BremHLR
Kompetenzzentrum für Höchstleistungsrechnen Bremen

Writing Message-Passing Parallel Programs with MPI

Lars Nerger (BremHLR & Alfred Wegener Institute)

The Message-Passing Interface (MPI) Standard

MPI

- MPI comprises a library.
  - A precompiled archive providing routines with specified interface
  - Library needs to be linked when compiling a program
- MPI process is a program (C / C++ / Fortran) that communicates with other MPI programs by calling MPI routines.
  - Still the MPI-parallelized program is a single executable program that is started using a single command
- Portability – MPI provides the programmer a consistent platform-independent interface.
  - Use the some routine calls on different computers. Just re-compile your program.

Goals and Scope of MPI

- MPI's prime goals are:
  - To provide source-code portability
  - To allow efficient implementation

- It also offers:
  - A wide range of functionality
  - Support for heterogeneous parallel architectures

The "MPI Forum"

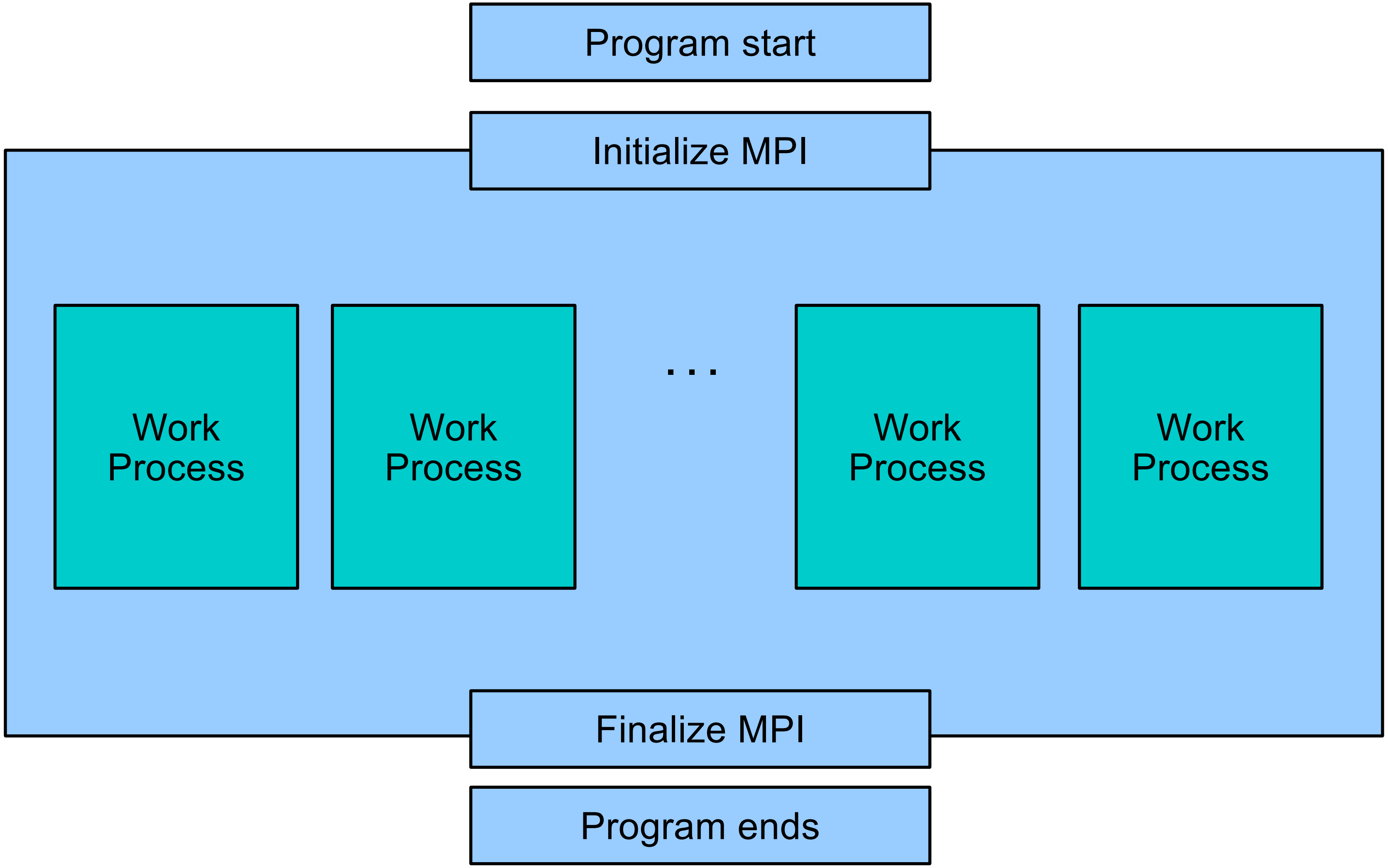
- Creator of MPI standards
- First message-passing interface standard.
  - Sixty people from forty different organizations.
  - Users (government, academia) and vendors represented, from the US and Europe.
  - Two-year process of proposals, meetings and review.
- *Message Passing Interface* documents produced (1994: MPI-1; 1997: MPI-1.2 & MPI-2;
  2008: MPI-2.1; 2009: MPI-2.2;
  2012: MPI-3; 2015: MPI-3.1;
  2021: MPI-4.0 2023: MPI-4.1; 2025: MPI-5.0).
- *The MPI Forum* at `http://www.mpi-forum.org`

MPI @ Home

- Using MPI possible on any PC or notebook computer
- Current PC processors have typically 2 to 6 cores
- Free MPI implementations:
  - MPICH2 (http://www.mcs.anl.gov/research/projects/mpich2)
  - OpenMPI (http://www.open-mpi.org)
- Binary packages available for various Linux distributions
- OpenMPI also for Windows
- OpenMPI also for Mac OS
  (via fink, macports or Homebrew)

Basic Functions of MPI

Structure of MPI program



MPI Function Format

- Name space "`MPI_`" (routines, constants)
- C:

```
error = MPI_Xxxxx(parameter, ...);

  MPI_Xxxxx(parameter, ...);
```

- Fortran:

```
  CALL MPI_XXXXX(parameter, ..., IERROR)
```

- MPI constants all upper case (C and Fortran)
- Return value can be tested with `MPI_SUCCESS`

```
if (IERROR != MPI_SUCCESS)
```

Handles

- MPI controls its own internal data structures
- MPI releases 'handles' to allow programmers to refer to these
- C handles are of defined `typedef`s
- Fortran handles are `INTEGER`s
  (unless using Fortran 2008 binding of MPI 3 – we don’t)
- Examples:
  - `MPI_SUCCESS` – To test MPI error codes
  - `MPI_COMM_WORLD` – A (predefined) communicator

Header files / MPI-module

Define constants

- C:

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

- Fortran:

```
use mpi
```

- Must be included by every MPI-calling routine

- Old style Fortran:

```
include 'mpif.h'
```

  - Disadvantage: no syntax checking
  - Advantage: Lesser issues with compiler compatibility

Initializing MPI

- C:

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

- Fortran:

```
CALL MPI_INIT(IERROR)
  INTEGER IERROR
```

- Must be first MPI routine called; only once.

Exiting MPI

- C:

```
int MPI_Finalize()
```

- Fortran:

```
CALL MPI_FINALIZE(IERROR)
   INTEGER IERROR
```

- Clean up all MPI data structures
- Must be called last by all MPI processes.
- No call to `MPI_Init` allowed after finalizing

Stopping MPI-program in case of error

- Consider case that some process-local computation is wrong
- C:

```
int MPI_Abort(MPI_Comm comm, int error_code)
```

- Fortran:

```
CALL MPI_ABORT(COMM, ERROR_CODE, IERROR)
     INTEGER COMM, ERROR_CODE, IERROR
```

- Use instead of
  - `exit(error_code)`
  - `return(error_code)`
  - `STOP`

WHY?

The minimal MPI Program: “Hello world” (in Fortran)

```
PROGRAM hello

  USE mpi

  IMPLICIT NONE
  INTEGER ierror

  CALL MPI_INIT(ierror)

  WRITE(*,*) 'Hello world!'

  CALL MPI_FINALIZE(ierror)

END
```

The minimal MPI Program:
"Hello world" (in C)

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

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello world!\n");
  MPI_Finalize();
}
```

Timers

- Measure time (wall clock) in seconds
- C:

```
double MPI_Wtime(void);
```

- Fortran:

```
REAL(8) MPI_WTIME()
```

- Time to perform a task is measured by consulting the timer before and after.

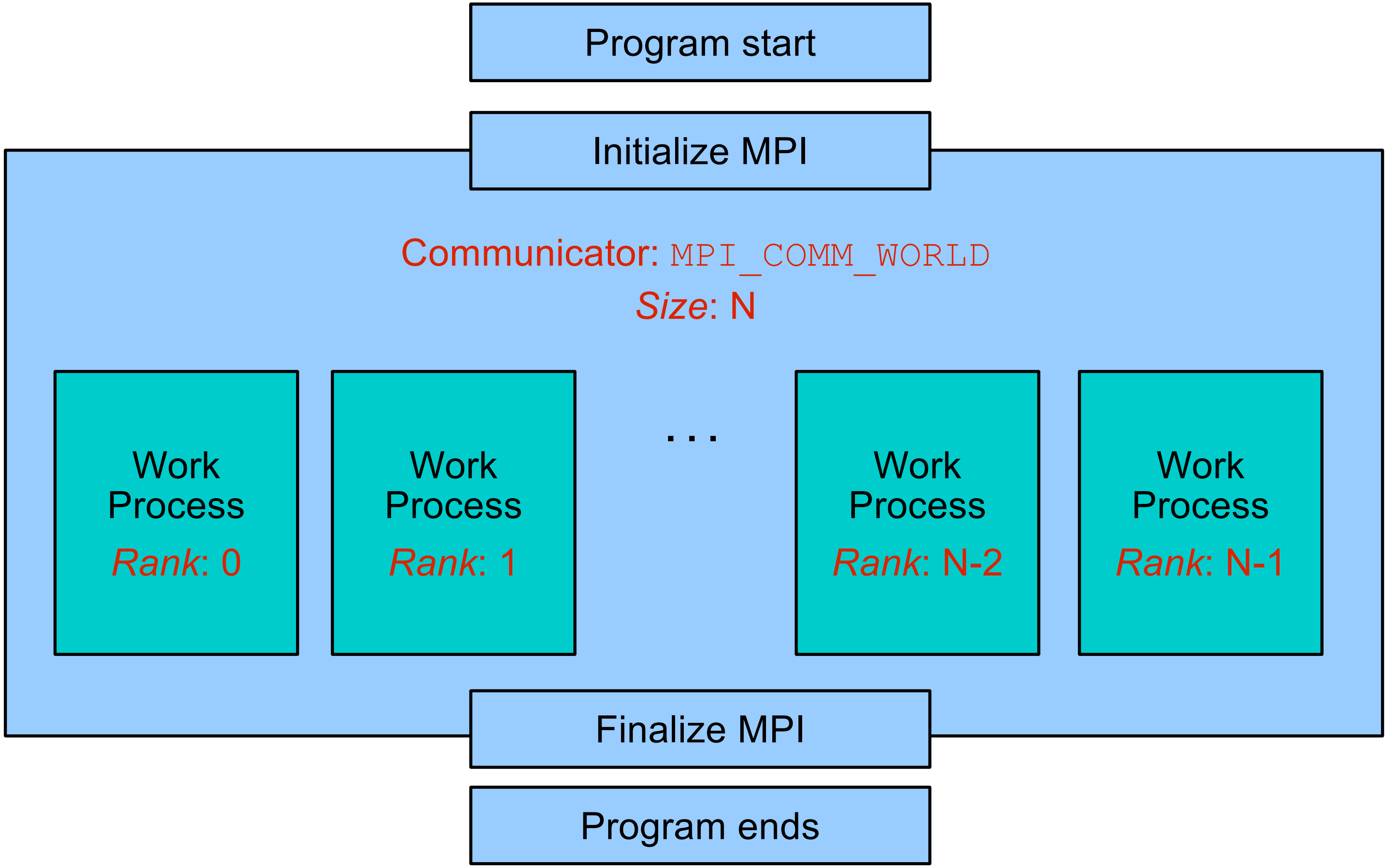
```
time1 = MPI_Wtime()
… work …
time2 = MPI_Wtime()
passed_time = time2-time1
```

- Query resolution of MPI_WTIME in seconds:

```
double MPI_Wtick()

REAL(8) MPI_WTICK()
```

Structure of MPI program



Size

- How many processes are contained within a communicator?
  - C:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

  - Fortran:

```
CALL MPI_COMM_SIZE(COMM, SIZE, IERROR)
   INTEGER COMM, SIZE, IERROR
```

Rank

- How do you identify different processes?
  - Process number within the group
  - rank = 0, 1, ... size-1
  - C:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

  - Fortran:

```
CALL MPI_COMM_RANK(COMM, RANK, IERROR)
     INTEGER COMM, RANK, IERROR
```

Hello-World in C (Variant 2)

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

int main(int argc, char *argv[])  /* hello world */
{
  int size;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("MPI: size = %3d rank = %3d", size, rank);

  MPI_Finalize();
}
```

The more complete Hello-World Program in Fortran

```
PROGRAM main  ! hello world

  USE mpi
  IMPLICIT NONE
  INTEGER :: size, rank, ierr

  CALL MPI_Init(ierr)
  CALL MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

  WRITE(*, *) " MPI: size = ", size, rank = ", rank

  CALL MPI_Finalize(ierr)

END
```

Reading C-prototypes

- Prototype

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- Implementation

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

- Fortran is easier

```
SUBROUTINE MPI_Comm_size(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

- Implementation

```
INTEGER :: size, ierror

CALL MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
```

Compiling and Linking MPI programs

Using OpenMPI

- C

```
mpicc -o simple simple.c
```

- Fortran

```
mpif90 -o simple simple.f90
```

mpicc/mpif90 are wrappers;
without one needs to explicitly link the MPI library:

- C

```
gcc -o simple simple.c -lmpi [-L… -I…]
```

- Fortran

```
gfortran -o simple simple.f90 -lmpi [-L… -I…]
```

Running MPI programs

Running MPI programs on the course computers

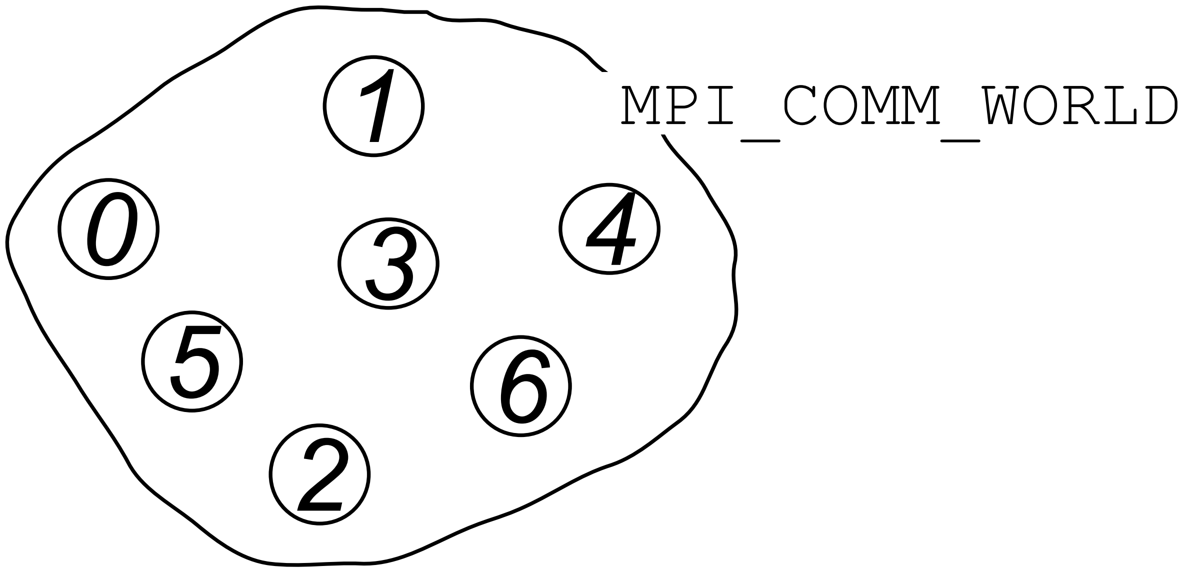
- In the shell:

```
mpirun -np TASKS EXE
```

  - `TASKS`   a number specifying the number of processes
  - `EXE`   name of the executable (program)

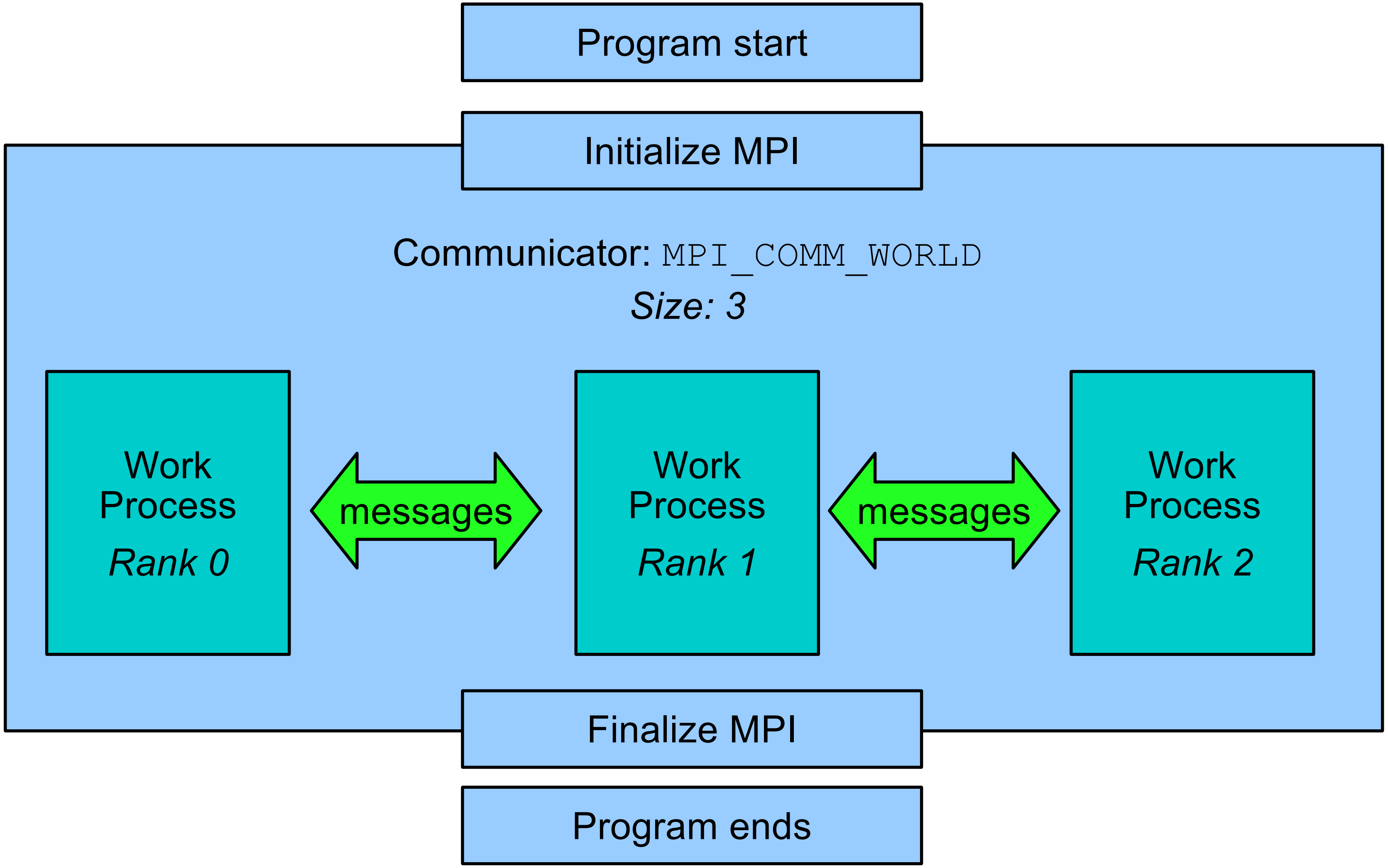
Communication

Communicators



- A **communicator** describes a group of processes with certain properties: The group is ordered and has a *context* (a virtual network), in which they can communicate with each other
- `MPI_COMM_WORLD` is predefined (default) standard communicator
- Many additional communicators can be defined as subsets of this group.
- `MPI_COMM_WORLD` can not be extended (MPI-1).

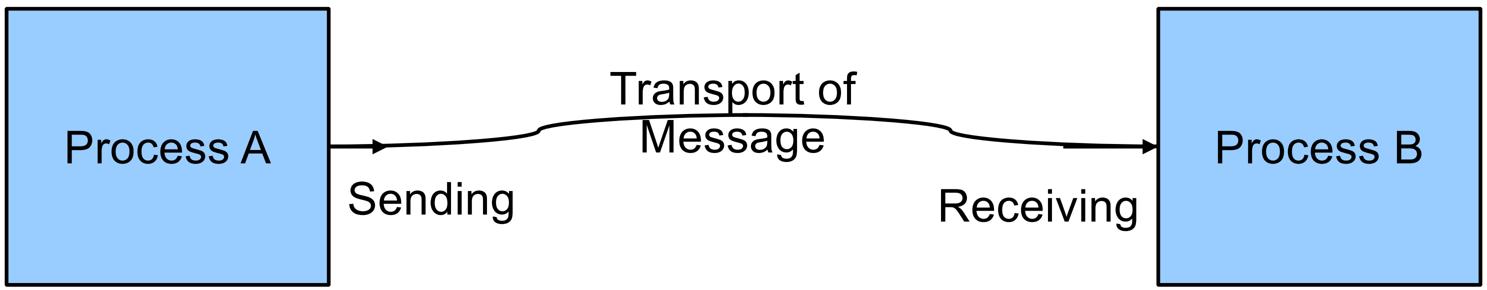
MPI Program with Communication



Messages

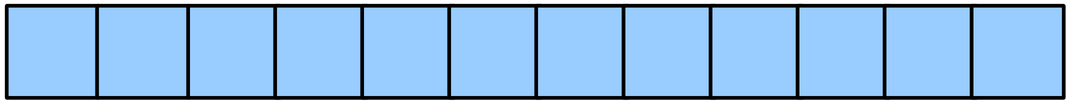
Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another
- Other process receives message



Messages

- A message contains a number of elements of some particular datatype.



- MPI datatypes:
  - Basic types
  - Derived types
- Derived types can be built up recursively from basic types and previously defined derived types.
- C types are different from Fortran types.
- In general *send type* = *receive type* required

Messages

- Messages are packets of data moving between processes of a message-passing group.
- The message-passing system has to be told the following information:
  - Sending processor
  - Source location
  - Data type
  - Data length
  - Receiving processor(s)
  - Destination location
  - Destination size
- Message transfer also provides synchronisation information

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.
- Multiple connections can be possible
- Access is provided by the communicator

Addressing

- Messages need to have addresses to be sent to.
- Addresses are similar to:
  - Mail address
  - Phone number
  - Fax number
  - etc.
- Relevant is only the address information, not the data content

Reception

- The receiving process has to be capable to deal with messages it is sent.
- Identify
  - Message type
  - Length
  - Origin

Sending a message

- C:

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

- Fortran:

```
CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST,
              TAG, COMM, IERROR)

    <type> BUF(*)

    INTEGER COUNT, DATATYPE, DEST, TAG
    INTEGER COMM, IERROR
```

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:

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

    <type> BUF(*)

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

MPI Basic Datatypes - C

MPI data type	C data type
MPI_CHAR	signed char
MPI_SHORT	signed short
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 data types are not C types!

MPI Basic Datatypes - Fortran

MPI data type	Fortran data type
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	-

With Fortran-90 also, e.g.:

MPI data type	Fortran data type
MPI_REAL8	REAL(kind=8) DOUBLE PRECISION

MPI data types are not Fortran types!

Why MPI Datatypes?

- Explicit data description is useful:
  - Simplifies programming,
    for example send row/column of a matrix with single call.
  - Heterogeneous machines (automatic data conversion)
    - inside the same MPI implementation, only.
    - no guarantee between different implementations
  - May improve performance
    - Reduce memory-to-memory copies
    - Allow use of scatter/gather hardware

Data Types in MPI and C (I)

```
#include <stdio.h>

typedef long long MPI_Datatype;
                    // define a new C type called 'MPI_Datatype'

const MPI_Datatype MPI_INT = 1;
                    // define a C constant called 'MPI_INT'
const MPI_Datatype MPI_DOUBLE = 2;
                    // define a C constant called 'MPI_DOUBLE'

void sum(void *p1, void *p2, void *p3, MPI_Datatype type);
                    // declare a C function using the new type
```

Data Types in MPI and C (II)

```
int main(int argc, char *argv[])
{
    int    i, j = 47, k = 11;
    double x, y = 48, z = 12;

    MPI_Datatype my_int_type = MPI_INT;
                              // define a C variable of type MPI_Datatype

    sum(&i, &j, &k, my_int_type);
    sum(&x, &y, &z, MPI_DOUBLE);

    printf("%d %f\n", i, x);
}
```

Data Types in MPI and C (III)

```
void sum(void *p1, void *p2, void *p3, MPI_Datatype type)
{
   int    *i1, *i2, *i3;
   double *x1, *x2, *x3;

  // evaluation of 'type' leads to a runtime polymorphism:

   if (type == MPI_INT) {
      i1 = (int *) p1;
      i2 = (int *) p2;
      i3 = (int *) p3;
      *i1 = *i2 + *i3;
   } else if (type == MPI_DOUBLE) {
      x1 = (double *) p1;
      x2 = (double *) p2;
      x3 = (double *) p3;
      *x1 = *x2 + *x3;
   }
}
```

Message Tags

- *Tags* are used to distinguish different messages
- The Tag in *Send* and *Recv* must be identical
- Tags are user defined
(be careful regarding uniqueness)
- MPI preserved the order of messages:
two *send*s to a target process using the same tag in the same communicator arrive in the order they have been sent
    - Thus: If you call the *Recv*s in the correct order, the tag is irrelevant

Exercise 1

Documentation of MPI functions

- Interface is not shown for all MPI functions
- Consult man-pages
  - e.g. man mpi_scatter

Terminal — ssh — 91x24

```
MPI_Scatter(3)                        MPI                       MPI_Scatter(3)



NAME
       MPI_Scatter  -  Sends data from one process to all other processes in a
       communicator

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

INPUT PARAMETERS
       sendbuf
              - address of send buffer (choice, significant only at root )
       sendcount
              - number of elements sent to each process (integer,  significant
              only at root )
       sendtype
              -  data type of send buffer elements (significant only at root )
              (handle)
       recvcount
 Manual page MPI_Scatter(3) line 1
```

Documentation of MPI functions

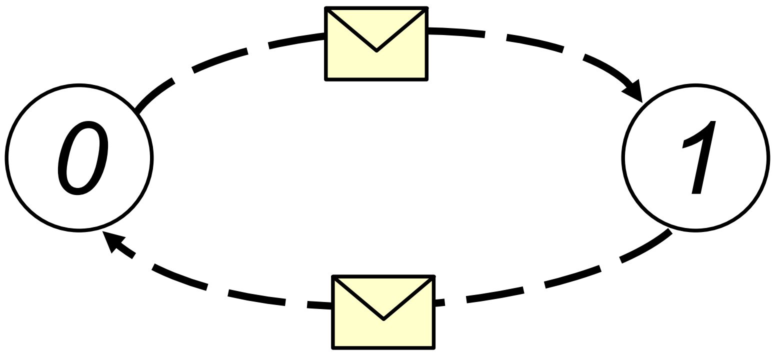
- Online

  `http://www.open-mpi.org`

  then ➜ Documentation

Exercise 1: Ping pong

- Write a program in which two processes repeatedly pass a message back and forth.
- Insert timing calls to measure the time taken for one message (send+receive).
- Investigate how the time taken varies with the size of the message



Exercise 1: Ping pong (II)

- Use message sizes 1, 2, 4, 8, 16, 32, ..., ~1 Mio
  or 1, 10, 100, 1000, ..., 1 Mio
- Loop over message sizes
- In the loop time this block of code with `MPI_Wtime` :

```
MPI_Ssend(...);
MPI_Recv(...);
```

- Print out a table:
  message_size   transfer_time
- Advanced: produce a graphical plot with double logarithmic axes for transfer_time(message_size)

Plotting with Python

- Double logarithmic plot:
- File test.dat holds 2 columns: (size, time)

```
~> python
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> field = np.loadtxt('test.dat’)
>>> plt.loglog(field[:,0],field[:,1])
>>> plt.show()
```

Plotting with Gnuplot (I)

- Double logarithmic plot:

```
hlogin1:~> cat test.dat
1 1
10 10
20 20
50 100
hlogin1:~> gnuplot
gnuplot> p
gnuplot> plot "test.dat" with lines
gnuplot> quit
hlogin1:~>
```

Plotting with Gnuplot (II)

- Generate output file as Encapsulated PostScript:

```
hlogin1:~> gnuplot
gnuplot> set terminal postscript eps
gnuplot> set output "test.eps"
gnuplot> set logscale xy
gnuplot> plot "test.dat" with points
gnuplot> quit
hlogin1:~>
```

- Display plot

```
hlogin1:~> display test.eps
```