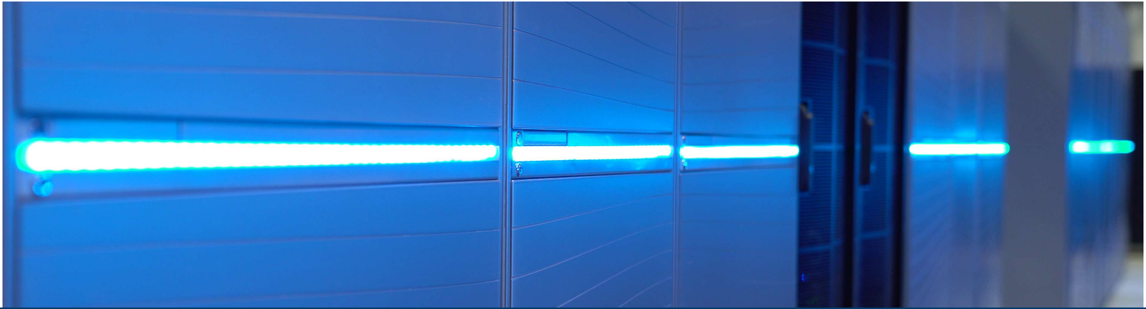
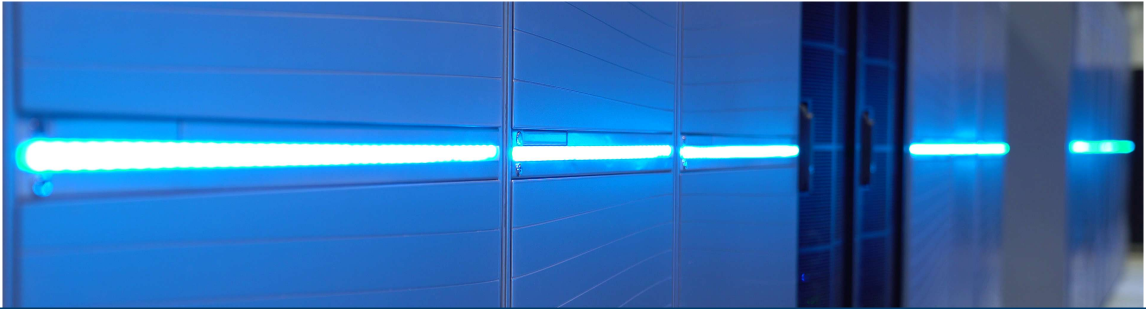


MPI ONBOARDING

November 4 2024 | Ilya Zhukov | Jülich Supercomputing Centre



Part I: First Steps with MPI

JÜLICH Forschungszentrum

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: 4.0 (June 2021)

[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`.

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

PROCESS ORGANIZATION [MPI-4.0, 7.2]

Process
An MPI program consists of autonomous processes, executing their own code, in an MIMD style.

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

Group
An ordered set of process identifiers

Context
A property that allows the partitioning of the communication space

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

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 foo.c -o foo
$ # Vendor specific wrapper for IBM's XL C compiler on BG/Q
$ bgxlc foo.c -o foo
```

Fortran Compiler Wrappers

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

However, neither the existence nor the interface of these wrappers is mandated by the standard.

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

LANGUAGE BINDINGS [MPI-4.0, 19, A]

C Language Bindings

C

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

Fortran Language Bindings

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

F08

```fortran
use mpi_f08
```

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

F90

```fortran
use mpi
```

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

F77

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

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()`.

JÜLICH Forschungszentrum

OTHER LANGUAGE BINDINGS

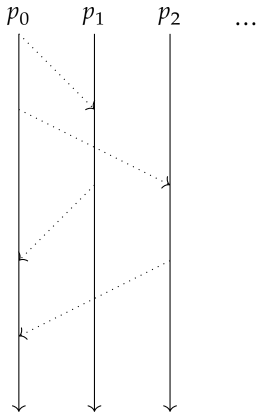
Besides the official bindings for C and Fortran mandated by the standard, unofficial bindings for other programming languages exist:

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.



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

JÜLICH Forschungszentrum

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

JÜLICH Forschungszentrum

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!
```


JÜLICH Forschungszentrum

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
```

Exception (can be used before initialization)

C

```c
int MPI_Initialized(int* flag)
```

F08

```fortran
MPI_Initialized(flag, ierror)
logical, intent(out) :: flag
integer, optional, intent(out) :: ierror
```

FINALIZATION [MPI-4.0, 11.2.2, 11.2.3]

Finalize MPI library when you are done using its functions

C
```
int MPI_Finalize(void)
```

F08
```
MPI_Finalize(ierror)
integer, optional, intent(out) :: ierror
```

Exception (can be used after finalization)

C
```
int MPI_Finalized(int *flag)
```

F08
```
MPI_Finalized(flag, ierror)
logical, intent(out) :: flag
integer, optional, intent(out) :: ierror
```

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
```

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
```

Examples

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

F08
```fortran
integer :: size
call MPI_Comm_size(MPI_COMM_WORLD, size)
```


JÜLICH Forschungszentrum

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
```

Examples

C

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

F08

```fortran
integer :: rank
call MPI_Comm_rank(MPI_COMM_WORLD, rank)
```

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

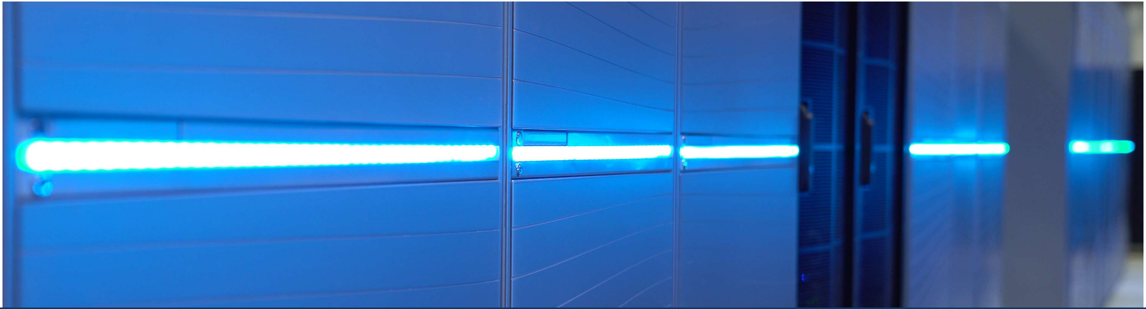
```
program example
  integer :: r, s
  call MPI_Comm_rank(..., r)
  call MPI_Comm_size(..., s)
  if (r == 0) then
    print *, "process", r, "of", s
  else
    print *, "process", r
  end if
  statement
end program
```

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

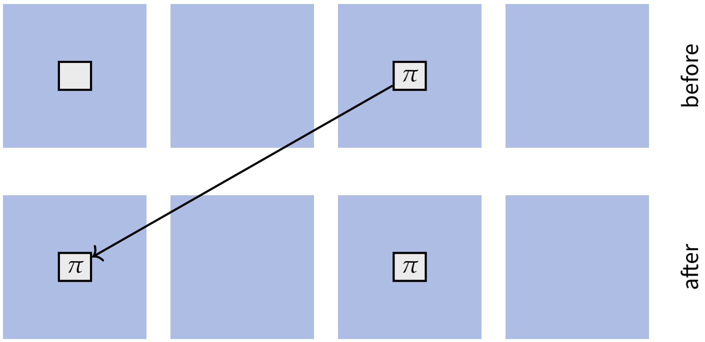
```
process 1
process 0 of 2
```



Part II: Blocking Point-to-Point Communication

JÜLICH
Forschungszentrum

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
If a procedure is nonblocking it will return as soon as possible. However, the user is not allowed to reuse resources specified in the call to the procedure before the communication has been completed using an appropriate completion procedure.

Examples:

- Blocking: Telephone call ✆
- Nonblocking: Email @

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
```


JÜLICH Forschungszentrum

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.

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`)

JÜLICH Forschungszentrum

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
```

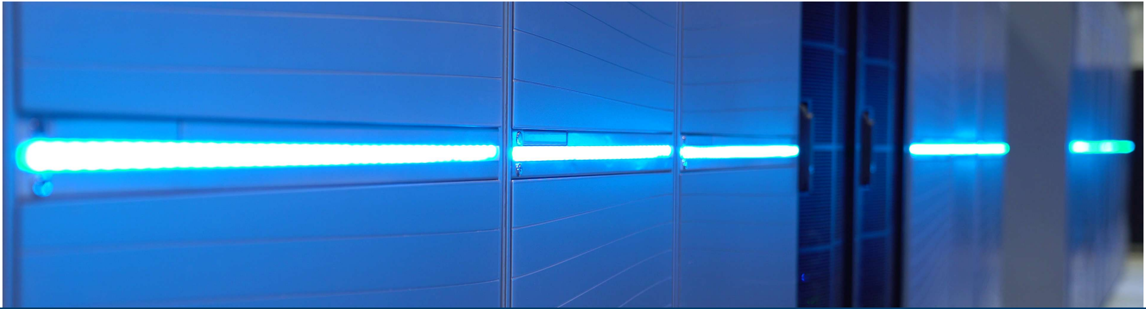
C

```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.



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
If a procedure is nonblocking it will return as soon as possible. However, the user is not allowed to reuse resources specified in the call to the procedure before the communication has been completed using an appropriate completion procedure.

Examples:

- Blocking: Telephone call ✆
- Nonblocking: Email @

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
- elide usage of buffers

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

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

F08

```fortran
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
```


JÜLICH Forschungszentrum

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

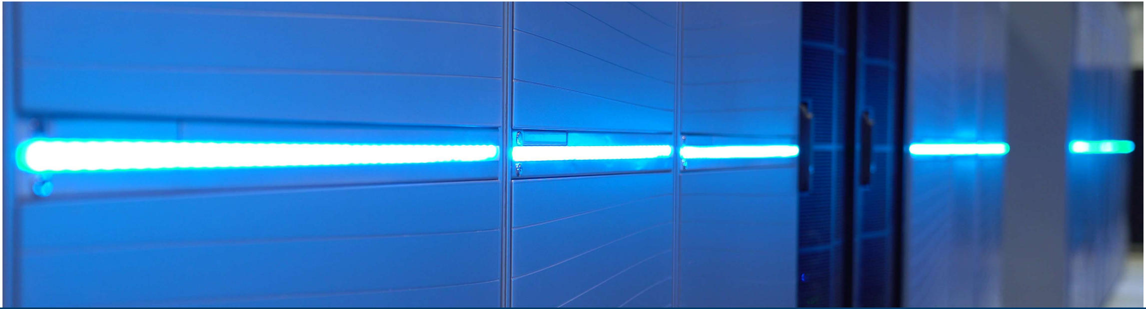
```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

F08

```
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



Part IV: Collective Communication

COLLECTIVE [MPI-4.0, 2.4, 6.1]

Collective

A procedure is collective if all processes in a group need to invoke the procedure.

- Collective communication implements certain communication patterns that involve all processes in a group
- Synchronization may or may not occur (except for `MPI_Barrier`)
- No tags are used
- No `MPI_Status` values are returned
- Receive buffer size must match the total amount of data sent (c.f. point-to-point communication where receive buffer capacity is allowed to exceed the message size)
- Point-to-point and collective communication do not interfere

CLASSIFICATION [MPI-4.0, 6.2.2]

One-to-all
MPI_Bcast, MPI_Scatter, MPI_Scatterv

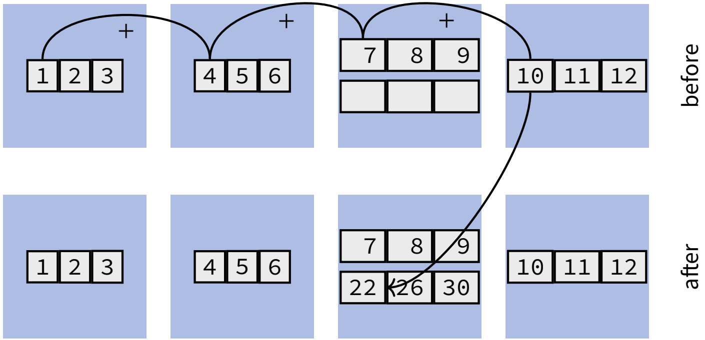
All-to-one
MPI_Gather, MPI_Gatherv, MPI_Reduce

All-to-all
MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw, MPI_Allreduce, MPI_Reduce_scatter, MPI_Barrier

Other
MPI_Scan, MPI_Exscan

REDUCE [MPI-4.0, 6.9.1]

Explanation



REDUCE [MPI-4.0, 6.9.1]

Signature

*

```
MPI_Reduce( <send buffer>, <receive buffer>, <operation>, <root> )
```

C

```
int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype
↪  datatype, MPI_Op op, int root, MPI_Comm comm)
```

F08

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
type(*), dimension(..), intent(in) :: sendbuf
type(*), dimension(..) :: recvbuf
integer, intent(in) :: count, root
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Op), intent(in) :: op
type(MPI_Comm), intent(in) :: comm
integer, optional, intent(out) :: ierror
```


JÜLICH Forschungszentrum

PREDEFINED OPERATIONS [MPI-4.0, 6.9.2]

Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and the first rank that holds it [MPI-4.0, 6.9.4]
MPI_MINLOC	Minimum and the first rank that holds it [MPI-4.0, 6.9.4]

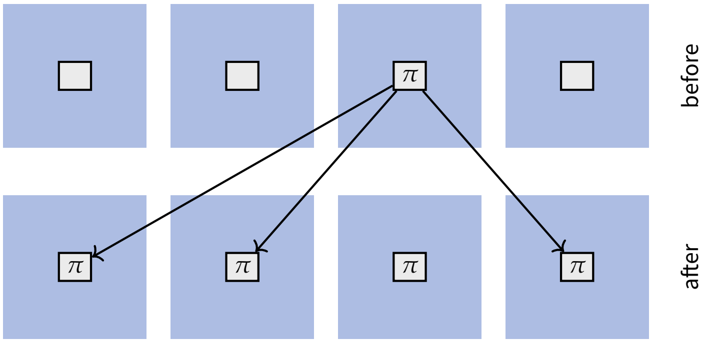
REDUCTION VARIANTS [MPI-4.0, 6.9 – 6.11]

Routines with extended or combined functionality:

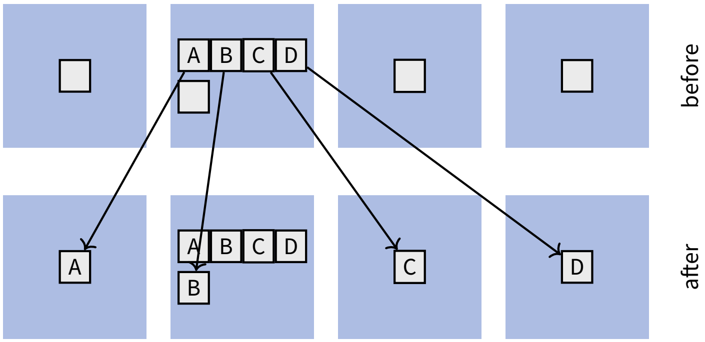
- `MPI_Allreduce`: perform a global reduction and copy the result onto all processes
- `MPI_Reduce_scatter`: perform a global reduction then copy different parts of the result onto all processes
- `MPI_Scan`: perform a global prefix reduction, include own data in result

JÜLICH Forschungszentrum

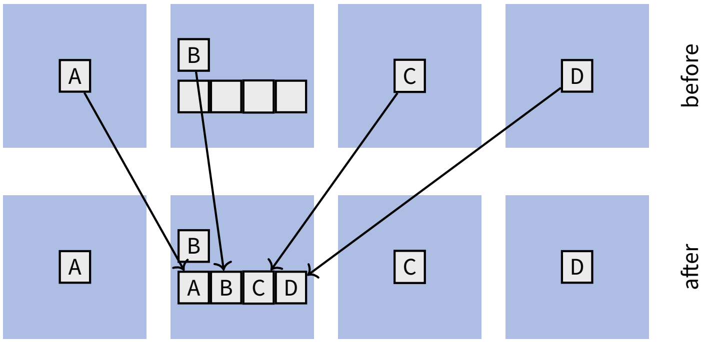
BROADCAST [MPI-4.0, 6.4]



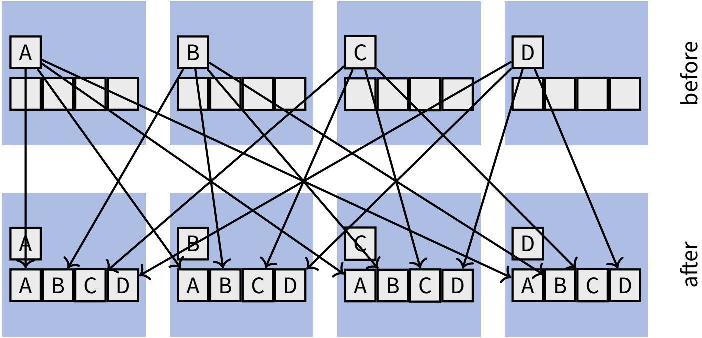
SCATTER [MPI-4.0, 6.6]



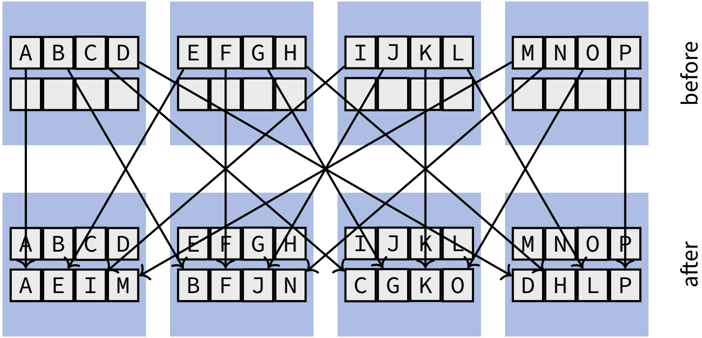
GATHER [MPI-4.0, 6.5]



GATHER-TO-ALL [MPI-4.0, 6.7]



ALL-TO-ALL SCATTER/GATHER [MPI-4.0, 6.8]



DATA MOVEMENT SIGNATURES

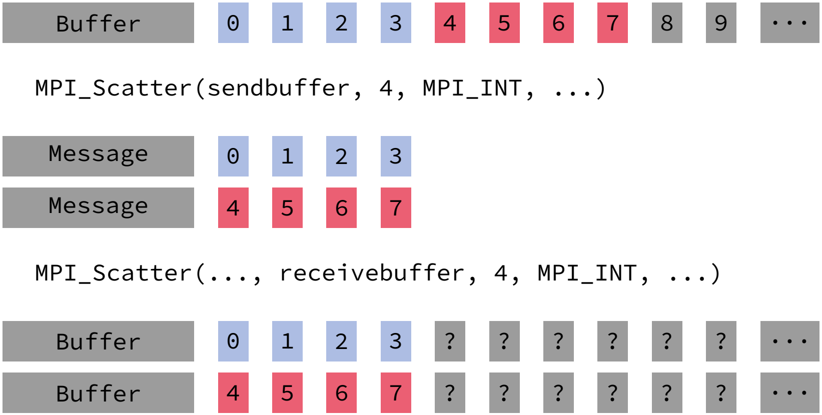
Single Message Size

```
MPI_Collective(<send buffer>, <receive buffer>, <root or communicator>)
```

- Both `send buffer` and `receive buffer` are `address`, `count`, `datatype`
- In One-to-all / All-to-one pattern
  - Specify `root` process by rank number
  - `send buffer` / `receive buffer` is only read / written on `root` process
- Buffers hold either one or n messages, where n is the number of processes
- If multiple messages are sent from / received into a buffer, associated `count` specifies the number of elements in a single message

MESSAGE ASSEMBLY

Single Message Size



DATA MOVEMENT VARIANTS [MPI-4.0, 6.5 – 6.8]

Routines with variable counts (and datatypes):

- `MPI_Scatterv`: scatter into parts of variable length
- `MPI_Gatherv`: gather parts of variable length
- `MPI_Allgatherv`: gather parts of variable length onto all processes
- `MPI_Alltoallv`: exchange parts of variable length between all processes
- `MPI_Alltoallw`: exchange parts of variable length and datatype between all processes

DATA MOVEMENT SIGNATURES

Varying Message Size

```
int MPI_Scatterv(const void *sendbuf, const int *sendcounts, const int
 ↪  *displs, MPI_Datatype sendtype, void *recvbuf, int recvcount,
 ↪  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Same high-level pattern as before.

In addition to `send/recvbuffer` following is specified:

- `send/recvcounts` array of length: number of MPI tasks that holds an individual `count` of number of message elements to be send
- `send/recvdispls` array of length: number of MPI tasks that holds the `displacements` (in units of message elements) from the beginning of the buffer at which to start taking elements

Note: Overlapping blocks
The blocks for different messages in send buffers can overlap. In receive buffers, they must not.

MESSAGE ASSEMBLY

Varying Message Size

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

```
MPI_Scatterv(sendbuffer, { 3, 2 }, { 1, 5 }, MPI_INT, ...)
```

Message	1	2	3
Message	5	6	

```
MPI_Scatterv(..., receivebuffer, (3 | 2), MPI_INT, ...)
```

Buffer	1	2	3	?	?	?	?	?	?	?	...
Buffer	5	6	?	?	?	?	?	?	?	?	...

BARRIER [MPI-4.0, 6.3]

C

```c
int MPI_Barrier(MPI_Comm comm)
```

F08

```fortran
MPI_Barrier(comm, ierror)
type(MPI_Comm), intent(in) :: comm
integer, optional, intent(out) :: ierror
```

Explicitly synchronizes all processes in the group of a communicator by blocking until all processes have entered the procedure.

JÜLICH Forschungszentrum

BARRIER CONTROL FLOW

Process 0

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

Process 1

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

BARRIER CONTROL FLOW

Process 0

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

Process 1

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

BARRIER CONTROL FLOW

Process 0

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

Process 1

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

BARRIER CONTROL FLOW

Process 0

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

Process 1

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

BARRIER CONTROL FLOW

Process 0

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

Process 1

```
program example
  statement1
  call MPI_Barrier(...)
  statement3
end program
```

EXERCISE 1

Exercise 1 – First Steps

1.1 Output of Ranks

Write a program `print_rank.{c|cxx|f90|py}` that has each process printing its rank.

```
I am process 0
I am process 1
I am process 2
```

Use: `MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`

1.2 Output of ranks and total number of processes

Write a program `print_rank_conditional.{c|cxx|f90|py}` in such a way that process 0 writes out the total number of processes

```
I am process 0 of 3
I am process 1
I am process 2
```

Use: `MPI_Comm_size`

EXERCISE 2

Exercise 2 – Collective Communication

2.1 Do it yourself

The template file `collectives.{c|F90|py}` is provided for you.
Write your own MPI parallel code with the following criteria:

- The MPI program should produce a sum of the rank of all processes.
- All processes should carry the summed value.
- The MPI program should only contain collective calls.
- All processes then prints the following message:
  `I am rank` *m*, I have obtained the sum of all rank=*i*.

There are multiple ways to achieve the end result. Experiment with different collective calls.