Introduction to the Message Passing Interface (MPI)

Rolf Rabenseifner
rabenseifner@hlrs.de

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

Questions from Participants

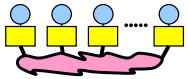
- Not yet an MPI user
  - Why should I use MPI?

  …
  …

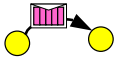
- How can I use
  - Derived datatypes?
  - Virtual topology communicators?
  - Non-blocking routines?

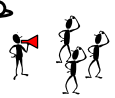
Parallel Programming Models

Chap.0 Parallel Programming Models



MPI_Init()
MPI_Comm_rank()





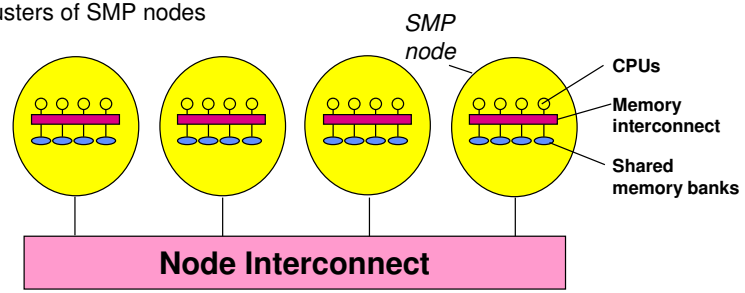
Acknowledgments

- **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.**

- **Further contributors:**
  - Michael Resch
  - Alfred Geiger
  - Matthias Müller

Hybrid architectures

- Most modern high-performance computing (HPC) systems are clusters of SMP nodes



- SMP (symmetric multi-processing) inside of each node
- DMP (distributed memory parallelization) on the node interconnect

Why?

- Why should I use parallel hardware architectures?
- Possible answers:
  - The response of only one processor is **not** just in time
  - Moore's Law:
    - **The number of transistors on a chip will double approximately every 18 month**
    - **➔ in the future, the number of processors on a chip will grow**
  - You own a
    - **network of workstations (NOW)**
    - **Beowulf-class systems = Clusters of Commercial Off-The-Shelf (COTS) PCs**
    - **a dual-board or quad-board PC**
  - Huge application with huge memory needs

Parallelization strategies — hardware resources

- Two major resources of computation:
  - processor
  - memory
- Parallelization means
  - **distributing work** to processors
  - **distributing data** (if memory is distributed)

  and
  - **synchronization** of the distributed work
  - **communication** of *remote* data to *local* processor (if memory is distr.)
- Programming models offer a combined method for
  - distribution of work & data, synchronization and communication

Distributing Work & Data

Work decomposition

- based on loop decomposition

do i=1,100
➔ i=1,25
i=26,50
i=51,75
i=76,100

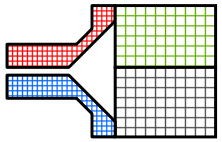
Data decomposition

- all work for a local portion of the data is done by the local processor

A(1:20, 1: 50)
A(1:20, 51:100)
A(21:40, 1: 50)
A(21:40, 51:100)

Domain decomposition

- decomposition of work and data is done in a higher model, e.g. in the reality



Synchronization

```
Do i=1,100                    i=1..25 | 26..50 | 51..75 | 76..100
    a(i) = b(i)+c(i)              execute on the 4 processors
Enddo
                              BARRIER synchronization
Do i=1,100
    d(i) = 2*a(101-i)         i=1..25 | 26..50 | 51..75 | 76..100
Enddo                             execute on the 4 processors
```

- **Synchronization**
  - is necessary
  - may cause
    - **idle time on some processors**
    - **overhead to execute the synchronization primitive**

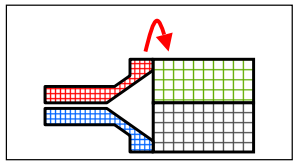
Communication

```
Do i=2,99
    b(i) = a(i) + f*(a(i-1)+a(i+1)-2*a(i))
Enddo
```

- **Communication** is necessary on the boundaries
  - e.g. b(26) = a(26) + f*(a(25)+a(27)-2*a(26))

a(1:25),	b(1:25)
a(26,50),	b(51,50)
a(51,75),	b(51,75)
a(76,100),	b(76,100)

  - e.g. at domain boundaries



Major Programming Models

1. **OpenMP**
   - Shared Memory **Directives**
   - to define the work decomposition
   - no data decomposition
   - synchronization is implicit (can be also user-defined)
2. **HPF (High Performance Fortran)**
   - Data Parallelism
   - User specifies data decomposition with **directives**
   - Communication (and synchronization) is implicit
3. **MPI (Message Passing Interface)**
   - User specifies how work & data is distributed
   - User specifies how and when communication has to be done
   - by calling MPI communication **library-routines**

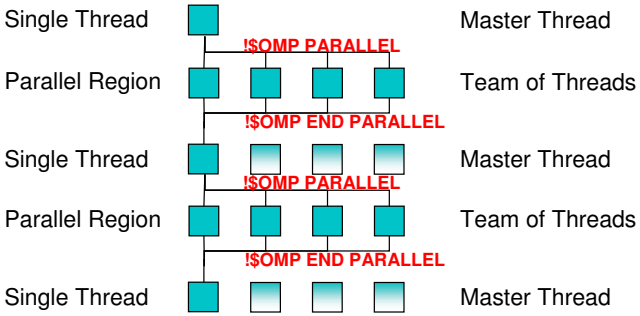
OpenMP

Shared Memory Directives – OpenMP, I.

```
Real :: A(n,m), B(n,m)                    Data definition

!$OMP PARALLEL DO
do j = 2, m-1                             Loop over y-dimension
  do i = 2, n-1                           Vectorizable loop over x-dimension
    B(i,j) = ... A(i,j)                   Calculate B,
             ... A(i-1,j) ... A(i+1,j)      using upper and lower,
             ... A(i,j-1) ... A(i,j+1)        left and right value of A
  end do
end do
!$OMP END PARALLEL DO
```

Shared Memory Directives – OpenMP, II.



Shared Memory Directives – OpenMP, III.

- OpenMP
  - **standardized shared memory parallelism**
  - **thread-based**
  - **the user has to specify the work distribution explicitly with directives**
  - **no data distribution, no communication**
  - **mainly loops can be parallelized**
  - **compiler translates OpenMP directives into thread-handling**
  - **standardized since 1997**
- Automatic SMP-Parallelization
  - **e.g., Compas (Hitachi), Autotasking (NEC)**
  - **thread based shared memory parallelism**
  - **with directives (similar programming model as with OpenMP)**
  - **supports automatic parallelization of loops**
  - **similar to automatic vectorization**

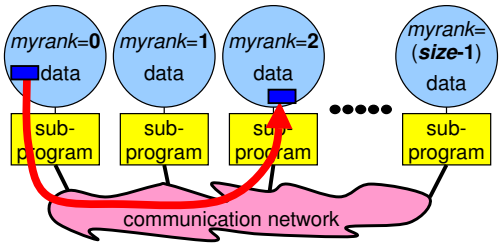
MPI

Major Programming Models – MPI

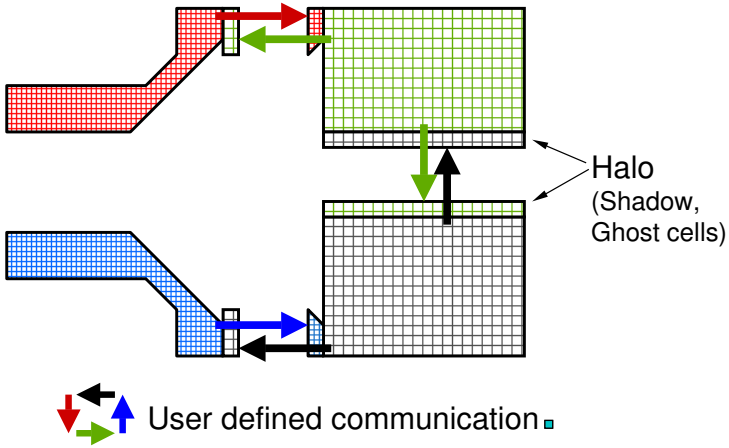
1. **OpenMP**
   - Shared Memory **Directives**
   - to define the work decomposition
   - no data decomposition
   - synchronization is implicit (can be also user-defined)
2. **HPF (High Performance Fortran)**
   - Data Parallelism
   - User specifies data decomposition with **directives**
   - Communication (and synchronization) is implicit
3. **MPI (Message Passing Interface)**
   - User specifies how work & data is distributed
   - User specifies how and when communication has to be done
   - by calling MPI communication **library-routines**

Message Passing Program Paradigm – MPI, I.

- 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)
- All work and data distribution is based on value of ***myrank***
  - returned by special library routine
- Communication via special send & receive routines (***message passing***)



Additional Halo Cells – MPI, II.



Message Passing – MPI, III.

```
Call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
Call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierror)
m1 = (m+size-1)/size;   ja=1+m1*myrank;   je=max(m1*(myrank+1), m)
jax=ja-1;  jex=je+1   // extended boundary with halo

Real :: A(n, jax:jex), B(n, jax:jex)          ➡ Data definition
do j = max(2,ja), min(m-1,je)                 ➡ Loop over y-dimension
  do i = 2, n-1                               ➡ Vectorizable loop over x-dimension
    B(i,j) = ... A(i,j)                       ➡ Calculate B,
             ... A(i-1,j) ... A(i+1,j)        ➡   using upper and lower,
             ... A(i,j-1) ... A(i,j+1)        ➡         left and right value of A
  end do
end do

Call MPI_Send(.......)  ! - sending the boundary data to the neighbors
Call MPI_Recv(.......)  ! - receiving from the neighbors,
                        !   storing into the halo cells
```


Summary — MPI, IV.

- MPI (Message Passing Interface)
  - standardized distributed memory parallelism with message passing
  - process-based
  - the user has to specify the work distribution & data distribution & all communication
  - synchronization implicit by completion of communication
  - the application processes are calling MPI library-routines
  - compiler generates normal sequential code
  - typically domain decomposition is used
  - communication across domain boundaries
  - standardized
    MPI-1: Version 1.0 (1994), 1.1 (1995), 1.2 (1997)
    MPI-2: since 1997

Limitations, I.

- Automatic Parallelization
  - the compiler
    - **has no global view**
    - **cannot detect independencies, e.g., of loop iterations**
    - **➔ parallelizes only parts of the code**
  - only for shared memory and ccNUMA systems, see OpenMP
- OpenMP
  - only for shared memory and ccNUMA systems
  - mainly for loop parallelization with directives
  - only for medium number of processors
  - explicit domain decomposition also via rank of the threads

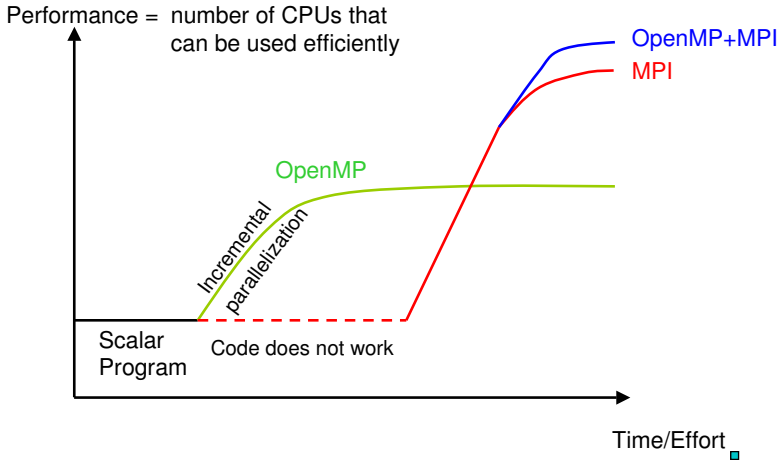
Limitations, II.

- HPF
  - set-compute-rule may cause a lot of communication
  - HPF-1 (and 2) not suitable for irregular and dynamic data
  - JaHPF may solve these problems, but with additional programming costs
  - can be used on any platform
- MPI
  - the amount of your hours available for MPI programming
  - can be used on any platform, but communication overhead on shared memory systems

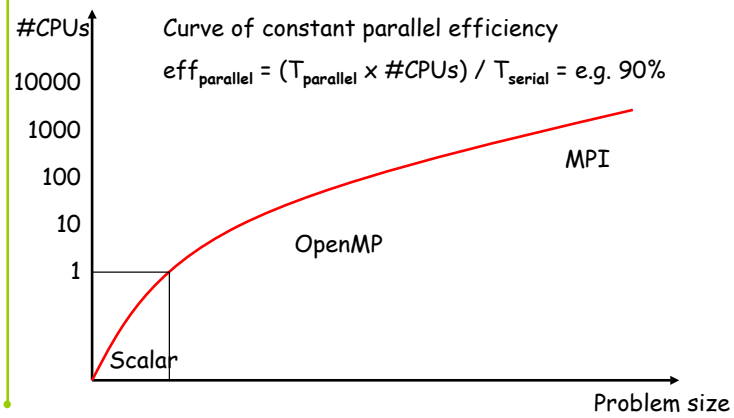
Advantages and Challenges

	OpenMP	HPF	MPI
Maturity of programming model	++	+	++
Maturity of standardization	+	+	++
Migration of serial programs	++	0	– –
Ease of programming (new progr.)	++	+	–
Correctness of parallelization	–	++	– –
Portability to any hardware architecture	–	++	++
Availability of implementations of the stand.	+	+	++
Availability of parallel libraries	0	0	0
Scalability to hundreds/thousands of processors	– –	0	++
Efficiency	–	0	++
Flexibility – dynamic program structures	–	–	++
– irregular grids, triangles, tetra-hedrons, load balancing, redistribut.	–	–	++

Comparing MPI and OpenMP – Programming effort

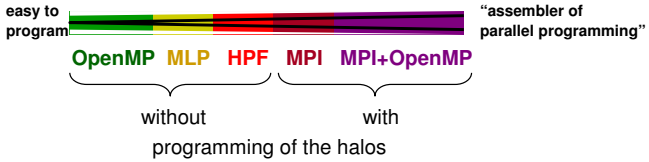


Number of CPUs / Problem size



Which Model is the Best for Me?

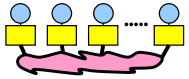
- Depends on
  - your application
  - Your problem size
  - your platform
  - which efficiency do you need on your platform
  - how much time do you want to spent on parallelization



MPI-1 Overview

Chap.1 MPI Overview

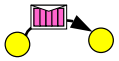
0. **Parallel Programming Models**
1. **MPI Overview**
   - **one program on several processors**
   - **work and data distribution**
   - **the communication network**



2. Process model and language bindings

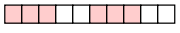
MPI_Init()
MPI_Comm_rank()

3. Messages and point-to-point communication



4. Non-blocking communication

5. Derived datatypes



6. Virtual topologies

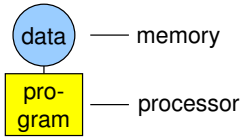
7. Collective communication

Information about MPI

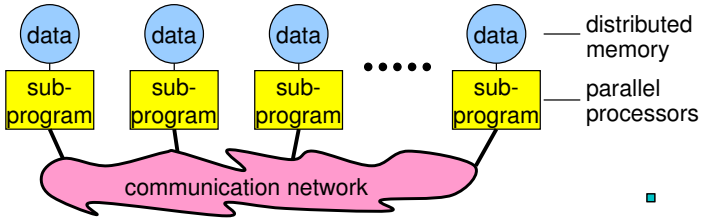
- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18,1997)
- Marc Snir and William Gropp et al.:
  MPI: The Complete Reference. (2-volume set).The MIT Press, 1998.
  (excellent catching up of the standard MPI-1.2 and MPI-2 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).*
- Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti: **Parallel Programming with MPI**. Handbook from EPCC.
  http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.pdf
  (Can be used together with these slides)
- **http://www.hlrs.de/organization/par/par_prog_ws/ → Online courses**
- http://www.hlrs.de/mpi/

The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**

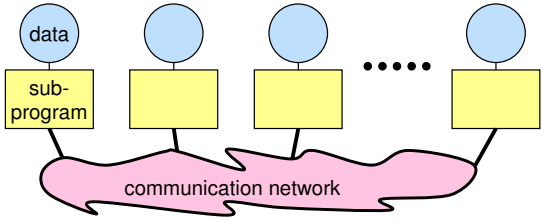


- **Message-Passing Programming Paradigm**



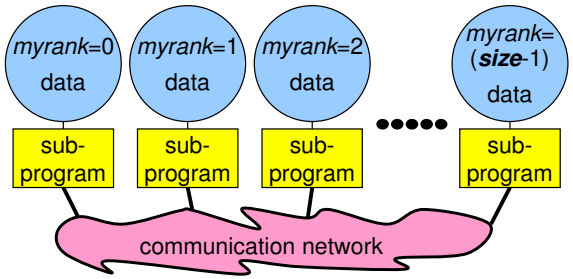
The Message-Passing Programming Paradigm

- 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***)

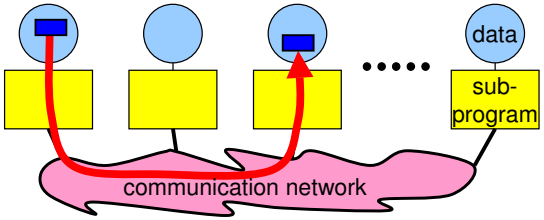


Data and Work Distribution

- 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



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
  - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool

Addressing

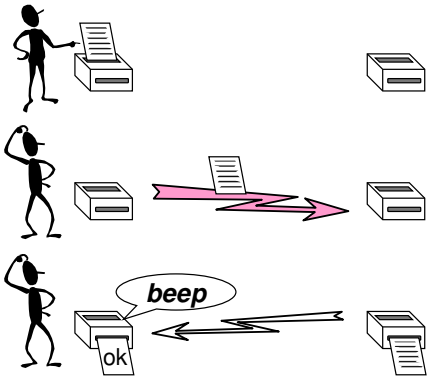
- Messages need to have addresses to be sent to.
- Addresses are similar to:
  - mail addresses
  - phone number
  - fax number
  - etc.
- MPI: addresses are ranks of the MPI processes (sub-programs)

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
- Different local routine interfaces:
  - blocking
  - non-blocking

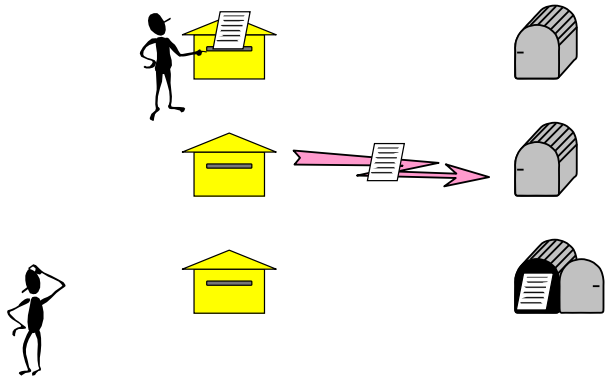
Synchronous Sends

- 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.

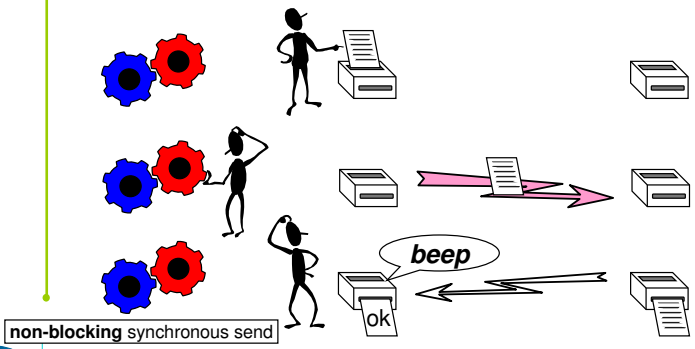


Blocking Operations

- 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 Operations (cont'd)

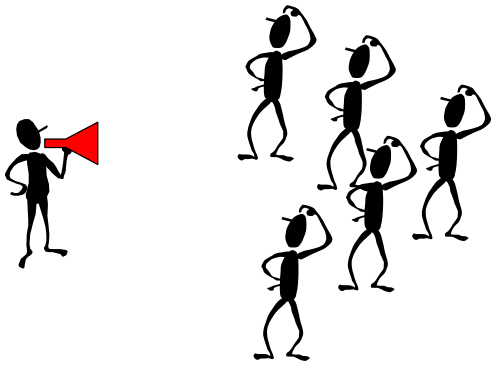
- All non-blocking operations must have matching wait (or test) operations. (Some system or application resources can be freed only when the non-blocking operation is completed.)
- A <u>non-blocking operation immediately followed by a matching wait</u> is equivalent to a <u>blocking operation</u>.
-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

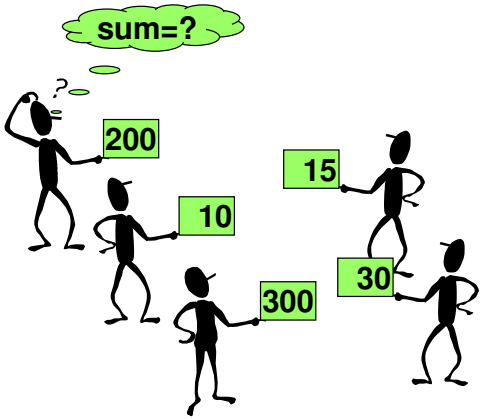
Broadcast

- A one-to-many communication.



Reduction Operations

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



MPI-2

- MPI-2, standard since July 18, 1997
- Chapters:
  - Version 1.2 of MPI (Version number, Clarifications)
  - Miscellany (Info Object, Language Interoperability, New Datatype Constructors, Canonical Pack & Unpack, C macros)
  - Process Creation and Management (MPI_Spawn, ...)
  - One-Sided Communications
  - Extended Collective Operations
  - External interfaces (..., MPI and Threads, ...)
  - I/O
  - Language Binding (C++, Fortran 90)
- All documents from http://www.mpi-forum.org/
  (or from www.hlrs.de/mpi/)

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.

Process Model & Lang.Bindings

Chap.2 Process Model and Language Bindings

0. **Parallel Programming Models**
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. Non-blocking communication
5. Derived datatypes
6. Virtual topologies
7. Collective communication

Header files & MPI Function Format

C

- C:
```
#include <mpi.h>
error = MPI_Xxxxxx( parameter, ... );
MPI_Xxxxxx( parameter, ... );
```

Fortran

- Fortran:
```
include ´mpif.h´
CALL MPI_XXXXXX( parameter, ..., IERROR )
```

forget absolutely never!

Initializing MPI

C

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

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

Fortran

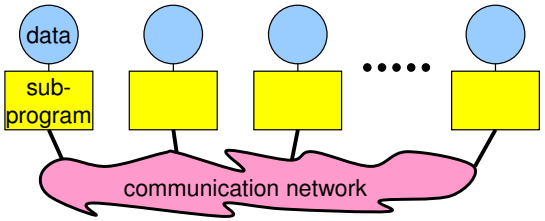
- Fortran: MPI_INIT(IERROR)
  INTEGER IERROR

```
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

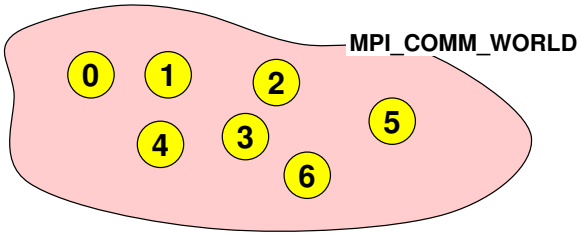
- Start mechanism is implementation dependent
- mpirun –np ***number_of_processes*** ./***executable*** (most implementations)
- mpiexec –n ***number_of_processes*** ./***executable*** (with MPI-2 standard)



- 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

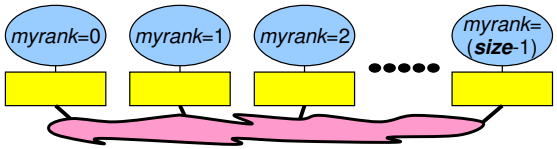
Rank

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

C

Fortran

- 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

Fortran

- C: int MPI_Comm_size(MPI_Comm comm, int **size*)
- Fortran: MPI_COMM_SIZE(comm, *size*, *ierror*)
  INTEGER comm, size, ierror

Exiting MPI

C

Fortran

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

- **Must** be called last by all processes.
- After MPI_Finalize:
  - Further MPI-calls are forbidden
  - Especially re-initialization with MPI_Init is forbidden

Example: Hello World

C

```c
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 %i of %i.\n", my_rank, size);
   MPI_Finalize();
}
```

Fortran

```fortran
PROGRAM hello
IMPLICIT NONE
INCLUDE "mpif.h"
INTEGER ierror, my_rank, size
CALL MPI_INIT(ierror)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,my_rank,ierror)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,   ierror)
IF (my_rank .EQ. 0) THEN
   WRITE(*,*) 'Hello world!'
ENDIF
WRITE(*,*) 'I am ', my_rank, ' of ', size
CALL MPI_FINALIZE(ierror)
END
```

Output:

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

- Why is the sequence non-deterministic?
- What must be done, that the output of all MPI processes on the terminal window is in the sequence of the ranks?

Point-to-Point Communication

Chap.3 Messages and Point-to-Point Communication

MPI_Init()
MPI_Comm_rank()

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

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	

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	

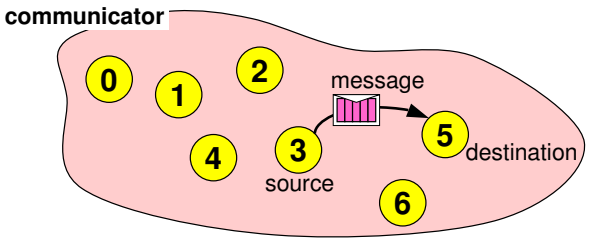
2345	654	96574	-12	7676

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

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

Fortran

- Fortran: MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, *IERROR*)
  <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

- 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*)
  <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.

Communication Envelope

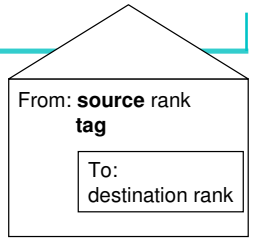
- Envelope information is returned from MPI_RECV in *status*.

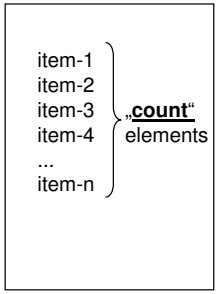
C

- C: status.MPI_SOURCE
  status.MPI_TAG
  count via MPI_Get_count()

Fortran

- Fortran: status(MPI_SOURCE)
  status(MPI_TAG)
  count via MPI_GET_COUNT()





Communication Modes — Definitions

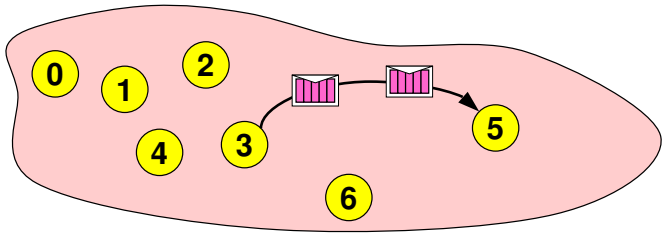
Sender mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Standard send **MPI_SEND**	Either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

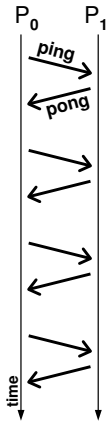
- Rule for messages on the same connection, i.e., same communicator, source, and destination rank:
- **Messages do not overtake each other.**
- This is true even for non-synchronous sends.



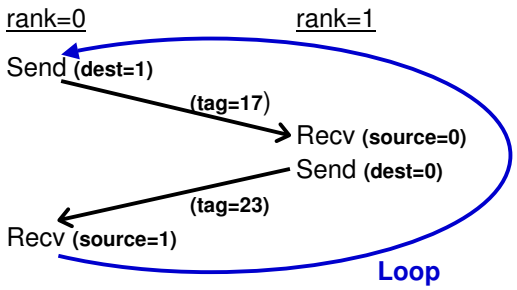
- If both receives match both messages, then the order is preserved.

Example — Benchmark program "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.



Example — 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
```

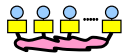
see also login-slides

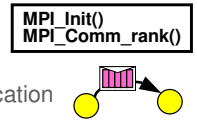
Solution (in C) — Ping pong

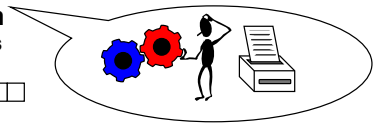
```
#include <stdio.h>
#include <mpi.h>
#define number_of_messages 50
int main(int argc, char *argv[])
{ int i, my_rank;  double start, finish, one_msg;  float buffer[1];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  for (i=0; i <= number_of_messages; i++)  /*includes 1 additional round*/
  { if(i==1) start = MPI_Wtime(); /*first round without time measurement*/
    if(my_rank == 0)
    { MPI_Ssend(buffer,1,MPI_FLOAT,/*rank*/ 1,17,MPI_COMM_WORLD);
      MPI_Recv( buffer,1,MPI_FLOAT,/*rank*/ 0,23,MPI_COMM_WORLD,&status);
    } else if(my_rank == 1)
    { MPI_Recv( buffer,1,MPI_FLOAT, /*rank*/0,17,MPI_COMM_WORLD,&status);
      MPI_Ssend(buffer,1,MPI_FLOAT,/*rank*/ 1,23,MPI_COMM_WORLD);
    }
  }
  finish = MPI_Wtime();  one_msg = (finish-start)/(2*number_of_messages);
  if (my_rank==0)printf("Time for one messsage: %f seconds.\n",one_msg);
  MPI_Finalize();
}
```

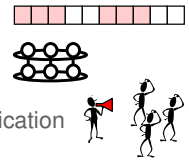
Non-Blocking Communication

Chap.4 Non-Blocking Communication





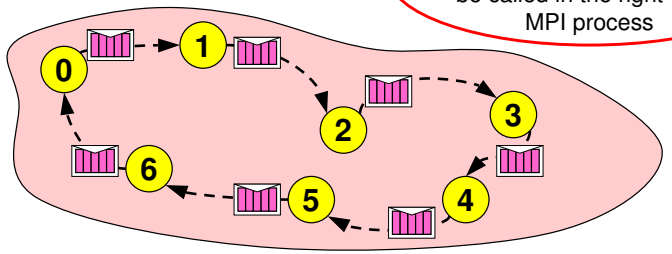




Deadlock

- 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 Synchronous Send

C

- C:

  MPI_Issend(buf, count, datatype, dest, tag, comm,
  OUT &*request_handle*);

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

Fortran

- 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

- Fortran problems, see MPI-2, Chap. 10.2.2, pp 284-290

Non-blocking Receive

C

- C:

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

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

Fortran

- 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

- 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

Request Handles

Request handles

- are used for non-blocking communication

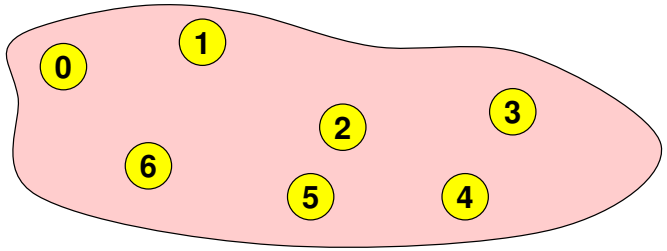
C

Fortran

- **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 corresponding call to MPI_WAIT

Non-Blocking Send (a)

- 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



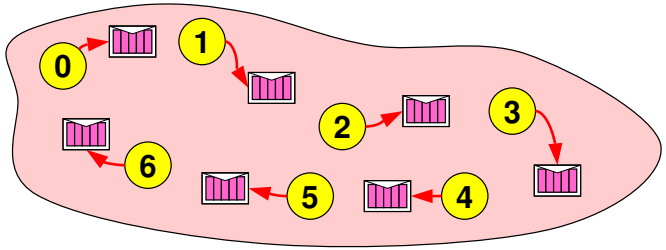
Non-Blocking Send (b)

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 Send (c)

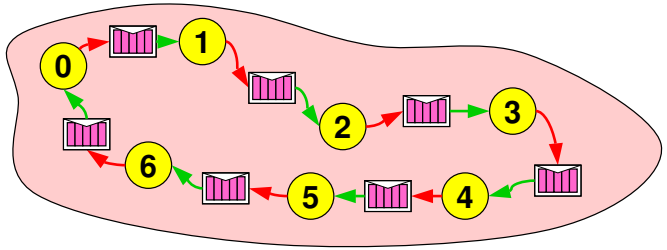
Initiate non-blocking send

→ 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 Send (d)

Initiate non-blocking send

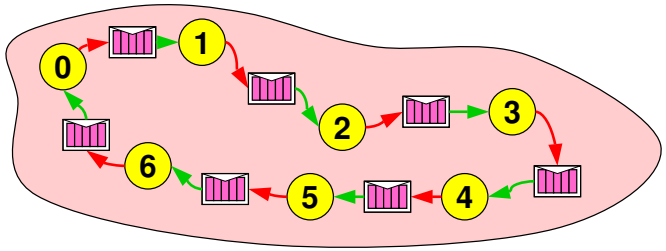
→ in the ring example: Initiate non-blocking send to the right neighbor

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 Send (e)

Initiate non-blocking send

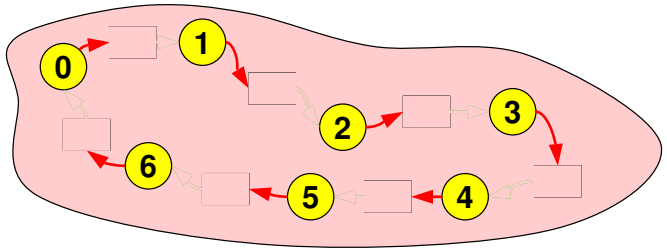
→ in the ring example: Initiate non-blocking send to the right neighbor

Do some work:

→ in the ring example: Receiving the message from left neighbor

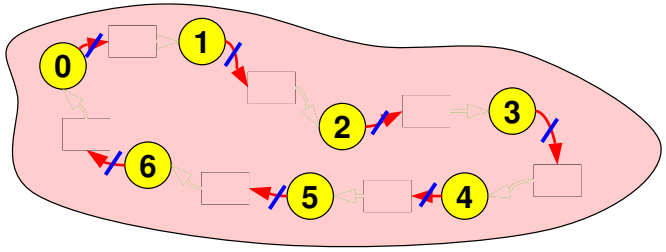
Now, the message transfer can be completed

- Wait for non-blocking send to complete



Non-Blocking Send (f)

- 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

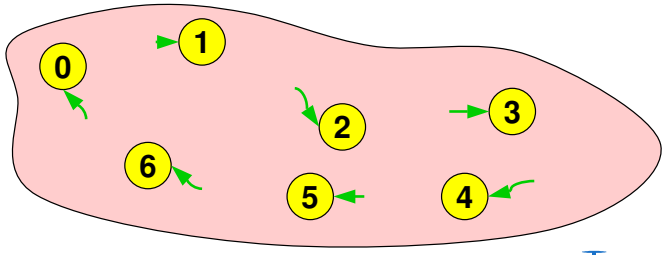


Non-Blocking Receive (a)

- 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

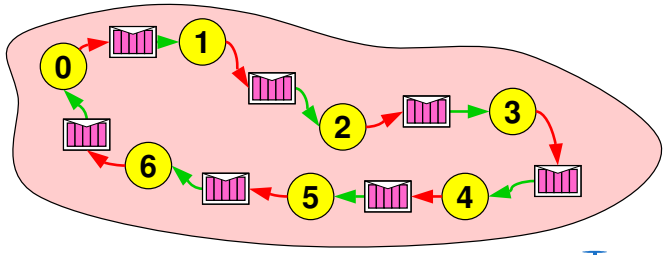
Non-Blocking Receive (b)

- 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



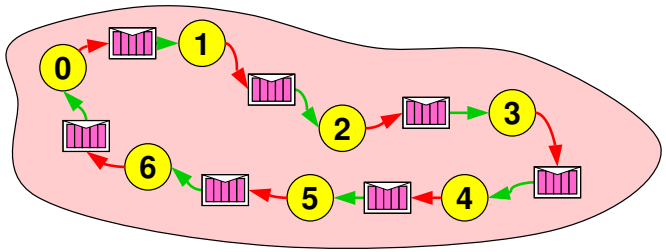
Non-Blocking Receive (c)

- 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



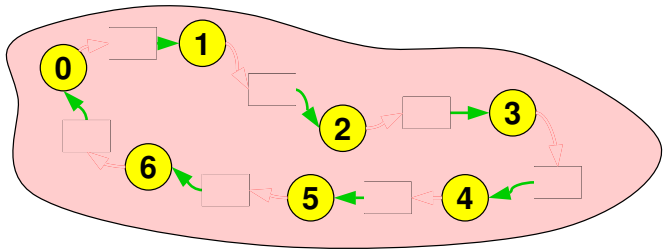
Non-Blocking Receive (d)

- 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



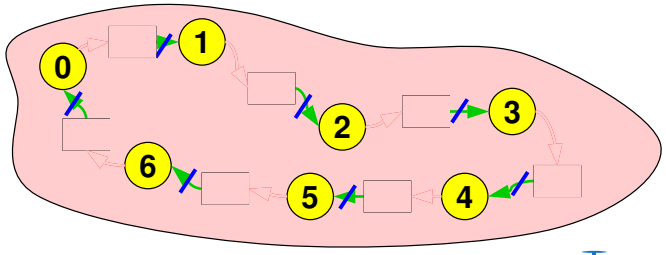
Non-Blocking Receive (e)

- 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



Non-Blocking Receive (f)

- 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 /



Fortran

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

Fortran

Non-blocking MPI routines and strided sub-arrays (a)

- **Fortran:**

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

  other work

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

Fortran

Non-blocking MPI routines and strided sub-arrays (b)

- **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*)

Fortran

Non-blocking MPI routines and strided sub-arrays (c)

- **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**!

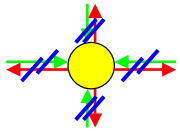
  MPI_WAIT(request_handle, *status*, *ierror*)

- **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!**

Multiple Non-Blocking Communications

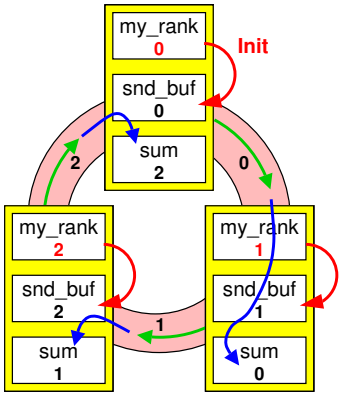
You have several request handles:

- 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**



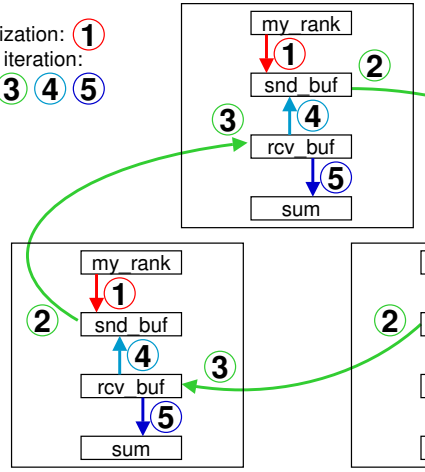
Example — Rotating information around a ring

- 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.
- Do it #CPUs times, 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



Example — 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

Solution (in C) — Ring

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char *argv[])
{ int my_rank, size, send_buf, recv_buf, sum, i, right, left;
  MPI_Request request;  MPI_Status  status;
  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;
1 send_buf = my_rank;  sum = 0;
  for( i = 0; i < size; i++)
  {
2 MPI_Issend(&send_buf,1,MPI_INT,right,117,MPI_COMM_WORLD,&request);
3 MPI_Recv  (&recv_buf,1,MPI_INT,left, 117,MPI_COMM_WORLD,&status);
  MPI_Wait(&request, &status);
4 send_buf = recv_buf;
5 sum += recv_buf;
  }
  printf ("[%3i]  Sum = %i\n", my_rank, sum);
  MPI_Finalize();
}
```


Derived Datatypes

Chap.5 Derived Datatypes

0. **Parallel Programming Models**
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

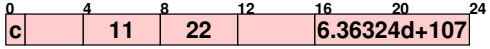
Derived Datatypes — Type Maps

- 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

Example:



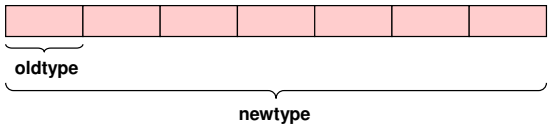
derived datatype handle

basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

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

Contiguous Data

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



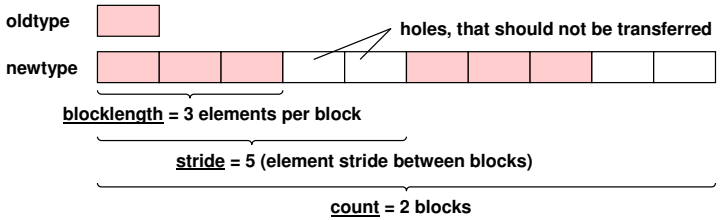
C

- C: int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

Fortran

- Fortran: MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
  INTEGER COUNT, OLDTYPE
  INTEGER NEWTYPE, IERROR

Vector Datatype



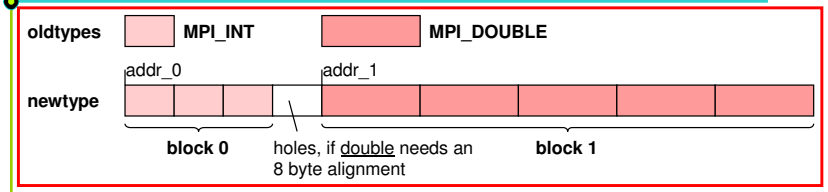
C

- 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)
  INTEGER COUNT, BLOCKLENGTH, STRIDE
  INTEGER OLDTYPE, NEWTYPE, IERROR

Struct Datatype



C

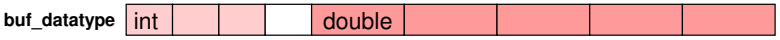
- C: int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

Fortran

- 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



C

- C

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

Fortran

- 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
```

How to compute the displacement

- array_of_displacements[i] := address(block_i) – address(block_0)
- MPI-1

C

Fortran

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

C

Fortran

  - 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
- Examples: MPI-1, pp 77-84 (**Typo: page 80 line 2: MPI_Aint base; instead of: int base;**)

Sub-Arrays Datatype Constructor

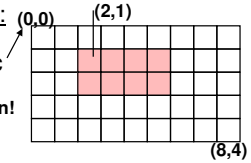
C

- C: int MPI_Type_create_subarray(int ndims, int *array_of_sizes, int *array_of_subsizes, int *array_of_starts, int order, MPI_Datatype oldtype, MPI_Datatype **newtype*)

Fortran

- Fortran: MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER, OLDTYPE, *NEWTYPE*, *IERROR*)
  INTEGER NDIMS, ARRAY_OF_SIZES,
  INTEGER ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS,
  INTEGER ORDER, NEWTYPE, IERROR

Example:

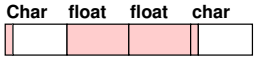


ndims	**= 2**
array_of_sizes	**= (9,5)**
array_of_subsizes	**= (4,2)**
array_of_starts	**= (2,1)**
order	**= MPI_ORDER_FORTRAN or MPI_ORDER_C**

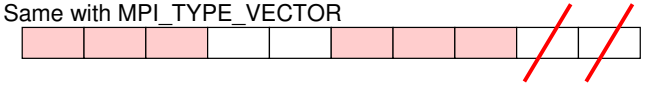
Caution 1

Be careful with derived of derived of … derived datatypes:

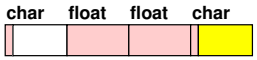
- By default, there isn't a hole at the end!
- For correct word- or double-alignment, such holes may be needed!



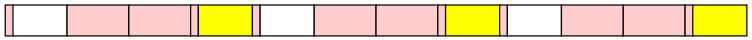
- Same with MPI_TYPE_VECTOR



- Use MPI_TYPE_CREATE_RESIZED to add a hole at the end



- Now, "arrays" of derived datatypes are possible



Caution 2

Performance

- Some MPI library have a poor performance with derived datatypes
- Always prefer
  - structure of arrays, or
  - independent arrays
- instead of
  - array of structures
- Transfer of non-contiguous data
  - Check which algorithm is faster:
    A) Usage of derived datatypes
    B) - Copy at sender into a local contiguous scratch-buffer
    - Transfer this scratch-buffer into a scratch-buffer at the receiver
    - Copy the receiver's scratch-buffer into its non-contiguous memory locations

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

- C: int MPI_Type_commit(MPI_Datatype **datatype*);

Fortran

- Fortran: MPI_TYPE_COMMIT(*DATATYPE*, *IERROR*)
  INTEGER DATATYPE, IERROR

IN-OUT argument

Virtual Topologies

Chap.6 Virtual Topologies

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

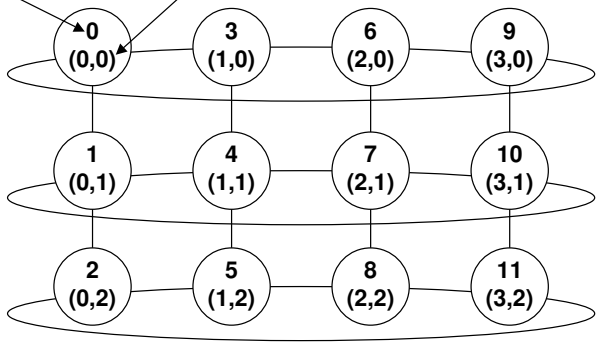
- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimize communications.

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,
  - not covered here.

Creating a Cartesian Virtual Topology

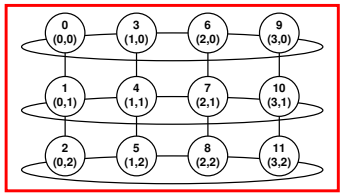
C

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

Fortran

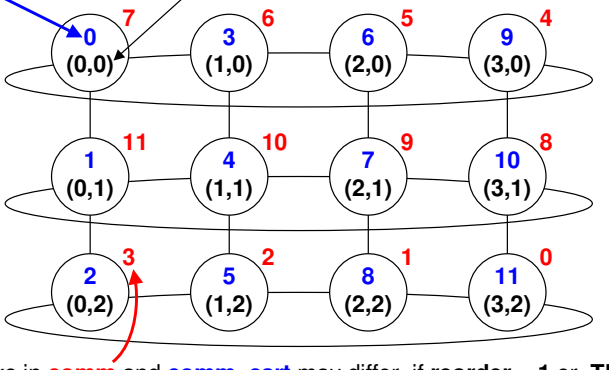
- Fortran: MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS,
  REORDER, *COMM_CART*, *IERROR*)
  INTEGER COMM_OLD, NDIMS, DIMS(*)
  LOGICAL PERIODS(*), REORDER
  INTEGER COMM_CART, IERROR

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



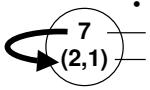
Example – A 2-dimensional Cylinder

- **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

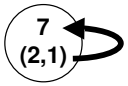
- C: int MPI_Cart_coords(MPI_Comm comm_cart, int rank, int maxdims, int **coords*)

Fortran

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

Cartesian Mapping Functions

- Mapping process grid coordinates to ranks



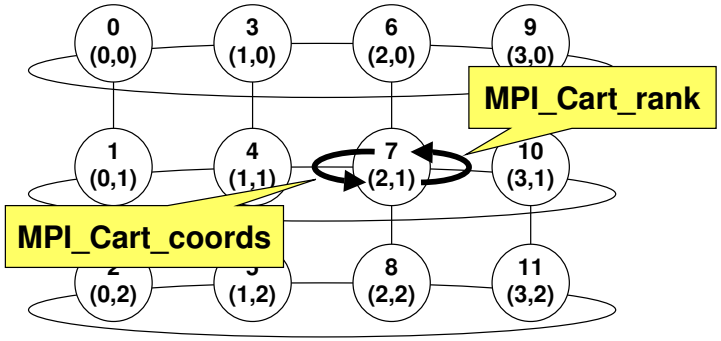
C

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

Fortran

- 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 (example in Fortran)
  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

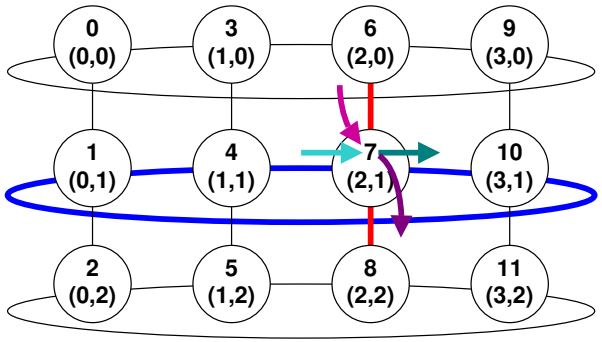
- C: int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int disp, int **rank_source*, int **rank_dest*)

Fortran

- Fortran: MPI_CART_SHIFT(COMM_CART, DIRECTION, DISP, *RANK_SOURCE*, *RANK_DEST*, *IERROR*)
  INTEGER COMM_CART, DIRECTION
  INTEGER DISP, RANK_SOURCE
  INTEGER 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 noop!

MPI_Cart_shift – Example



invisible input argument: **my_rank** in cart

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

example on	0 or	+1	4	10
process rank=7	1	+1	6	8

Cartesian Partitioning

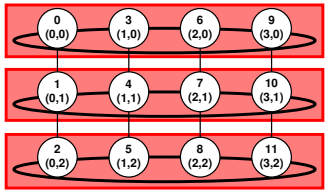
- Cut a grid up into *slices*.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.

C

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

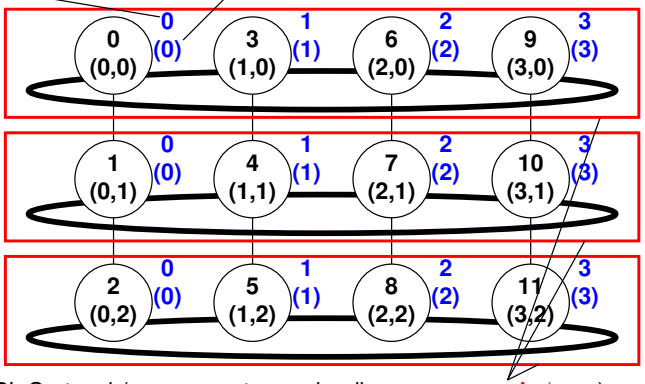
Fortran

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

Collective Communication

Chap.7 Collective Communication

MPI_Init()
MPI_Comm_rank()

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.

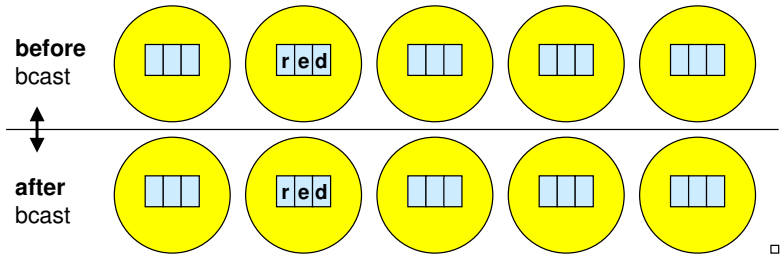
Broadcast – before calling the routine (a)

C

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

Fortran

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



e.g., root=1

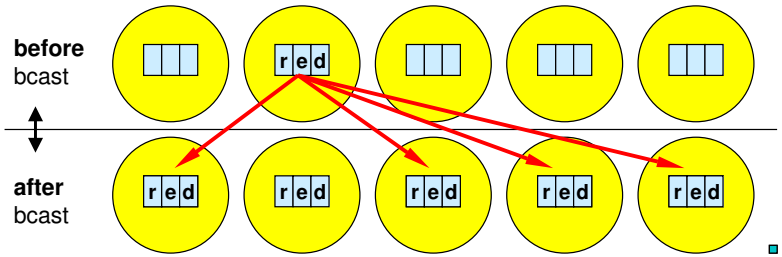
Broadcast – after the routine has finished (b)

C

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

Fortran

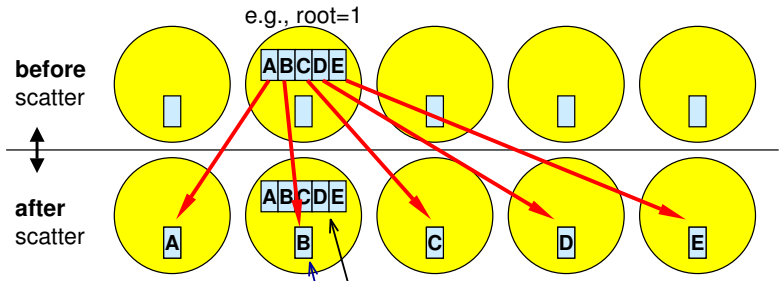
- Fortran: MPI_Bcast(BUF, COUNT, DATATYPE, ROOT, COMM, *IERROR*)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, ROOT
  INTEGER COMM, IERROR



e.g., root=1

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

Scatter



- 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)

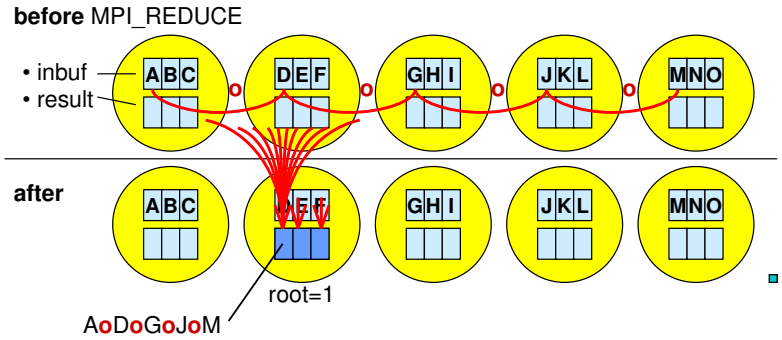
Example of Global Reduction

- 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.

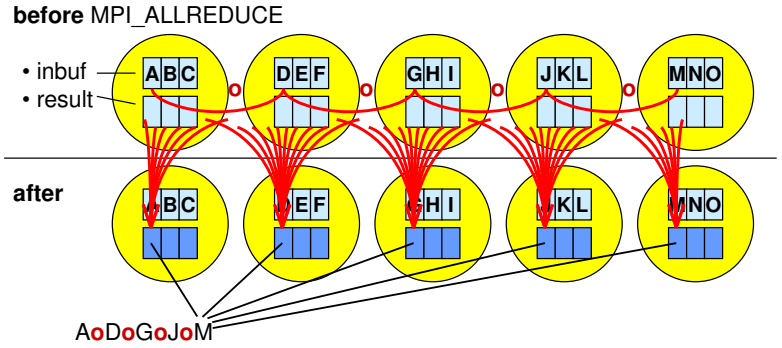
Predefined Reduction Operation Handles

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



MPI_ALLREDUCE



MPI Provider

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 or OpenMPI – another public domain MPI library
- Other info at www.hlrs.de/mpi/

MPI-1 Summary

Summary

Parallel Programming Models

- OpenMP on shared memory systems / incremental parallelization
- MPI for distributed memory systems and any number of processes

MPI

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