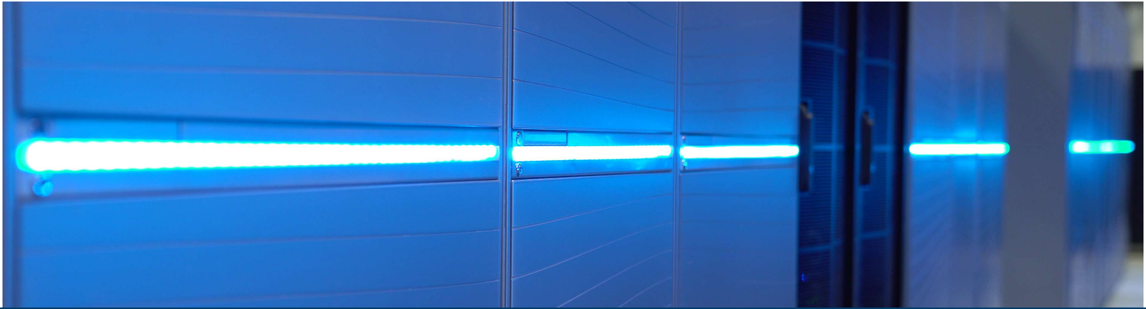
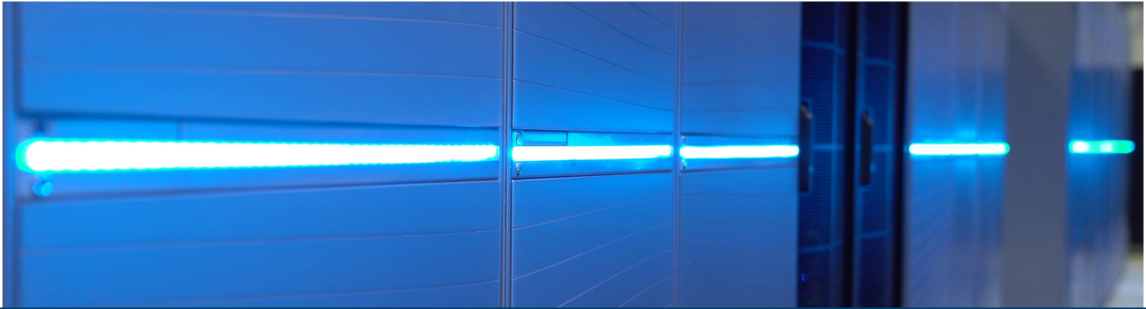


# INTRODUCTION TO PARALLEL PROGRAMMING WITH MPI AND OPENMP

March 18-20 2024 | Junxian Chew, Michael Knobloch, Ilya Zhukov, Jolanta Zjupa | Jülich Supercomputing Centre



# Part I: First Steps with MPI

# WHAT IS MPI?

> *MPI (**M**essage-**P**assing **I**nterface) is a message-passing library interface specification. [. . .] MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. (MPI Forum[1])*

- Industry standard for a message-passing programming model
- Provides specifications (no implementations)
- Implemented as a library with language bindings for Fortran and C
- Portable across different computer architectures

Current version of the standard: 4.1 (November 2023)

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 4.0. June 9, 2021. URL: `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`.



JÜLICH Forschungszentrum

# BRIEF HISTORY

| | |
|---|---|
| <1992 | several message-passing libraries were developed, PVM, P4,... |
| 1992 | At SC92, several developers for message-passing libraries agreed to develop a standard for message-passing |
| 1994 | MPI-1.0 standard published |
| 1997 | MPI-2.0 standard adds process creation and management, one-sided communication, extended collective communication, external interfaces and parallel I/O |
| 2008 | MPI-2.1 combines MPI-1.3 and MPI-2.0 |
| 2009 | MPI-2.2 corrections and clarifications with minor extensions |
| 2012 | MPI-3.0 nonblocking collectives, new one-sided operations, Fortran 2008 bindings |
| 2015 | MPI-3.1 nonblocking collective I/O |
| 2021 | MPI-4.0 large counts, persistent collective communication, partitioned communication, session model |
| 2023 | MPI-4.1 clarifications and minor extensions to MPI-4.0 |

# READING THE STANDARD

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

| | | |
|---|---|---|
| IN | buf | initial address of send buffer (choice) |
| IN | count | number of elements in send buffer (non-negative integer) |
| IN | datatype | datatype of each send buffer element (handle) |
| IN | dest | rank of destination (integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

# LITERATURE

## Official Resources

- Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 4.1. Nov. 2, 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf
- https://www.mpi-forum.org

## Further Resources

- MPICH C/C++/FORTRAN implementation: https://www.mpich.org/static/docs/latest/
- MPI for Python: https://mpi4py.readthedocs.io/en/stable/index.html

## Additional Literature

- William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI. Portable Parallel Programming with the Message-Passing Interface. 3rd ed. The MIT Press, Nov. 2014. 336 pp. ISBN: 9780262527392
- William Gropp et al. Using Advanced MPI. Modern Features of the Message-Passing Interface. 1st ed. Nov. 2014. 392 pp. ISBN: 9780262527637

## Acknowledgements

- Rolf Rabenseifner for his comprehensive course on MPI and OpenMP
- Marc-André Hermanns, Florian Janetzko, Alexander Trautmann and Benedikt Steinbusch for their course material on MPI and OpenMP

# COMPILING & LINKING [MPI-4.0, 19.1.7]

MPI libraries or system vendors usually ship compiler wrappers that set search paths and required libraries, e.g.:

**C Compiler Wrappers**

```
$ # Generic compiler wrapper shipped with e.g. OpenMPI
$ mpicc example.c -o example.exec
$ # Vendor specific wrapper for IBM's XL C compiler on BG/Q
$ bgxlc example.c -o example.exec
```

**Fortran Compiler Wrappers**

```
$ # Generic compiler wrapper shipped with e.g. OpenMPI
$ mpifort example.f90 -o example.exec
$ # Vendor specific wrapper for IBM's XL Fortran compiler on BG/Q
$ bgxlf90 example.f90 -o example.exec
```

However, neither the existence nor the interface of these wrappers is mandated by the standard.
**PYTHON**: no compilation is needed.

# PROCESS STARTUP [MPI-4.0, 11.5]

The MPI standard does not mandate a mechanism for process startup. It suggests that a command **`mpiexec`** with the following interface should exist:

Process Startup

```
$ # startup mechanism suggested by the standard
$ mpiexec -n <numprocs> <program.exec>
```

Sometimes one can also find the **`mpistart`** and **`mpirun`** command.

Process Startup

```
$ # Slurm startup mechanism as found on JSC systems
$ srun -n <numprocs> <program.exec>
```

**PYTHON**: `$ srun -n <numprocs>` *python* `<program.py>`

# LANGUAGE BINDINGS [MPI-4.0, 19, A]

## C Language Bindings

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

## Fortran Language Bindings

Consistent with F08 standard; good type-checking; highly recommended

F08
```
use mpi_f08
```

Not consistent with standard; so-so type-checking; not recommended

F90
```
use mpi
```

Not consistent with standard; no type-checking; strongly discouraged

F77
```
include 'mpif.h'
```



JÜLICH Forschungszentrum

# FORTRAN HINTS [MPI-4.0, 19.1.2 – 19.1.4]

This course uses the Fortran 2008 MPI interface (**`use`** `mpi_f08`) which is not available in all MPI implementations. The Fortran 90 bindings differ from the Fortran 2008 bindings in the following points:

- All derived **`type`** arguments are instead **`integer`** (some are arrays of **`integer`** or have a non-default `kind`)
- Argument **`intent`** is not mandated by the Fortran 90 bindings
- The `ierror` argument is mandatory instead of **`optional`**
- Further details can be found in [MPI-4.0, 19.1]

# MPI4PY HINTS

All exercises in the MPI part can be solved using Python with the `mpi4py` package. The slides do not show Python syntax, so here is a translation guide from the standard bindings to `mpi4py`.

- Everything lives in the `MPI` module (`from mpi4py import MPI`).
- Constants translate to attributes of that module: `MPI_COMM_WORLD` is `MPI.COMM_WORLD`.
- Central types translate to Python classes: `MPI_Comm` is `MPI.Comm`.
- Functions related to point-to-point and collective communication translate to methods on `MPI.Comm`: `MPI_Send` becomes `MPI.Comm.Send`.
- Functions related to I/O translate to methods on `MPI.File`: `MPI_File_write` becomes `MPI.File.Write`.
- Communication functions come in two flavors:
  - high level, uses `pickle` to (de)serialize python objects, method names start with lower case letters, e.g. `MPI.Comm.send`,
  - low level, uses MPI Datatypes and Python buffers, method names start with upper case letters, e.g. `MPI.Comm.Scatter`.

See also `https://mpi4py.readthedocs.io` and the built-in Python `help()`.

# OTHER LANGUAGE BINDINGS

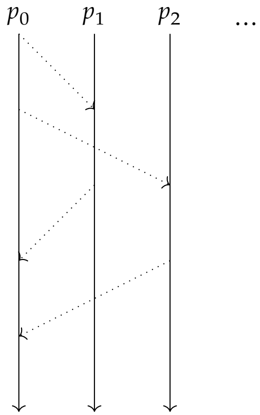
Besides the official bindings for C and Fortran mandated by the standard, unofficial bindings for other programming languages exist:

| | |
|---|---|
| C++ | Boost.MPI |
| MATLAB | Parallel Computing Toolbox |
| Python | pyMPI, mpi4py, pypar, MYMPI, … |
| R | Rmpi, pdbMPI |
| julia | MPI.jl |
| .NET | MPI.NET |
| Java | mpiJava, MPJ, MPJ Express |

And many others, ask your favorite search engine.

# WORLD ORDER IN MPI

- Program starts as $N$ distinct processes.
- Stream of instructions might be different for each process.
- Each process has access to its own private memory.
- Information is exchanged between processes via messages.
- Processes may consist of multiple threads (see OpenMP part on day 1).



# SERIAL CONTROL FLOW

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

# SERIAL CONTROL FLOW

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

# SERIAL CONTROL FLOW

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

# SERIAL CONTROL FLOW

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

```
Hello world!
```

# SERIAL CONTROL FLOW

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

```
Hello world!
```

# PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Process 1

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

# PARALLEL CONTROL FLOW (IN MPI)

**Process 0**

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

**Process 1**

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

**Console**

# PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Process 1

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

# PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Process 1

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

```
Hello world!
```

# PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Process 1

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

```
Hello world!
Hello world!
```

# PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Process 1

```
program example
  statement1
  if .true. then
    print *, "Hello world!"
  else
    print *, "Nonsense!"
  end if
  statement4
end program
```

Console

```
Hello world!
Hello world!
```

# INITIALIZATION [MPI-4.0, 11.2.1, 11.2.3]

Initialize MPI library, needs to happen before most other MPI functions can be used

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

F08
```fortran
MPI_Init(ierror)
integer, optional, intent(out) :: ierror
```

**PYTHON**: no initialisation needed.

# FINALIZATION [MPI-4.0, 11.2.2, 11.2.3]

Finalize MPI library when you are done using its functions

C

```c
int MPI_Finalize(void)
```

F08

```fortran
MPI_Finalize(ierror)
integer, optional, intent(out) :: ierror
```

**PYTHON**: no finalisation needed.

# PROCESS ORGANIZATION [MPI-4.0, 7.2]

**Process**

An MPI program consists of autonomous processes, executing their own code, in an MIMD style (multiple instruction, multiple data).

**Rank**

A unique number assigned to each process within a group (start at 0).

**Group**

An ordered set of process identifiers.

**Context**

A property of communicators that allows partitioning of the communication space. A message sent in one context cannot be received in another context.

**Communicator**

Scope for communication operations within or between groups, combines the concepts of group and context.

# OBJECTS [MPI-4.0, 2.5.1]

**Opaque Objects**

Most objects such as communicators, groups, etc. are opaque to the user and kept in regions of memory managed by the MPI library. They are created and marked for destruction using dedicated routines. Objects are made accessible to the user via handle values.

**Handle**

Handles are references to MPI objects. They can be checked for referential equality and copied, however copying a handle does not copy the object it refers to. Destroying an object that has operations pending will not disrupt those operations.

**Predefined Handles**

MPI defines several constant handles to certain objects, e.g. `MPI_COMM_WORLD` a communicator containing all processes initially partaking in a parallel execution of a program.

# PREDEFINED COMMUNICATORS

After MPI_Init has been called, MPI_COMM_WORLD is a valid handle to a predefined communicator that includes all processes available for communication. Additionally, the handle MPI_COMM_SELF is a communicator that is valid on each process and contains only the process itself.

C

```c
MPI_Comm MPI_COMM_WORLD;
MPI_Comm MPI_COMM_SELF;
```

F08

```fortran
type(MPI_Comm) :: MPI_COMM_WORLD
type(MPI_Comm) :: MPI_COMM_SELF
```

Py

```python
mpi4py.MPI.COMM_WORLD
mpi4py.MPI.COMM_SELF
```

# COMMUNICATOR SIZE [MPI-4.0, 7.4.1]

Determine the total number of processes in a communicator

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

F08
```fortran
MPI_Comm_size(comm, size, ierror)
type(MPI_Comm), intent(in) :: comm
integer, intent(out) :: size
integer, optional, intent(out) :: ierror
```

Py
```python
size = mpi4py.MPI.Comm.Get_size()
```

## Examples

C
```c
int size;
int ierror = MPI_Comm_size(MPI_COMM_WORLD, &size);
```

# PROCESS RANK [MPI-4.0, 7.4.1]

Determine the rank of the calling process within a communicator

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

F08
```fortran
MPI_Comm_rank(comm, rank, ierror)
type(MPI_Comm), intent(in) :: comm
integer, intent(out) :: rank
integer, optional, intent(out) :: ierror
```

Py
```python
rank = mpi4py.MPI.Comm.Get_rank()
```

## Examples

C
```c
int rank;
int ierror = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

# ERROR HANDLING [MPI-4.0, 9.3, 9.4, 9.5]

- Flexible error handling through error handlers which can be attached to
  - Communicators
  - Files
  - Windows
- Error handlers can be

| | |
|---|---|
| `MPI_ERRORS_ARE_FATAL` | Errors encountered in MPI routines abort execution |
| `MPI_ERRORS_RETURN` | An error code is returned from the routine |
| Custom error handler | A user supplied function is called on encountering an error |

- By default
  - Communicators use `MPI_ERRORS_ARE_FATAL`
  - Files use `MPI_ERRORS_RETURN`
  - Windows use `MPI_ERRORS_ARE_FATAL`

# BASIC CODE STRUCTURE IN C

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

int main(int argc, char **argv)
{
  int size;
  int rank;

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

  // here comes your MPI code

  MPI_Finalize();
  return(0);
}
```

# BASIC CODE STRUCTURE IN PYTHON

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

# here comes your MPI code
```

# MORE PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Process 1

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Console

# MORE PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Process 1

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Console

# MORE PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Process 1

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Console

# MORE PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Process 1

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Console

# MORE PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Process 1

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Console

```
process 1
```

# MORE PARALLEL CONTROL FLOW (IN MPI)

Process 0

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Process 1

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Console

```
process 1
process 0 of 2
```

# MORE PARALLEL CONTROL FLOW (IN MPI)

Process 0

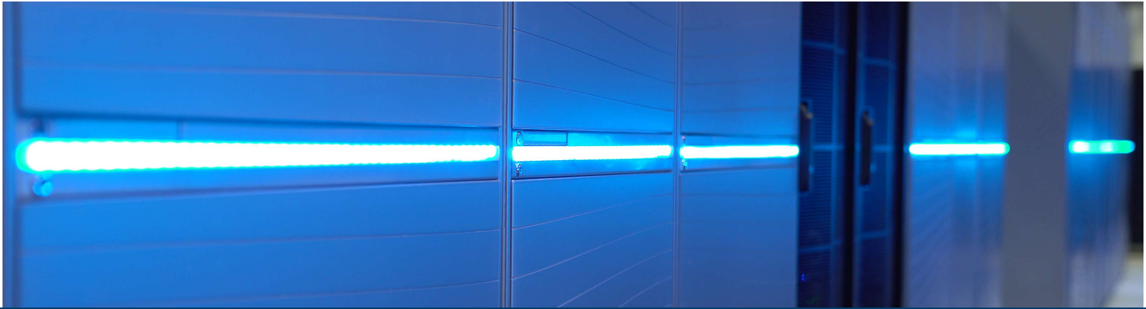
```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

Process 1

```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

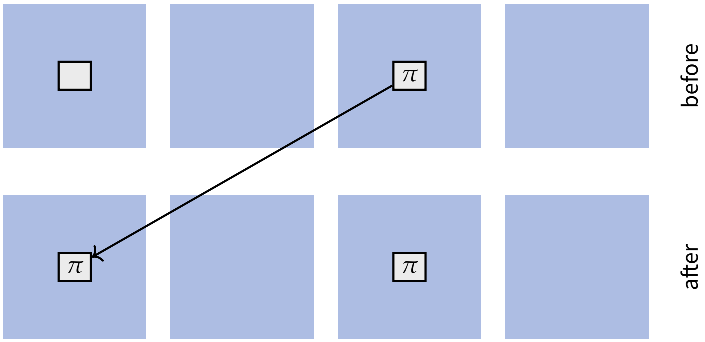
Console

```
process 1
process 0 of 2
```



# Part II: Blocking Point-to-Point Communication

# MESSAGE PASSING



# BLOCKING & NONBLOCKING PROCEDURES

| Blocking |
| --- |
| A procedure is blocking if return from the procedure indicates that the user is allowed to reuse resources specified in the call to the procedure. |

| Nonblocking |
| --- |
| All calls are local and return immediately. All associated send buffers and buffers associated with input arguments should not be modified, and all associated receive buffers should not be accessed, until the communication has been completed using an appropriate completion procedure. The call returns a request handle, which must be passed to a completion call. |

# PROPERTIES

- Communication between two processes within the same communicator
  *A process can send messages to itself.*
- A source process sends a message to a destination process using an MPI send routine
- A destination process needs to post a receive using an MPI receive routine
- The source process and the destination process are specified by their ranks in the communicator
- Every message sent with a point-to-point operation needs to be matched by a receive operation

# SENDING MESSAGES [MPI-4.0, 3.2.1]

*

```
MPI_Send( <buffer>, <destination> )
```

C

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

F08

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count, dest, tag
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Comm), intent(in) :: comm
integer, optional, intent(out) :: ierror
```

# MESSAGES [MPI-4.0, 3.2.2, 3.2.3]

A message consists of two parts:

**Envelope**

- Source process `source`
- Destination process `dest`
- Tag `tag`
- Communicator `comm`

**Data**

Message data is read from/written to buffers specified by:

- Address in memory `buf`
- Number of elements found in the buffer `count`
- Structure of the data `datatype`

# DATA TYPES [MPI-4.0, 3.2.2, 3.3, 5.1]

**Data Type**

Describes the structure of a piece of data

**Basic Data Types**

Named by the standard, most correspond to basic data types of C or Fortran

| C type | MPI basic data type |
|---|---|
| `signed int` | `MPI_INT` |
| `float` | `MPI_FLOAT` |
| `char` | `MPI_CHAR` |
| … | |

| Fortran type | MPI basic data type |
|---|---|
| `integer` | `MPI_INTEGER` |
| `real` | `MPI_REAL` |
| `character` | `MPI_CHARACTER` |
| … | |

**Derived Data Type**

Data types which are not basic datatypes. These can be constructed from other (basic or derived) datatypes.

# DATA TYPE MATCHING [MPI-4.0, 3.3]

## Untyped Communication

- Contents of send and receive buffers are declared as `MPI_BYTE`.
- Actual contents of buffers can be any type (possibly different).
- Use with care.

## Typed Communication

- Type of buffer contents must match MPI data type (e.g. in C **`int`** and `MPI_INT`).
- Data type on send must match data type on receive operation.
- Allows data conversion when used on heterogeneous systems.

## Packed data

See [MPI-4.0, 5.2]

# QUIZ

**How are buffers typically specified in MPI?**

1. Start address and end address
2. Start address and count
3. Start address, count, and data type

# RECEIVING MESSAGES [MPI-4.0, 3.2.4]

*

```
MPI_Recv( <buffer>, <source> ) -> <status>
```

C

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

F08

```
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
```

- `count` specifies the capacity of the buffer
- Wildcard values are permitted (`MPI_ANY_SOURCE` & `MPI_ANY_TAG`)

# THE MPI_STATUS TYPE [MPI-4.0, 3.2.5]

Contains information about received messages

C
```c
MPI_Status status;
status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
```

F08
```fortran
type(MPI_status) :: status
status%MPI_SOURCE
status%MPI_TAG
status%MPI_ERROR
```

C
```c
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int
 ↪ *count)
```

F08
```fortran
MPI_Get_count(status, datatype, count, ierror)
type(MPI_Status), intent(in) :: status
type(MPI_Datatype), intent(in) :: datatype
integer, intent(out) :: count
integer, optional, intent(out) :: ierror
```

Pass `MPI_STATUS_IGNORE` to `MPI_Recv` if not interested.

# PROBE [MPI-4.0, 3.8.1]

*

```
MPI_Probe( <source> ) -> <status>
```

C

```c
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

F08

```fortran
MPI_Probe(source, tag, comm, status, ierror)
integer, intent(in) :: source, tag
type(MPI_Comm), intent(in) :: comm
type(MPI_Status), intent(out) :: status
integer, optional, intent(out) :: ierror
```

Returns after a matching message is ready to be received.

- Same rules for message matching as receive routines
- Wildcards permitted for `source` and `tag`
- `status` contains information about message (e.g. number of elements)

# MESSAGE ASSEMBLY

| Buffer | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|

```
MPI_Send(buffer, 4, MPI_INT, ...)
```

| Message | 0 | 1 | 2 | 3 |
|---|---|---|---|---|

```
MPI_Recv(buffer, 4, MPI_INT, ...)
```

| Buffer | 0 | 1 | 2 | 3 | ? | ? | ? | ? | ? | ? | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|

# SEND MODES [MPI-4.0, 3.4]

**Synchronous send: `MPI_Ssend`**

Only completes when the receive has started.

**Buffered send: `MPI_Bsend`**

- May complete before a matching receive is posted
- Needs a user-supplied buffer (see `MPI_Buffer_attach`)

**Standard send: `MPI_Send`**

- Either synchronous or buffered, leaves decision to MPI
- If buffered, an internal buffer is used

**Ready send: `MPI_Rsend`**

- Asserts that a matching receive has already been posted (otherwise generates an error)
- Might enable more efficient communication

# SYNCHRONOUS SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Ssend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```



JÜLICH Forschungszentrum

# SYNCHRONOUS SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Ssend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# SYNCHRONOUS SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Ssend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# SYNCHRONOUS SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Ssend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# SYNCHRONOUS SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Ssend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# SYNCHRONOUS SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Ssend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# BUFFERED SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Bsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# BUFFERED SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Bsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# BUFFERED SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Bsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# BUFFERED SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Bsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# BUFFERED SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Bsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# BUFFERED SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Bsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# BUFFERED SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Bsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# QUIZ

**In the example, what would happen if process 0 finished executing before process 1 started receiving?**

1. The computation would abort.
2. The computation would behave in an implementation defined way.
3. Trick question! Before it finishes, a conforming program has to call `MPI_Finalize` which can block until outstanding buffered messages have been sent.

# READY SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Rsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

Console

# READY SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Rsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

Console

# READY SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Rsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

Console

```
CRASH!
```



JÜLICH Forschungszentrum

# READY SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Rsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# READY SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Rsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# READY SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Rsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# READY SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Rsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# READY SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Rsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# READY SEND CONTROL FLOW

Process 0

```
subroutine A
  statement1
  call MPI_Rsend(..., 1, ...)
  statement3
end subroutine
```

Process 1

```
subroutine B
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end subroutine
```

# RECEIVE MODES [MPI-4.0, 3.4]

Only one receive routine for all send modes:

**Receive: `MPI_Recv`**

- Completes when a message has arrived and message data has been stored in the buffer
- Same routine for all communication modes

All blocking routines, both send and receive, guarantee that buffers can be reused after control returns.

# POINT-TO-POINT SEMANTICS [MPI-4.0, 3.5]

**Order**

In single threaded programs, messages are non-overtaking. Between any pair of processes, messages will be received in the order they were sent.

**Progress**

Out of a pair of matching send and receive operations, at least one is guaranteed to complete.

**Fairness**

Fairness is not guaranteed by the MPI standard.

**Resource limitations**

Resource starvation may lead to deadlock, e.g. if progress relies on availability of buffer space for standard mode sends.

# DEADLOCK

Structure of program prevents blocking routines from ever completing, e.g.:

Process 0

```
call MPI_Ssend(..., 1, ...)
call MPI_Recv(..., 1, ...)
```

Process 1

```
call MPI_Ssend(..., 0, ...)
call MPI_Recv(..., 0, ...)
```

Mitigation Strategies

- Changing communication structure (e.g. checkerboard)
- Using `MPI_Sendrecv`
- Using nonblocking routines

# DEADLOCK

Structure of program prevents blocking routines from ever completing, e.g.:

Process 0

```
call MPI_Ssend(..., 1, ...)
call MPI_Recv(..., 1, ...)
```

Process 1

```
call MPI_Ssend(..., 0, ...)
call MPI_Recv(..., 0, ...)
```

Mitigation Strategies

- Changing communication structure (e.g. checkerboard)
- Using `MPI_Sendrecv`
- Using nonblocking routines

# DEADLOCK

Structure of program prevents blocking routines from ever completing, e.g.:

Process 0

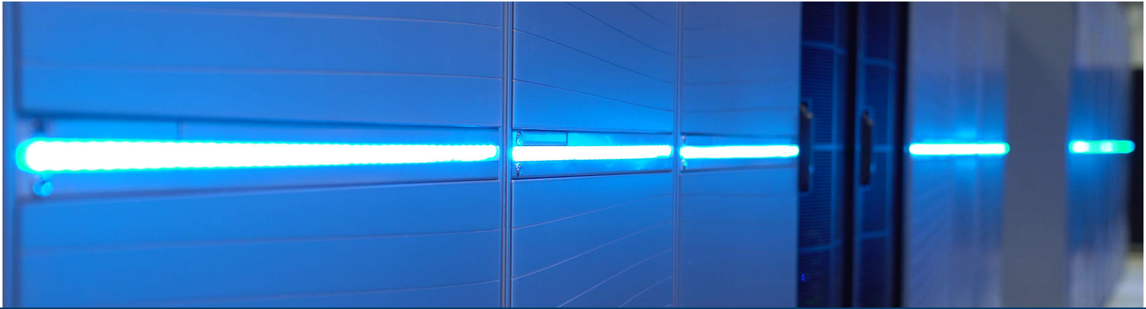
```
call MPI_Ssend(..., 1, ...)
call MPI_Recv(..., 1, ...)
```

Process 1

```
call MPI_Ssend(..., 0, ...)
call MPI_Recv(..., 0, ...)
```

Mitigation Strategies

- Changing communication structure (e.g. checkerboard)
- Using `MPI_Sendrecv`
- Using nonblocking routines



# Part III: Nonblocking Point-to-Point Communication

# BLOCKING & NONBLOCKING PROCEDURES

| Blocking |
|---|
| A procedure is blocking if return from the procedure indicates that the user is allowed to reuse resources specified in the call to the procedure. |

| Nonblocking |
|---|
| All calls are local and return immediately. All associated send buffers and buffers associated with input arguments should not be modified, and all associated receive buffers should not be accessed, until the communication has been completed using an appropriate completion procedure. The call returns a request handle, which must be passed to a completion call. |

# RATIONALE [MPI-4.0, 3.7]

## Premise

Communication operations are split into start and completion. The start routine produces a request handle that represents the in-flight operation and is used in the completion routine. The user promises to refrain from accessing the contents of message buffers while the operation is in flight.

## Benefit

A single process can have multiple nonblocking operations in flight at the same time. This enables communication patterns that would lead to deadlock if programmed using blocking variants of the same operations. Also, the additional leeway given to the MPI library may be utilized to, e.g.:

- overlap computation and communication
- overlap communication
- pipeline communication

# INITIATION ROUTINES [MPI-4.0, 3.7.2]

| Send | | | |
|---|---|---|---|
| Synchronous | `MPI_Issend` | Buffered | `MPI_Ibsend` |
| Standard | `MPI_Isend` | Ready | `MPI_Irsend` |

| Receive |
|---|
| `MPI_Irecv` |

| Probe |
|---|
| `MPI_Iprobe` |

- "I" is for immediate.
- Signature is similar to blocking counterparts with additional request object.
- Initiate operations and relinquish access rights to any buffer involved.

# NONBLOCKING SEND [MPI-4.0, 3.7.2]

*

```
MPI_Isend( <buffer>, <destination> ) -> <request>
```

C

```
int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest,
↪ int tag, MPI_Comm comm, MPI_Request *request)
```

F08

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
type(*), dimension(..), intent(in), asynchronous :: buf
integer, intent(in) :: count, dest, tag
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Comm), intent(in) :: comm
type(MPI_Request), intent(out) :: request
integer, optional, intent(out) :: ierror
```

# NONBLOCKING RECEIVE [MPI-4.0, 3.7.2]

*

```
MPI_Irecv( <buffer>, <source> ) -> <request>
```

C

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int
↪  tag, MPI_Comm comm, MPI_Request *request)
```

F08

```
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
type(*), dimension(..), asynchronous :: buf
integer, intent(in) :: count, source, tag
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Comm), intent(in) :: comm
type(MPI_Request), intent(out) :: request
integer, optional, intent(out) :: ierror
```

# NONBLOCKING PROBE [MPI-4.0, 3.8.1]

*

```
MPI_Iprobe( <source> ) -> <status>?
```

C

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status
 ↪ *status)
```

F08

```
MPI_Iprobe(source, tag, comm, flag, status, ierror)
integer, intent(in) :: source, tag
type(MPI_Comm), intent(in) :: comm
logical, intent(out) :: flag
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- Does not follow start/completion model.
- Uses true/false flag to indicate availability of a message.

# WAIT [MPI-4.0, 3.7.3]

*

```
MPI_Wait( <request> ) -> <status>
```

C

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

F08

```
MPI_Wait(request, status, ierror)
type(MPI_Request), intent(inout) :: request
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- Blocks until operation associated with `request` is completed
- To wait for the completion of several pending operations

| | |
|---|---|
| `MPI_Waitall` | All events complete |
| `MPI_Waitsome` | At least one event completes |
| `MPI_Waitany` | Exactly one event completes |

# TEST [MPI-4.0, 3.7.3]

*

```
MPI_Test( <request> ) -> <status>?
```

C

```c
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

F08

```fortran
MPI_Test(request, flag, status, ierror)
type(MPI_Request), intent(inout) :: request
logical, intent(out) :: flag
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- Does not block
- `flag` indicates whether the associated operation has completed
- Test for the completion of several pending operations

| | |
|---|---|
| `MPI_Testall` | All events complete |
| `MPI_Testsome` | At least one event completes |
| `MPI_Testany` | Exactly one event completes |



JÜLICH Forschungszentrum

# FREE [MPI-4.0, 3.7.3]

*

```
MPI_Request_free( <request> )
```

C

```c
int MPI_Request_free(MPI_Request *request)
```

F08

```fortran
MPI_Request_free(request, ierror)
type(MPI_Request), intent(inout) :: request
integer, optional, intent(out) :: ierror
```

- Marks the request for deallocation
- Invalidates the request handle
- Operation is allowed to complete
- Completion cannot be checked for

# CANCEL [MPI-4.0, 3.8.4]

*

```
MPI_Cancel( <request> )
```

C

```c
int MPI_Cancel(MPI_Request *request)
```

F08

```fortran
MPI_Cancel(request, ierror)
type(MPI_Request), intent(in) :: request
integer, optional, intent(out) :: ierror
```

- Marks an operation for cancellation
- Request still has to be completed via `MPI_Wait`, `MPI_Test` or `MPI_Request_free`
- Operation is either cancelled completely or succeeds (indicated in status value)



JÜLICH Forschungszentrum

# BLOCKING VS. NONBLOCKING OPERATIONS

- A blocking send can be paired with a nonblocking receive and vice versa
- Nonblocking sends can use any mode, just like the blocking counterparts
  - Synchronization of `MPI_Issend` is enforced at completion (wait or test)
  - Asserted readiness of `MPI_Irsend` must hold at start of operation
- A nonblocking operation immediately followed by a matching wait is equivalent to the blocking operation

## The Fortran Language Bindings and nonblocking operations

- Arrays with subscript triplets (e.g. `a(1:100:5)`) can only be reliably used as buffers if the compile time constant `MPI_SUBARRAYS_SUPPORTED` equals `.true.` [MPI-4.0, 19.1.12]
- Arrays with vector subscripts must not be used as buffers [MPI-4.0, 19.1.13]
- Fortran compilers may optimize your program beyond the point of being correct. Communication buffers should be protected by the **`asynchronous`** attribute (make sure `MPI_ASYNC_PROTECTS_NONBLOCKING` is `.true.`) [MPI-4.0, 19.1.16–19.1.20]

# OVERLAPPING COMMUNICATION

- Main benefit is overlap of communication with communication
- Overlap with computation
  - Progress may only be done inside of MPI routines
  - Not all platforms perform significantly better than well placed blocking communication
  - If hardware support is present, application performance may significantly improve due to overlap
- General recommendation
  - Initiation of operations should be placed as early as possible
  - Completion should be placed as late as possible

# QUIZ

**What are the semantics of synchronous send (`MPI_Ssend`)?**

1. It buffers the message data and returns independent of the recipients progress.
2. It blocks until the recipient has started receiving.
3. It creates an error if the recipient has not already initiated the receive operation.

# NONBLOCKING CONTROL FLOW

**Process 0**

```
program example
  call MPI_Issend(..., 1, ...)
  statement2
  call MPI_Wait(...)
  statement4
end program
```

**Process 1**

```
program example
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end program
```

# NONBLOCKING CONTROL FLOW

**Process 0**

```
program example
  call MPI_Issend(..., 1, ...)
  statement2
  call MPI_Wait(...)
  statement4
end program
```

**Process 1**

```
program example
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end program
```

# NONBLOCKING CONTROL FLOW

Process 0

```
program example
  call MPI_Issend(..., 1, ...)
  statement2
  call MPI_Wait(...)
  statement4
end program
```

Process 1

```
program example
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end program
```

# NONBLOCKING CONTROL FLOW

Process 0

```
program example
  call MPI_Issend(..., 1, ...)
  statement2
  call MPI_Wait(...)
  statement4
end program
```

Process 1

```
program example
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end program
```

# NONBLOCKING CONTROL FLOW

Process 0

```
program example
  call MPI_Issend(..., 1, ...)
  statement2
  call MPI_Wait(...)
  statement4
end program
```

Process 1

```
program example
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end program
```

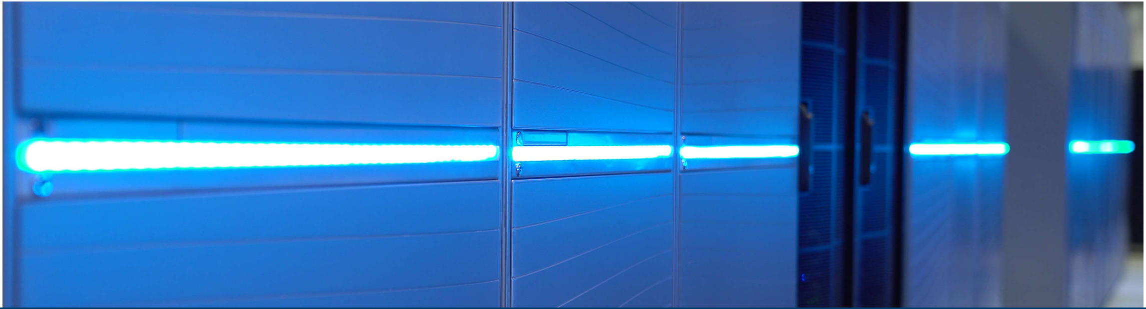
# NONBLOCKING CONTROL FLOW

Process 0

```
program example
  call MPI_Issend(..., 1, ...)
  statement2
  call MPI_Wait(...)
  statement4
end program
```

Process 1

```
program example
  statement1
  call MPI_Recv(..., 0, ...)
  statement3
end program
```



# Part IV: Blocking Point-to-Point Communication Exercises

# EXERCISES

Exercise 1 – MPI Hello World!

## 1.1 Hello World

An empty file `hello_world.{c|F90|py}` is provided for you. Your tasks are:

- Write a parallel programme, such that each process should print the following text:
  `hello world. from process` *i* `out of` *n* `processes.`
- *i* denotes the rank of the process, and *n* the total number of participating processes.
- Compile and run the application on 8 processes. You can use the following command:
  C|Fortran: `srun --ntasks-per-node=8 <your_application_name>`
  Python: `srun --ntasks-per-node=8 python ./hello_world.py`

Remember to include the required `MPI` libraries in the header of the file.

Use:
`MPI_Comm_size` for C|Fortran, `mpi4py.MPI.COMM_WORLD.Get_size()` for Python
`MPI_Comm_rank` for C|Fortran, `mpi4py.MPI.COMM_WORLD.Get_rank()` for Python

# EXERCISES

Exercise 2 – Send and Recv

## 2.1 Sending a number

A template file `skeleton.{c|F90|py}` is provided for you.
Copy the file into a new file named `neighbour_sendrecv_1way.{c|F90|py}`
The task:

- The program is intended to run on two processes.
- Write a parallel program that Rank 0 process sends its rank number to Rank 1 process.
- The message should be sent with tag value of `42`.
- Rank 1 then prints the following message:
  `I am rank 1, I have received message` *i* from rank 0.
- *i* denotes the number that is sent by Rank 0.

Use:
`MPI_Send` and `MPI_Recv` for C|Fortran, `comm.send()` and `comm.recv()` for Python
Consider/Read up on `MPI_ANY_TAG` and `MPI_STATUS_IGNORE`.

# EXERCISES

Exercise 3 – Send and Recv

## 3.1 Sending a number 2

A template file `skeleton.{c|F90|py}` is provided for you.
Copy the file into a new file named `neighbour_sendrecv_2way.{c|F90|py}`
The task:

- The program is intended to run on two processes.
- Write a parallel program that participating processes send their rank number to each other.
- Both processes then prints the following message:
  `I am rank` *m*, I have received message *i* from rank *s*.
- *m* denotes the rank number of self, *i* is the content of the passed message, and *s* is the rank of the sender.
- In this very simple scenario, *i* and *s* is identitcal.

Use:
`MPI_Send` and `MPI_Recv` for C|Fortran, `comm.send()` and `comm.recv()` for Python
Alternative is the `MPI_Sendrecv`, or `sendrecv()`.

# EXERCISES

Exercise 4 – $N$ processes comm pattern

## 4.1 Summing the ranks

A template file `skeleton.{c|F90|py}` is provided for you.
Copy the file into a new file named `ring_sendrecv.{c|F90|py}`
Descriptions of the MPI programme:

- The MPI program should produce a sum of the rank of all processes.
- All processes should carry the summed value.
- All processes then prints the following message:
  `I am rank` *m*, I have obtained the sum of all rank=*i*.
- *m* denotes the rank number of self, *i* is the total sum of ranks.
- The MPI program should be tested with 4, 8 and 12 processes. The sums should then be 6, 28, and 66.

Feel free to use any of the P2P communication calls, beware of deadlocks!