Introduction to the Message Passing Interface (MPI)

Rolf Rabenseifner
rabenseifner@hlrs.de

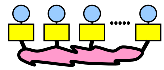
University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Acknowledgments

- 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.

Outline

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



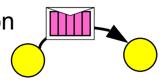
[2.3, 2.6]
slides 7–…

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

MPI_Init()
MPI_Comm_rank()

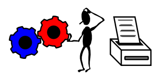
[2.5, 5.4.1, 7.5]
slides 31–…

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



[3.1-5, 7,4]
slides 44–…

4. Non-blocking communication
   – **to avoid idle time and deadlocks**

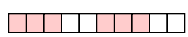


[3.7]
slides 62–…

[...] = **MPI 1.1 chapter**

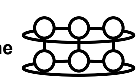
Outline

5. Derived datatypes
   – **transfer any combination of typed data**



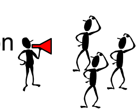
[3.12]
slides 80–…

6. Virtual topologies
   – **a multi-dimensional process naming scheme**



[6]
slides 96–…

7. Collective communication
   – **e.g., broadcast**



[4]
slides 113–…

8. All other MPI-1 features

slides 130–…

[...] = **MPI 1.1 chapter**

Information about MPI

- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July18,1997)
- Marc Snir and William Gropp et al.:
  MPI: The Complete Reference. (2-volume set).The MIT Press, 1998.
  (excellent catching up of the standard MPI-1.2 and MPI-2 in a readable form)
- 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)
- http://www.hlrs.de/mpi/

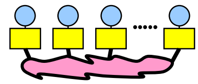
Compilation and Parallel Start

• Your working directory:	~/MPI/*#nr* *with #nr = number of your PC*	
• Initialization:	in .profile: **USE_MPI=1**	(on many systems)
• Compilation in C:	**cc -o** *prg* *prg*.**c**	(on T3E)
	cc -o *prg* *prg*.**c -lmpi**	(on IRIX)
	cc -nx -o *prg* *prg*.**c -lmpi**	(on Paragon)
	mpicc -o *prg* *prg*.**c**	(Hitachi,HP,NEC)
• Compilation in Fortran:	**f90 -o** prg prg.**f**	(on T3E)
	f90 -o prg prg.**f -lmpi**	(on IRIX)
	f77 -nx -o prg prg.**f -lmpi**	(on Paragon)
	mpif90 -o *prg* *prg*.**f**	(Hitachi,HP,NEC)
• Program start on *num* PEs:	**mpirun -np** *num* *./prg*	(all, except ...:)
	isub -sz *num* *./prg*	(Paragon)
	mpiexec -n *num* *./prg*	(standard MPI-2)
• Empty and used partitions:	**fpart; grmview -rw**	(on T3E)
	freepart	(Hitachi, Paragon)
• MPI Profiling:	**export MPIPROFOUT=stdout**	(on T3E)
• C examples	~/MPI/course/**C**/Ch[2-8]/*.c	
• Fortran examples	~/MPI/course/**F**/Ch[2-8]/*.f .../**F**/heat/* .../**F**/mpi_io/*	

(the examples of a chapter are only readable after the end of the practical of that chapter)

Chap.1 MPI Overview

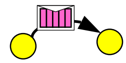
1. **MPI Overview**
   – **one program on several processors**
   – **work and data distribution**
   – **the communication network**



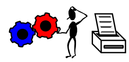
2. Process model and language bindings

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

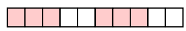
3. Messages and point-to-point communication



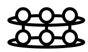
4. Non-blocking communication



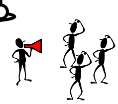
5. Derived datatypes



6. Virtual topologies



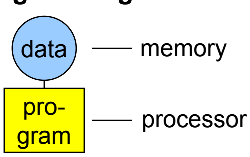
7. Collective communication



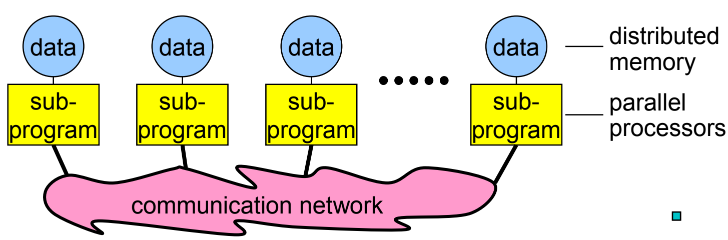
8. All other MPI-1 features

The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**

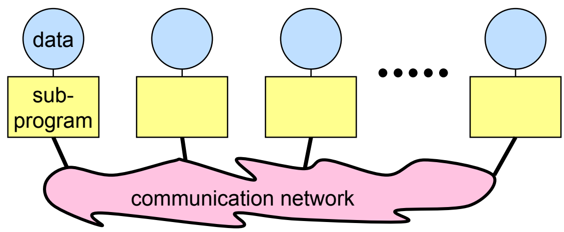


- **Message-Passing Programming Paradigm**



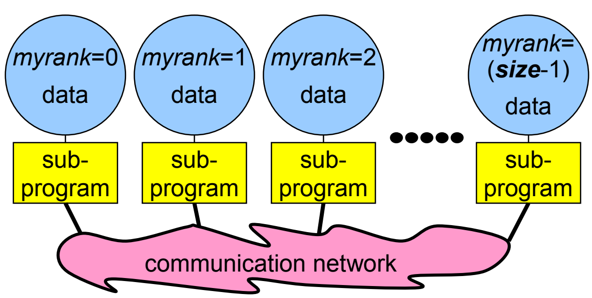
The Message-Passing Programming Paradigm

- 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***)



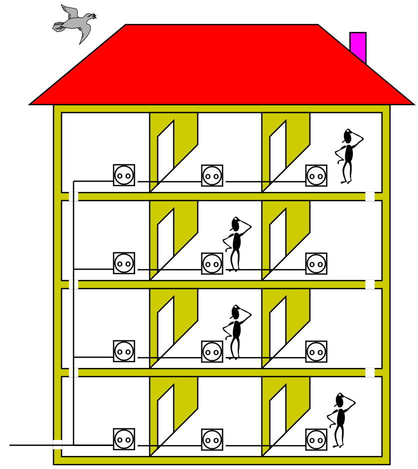
Data and Work Distribution

- 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?

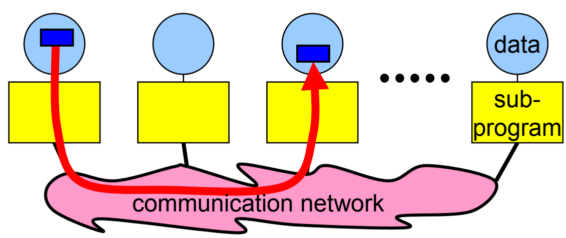
- **S**ingle **P**rogram, **M**ultiple **D**ata
- Same (sub-)program runs on each processor
- MPI allows also MPMD, i.e., **Multiple** Program, ...
- but some vendors may be restricted to SPMD
- MPMD can be emulated with SPMD

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	} (blue box)
– 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
  - 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

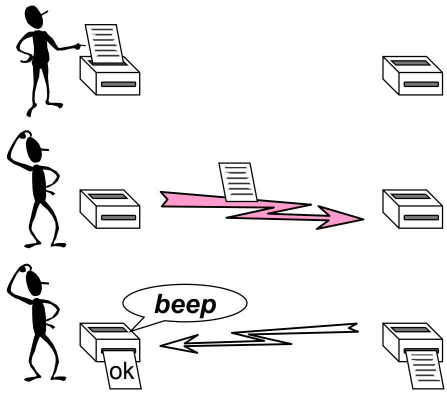
- All messages must be received.

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

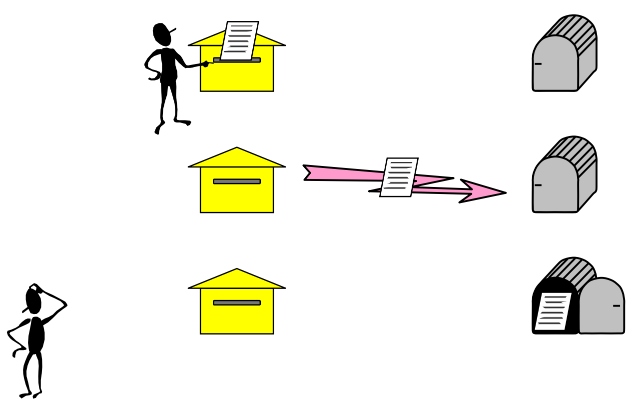
Synchronous Sends

- 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.

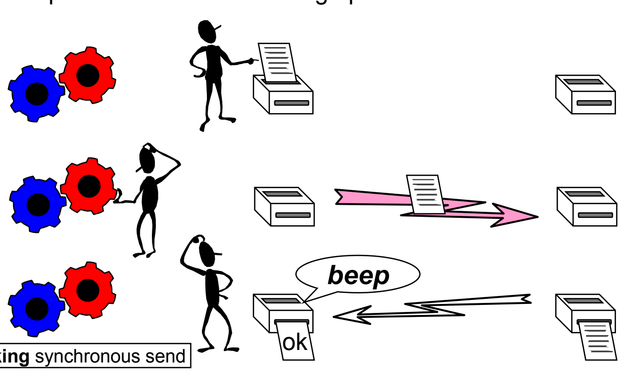


Blocking Operations

- 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 is sent.
- Relates to the completion of an operation.
- Blocking subroutine returns only when the operation has completed.

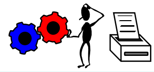
Non-Blocking Operations

- Non-blocking 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 non-blocking operation.



non-blocking synchronous send

Non-Blocking Operations (cont‘d)



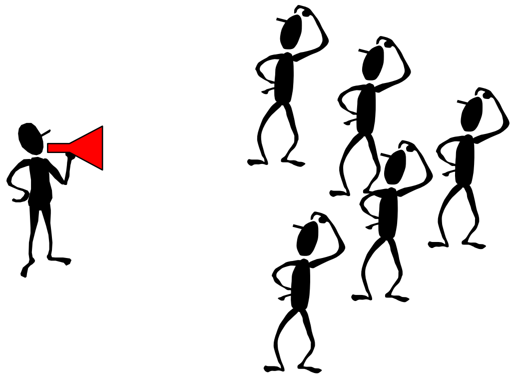
- All non-blocking operations must have matching wait (or test) operations. (Some system or application resources can be freed only when the non-blocking operation is completed.)
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.
- Non-blocking operations 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.

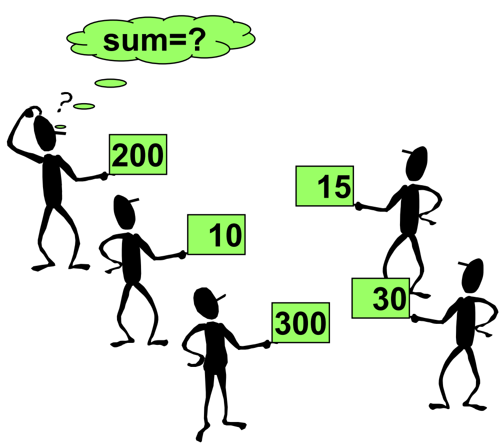
Broadcast

- A one-to-many communication.



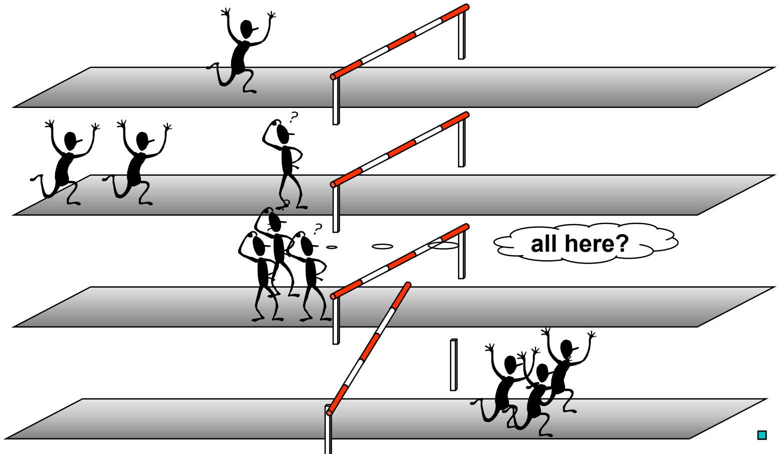
Reduction Operations

- 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 Forum

- MPI-2 Forum
  - Same procedure.
  - *MPI-2: Extensions to the Message-Passing Interface* document (July 18, 1997).
  - MPI 1.2 — mainly clarifications.
  - MPI 2.0 — extensions to MPI 1.2.

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.

Chap.2 Process Model and Language Bindings

1. MPI Overview
2. **Process model and language bindings**
   – **starting several MPI processes**
3. Messages and point-to-point communication
4. Non-blocking communication
5. Derived datatypes
6. Virtual topologies
7. Collective communication
8. All other MPI-1 features

MPI_Init()
MPI_Comm_rank()

Header files

- C

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

- Fortran

```
include ´mpif.h´
```

MPI Function Format

- C: error = MPI_Xxxxxx(parameter, ...);
  MPI_Xxxxxx(parameter, ...);
- Fortran: CALL MPI_XXXXXX(parameter, ..., *IERROR*)

forget absolutely never!

MPI Function Format Details

- Have a look into the MPI standard, e.g., MPI 1.1, page 20.
  Each MPI routine is defined:
  - language independent,
  - in several programming languages (C, Fortran, C++ [in MPI-2]).

Output arguments in 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);

- Last two pages of the standard is the MPI function index,
  - it is ±1 page inexact — test it, e.g., find MPI_Init!
- MPI_...... namespace is reserved for MPI constants and routines,
  i.e. application routines and variable names must not begin with MPI_ .

Initializing MPI

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

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

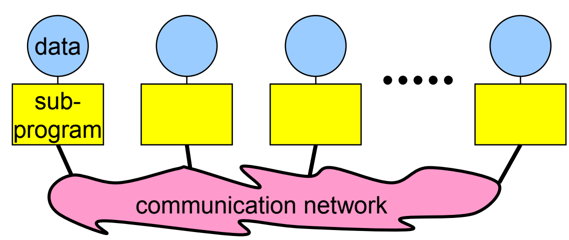
- Fortran: MPI_INIT(IERROR)
  INTEGER IERROR

```
program xxxxx
implicit none
include ´mpif.h´
integer ierror
call MPI_Init(ierror)
....
```

- Must be first MPI routine that is called.

Starting the MPI Program

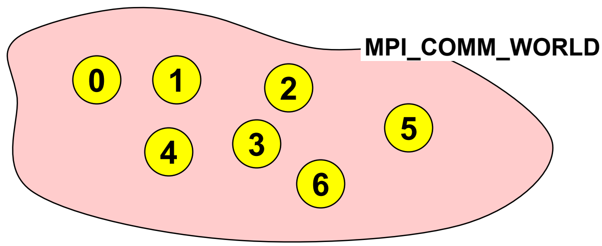
- Start mechanism is implementation dependent
- mpirun –np ***number_of_processes*** ./***executable*** (most implementations)
- mpiexec –n ***number_of_processes*** ./***executable*** (with MPI-2 standard)



- 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 mpif.h.
- Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)

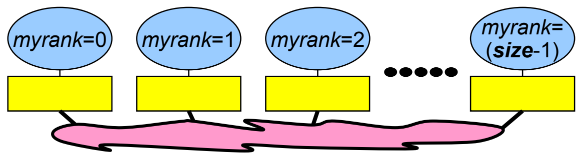


Handles

- Handles identify MPI objects.
- For the programmer, handles are
  - predefined constants in mpi.h or mpif.h
    - example: MPI_COMM_WORLD
    - predefined values exist only **after MPI_Init** was called
  - values returned by some MPI routines, to be stored in variables, that are defined as
    - in Fortran: INTEGER
    - in C: special MPI typedefs
- Handles refer to internal MPI data structures

Rank

- The rank identifies different processes.
- The rank is the basis for any work and data distribution.
- C: int MPI_Comm_rank(MPI_Comm comm, int **rank*)
- Fortran: MPI_COMM_RANK(comm, *rank*, *ierror*)
  INTEGER comm, rank, ierror



CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierror)

Size

- How many processes are contained within a communicator?
- C: int MPI_Comm_size(MPI_Comm comm, int **size*)
- Fortran: MPI_COMM_SIZE(comm, *size*, *ierror*)
  INTEGER comm, size, ierror

Exiting MPI

- C: int MPI_Finalize()
- Fortran: MPI_FINALIZE(*ierror*)
  INTEGER ierror

- Must be called last by all processes.

Exercise: Hello World

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

hint for C: #include <stdio.h>

- 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?

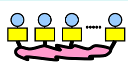
```
I am 2 of 4
Hello world
I am 0 of 4
I am 3 of 4
I am 1 of 4
```

see also login-slides

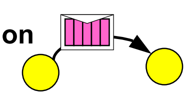
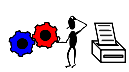
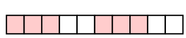
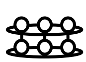
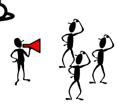
Advanced Exercises: Hello World with deterministic output

- 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.

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. Non-blocking communication 
5. Derived datatypes 
6. Virtual topologies 
7. Collective communication 
8. All other MPI-1 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**

MPI Basic Datatypes — C

MPI Datatype	**C datatype**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

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	

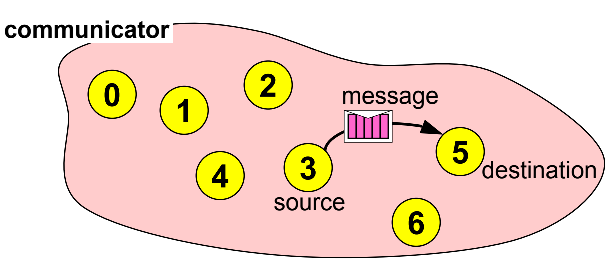
2345	**654**	**96574**	**-12**	**7676**

count=5
datatype=MPI_INTEGER

INTEGER arr(5)

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: 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*)
  <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.

Receiving a Message

- C: int MPI_Recv(void **buf*, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status **status*)
- Fortran: MPI_RECV(*BUF*, COUNT, DATATYPE, SOURCE, TAG, COMM, *STATUS*, *IERROR*)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM
  INTEGER STATUS(MPI_STATUS_SIZE), IERROR

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

Requirements for Point-to-Point Communications

For a communication to succeed:

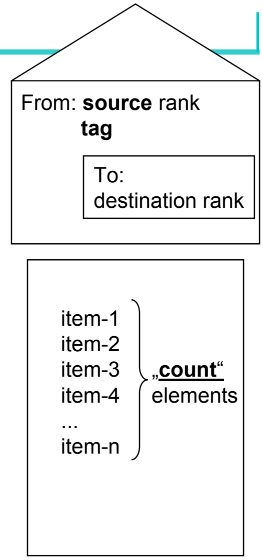
- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message datatypes must match.
- Receiver's buffer must be large enough.

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

- Envelope information is returned from MPI_RECV in *status*.
- C: status.MPI_SOURCE
  status.MPI_TAG
  count via MPI_Get_count()
- Fortran: status(MPI_SOURCE)
  status(MPI_TAG)
  count via MPI_GET_COUNT()



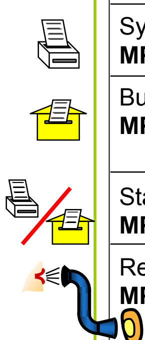
Receive Message Count

- C: int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
  int **count*)
- Fortran: MPI_GET_COUNT(STATUS, DATATYPE, *COUNT*, *IERROR*)
  INTEGER STATUS(MPI_STATUS_SIZE)
  INTEGER 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



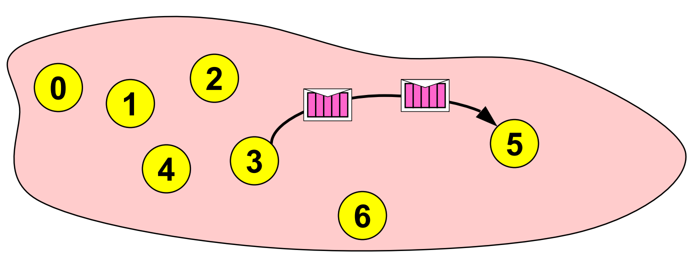
Sender mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Standard send **MPI_SEND**	Either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

Message Order Preservation

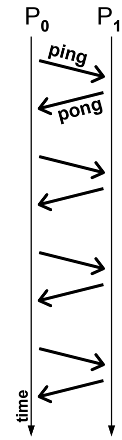
- Rule for messages on the same connection, i.e., same communicator, source, and destination rank:
- **Messages do not overtake each other.**
- This is true even for non-synchronous sends.



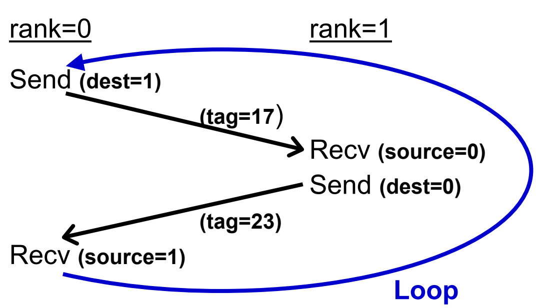
- If both receives match both messages, then the order is preserved.

Exercise — 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: *double MPI_Wtime*(void);
- Fortran: *DOUBLE PRECISION FUNCTION MPI_WTIME*()
- MPI_WTIME returns a wall-clock time in seconds.
- At process 0, print out the transfer time of **one** message
  - in seconds
  - in µs.



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
```

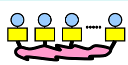
see also login-slides

Advanced Exercises — Ping pong latency and bandwidth

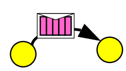
- latency = transfer time for short messages
- bandwidth = message size (in bytes) / transfer time

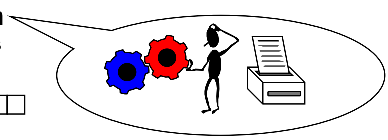
- Print out message transfer time and bandwidth
  - for following send modes:
    - **for standard send (MPI_Send)**
    - **for synchronous send (MPI_Ssend)**
  - for following message sizes:
    - **8 bytes (e.g., one double or double precision value)**
    - **512 B (= 8*64 bytes)**
    - **32 kB (= 8*64**2 bytes)**
    - **2 MB (= 8*64**3 bytes)**

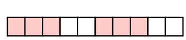
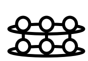
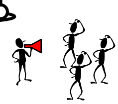
Chap.4 Non-Blocking Communication

1. MPI Overview 
2. Process model and language bindings

MPI_Init()
MPI_Comm_rank()

3. Messages and point-to-point communication 
4. **Non-blocking communication**
   - **to avoid idle time and deadlocks**



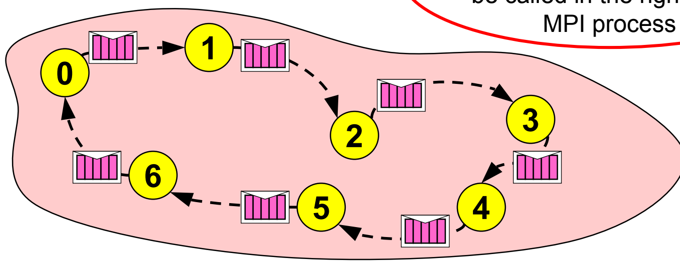
5. Derived datatypes 
6. Virtual topologies 
7. Collective communication 
8. All other MPI-1 features

Deadlock

- Code in each MPI process:

```
MPI_Ssend(…, right_rank, …)
MPI_Recv( …, left_rank,  …)
```

Will block and never return, because MPI_Recv cannot be called in the right-hand MPI process



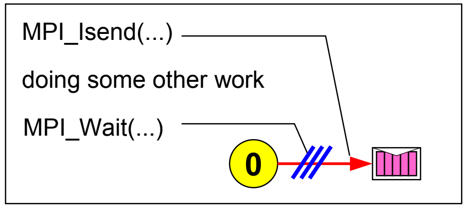
- Same problem with standard send mode (MPI_Send), if MPI implementation chooses synchronous protocol

Non-Blocking Communications

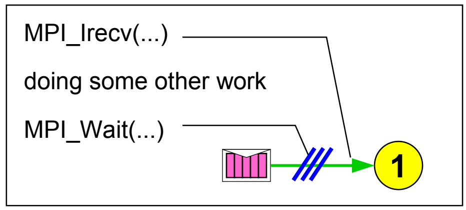
- Separate communication into three phases:
- Initiate non-blocking communication
  - returns **I**mmediately
  - routine name starting with MPI_**I**…
- Do some work (perhaps involving other communications?)
- Wait for non-blocking communication to complete

Non-Blocking Examples

- Non-blocking **send**



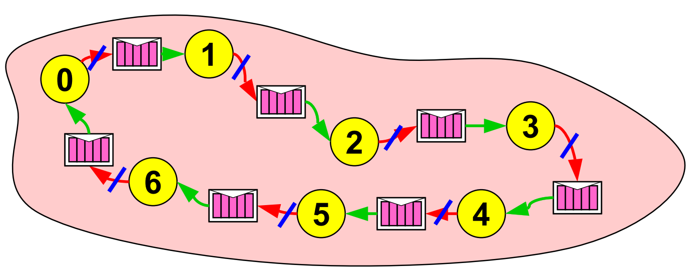
- Non-blocking **receive**



/// = waiting until operation locally completed

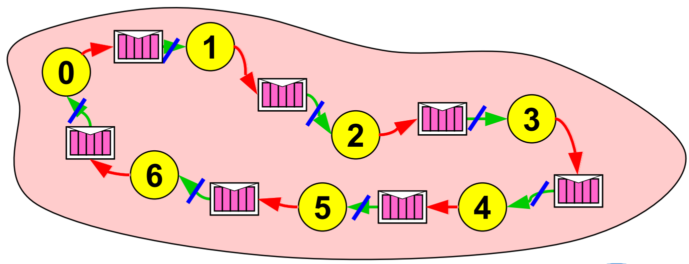
Non-Blocking Send

- Initiate non-blocking send
  - → in the ring example: Initiate non-blocking 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 non-blocking send to complete /



Non-Blocking Receive

- Initiate non-blocking receive
  → in the ring example: Initiate non-blocking 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 non-blocking receive to complete /



Handles, already known

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

- Handles **can** also be stored in local variables
  - memory for datatype handles – in C: MPI_Datatype
    – in Fortran: INTEGER
  - memory for communicator handles – in C: MPI_Comm
    – in Fortran: INTEGER

Request Handles

Request handles

- are used for non-blocking communication
- **must** be stored in local variables
  – in C: MPI_Request
  – in Fortran: INTEGER
- the value
  – **is generated** by a non-blocking communication routine
  – **is used** (and freed) in the MPI_WAIT routine

Non-blocking Synchronous Send

- C:

```
MPI_Issend( buf, count, datatype, dest, tag, comm,
            OUT &request_handle);
                    ↓
MPI_Wait( INOUT &request_handle, &status);
```

- Fortran:

```
CALL MPI_ISSEND( buf, count, datatype, dest, tag, comm,
                 OUT request_handle, ierror)
                           ↓
CALL MPI_WAIT(  INOUT request_handle, status, ierror)
```

- buf must not be used between Issend and Wait (in all progr. languages)
  MPI 1.1, page 40, lines 44-45
- "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 problems, see MPI-2, Chap. 10.2.2, pp 284-290

Non-blocking Receive

- C:

  MPI_Irecv (*buf*, count, datatype, source, tag, comm,
  OUT &*request_handle*);

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

- Fortran:

  CALL MPI_IRECV (*buf*, count, datatype, source, tag, comm,
  OUT *request_handle*, *ierror*)

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

- buf must not be used between Irecv and Wait (in all progr. languages)
- Fortran problems, see MPI-2, Chap. 10.2.2, pp 284-290
- e.g., compiler does not see modifications in buf in MPI_WAIT,
  workaround: call **MPI_ADDRESS**(buf, *iaddrdummy*, *ierror*) after MPI_WAIT

Non-blocking Receive and Register Optimization

- **Fortran:**

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

- **may be compiled as**

  MPI_IRECV (*buf*, ..., *request_handle*, *ierror*)
  registerA = buf
  MPI_WAIT(request_handle, *status*, *ierror*) **may receive data into *buf***
  write (*,*) **registerA**

- **i.e. old data is written instead of received data!**
- **Workarounds:**
  - *buf* may be allocated in a common block, or
  - calling **MPI_ADDRESS**(buf, *iaddr_dummy*, *ierror*) after MPI_WAIT

Non-blocking MPI routines and strided sub-arrays

- **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*)

- **Do not use non-contiguous sub-arrays in non-blocking 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!**

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive, and vice-versa.
- Non-blocking 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>non-blocking operation immediately followed by a matching wait</u> is equivalent to the <u>blocking operation,</u> except the Fortran problems.

Completion

- C:

  MPI_Wait(&request_handle, &*status*);

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

- Fortran:

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

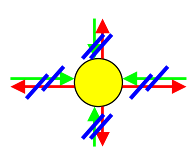
  CALL MPI_TEST(request_handle, *flag*, *status*, *ierror*)

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

Multiple Non-Blocking Communications

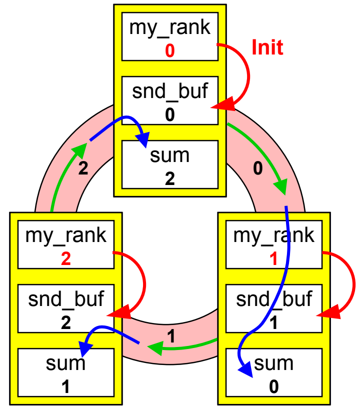
You have several request handles:

- 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**



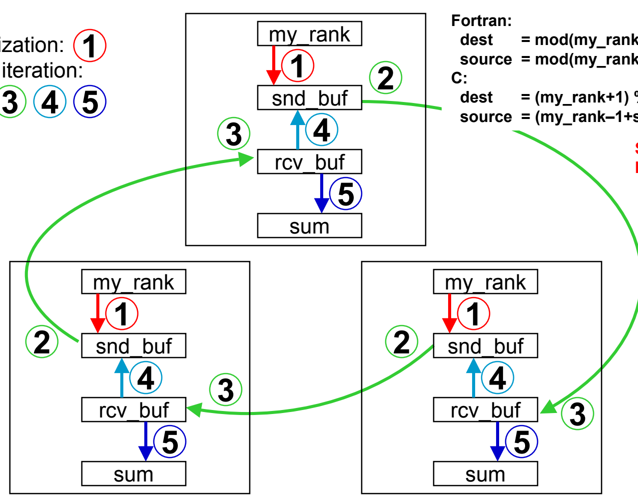
Exercise — Rotating information around a ring

- 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.
- Keep passing it around the ring until the value is back where it started, i.e.
- each process calculates sum of all ranks.
- Use non-blocking 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:
dest = (my_rank+1) % size;
source = (my_rank–1+size) % size;

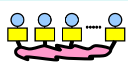
Single Program !!!

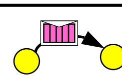
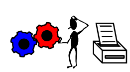
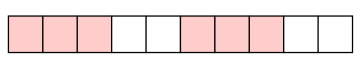
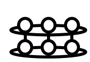
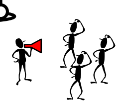
see also login-slides

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.

Chap.5 Derived Datatypes

1. MPI Overview 
2. Process model and language bindings

   MPI_Init()
   MPI_Comm_rank()
3. Messages and point-to-point communication 
4. Non-blocking communication 
5. **Derived datatypes** 
   - **transfer of any combination of typed data**
6. Virtual topologies 
7. Collective communication 
8. All other MPI-1 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_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

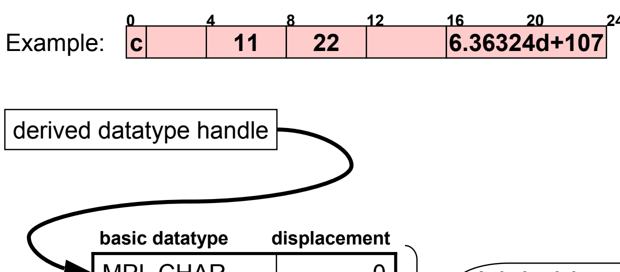
int				double				

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

Derived Datatypes — Type Maps

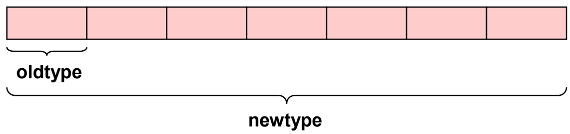


basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

A derived datatype describes the memory layout of, e.g., structures, common blocks, subarrays, some variables in the memory

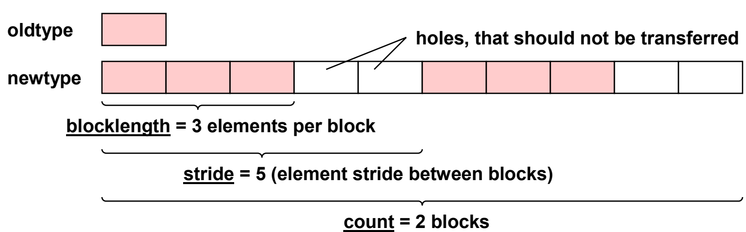
Contiguous Data

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype



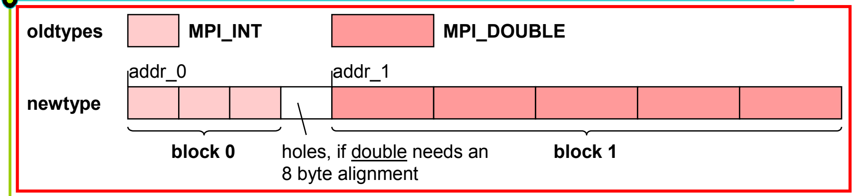
- C: int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype*)
- Fortran: MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, *NEWTYPE*, *IERROR*)
  INTEGER COUNT, OLDTYPE
  INTEGER NEWTYPE, IERROR

Vector Datatype



- C: int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype*)
- Fortran: MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, *NEWTYPE*, *IERROR*)
  INTEGER COUNT, BLOCKLENGTH, STRIDE
  INTEGER OLDTYPE, NEWTYPE, IERROR

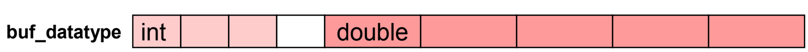
Struct Datatype



- C: int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *_newtype_)
- Fortran: MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, 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**	**)**

Memory Layout of Struct Datatypes



Fixed memory layout:

- C

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

- Fortran, common block

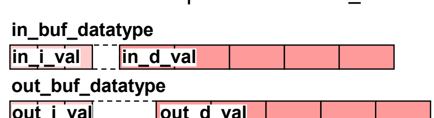
```
integer i_val(3)
double precision d_val(5)
common /bcomm/ i_val, d_val
```

- 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
```

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



How to compute the displacement

- array_of_displacements[i] := address(block_i) – address(block_0)
- MPI-1
  - C: int MPI_Address(void* location, MPI_Aint **address*)
  - Fortran: MPI_ADDRESS(LOCATION, *ADDRESS*, *IERROR*)

    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR
- MPI-2
  - C: int MPI_**Get_**address(void* location, MPI_Aint **address*)
  - Fortran: MPI_**GET_**ADDRESS(LOCATION, *ADDRESS*, *IERROR*)

    <type> LOCATION(*)
    INTEGER**(KIND=MPI_ADDRESS_KIND)** ADDRESS
    INTEGER IERROR

Committing a Datatype

- Before a dataytype handle is used in message passing communication, **it needs to be committed with MPI_TYPE_COMMIT**.
- This must be done only once.

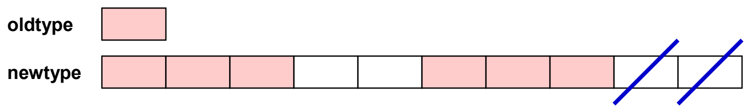
- C: int MPI_Type_commit(MPI_Datatype **datatype*);
- Fortran: MPI_TYPE_COMMIT(*DATATYPE*, *IERROR*)

  INTEGER DATATYPE, IERROR

IN-OUT argument

Size and Extent of a Datatype, I.

- Size := number of bytes that have to be transferred.
- Extent := spans from first to last byte.
- Basic datatypes: Size = Extent = number of bytes used by the compiler.
- Derived datatypes, an example:



size := 6 * size(oldtype)

extent := 8 * extent(oldtype)

better visualization of newtype:

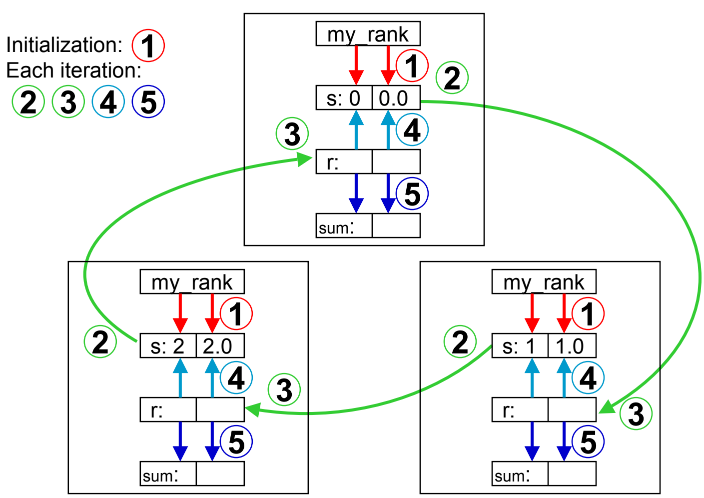
Size and Extent of a Datatype, II.

- MPI-1:
  - C: int MPI_Type_size(MPI_Datatype datatype, int **size*)
    int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint **extent*)
  - Fortran: MPI_TYPE_SIZE(DATATYPE, *SIZE*, *IERROR*)
    INTEGER DATATYPE, SIZE, IERROR
    MPI_TYPE_EXTENT(DATATYPE, *EXTENT*, *IERROR*)
    INTEGER DATATYPE, EXTENT, IERROR
- MPI-2:
  - C: int MPI_Type_get_extent(MPI_Datatype datatype,
    MPI_Aint **lb*, MPI_Aint **extent*)
  - Fortran: MPI_TYPE_GET_EXTENT(DATATYPE, *LB*, *EXTENT*, *IERROR*)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT

Exercise — Derived Datatypes

- Modify the pass-around-the-ring exercise.
- Use your own result from Chap. 4 or copy our solution:
  cp ~/MPI/course/**F**/Ch4/ring.f .
  cp ~/MPI/course/**C**/Ch4/ring.c .
- 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.

Exercise — Derived Datatypes

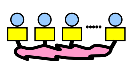
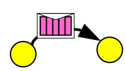
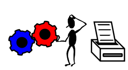
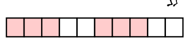


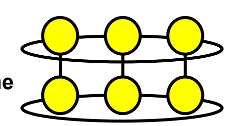
see also login-slides

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-1 standard. (You can find MPI_Sendrecv by looking at the function index on the last page 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.**

Chap.6 Virtual Topologies

1. MPI Overview 
2. Process model and language bindings — **MPI_Init()** **MPI_Comm_rank()**
3. Messages and point-to-point communication 
4. Non-blocking communication 
5. Derived datatypes 
6. **Virtual topologies**
   - **a multi-dimensional process naming scheme**



7. Collective communication 
8. All other MPI-1 features

Example

- Global array A(1:3000, 1:4000, 1:500) = 6•10⁹ 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•10⁹ words

- **process coordinates:** handled with **virtual Cartesian topologies**
- Array decomposition: handled by the application program directly

Virtual Topologies

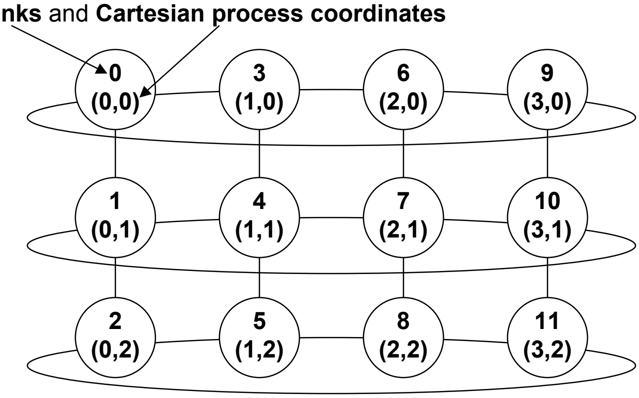
- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimize communications.

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,
  - not covered here.

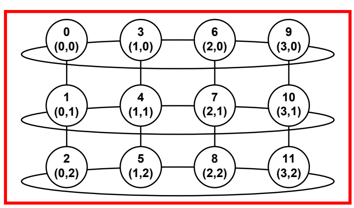
Creating a Cartesian Virtual Topology

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

- Fortran: MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS,
  REORDER, *COMM_CART*, *IERROR*)

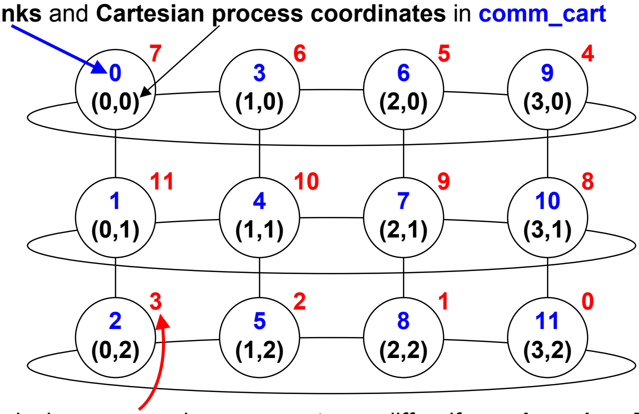
  INTEGER COMM_OLD, NDIMS, DIMS(*)
  LOGICAL PERIODS(*), REORDER
  INTEGER COMM_CART, IERROR

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



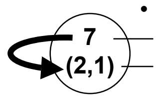
Example – A 2-dimensional Cylinder

- **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



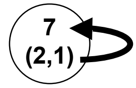
- Mapping
  ranks to
  process grid coordinates

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

- Fortran: MPI_CART_COORDS(COMM_CART, RANK,
  MAXDIMS, *COORDS*, *IERROR*)
  INTEGER COMM_CART, RANK
  INTEGER MAXDIMS, COORDS(*), IERROR

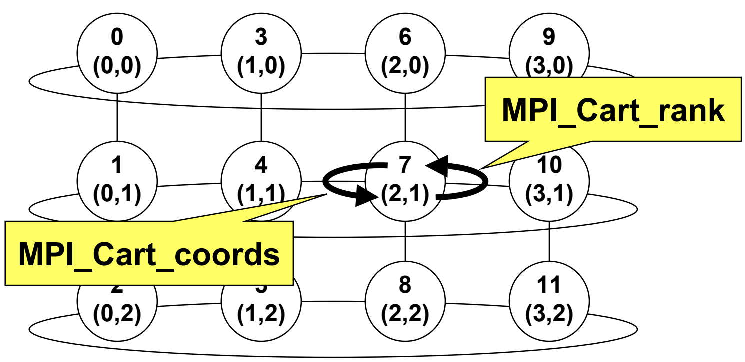
Cartesian Mapping Functions

- Mapping process grid coordinates to ranks



- C: int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *_rank_)
- Fortran: MPI_CART_RANK(COMM_CART, COORDS, _RANK_, _IERROR_)
  INTEGER COMM_CART, COORDS(*)
  INTEGER RANK, IERROR

Own coordinates



- Each process gets its own coordinates with
  MPI_Comm_rank(comm_cart, ***my_rank***, _ierror_)
  MPI_Cart_coords(comm_cart, **my_rank**, maxdims, ***my_coords***, _ierror_)

Cartesian Mapping Functions

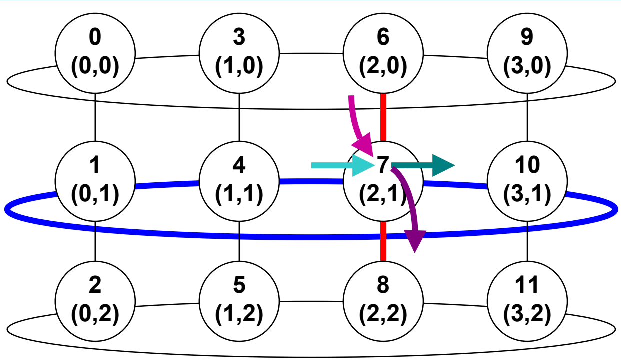
- Computing ranks of neighboring processes

- 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*)
  INTEGER COMM_CART, DIRECTION
  INTEGER DISP, RANK_SOURCE
  INTEGER RANK_DISP, 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 noop!

MPI_Cart_shift – Example



invisible input argument: **my_rank** in cart

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

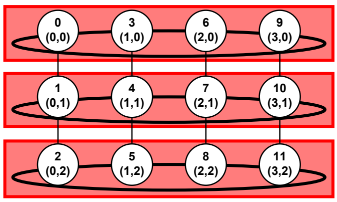
	direction	displace	rank_source	rank_dest
example on	**0** or	**+1**	**4**	**10**
process rank=**7**	**1**	**+1**	**6**	**8**

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: int MPI_Cart_sub(MPI_Comm comm_cart, int *remain_dims, MPI_Comm **comm_slice*)

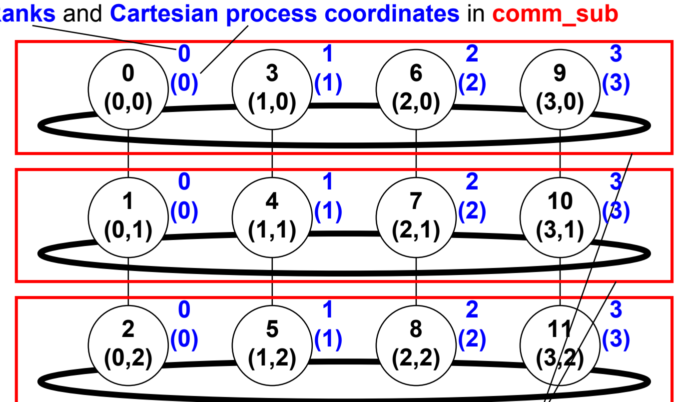
- Fortran: MPI_CART_SUB(COMM_CART, REMAIN_DIMS, *COMM_SLICE*, *IERROR*)




```
INTEGER  COMM_CART
LOGICAL   REMAIN_DIMS(*)
INTEGER  COMM_SLICE, IERROR
```


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

- Rewrite the pass-around-the-ring program using a one-dimensional ring topology.
- Use the results from Chap. 4 (non-blocking, without derived datatype):
  - ~/MPI/course/**F**/Ch4/ring.f
  - ~/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
    - → **`dims`** and **`period`** as normal variables, i.e., no arrays
    - → coordinates are not necessary, because coord==rank

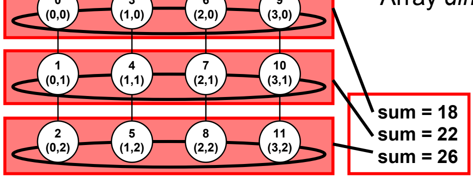
see also login-slides

Advanced Exercises — Two-dimensional topology

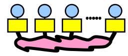
- Rewrite the exercise in two dimensions, as a cylinder.
- Each row of the cylinder, i.e. each ring, should compute its own separate sum of the original ranks in the two dimensional comm_cart.
- Compute the two dimensional factorization with MPI_Dims_create().

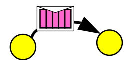
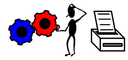
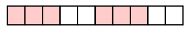
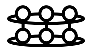
- C: int MPI_Dims_create(int nnodes, int ndims, int **dims*)
- Fortran: MPI_DIMS_CREATE(NNODES, NDIMS, *DIMS*, *IERROR*)
  INTEGER NNODES, NDIMS, DIMS(*)
  INTEGER IERROR

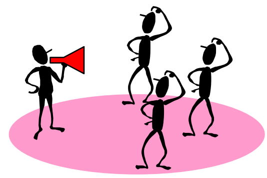
Array *dims* must be **initialized** with **(0,0)**



Chap.7 Collective Communication

1. MPI Overview 
2. Process model and language bindings

   MPI_Init()
   MPI_Comm_rank()
3. Messages and point-to-point communication 
4. Non-blocking communication 
5. Derived datatypes 
6. Virtual topologies 
7. **Collective communication**
   - **e.g., broadcast**



8. All other MPI-1 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.

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.
- All collective operations are blocking.
- No tags.
- Receive buffers must have exactly the same size as send buffers.

Barrier Synchronization

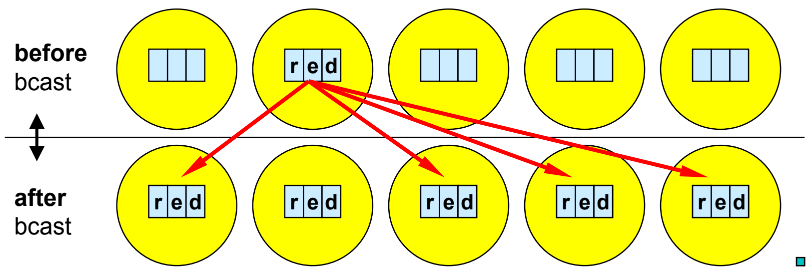
- C: int MPI_Barrier(MPI_Comm comm)
- Fortran: MPI_BARRIER(COMM, *IERROR*)
  INTEGER COMM, IERROR

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

- Fortran: MPI_Bcast(BUF, COUNT, DATATYPE, ROOT, COMM, *IERROR*)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, ROOT
  INTEGER COMM, IERROR

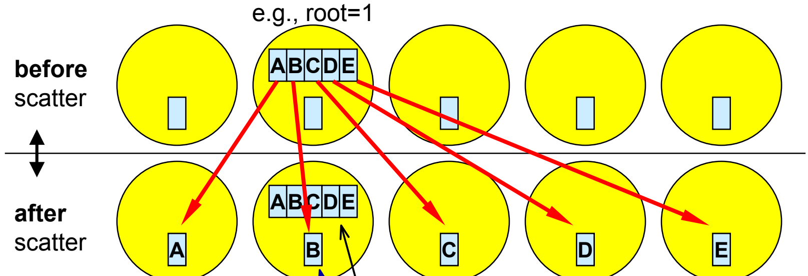


e.g., root=1

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

Scatter

e.g., root=1

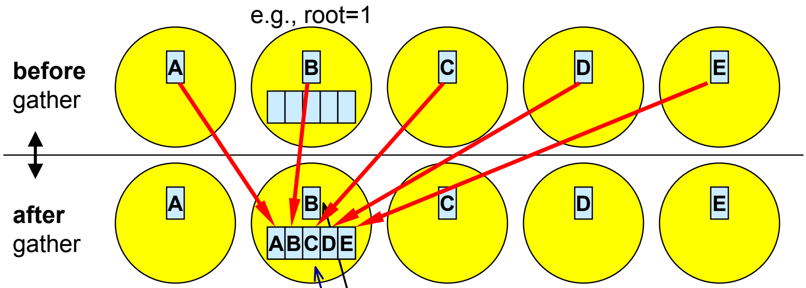


- C: int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void ****recvbuf***, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)

- Fortran: MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, ***RECVBUF***,
  RECVCOUNT, RECVTYPE, ROOT, COMM, ***IERROR***)
  <type> SENDBUF(*), RECVBUF(*)
  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE
  INTEGER ROOT, COMM, IERROR

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

Gather



- C:
```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```
- Fortran:
```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
           RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type>     SENDBUF(*), RECVBUF(*)
INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE
INTEGER    ROOT, COMM, IERROR
```


Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- $d_0 \circ d_1 \circ d_2 \circ d_3 \circ \ldots \circ d_{s-2} \circ d_{s-1}$
  - d_i = data in process rank i
    - **single variable, or**
    - **vector**
  - $\circ$ = associative operation
  - Example:
    - **global sum or product**
    - **global maximum or minimum**
    - **global user-defined operation**
- floating point rounding may depend on usage of associative law:
  - $[(d_0 \circ d_1) \circ (d_2 \circ d_3)] \circ [\ldots \circ (d_{s-2} \circ d_{s-1})]$
  - $((((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \ldots) \circ d_{s-2}) \circ d_{s-1})$

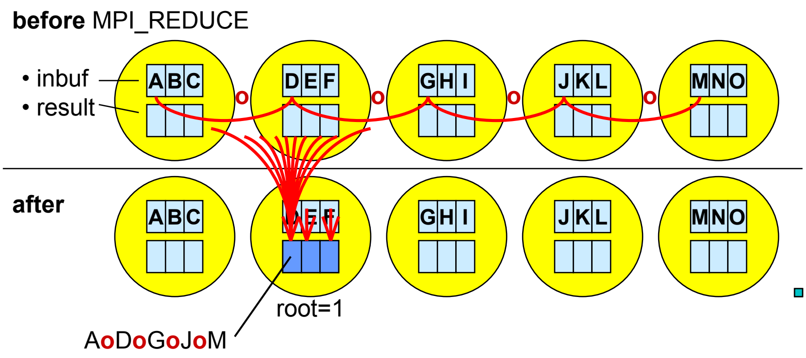
Example of Global Reduction

- Global integer sum.
- Sum of all inbuf values should be returned in *resultbuf.*
- C: root=0;
  MPI_Reduce(&inbuf, &*resultbuf*, 1, MPI_INT, MPI_SUM,
  root, MPI_COMM_WORLD);
- 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



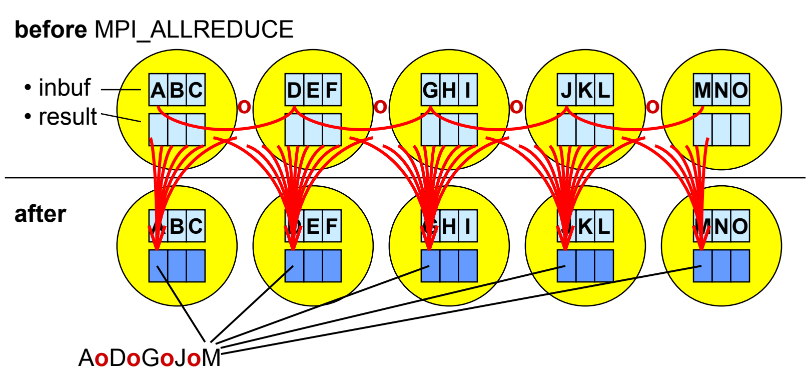
User-Defined Reduction Operations

- 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-1 standard
- Registering a user-defined reduction function:
  - C: MPI_Op_create(MPI_User_function *func, int commute, MPI_Op **op*)
  - Fortran: MPI_OP_CREATE(FUNC, COMMUTE, *OP*, *IERROR*)
- COMMUTE tells the MPI library whether FUNC is commutative.

Variants of Reduction Operations

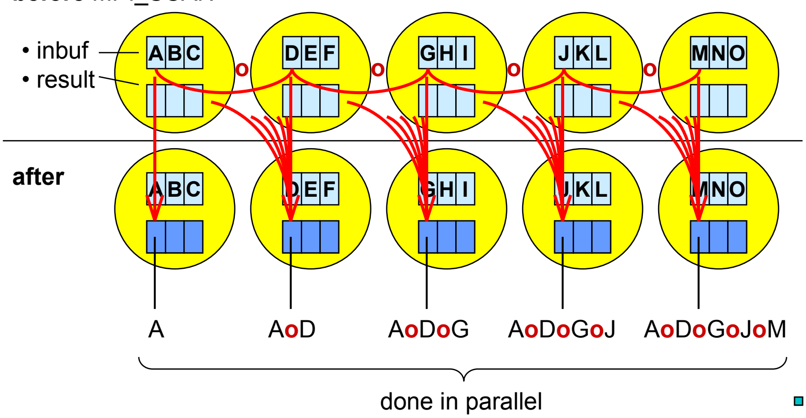
- MPI_ALLREDUCE
  - no root,
  - returns the result in all processes
- MPI_REDUCE_SCATTER
  - result vector of the reduction operation is scattered to the processes into the real result buffers
- MPI_SCAN
  - prefix reduction
  - result at process with rank i := reduction of inbuf-values from rank 0 to rank i

MPI_ALLREDUCE



MPI_SCAN

before MPI_SCAN



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**/Ch4/ring.f
  ~/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.

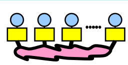
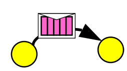
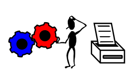
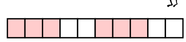
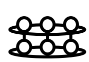
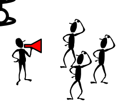
see also login-slides

Advanced Exercises — Global scan and sub-groups

- Global scan:
  - Rewrite the last program so that each process computes a partial sum.
  - Rewrite in a way that each process prints out its partial result in the correct order:

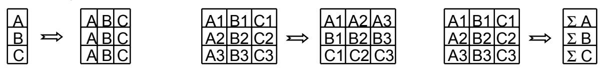
    rank=0 → sum=0
    rank=1 → sum=1
    rank=2 → sum=3
    rank=3 → sum=6
    rank=4 → sum=10
  - This can be done, e.g., by sending a token (empty message) from process 0 to process 1, from 1 to 2, and so on (expecting that all MPI-processes' stdout are synchronously merged to the program's stdout).
- Global sum in sub-groups:
  - Rewrite the result of the advanced exercise of chapter 6.
  - Compute the sum in each slice with the global reduction.

Chap.8 All Other MPI-1 Features

1. MPI Overview 
2. Process model and language bindings **MPI_Init()** **MPI_Comm_rank()**
3. Messages and point-to-point communication 
4. Non-blocking communication 
5. Derived datatypes 
6. Virtual topologies 
7. Collective communication 
8. **All other MPI-1 features**

Other MPI features (1)

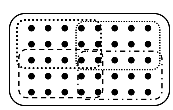
- Point-to-point
  - MPI_Sendrecv & MPI_Sendrecv_replace (see advanced exercise of Chap. 5)
  - Null processes, MPI_PROC_NULL (see Chap. 7, slide on MPI_Cart_shift)
  - MPI_Pack & MPI_Unpack
  - MPI_Probe: check length (tag, source rank) before calling MPI_Recv
  - MPI_Iprobe: check whether a message is available
  - MPI_Request_free, MPI_Cancel
  - Persistent requests
  - MPI_BOTTOM (in point-to-point and collective communication)
- Collective Operations
  - MPI_Allgather – MPI_Alltoall – MPI_Reduce_scatter



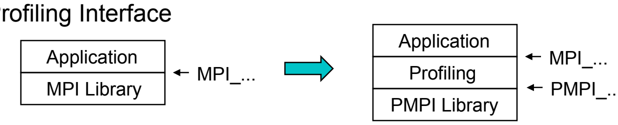
  - MPI_..........v (Gatherv, Scatterv, Allgatherv, Alltoallv)
- Topologies
  - MPI_DIMS_CREATE (see advanced exercise of Chap. 7)

Other MPI features (2)

- Groups of processes and their communicators
  - subgroups / subcommunicators
  - intracommunicator / intercommunicator



- Attribute caching
- Environmental management
  - inquire MPI_TAG_UB, MPI_HOST, MPI_IO, MPI_WTIME_IS_GLOBAL (tag = 0…32767 always possible)
- Profiling Interface



- Each generated handle can be freed.
- Lower and upper bound marker in derived datatypes:
  - reviewed and modified in MPI-2 – MPI_Type_create_resized()

Other MPI features (3)

- Error Handling
  - 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_Errhandler_set (comm, MPI_ERRORS_RETURN, ierr)**
      C: MPI_Errhandler_set (comm, MPI_ERRORS_RETURN);
      then
      - **ierror returned by each MPI routine**
      - **undefined state after an erroneous MPI call has occurred (only MPI_ABORT(…) should be still callable)**

MPI provider

- The vendor of your computers
- The network provider (e.g. with MYRINET)
- MPICH – the public domain MPI library from Argonne
  - for all UNIX platforms
  - for Windows NT, ...
- LAM – another public domain MPI library
- see also at www.lam-mpi.org/mpi/implementations/
  - list of MPI implementations
- other info at www.hlrs.de/mpi/

Summary

MPI-1

- Parallel MPI process model
- Message passing
  - blocking → several modes (**standard, buffered, synchronous,** ready)
  - non-blocking
    - → to allow message passing from all processes <u>in parallel</u>
    - → to avoid deadlocks
  - derived datatypes
    - → to transfer any combination of data in <u>one</u> message
- Virtual topologies → a convenient processes naming scheme
- Collective communications → a major chance for optimization
- Overview on other MPI-1 features