MPI Program Structure

- Handles
- MPI communicator
- MPI_COMM_WORLD
- Header files
- MPI function format
- Initializing MPI
- Communicator size
- Process rank
- Exiting MPI

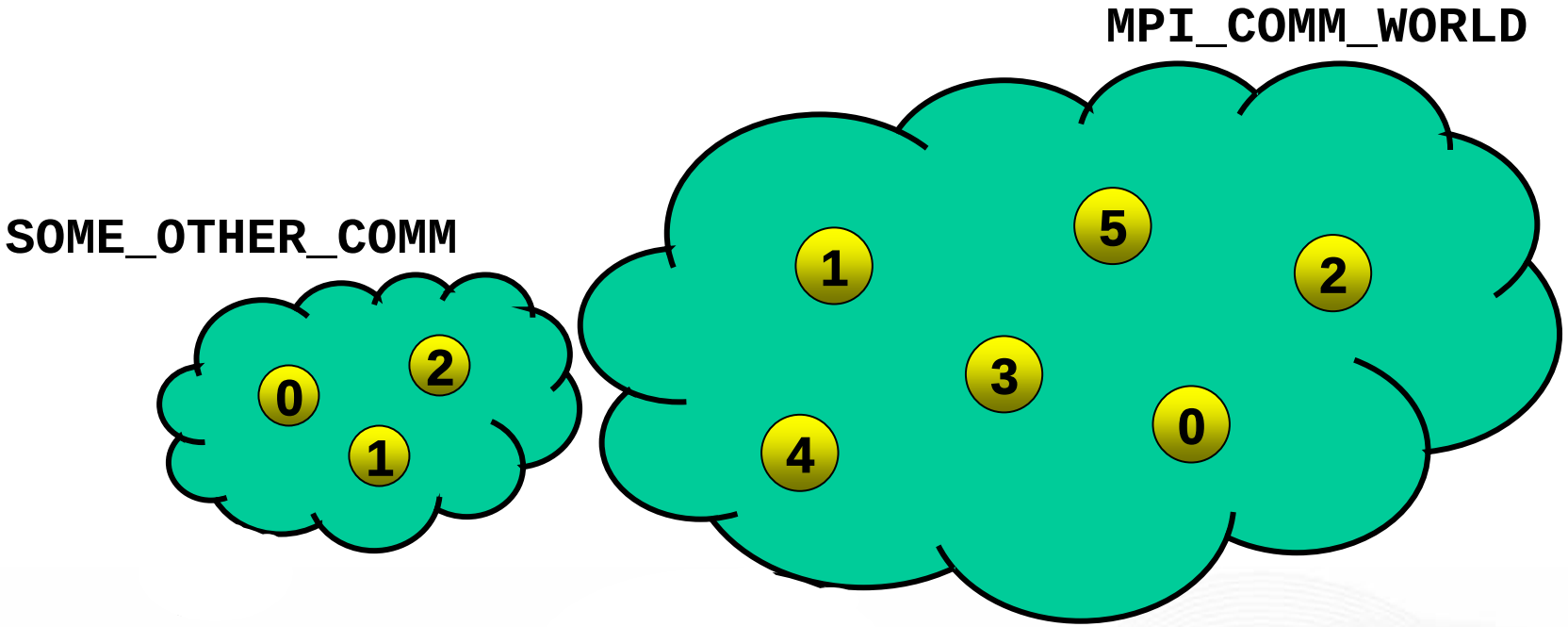
Handles

- MPI controls its own internal data structures (supports heterogeneity)
- MPI releases “handles” to allow programmers to refer to these
- C handles are pointers to `typedefs`
- In Fortran, all handles have type `INTEGER`

MPI communicator

- A handle representing a group of processes that can communicate with each other
- **All** MPI communication calls have a communicator argument
- Most often you will use `MPI_COMM_WORLD`
  - Defined when you call `MPI_Init`
  - It is all of your processors...

MPI communicator (cont.)



Processes 0, 1 and 2 are in **both** communicators; their **ranks** within their respective communicators will generally be different. `SOME_OTHER_COMM` should be thought of as living **within** `MPI_COMM_WORLD`.

Header files

- MPI constants, macros, definitions, function prototypes and handles are defined in a header file

C:

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

Fortran:

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

MPI function format

C (case sensitive):

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

Fortran (case unimportant):

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

Initializing MPI

- `MPI_Init` must be the **first** MPI routine called (only once)

C:

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

Fortran:

```
CALL MPI_INIT(IERROR)

INTEGER IERROR
```

Communicator size

- How many processes are contained within a communicator?

C:

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

Fortran:

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

INTEGER COMM, SIZE, IERROR
```

Process rank

- Process ID number within the communicator
  - Starts with zero and goes to (*n* - 1) where *n* is the number of processes requested
- Used to identify the source and destination of messages

C:

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

Fortran:

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

INTEGER COMM, RANK, IERROR
```

Exiting MPI

- No calls to MPI routines after finalization

C:

```
MPI_Finalize()
```

Fortran:

```
CALL MPI_FINALIZE(IERROR)
```

If any one process does not reach the finalization statement, the program will appear to hang.

bones.c

```
#include <mpi.h>

int main(int argc, char *argv[]) {
   int rank, size;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

/* … your code here … */

   MPI_Finalize ();
}
```

bones.f

```
      PROGRAM bones
      INCLUDE 'mpif.h'
      INTEGER ierror, rank, size
      CALL MPI_INIT(ierror)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)

C     ... your code here ...

      CALL MPI_FINALIZE(ierror)
      END
```

Exercise #1: Hello World

- Write a minimal MPI program that prints “hello world”
- Run it on several processors in parallel
- Modify your program so that only the process ranked 2 in `MPI_COMM_WORLD` prints out “hello world”
- Modify your program so that each process prints out its rank and the total number of processors

What's in a Message?

- Messages
- MPI basic datatypes - C
- MPI basic datatypes - Fortran
- Rules and rationale

Messages

- A message contains an array of elements of some particular MPI datatype

- MPI datatypes:
  - Basic types
  - Derived types

- Derived types can be build up from basic types

- C types are different from Fortran types

MPI basic datatypes - C

MPI Datatype	C Datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed log int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	
MPI_PACKED	

MPI basic datatypes - Fortran

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Rules and rationale

- Programmer declares variables to have “normal” C/Fortran type, but uses matching MPI datatypes as arguments in MPI routines

- Mechanism to handle type conversion in a **heterogeneous collection** of machines

- General rule: MPI datatype specified in a receive must match the MPI datatype specified in the send