Parallel programming / computation

Sultan ALPAR
s.alpar@iitu.edu.kz

IITU

Lecture 2

Process Model and Language Bindings

Header files

C

- C / C++

  #include <mpi.h>

Python

- Python

  from mpi4py import MPI

Fortran

- Fortran

  use mpi_f08

  use mpi (or: include ´mpif.h´)

• Available since MPI-3.0
• Full consistency with Fortran standard
• Compile-time argument checking in all MPI libraries
→ Recommended

Compile-time argument checking:
MPI-2.0 – 2.2: may be
MPI-3.0 and later: mandatory
but some MPI libraries still without.

Normally <u>without</u> any compile-time argument checking

The use of mpif.h is
- since MPI-3.0: strongly discouraged
- since MPI-4.1: **deprecated**

MPI Function Format

In C and Python: case sensitive

C

Python

Fortran

- C / C++: error = MPI_Xxxxxx(parameter, ...);
  MPI_Xxxxxx(parameter, ...);

Python interfaces: In analogy to the former MPI C++ interfaces in MPI-2.0 – MPI.2.2

- Python: result_value_or_object = input_mpi_object.mpi_action(parameter, …)

direct communication of numPy arrays (like in C)	comm_world = **MPI.COMM_WORLD** comm_world.**Send**((snd_buf, …), …) comm_world.**Recv**((rcv_buf, …), …)

Or with object-serialization:
comm_world.**send**(snd_buf, …)
rcv_buf = comm_world.**recv**(…)

- Fortran: CALL MPI_XXXXXX(parameter, ..., *ierror*)

With mpi_f08 module: ierror is optional (since MPI-3.0)

In Fortran: **not** case sensitive

With mpi module or mpif.h:

Absolutely Never forget!

MPI Function Format

In C and Python: case sensitive

C

Python

Fortran

- C / C++: error = MPI_Xxxxxx(parameter, ...);
  MPI_Xxxxxx(parameter, ...);

Python interfaces: In analogy to the former MPI C++ interfaces in MPI-2.0 – MPI.2.2

- Python: result_value_or_object = input_mpi_object.mpi_action(parameter, …)

direct communi-cation of numPy arrays (like in C)	comm_world = **MPI.COMM_WORLD** comm_world.**Send**((snd_buf, …), …) comm_world.**Recv**((rcv_buf, …), …)

Or with object-serialization:
comm_world.**send**(snd_buf, …)
rcv_buf = comm_world.**recv**(…)

- Fortran: CALL MPI_XXXXXX(parameter, ..., *ierror*)

With mpi_f08 module: ierror is optional (since MPI-3.0)

With mpi module or mpif.h:
Absolutely Never forget!

In Fortran: **not** case sensitive

Upper/mixed case	MPI standard	In this course
MPI_Xxx_mixed	MPI procedures in C	ditto in C and Python
	and in Fortran **mpi_f08** module	
		Language independent proc. Specifications
MPI_XXX_UPPER	Language independent proc. specifications	
	MPI procedures in **mpi** module and **mpif.h**	**all Fortran MPI procedures**
MPI_Xxx_mixed	MPI type declarations	ditto.
MPI_XXX_UPPER	MPI constants	ditto.

Recommendation: Use "mixed case" in your Fortran code

ierror with old mpif.h and new mpi_f08

Deprecated in MPI-4.1

mpi + mpif.h:
ierror is mandatory
➔NEVER FORGET!

- Unused ierror

  INCLUDE 'mpif.h'

  ! wrong call:

  CALL MPI_SEND(…., MPI_COMM_WORLD)

  ! → terrible implications because ierror=0 is written somewhere to the memory

mpi_f08:
ierror is OPTIONAL

- With the new module

  USE mpi_f08

  ! Correct call, because ierror is **optional**:

  CALL MPI_SEND(…., MPI_COMM_WORLD)

- **Conclusion:** You may switch to the **mpi_f08** module

MPI Function Format Details

- Linux, e.g., with xdg-open
- Windows: Acrobat is recommended

- Have a look into the MPI standard, e.g., MPI-4.0 page 37 (or MPI-3.1, page 28). Each MPI routine is defined:
  - language independent ($page_{lines}$ – $p37_{1\text{-}12}$ / $p28_{21\text{-}33}$),
  - programming languages: C / Fortran **mpi_f08** / **mpi** & **mpif.h** ($p37_{14\text{-}41}$ / $p28_{34\text{-}48}$).

New in MPI-3.0

C

Output arguments in **C/C++:**

definition in the standard

```
MPI_Comm_rank( ...., int *rank)
MPI_Recv(..., MPI_Status *status)
```

usage in your code:

```
main...
{  int myrank;  MPI_Status rcv_status;
   MPI_Comm_rank(..., &myrank);
   MPI_Recv(..., &rcv_status);
```

MPI Function Format Details

- Linux, e.g., with xdg-open
- Windows: Acrobat is recommended

- Have a look into the MPI standard, e.g., MPI-4.0 page 37 (or MPI-3.1, page 28). Each MPI routine is defined:
  - language independent ($page_{lines}$ – $p37_{1-12}$ / $p28_{21-33}$),
  - programming languages: C / Fortran **mpi_f08** / **mpi** & **mpif.h** ($p37_{14-41}$ / $p28_{34-48}$).

New in MPI-3.0

C

Output arguments in **C/C++:**

definition in the standard	MPI_Comm_rank(...., int *rank) MPI_Recv(..., MPI_Status *status)
usage in your code:	main... { int myrank; MPI_Status rcv_status; MPI_Comm_rank(..., &myrank); MPI_Recv(..., &rcv_status);

New in MPI-3.1

- Several index sections at the end: **General Index**, Examples, **Constant and Predefined Handle**, Declarations, Callback Function Prototype, **Function Index.**

MPI Function Format Details

- Linux, e.g., with xdg-open
- Windows: Acrobat is recommended

- Have a look into the MPI standard, e.g., MPI-4.0 page 37 (or MPI-3.1, page 28). Each MPI routine is defined:
  - language independent ($page_{lines}$ – $p37_{1\text{-}12}$ / $p28_{21\text{-}33}$),
  - programming languages: C / Fortran **mpi_f08** / **mpi** & **mpif.h** ($p37_{14\text{-}41}$ / $p28_{34\text{-}48}$).

New in MPI-3.0

C

Output arguments in **C/C++:**

definition in the standard
```
MPI_Comm_rank( ...., int *rank)
MPI_Recv(..., MPI_Status *status)
```

usage in your code:
```
main...
{  int myrank;  MPI_Status rcv_status;
   MPI_Comm_rank(..., &myrank);
   MPI_Recv(..., &rcv_status);
```

New in MPI-3.1

- Several index sections at the end: **General Index**, Examples, **Constant and Predefined Handle**, Declarations, Callback Function Prototype, **Function Index.**

- MPI_...... namespace is reserved for MPI constants and routines, i.e. application routines and variable names must not begin with MPI_ .

MPI Function Format Details

- Linux, e.g., with xdg-open
- Windows: Acrobat is recommended

- Have a look into the MPI standard, e.g., MPI-4.0 page 37 (or MPI-3.1, page 28). Each MPI routine is defined:
  - language independent (page$_{lines}$ – p37$_{1-12}$ / p28$_{21-33}$),
  - programming languages: C / Fortran **mpi_f08** / **mpi** & **mpif.h** (p37$_{14-41}$ / p28$_{34-48}$).

New in MPI-3.0

C

Output arguments in **C/C++:**

definition in the standard
```
MPI_Comm_rank( ...., int *rank)
MPI_Recv(..., MPI_Status *status)
```
usage in your code:
```
main...
{  int myrank;  MPI_Status rcv_status;
   MPI_Comm_rank(..., &myrank);
   MPI_Recv(..., &rcv_status);
```

New in MPI-3.1

- Several index sections at the end: **General Index**, Examples, **Constant and Predefined Handle**, Declarations, Callback Function Prototype, **Function Index.**
- MPI_...... namespace is reserved for MPI constants and routines, i.e. application routines and variable names must not begin with MPI_ .

Python

- mpi4py is not part of the MPI standard. Internally a wrapper to the C binding.

MPI 3.1 page 28
MPI 4.0 page 37

- **Language independent definition**
- **C interface**
- **Fortran 2008** interface through **mpi_f08** module
- Old **Fortran** interface through **mpi** module and **mpif.h**

17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
```

https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf#page=60
https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf#page=77

MPI 3.1 page 28
MPI 4.0 page 37

- **Language independent definition**

- **C interface**

- **Fortran 2008** interface through **mpi_f08** module

- Old **Fortran** interface through **mpi** module and **mpif.h**

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

Large count version in **MPI-4.0**
MPI_Recv_c(…) in C
with MPI_Count count
MPI_Recv(…) !(_c) in Fortran
with INTEGER(KIND=MPI_COUNT_KIND) :: count

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
```

https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf#page=60
https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf#page=77

MPI 3.1 page 28
MPI 4.0 page 37

- **Language independent definition**
- **C interface**
- **Fortran 2008** interface through **mpi_f08** module
- Old **Fortran** interface through **mpi** module and **mpif.h**

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Large count version in **MPI-4.0**
MPI_Recv_c(…) in C
with MPI_Count count
MPI_Recv(…) !(_c) in Fortran
with INTEGER(KIND=MPI_COUNT_KIND) :: count

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
```

No large count in mpi / mpif.h

https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf#page=60
https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf#page=77

Initializing MPI

MPI_Init() must be called before any other MPI routine
(only a few exceptions, e.g., MPI_Initialized)

C

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

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

MPI-2.0 and higher: Also MPI_Init(NULL, NULL);

Fortran

- MPI_INIT(IERROR)
  INTEGER IERROR

```fortran
program xxxxx
use mpi_f08
implicit  none
call MPI_INIT()
....
```

```fortran
! With MPI-2.0:
program xxxxx
use mpi
implicit  none
integer ierror
call MPI_INIT(ierror)
```

```fortran
! With MPI-1.1:
program xxxxx
implicit none
include ´mpif.h´
integer ierror
call MPI_INIT(ierror)
```

Deprecated in MPI-4.1

Initializing MPI

MPI_Init() must be called before any other MPI routine
(only a few exceptions, e.g., MPI_Initialized)

C

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

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

MPI-2.0 and higher:
Also
MPI_Init(NULL, NULL);

Fortran

- MPI_INIT(IERROR)
  INTEGER IERROR

With MPI-3.0 and later recommended:
use mpi_f08

```
program xxxxx
use mpi_f08
implicit  none
call MPI_INIT()
....
```

```
! With MPI-2.0:
program xxxxx
use mpi
implicit  none
integer ierror
call MPI_INIT(ierror)
```

Deprecated in MPI-4.1

```
! With MPI-1.1:
program xxxxx
implicit none
include ´mpif.h´
integer ierror
call MPI_INIT(ierror)
```

Initializing MPI

MPI_Init() must be called before any other MPI routine
(only a few exceptions, e.g., MPI_Initialized)

C

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

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

MPI-2.0 and higher: Also MPI_Init(NULL, NULL);

Fortran

- MPI_INIT(IERROR)
  INTEGER IERROR

```
program xxxxx
use mpi_f08
implicit none
call MPI_INIT()
....
```

```
! With MPI-2.0:
program xxxxx
use mpi
implicit none
integer ierror
call MPI_INIT(ierror)
```

Deprecated in MPI-4.1

```
! With MPI-1.1:
program xxxxx
implicit none
include ´mpif.h´
integer ierror
call MPI_INIT(ierror)
```

With MPI-3.0 and later recommended: use mpi_f08

If you install an MPI library with Fortran support: Never install MPI without mpi_f08 !

Test it with the version test → 2nd Advanced Exercise

Initializing MPI

MPI_Init() must be called before any other MPI routine
(only a few exceptions, e.g., MPI_Initialized)

C

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

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

MPI-2.0 and higher: Also MPI_Init(NULL, NULL);

Fortran

- MPI_INIT(IERROR)
  INTEGER IERROR

With MPI-3.0 and later recommended: use mpi_f08

```
program xxxxx
use mpi_f08
implicit  none
call MPI_INIT()
....
```

```
! With MPI-2.0:
program xxxxx
use mpi
implicit  none
integer ierror
call MPI_INIT(ierror)
```

Deprecated in MPI-4.1

```
! With MPI-1.1:
program xxxxx
implicit none
include ´mpif.h´
integer ierror
call MPI_INIT(ierror)
```

If you install an MPI library with Fortran support: Never install MPI without mpi_f08 !

Test it with the version test → 2nd Advanced Exercise

Python

- # MPI.Init()

This call is not needed, because automatically called at the import of MPI at the begin of the program

```
from mpi4py import MPI
# MPI.Init() is not needed
....
```

Fortran

The Fortran support methods

In MPI-4.0, new large count interfaces only in mpi_f08 !

Fortran support method	MPI-1.1	MPI-2	MPI-3	MPI-4.0	MPI-4.1	MPI-next	far future
USE mpi_f08	x	x	5	5	5	5	5
USE mpi	x	3	4	4 / 2b	2b	1	0
INCLUDE ´mpif.h´	3	3	2a	2a/b	1	0	0

Past (MPI-1.1 – MPI-4.0) · Today (MPI-4.1) · Maybe in the future (MPI-next, far future)

Level of Quality:

- 5 – valid and consistent with the Fortran standard (Fortran 2008 + TS 29113) [1)]
- 4 – valid and only partially consistent
- 3 – valid and small consistency (e.g., without argument checking)
- 2 – use is strongly (a) discouraged or (b) partially frozen (i.e., not with all new functions)
- 1 – deprecated
- 0 – removed
- x – not yet existing

[1)] For full consistency, Fortran 200**3** + TS29113 is enough.
Fortran 2018 and later versions include TS 29113.
Without TS29113, same partial consistency as with the mpi module.

Exiting MPI

C
Fortran
Python

- C/C++: int MPI_Finalize()
- Fortran: MPI_FINALIZE(*ierror*)
  mpi_f08: INTEGER, OPTIONAL :: ierror
  mpi & mpif.h: INTEGER ierror
- Python: # MPI.Finalize()

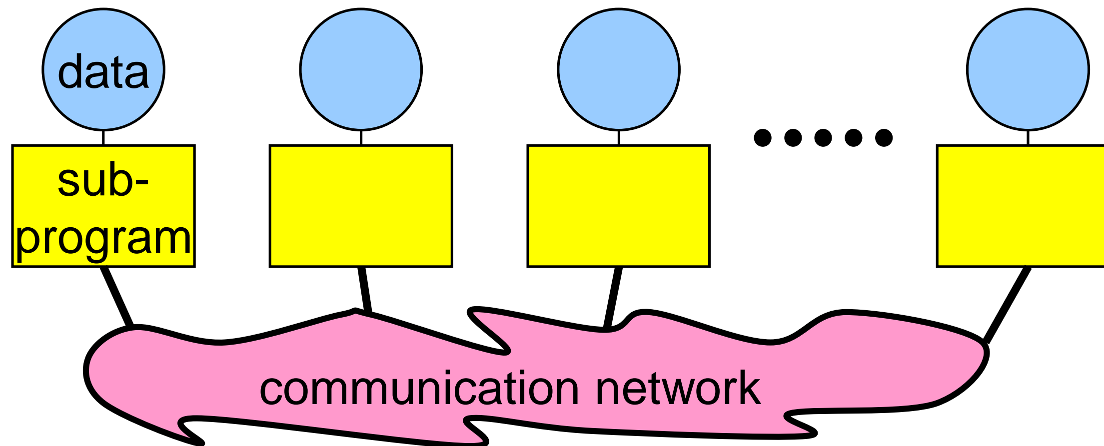
This call is not needed, because automatically called at the end of the program

- **Must** be called last by all processes.
- User must ensure the completion of all pending communications (locally) before calling finalize
- After MPI_Finalize:
  - Further MPI-calls are forbidden
  - Especially re-initialization with MPI_Init is forbidden
  - **May** abort the calling process if its rank in MPI_COMM_WORLD is ≠ 0
- Alternatives in MPI-4.0
  - World Model: MPI_Init/Finalize and MPI_COMM_WORLD
  - Sessions Model: See course chapter 8-(2)

New in MPI-4.0

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 and later)



- The parallel MPI processes exist at least after MPI_Init was called.

Exercise 1

Exercise 1: Hello World

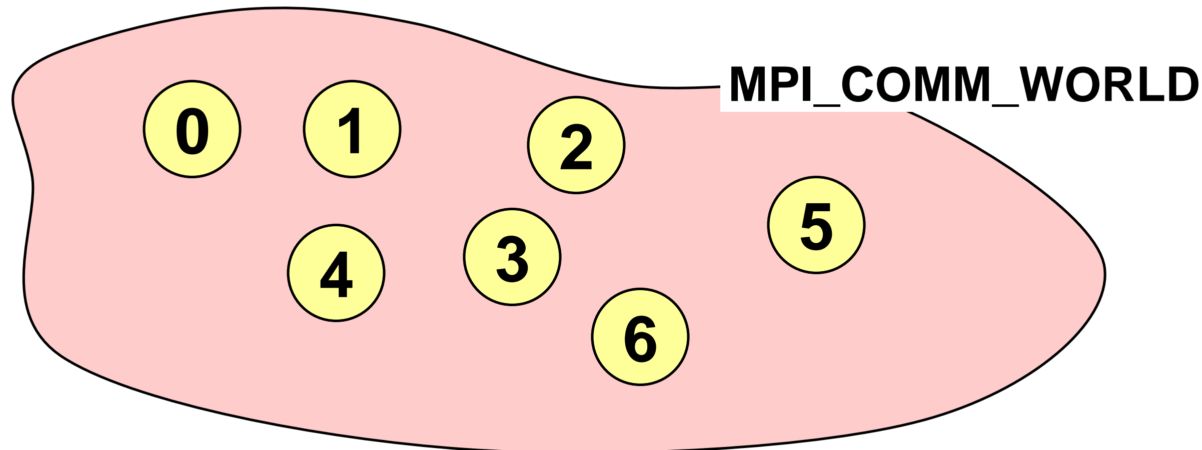
In MPI/tasks/…

- Use: C C/Ch2/hello-skel.c or Fortran F_30/Ch2/hello-skel_30.f90
  or Python PY/Ch2/hello-skel.py
- Write a minimal **MPI** program which prints „hello world" by each MPI process.
- Compile and run it on a single processor.
- Run it on several processors in parallel.
- Expected output on 4 processes

```
Hello world
Hello world
Hello world
Hello world
```

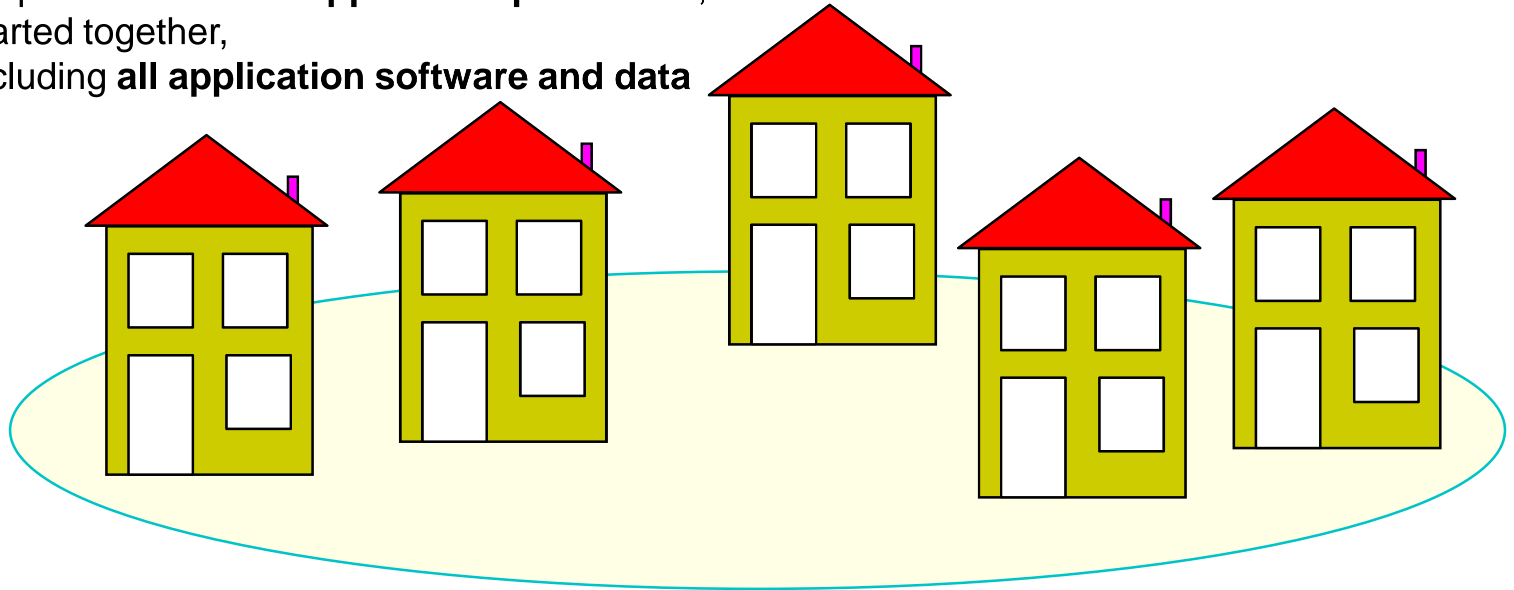
Communicator MPI_COMM_WORLD

- 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
  - mpi_f08 and mpi modules and mpif.h.
- Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)



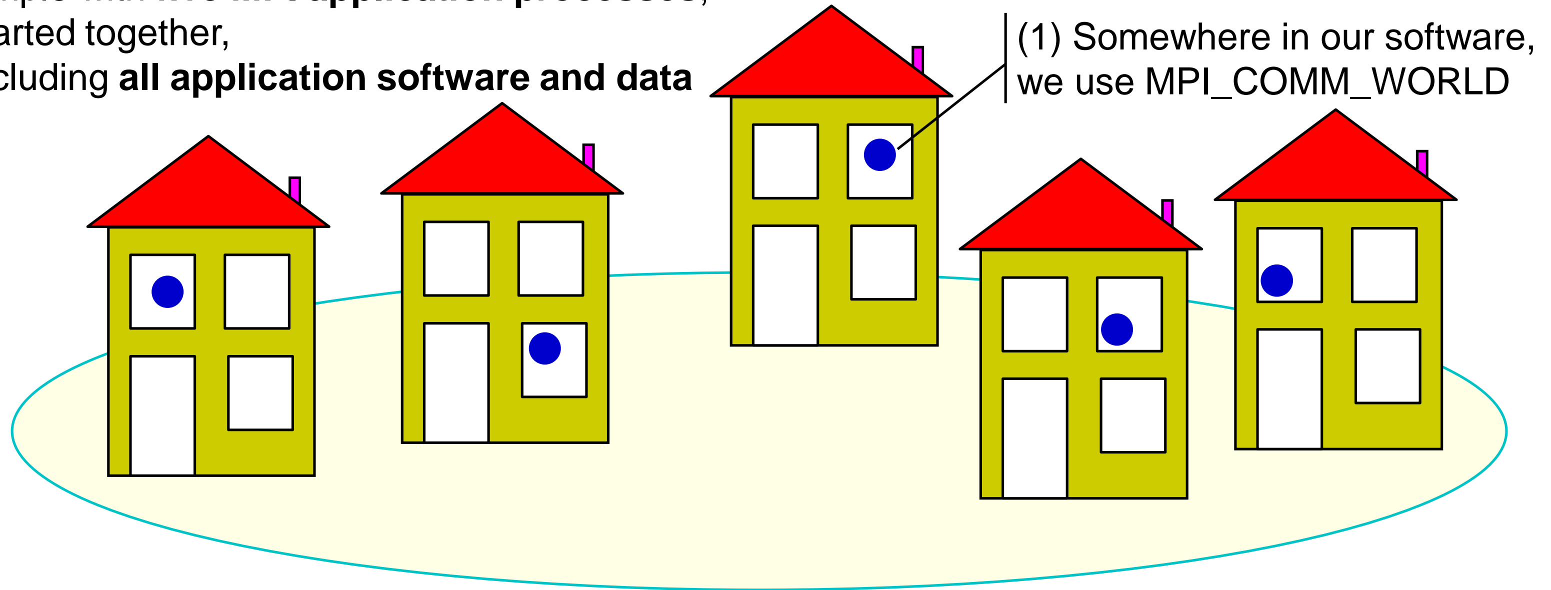
Handles refer to internal MPI data structures

Example with **five MPI application processes**,
- started together,
- including **all application software and data**

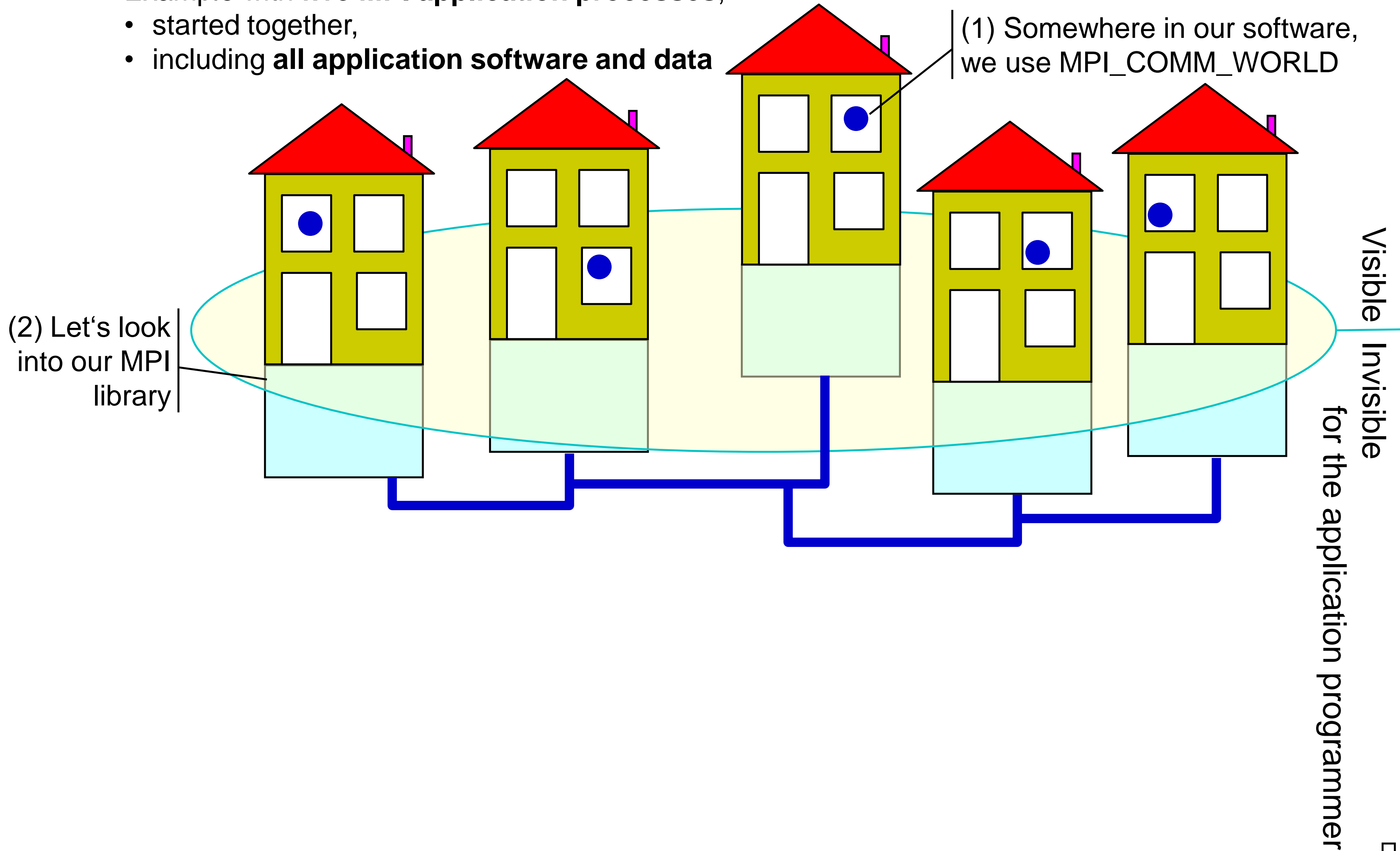


Handles refer to internal MPI data structures

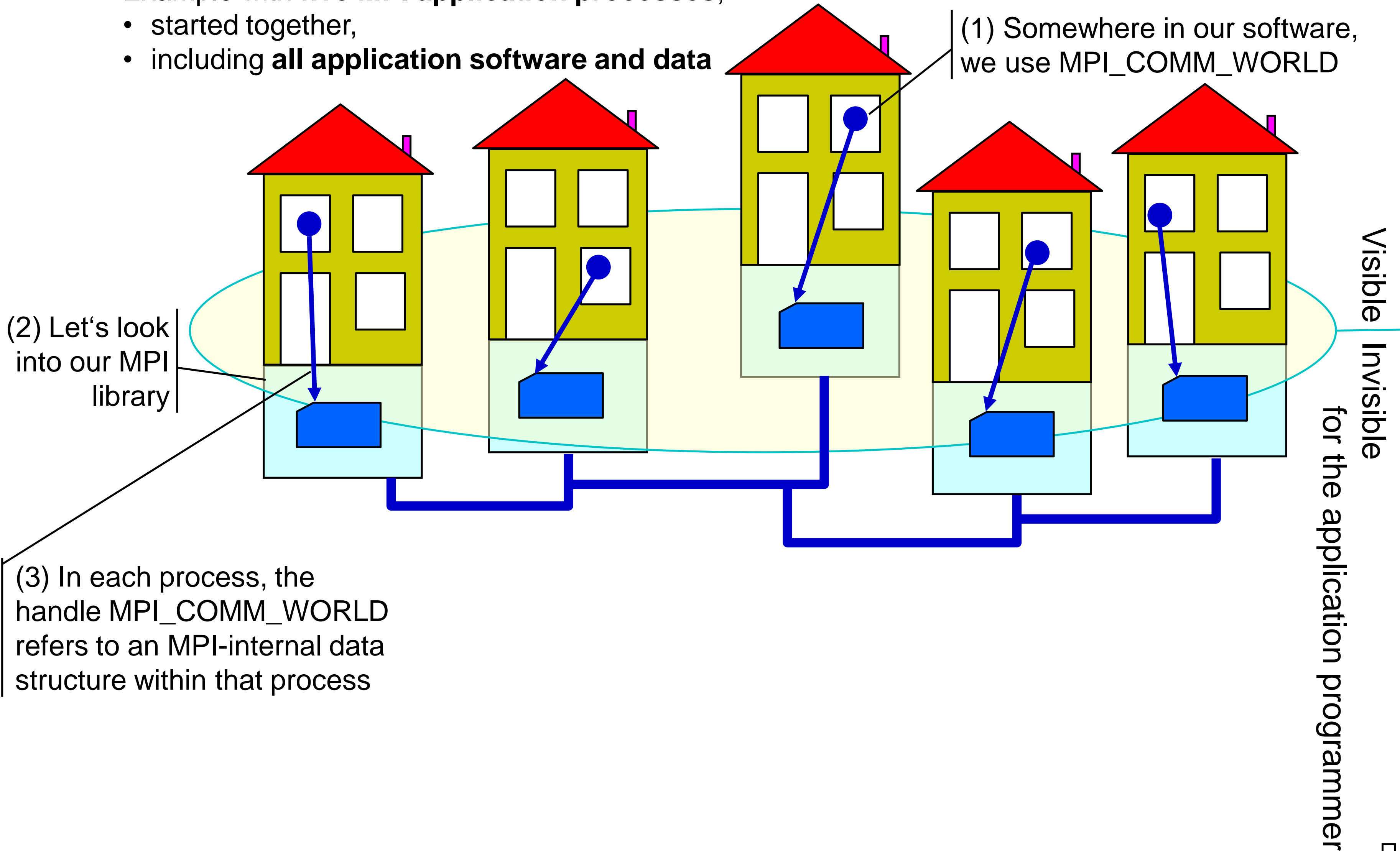
Example with **five MPI application processes**,
- started together,
- including **all application software and data**



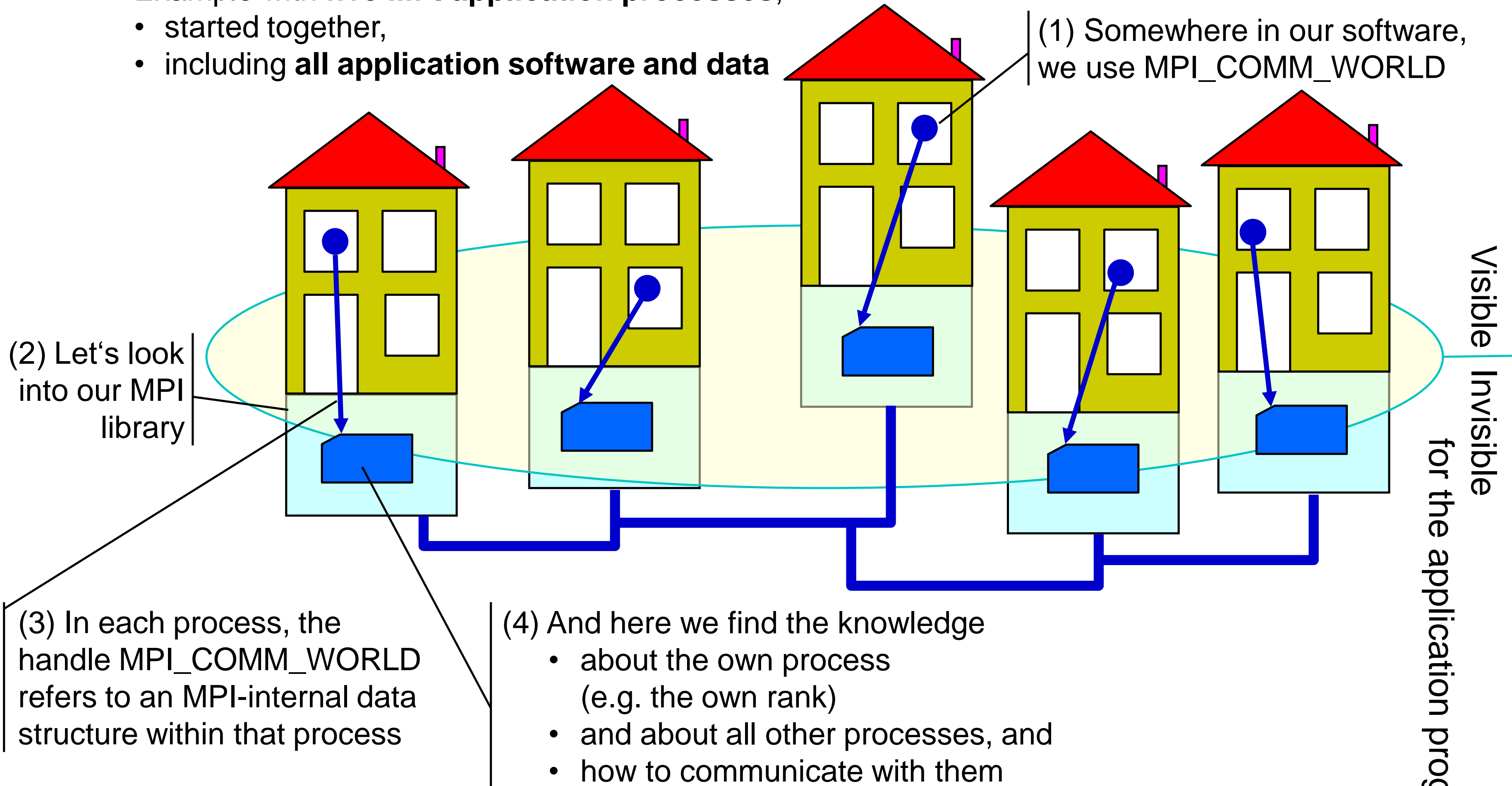
Handles refer to internal MPI data structures



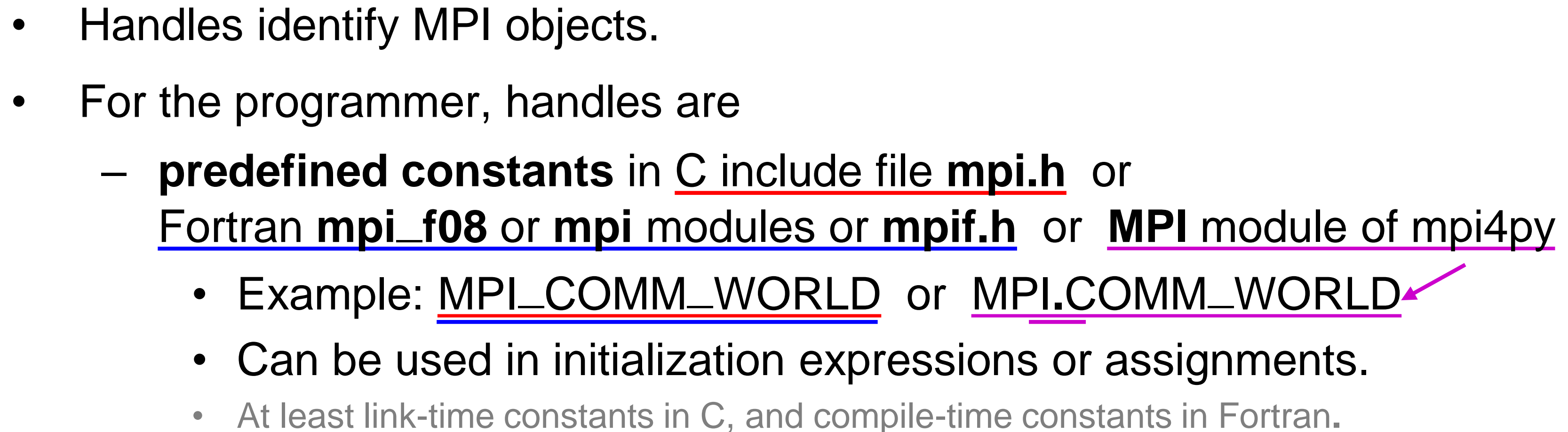
Handles refer to internal MPI data structures



Handles refer to internal MPI data structures



Handles



Handles

- Handles identify MPI objects.
- For the programmer, handles are
  - **predefined constants** in C include file **mpi.h** or Fortran **mpi_f08** or **mpi** modules or **mpif.h** or **MPI** module of mpi4py
    - Example: MPI_COMM_WORLD or MPI.COMM_WORLD
    - Can be used in initialization expressions or assignments.
    - At least link-time constants in C, and compile-time constants in Fortran.
  - **values returned** by some MPI routines, to be stored in variables, that are defined as
    - Fortran:
      - mpi_f08 module: TYPE(MPI_Comm) :: sub_comm
      - mpi module and mpif.h: INTEGER sub_comm
    - C: special MPI typedefs, e.g., MPI_Comm sub_comm;
    - Python: Type of object defined by the creating function, e.g., sub_comm = MPI.COMM_WORLD.Split(…)

New in MPI-3.0

Fortran

C

Python

Handles

- Handles identify MPI objects.
- For the programmer, handles are
  - **predefined constants** in C include file **mpi.h** or Fortran **mpi_f08** or **mpi** modules or **mpif.h** or **MPI** module of mpi4py
    - Example: MPI_COMM_WORLD or MPI**.**COMM_WORLD
    - Can be used in initialization expressions or assignments.
    - At least link-time constants in C, and compile-time constants in Fortran.
  - **values returned** by some MPI routines, to be stored in variables, that are defined as
    - Fortran:
      - mpi_f08 module: TYPE(MPI_Comm) :: sub_comm
      - mpi module and mpif.h: INTEGER sub_comm
    - C: special MPI typedefs, e.g., MPI_Comm sub_comm;
    - Python: Type of object defined by the creating function, e.g., sub_comm = MPI.COMM_WORLD.Split(…)
- Handles refer to internal MPI data structures

New in MPI-3.0

Fortran

C

Python

Rank

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

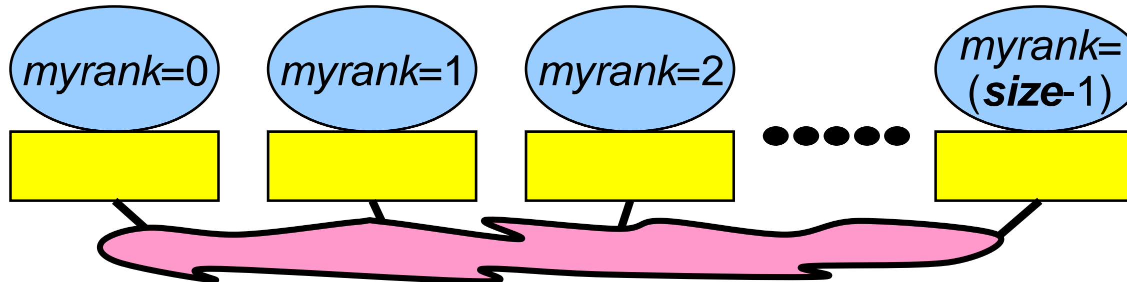
C

Fortran

Python

- C/C++: int MPI_Comm_rank(MPI_Comm comm, int **rank*)
- Fortran: MPI_COMM_RANK(comm, *rank*, *ierror*)
  mpi_f08: TYPE(MPI_Comm) :: comm
  INTEGER :: rank; INTEGER, OPTIONAL :: ierror
  mpi & mpif.h: INTEGER comm, rank, ierror
- Python: *rank* = comm.Get_rank()

INTENT(IN/OUT) is omitted on these slides



CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierror)

Size

- How many processes are contained within a communicator?

C

Fortran

Python

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

  mpi_f08: TYPE(MPI_Comm) :: comm
  INTEGER :: size
  INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: INTEGER comm, size, ierror
- Python: *size* = comm.Get_size()

Size

- How many processes are contained within a communicator?

C

Fortran

Python

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

  mpi_f08:
  ```
  TYPE(MPI_Comm) :: comm
  INTEGER :: size
  INTEGER, OPTIONAL :: ierror
  ```
  mpi & mpif.h:
  ```
  INTEGER comm, size, ierror
  ```
- Python: *size* = comm.Get_size()

Fortran & C: Interface definitions

- On these slides & in the MPI standard
- You have to write the corresponding procedure **calls**
- E.g., in C:
  ```
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  ```
- E.g., in Fortran:
  ```
  CALL MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierror)
  ```

MPI for Python (mpi4py.github.io)

MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)

The API reference (mpi4py.github.io/apiref/index.html)

Size

- How many processes are contained within a communicator?

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

  mpi_f08: TYPE(MPI_Comm) :: comm
  INTEGER :: size
  INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: INTEGER comm, size, ierror
- Python: size = comm.Get_size()

Fortran & C: Interface definitions

- On these slides & in the MPI standard
- You have to write the corresponding procedure **calls**
- E.g., in C:
  MPI_Comm_size (MPI_COMM_WORLD, **&**size)**;**
- E.g., in Fortran:
  CALL MPI_COMM_SIZE (MPI_COMM_WORLD, size**, ierror**)

Python: Mix of usage of the interface and typed argument list

- See MPI for Python (mpi4py.github.io), and MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io), and The API reference (mpi4py.github.io/apiref/index.html)

Exercise 2: I am my_rank of size

Exercise 2

In MPI/tasks/...

- Use: C C/Ch2/myrank-skel.c or Fortran F_30/Ch2/myrank-skel_30.f90
  or Python PY/Ch2/myrank-skel.py
- Modify this program so that
  - every process writes its rank and the size of MPI_COMM_WORLD,
  - only process ranked 0 in MPI_COMM_WORLD prints “hello world”.
- Why is the sequence of the output non-deterministic?

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

Exercise 3 – Advanced Exercises: Hello World with deterministic output

- Discuss with your neighbor or in your break out group, what must be done, that the output of all MPI processes on the terminal window is in the sequence of the ranks.
- Or is there no chance to guarantee this?

Exercise 4: Version test

- Copy the version test programs into your local working directory
  - C C/Ch2/ and Fortran F_30/Ch2 and Python PY/Ch2 contain following version test programs
- Compile and run → **Besides the version of MPI, it also tests …**

C

Fortran

- version_test.c → … exists mpi.h and the C bindings
- version_test_11.f → … exists mpif.h and the Fortran bindings
- version_test_20.f90 → … exists the mpi module + bindings
- version_test_30.f90 → … exists the mpi_f08 module + bindings
- version_test_keyarg_20.f90 → Contains the mpi module a correct bindings according to MPI-3.0 and higher, i.e., allowing also keyword based argument lists?
- version_test_keyarg_30.f90 → Same test for the mpi_f08 module

Python

- version_test.py