First steps in MPI

Joachim Hein (LUNARC, Lund University)
Xin Li (PDC, KTH)
Viktor Rehnberg (NAISS & Chalmers)

Overview

- This lecture: Basic set up inside the code for MPI
- Header files
- Initialisation of the MPI library
- Finalisation of the MPI library

Header files

- Every compilation module accessing MPI requires inclusion of a header file:

  - F77 style:
    ```
    include "mpif.h"
    ```
  - Fortran90:
    ```
    use mpi
    ```
  - Fortran08, **MPI 3.0**, Fortran standard compliant!
    ```
    use mpi_f08
    ```

  - C:
    ```
    #include "mpi.h"
    ```

Importing mpi4py in Python

- Python code does not require compilation or header file, but the MPI module needs to be imported

  - Python:

    ```
    from mpi4py import MPI
    ```

MPI command in C

- In C all MPI commands are functions with return type **`int`**

```
int MPI_Abcdef( arguments )
```

- The returned value is the error code
    - Detailing problems with the command
- Typically very hard to recover from MPI-errors
- Most codes do not check these error codes

- **Rem:** MPI commands can modify arguments
    - pass a pointer

MPI command in Fortran

- In Fortran all MPI commands are subroutines

```
MPI_ABCDEF( arguments, ierror )
```

- MPI commands in Fortran carry **one more argument** than their C counter part
  - This is optional in Fortran 2008
  - This is of type `int` and returns the error code
- Again, this is typically unchecked, hence easily forgotten while coding
- Forgetting this in F77/F90 typically leads to segmentation faults at runtime

MPI command in Python

- In Python all MPI commands are methods of an MPI communicator

```
comm.method(arguments)
```

- In general fewer arguments are needed, compared to C and Fortran
- Communication of generic Python objects is done via all-lowercase methods (e.g. comm.send(…))
- Communication of buffer-like objects is done via methods with an uppercase letter (e.g. comm.Send(…))

C++ bindings: depreciated/removed

- MPI used to have special C++ bindings
- Depreciated since MPI standard 2.2 September 2009
- Removed in MPI standard 3.0 September 2012
- Use C bindings in C++ programs
  - Consider wrapping in OO-style for your app's needs

MPI_Init

- The first MPI call of any MPI program has to be **`MPI_Init`**
- In C:

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

  - Arguments are same as **`main`**
  - Alternatively modern MPI libraries allow to pass **`Null`**

- In Fortran

```
MPI_INIT(IERROR)
  INTEGER:: IERROR
```

MPI_Finalize

- The last MPI call has to be **MPI_Finalize**
- In C:

```
int MPI_Finalize(void)
```

- In Fortran:

```
MPI_FINALIZE(IERROR)
  INTEGER:: IERROR
```

No init and finalize required in Python

- In Python, MPI is initialized upon import, and finalized upon exit

```
from mpi4py import MPI
```

Minimal program in C

```
#include "mpi.h"

int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv); // alt.: NULL,NULL

   // further MPI calls go here!

   MPI_Finalize();
}
```

Minimal program in Fortran

```
program minimpi
  use mpi     ! alt.: include ”mpif.h”
  integer:: ierror

  call MPI_INIT(ierror)
              ! further MPI calls go here
  call MPI_FINALIZE(ierror)

end program minimpi
```

Minimal program in Python

```
from mpi4py import MPI

# further MPI calls go here
```

Summary

- Basic requirements for an MPI program
  - Header files
  - Initialising MPI
  - Finalising MPI

Communicators

Overview

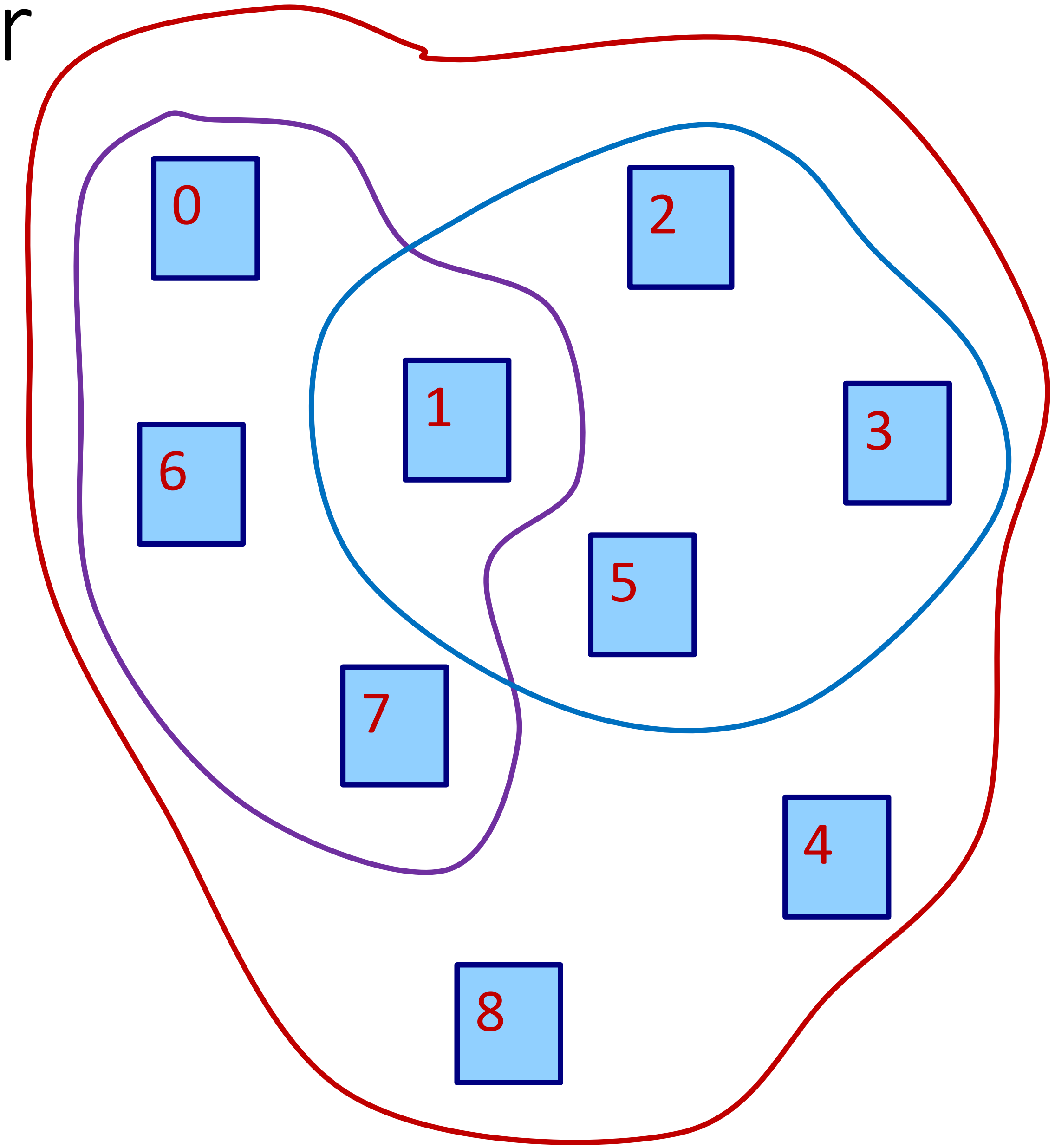
- Concept of communicators
- Predefined communicator
- Querying basic properties of the communicator

Communicator

- Most messages passed inside (intra-)communicator
- Communicator
  - Group of processeses
  - Stores communication universe
  - Order
  - Can have additional topology

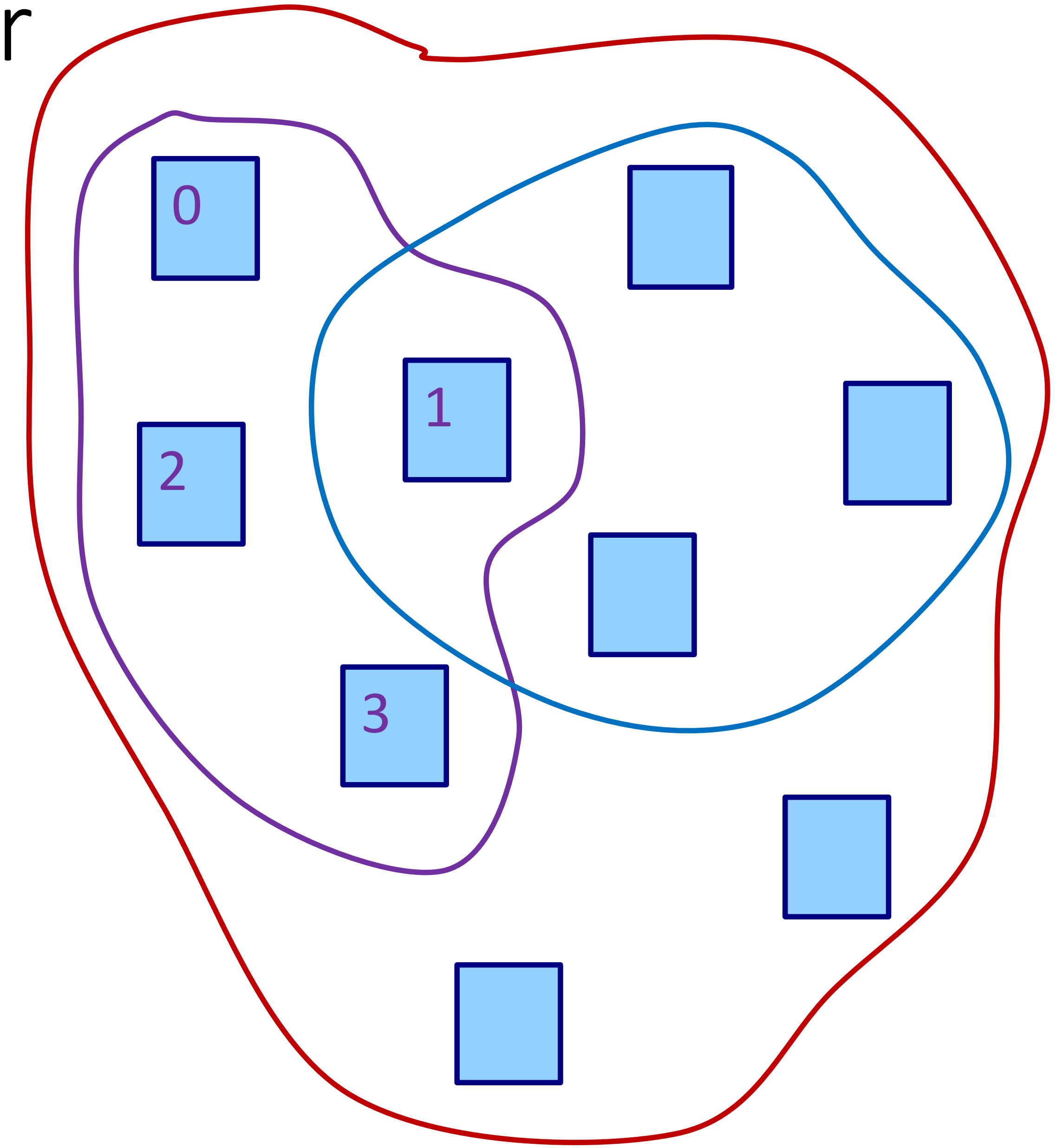
Example communicator

- Picture shows:
  - 9 processes
  - 3 communicators
- Processes carry label
  - Here: labels for red communicator
- Labels start at 0



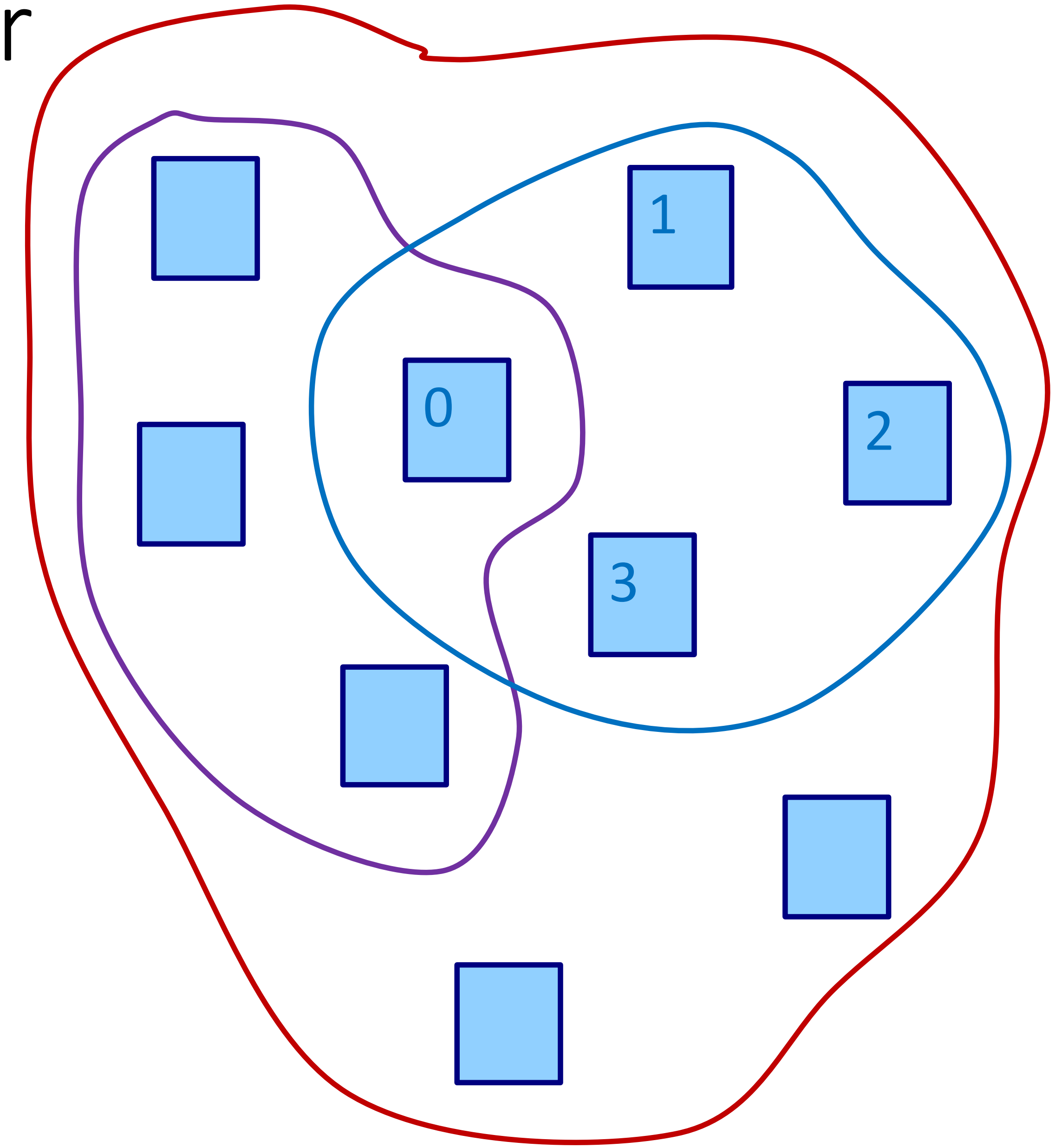
Example communicator

- Picture shows:
  - 9 processes
  - 3 communicators
- Processes carry label
  - Here: labels for violet communicator
- Labels start at 0



Example communicator

- Picture shows:
  - 9 processes
  - 3 communicators
- Processes carry label
  - Here: labels for blue communicator
- Labels start at 0
- **Label depends on the communicator**



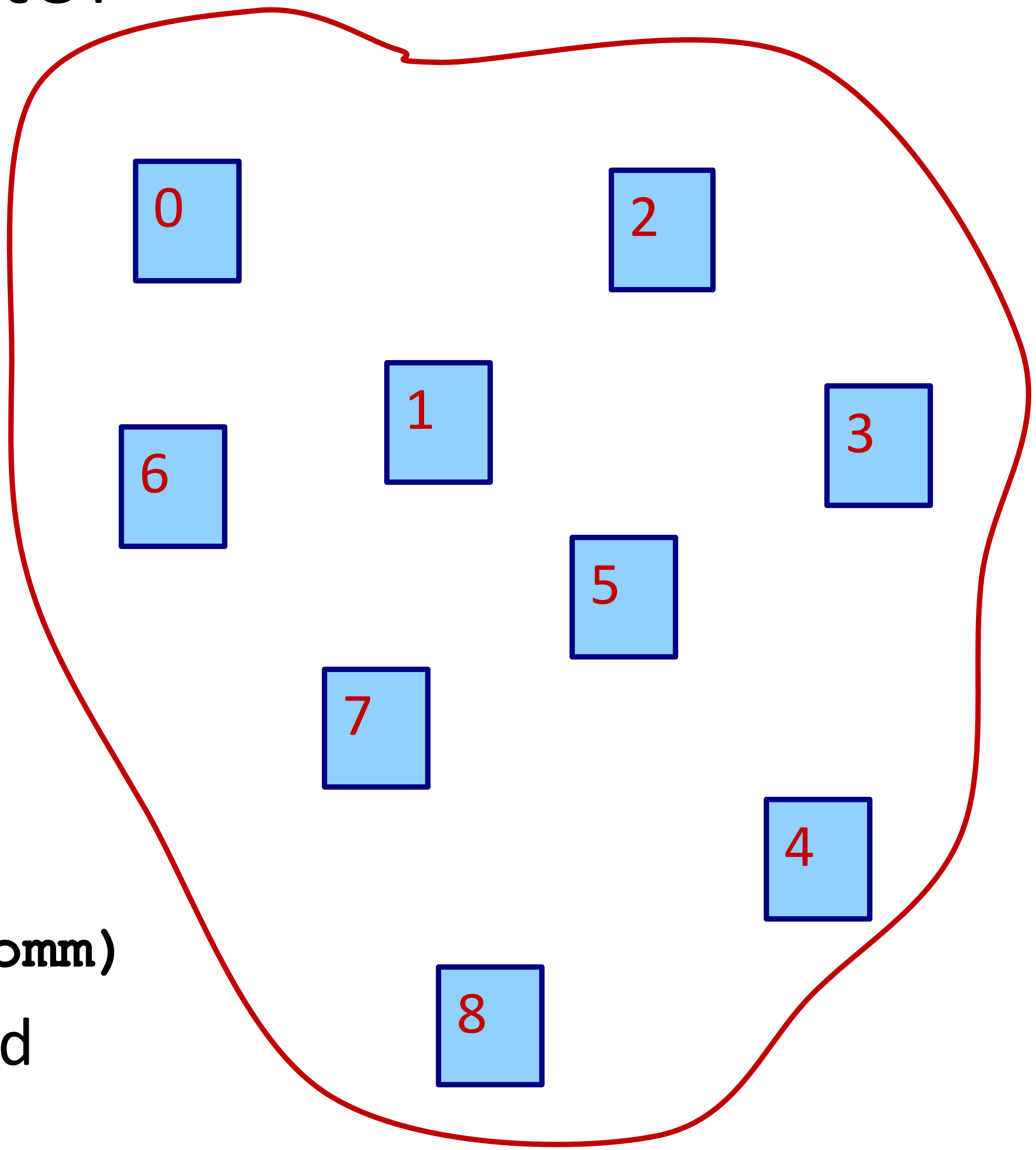
Predefined communicator `MPI_COMM_WORLD`

- After **`MPI_Init`** one predefined communicator:

  `MPI_COMM_WORLD`

  (Python: **`MPI.COMM_WORLD`**)

- This contains all processes
  - In C, this is typedef: **`MPI_Comm`**
  - In Fortran90 this is: **`INTEGER`**
  - In Fortran 2008 this is: **`type(MPI_Comm)`**
- Further communicators: user created



MPI_Comm_size

- Number of processes in a communicator
- In C:

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

- In Fortran 90:

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

- Arguments:

  comm: communicator (input)

  size: number of processes (output)

Get_size in Python

- Number of processes in a communicator

```
comm.Get_size()
```

- No arguments
- Returns the number of processes in a communicator

- (alt. use `comm.size`)

MPI_Comm_rank

- Rank (label) of the process
- In C:

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

- In Fortran 90:

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

- Arguments:

  comm: communicator (input)

  rank: rank of processes (output)

Get_rank in Python

- Rank (label) of the process

```
comm.Get_rank()
```

- No arguments
- Return the rank of this process in a communicator

- (alt. use `comm.rank`)

Copying communicators

- Extensive use of **`MPI_COMM_WORLD`** is *discouraged*
- Exactly **one** reference to **`MPI_COMM_WORLD`** in the program (apart from **`MPI_Abort`**):
- Copy it, e.g.:

```
my_world = MPI_COMM_WORLD
```

- Use **`my_world`** in the rest of the program
- Declare **`my_world`** as
  - **`MPI_Comm`** in C
  - **`INTEGER`** in Fortran 90
  - **`type (MPI_Comm)`** in Fortran 08

Copying communicators in Python

- Extensive use of **`MPI.COMM_WORLD`** is *discouraged*
- Exactly **one** reference to **`MPI.COMM_WORLD`** in the program (apart from **`Abort`**):

```
my_world = MPI.COMM_WORLD
```

- Use **`my_world`** in the rest of the program, example:

```
my_rank = my_world.Get_rank()
```

MPI_Abort

- Aborting all MPI tasks from any task (e.g. read corrupt input file, failed safety check)

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

In Fortran 90:

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
  INTEGER:: COMM, ERRORCODE, IERROR
```

- **`COMM`** is the communicator with the task to abort
  - typically `MPI_COMM_WORLD`
- **`ERRORCODE`** returned to the UNIX shell to flag a problem
  - Return a **`1`** if you do not understand this
- All arguments: input

Abort in Python

- Aborting all MPI tasks from any task (e.g. read corrupt input file, failed safety check)

```
comm.Abort(errorcode)
```

- Typically called by **`MPI.COMM_WORLD`**
- **`errorcode`** returned to the UNIX shell to flag a problem
  - Return a **`1`** if you do not understand this

Summary

- Concept of communicator
- Predefined communicator MPI_COMM_WORLD (MPI.COMM_WORLD in Python)
- Querying task rank and size of a communicator
- Aborting a program on error
- You should now be able to write simple MPI programs, which are useful (e.g. task farm)