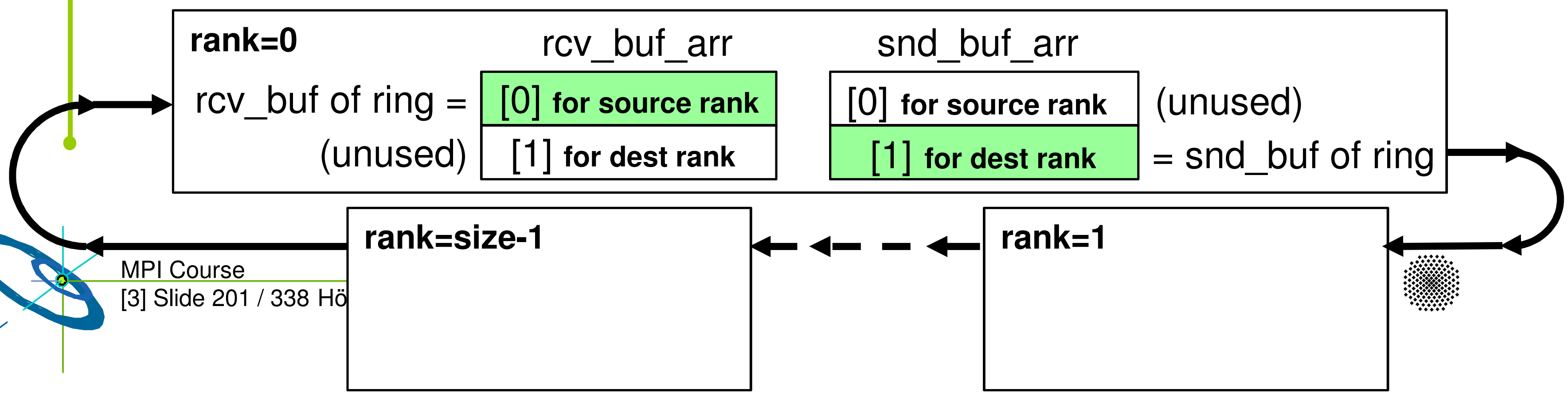
     rank= 0 → sum=0
     rank= 1 → sum=1
     rank= 2 → sum=3
     rank= 3 → sum=6
     rank= 4 → sum=10

2. Global sum in sub-groups:
   – Rewrite the result of the advanced exercise of course Chap. 7.
   – Compute the sum in each slice with the global reduction.

New in MPI-3.0

Advanced Exercises — Neighbor Collective Communicat.

3. Keep the ring communication in the virtual topology example, but substitute the point-to-point communication by neighborhood collective:
   - Use the results from course Chap. 7 *Virtual Topologies*:
     ~/MPI/course/**F_30**/Ch7/topology_ring_30.f90 (with mpi_f08 module)
     ~/MPI/course/**F_20**/Ch7/topology_ring_20.f90 (with mpi module)
     ~/MPI/course/**C**/Ch7/topology_ring.c
   - I.e., Isend-Recv-Wait → one call to MPI_Neighbor_alltoall
   - rcv_buf and snd_buf must be combined into a 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.



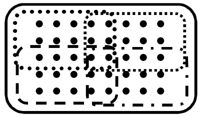
For private notes

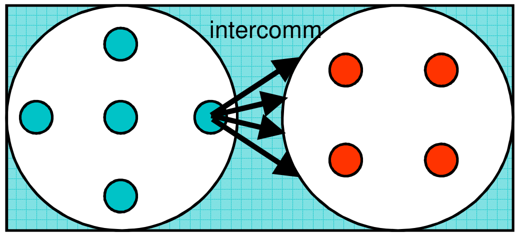
- private notes

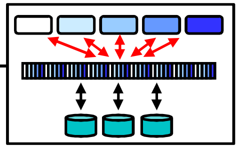
For private notes

Chap.10 Process Creation and Management

MPI_Init()
MPI_Comm_rank()





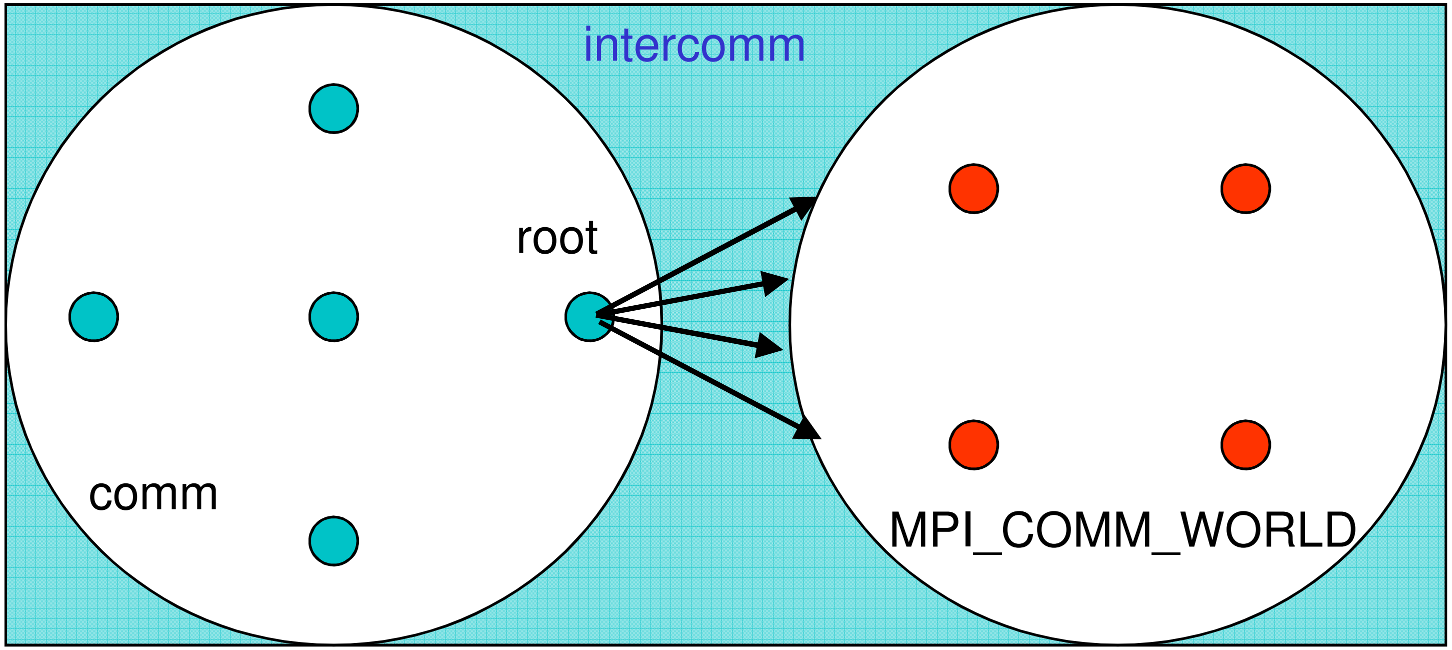


Skip this section (4 slides + 4 hidden slides)

Dynamic Process Management

- Two independent goals
  1. starting new MPI processes
  2. connecting independently started MPI processes
- Issues
  - maintaining simplicity, flexibility, and correctness
  - interaction with operating systems, resource manager, and process manager
- Starting new 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**

Dynamic Process Management — Get the intercomm, I.



Parents:

MPI_COMM_SPAWN (.....,
root,comm, *intercomm*,...)

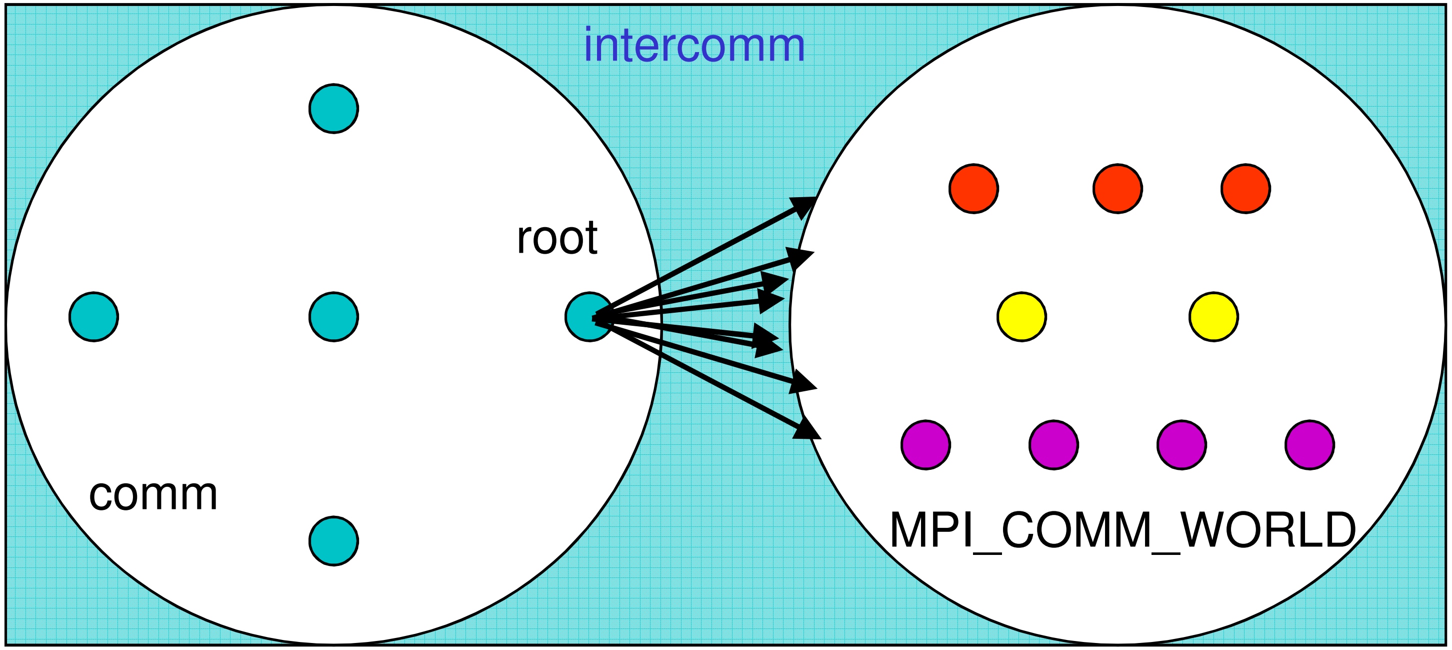
Children:

MPI_Init(...)
MPI_COMM_GET_PARENT(*intercomm*)

1 slides on MPI_Comm_spawn_multiple is skipped / step to this slides: step

— skipped —

Dynamic Process Management — Get the intercomm, II.



Parents:

MPI_COMM_SPAWN
_MULTIPLE (3, ...,
root,comm, *intercomm*,...)

Children:

MPI_Init(...)
MPI_COMM_GET_PARENT(*intercomm*)

HLRS

step

Major Problem with Spawning of Dynamic Processes

- **Typical static environment**
  - MPI processes within a batch job (→ 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

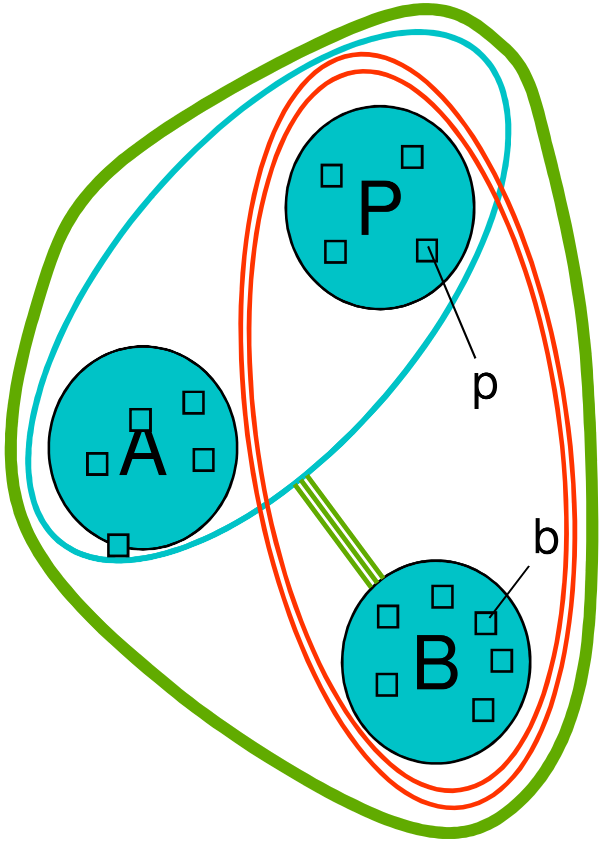
→ **In most cases not useful**

3 slides on MPI_Intercomm_merge, MPI_Comm_connect, MPI_Comm_join are skipped / step to these slides: step

— skipped —

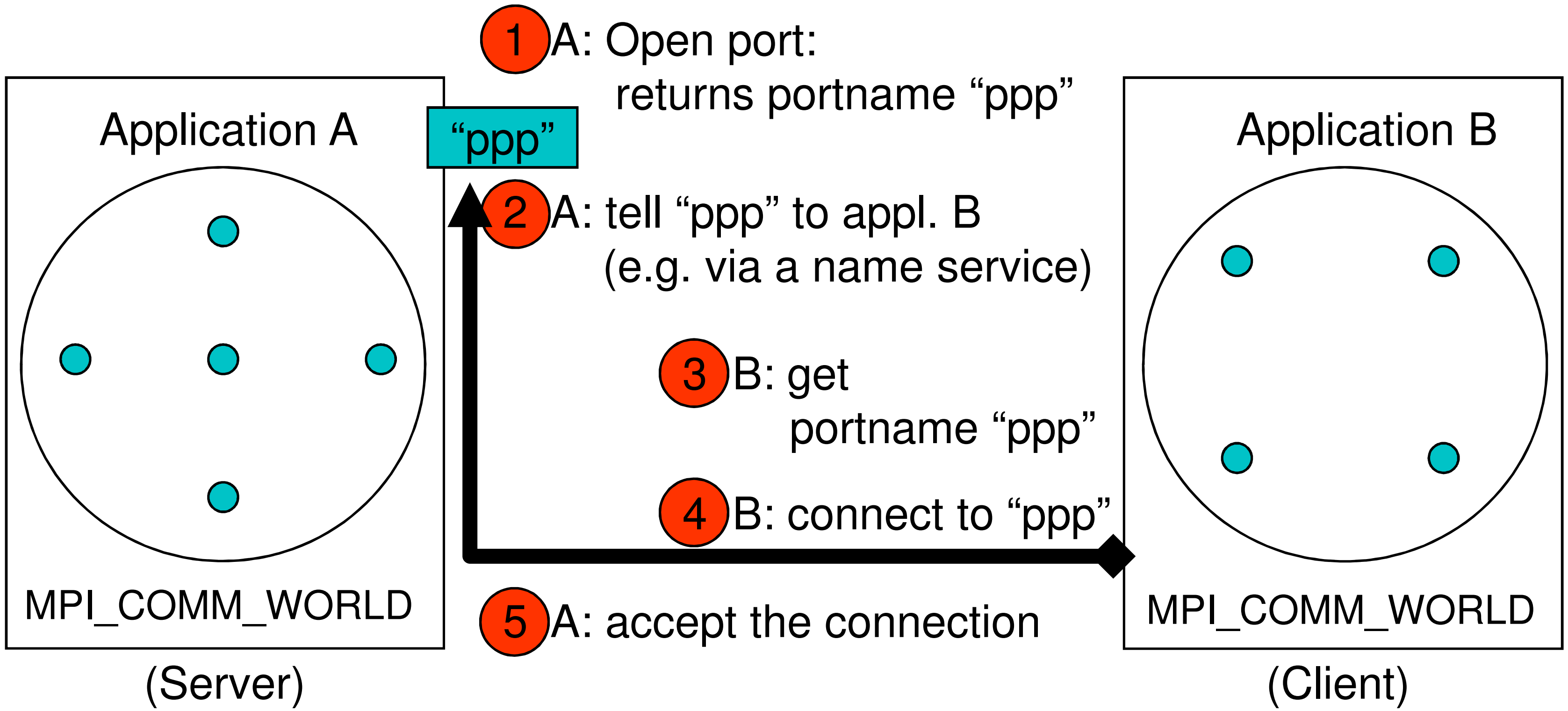
Dynamic Process Management — Multi-merging, a Challenge

- If a comm. 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

Dynamic Process Management — Singleton INIT

- High quality MPI's will allow single processes to start, call MPI_INIT(), and later join in with other MPI programs
- 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.

For private notes

For private notes

For private notes

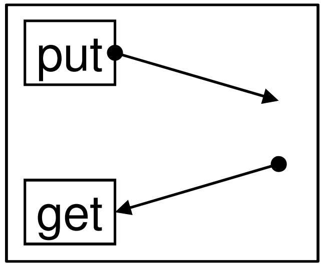
Chap.11 One-sided Communication

MPI_Init()
MPI_Comm_rank()

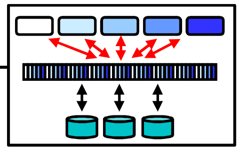
1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. Probe, Persistent Requests, Cancel
6. Derived datatypes
7. Virtual topologies
8. Groups & communicators, environment management
9. Collective communication
10. Process creation and management

11. One-sided communication

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



12. Shared memory one-sided communication
13. MPI and threads
14. Parallel file I/O
15. Other MPI features



Two skip-points: 1st after 1 slide / 2nd after 11 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

1st skip-point: Skip rest of this section

Synchronization Taxonomy

Message Passing:

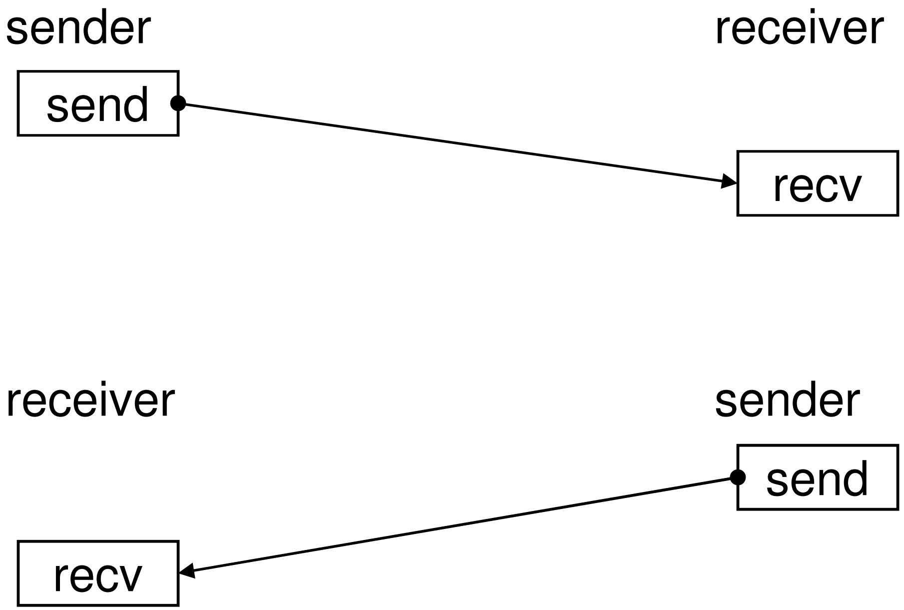
explicit transfer, implicit synchronization,
implicit cache operations

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

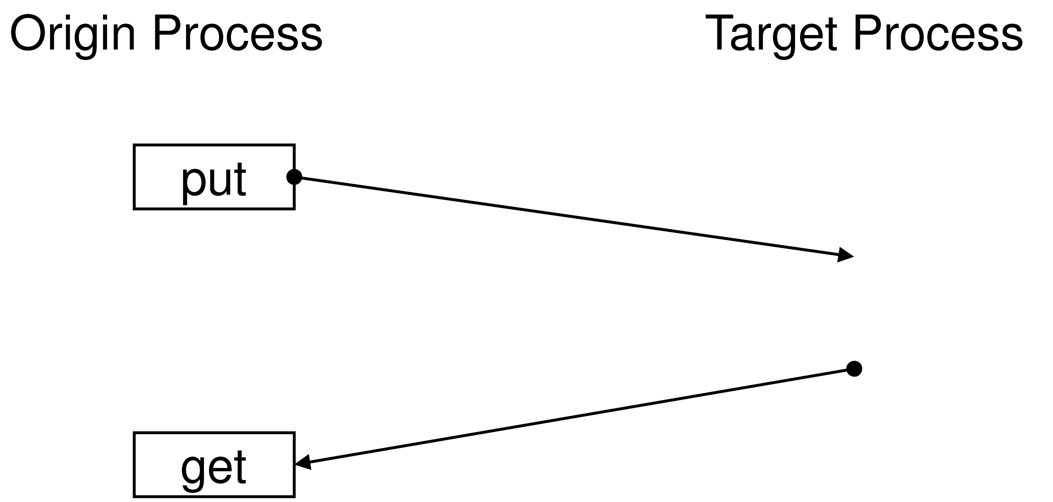
Cooperative Communication

- 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



One-sided Operations

<u>Three major set of routines:</u>

- 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 access by all other processes of the group.
- **R**emote **M**emory **A**ccess (RMA, nonblocking) routines
  - Access to remote windows:
    - **put, get, accumulate, …**
- Synchronization
  - The RMA routines are nonblocking and
  - must be surrounded by synchronization routines,
  - which guarantee
    - **that the RMA is locally and remote 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

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

New in MPI-3.0

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**

RMA Operations

- Nonblocking RMA routines
  - that are finished by subsequent window synchronization
    - **MPI_Get**
    - **MPI_Put**
    - **MPI_Accumulate**
    - **MPI_Get_accumulate**
    - **MPI_Fetch_and_op**
    - **MPI_Compare_and_swap**
  - that are finished with regular MPI_Wait, …
    - **MPI_Rget**
    - **MPI_Rput**
    - **MPI_Raccumulate**
    - **MPI_Rget_accumulate**

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

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.

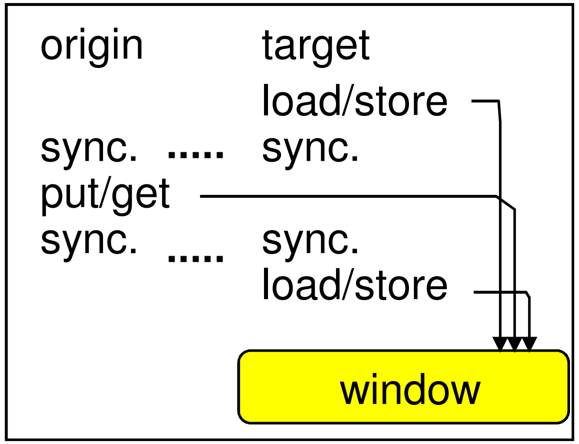
Substitute target element by origin buffer element if target element == compare buffer element.

New in MPI-3.0

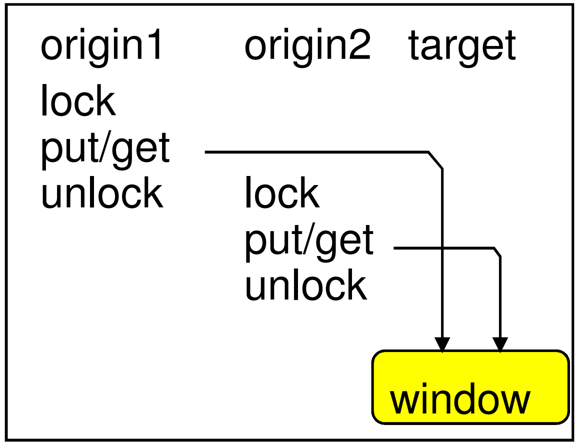
R = request-based

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, (New in MPI-3.0)
  - MPI_Win_flush(_all), MPI_Win_flush_local(_all), MPI_Win_sync (New in MPI-3.0)

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 without 1-sised: MPI_ALLTOALL is needed
  - Each sender tells all processes whether they will get a message or not.
- Solution with 1-sided communication:
  - Each process in the role being a receiver:
    - **MPI_Win_create(&nsp, ...); nsp=0;** (i.e., I do not yet know the number of my sending neighbors)
  - Each process as a sender tells the receiver "here is 1 neighbor from you"
    - **MPI_Win_fence**
    - **Multiple calls to MPI_Accumulate to add 1 in the nsp of its neighbors.**
    - **MPI_Win_fence**
  - Now, each process as a receiver knows in its nsp the number of its neighbors. Therefore:
    - **Loop over nsp with MPI_Irecv(MPI_ANY_SOURCE)**
  - Each process as a sender
    - **Loop over its neigbors, sending the data.**
  - As receiver: **MPI_Waitall()** – in the statuses array, the receiver can see the neighbor's ranks

2nd skip-point: Skip rest of this section

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

byte size, MPI_Aint

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

byte size, int

Fortran C/C++ language bindings – see MPI-2 Standard

- Only in the mpi module and mpif.h
- In mpi_f08: 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)
```

New in MPI-3.0

In all three Fortran support methods

All Memory Allocation with modern C-Pointer

C

```
float *buf;  MPI_Win win; int max_length; max_length = …;
MPI_Win_allocate( (MPI_Aint)(max_length*sizeof(float)),  sizeof(float),
                  MPI_INFO_NULL,  MPI_COMM_WORLD,  &buf,  &win);
```

Fortran

```
USE mpi_f08
USE, INTRINSIC :: ISO_C_BINDING
INTEGER :: max_length,  disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: lb, size_of_real
REAL, POINTER, ASYNCHRONOUS :: buf(:)
TYPE(MPI_Win) :: win
INTEGER(KIND=MPI_ADDRESS_KIND) :: buf_size, target_disp
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/) )
```

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
- The target buffer is at address
  $\text{target\_addr} = \text{win\_base}_{\textbf{target\_process}}$
  $+ \text{target\_disp}_{\textbf{origin\_process}} * \text{disp\_unit}_{\textbf{target\_process}}$

MPI_PUT(origin_address, origin_count, origin_datatype,
target_rank, *target*_disp$_{\text{origin}}$, target_count,
target_datatype, win)

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

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

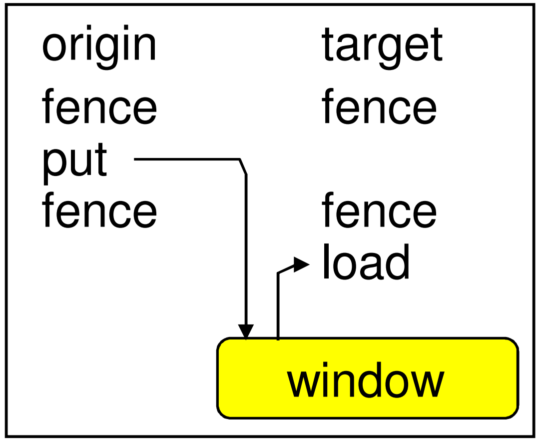
- Accumulates the contents of the origin buffer to the target area specified using the predefined operation `op`
- User-defined operations cannot be used
- Accumulate is atomic: many accumulates can be done by many origins to one target
  -> [*may be very expensive*]

```
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

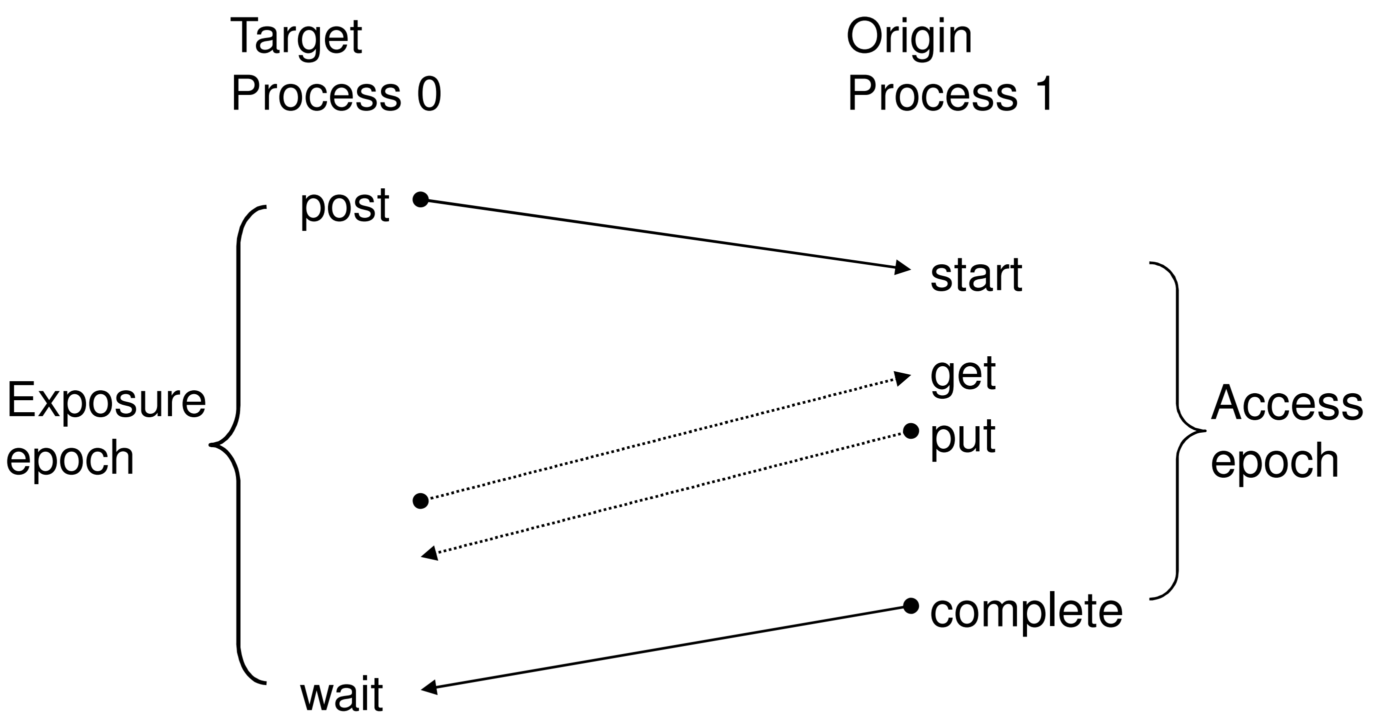
- Synchronizes RMA operations on specified window
- Collective over the window
- Like a barrier
- Should be used before and after calls to put, get, and accumulate
- The `assert` argument is used to provide optimization hints to the implementation
- Used for active target communication




```
MPI_WIN_FENCE(assert, win)
```

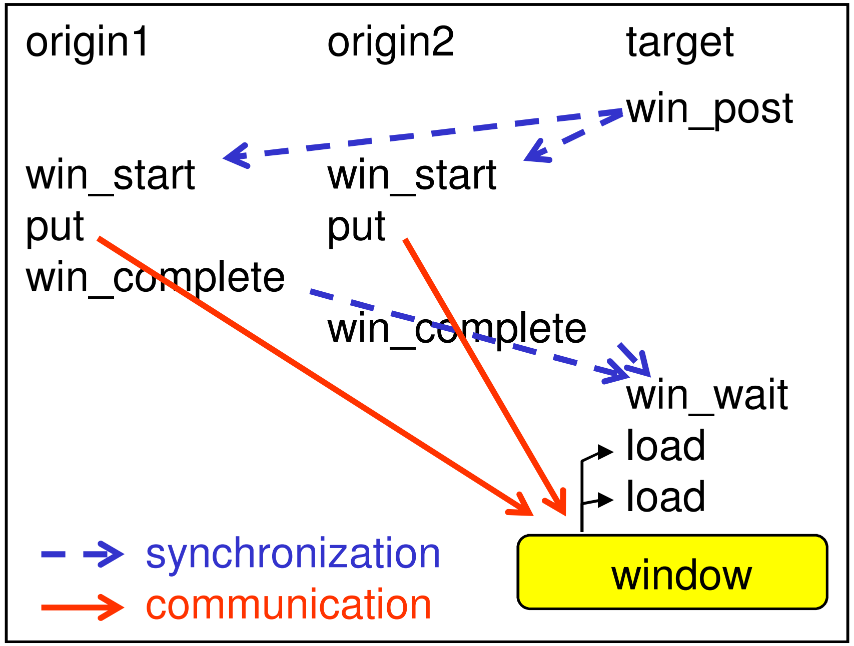
Start/Complete and Post/Wait, I.

- Used for active target communication with weak synchronization



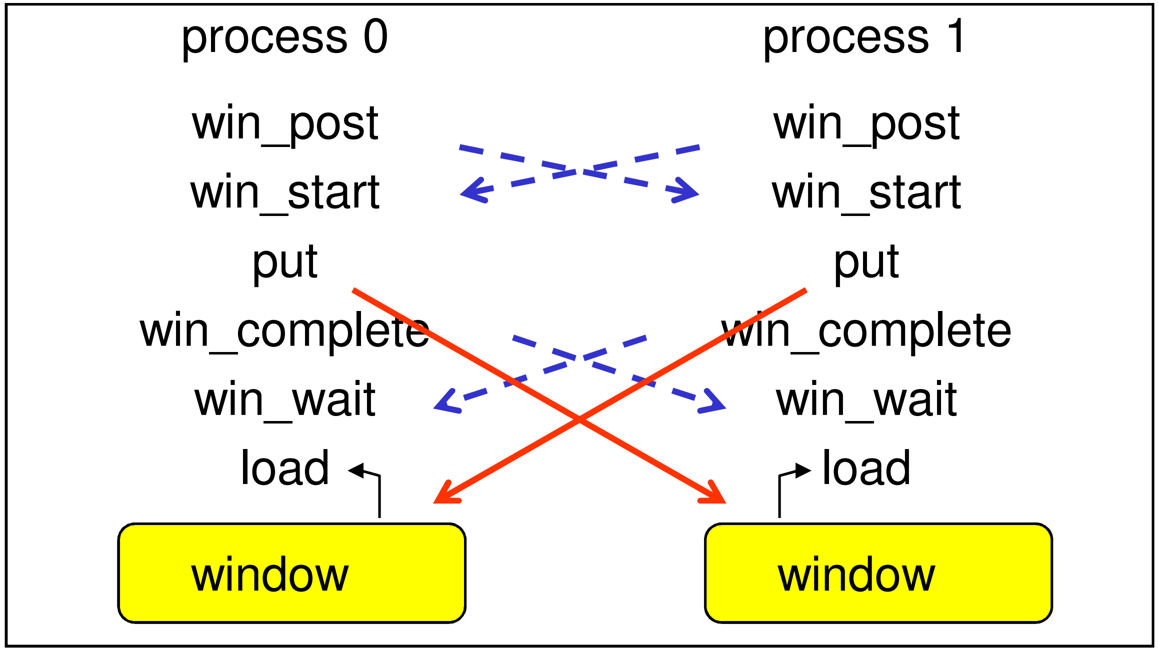
Start/Complete and Post/Wait, II.

- 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



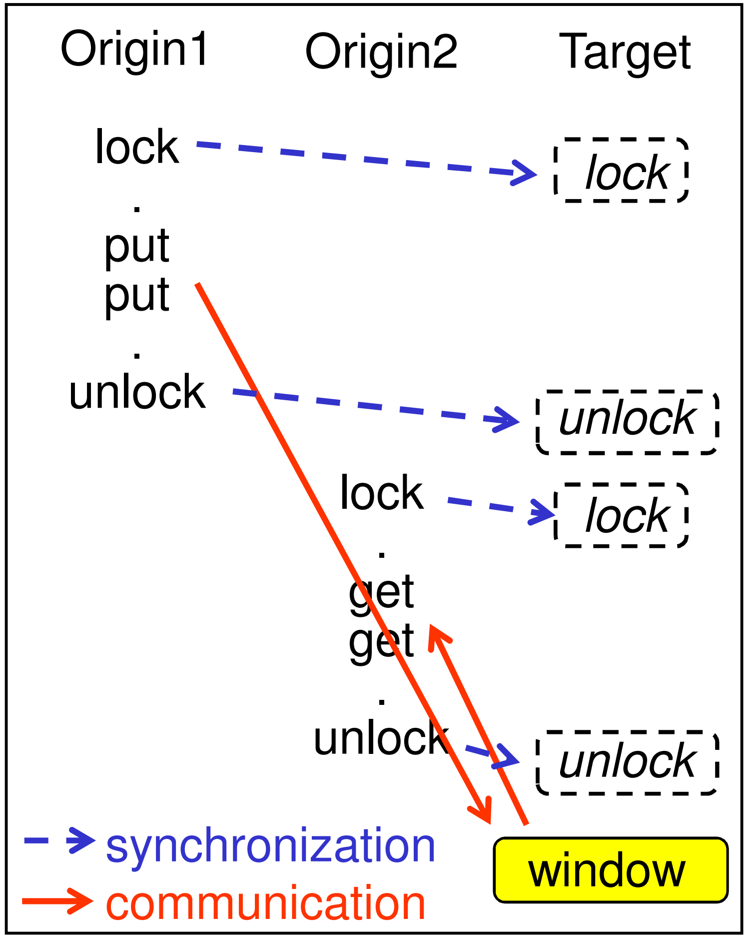
Start/Complete and Post/Wait, III.

- symmetric communication possible,
  only win_start and win_wait may block



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`**
- RMA completed after **`UNLOCK`** at both origin and target



Fortran Problems with 1-Sided

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

- Fortran register optimization
- Result: 999 is printed instead of expected 777
- How to avoid: (see MPI-2.2, Chap. 11.7.3, pp. 371f)
  - 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

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 operations
- Performance-opportunities depend largely on the quality of the MPI library
  - See also halo example in next course chapter

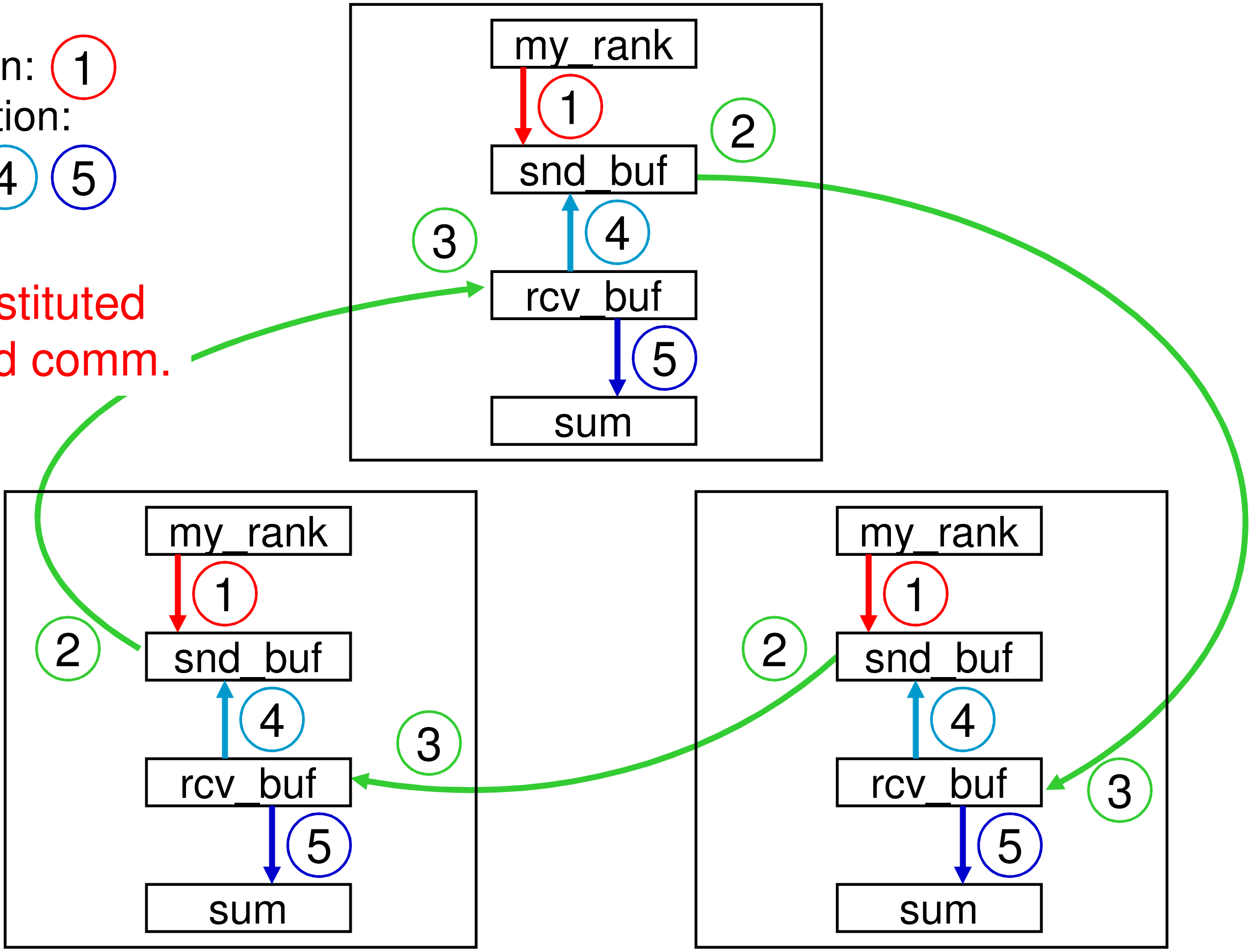
MPI–One-sided Exercise 1: Ring communication with fence

- Copy to your local directory:
  cp ~/MPI/course/**C**/1sided/ring.c **my_1sided_exa1.c** *(or copy your Chap-4-result)*
  cp ~/MPI/course/**F_30**/1sided/ring_1sided_skel_30.f90 **my_1sided_exa1_30.f90** *or*
  cp ~/MPI/course/**F_20**/1sided/ring_1sided_skel_20.f90 **my_1sided_exa1_20.f90**
- Tasks:
  - 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
  - Compile and run your `my_1sided_exa1.c / .f90`

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

Initialization: 1
Each iteration:
2 3 4 5

to be substituted
by 1-sided comm.



MPI–One-sided Exercise 1: additional hints

- MPI_Win_create:
  - base = reference to your rcv_buf or snd_buf variable
  - disp_unit = number of bytes of one int / integer, because this is the datatype of the buffer (=window)
  - size = same number of bytes, because buffer size = 1 value
  - size and disp_unit have different internal representations, therefore:
    - **C/C++:** C
```
MPI_Win_create(&rcv_buf, (MPI_Aint) sizeof(int), sizeof(int),
               MPI_INFO_NULL, ..., &win);
```
    - **Fortran:** Fortran
```
INTEGER disp_unit
INTEGER (KIND=MPI_ADDRESS_KIND) winsize, lb, extent
CALL MPI_TYPE_GET_EXTENT(MPI_INTEGER, lb, extent, ierror)
disp_unit = extent
winsize = disp_unit * 1
CALL MPI_WIN_CREATE(rcv_buf, winsize, disp_unit, MPI_INFO_NULL, ...,
                    ierror)
```
- see MPI-3.0, Sect. 11.2.1, pages 404ff

MPI–One-sided Exercise 1: additional hints

C

Fortran

- MPI_Put (or MPI_Get):
  - target_disp
    - **C/C++:** MPI_Put(&snd_buf, 1, MPI_INT, right**, (MPI_Aint) 0**, 1, MPI_INT, win);
    - **Fortran:** **INTEGER (KIND=MPI_ADDRESS_KIND) target_disp**
      target_disp = 0
      CALL MPI_PUT(snd_buf, 1, MPI_INTEGER, right, **target_disp**, 1,
      MPI_INTEGER, win, ierror)
  - Register problem with Fortran:
    - **Access to the rcv_buf before 1st and after 2nd MPI_WIN_FENCE:**
      INTEGER (KIND=MPI_ADDRESS_KIND) idummy_addr
      CALL MPI_GET_ADDRESS(rcv_buf, idummy_addr, ierror)

      or with MPI-3.0:
      …, ASYNCHRONOUS :: rcv_buf
      IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) &
      & CALL MPI_F_SYNC_REG(rcv_buf)
- see MPI-3.0, Sect. 11.3.1, pages 419f
  and Sect. 17.1.10-17.1.19, pages 624-642

see also login-slides

Sol.

MPI–One-sided Exercise 2: Post-start-complete-wait

Exercise 2

- Use your result of exercise 1 or copy to your local directory:
  cp ~/MPI/course/**C**/1sided/ring_1sided_put.c **my_1sided_exa2.c**
  cp ~/MPI/course/**F_30**/1sided/ring_1sided_put_30.f90 **my_1sided_exa2_30.f90**
  or
  cp ~/MPI/course/**F_20**/1sided/ring_1sided_put_20.f90 **my_1sided_exa2_20.f90**
- Tasks:
  – Substitute the two calls to MPI_Win_fence
    by calls to MPI_Win_post / _start / _complete / _wait
  – Use to group mechanism to address the neighbors:
    - **MPI_COMM_GROUP(comm, *group*)**
    - **MPI_GROUP_INCL(group, n, ranks, *newgroup*)**
      – do not forget **ierror** with Fortran!
      – Fortran: integer comm, group, newgroup, n, ranks(...)
      – C: MPI_Comm comm; MPI_Group group, newgroup; int n, ranks[...];
  – Compile and run your `my_1sided_exa2.c / .f`

see also login-slides

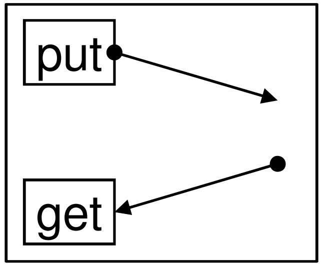
For private notes

Chap.12 Shared Memory One-sided Communication

MPI_Init()
MPI_Comm_rank()

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. Probe, Persistent Requests, Cancel
6. Derived datatypes
7. Virtual topologies
8. Groups & communicators, environment management
9. Collective communication
10. Process creation and management
11. One-sided communication

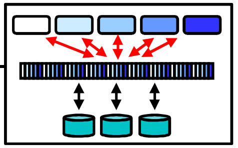
New in MPI-3.0



12. Shared memory one-sided communication

- **MPI_Comm_split_type & MPI_Win_allocate_shared**
- **Hybrid MPI and MPI-3 shared memory programming**

13. MPI and threads
14. Parallel file I/O
15. Other MPI features



One skip-point: after 2 slides (total: 15+(1) talk + 4 exercise-slides)

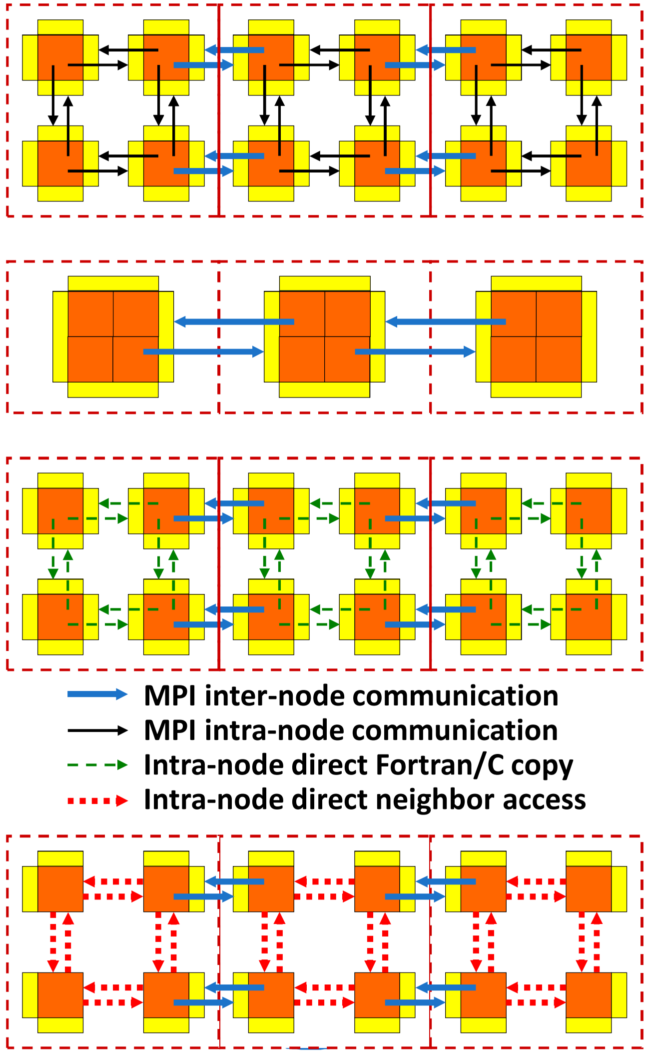
MPI-3 shared memory

- Split main communicator into shared memory islands
  - **MPI_Comm_split_type**
- Define a shared memory window on each island
  - **MPI_Win_allocate_shared**
  - Result (by default):
    contiguous array, directly accessible by all processes of the island
- Accesses and synchronization
  - Normal assignments and expressions
  - No **MPI_PUT/GET** !
  - Normal MPI one-sided synchronization, e.g., **MPI_WIN_FENCE**

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

Hybrid shared/cluster programming models

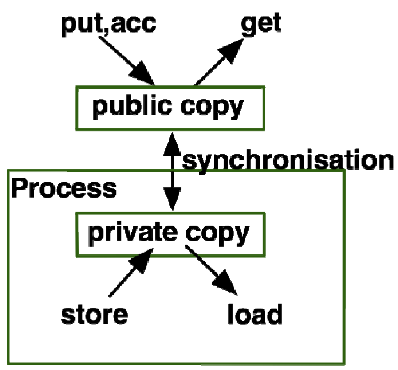
- MPI on each core (not hybrid)
  - Halos between all cores
  - MPI uses internally shared memory and cluster communication protocols
- MPI+OpenMP
  - Multi-threaded MPI processes
  - Halos communica. only between MPI processes
- new MPI cluster communication + MPI shared memory communication
  - Same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



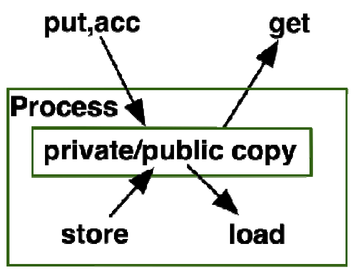
Skip rest of this section

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**

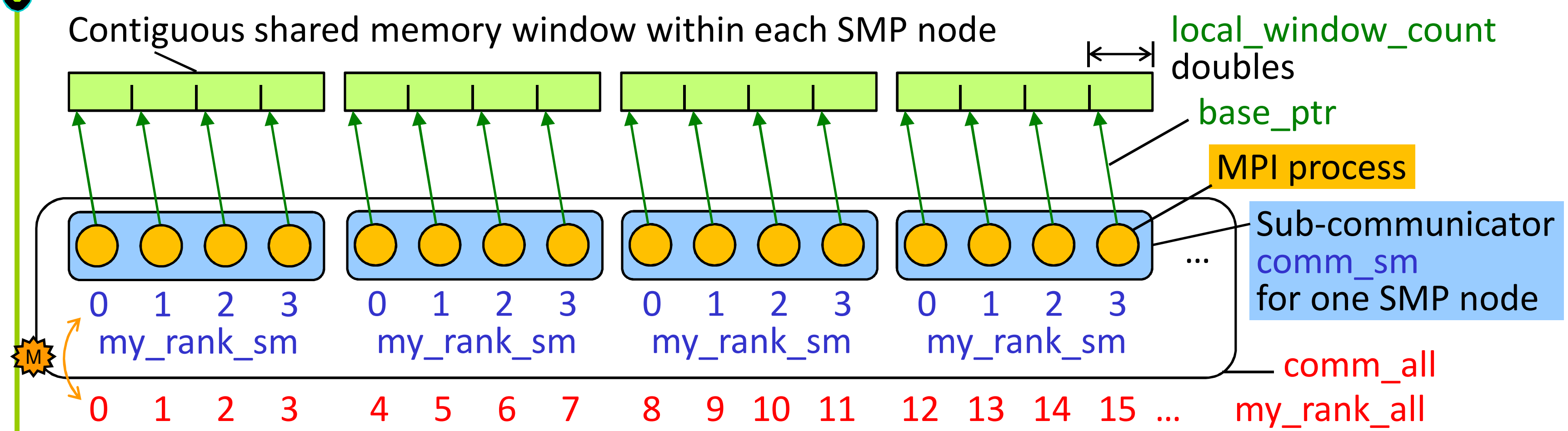


- Shared memory windows always use the MPI_WIN_UNIFIED model
  - Public and private copies are **eventually** synchronized at a byte-level granularity without additional RMA calls **(MPI-3.1 errata ticket #456)**
  - For synchronization **without delay**: MPI_WIN_SYNC() **(MPI-3.1 errata ticket #456)** (also included in other RMA synchronization)



Figures: Courtesy of Torsten Hoefler

Splitting the communicator & contiguous shared memory allocation




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

Sequence in comm_sm as in comm_all

F In Fortran, MPI-3.0, page 341, Examples 8.1 (and 8.2) show how to convert buf_ptr into a usable array a.

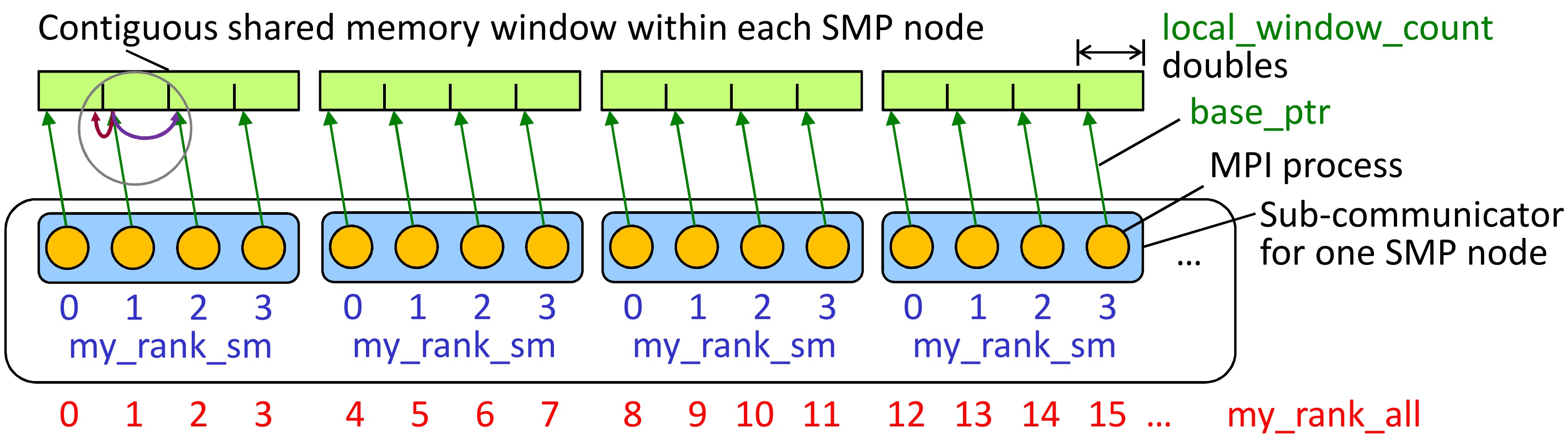
M This mapping is based on a sequential ranking of the SMP nodes in comm_all.

Within each SMP node – Essentials

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

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

Shared memory access example




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

Synchroni-zation

Synchroni-zation

Local stores

Direct load access to remote window portion

F In Fortran, before and after the synchronization, on must add: CALL MPI_F_SYNC_REG (buffer)
to guarantee that register copies of buffer are written back to memory, respectively read again from memory.

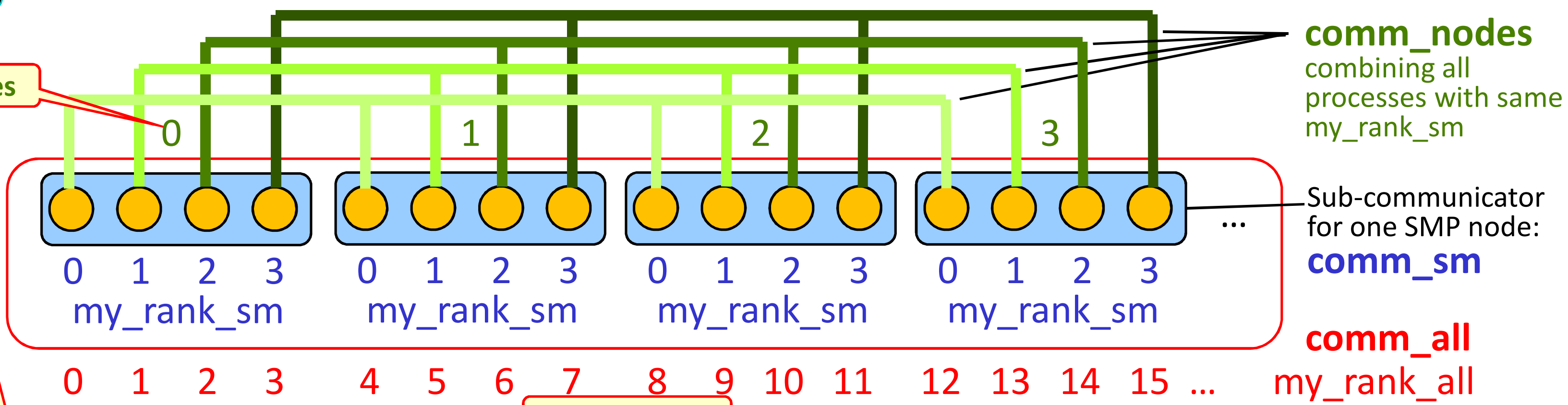
step

skipped

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

Hybrid MPI+MPI
MPI for inter-node communication
+ MPI-3.0 shared memory programming

my_rank_nodes



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

Input

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

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

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

Exscan does not return value on the first rank, therefore

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

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

step

Alternative: Non-contiguous shared memory

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

Pros:

- Faster local data accesses especially on ccNUMA nodes

Cons:

- Higher programming effort for neighbor accesses: MPI_WIN_SHARED_QUERY

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

NUMA effects?
Significant impact of alloc_shared_noncontig

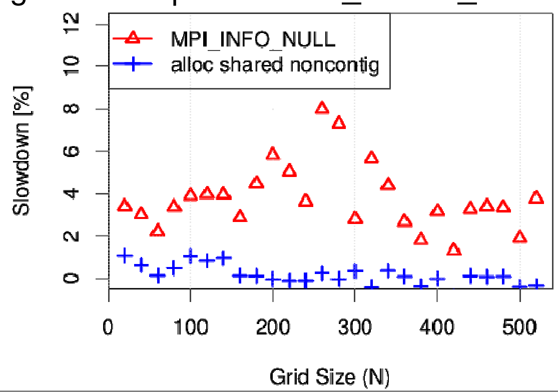
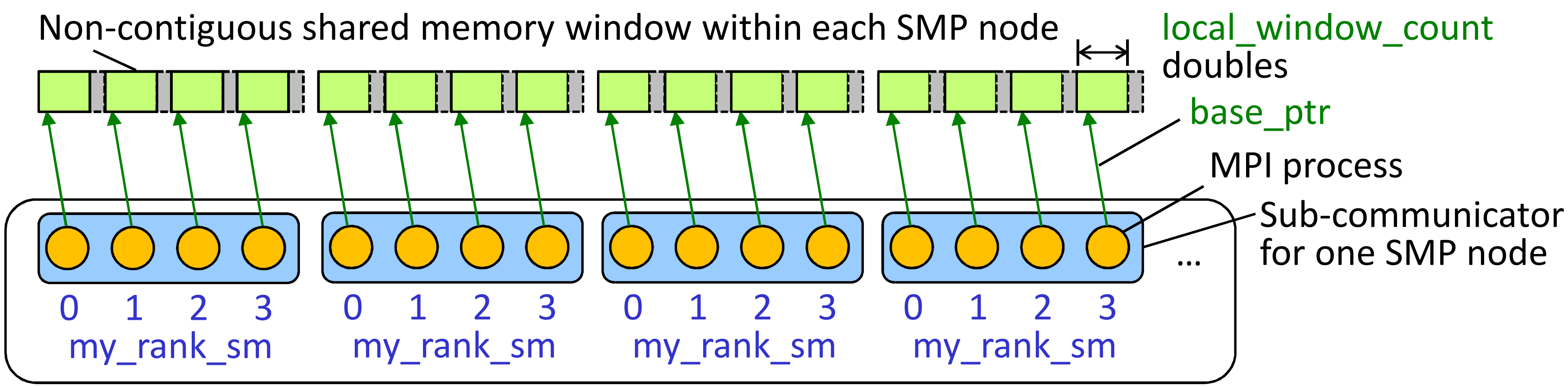


Image: Courtesy of Torsten Hoefler

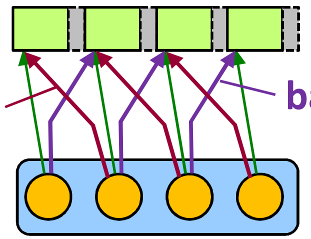
Non-contiguous shared memory allocation




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

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

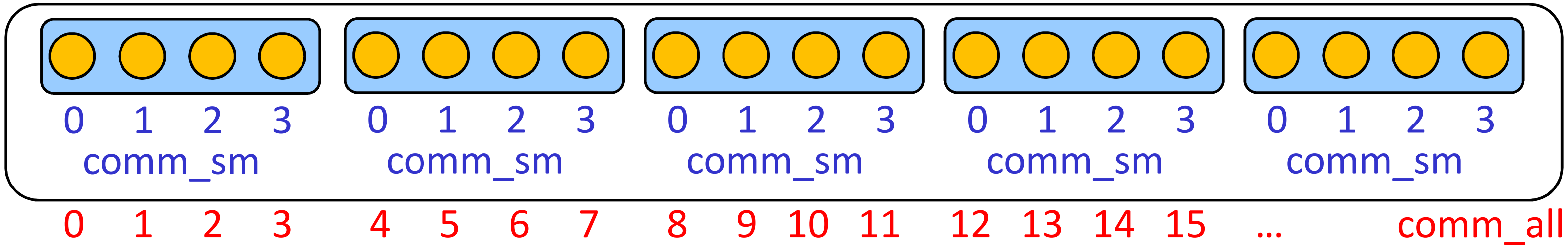
Due to default limit of context IDs in mpich

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

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

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

Splitting the communicator without MPI_COMM_SPLIT_TYPE



Input from outside

Alternative, if you want to group based on a fixed amount size_sm of shared memory cores in comm_all:

- Based on sequential ranks in comm_all
- Pro: comm_sm can be restricted to ccNUMA locality domains
- Con: MPI does not guarantee MPI_WIN_ALLOCATE_SHARED() on whole SMP node
  (MPI_COMM_SPLIT_TYPE() may return MPI_COMM_SELF or partial SMP node)

```
MPI_Comm_rank (comm_all, &my_rank);
MPI_Comm_split (comm_all, /*color*/ my_rank / size_sm,  0, &comm_sm);
MPI_Win_allocate_shared (…);
```

To guarantee shared memory, one may add an additional **MPI_Comm_split_type** (comm_sm, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm_really);

Other synchronization 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_PUT or MPI_GET) and the synchronization between the processes is done with MPI communication (instead of RMA synchronization routines).

```
Process A                          Process B

MPI_WIN_LOCK_ALL(                  MPI_WIN_LOCK_ALL(
MPI_MODE_NOCHECK,win)              MPI_MODE_NOCHECK,win)

DO ...                             DO ...
  X=...
  MPI_F_SYNC_REG(X) 1)
  MPI_WIN_SYNC(win)
  MPI_Send
                                     MPI_Recv
                                     MPI_WIN_SYNC(win)
                                     MPI_F_SYNC_REG(X) 1)
                                     print X

                                     MPI_F_SYNC_REG(X) 1)
                                     MPI_WIN_SYNC(win)
                                     MPI_Send
  MPI_Recv
  MPI_WIN_SYNC(win)
  MPI_F_SYNC_REG(X) 1)
END DO                             END DO

MPI_WIN_UNLOCK_ALL(win)            MPI_WIN_UNLOCK_ALL(win)
```

Data exchange in this direction, therefore **MPI_WIN_SYNC** is needed in both processes: **Write-read-rule**

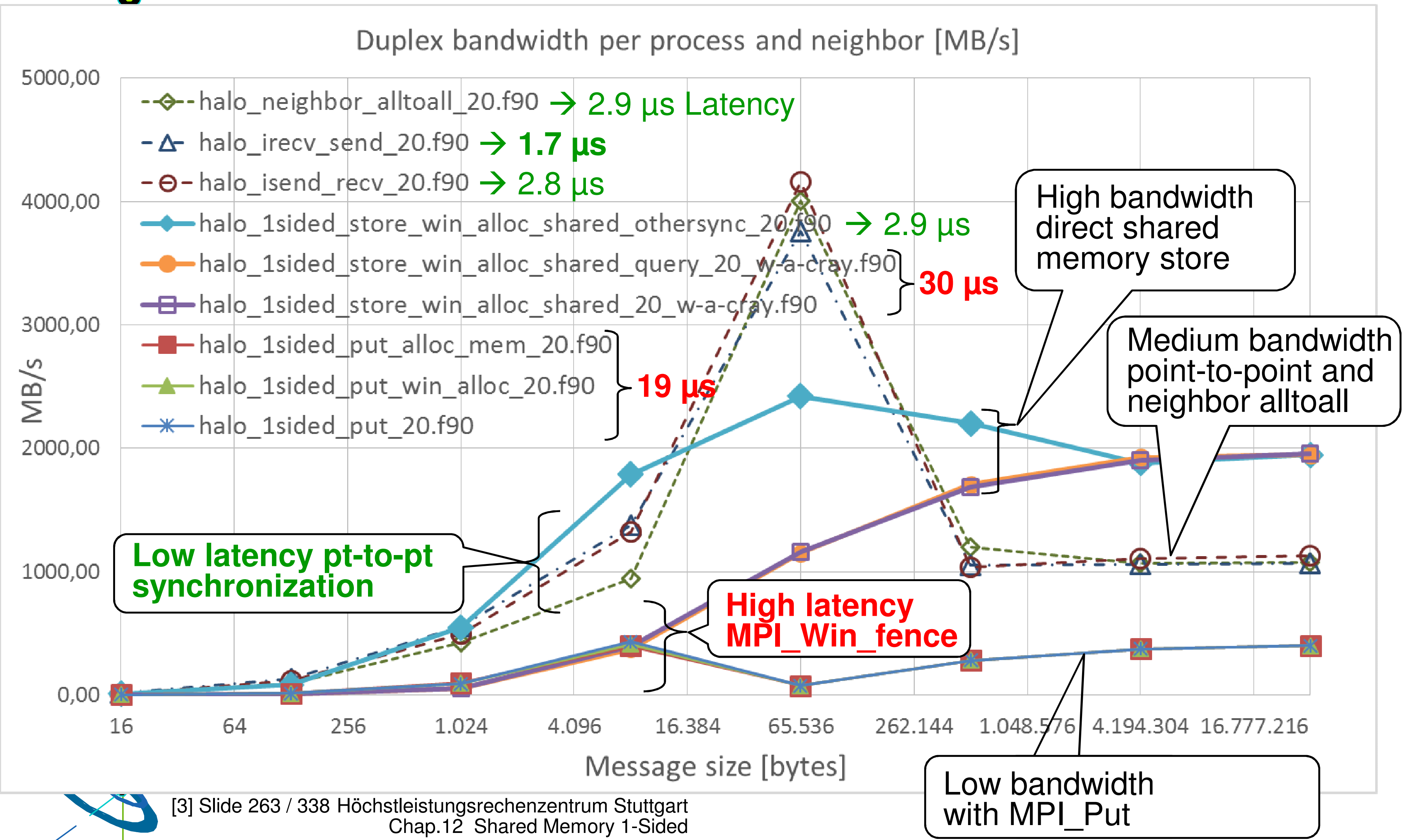
Also needed due to **read-write-rule**

1) Fortran only.

- The used synchronization must be supplemented with MPI_WIN_SYNC, which acts only locally as a processor-memory-barrier. For MPI_WIN_SYNC, a passive target epoch is established with MPI_WIN_LOCK_ALL.
- **X** is part of a shared memory window and should be **the same** memory location **in both processes**.

- MPI-3.0 forgot to define the synchronization methods
- See errata coming Dec. 2014 or March 2015
- Current proposal see https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/456

Benchmark results on a Cray XE6 –
1-dim ring communication on 1 node with 32 cores



Benchmark 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-3 Shared Memory – a Summary

- Shared Memory was introduced in MPI-3.
- 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 should not be combined with low latency synchronization.
- Which communication option is the fastest?

No answer by the MPI standard, because:

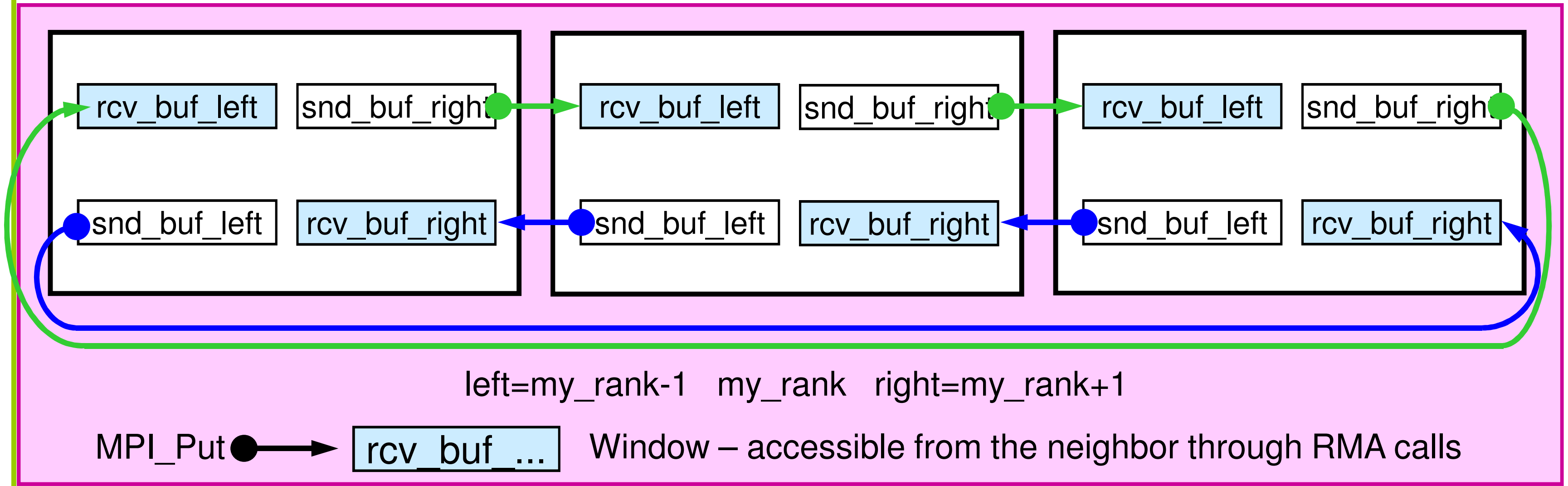
MPI targets portable and efficient message-passing programming

but

efficiency of MPI application-programming is not portable!

Exercise 1: Halo communication with MPI_Put

- Copy to your local directory and analyze the source code:
  cp ~/MPI/course/**C**/1sided/halo_1sided_put_win_alloc.c ./
  cp ~/MPI/course/**F_30**/1sided/halo_1sided_put_win_alloc_30.f90 ./ (or _20 with mpi module)
- 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

Exercise 2: Shared memory halo communication

- Use the given program as your baseline for the following exercise:
  - **cp halo… my_shared_exa1.c** *or* **…_20.f90** *or* **…_30.f90**
- Tasks: Substitute the distributed window by a shared window
  - Substitute **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 without errata: The role of RMA synchronization used as process synchronization for direct shared memory accesses (i.e., without RMA calls) is undefined! Implication: **MPI_Win_fence can be used, but only with assert = 0**. (State July 27, 2014)

Sol.

Exercise 3 (advanced): Using *other* synchronization

- Use your exa1 result or
  - ~/MPI/course/**C**/1sided/halo_1sided_store_win_alloc_shared.c
    ~/MPI/course/**F_30**/1sided/halo_1sided_store_win_alloc_shared_30.f90 (or _20)

  as your baseline for the following exercise:
  - **cp my_shared_exa1.c my_shared_exa2.c** *or* **..._20.f90** *or* **..._30.f90**
- Tasks: Substitute the MPI_Fence synchronization by pt-to-pt communication
  - Use empty messages for synchronizing
  - Substitute each pair of MPI_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 to sync local processor with memory after another process has written to the memory or before providing new data in own memory to another processor.
- Compile and run shared memory program
  - With MPI processes on **4** cores & **all cores** of a shared memory node

see also login-slides

Summary of halo files

ring.c ring_1sided_get.c ring_1sided_put.c ring_1sided_exa2.c

- halo_**isend_recv**.c
  - halo_**irecv_send**.c
    - halo_**neighbor_alltoall**.c
    - halo_**1sided_put**.c
      - halo_1sided_put_**alloc_mem**.c
        - halo_1sided_put_**win_alloc**.c
          - halo_1sided_store_win_**alloc_shared**.c
            - halo_1sided_store_win_alloc_shared_**w-a-cray**.c
            - halo_1sided_store_win_alloc_shared_**query**.c
            - halo_1sided_store_win_alloc_shared_query_**w-a-cray**.c
            - halo_1sided_store_win_alloc_shared_**pscw**.c
            - halo_1sided_store_win_alloc_shared_**othersync**.c

MPI and Threads

Chap.13 MPI and Threads

MPI_Init()
MPI_Comm_rank()

13. MPI and Threads

– **e.g., hybrid MPI and OpenMP**

Skip this section (4 slides)

MPI rules with OpenMP / Automatic SMP-parallelization

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

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

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

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!

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

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

… the barrier is necessary – example with MPI_Recv

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

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

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

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

}
/* omp end parallel */
```

No barrier inside

Barriers needed to prevent data races

MPI + threading methods

- MPI + OpenMP
  - Often better one process per NUMA domain (not per ccNUMA node)
  - (Perfect) compiler support for threading
  - Called libraries must be thread-safe
- MPI + MPI-3 shared memory
  - Efficient placement of MPI processes on ccNUMA nodes *is not trivial*
    - **Hard for applications with unstructured grids**
    - **Possible solution: Domain decomposition on core level.
      Then recombining for (cc)NUMA domains.**
    - **See in Chapter 8. Groups&Communicators, the slide on "Unstructured Grids – Multi-level Domain Decomposition through Recombination"**
  - Presented measurements show (limited portability to other systems!)
    - **MPI-3 shared memory could be used for direct access on fully shared data structures**
    - **Better than halo copying within shared memory nodes** (as in the halo examples)
    - **→ Less memory and less communication effort!**
- General
  - Efficient placement of cores and processes and threads on ccNUMA nodes

For private notes

· private notes

For private notes

Chap.14 Parallel File I/O

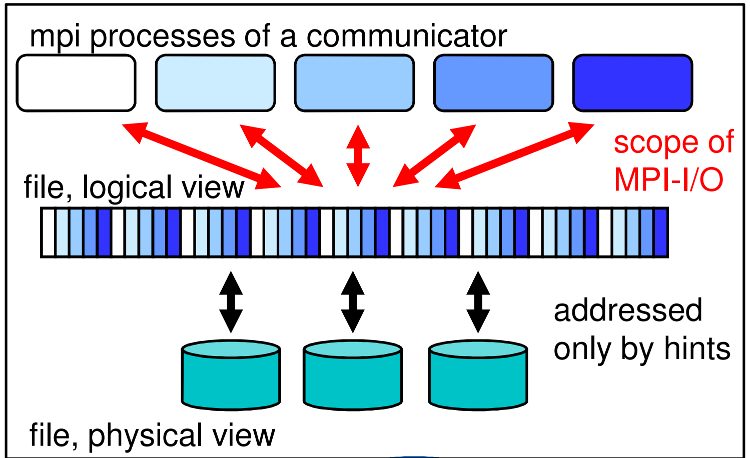
MPI_Init()
MPI_Comm_rank()

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Nonblocking communication
5. Probe, Persistent Requests, Cancel
6. Derived datatypes
7. Virtual topologies
8. Groups & communicators, environment management
9. Collective communication
10. Process creation and management
11. One-sided communication
12. Shared memory one-sided communication
13. MPI and threads

14. Parallel file I/O

– **Writing and reading a file in parallel by many processes**

15. Other MPI features



1st skip-point: Skip rest of this chapter

Short tour – 6 slides

Outline

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 to overlap computation & I/O -> MPI_I..., MPI_Wait, ... & new **split** collective interface
  - non-contiguous access -> MPI derived datatypes

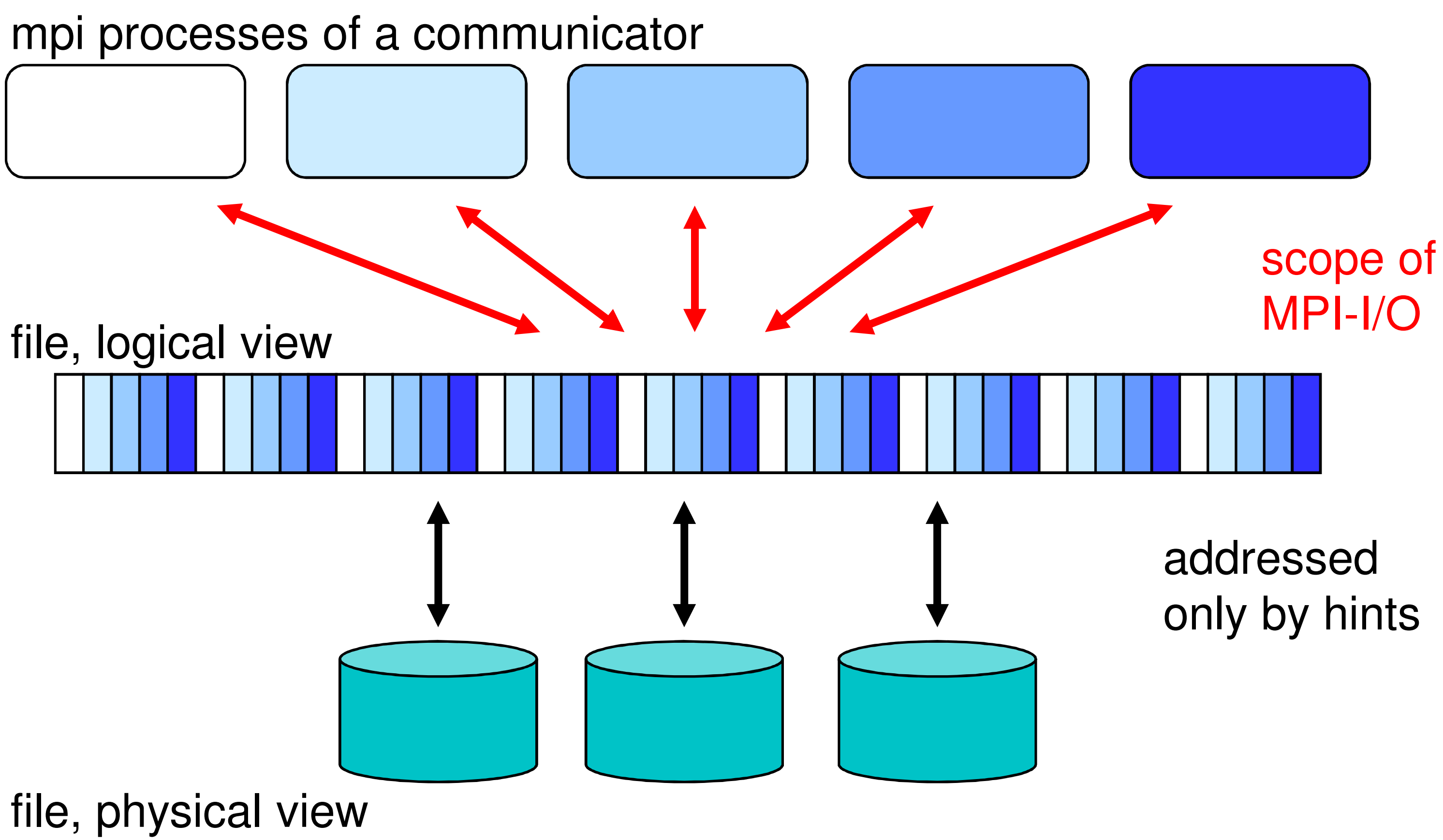
MPI-I/O Features

- 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

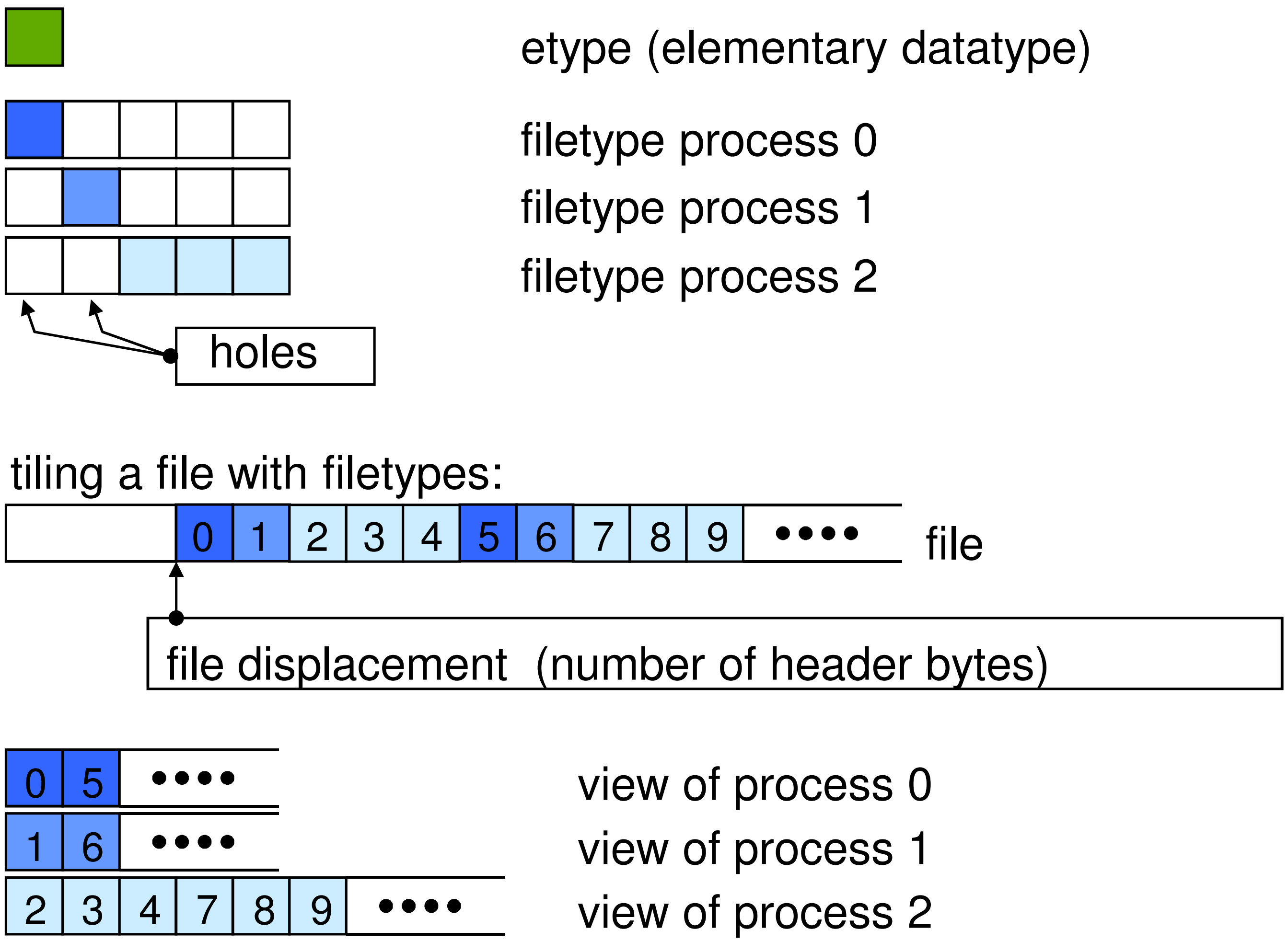
MPI-I/O, Principles

- 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
- automatic data conversion in heterog. systems
- file interoperability with external representation

Logical view / Physical view



Definitions



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., <u>multiples of etype extent</u>)

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*)

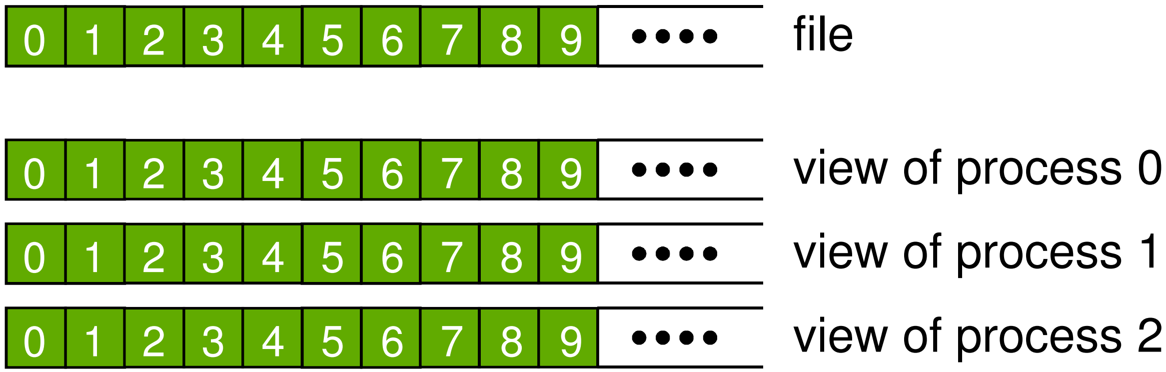
Fortran C/C++ language bindings – see MPI-2 Standard

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



- Sequence of MPI_BYTE matches with any datatype (see MPI-3.0, Section 13.6.5)
- 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*]

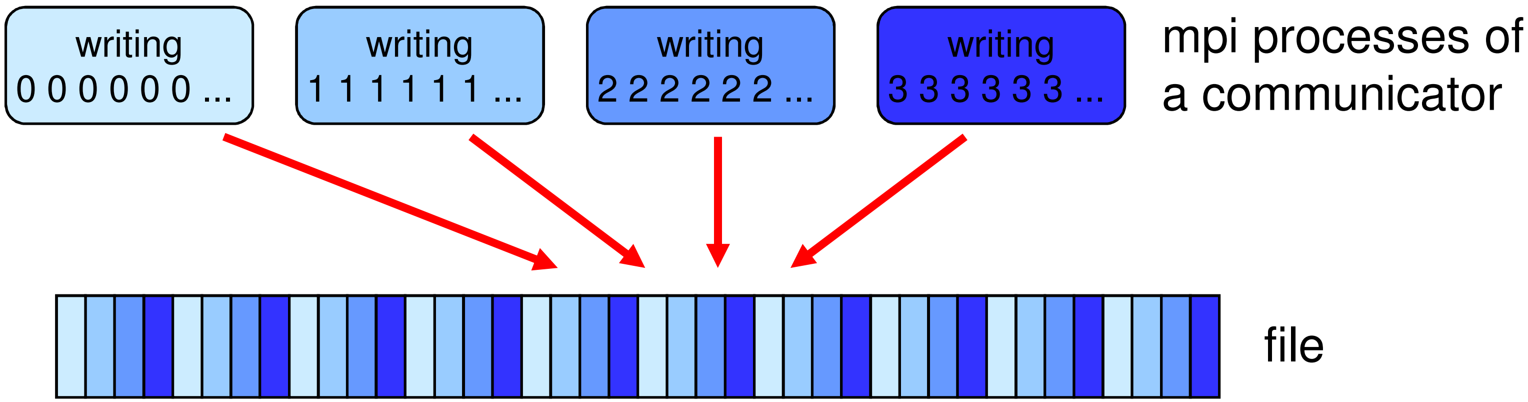
Writing with Explicit Offsets

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

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 = my_rank + i * size_of_MPI_COMM_WORLD, i=0..9
- Result: “0123012301230123012301230123012301230123“
- Each process uses the default view



- please, use skeleton:

```
cp ~/MPI/course/C/mpi_io/mpi_io_exa1_skel.c  my_exa1.c
cp ~/MPI/course/F/mpi_io/mpi_io_exa1_skel.f  my_exa1.f
```


see also login-slides

Sol.

Outline – Block 2

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)
```

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

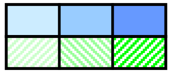
Fileview examples with SUBARRAY and DARRAY

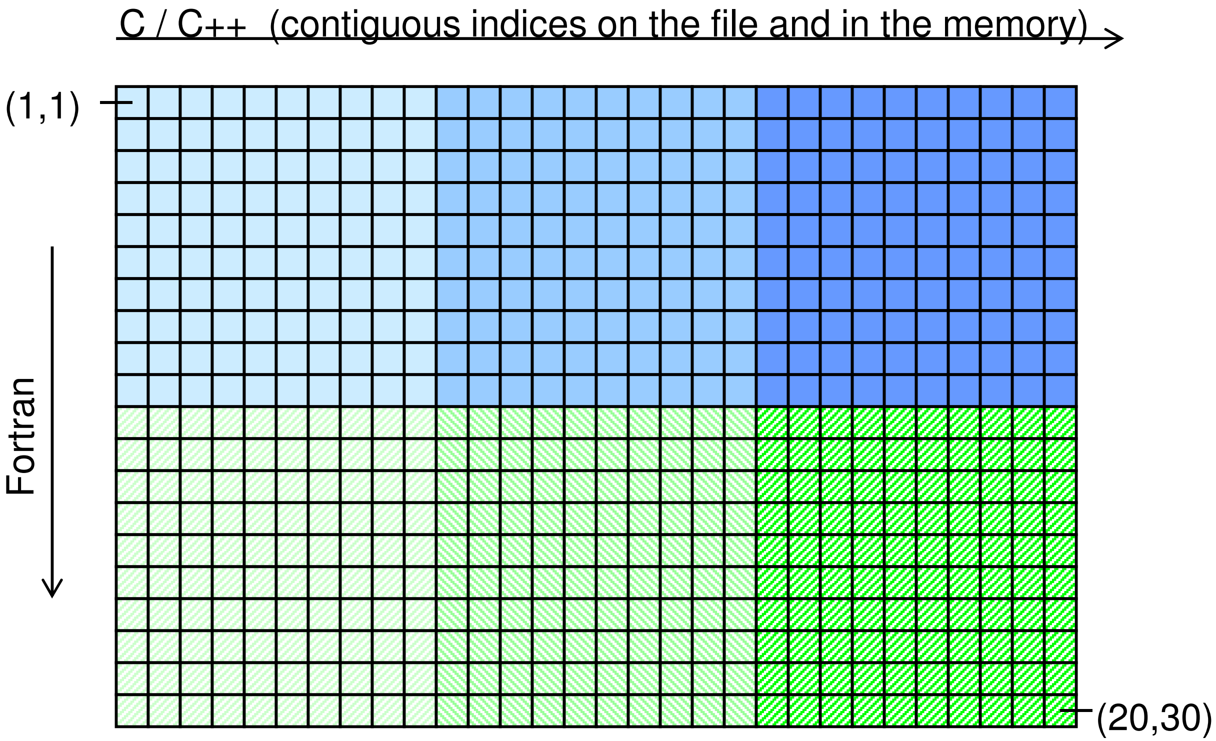
- Task
  - reading a global matrix from a file
  - storing a subarray into a local array on each process
  - according to a given distribution scheme

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 processor: 10x10



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,
                                                       .TRUE., comm, ierror)
      call MPI_COMM_RANK(comm, rank, 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)
```


HLRS

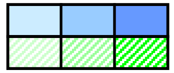
Skip-point: Skip rest of this chapter

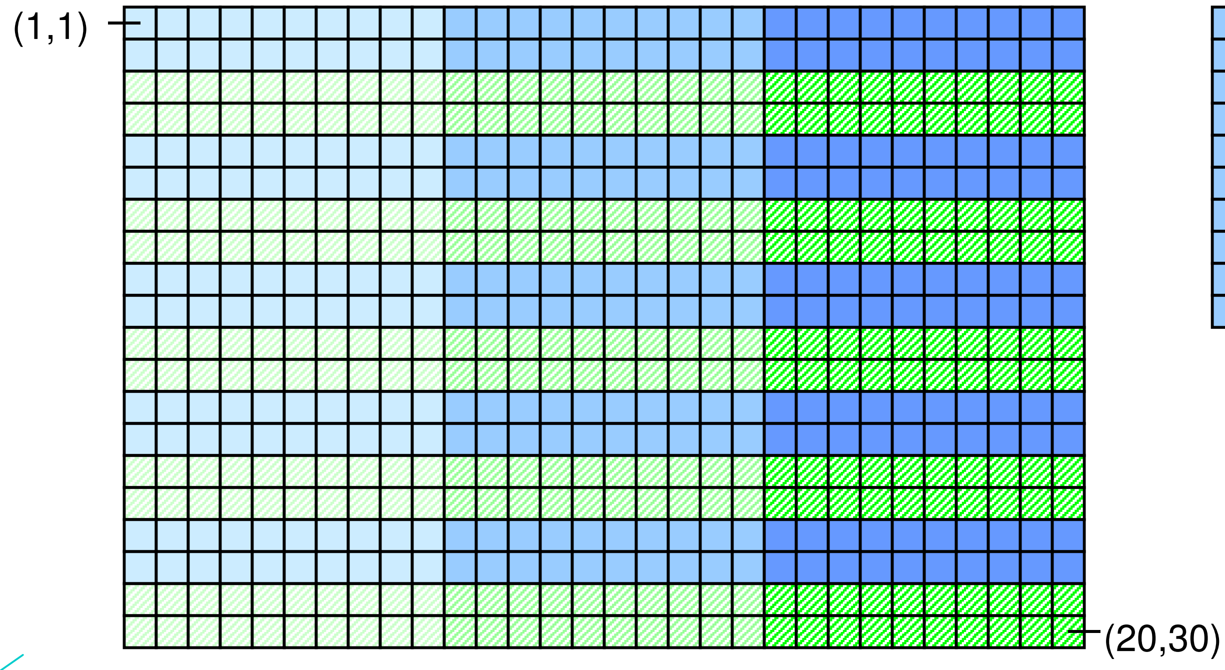
Example with Subarray, IV.

- 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:

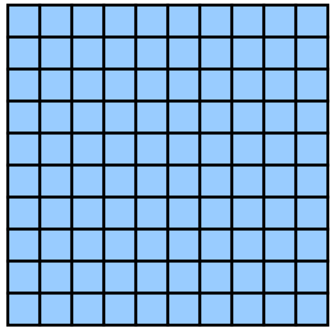
rank = 0 coords = (0, 0) starts = (0, 0) garray(1:10, 1:10) = larray (1:10, 1:10)	rank = 1 coords = (0, 1) starts = (0, 10) garray(1:10, 11:20) = larray (1:10, 1:10)	rank = 2 coords = (0, 2) starts = (0, 20) garray(1:10, 21:30) = larray (1:10, 1:10)
rank = 3 coords = (1, 0) starts = (10, 0) garray(11:20, 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:

rank = 0 coords = (0, 0) $\text{garray}\left(\begin{bmatrix}1:2\\5:6\\9:10\\13:14\\17:18\end{bmatrix}, 1:10\right)$ = larray (1:10, 1:10)	rank = 1 coords = (0, 1) $\text{garray}\left(\begin{bmatrix}1:2\\5:6\\9:10\\13:14\\17:18\end{bmatrix}, 11:20\right)$ = larray (1:10, 1:10)	rank = 2 coords = (0, 2) $\text{garray}\left(\begin{bmatrix}1:2\\5:6\\9:10\\13:14\\17:18\end{bmatrix}, 21:30\right)$ = larray (1:10, 1:10)
rank = 3 coords = (1, 0) $\text{garray}\left(\begin{bmatrix}3:4\\7:8\\11:12\\15:16\\19:20\end{bmatrix}, 1:10\right)$ = larray (1:10, 1:10)	rank = 4 coords = (1, 1) $\text{garray}\left(\begin{bmatrix}3:4\\7:8\\11:12\\15:16\\19:20\end{bmatrix}, 11:20\right)$ = larray (1:10, 1:10)	rank = 5 coords = (1, 2) $\text{garray}\left(\begin{bmatrix}3:4\\7:8\\11:12\\15:16\\19:20\end{bmatrix}, 21:30\right)$ = larray (1:10, 1:10)

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	Synchronization	Non-collective	Collective
Explicit offsets	blocking	READ_AT WRITE_AT	READ_AT_ALL WRITE_AT_ALL
	non-blocking & split collective	IREAD_AT IWRITE_AT	READ_AT_ALL_BEGIN READ_AT_ALL_END WRITE_AT_ALL_BEGIN WRITE_AT_ALL_END
Individual file pointers	blocking	READ WRITE	READ_ALL WRITE_ALL
	non-blocking & split collective	IREAD IWRITE	READ_ALL_BEGIN READ_ALL_END WRITE_ALL_BEGIN WRITE_ALL_END
Shared file pointers	blocking	READ_SHARED WRITE_SHARED	READ_ORDERED WRITE_ORDERED
	non-blocking & split collective	IREAD_SHARED IWRITE_SHARED	READ_ORDERED_BEGIN READ_ORDERED_END WRITE_ORDERED_BEGIN WRITE_ORDERED_END

Read e.g. MPI_FILE_READ_AT

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*)

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

(*formula is not valid if EOF is reached*)

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_EOF

MPI_FILE_GET_POSITION(fh, *offset*)

MPI_FILE_GET_BYTE_OFFSET(fh, offset, disp)

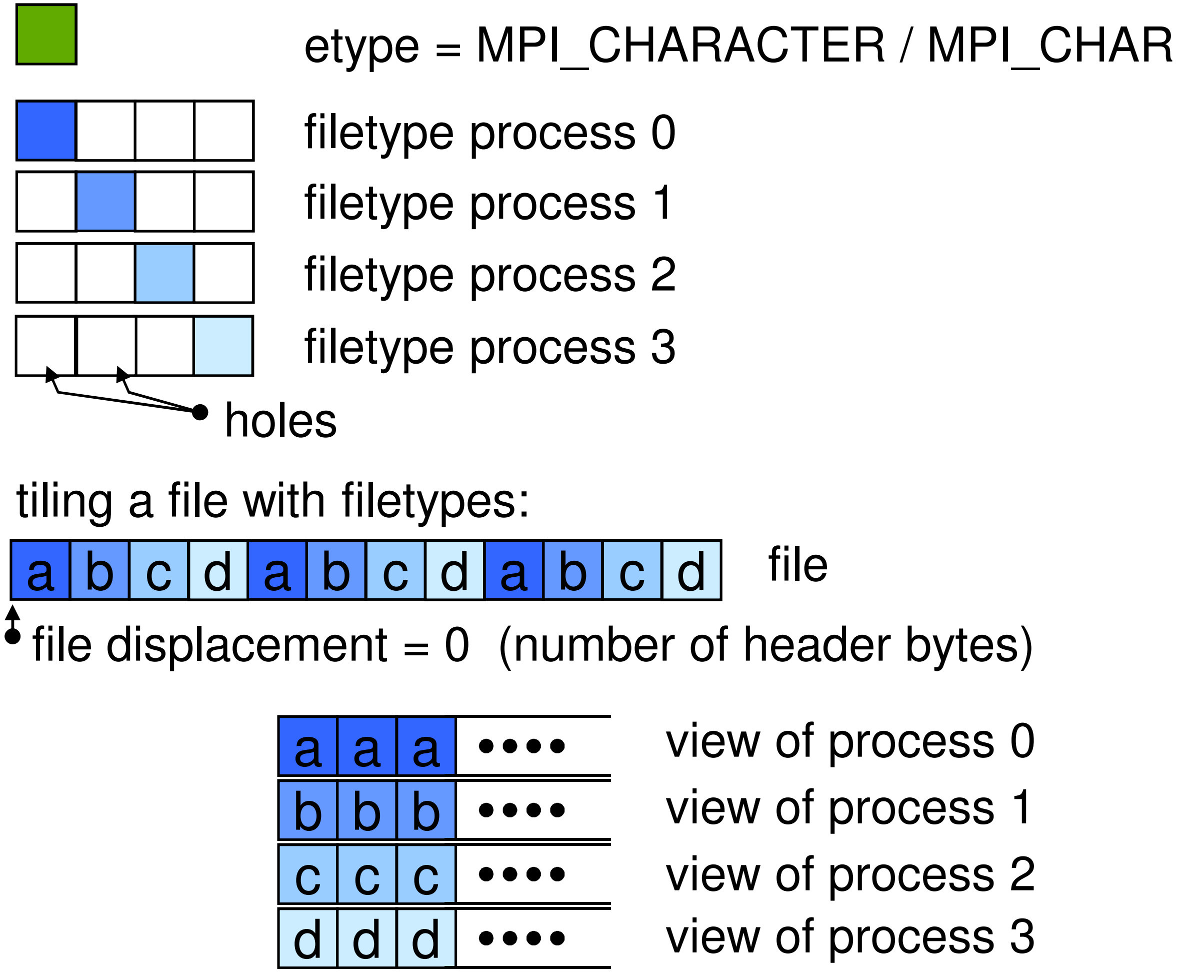
- to inquire offset
- to convert offset into byte displacement
  [*e.g. for* `disp` *argument in a new view*]

MPI–IO Exercise 2: Using fileviews and individual filepointers

- Copy to your local directory:
  cp ~/MPI/course/**C**/mpi_io/mpi_io_exa2_skel.c **my_exa2.c**
  cp ~/MPI/course/**F**/mpi_io/mpi_io_exa2_skel.f **my_exa2.f**
- 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 / .f`
  - Compile and run your `my_exa2.c / .f`

H L R S

MPI–IO Exercise 2: Using fileviews and individual filepointers, continued



Outline – Block 3

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)
```

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

Collective Data Access

- Explicit offsets / individual file pointer:
  - same as non-collective calls by all processes "of `fh`"
  - ***chance for best speed!!!***
- shared file pointer:
  - accesses are ordered by the ranks of the processes
  - optimization chance:
    - **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,
    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

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*)

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 ..._END call
- Chance 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

Scenery – Split Collective

- Scenery A:
  - Task: Each process has to read the whole file
  - Solution:
    - 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_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 [*may be expensive*]
  MPI_FILE_ PREALLOCATE(fh, size)
- Resizing a file [*may speed up first writing on a file*]
  MPI_FILE_SET_SIZE(fh, size)
- 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)
  MPI_FILE_GET_INFO(fh, ***info_used***)

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:

  MPI_File_set_errhandler(MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL);

  CALL MPI_FILE_SET_ERRHANDLER(MPI_FILE_NULL,MPI_ERRORS_ARE_FATAL,***ierr***)
- 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
- Initial implementations of MPI I/O available
  (eg, ROMIO from Argonne)
- Available nearly on every MPI implementation

I/O – Exercise 3

MPI–IO Exercise 3: Collective ordered I/O

- Copy to your local directory:

  cp ~/MPI/course/**C**/mpi_io/mpi_io_exa3_skel.c **my_exa3.c**

  cp ~/MPI/course/**F**/mpi_io/mpi_io_exa3_skel.f **my_exa3.f**
- Tasks:
  - Substitute the write call with individual filepointers by a collective write call with shared filepointers
  - Compile and run your `my_exa3.c / .f`

H L R I S

see also login-slides

Sol.

MPI–IO Exercise 4: I/O Benchmark

- Copy to your local directory:

  ```
  cp ~/MPI/course/F/mpi_io/[am]* .
  ```

- You receive:

  `mpi_io_exa4.f`

  `ad_ufs_open.o, ad_ufs_read.o, ad_ufs_write.o` *)

- Tasks:
  - compile and execute mpi_io_exa4 on 4 PEs
  - compile and link with `ad_ufs*.o` and execute on 4 Pes *)
  - duplicate lines 65 –93 three times and substitute “WRITE_ALL” by “WRITE”, “READ_ALL”, “READ” and execute on 4 PEs
  - double the value of gsize and compile and execute on 8 PEs
  - link without `ad_ufs*.o` and execute on 8 Pes *)

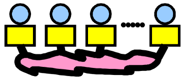
  *) ad_ufs only on T3Es with striped file system

For private notes

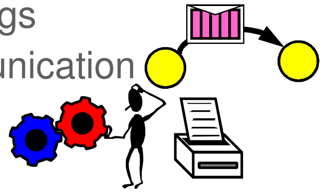
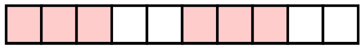
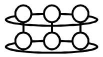
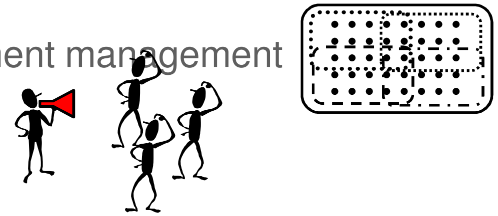
· private notes

For private notes

Chap.15 Other MPI Features

1. MPI Overview 

MPI_Init()
MPI_Comm_rank()

2. Process model and language bindings
3. Messages and point-to-point communication 
4. Nonblocking communication
5. Probe, Persistent Requests, Cancel
6. Derived datatypes 
7. Virtual topologies 
8. Groups & communicators, environment management 
9. Collective communication
10. Process creation and management
11. One-sided communication
12. Shared memory one-sided c.
13. MPI and threads
14. Parallel file I/O

15. Other MPI features

Further MPI-3.0 chapters and sections:

[1, 2] – **Introduction, Terms & Conventions**
[12.1-3] – **Generalized requests**
[14] – **Profiling & Tools support**
[15, 16] – **Deprecated & removed interfaces**
[17.2] – **Language interoperability**
[A] – **Language bindings summary**
[B] – **Change-log**

Other MPI Features

Further MPI-3.0 chapters and sections:

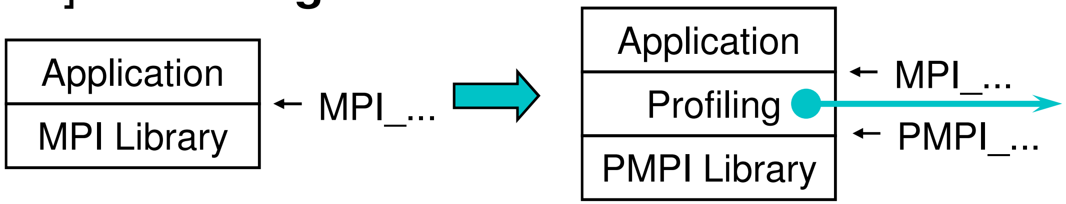
- [1, 2] **Introduction, Terms & Conventions**
  - This course is an introduction to MPI
  - MPI-3.0 Chap. 1+2 gives a good overview of
    - the MPI standard, and
    - all major terms used within this standard

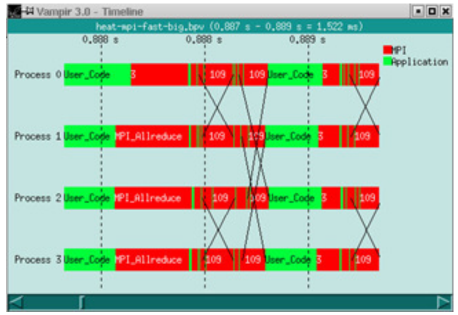
- [12.1-3] **Generalized requests**
  - If you want to use the MPI request handling for an own interface.
  - Needed, e.g., if you want to implement a part of the MPI standard (e.g. I/O) as a portable software for all MPI libraries.

Other MPI Features

<u>Further MPI-3.0 chapters and sections:</u>

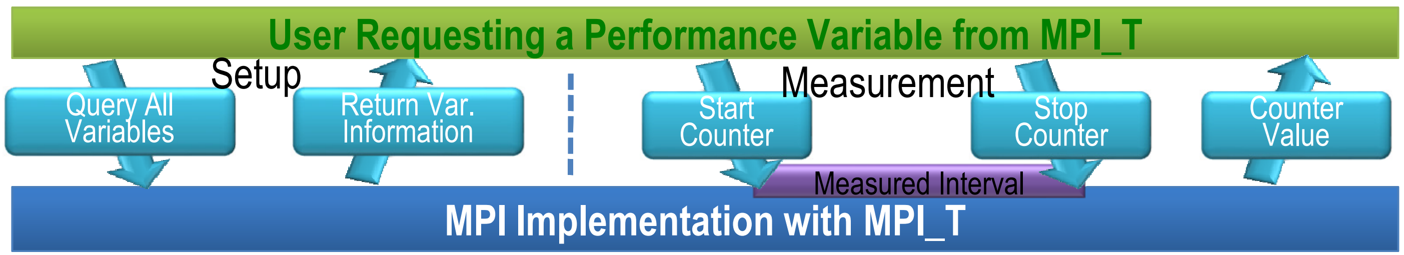
- [14.1-2] **Profiling Interface**





e.g. with Vampir

- [14.3] **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



Tools figure: Courtesy of Martin Schulz

Other MPI Features

Further MPI-3.0 chapters and sections:

- [2.6.1] **Deprecated and Removed Names and Functions**
  - Table 2.1 on page 18 presents a good overview
- [15, 16] **Deprecated & Removed Interfaces**
- [17.2] **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
- [B] **Change-log**
  - What is new in MPI-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

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 chance for optimization
- One-sided communication and shared memory → functionality & perform.
- Parallel file I/O → important option on large systems / part of 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

For private notes

For private notes

For private notes

APPENDIX

Solutions

- Chapter 2: Hello world / I am *my_rank* of *size*
- Chapter 3: Ping-pong with point-to-point communication
- Chapter 4: Nonblocking halo-copy in a ring
- Chapter 6: Halo-copy with derived types
- Chapter 7: Ring with virtual Cartesian topology
- Chapter 9: Collective communication with MPI_Allreduce
- Chapter 11: Halo-copy with one-sided communication
- Chapter 12: Halo-copy with MPI-3.0 shared memory one-sided communication
- Chapter 14: Parallel file I/O exercises

Chapter 2: Hello world / I am my_rank of size

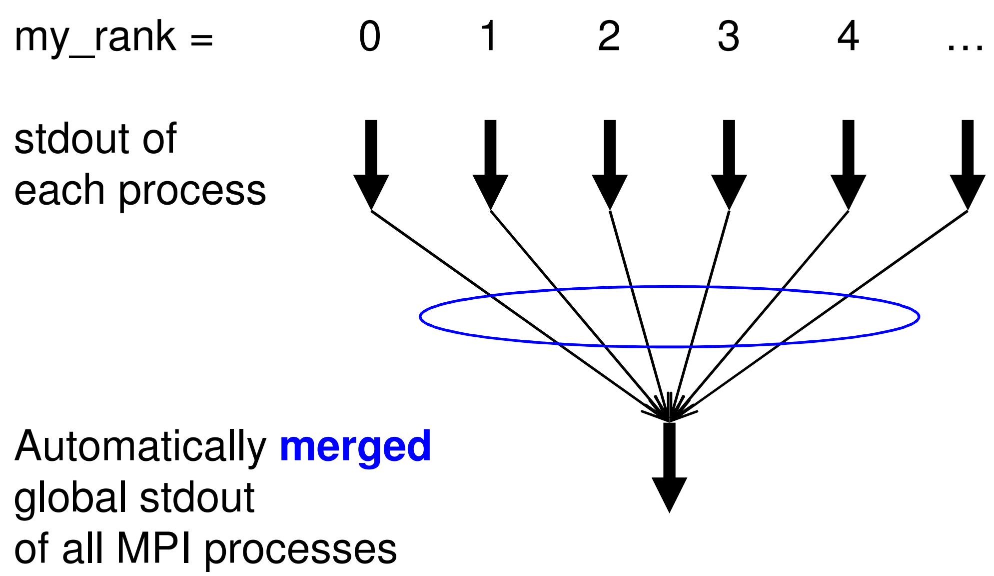
C

```
#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();
}
```

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
```

Chapter 2: 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: Ping-pong with point-to-point communication

C

```
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();
if (my_rank == 0)
   printf("Time for one messsage: %f micro seconds.\n",
          (finish - start) / (2 * 50) * 1e6 );
```

Fortran

```
start = MPI_Wtime()
DO i = 1, number_of_messages
   IF (my_rank .EQ. proc_a) THEN
      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)
   ELSE IF (my_rank .EQ. proc_b) THEN
      CALL MPI_Recv(buffer, 1, MPI_REAL, 0, 17, MPI_COMM_WORLD, status)
      CALL MPI_Send(buffer, 1, MPI_REAL, 0, 23, MPI_COMM_WORLD)
   END IF
END DO
finish = MPI_Wtime()
IF (my_rank .EQ. proc_a) THEN
   WRITE(*,*) 'One message:',(finish-start)/(2*50)*1e6,' micro seconds'
ENDIF
```

Chapter 4: Nonblocking halo-copy in a ring

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;
snd_buf = my_rank;                                   1
for( i = 0; i < size; i++)
{
  MPI_Issend(&snd_buf, 1, MPI_INT, right, 17, MPI_COMM_WORLD, &request);   2
  MPI_Recv ( &rcv_buf, 1, MPI_INT, left,  17, MPI_COMM_WORLD, &status);    3
  MPI_Wait(&request, &status);
  snd_buf = rcv_buf;                                 4
  sum += rcv_buf;                                    5
}
printf ("PE%i:\tSum = %i\n", my_rank, sum);
MPI_Finalize();
```

Chapter 4: Nonblocking halo-copy in a ring

Fortran

1

2

3

4

5

```
INTEGER, ASYNCHRONOUS :: snd_buf    2
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)
!    CALL MPI_GET_ADDRESS(snd_buf, iadummy)
!    ... should be substituted as soon as possible by:
   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()
```

Chapter 6: Halo-copy with derived types (major changes)

C

Provided in the skeleton

```
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] = 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 = 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);
```

Chapter 6: Halo-copy with derived types (major changes)

Fortran

Provided in the skeleton

```
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) = 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 = 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
```

back

Chapter 7: Ring with virtual Cartesian topology (major changes)

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);
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
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)
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)
```

Chapter 9: Collective communication with MPI_Allreduce

C

```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();
}
```

Fortran

```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
```

Chapter 11: Halo-copy with one-sided communication

C

```
MPI_Win  win;
-----------------------------------------------------------------------
/* Create the window once before the loop:  */
MPI_Win_create(&rcv_buf, sizeof(int), sizeof(int), MPI_INFO_NULL,
               MPI_COMM_WORLD, &win);
-----------------------------------------------------------------------
  /* Inside of the loop; instead of MPI_Issend / MPI_Recv / MPI_Wait:  */
  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);
```

Fortran

```
TYPE(MPI_Win) :: win
INTEGER :: disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: integer_size, lb, buf_size, target_disp
-----------------------------------------------------------------------
target_disp = 0 ! This "long" integer zero is needed in the call to MPI_PUT
! 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)
-----------------------------------------------------------------------
! Inside of the loop; instead of MPI_Issend / MPI_Recv / MPI_Wait:
  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)
  CALL MPI_PUT(snd_buf,1,MPI_INTEGER,right,target_disp,1,MPI_INTEGER, win)
  CALL MPI_WIN_FENCE(IOR(MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOSUCCEED),win)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
```

Provided in the skeleton

Chapter 12: Halo-copy shared memory one-sided comm.

C

```
int k;
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 substited 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 12: Halo-copy shared memory one-sided comm.

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 12: 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 substited 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_uf_right/left = …
must not be moved across both MPI_Win_fence

Chapter 14: Parallel file I/O exercise 1 – explicit file-pointer

C

```
MPI_Offset offset;
…
                                   or MPI_MODE_WRONLY
    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);
    }
```

Fortran

```
    INTEGER (KIND=MPI_OFFSET_KIND) offset
    …
                                            or MPI_MODE_WRONLY
    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
```

Chapter 14: Parallel file I/O exercise 2 – with fileview

C

```
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);
    }
```

or 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 MPI_MODE_**WRONLY**

or **MPI_CHARACTER**

back

Chapter 14: Parallel file I/O exercise 3 – shared filepointer

C

```
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);
```

Fortran

```
      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)
```

For private notes

For private notes

For private notes

For private notes

For private notes

For private notes

For private notes