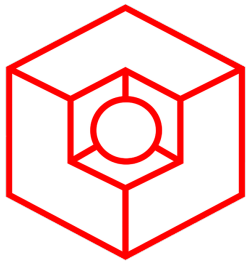


BremHLR
Kompetenzzentrum für Höchstleistungsrechnen Bremen

# Writing Message-Passing Parallel Programs with MPI

## Hands-on Exercises

Lars Nerger (BremHLR & Alfred Wegener Institute)

Universität Bremen

CONSTRUCTOR UNIVERSITY

HSB

Hochschule Bremerhaven

NATIONALES HOCHLEISTUNGS RECHNEN

# Computer setup

## for compiling and running parallel programs

http://bremhlr.uni-bremen.de/mpi-openmp-course

# Computer Setup

- We will do several hands-on exercises
  - Practice implementation of different MPI functions

- Required
  - C or Fortran compiler (typically GCC – GNU compiler collection)
  - MPI library and runtime (typically OpenMPI)
  - Plot tool (typically gnuplot)
  - Software availability
    - Linux: available as packages of Linux distribution
    - Windows: available as packages in Cygwin
    - MacOS: available with Homebrew, Fink, orMacports

# Initial Exercise

# MPI Preliminaries

- 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
- The is a special starter command for MPI-parallel programs (usually `mpirun`)

# Compiling and Linking MPI programs

Using OpenMPI on course computers

- 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
```

- Fortran

```
gfortran -o simple simple.f90 -lmpi
```

# 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)

# 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();
}
```

# Size

- How many processes are running in my program?
  - C:

```
int MPI_Comm_size(MPI_COMM_WORLD, int *size)
```

  - Fortran:

```
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, SIZE, IERROR)
        INTEGER :: 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_WORLD, int *rank)
```

  - Fortran:

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERROR)
        INTEGER :: RANK, IERROR
```

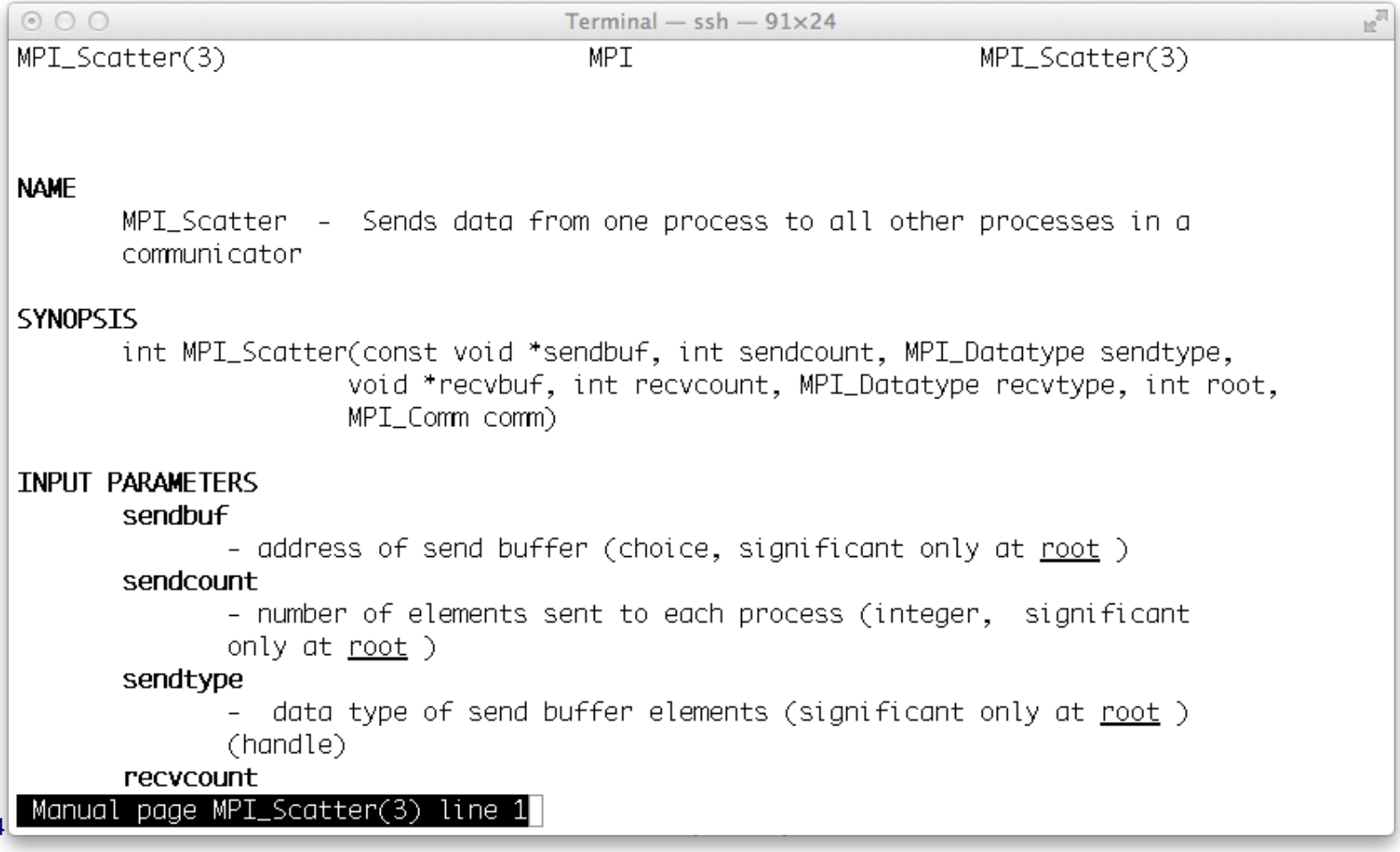
# Exercise

- Type (do not use cut and paste!) the Hello-world program using an editor on the course computers (use either C or Fortran)
- Compile the program
- Run the program
- Modify your program, so that only the process with rank 0 prints out
- Extend the Hello-world program, so that it also writes the total number of processes and the rank of the process in each output line
- Compile and run the program with different numbers of processes

# Exercise 1

# Documentation of MPI functions

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



# Documentation of MPI functions

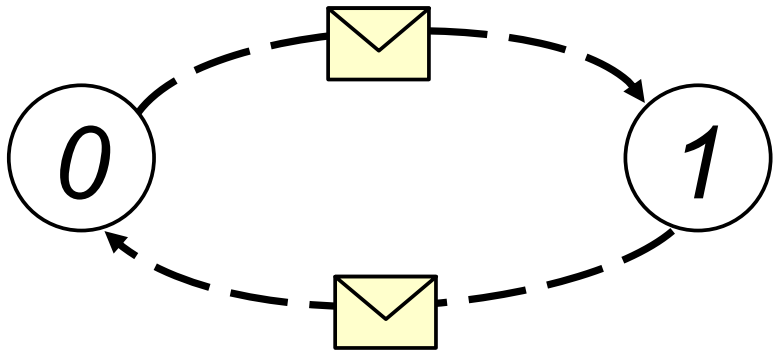
- Online

  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)

# Extra exercise (P2P Comm)

- Write a program in which the process with rank 0 sends the same message to all other processes in MPI_COMM_WORLD and then receives a message of the same length from all other processes.
  How does the time taken vary with the size of the messages and with the number of processes?

# 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
```

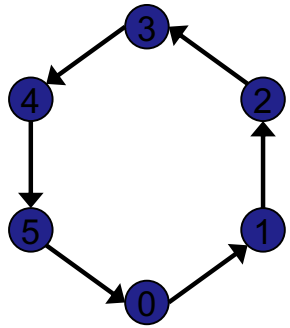
# Exercise 2

# Exercise 2: Rotating information around a ring

- A set of processes are arranged in a ring.
- Each process stores its rank in `MPI_COMM_WORLD` in an integer.
- Each process passes this on to its neighbour on the right.
- Keep passing what is received until the own rank is back where it started.
- Each processor calculates the sum of the values.

# Note on Exercise 2: Rotating information around a ring

- Different from Ping-Pong
  - There is no start point – all processes do the same work
  - SIMD (single instruction - multiple data)



# Exercise 2, Part 2: Ring with collective communication

- Rewrite the pass-around-the-ring program to use MPI global reduction to perform its global sums.
- Then rewrite it so that each process computes a partial sum.
- Then rewrite this so that the program prints out the partial results in the correct order (process 0, then process 1, etc.).

# Exercise 3

# Exercise 3, Part 1: Derived Datatypes

**Part 1:**

- Modify the passing-around-a-ring exercise.
- Calculate two separate sums:
  - rank integer sum, as before
  - rank floating point sum
- Use a *struct* datatype for this.

**Part 2:**

- Instead of passing around the ring use *global reduction* and a *user-defined function* for the sum of the struct.

# Exercise 3: Derived Datatypes (2)

- Defining the *struct* data type
- Fortran

```
module module_my_struct
    type my_struct
        real    :: x
        integer :: i
    end type my_struct
end
```

Later declare a variable

```
use module_my_struct
type(my_struct) :: s
```

Access values:

```
s%x
s%i
```

- C

```
struct myStruct { float x; int i; };
```

Later declare a variable

```
struct myStruct s;
```

Access values:

```
s.x
s.i
```

(or with -> for pointers)

# Exercise 3, Part 2: Ring Topology

- Rewrite the exercise passing numbers round the ring using a one-dimensional ring topology.
- Rewrite the exercise in two dimensions, as a torus. Each row of the torus should compute its own separate result.
- **Extra exercise**: Let MPI calculate the number of processes per coordinate direction (`MPI_Dims_create`).

Write the program such that it can handle an arbitrary number of tasks (e.g. a prime number) for the generation of the two-dimensional topology where some processes remain unused (`MPI_COMM_NULL`).