

INTRODUCTION TO PARALLEL PROGRAMMING WITH MPI AND OPENMP

March 19–22 2018 | Benedikt Steinbusch | Jülich Supercomputing Centre

TIMETABLE

	Day 1	Day 2	Day 3	Day 4
9:00 10:30	Fundamentals of Parallel Computing	Blocking Collective Communication	I/O	First Steps with OpenMP
	Coffee Break	Coffee Break	Coffee Break	Coffee Break
11:00 12:30	First Steps with MPI	Nonblocking Collective Comm.	I/O	Low-Level Constructs
	Lunch Break	Lunch Break	Lunch Break	Lunch Break
13:30 14:30	Blocking P2P Communication	Derived Datatypes	Tools <i>or</i> Communicators	Loop Worksharing
	Coffee Break	Coffee Break	Coffee Break	Coffee Break
15:00 16:30	Nonblocking P2P Communication	Derived Datatypes	Thread Compliance	Task Worksharing

Part I: Fundamentals of Parallel Computing

PARALLEL COMPUTING

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously.

Wikipedia¹

¹Wikipedia. *Parallel computing* — Wikipedia, The Free Encyclopedia. 2017. URL: https://en.wikipedia.org/w/index.php?title=Parallel_computing&oldid=787466585 (visited on 06/28/2017).

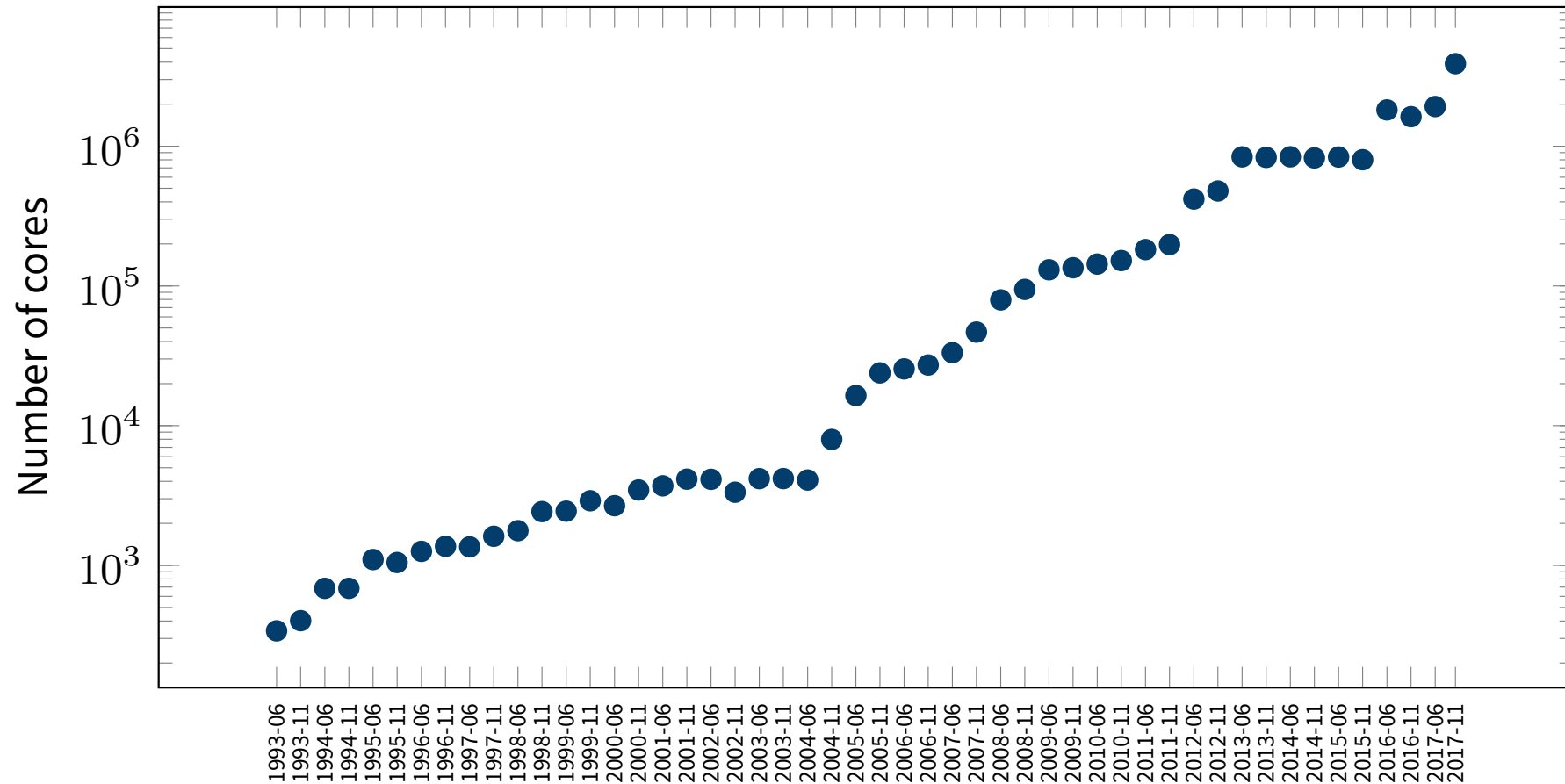
WHY AM I HERE?

The Way Forward

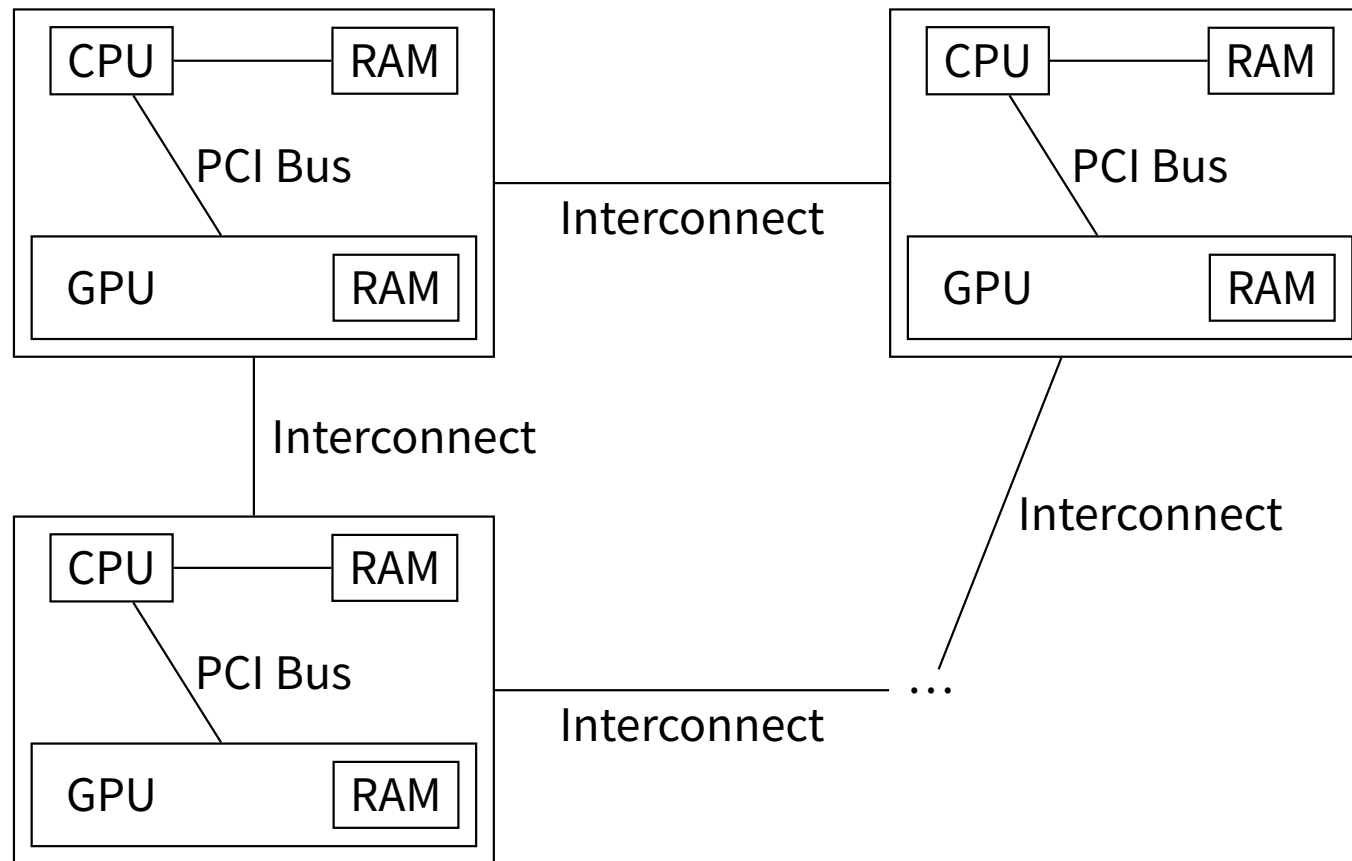
- Frequency scaling has stopped
- Performance increase through more parallel hardware
- Treating scientific problems
 - of larger scale
 - in higher accuracy
 - of a completely new kind

PARALLELISM IN THE TOP 500 LIST

Average Number of Cores of the Top 10 Systems



A MODERN SUPERCOMPUTER



JUQUEEN

Compute Nodes:

- 28 racks = 458 752 cores
- 64-bit PowerPC A2 1.6 GHz, 16 cores
- 16 GiB main memory per node (448 TiB total main memory)
- 5.87 PFLOP/s peak performance
- Lightweight proprietary compute node kernel (CNK), Linux

Login Nodes:

- 2 IBM Power7 nodes
- 8 cores at 3.55 GHz
- 128 GiB main memory each
- RedHat Linux v6.2

I/O Nodes:

- 248 I/O nodes
- 27 racks with 8 I/O nodes
- 1 rack with 32 I/O nodes

Network:

- 5 dimensional torus
- Collective and global Barrier/Interrupt network (part of 5D torus)
- 1 Gbit/s control network

File System:

- Global Parallel File System (GPFS)
- 16 PB online disk capacity
- 90 PB offline tape capacity

JURECA

Compute Nodes:

- 1872 compute nodes = 44 928 cores + 150 GPUs
- 2 Intel Haswell, 12 cores each, 2.5 GHz, SMT (HT)
- Memory (DDR4, 2133 MHz)
 - 1605 nodes with 128 GiB
 - 128 nodes with 256 GiB
 - 64 nodes with 512 GiB
- GPUs:
 - 75 nodes with 2 NVIDIA K80 each
 - K80: 4992 CUDA cores, 24 GiB GDDR5
- CentOS 7
- 2.24 PFLOP/s peak performance

Network:

- Mellanox EDR InfiniBand
- nonblocking fat tree topology
- 100 GiB/s to file system server

File System:

- Global Parallel File System (GPFS)
- 16 PB online disk capacity
- 90 PB offline disk capacity

JURECA

Booster Nodes:

- 1640 compute nodes = 111 520 cores
- 1 Intel Knights Landing, 68 cores each, 1.4 GHz, SMT (HT)
- 96 GiB memory plus 16 GiB of high-bandwidth MCDRAM each
- 5 PFLOP/s peak performance
- Intel Omni-Path Architecture high-speed network with non-blocking fat tree topology

Visualization Nodes:

- 12 nodes
- 2 Intel Haswell, 12 cores each, 2.5 GHz, SMT (HT)
- Memory (DDR4, 2133 MHz):
 - 10 nodes with 512 GiB
 - 2 nodes with 1024 GiB
- GPUs:
 - 2 NVIDIA K40 GPUs per node
 - K40: 12 GiB GDDR5
- CentOS 7

Login Nodes:

- Same hardware as compute nodes
- 256 GiB DDR4
- CentOS 7

PARALLEL COMPUTATIONAL UNITS

Implicit Parallelism

- Parallel execution of different (parts of) processor instructions
- Happens automatically
- Can only be influenced indirectly by the programmer

Multi-core / Multi-CPU

- Found in commodity hardware today
- Computational units share the same memory

Cluster

- Found in computing centers
- Independent systems linked via a (fast) interconnect
- Each system has its own memory

PARALLEL COMPUTATIONAL UNITS

Accelerators

- Strive to perform certain tasks faster than is possible on a general purpose CPU
- Make different trade-offs
- Often have their own memory
- Often not autonomous

Vector Processors / Vector Units

- Perform same operation on multiple pieces of data simultaneously
- Making a come-back as SIMD units in commodity CPUs (AVX-512) and GPGPU

MEMORY DOMAINS

Shared Memory

- All memory is directly accessible by the parallel computational units
- Single address space
- Programmer might have to synchronize access

Distributed Memory

- Memory is partitioned into parts which are private to the different computational units
- "Remote" parts of memory are accessed via an interconnect
- Access is usually nonuniform

PROCESSES & THREADS & TASKS

Abstractions for the independent execution of (part of) a program.

Process

Usually, multiple processes, each with their own associated set of resources (memory, file descriptors, etc.), can coexist

Thread

- Typically "smaller" than processes
- Often, multiple threads per one process
- Threads of the same process can share resources

Task

- Typically "smaller" than threads
- Often, multiple tasks per one thread
- Here: user-level construct

DISTRIBUTED STATE & MESSAGE PASSING

Distributed State

Program state is partitioned into parts which are private to the different processes.

Message Passing

- Parts of program state are transferred from one process to another for coordination
- Primitive operations are send and receive

MPI

- Implements a form of Distributed State and Message Passing
- (But also Shared State and Synchronization)

SHARED STATE & SYNCHRONIZATION

Shared State

The whole program state is directly accessible by the parallel threads.

Synchronization

- Threads can manipulate shared state using common loads and stores
- Establish agreement about progress of execution using synchronization primitives, e.g. barriers, critical sections, ...

OpenMP

- Implements Shared State and Synchronization
- (But also higher level constructs)

Part II: 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 Forum²

- 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: 3.1 (June 2015)

²Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard. Version 3.1.* June 4, 2015. URL: <http://mpi-forum.org/docs/mpi-3.1/mpi31-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, current version of the standard

COVERAGE

1. Introduction to MPI ✓
2. MPI Terms and Conventions ✓
3. Point-to-Point Communication ✓
4. Datatypes ✓
5. Collective Communication ✓
6. Groups, Contexts, Communicators and Caching
7. Process Topologies
8. MPI Environmental Management (✓)
9. The Info Object
10. Process Creation and Management
11. One-Sided Communications
12. External interfaces (✓)
13. I/O ✓
14. Tool Support
15. ...

LITERATURE & ACKNOWLEDGEMENTS

Literature

- Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 3.1. June 4, 2015. URL: <http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>
- 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
- <http://www.mpi-forum.org>

Acknowledgements

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

PROCESS ORGANIZATION [MPI-3.1, 6.2]

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

OBJECTS [MPI-3.1, 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.

COMPILING & LINKING [MPI-3.1, 17.1.7]

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

C Compiler Wrappers

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

Fortran Compiler Wrappers

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

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

PROCESS STARTUP [MPI-3.1, 8.8]

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

```
$ mpiexec -n <numprocs> <program> # startup mechanism suggested by the standard  
$ srun -n <numprocs> <program>    # Slurm startup mechanism as found on Jureca
```

LANGUAGE BINDINGS [MPI-3.1, 17, A]

C Language Bindings

```
⌋ #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'
```

FORTRAN HINTS [MPI-3.1, 17.1.2 – 17.1.4]

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 `error` argument is mandatory instead of **optional**
- Further details can be found in [MPI-3.1, 17.1]

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

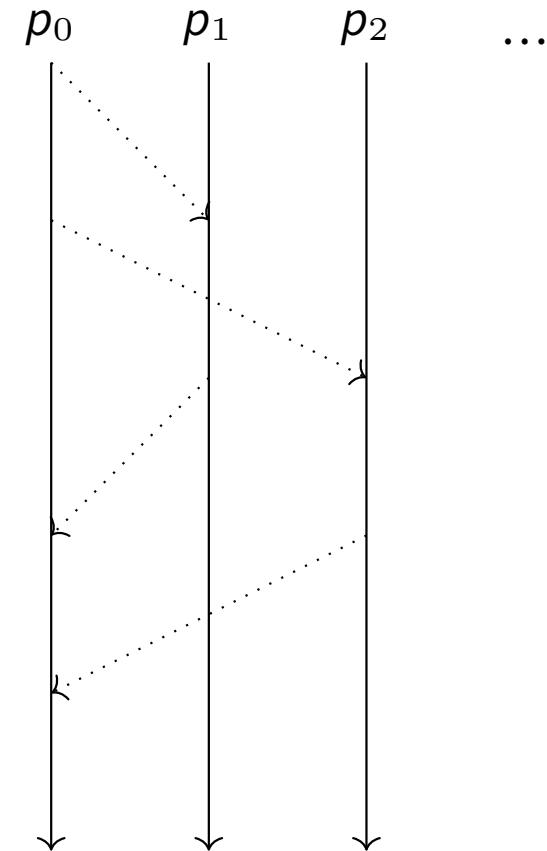
.NET MPI.NET

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 4).



INITIALIZATION [MPI-3.1, 8.7]

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

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

```
FO8 MPI_Init(ierr)  
integer, optional, intent(out) :: ierr
```

Exception (can be used before initialization)

```
⌋ int MPI_Initialized(int* flag)
```

```
FO8 MPI_Initialized(flag, ierr)  
logical, intent(out) :: flag  
integer, optional, intent(out) :: ierr
```

FINALIZATION [MPI-3.1, 8.7]

Finalize MPI library when you are done using its functions

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

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

Exception (can be used after finalization)

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

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

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.

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

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

COMMUNICATOR SIZE [MPI-3.1, 6.4.1]

Determine the total number of processes in a communicator

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

```
FO8 MPI_Comm_size(comm, size, ierror)  
type(MPI_Comm), intent(in) :: comm  
integer, intent(out) :: size  
integer, optional, intent(out) :: ierror
```

Examples

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

```
FO8 integer :: size  
call MPI_Comm_size(MPI_COMM_WORLD, size)
```

PROCESS RANK [MPI-3.1, 6.4.1]

Determine the rank of the calling process within a communicator

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

```
FO8 MPI_Comm_rank(comm, rank, ierror)  
type(MPI_Comm), intent(in) :: comm  
integer, intent(out) :: rank  
integer, optional, intent(out) :: ierror
```

Examples

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

```
FO8 integer :: rank  
call MPI_Comm_rank(MPI_COMM_WORLD, rank)
```

ERROR HANDLING [MPI-3.1, 8.3, 8.4, 8.5]

- Flexible error handling through error handlers which can be attached to
 - Communicators
 - Files
 - Windows (not part of this course)

- 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

LOGIN & PROGRAMMING ENVIRONMENT

PC Login Credentials

See paper slips

Compilers

C

```
$ mpicc
```

Fortran

```
$ mpif90
```

C++

```
$ mpicxx
```

JURECA Login

1. Open a terminal on your PC

```
$ ssh jureca
```

2. Enter passphrase, *see paper slip*

3. Load modules

```
$ module load Intel IntelMPI
```

Course Material

```
/homea/hpclab/trainXXX/mpi-omp/  
|--C/  
|--Fortran/  
|--C++/  
|--mpi-omp-article.pdf  
\\--mpi-omp-dhandout.pdf
```

RUNNING PARALLEL PROGRAMS ON JURECA

Interactive Mode

1. Start an interactive session

```
$ salloc --reservation=mpi-omp --nodes=1  
↪ --time=08:00:00
```

2. Wait for the prompt...

3. Load compiler and MPI

```
$ module load Intel IntelMPI
```

4. Start applications with n processes

```
$ srun --ntasks=<n> <application>
```

Batch Mode

1. To start an application with n processes, submit the following job script with

```
sbatch --reservation=mpi-omp <script>
```

```
#!/bin/bash  
#SBATCH --nodes=1  
#SBATCH --ntasks=<n>  
#SBATCH --ntasks-per-node=<n>  
#SBATCH --output=output.%j  
#SBATCH --error=error.%j  
#SBATCH --time=00:05:00  
#SBATCH --partition=batch  
module load Intel IntelMPI  
srun <application>
```

2. Output/error messages will be in output.<jobid> and error.<jobid>

USEFUL COMMANDS ON JURECA

Command	Description
<code>queue -u <user-id></code>	Shows the status of jobs
<code>scancel <job-id></code>	Aborts the job with ID <job-id>
<code>scontrol show job <job-id></code>	Show detailed information about a pending, running or recently completed job
<code>watch <command></code>	Executes <command> every 2 seconds and shows the output
<code>top</code>	Shows all processes running on the current node with continuous updates
<code><command> &</code>	Execute <command> in the background

EXERCISES

Exercise 1 – First Steps

1.1 Output of Ranks

Write a program in C/C++ or Fortran that has each process printing its rank.

```
I have rank 0  
I have rank 1  
I have rank 2  
I have rank 3
```

1.2 Output of ranks and total number of processes

Extend the program in such a way that process 0 writes out the total number of processes

```
I have rank 0 and am master of 4 processes!  
I have rank 1  
I have rank 2  
I have rank 3
```

1.3 Conditional output

Modify the program in such a way that only processes with odd ranks print

```
I have rank 1  
I have rank 3
```

Part III: Blocking 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

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-3.1, 3.2.1]

* MPI_Send(<buffer>, <destination>)

⌋ **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-3.1, 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-3.1, 3.2.2, 3.3, 4.1]

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

Fortran type

MPI basic data type

signed int

MPI_INT

integer

MPI_INTEGER

float

MPI_FLOAT

real

MPI_REAL

char

MPI_CHAR

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-3.1, 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-3.1, 4.2]

RECEIVING MESSAGES [MPI-3.1, 3.2.4]

```
* MPI_Recv( <buffer>, <source> ) -> <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  
F08 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-3.1, 3.2.5]

Contains information about received messages

```
MPI_Status status;  
status.MPI_SOURCE  
status.MPI_TAG  
↳ status.MPI_ERROR
```

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

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

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

Pass MPI_STATUS_IGNORE to MPI_Recv if not interested.

PROBE [MPI-3.1, 3.8.1]

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

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

```
F08 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)

SEND MODES [MPI-3.1, 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
- Might enable more efficient communication

RECEIVE MODES [MPI-3.1, 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-3.1, 3.5]

Order

In singlethreaded programs, messages are nonovertaking. 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.

PITFALLS

Deadlock

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

<code>MPI_Ssend(..., 1, ...)</code>	<code>MPI_Ssend(..., 0, ...)</code>
<code>⌋ MPI_Recv(..., 1, ...)</code>	<code>MPI_Recv(..., 0, ...)</code>

Mitigation Strategies

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

EXERCISES

Exercise 2 – Blocking P2P communication	2.1 Ping-Pong
	• Process 0 sends a small message (ping) and receives an answer (pong), first to/from process 1, then process 2,.... • Processes $1, \dots, n - 1$ receive a message from process 0 then send back an answer. • If the program is not run on at least two processes, it should issue an error message and abort (<code>MPI_Abort</code> [MPI-3.1, 8.7]).
	2.2 Ping-Pong – <code>MPI_Sendrecv</code>
	Use <code>MPI_Sendrecv</code> [MPI-3.1, 3.10] instead of <code>MPI_Send</code> and <code>MPI_Recv</code> .
	2.3 Ping-Pong – Benchmarking
	• For each pair, perform 10^6 iterations of the ping-pong communication. • Measure the duration of the loop with <code>MPI_Wtime</code> [MPI-3.1, 8.6]. • Process 0 should print the results.
	Bonus
	Benchmark the links between all unique pairs of processes.

Part IV: Nonblocking Point-to-Point Communication

RATIONALE [MPI-3.1, 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

INITIATION ROUTINES [MPI-3.1, 3.7.2]

Send	
Synchronous MPI_Issend	Buffered MPI_Ibsend
Standard MPI_Isend	Ready MPI_Irsend

Receive	Probe
MPI_Irecv	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-3.1, 3.7.2]

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

```
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-3.1, 3.7.2]

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

```
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-3.1, 3.8.1]

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

↪ **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-3.1, 3.7.3]

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

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

```
FO8 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-3.1, 3.7.3]

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

↪ **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

FREE [MPI-3.1, 3.7.3]

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

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

```
FO8 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-3.1, 3.8.4]

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

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

```
FO8 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)

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_Isend is enforced at completion (wait or test)
 - Asserted readiness of MPI_Ir send 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.` [\[MPI-3.1, 17.1.12\]](#)
- Arrays with vector subscripts must not be used as buffers [\[MPI-3.1, 17.1.13\]](#)
- 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.`) [\[MPI-3.1, 17.1.16–17.1.20\]](#)

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

EXERCISES

Exercise 3 – Nonblocking P2P Communication

3.1 Global Summation – Sequential

In the file `3.{c | c++ | f90}` implement the function/subroutine `global_sum_sequential(x, y, root, comm)`. It will be called by all processes on the communicator `comm` and on each one accepts an integer `x`. It should compute the global sum of all values of `x` across all processes and return the result in `y` only on the process with rank `root`.

Use the following strategy:

1. On all processes, initialize the partial result for the sum to the local value of `x`.
2. Starting at the process with rank `root`, send the partial result on to the process with the next higher rank (wrap around to rank 0 on the process with the highest rank).
3. Receive the partial sum from the process with the next lower rank (or from the process with the highest rank on process 0) and add it to the local partial result.
4. The `root` process will eventually have the global result which it will return in `y`.

The file contains a small `main()` function / **program** that can be used to test whether your implementation works.

EXERCISES

Exercise 3 – Nonblocking P2P Communication

3.2 Global Summation – Tree

In the same file, implement a function/subroutine `global_sum_tree(x, y, root, comm)` that performs the same operation.

Use the following strategy:

1. On all processes, initialize the partial result for the sum to the local value of `x`.
2. Now proceed in rounds until only a single process remains:
 1. Group processes into pairs – let us call them the left and the right process.
 2. The right process sends its partial result to the left process.
 3. The left process receives the partial result and adds it to its own one.
 4. The left process continues on into the next round, the right one does not.
3. The process that made it to the last round now has the global result which it sends to the process with rank `root`.
4. The `root` process receives the global result and returns it in `y`.

Modify the `main()` function / **program** so that the new function/subroutine `global_sum_tree()` is also tested and check your implementation.

Part V: Blocking Collective Communication

COLLECTIVE [MPI-3.1, 2.4, 5.1]

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-3.1, 5.2.2]

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

BARRIER SYNCHRONIZATION [MPI-3.1, 5.3]

```
* MPI_Barrier( <comm> )
```

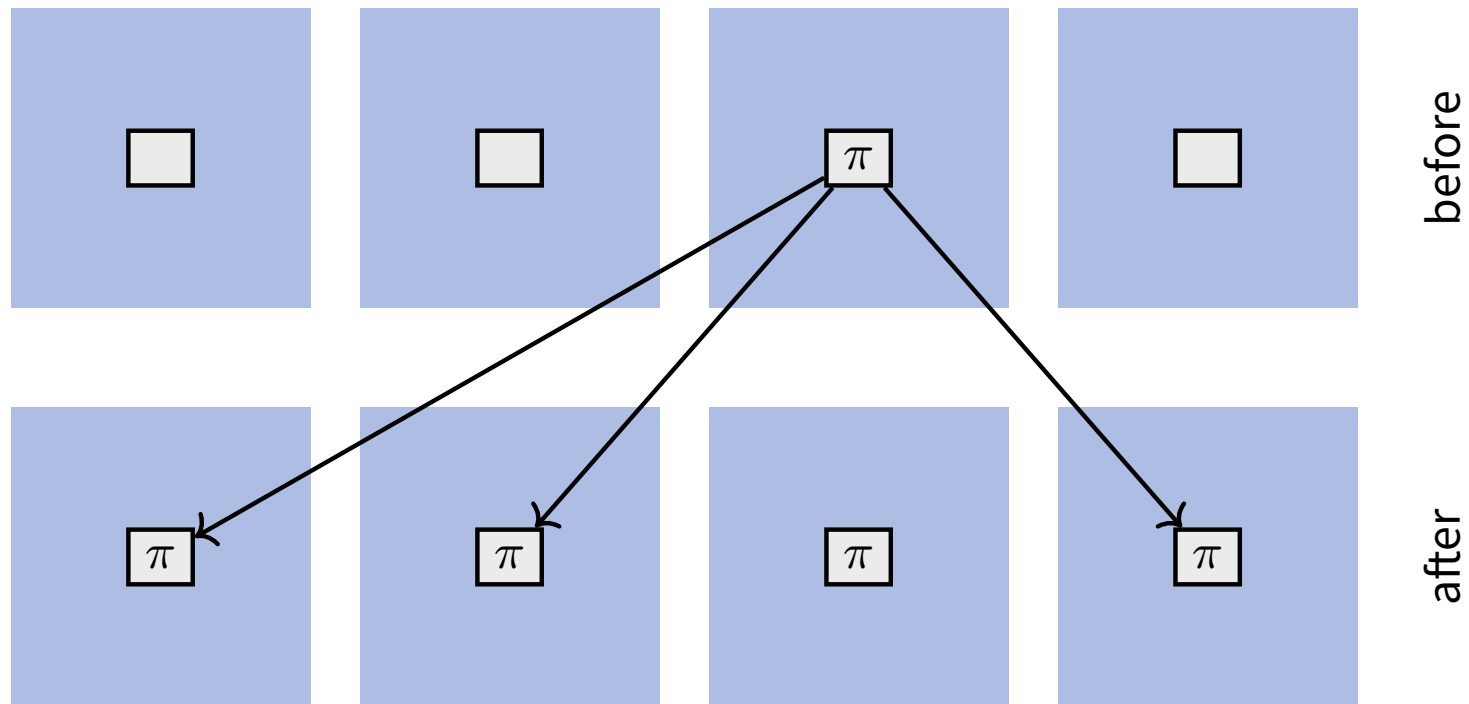
```
⌋ int MPI_Barrier(MPI_Comm comm)
```

```
FO8 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.

BROADCAST [MPI-3.1, 5.4]

Explanation



BROADCAST [MPI-3.1, 5.4]

Signature

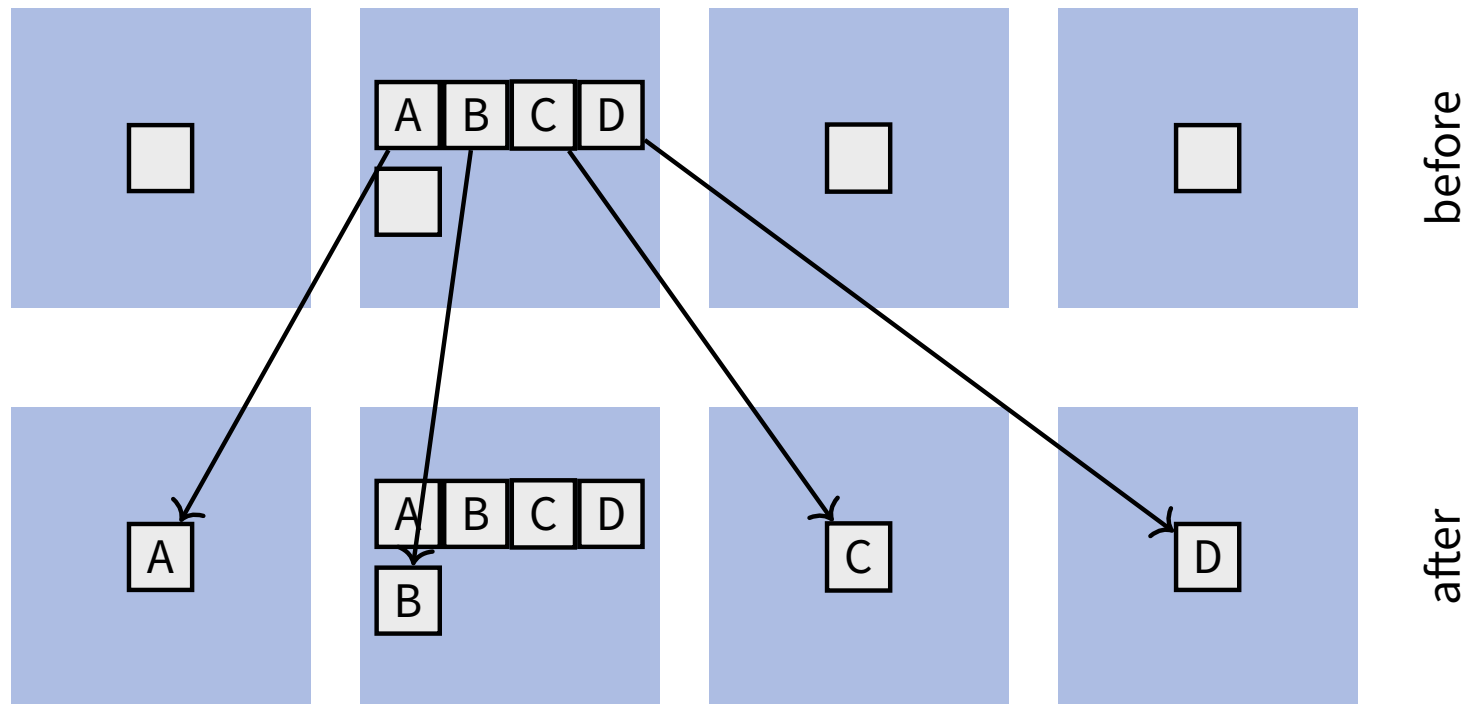
* MPI_Bcast(<buffer>, <root>)

↪ **int** MPI_Bcast(**void*** buffer, **int** count, MPI_Datatype
↪ datatype, **int** root, MPI_Comm comm)

F08 MPI_Bcast(buffer, count, datatype, root, comm, ierror)
type(*), **dimension**(..) :: buffer
integer, **intent**(in) :: count, root
type(MPI_Datatype), **intent**(in) :: datatype
type(MPI_Comm), **intent**(in) :: comm
integer, **optional**, **intent**(out) :: ierror

SCATTER [MPI-3.1, 5.6]

Explanation



If `MPI_IN_PLACE` is used for `recvbuf` on the root process, `recvcount` and `recvtype` are ignored and the root process does not send data to itself

SCATTER [MPI-3.1, 5.6]

Signature

```
* MPI_Scatter( <send buffer>, <receive buffer>, <root> )
```

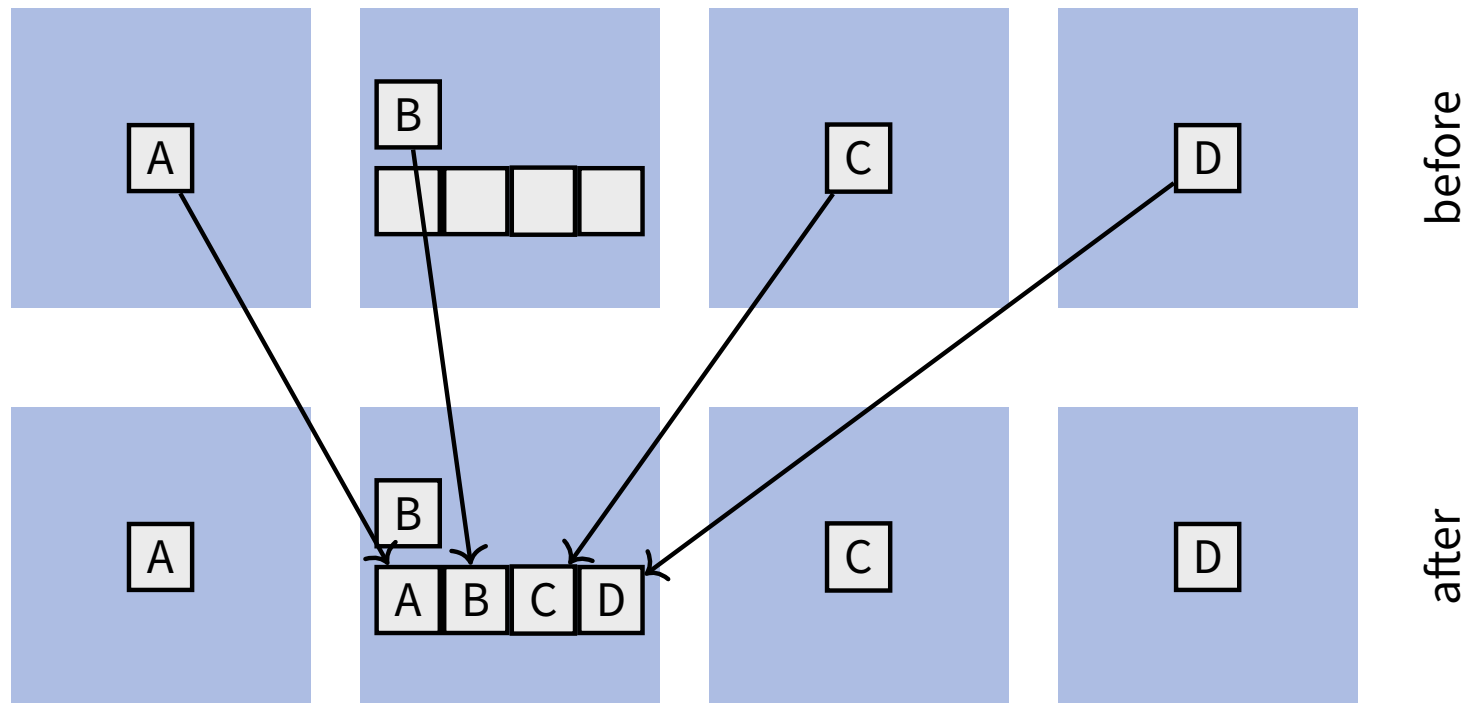
```
int MPI_Scatter(const void* sendbuf, int sendcount,  
    ↪ MPI_Datatype sendtype, void* recvbuf, int recvcount,  
    ↪ MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,  
    ↪ recvcount, recvtype, root, comm, ierror)  
type(*), dimension(..), intent(in) :: sendbuf  
type(*), dimension(..) :: recvbuf  
integer, intent(in) :: sendcount, recvcount, root  
type(MPI_Datatype), intent(in) :: sendtype, recvtype  
type(MPI_Comm), intent(in) :: comm  
integer, optional, intent(out) :: ierror
```

FO8

GATHER [MPI-3.1, 5.5]

Explanation



If `MPI_IN_PLACE` is used for `sendbuf` on the root process, `sendcount` and `sendtype` are ignored on the root process and the root process will not send data to itself.

GATHER [MPI-3.1, 5.5]

Signature

```
* MPI_Gather( <send buffer>, <receive buffer>, <root> )
```

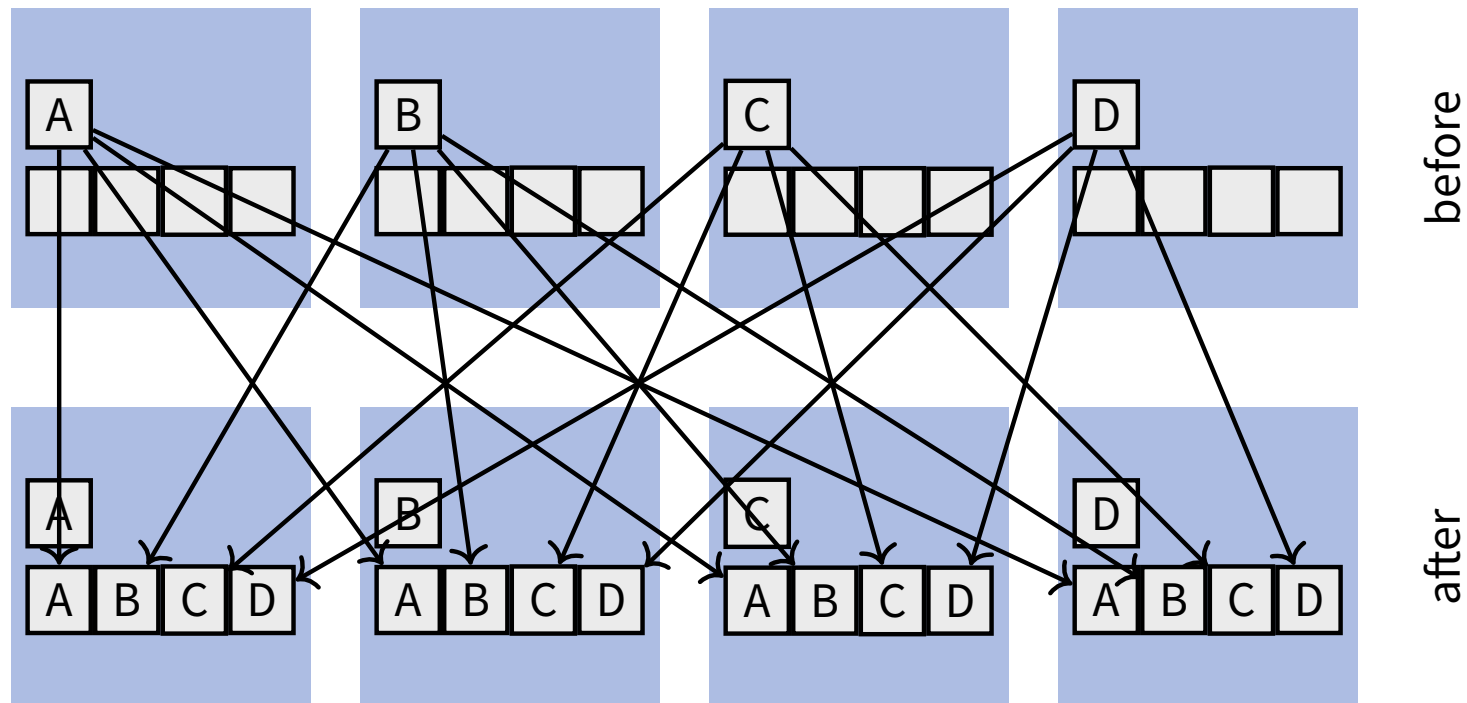
```
int MPI_Gather(const void* sendbuf, int sendcount,  
  ↪ MPI_Datatype sendtype, void* recvbuf, int recvcount,  
  ↪ MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,  
  ↪ recvcount, recvtype, root, comm, ierror)  
type(*), dimension(..), intent(in) :: sendbuf  
type(*), dimension(..) :: recvbuf  
integer, intent(in) :: sendcount, recvcount, root  
type(MPI_Datatype), intent(in) :: sendtype, recvtype  
type(MPI_Comm), intent(in) :: comm  
integer, optional, intent(out) :: ierror
```

FO8

GATHER-TO-ALL [MPI-3.1, 5.7]

Explanation



If `MPI_IN_PLACE` is used for `sendbuf` on all processes, `sendcount` and `sendtype` are ignored and the input data is assumed to already be in the correct position in `recvbuf`.

GATHER-TO-ALL [MPI-3.1, 5.7]

Signature

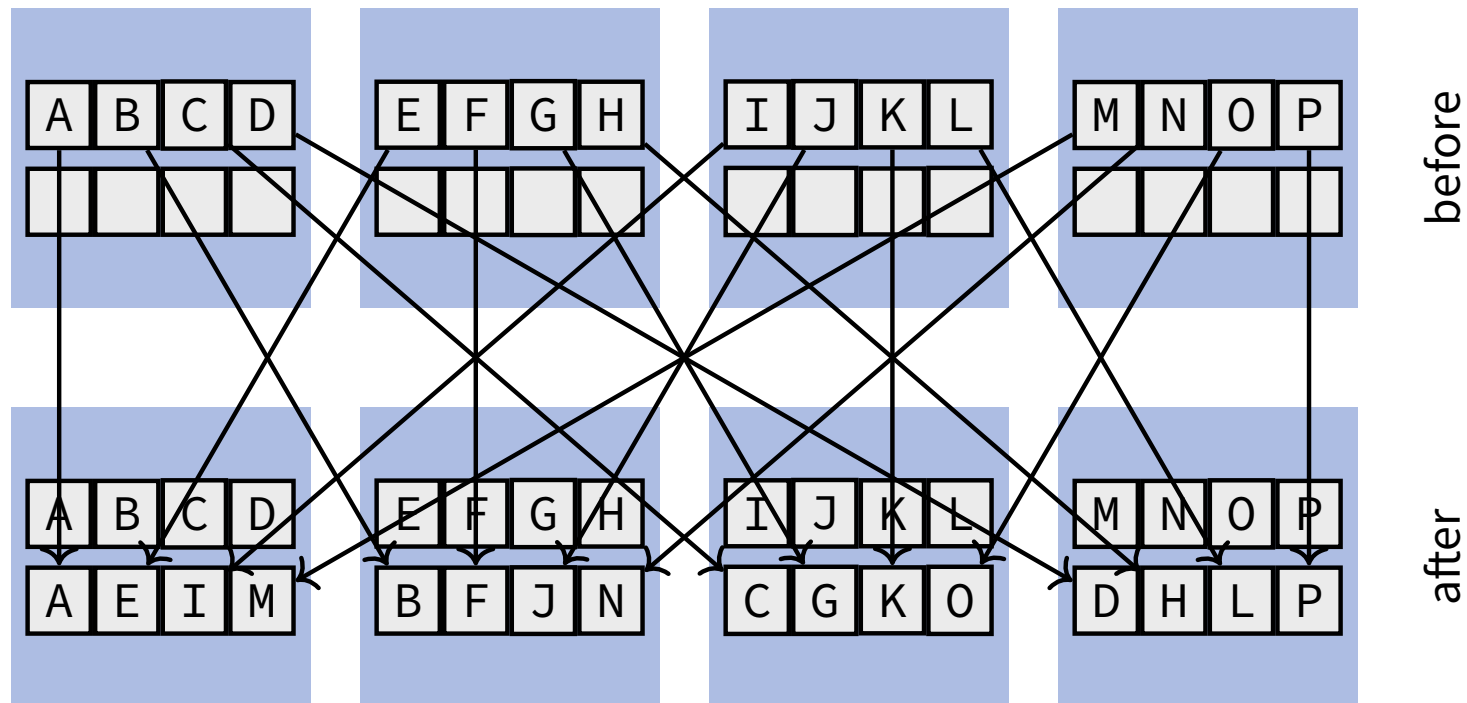
```
MPI_Allgather( <send buffer>, <receive buffer>,  
* ↪ <communicator> )
```

```
int MPI_Allgather(const void* sendbuf, int sendcount,  
↪ MPI_Datatype sendtype, void* recvbuf, int recvcount,  
↪ MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,  
↪ recvcount, recvtype, comm, ierror)  
type(*), dimension(..), intent(in) :: sendbuf  
type(*), dimension(..) :: recvbuf  
integer, intent(in) :: sendcount, recvcount  
type(MPI_Datatype), intent(in) :: sendtype, recvtype  
type(MPI_Comm), intent(in) :: comm  
F08 integer, optional, intent(out) :: ierror
```

ALL-TO-ALL SCATTER/GATHER [MPI-3.1, 5.8]

Explanation



If `MPI_IN_PLACE` is used for `sendbuf` on all processes, `sendcount` and `sendtype` are ignored and the input data is assumed to already be in the correct position in `recvbuf`.

ALL-TO-ALL SCATTER/GATHER [MPI-3.1, 5.8]

Signature

```
MPI_Alltoall( <send buffer>, <receive buffer>,  
*    ↪ <communicator> )
```

```
int MPI_Alltoall(const void* sendbuf, int sendcount,  
    ↪ MPI_Datatype sendtype, void* recvbuf, int recvcount,  
    ↪ MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf,  
    ↪ recvcount, recvtype, comm, ierror)  
type(*), dimension(..), intent(in) :: sendbuf  
type(*), dimension(..) :: recvbuf  
integer, intent(in) :: sendcount, recvcount  
type(MPI_Datatype), intent(in) :: sendtype, recvtype  
type(MPI_Comm), intent(in) :: comm  
F08 integer, optional, intent(out) :: ierror
```

GLOBAL REDUCTION OPERATIONS [MPI-3.1, 5.9]

Associative operations over distributed data

$d_0 \oplus d_1 \oplus d_2 \oplus \dots \oplus d_{n-1}$, where

d_i , data of process with rank i

$\oplus$, associative operation

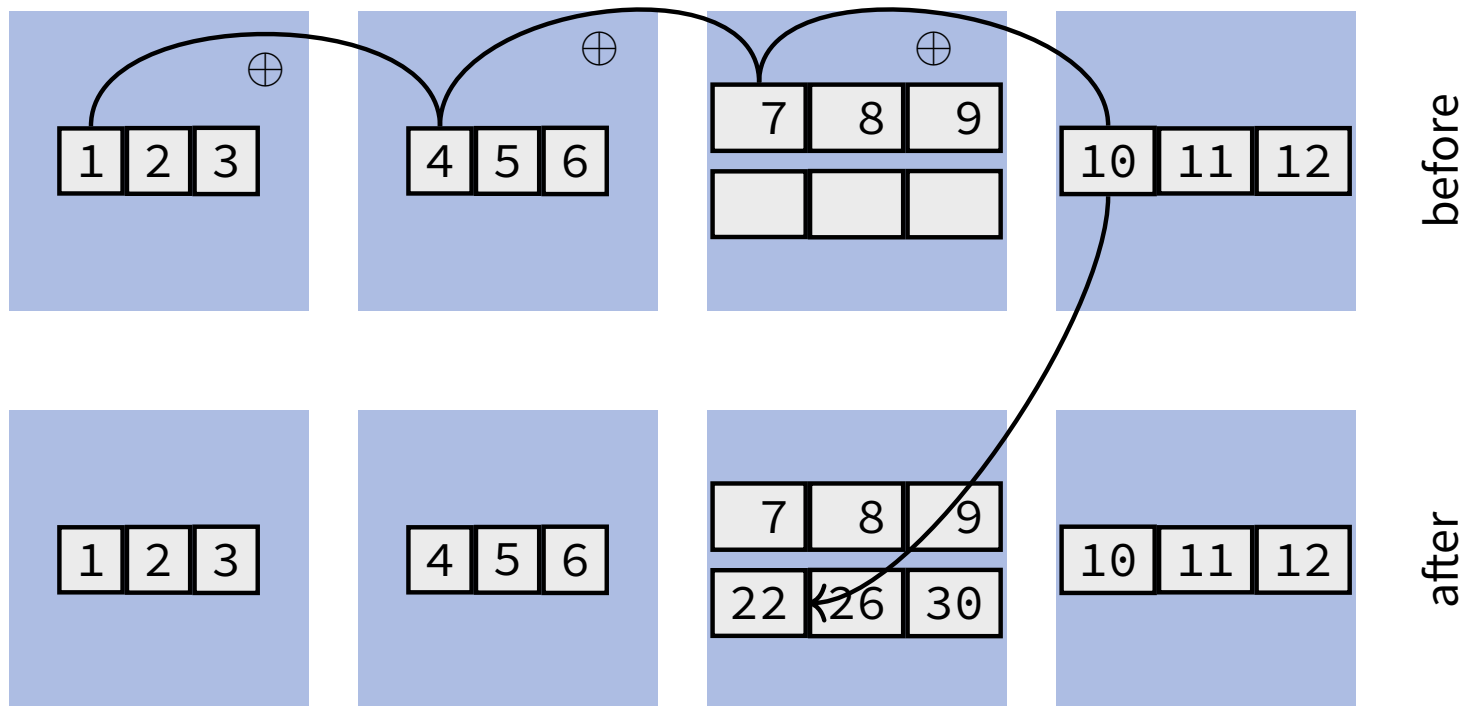
Examples for $\oplus$:

- Sum $+$ and product $\cdot$
- Maximum $\max$ and minimum $\min$
- User-defined operations

Order of application is not defined, watch out for floating point rounding.

REDUCE [MPI-3.1, 5.9.1]

Explanation



If `MPI_IN_PLACE` is used for `sendbuf` on the root process, the input data for the root process is taken from `recvbuf`.

REDUCE [MPI-3.1, 5.9.1]

Signature

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

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

```
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  
F08 type(MPI_Comm), intent(in) :: comm
```

PREDEFINED OPERATIONS [MPI-3.1, 5.9.2]

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-3.1, 5.9.4]
MPI_MINLOC	Minimum and the first rank that holds it [MPI-3.1, 5.9.4]

VARIANTS [MPI-3.1, 5.5 – 5.11]

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

Routines with extended or combined functionality:

- `MPI_Allreduce`: perform a global reduction and replicate the result onto all ranks
- `MPI_Reduce_scatter`: perform a global reduction then scatter the result onto all ranks
- `MPI_Scan`: perform a global prefix reduction, include own data in result
- `MPI_Exscan`: perform a global prefix reduction, exclude own data from result

Part VI: Nonblocking Collective Communication

PROPERTIES

Properties similar to nonblocking point-to-point communication

1. Initiate communication

- Routine names: `MPI_I . . .` (I for immediate)
- Nonblocking routines return before the operation has completed.
- Nonblocking routines have the same arguments as their blocking counterparts plus an extra request argument.

2. User-application proceeds with something else

3. Complete operation

- Same completion routines (`MPI_Test`, `MPI_Wait`, ...)

Nonblocking collective operations cannot be matched with blocking collective operations.

Nonblocking Barrier

Barrier is entered through `MPI_Ibarrrier` (which returns immediately). Completion (e.g. `MPI_Wait`) blocks until all processes have entered.

NONBLOCKING BROADCAST [MPI-3.1, 5.12.2]

Blocking operation

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype  
↳ datatype, int root, MPI_Comm comm)
```

Nonblocking operation

```
int MPI_Ibcast(void* buffer, int count, MPI_Datatype  
↳ datatype, int root, MPI_Comm comm, MPI_Request*  
↳ request)
```

NONBLOCKING BROADCAST [MPI-3.1, 5.12.2]

F08

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
type(*), dimension(..) :: buffer
integer, intent(in) :: count, root
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Comm), intent(in) :: comm
integer, optional, intent(out) :: ierror
```

F08

```
MPI_Ibcast(buffer, count, datatype, root, comm, request,
↪ ierror)
type(*), dimension(..), asynchronous :: buffer
integer, intent(in) :: count, root
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Comm), intent(in) :: comm
type(MPI_Request), intent(out) :: request
integer, optional, intent(out) :: ierror
```

4.1 Monte Carlo Calculation of π

Write a program that calculates π using the Monte Carlo method. The program should print the number of processes used, the difference to the exact value of π (can be computed as $\pi = 2 \arccos(0)$) and the time needed for the calculation in seconds.

Hints:

- Inscribe a circle in a square.
- Sample a number of points n_{total} from a random uniform distribution inside the square.
- Count the number of points that lie inside the circle n_{circle} .
- For a large number of samples, the ratio between the two numbers should be roughly equal to the ratio between the area of the two shapes

$$n_{\text{circle}}/n_{\text{total}} \simeq A_{\text{circle}}/A_{\text{square}}.$$

Some geometry:

$$A_{\text{circle}} = \pi r^2$$

$$A_{\text{square}} = 4r^2$$

$$\pi = 4 \frac{A_{\text{circle}}}{A_{\text{square}}}$$

Random numbers:

```
#include <stdlib.h>
C int rand(void);
```

```
F08 real :: x
call random_number(x)
```

4.2 Redistribution of Points with Collectives

Write a program that redistributes point data among the processes. Proceed as follows:

1. Every process draws `npoints` random numbers – the points – from a uniform random distribution on $[0, 1)$.
2. Partition $[0, 1)$ among the `nranks` processes: process i gets partition $[i/nranks, (i + 1)/nranks)$.
3. Redistribute the points, so that every process is left with only those particles that lie inside its partition.

Guidelines:

- Use collectives, either `MPI_Gather` and `MPI_Scatter` or `MPI_Alltoall(v)` (see next slide)
- It helps to partition the points so that consecutive blocks can be sent to other processes
- `MPI_Alltoall` can be used to distribute the information that is needed to call `MPI_Alltoallv`
- Dynamic memory management could be necessary
- Check that all points lie in the desired partition after redistribution
- Check that no points spontaneously vanish or emerge

ALL-TO-ALL WITH VARYING COUNTS

```
MPI_Alltoallv(const void* sendbuf, const int sendcounts[],  
    ↪ const int sdispls[], MPI_Datatype sendtype, void*  
    ↪ recvbuf, const int recvcunts[], const int rdispls[],  
    ↪ MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype,  
    ↪ recvbuf, recvcunts, rdispls, recvtype, comm, ierror)  
type(*), dimension(..), intent(in) :: sendbuf  
type(*), dimension(..) :: recvbuf  
integer, intent(in) :: sendcounts(*), sdispls(*),  
    ↪ recvcunts(*), rdispls(*)  
type(MPI_Datatype), intent(in) :: sendtype, recvtype  
type(MPI_Comm), intent(in) :: comm  
integer, optional, intent(out) :: ierror
```

F08

Part VII: Derived Datatypes

MOTIVATION [MPI-3.1, 4.1]

Reminder: Buffer

- Message buffers are defined by a triple (address, count, datatype).
- Basic data types restrict buffers to homogeneous, contiguous sequences of values in memory.

Scenario A

Problem: Want to communicate data describing particles that consists of a position (3 `double`) and a particle species (encoded as an `int`).

Solution(?): Communicate positions and species in two separate operations.

Scenario B

Problem: Have an array `real :: a(:)`, want to communicate only every second entry `a(1:n:2)`.

Solution(?): Copy data to a temporary array.

Derived datatypes are a mechanism for describing arrangements of data in buffers. Gives the MPI library the opportunity to employ the optimal solution.

TYPE MAP & TYPE SIGNATURE [MPI-3.1, 4.1]

Type map

A *general datatype* is described by its *type map*, a sequence of pairs of *basic datatype* and *displacement*:

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$

Type signature

A *type signature* describes the contents of a message read from a buffer with a *general datatype*:

$$Typesig = \{type_0, \dots, type_{n-1}\}$$

Type matching is done based on *type signatures* alone.

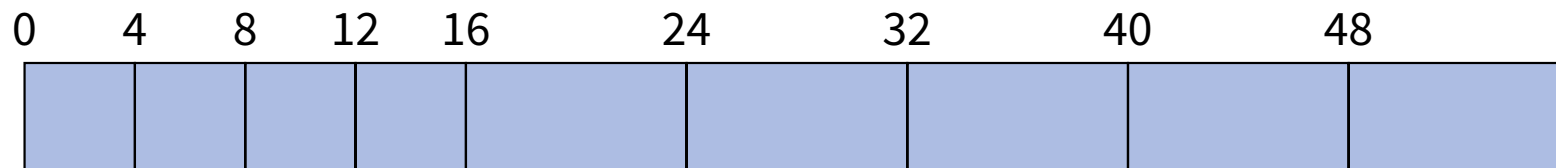
EXAMPLE

```
struct heterogeneous {  
    int i[4];  
    double d[5];  
};
```

```
type heterogeneous  
    integer :: i(4)  
    double precision :: d(5)  
end type
```

Basic Datatype (C)

0	MPI_INT
4	MPI_INT
8	MPI_INT
12	MPI_INT
16	MPI_DOUBLE
24	MPI_DOUBLE
32	MPI_DOUBLE
40	MPI_DOUBLE
48	MPI_DOUBLE



EXAMPLE

```
struct heterogeneous {  
    int i[4];  
    double d[5];  
}
```

```
type heterogeneous  
    integer :: i(4)  
    double precision :: d(5)  
end type
```

Basic Datatype (Fortran)

0	MPI_INTEGER
4	MPI_INTEGER
8	MPI_INTEGER
12	MPI_INTEGER
16	MPI_DOUBLE_PRECISION
24	MPI_DOUBLE_PRECISION
32	MPI_DOUBLE_PRECISION
40	MPI_DOUBLE_PRECISION
48	MPI_DOUBLE_PRECISION



TYPE CONSTRUCTORS [MPI-3.1, 4.1]

A new derived type is constructed from an existing type `oldtype` (basic or derived) using type constructors. In order of increasing generality/complexity:

1. `MPI_Type_contiguous` n consecutive instances of `oldtype`
2. `MPI_Type_vector` n blocks of m instances of `oldtype` with stride s
3. `MPI_Type_indexed` n blocks of m_i instances of `oldtype` with displacement d_i for each $i = 1, \dots, n$
4. `MPI_Type_indexed_block` n blocks of m instances of `oldtype` with displacement d_i for each $i = 1, \dots, n$
5. `MPI_Type_create_struct` n blocks of m_i instances of `oldtypei` with displacement d_i for each $i = 1, \dots, n$
6. `MPI_Type_create_subarray` n dimensional subarray out of an array with elements of type `oldtype`
7. `MPI_Type_create_darray` distributed array with elements of type `oldtype`

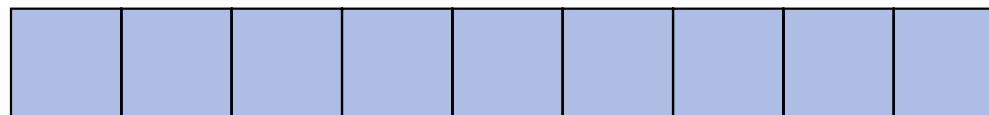
CONTIGUOUS DATA [MPI-3.1, 4.1.2]

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,  
↳ MPI_Datatype* newtype)
```

```
MPI_Type_contiguous(count, oldtype, newtype, ierror)  
integer, intent(in) :: count  
type(MPI_Datatype), intent(in) :: oldtype  
type(MPI_Datatype), intent(out) :: newtype  
F08 integer, optional, intent(out) :: ierror
```

- Simple concatenation of `oldtype`
- Results in the same access pattern as using `oldtype` and specifying a buffer with count greater than one.

oldtype



count = 9

STRUCT DATA [MPI-3.1, 4.1.2]

```
int MPI_Type_create_struct(int count, const int  
    ↪ array_of_blocklengths[], const MPI_Aint  
    ↪ array_of_displacements[], const MPI_Datatype  
    ↪ array_of_types[], MPI_Datatype* newtype)
```

```
MPI_Type_create_struct(count, array_of_blocklengths,  
    ↪ array_of_displacements, array_of_types, newtype,  
    ↪ ierror)  
integer, intent(in) :: count,  
    ↪ array_of_blocklengths(count)  
integer(kind=MPI_ADDRESS_KIND), intent(in) ::  
    ↪ array_of_displacements(count)  
type(MPI_Datatype), intent(in) :: array_of_types(count)  
type(MPI_Datatype), intent(out) :: newtype  
integer, optional, intent(out) :: ierror
```

F08

EXAMPLE

```
struct heterogeneous {  
    int i[4];  
    double d[5];  
}
```

```
count = 2;  
array_of_blocklengths[0] = 4;  
array_of_displacements[0] = 0;  
array_of_types[0] = MPI_INT;  
array_of_blocklengths[1] = 5;  
array_of_displacements[1] = 16;  
array_of_types[1] = MPI_DOUBLE;
```



EXAMPLE

```
type heterogeneous  
  integer :: i(4)  
  double precision :: d(5)  
end type
```

```
count = 2;  
array_of_blocklengths(1) = 4  
array_of_displacements(1) = 0  
array_of_types(1) = MPI_INTEGER  
array_of_blocklengths(2) = 5  
array_of_displacements(2) = 16  
FO8 array_of_types(2) = MPI_DOUBLE_PRECISION
```



SUBARRAY DATA [MPI-3.1, 4.1.3]

```
int MPI_Type_create_subarray(int ndims, const int
    ↪ array_of_sizes[], const int array_of_subsizes[],
    ↪ const int array_of_starts[], int order, MPI_Datatype
    ↪ oldtype, MPI_Datatype* newtype)
```

```
MPI_Type_create_subarray(ndims, array_of_sizes,
    ↪ array_of_subsizes, array_of_starts, order, oldtype,
    ↪ newtype, ierror)
integer, intent(in) :: ndims, array_of_sizes(ndims),
    ↪ array_of_subsizes(ndims), array_of_starts(ndims),
    ↪ order
type(MPI_Datatype), intent(in) :: oldtype
type(MPI_Datatype), intent(out) :: newtype
integer, optional, intent(out) :: ierror
```

F08

EXAMPLE

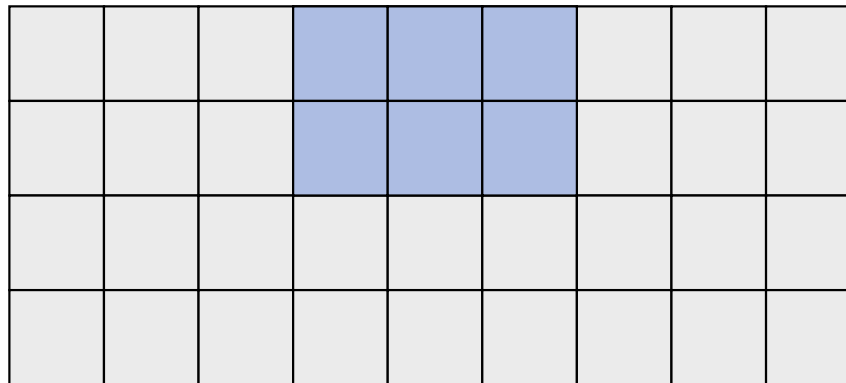
```
ndims = 2;  
array_of_sizes[] = { 4, 9 };  
array_of_subsizes[] = { 2, 3 };  
array_of_starts[] = { 0, 3 };  
order = MPI_ORDER_C;  
⌋ oldtype = MPI_INT;
```

An array with global size 4×9 containing a subarray of size 2×3 at offsets 0, 3:

EXAMPLE

```
ndims = 2  
array_of_sizes(:) = (/ 4, 9 /)  
array_of_subsizes(:) = (/ 2, 3 /)  
array_of_starts(:) = (/ 0, 3 /)  
order = MPI_ORDER_FORTRAN  
oldtype = MPI_INTEGER
```

An array with global size 4×9 containing a subarray of size 2×3 at offsets 0, 3:



COMMIT & FREE [MPI-3.1, 4.1.9]

Before using a derived datatype in communication it needs to be committed

```
⌋ int MPI_Type_commit(MPI_Datatype* datatype)
```

```
FO8 MPI_Type_commit(datatype, ierror)  
type(MPI_Datatype), intent(inout) :: datatype  
integer, optional, intent(out) :: ierror
```

Marking derived datatypes for deallocation

```
⌋ int MPI_Type_free(MPI_Datatype *datatype)
```

```
FO8 MPI_Type_free(datatype, ierror)  
type(MPI_Datatype), intent(inout) :: datatype  
integer, optional, intent(out) :: ierror
```

ALIGNMENT & PADDING

```
struct heterogeneous {  
    int i[3];  
    double d[5];  
}
```

```
count = 2;  
array_of_blocklengths[0] = 3;  
array_of_displacements[0] = 0;  
array_of_types[0] = MPI_INT;  
array_of_blocklengths[1] = 5;  
array_of_displacements[1] = 16;  
array_of_types[1] = MPI_DOUBLE;
```



ALIGNMENT & PADDING

```
type heterogeneous  
  integer :: i(3)  
  double precision :: d(5)  
end type
```

```
count = 2;  
array_of_blocklengths(1) = 3  
array_of_displacements(1) = 0  
array_of_types(1) = MPI_INTEGER  
array_of_blocklengths(2) = 5  
array_of_displacements(2) = 16  
F08 array_of_types(2) = MPI_DOUBLE_PRECISION
```



ADDRESS CALCULATION [MPI-3.1, 4.1.5]

Displacements are calculated as the difference between the addresses at the start of a buffer and at a particular piece of data in the buffer. The address of a location in memory is found using:

```
int MPI_Get_address(const void* location, MPI_Aint*  
↪ address)
```

```
MPI_Get_address(location, address, ierror)  
type(*), dimension(..), asynchronous :: location  
integer(kind=MPI_ADDRESS_KIND), intent(out) :: address  
FOR integer, optional, intent(out) :: ierror
```

Using the C operator & to determine addresses is discouraged, since it returns a pointer which is not necessarily the same as an address.

ADDRESS ARITHMETIC [MPI-3.1, 4.1.5]

Addition

```
⌋ MPI_Aint MPI_Aint_add(MPI_Aint a, MPI_Aint b)
```

```
FO8 integer(kind=MPI_ADDRESS_KIND) MPI_Aint_add(a, b)  
FO8 integer(kind=MPI_ADDRESS_KIND), intent(in) :: a, b
```

Subtraction

```
⌋ MPI_Aint MPI_Aint_diff(MPI_Aint a, MPI_Aint b)
```

```
FO8 integer(kind=MPI_ADDRESS_KIND) MPI_Aint_diff(a, b)  
FO8 integer(kind=MPI_ADDRESS_KIND), intent(in) :: a, b
```

EXAMPLE

```
struct heterogeneous h;  
MPI_Aint base, displ[2];  
MPI_Datatype newtype;  
MPI_Datatype types[2] = { MPI_INT, MPI_DOUBLE };  
int blocklen[2] = { 3, 5 };  
  
MPI_Get_address(&h, &base);  
MPI_Get_address(&h.i, &displ[0]);  
displ[0] = MPI_Aint_diff(displ[0], base);  
MPI_Get_address(&h.d, &displ[1]);  
displ[1] = MPI_Aint_diff(displ[1], base);  
MPI_Type_create_struct(2, blocklen, displ, types,  
    ↪ &newtype);  
⌋ MPI_Type_commit(&newtype);
```

EXAMPLE

```
type(heterogeneous) :: h
integer(kind=MPI_ADDRESS_KIND) :: base, displ(2)
type(MPI_Datatype) :: types(2), newtype
integer :: blocklen(2)

types = (/ MPI_INTEGER, MPI_DOUBLE_PRECISION /)
blocklen = (/ 3, 5 /)

call MPI_Get_address(h, base)
call MPI_Get_address(h%i, displ(1))
displ(1) = MPI_Aint_diff(displ(1), base)
call MPI_Get_address(h%d, displ(2))
displ(2) = MPI_Aint_diff(displ(2), base)
call MPI_Type_create_struct(2, blocklen, displ, types,
    ↪ newtype)
call MPI_Type_commit(newtype)
```

F08

TYPE EXTENT [MPI-3.1, 4.1]

Extent

The *extent* of a type is determined from its *lower bounds* and *upper bounds*:

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$

$$lb(Typemap) = \min_j disp_j$$

$$ub(Typemap) = \max_j (disp_j + sizeof(type_j)) + \epsilon$$

$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

Extent and spacing

Let t be a type with type map $\{(MPI_CHAR, 1)\}$ and b an array of **char**, $b = \{ 'a', 'b', 'c', 'd', 'e', 'f' \}$, then $MPI_Send(b, 3, t, \dots)$ will result in a message $\{ 'b', 'c', 'd' \}$ and not $\{ 'b', 'd', 'f' \}$.

Explicit padding can be added by *resizing* the type.

RESIZE [MPI-3.1, 4.1.7]

```
int MPI_Type_create_resized(MPI_Datatype oldtype,  
↪ MPI_Aint lb, MPI_Aint extent, MPI_Datatype* newtype)
```

```
MPI_Type_create_resized(oldtype, lb, extent, newtype,  
↪ ierror)  
integer(kind=MPI_ADDRESS_KIND), intent(in) :: lb, extent  
type(MPI_Datatype), intent(in) :: oldtype  
type(MPI_Datatype), intent(out) :: newtype  
FOR integer, optional, intent(out) :: ierror
```

Creates a new derived type `newtype` with the same type map as `oldtype` but explicit lower bound `lb` and explicit upper bound `lb + extent`.

Extent and true extent of a type can be queried using `MPI_Type_get_extent` and `MPI_Type_get_true_extent`. The size of resulting messages can be queried with `MPI_Type_size`.

EXERCISES

Exercise 5 – Derived Datatypes

5.1 Structs

Each point in a 3 dimensional lattice has the following properties:

- 3 color values (r, g, b, $\in [0, 255]$)
- 3 coordinates (x, y, z, double precision)
- 1 tag (1 character)

Write a program where process 0 initializes a point with meaningful values and broadcasts it to all other processes using a derived datatype. Print the received point on any process but 0.

Modification Change the order of the components of the point structure. Does your program still produce correct results?

EXERCISES

Exercise 5 – Derived Datatypes

5.2 Matrix Access – Diagonal

In the file `5_2.{c | c++ | f90}` implement the function/subroutine `get_diagonal` that extracts the elements on the diagonal of an $N \times N$ matrix into a vector:

$$\text{vector}_i = \text{matrix}_{i,i}, \quad i = 1 \dots N.$$

Do not access the elements of either the matrix or the vector directly. Rather, use MPI datatypes for accessing your data.

Hint

- `MPI_Sendrecv` on the `MPI_COMM_SELF` communicator can be used for copying the data.
- Use `MPI_Type_vector` to express the access pattern

EXERCISES

Exercise 5 – Derived Datatypes

5.3 Matrix Access – Upper Triangle

In the file `5_2.{c | c++ | f90}` implement the function/subroutine `get_upper` that copies all elements on or above the diagonal of an $N \times N$ matrix to a second matrix and leaves all other elements untouched.

$$\text{upper}_{i,j} = \text{matrix}_{i,j}, \quad i = 1 \dots N, j = i \dots N$$

As in the previous exercise, do not access the matrix elements directly. Make sure to un-comment the call to `test_get_upper()` to have your solution tested.

Hint

- `MPI_Sendrecv` on the `MPI_COMM_SELF` communicator can be used for copying the data.
- Use `MPI_Type_indexed` to express the access pattern

Part VIII: Input/Output

MOTIVATION

I/O on HPC Systems

- "This is not your parents' I/O subsystem"
- File system is a shared resource
 - Modification of metadata might happen sequentially
 - File system blocks might be shared among processes
- File system access might not be uniform across all processes
- Interoperability of data originating on different platforms

MPI I/O

- MPI already defines a language that describes data layout and movement
- Extend this language by I/O capabilities
- More expressive/precise API than POSIX I/O affords better chances for optimization

COMMON I/O STRATEGIES

Funnelled I/O

- + Simple to implement
- I/O bandwidth is limited to the rate of this single process
- Additional communication might be necessary
- Other processes may idle and waste resources during I/O operations

All or several processes use one file

- + Number of files is independent of number of processes
- + File is in canonical representation (no post-processing)
- Uncoordinated client requests might induce time penalties
- File layout may induce false sharing of file system blocks

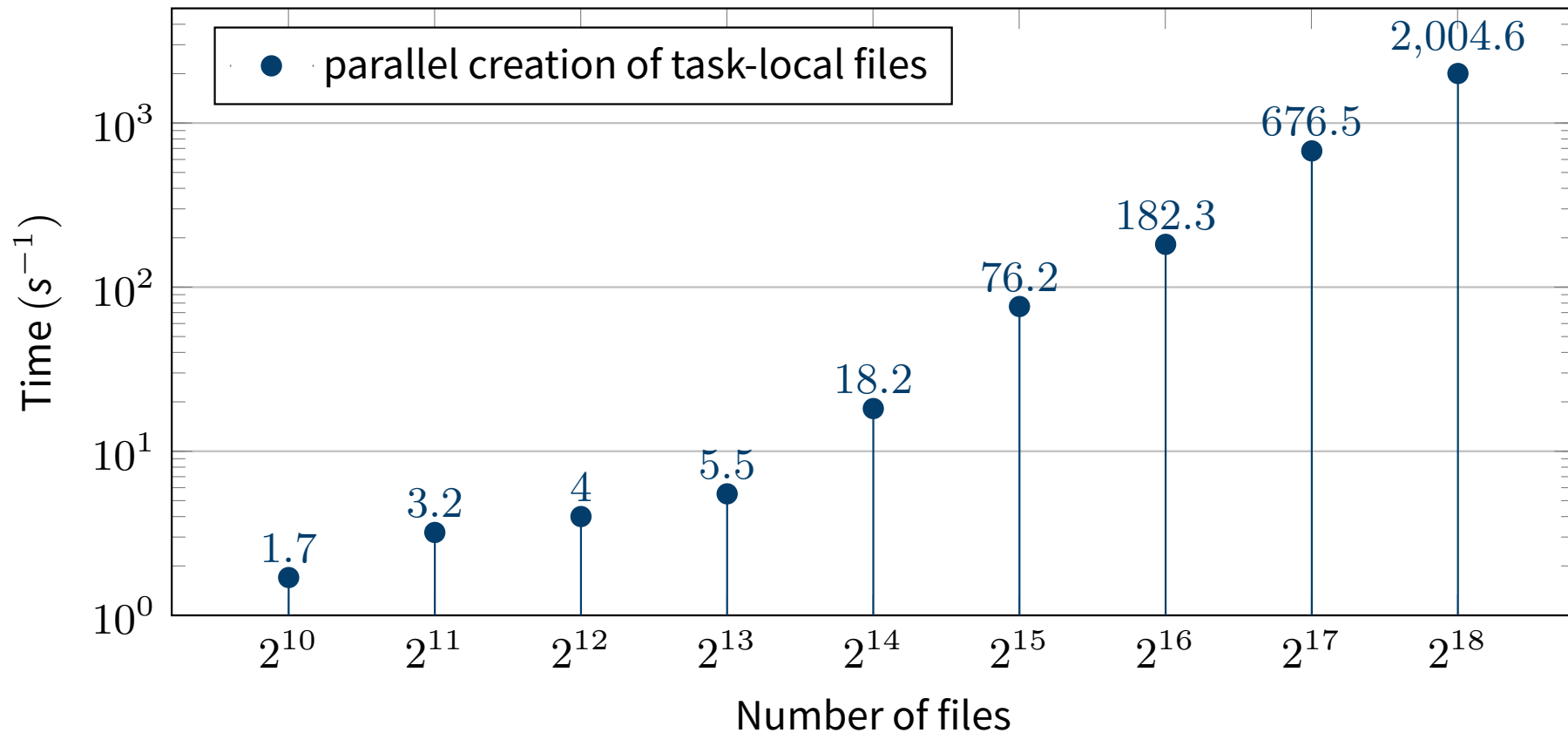
COMMON I/O STRATEGIES

Task-Local Files

- + Simple to implement
- + No explicit coordination between processes needed
- + No false sharing of file system blocks
- Number of files quickly becomes unmanageable
- Files often need to be merged to create a canonical dataset (post-processing)
- File system might introduce implicit coordination (metadata modification)

SEQUENTIAL ACCESS TO METADATA

Jugene, IBM Blue Gene/P, GPFS, filesystem /work using fopen()



FILE, FILE POINTER & HANDLE [MPI-3.1, 13.1]

File

An MPI file is an ordered collection of typed data items.

File Pointer

A file pointer is an implicit offset into a file maintained by MPI.

File Handle

An opaque MPI object. All operations on an open file reference the file through the file handle.

OPENING A FILE [MPI-3.1, 13.2.1]

```
int MPI_File_open(MPI_Comm comm, const char* filename,  
↪ int amode, MPI_Info info, MPI_File* fh)
```

```
MPI_File_open(comm, filename, amode, info, fh, ierror)  
type(MPI_Comm), intent(in) :: comm  
character(len=*), intent(in) :: filename  
integer, intent(in) :: amode  
type(MPI_Info), intent(in) :: info  
type(MPI_File), intent(out) :: fh  
F08 integer, optional, intent(out) :: ierror
```

- Collective operation on communicator comm
- Filename must reference the same file on all processes
- Process-local files can be opened using MPI_COMM_SELF
- info object specifies additional information (MPI_INFO_NULL for empty)

ACCESS MODE [MPI-3.1, 13.2.1]

amode denotes the access mode of the file and must be the same on all processes. It *must* contain exactly one of the following:

MPI_MODE_RDONLY read only access
MPI_MODE_RDWR read and write access
MPI_MODE_WRONLY write only access

and may contain some of the following:

MPI_MODE_CREATE create the file if it does not exist
MPI_MODE_EXCL error if creating file that already exists
MPI_MODE_DELETE_ON_CLOSE delete file on close
MPI_MODE_UNIQUE_OPEN file is not opened elsewhere
MPI_MODE_SEQUENTIAL access to the file is sequential
MPI_MODE_APPEND file pointers are set to the end of the file

Combine using bit-wise or (| operator in C, i or intrinsic in Fortran).

CLOSING A FILE [MPI-3.1, 13.2.2]

```
⌋ int MPI_File_close(MPI_File* fh)
```

```
FO8 MPI_File_close(fh, ierror)  
    type(MPI_File), intent(out) :: fh  
    integer, optional, intent(out) :: ierror
```

- Collective operation
- User must ensure that all outstanding nonblocking and split collective operations associated with the file have completed

DELETING A FILE [MPI-3.1, 13.2.3]

```
⌋ int MPI_File_delete(const char* filename, MPI_Info info)
```

```
FO8 MPI_File_delete(filename, info, ierror)  
character(len=*), intent(in) :: filename  
type(MPI_Info), intent(in) :: info  
integer, optional, intent(out) :: ierror
```

- Deletes the file identified by `filename`
- File deletion is a local operation and should be performed by a single process
- If the file does not exist an error is raised
- If the file is opened by any process
 - all further and outstanding access to the file is implementation dependent
 - it is implementation dependent whether the file is deleted; if it is not, an error is raised

FILE PARAMETERS

Setting File Parameters

MPI_File_set_size Set the size of a file [\[MPI-3.1, 13.2.4\]](#)

MPI_File_preallocate Preallocate disk space [\[MPI-3.1, 13.2.5\]](#)

MPI_File_set_info Supply additional information [\[MPI-3.1, 13.2.8\]](#)

Inspecting File Parameters

MPI_File_get_size Size of a file [\[MPI-3.1, 13.2.6\]](#)

MPI_File_get_amode Access mode [\[MPI-3.1, 13.2.7\]](#)

MPI_File_get_group Group of processes that opened the file [\[MPI-3.1, 13.2.7\]](#)

MPI_File_get_info Additional information associated with the file [\[MPI-3.1, 13.2.8\]](#)

I/O ERROR HANDLING [MPI-3.1, 8.3, 13.7]

Communication, by default, aborts the program when an error is encountered. I/O operations, by default, return an error code.

```
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler  
↪ errhandler)
```

```
MPI_File_set_errhandler(file, errhandler, ierror)  
type(MPI_File), intent(in) :: file  
type(MPI_Errhandler), intent(in) :: errhandler  
F08 integer, optional, intent(out) :: ierror
```

- The default error handler for files is MPI_ERRORS_RETURN
- Success is indicated by a return value of MPI_SUCCESS
- MPI_ERRORS_ARE_FATAL aborts the program
- Can be set for each file individually or for all files by setting the error handler on MPI_FILE_NULL

FILE VIEW [MPI-3.1, 13.3]

File View

A file view determines what part of the contents of a file is visible to a process. It is defined by a *displacement* (given in bytes) from the beginning of the file, an *elementary datatype* and a *file type*. The view into a file can be changed multiple times between opening and closing.

File Types and Elementary Types are Data Types

- Can be predefined or derived
- The usual constructors can be used to create derived file types and elementary types, e.g.
 - MPI_Type_indexed,
 - MPI_Type_create_struct,
 - MPI_Type_create_subarray
- Displacements in their typemap must be non-negative and monotonically nondecreasing
- Have to be committed before use

DEFAULT FILE VIEW [MPI-3.1, 13.3]

When newly opened, files are assigned a default view that is the same on all processes:

- Zero displacement
- File contains a contiguous sequence of bytes
- All processes have access to the entire file

File	0: byte	1: byte	2: byte	3: byte	...
Process 0	0: byte	1: byte	2: byte	3: byte	...
Process 1	0: byte	1: byte	2: byte	3: byte	...
...	0: byte	1: byte	2: byte	3: byte	...

ELEMENTARY TYPE [MPI-3.1, 13.3]

Elementary Type

An elementary type (or *etype*) is the unit of data contained in a file. Offsets are expressed in multiples of etypes, file pointers point to the beginning of etypes. Etypes can be basic or derived.

Changing the Elementary Type

E.g. `etype = MPI_INT`:

File	0: int	1: int	2: int	3: int	...
Process 0	0: int	1: int	2: int	3: int	...
Process 1	0: int	1: int	2: int	3: int	...
...	0: int	1: int	2: int	3: int	...

FILE TYPE [MPI-3.1, 13.3]

File Type

A file type describes an access pattern. It can contain either instances of the *etype* or holes with an extent that is divisible by the extent of the *etype*.

Changing the File Type

E.g. $Filetype_0 = \{(int, 0), (hole, 4), (hole, 8)\}$, $Filetype_1 = \{(hole, 0), (int, 4), (hole, 8)\}$,
...

File	0: int	1: int	2: int	3: int	...
Process 0	0: int			1: int	
Process 1		0: int			...
...			0: int		

CHANGING THE FILE VIEW [MPI-3.1, 13.3]

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,  
    ↪ MPI_Datatype etype, MPI_Datatype filetype, const  
    ↪ char* datarep, MPI_Info info)
```

```
MPI_File_set_view(fh, disp, etype, filetype, datarep,  
    ↪ info, ierror)  
type(MPI_File), intent(in) :: fh  
integer(kind=MPI_OFFSET_KIND), intent(in) :: disp  
type(MPI_Datatype), intent(in) :: etype, filetype  
character(len=*), intent(in) :: datarep  
type(MPI_Info), intent(in) :: info  
F08 integer, optional, intent(out) :: ierror
```

- Collective operation
- datarep and extent of etype must match
- disp, filetype and info can be distinct
- File pointers are reset to zero
- May not overlap with nonblocking or split collective operations

DATA REPRESENTATION [MPI-3.1, 13.5]

- Determines the conversion of data in memory to data on disk
- Influences the interoperability of I/O between heterogeneous parts of a system or different systems

"native"

Data is stored in the file exactly as it is in memory

- + No loss of precision
- + No overhead
- On heterogeneous systems loss of transparent interoperability

DATA REPRESENTATION [MPI-3.1, 13.5]

"internal"

Data is stored in implementation-specific format

- + Can be used in a homogeneous *and* heterogeneous environment
- + Implementation will perform conversions if necessary
- Can incur overhead
- Not necessarily compatible between different implementations

"external32"

Data is stored in standardized data representation (big-endian IEEE)

- + Can be read/written also by non-MPI programs
- Precision and I/O performance may be lost due to type conversions between native and external32 representations
- Not available in all implementations

DATA ACCESS

Three orthogonal aspects

1. Synchronism
 1. Blocking
 2. Nonblocking
 3. Split collective
2. Coordination
 1. Noncollective
 2. Collective
3. Positioning
 1. Explicit offsets
 2. Individual file pointers
 3. Shared file pointers

POSIX <code>read()</code> and <code>write()</code>
These are blocking, noncollective operations with individual file pointers.

SYNCHRONISM

Blocking I/O

Blocking I/O routines do not return before the operation is completed.

Nonblocking I/O

- Nonblocking I/O routines do not wait for the operation to finish
- A separate completion routine is necessary [[MPI-3.1](#), [3.7.3](#), [3.7.5](#)]
- The associated buffers must not be used while the operation is in flight

Split Collective

- "Restricted" form of nonblocking collective
- Buffers must not be used while in flight
- Does not allow other collective accesses to the file while in flight
- `begin` and `end` must be used from the same thread

COORDINATION

Noncollective

The completion depends only on the activity of the calling process.

Collective

- Completion may depend on activity of other processes
- Opens opportunities for optimization

POSITIONING [MPI-3.1, 13.4.1 – 13.4.4]

Explicit Offset

- No file pointer is used
- File position for access is given directly as function argument

Individual File Pointers

- Each process has its own file pointer
- After access, pointer is moved to first *etype* after the last one accessed

Shared File Pointers

- All processes share a single file pointer
- All processes must use the same file view
- Individual accesses appear as if serialized (with an unspecified order)
- Collective accesses are performed in order of ascending rank

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	nonblocking	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
	split collective	N/A	MPI_File_read_at_all_begin MPI_File_read_at_all_end MPI_File_write_at_all_begin MPI_File_write_at_all_end
individual file pointers	blocking	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	nonblocking	MPI_File_iread MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
	split collective	N/A	MPI_File_read_all_begin MPI_File_read_all_end MPI_File_write_all_begin MPI_File_write_all_end
shared file pointers	blocking	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	nonblocking	MPI_File_iread_shared MPI_File_iwrite_shared	N/A
	split collective	N/A	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

WRITING

blocking, noncollective, explicit offset [MPI-3.1, 13.4.2]

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset,  
    ↪ const void* buf, int count, MPI_Datatype datatype,  
    ↪ MPI_Status *status)
```

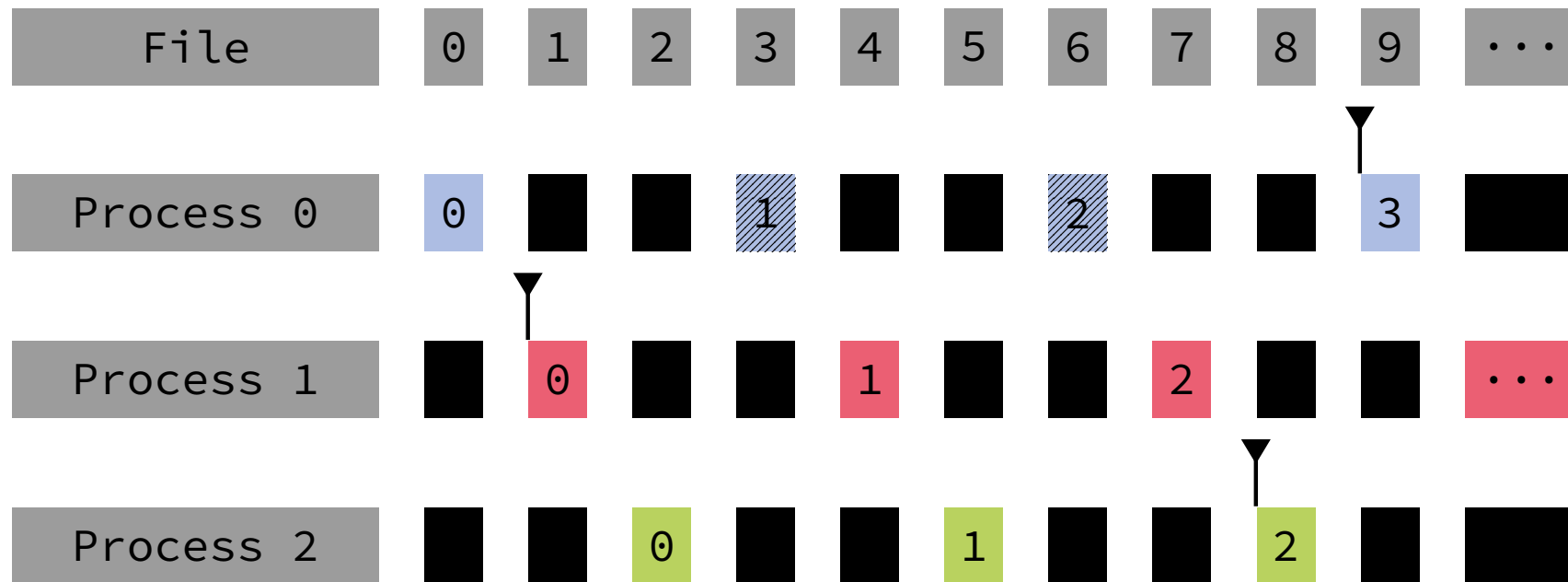
```
MPI_File_write_at(fh, offset, buf, count, datatype,  
    ↪ status, ierror)  
type(MPI_File), intent(in) :: fh  
integer(kind=MPI_OFFSET_KIND), intent(in) :: offset  
type(*), dimension(..), intent(in) :: buf  
integer, intent(in) :: count  
type(MPI_Datatype), intent(in) :: datatype  
F08 integer, optional, intent(out) :: ierror
```

- Starting offset for access is explicitly given
- No file pointer is updated
- Writes count elements of datatype from memory starting at buf
- $Typesig(datatype) = Typesig(etype) \dots Typesig(etype)$
- Writing past end of file increases the file size

EXAMPLE

blocking, noncollective, explicit offset [MPI-3.1, 13.4.2]

Process 0 calls `MPI_File_write_at(offset = 1, count = 2):`



WRITING

blocking, noncollective, individual [MPI-3.1, 13.4.3]

```
int MPI_File_write(MPI_File fh, const void* buf, int
↳ count, MPI_Datatype datatype, MPI_Status* status)
```

```

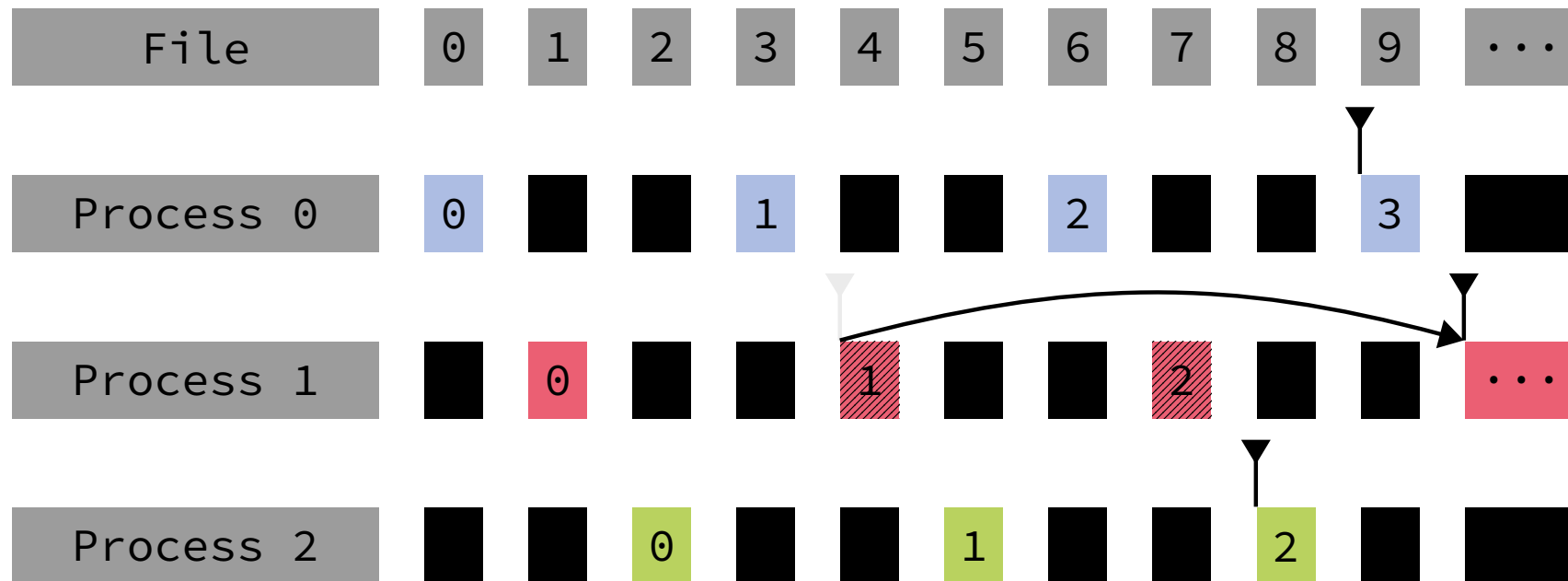
MPI_File_write(fh, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Status) :: status
F08 integer, optional, intent(out) :: ierror
```

- Starts writing at the current position of the individual file pointer
- Moves the individual file pointer by the count of *etypes* written

EXAMPLE

blocking, noncollective, individual [MPI-3.1, 13.4.3]

With its file pointer at element 1, process 1 writes count = 2:



WRITING

nonblocking, noncollective, individual [MPI-3.1, 13.4.3]

```
int MPI_File_iwrite(MPI_File fh, const void* buf, int  
↪ count, MPI_Datatype datatype, MPI_Request* request)
```

```
MPI_File_iwrite(fh, buf, count, datatype, request,  
↪ ierror)  
type(MPI_File), intent(in) :: fh  
type(*), dimension(..), intent(in) :: buf  
integer, intent(in) :: count  
type(MPI_Datatype), intent(in) :: datatype  
type(MPI_Request), intent(out) :: request  
F08 integer, optional, intent(out) :: ierror
```

- Starts the same operation as `MPI_File_write` but does not wait for completion
- Returns a request object that is used to complete the operation

WRITING

blocking, collective, individual [MPI-3.1, 13.4.3]

```
int MPI_File_write_all(MPI_File fh, const void* buf, int  
↪ count, MPI_Datatype datatype, MPI_Status* status)
```

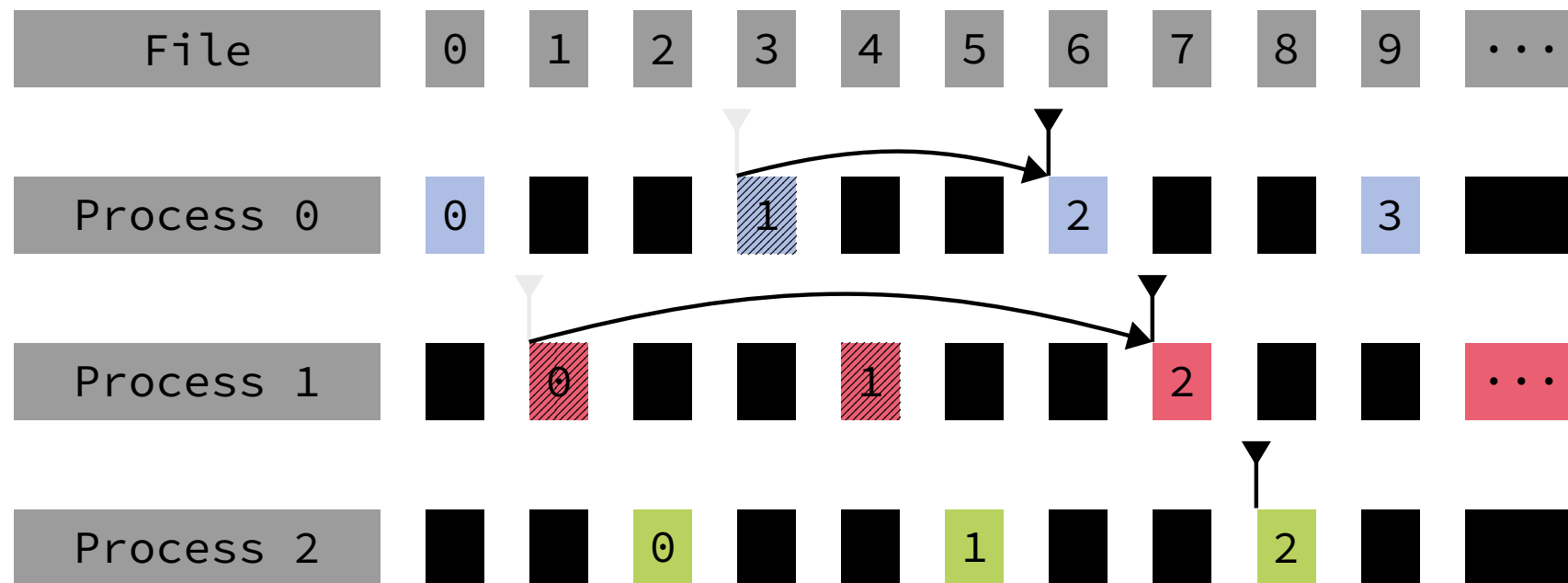
```
MPI_File_write_all(fh, buf, count, datatype, status,  
↪ ierror)  
type(MPI_File), intent(in) :: fh  
type(*), dimension(..), intent(in) :: buf  
integer, intent(in) :: count  
type(MPI_Datatype), intent(in) :: datatype  
type(MPI_Status) :: status  
F08 integer, optional, intent(out) :: ierror
```

- Same signature as MPI_File_write, but collective coordination
- Each process uses its individual file pointer
- MPI can use communication between processes to funnel I/O

EXAMPLE

blocking, collective, individual [MPI-3.1, 13.4.3]

- With its file pointer at element 1, process 0 writes count = 1,
- With its file pointer at element 0, process 1 writes count = 2,
- With its file pointer at element 2, process 2 writes count = 0:



WRITING

split-collective, individual [MPI-3.1, 13.4.5]

```
int MPI_File_write_all_begin(MPI_File fh, const void* buf,  
↪ int count, MPI_Datatype datatype)
```

```
MPI_File_write_all_begin(fh, buf, count, datatype,  
↪ ierror)  
type(MPI_File), intent(in) :: fh  
type(*), dimension(..), intent(in) :: buf  
integer, intent(in) :: count  
type(MPI_Datatype), intent(in) :: datatype  
F08 integer, optional, intent(out) :: ierror
```

- Same operation as MPI_File_write_all, but split-collective
- status is returned by the corresponding end routine

WRITING

split-collective, individual [MPI-3.1, 13.4.5]

```
int MPI_File_write_all_end(MPI_File fh, const void* buf,  
↪ MPI_Status* status)
```

```
MPI_File_write_all_end(fh, buf, status, ierror)  
type(MPI_File), intent(in) :: fh  
type(*), dimension(..), intent(in) :: buf  
type(MPI_Status) :: status  
F08 integer, optional, intent(out) :: ierror
```

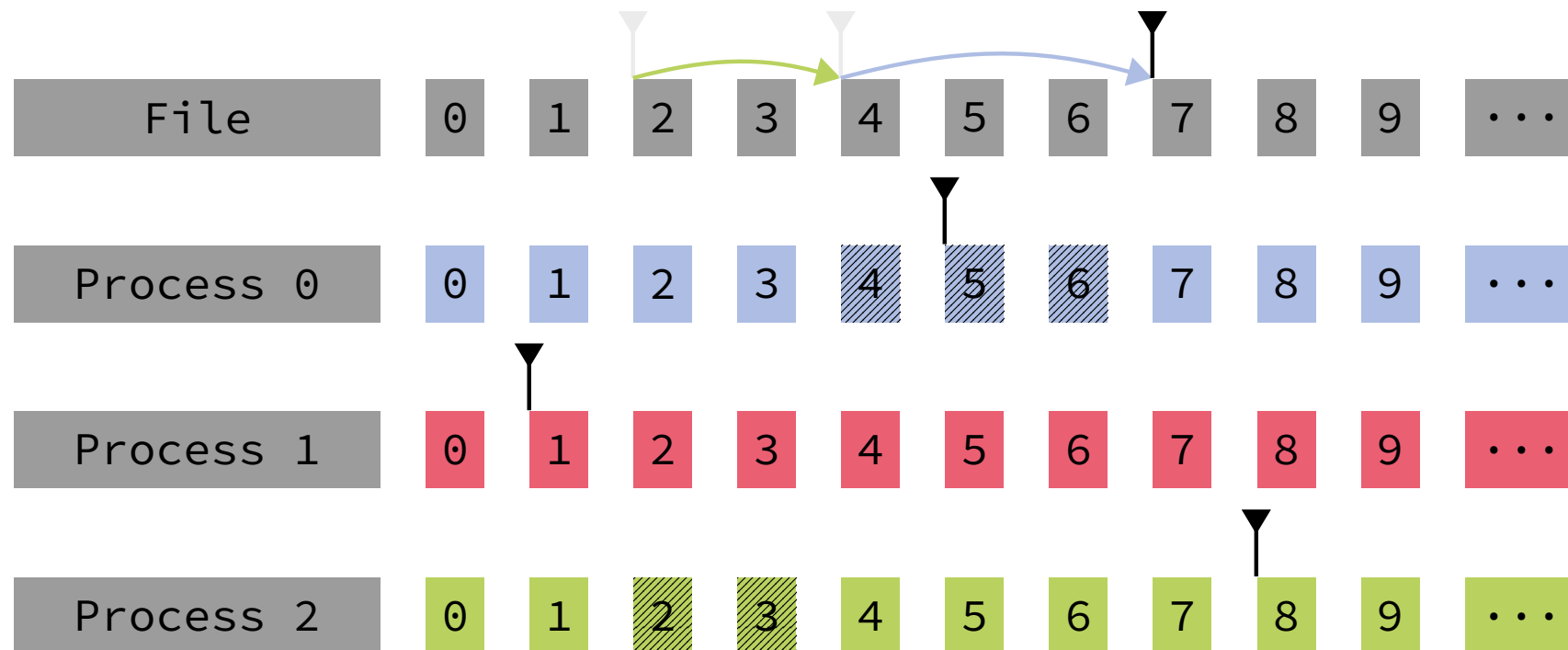
- buf argument must match corresponding begin routine

EXAMPLE

blocking, noncollective, shared [MPI-3.1, 13.4.4]

With the shared pointer at element 2,

- process 0 writes count = 3,
- process 2 writes count = 2:

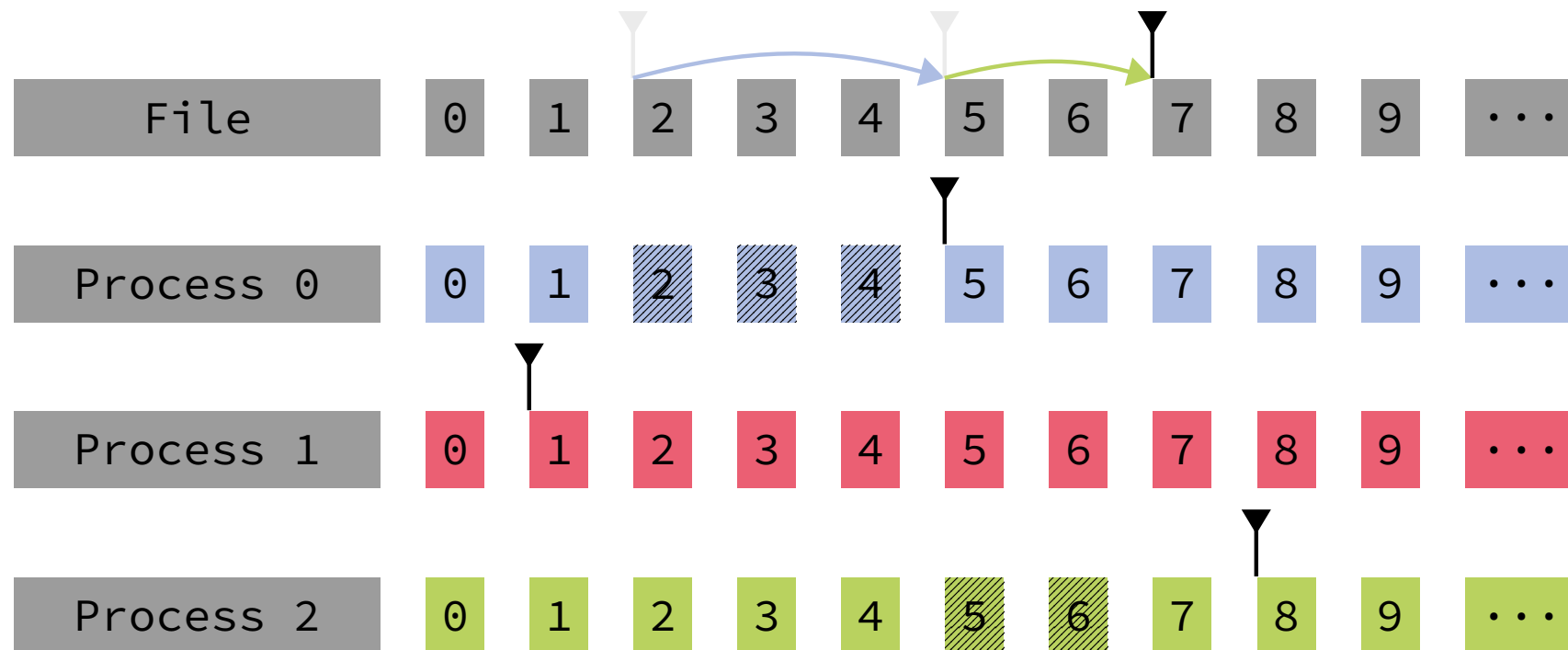


EXAMPLE

blocking, noncollective, shared [MPI-3.1, 13.4.4]

With the shared pointer at element 2,

- process 0 writes count = 3,
- process 2 writes count = 2:

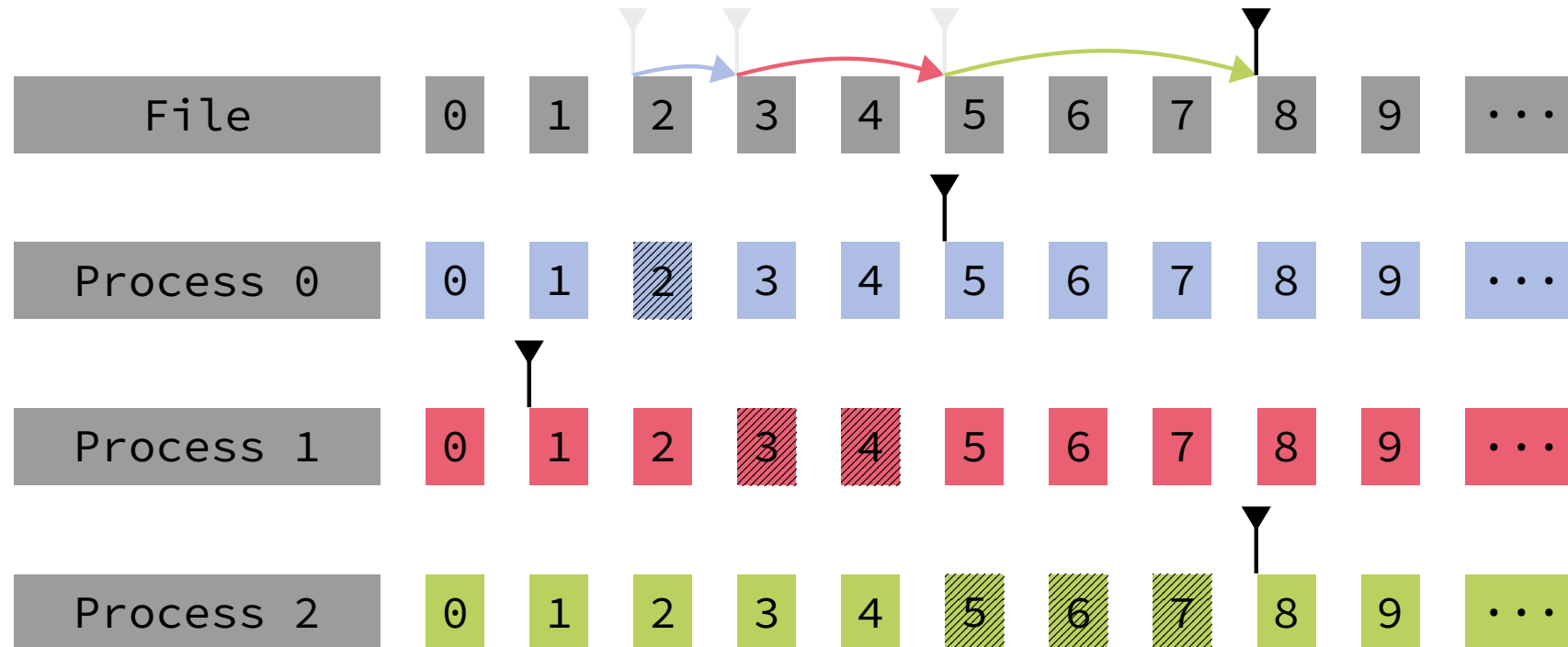


EXAMPLE

blocking, collective, shared [MPI-3.1, 13.4.4]

With the shared pointer at element 2,

- process 0 writes count = 1,
- process 1 writes count = 2,
- process 2 writes count = 3:



READING

blocking, noncollective, individual [MPI-3.1, 13.4.3]

```
int MPI_File_read(MPI_File fh, void* buf, int count,  
↪ MPI_Datatype datatype, MPI_Status* status)
```

```
MPI_File_read(fh, buf, count, datatype, status, ierror)  
type(MPI_File), intent(in) :: fh  
type(*), dimension(..) :: buf  
integer, intent(in) :: count  
type(MPI_Datatype), intent(in) :: datatype  
type(MPI_Status) :: status  
F08 integer, optional, intent(out) :: ierror
```

- Starts reading at the current position of the individual file pointer
- Reads up to count elements of datatype into the memory starting at buf
- status indicates how many elements have been read
- If status indicates less than count elements read, the end of file has been reached

FILE POINTER POSITION [MPI-3.1, 13.4.3]

```
int MPI_File_get_position(MPI_File fh, MPI_Offset*  
↪ offset)
```

```
FO8 MPI_File_get_position(fh, offset, ierror)  
type(MPI_File), intent(in) :: fh  
integer(kind=MPI_OFFSET_KIND), intent(out) :: offset  
integer, optional, intent(out) :: ierror
```

- Returns the current position of the individual file pointer in units of *etype*
- Value can be used for e.g.
 - return to this position (via seek)
 - calculate a displacement
- `MPI_File_get_position_shared` queries the position of the shared file pointer

SEEKING TO A FILE POSITION [MPI-3.1, 13.4.3]

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int  
↳ whence)
```

```
MPI_File_seek(fh, offset, whence, ierror)  
type(MPI_File), intent(in) :: fh  
integer(kind=MPI_OFFSET_KIND), intent(in) :: offset  
integer, intent(in) :: whence  
F08 integer, optional, intent(out) :: ierror
```

- whence controls how the file pointer is moved:
 - MPI_SEEK_SET sets the file pointer to offset
 - MPI_SEEK_CUR offset is added to the current value of the pointer
 - MPI_SEEK_END offset is added to the end of the file
- offset can be negative but the resulting position may not lie before the beginning of the file
- MPI_File_seek_shared manipulates the shared file pointer

CONVERTING OFFSETS [MPI-3.1, 13.4.3]

```
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset  
↪ offset, MPI_Offset* disp)
```

```
MPI_File_get_byte_offset(fh, offset, disp, ierror)  
type(MPI_File), intent(in) :: fh  
integer(kind=MPI_OFFSET_KIND), intent(in) :: offset  
integer(kind=MPI_OFFSET_KIND), intent(out) :: disp  
F08 integer, optional, intent(out) :: ierror
```

- Converts a view relative offset (in units of *etype*) into a displacement in bytes from the beginning of the file

CONSISTENCY [MPI-3.1, 13.6.1]

Sequential Consistency

If a set of operations is sequentially consistent, they behave as if executed in some serial order. The exact order is unspecified.

- To guarantee sequential consistency, certain requirements must be met
- Requirements depend on access path and file atomicity

Result of operations that are not sequentially consistent is implementation dependent.

ATOMIC MODE [MPI-3.1, 13.6.1]

Requirements for sequential consistency

Same file handle: always sequentially consistent

File handles from same open: always sequentially consistent

File handles from different open: not influenced by atomicity, see nonatomic mode

- Atomic mode is not the default setting
- Can lead to overhead, because MPI library has to uphold guarantees in general case

```
⌋ int MPI_File_set_atomicity(MPI_File fh, int flag)
```

```
FO8 MPI_File_set_atomicity(fh, flag, ierror)  
    type(MPI_File), intent(in) :: fh  
    logical, intent(in) :: flag  
    integer, optional, intent(out) :: ierror
```

NONATOMIC MODE [MPI-3.1, 13.6.1]

Requirements for sequential consistency

Same file handle: operations must be either nonconcurrent, nonconflicting, or both

File handles from same open: nonconflicting accesses are sequentially consistent, conflicting accesses have to be protected using `MPI_File_sync`

File handles from different open: all accesses must be protected using `MPI_File_sync`

Conflicting Accesses

Two accesses are conflicting if they touch overlapping parts of a file and at least one is writing.

```
⌋ int MPI_File_sync(MPI_File fh)
```

```
FO8 MPI_File_sync(fh, ierror)  
      type(MPI_File), intent(in) :: fh  
      integer, optional, intent(out) :: ierror
```

NONATOMIC MODE [MPI-3.1, 13.6.1]

The Sync-Barrier-Sync construct

```
// writing access
↳ sequence through one
↳ file handle
MPI_File_sync(fh0);
MPI_Barrier(
↳ MPI_COMM_WORLD);
MPI_File_sync(fh0);
↳ // ...
```

```
// ...
MPI_File_sync(fh1);
MPI_Barrier(
↳ MPI_COMM_WORLD);
MPI_File_sync(fh1);
// access sequence to
↳ the same file
↳ through a different
↳ file handle
↳
```

- MPI_File_sync is used to delimit sequences of accesses through different file handles
- Sequences that contain a write access may not be concurrent with any other access sequence

EXERCISES

Exercise 6 – Data Access	6.1 Writing and Reading Data
	Write a program: • Each process writes its own rank to the common file <code>rank.dat</code> • The ranks should be in order in the file $0 \dots n - 1$ • Process 0 reads the whole file and prints the contents to screen
	6.2 Accessing Parts of Files
	Take the file <code>rank.dat</code> from the previous exercise • The processes read the integers in the file in reverse order, i.e. process 0 reads the last entry, process 1 reads the one before, ... • Each process prints its rank and the integer it read to the screen Careful: This program might be run on a different number of processes

EXERCISES

Exercise 6 – Data Access

6.3 Phone Book

The file `phonebook.dat` contains several records of the following form:

```
struct dbentry {  
    int key;  
    int room_number;  
    int phone_number;  
    char name[200];  
}
```

```
type :: dbentry  
    integer :: key  
    integer :: room_number  
    integer :: phone_number  
    character(len=200) ::  
        ↪ name  
end type
```

Using MPI I/O, find out who sits in room 234.

Note: This exercise can be solved by either a serial or a parallel program. Try a serial solution first.

Part IX: Tools

MUST

Marmot Umpire Scalable Tool



<https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST>

MUST checks for correct usage of MPI. It includes checks for the following classes of mistakes:

- Constants and integer values
- Communicator usage
- Datatype usage
- Group usage
- Operation usage
- Request usage
- Leak checks (MPI resources not freed before calling `MPI_Finalize`)
- Type mismatches
- Overlapping buffers passed to MPI
- Deadlocks resulting from MPI calls
- Basic checks for thread level usage (`MPI_Init_thread`)

MUST USAGE

On JURECA, load the MUST module:

```
$ module load MUST
```

Build your application:

```
$ mpicc -o application.x application.c  
$ # or  
$ mpif90 -o application.x application.f90
```

Replace the MPI starter (e.g. srun) with MUST's own mustrun:

```
$ mustrun -n 4 --must:mpiexec srun --must:np -n ./application.x
```

Different modes of operation (for improved scalability or graceful handling of application crashes) are available via command line switches.

MPI Fortran 2008 Interface Compatibility

MUST is not compatible with MPI's Fortran 2008 interface.

EXERCISES

Exercise 7 – MPI Tools	7.1 Must
	Have a look at the file <code>must.{c c++ f90}</code>. It contains a variation of the solution to exercise 3.1 – it should calculate the sum of all ranks and make the result available on all processes. 1. Compile the program and try to run it. 2. Use MUST to discover what is wrong with the program. 3. If any mistakes were found, fix them and go back to 1. Note <code>must.f90</code> uses the MPI Fortran 90 interface.

Part X: Communicators

MOTIVATION

Communicators are a scope for communication within or between groups of processes. New communicators with different scope or topological properties can be used to accommodate certain needs.

- **Separation of communication spaces:** A software library that uses MPI underneath is used in an application that directly uses MPI itself. Communication due to the library should not conflict with communication due to the application.
- **Partitioning of process groups:** Parts of your software exhibit a collective communication pattern, but only across a subset of processes.
- **Exploiting inherent topology:** Your application uses a regular cartesian grid to discretize the problem and this translates into certain nearest neighbor communication patterns.

DUPLICATE [MPI-3.1, 6.4.2]

```
⌋ int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

```
FO8 MPI_Comm_dup(comm, newcomm, ierror)  
      type(MPI_Comm), intent(in) :: comm  
      type(MPI_Comm), intent(out) :: newcomm  
      integer, optional, intent(out) :: ierror
```

- Duplicates an existing communicator comm
- New communicator has the same properties but a new context

SPLIT [MPI-3.1, 6.4.2]

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,  
↳ MPI_Comm *newcomm)
```

```
FOR MPI_Comm_split(comm, color, key, newcomm, ierror)  
type(MPI_Comm), intent(in) :: comm  
integer, intent(in) :: color, key  
type(MPI_Comm), intent(out) :: newcomm  
integer, optional, intent(out) :: ierror
```

- Splits the processes in a communicator into disjoint subgroups
- Processes are grouped by color, one new communicator per distinct value
- Special color value MPI_UNDEFINED does not create a new communicator (MPI_COMM_NULL is returned in newcomm)
- Processes are ordered by ascending value of key in new communicator

CARTESIAN TOPOLOGY [MPI-3.1, 7.5.1]

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const  
    ↪ int dims[], const int periods[], int reorder,  
    ↪ MPI_Comm *comm_cart)
```

```
MPI_Cart_create(comm_old, ndims, dims, periods, reorder,  
    ↪ comm_cart, ierror)  
type(MPI_Comm), intent(in) :: comm_old  
integer, intent(in) :: ndims, dims(ndims)  
logical, intent(in) :: periods(ndims), reorder  
type(MPI_Comm), intent(out) :: comm_cart  
F08 integer, optional, intent(out) :: ierror
```

- Creates a new communicator with processes arranged on a (possibly periodic) Cartesian grid
- The grid has `ndims` dimensions and `dims[i]` points in dimension `i`
- If `reorder` is true, MPI is free to assign new ranks to processes

COMPARISON [MPI-3.1, 6.4.1]

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int  
↪ *result)
```

```
MPI_Comm_compare(comm1, comm2, result, ierror)  
type(MPI_Comm), intent(in) :: comm1, comm2  
integer, intent(out) :: result  
FO8 integer, optional, intent(out) :: ierror
```

Compares two communicators. The result is one of:

MPI_IDENT The two communicators are the same.

MPI_CONGRUENT The two communicators consist of the same processes in the same order but communicate in different contexts.

MPI_SIMILAR The two communicators consist of the same processes in a different order.

MPI_UNEQUAL Otherwise.

RANK TO COORDINATE [MPI-3.1, 7.5.5]

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,  
↪ int coords[])
```

```
MPI_Cart_coords(comm, rank, maxdims, coords, ierror)  
type(MPI_Comm), intent(in) :: comm  
integer, intent(in) :: rank, maxdims  
integer, intent(out) :: coords(maxdims)  
F08 integer, optional, intent(out) :: ierror
```

Translates the rank of a process into its coordinate on the Cartesian grid.

COORDINATE TO RANK [MPI-3.1, 7.5.5]

```
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int  
↪ *rank)
```

```
MPI_Cart_rank(comm, coords, rank, ierror)  
type(MPI_Comm), intent(in) :: comm  
integer, intent(in) :: coords(*)  
integer, intent(out) :: rank  
F08 integer, optional, intent(out) :: ierror
```

Translates the coordinate on the Cartesian grid of a process into its rank.

CARTESIAN SHIFT [MPI-3.1, 7.5.6]

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,  
↪ int *rank_source, int *rank_dest)
```

```
MPI_Cart_shift(comm, direction, disp, rank_source,  
↪ rank_dest, ierror)  
type(MPI_Comm), intent(in) :: comm  
integer, intent(in) :: direction, disp  
integer, intent(out) :: rank_source, rank_dest  
F08 integer, optional, intent(out) :: ierror
```

- Calculates the ranks of source and destination processes in a shift operation on a Cartesian grid
- `direction` gives the number of the axis (starting at 0)
- `disp` gives the displacement

NULL PROCESSES [MPI-3.1, 3.11]

```
C int MPI_PROC_NULL = /* implementation defined */
```

```
FOR integer, parameter :: MPI_PROC_NULL = ! implementation  
    ↪ defined
```

- Can be used as source or destination for point-to-point communication
- Communication with MPI_PROC_NULL has no effect
- May simplify code structure (communication with special source/destination instead of branch)
- MPI_Cart_shift returns MPI_PROC_NULL for out of range shifts

FREE [MPI-3.1, 6.4.3]

```
⌋ int MPI_Comm_free(MPI_Comm *comm)
```

```
⌋ MPI_Comm_free(comm, ierror)  
type(MPI_Comm), intent(inout) :: comm  
integer, optional, intent(out) :: ierror
```

Marks a communicator for deallocation.

EXERCISES

Exercise 8 – Communicators	8.1 Cartesian Topology
	Redo exercise 3.1 using a Cartesian communicator.
	8.2 Split
	Redo exercise 3.2 using a new split communicator per communication round.

Part XI: Thread Compliance

THREAD COMPLIANCE [MPI-3.1, 12.4]

- An MPI library is thread compliant if
 1. Concurrent threads can make use of MPI routines and the result will be as if they were executed in some order.
 2. Blocking routines will only block the executing thread, allowing other threads to make progress.
- MPI libraries are not required to be thread compliant
- Alternative initialization routines to request certain levels of thread compliance
- These functions are always safe to use in a multithreaded setting:
MPI_Initialized, MPI_Finalized, MPI_Query_thread,
MPI_Is_thread_main, MPI_Get_version,
MPI_Get_library_version

THREAD SUPPORT LEVELS [MPI-3.1, 12.4.3]

The following predefined values are used to express all possible levels of thread support:

MPI_THREAD_SINGLE program is single threaded

MPI_THREAD_FUNNELED MPI routines are only used by the *main thread*

MPI_THREAD_SERIALIZED MPI routines are used by multiple threads, but not concurrently

MPI_THREAD_MULTIPLE MPI is thread compliant, no restrictions

$\text{MPI_THREAD_SINGLE} < \text{MPI_THREAD_FUNNELED} < \text{MPI_THREAD_SERIALIZED} < \text{MPI_THREAD_MULTIPLE}$

INITIALIZATION [MPI-3.1, 12.4.3]

```
int MPI_Init_thread(int* argc, char*** argv, int required,  
↪ int* provided)
```

```
MPI_Init_thread(required, provided, ierror)  
integer, intent(in) :: required  
integer, intent(out) :: provided  
F08 integer, optional, intent(out) :: ierror
```

- required and provided specify thread support levels
- If possible, provided = required
- Otherwise, if possible, provided > required
- Otherwise, provided < required
- MPI_Init is equivalent to required = MPI_THREAD_SINGLE

INQUIRY FUNCTIONS [MPI-3.1, 12.4.3]

Query level of thread support:

```
⌋ int MPI_Query_thread(int *provided)
```

```
FO8 MPI_Query_thread(provided, ierror)  
      integer, intent(out) :: provided  
      integer, optional, intent(out) :: ierror
```

Check whether the calling thread is the *main thread*:

```
⌋ int MPI_Is_thread_main(int* flag)
```

```
FO8 MPI_Is_thread_main(flag, ierror)  
      logical, intent(out) :: flag  
      integer, optional, intent(out) :: ierror
```

MATCHING PROBE [MPI-3.1, 3.8.2]

```
int MPI_Mprobe(int source, int tag, MPI_Comm comm,  
↳ MPI_Message* message, MPI_Status* status)
```

```
MPI_Mprobe(source, tag, comm, message, status, ierror)  
integer, intent(in) :: source, tag  
type(MPI_Comm), intent(in) :: comm  
type(MPI_Message), intent(out) :: message  
type(MPI_Status) :: status  
F08 integer, optional, intent(out) :: ierror
```

- Works like MPI_Probe, except for the returned MPI_Message value which may be used to receive exactly the probed message
- Nonblocking variant MPI_Improbe exists

MATCHED RECEIVE [MPI-3.1, 3.8.3]

```
int MPI_Mrecv(void* buf, int count, MPI_Datatype datatype,  
↪ MPI_Message* message, MPI_Status* status)
```

```
MPI_Mrecv(buf, count, datatype, message, status, ierror)  
type(*), dimension(..) :: buf  
integer, intent(in) :: count  
type(MPI_Datatype), intent(in) :: datatype  
type(MPI_Message), intent(inout) :: message  
type(MPI_Status) :: status  
F08 integer, optional, intent(out) :: ierror
```

- Receives the previously probed message message
- Sets the message handle to MPI_MESSAGE_NULL
- Nonblocking variant MPI_Imrecv exists

CLARIFICATIONS [MPI-3.1, 12.4.2]

Initialization and Completion

Initialization and finalization of MPI should occur on the same thread, the *main thread*.

Request Completion

Multiple threads must not try to complete the same request (e.g. MPI_Wait).

Probe

In multithreaded settings, MPI_Probe might match a different message as a subsequent MPI_Recv.

Part XII: First Steps with OpenMP

WHAT IS OPENMP?

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs.

OpenMP FAQ³

- Initially targeted SMP systems, now also DSPs, accelerators, etc.
- Provides *specifications* (not implementations)
- Portable across different platforms

Current version of the specification: 4.5 (November 2015)

³Matthijs van Waveren et al. *OpenMP FAQ. Version 2.0.* Nov. 12, 2013. URL: <http://www.openmp.org/about/openmp-faq/> (visited on 07/19/2017).

BRIEF HISTORY

1997	FORTRAN version 1.0	2013	Version 4.0, thread affinity, SIMD, devices, tasks (dependencies, groups, and cancellation), improved Fortran 2003 compatibility
1998	C/C++ version 1.0		
1999	FORTRAN version 1.1		
2000	FORTRAN version 2.0		
2002	C/C++ version 2.0	2015	Version 4.5, extended SIMD and devices facilities, task priorities
2005	First combined version 2.5, memory model, internal control variables, clarifications	2017	Version 5.0 <i>Preview 2</i> (not standardized yet), memory model, base language compatibility, allocators, extended task and devices facilities
2008	Version 3.0, tasks		
2011	Version 3.1, extended task facilities		

COVERAGE

- Directives

- Directive Format ✓
- Conditional Compilation ✓
- Internal Control Variables ✓
- Array Sections
- parallel Construct ✓
- Canonical Loop Form ✓
- Worksharing Constructs ✓
- SIMD Constructs
- Tasking Constructs ✓
- Device Constructs
- Combined Constructs (✓)
- if Clause ✓
- Master and Synchronization Constructs and Clauses ✓
- Cancellation Constructs
- Data Environment ✓
- declare reduction Directive
- Nesting of Regions

- Runtime Library Routines

- Runtime Library Definitions (✓)
- Execution Environment Routines (✓)
- Lock Routines ✓
- Timing Routines
- Device Memory Routines
- ...

- Environment Variables (✓)

- ...

LITERATURE

Official Resources

- OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Version 4.5. Nov. 2015. URL: <http://www.openmp.org/mp-documents/openmp-4.5.pdf>
- OpenMP Architecture Review Board. *OpenMP Application Programming Interface. Examples*. Version 4.5.0. Nov. 2016. URL: <http://www.openmp.org/mp-documents/openmp-examples-4.5.0.pdf>
- <http://www.openmp.org>

Recommended by <http://www.openmp.org/resources/openmp-books/>

- Ruud van der Pas, Eric Stotzer, and Christian Terboven. *Using OpenMP—The Next Step. Affinity, Accelerators, Tasking, and SIMD*. 1st ed. The MIT Press, Oct. 13, 2017. 392 pp. ISBN: 9780262534789

Additional Literature

- Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming. Patterns for Efficient Computation*. 1st ed. Morgan Kaufmann, July 31, 2012. 432 pp. ISBN: 9780124159938
- Victor Alessandrini. *Shared Memory Application Programming. Concepts and Strategies in Multicore Application Programming*. 1st ed. Morgan Kaufmann, Oct. 27, 2015. 556 pp. ISBN: 9780128037614

LITERATURE

Older Works (<http://www.openmp.org/resources/openmp-books/>)

- Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP. Portable Shared Memory Parallel Programming*. 1st ed. Scientific and Engineering Computation. The MIT Press, Oct. 12, 2007. 384 pp. ISBN: 9780262533027
- Rohit Chandra et al. *Parallel Programming in OpenMP*. 1st ed. Morgan Kaufmann, Oct. 11, 2000. 231 pp. ISBN: 9781558606715
- Michael Quinn. *Parallel Programming in C with MPI and OpenMP*. 1st ed. McGraw-Hill, June 5, 2003. 544 pp. ISBN: 9780072822564
- Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. 1st ed. Software Patterns. Sept. 15, 2004. 384 pp. ISBN: 9780321228116

THREADS & TASKS

Thread

An execution entity with a stack and associated static memory, called *threadprivate memory*.

OpenMP Thread

A *thread* that is managed by the OpenMP runtime system.

Team

A set of one or more *threads* participating in the execution of a *parallel region*.

Task

A specific instance of executable code and its *data environment*, generated when a *thread* encounters a *task*, *taskloop*, *parallel*, *target* or *teams construct*.

LANGUAGE

Base Language

A programming language that serves as the foundation of the OpenMP specification.

The following base languages are given in [\[OpenMP-4.5, 1.6\]](#): C90, C99, C++98, Fortran 77, Fortran 90, Fortran 95 and a subset of Fortran 2003.

Base Program

A program written in the *base language*.

OpenMP Program

A program that consists of a *base program*, annotated with OpenMP *directives* and runtime library routines.

Directive

In C/C++, a *#pragma*, and in Fortran, a comment, that specifies *OpenMP program* behavior.

COMPILING & LINKING

Compilers that conform to the OpenMP specification usually accept a command line argument that turns on OpenMP support, e.g.:

Intel C Compiler OpenMP Command Line Switch

```
$ icc -qopenmp ...
```

GNU Fortran Compiler OpenMP Command Line Switch

```
$ gfortran -fopenmp ...
```

The name of this command line argument is not mandated by the specification and differs from one compiler to another.

Naturally, these arguments are then also accepted by the MPI compiler wrappers:

Compiling Programs with Hybrid Parallelization

```
$ mpicc -qopenmp ...
```

RUNTIME LIBRARY DEFINITIONS

[OpenMP-4.5, 3.1]

C/C++ Runtime Library Definitions

Runtime library routines and associated types are defined in the `omp.h` header file.

```
␣ #include <omp.h>
```

Fortran Runtime Library Definitions

Runtime library routines and associated types are defined in either a Fortran **include** file

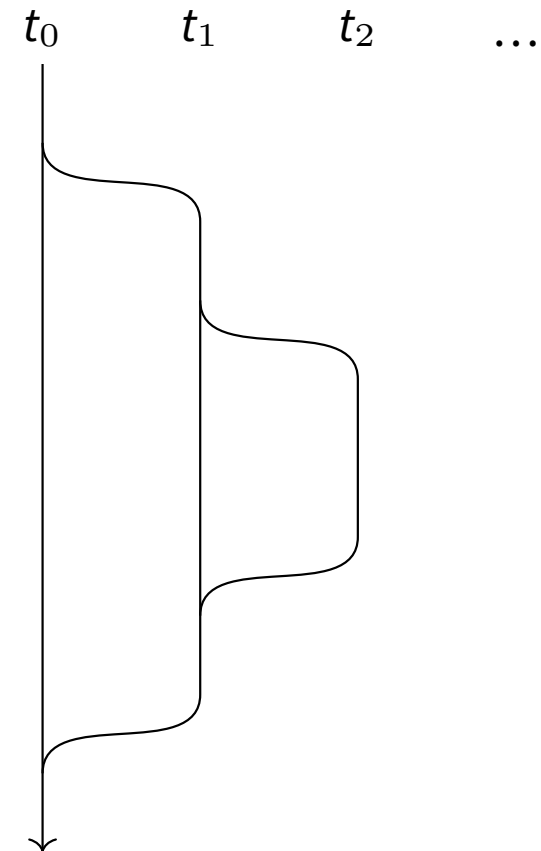
```
F77 include "omp_lib.h"
```

or a Fortran 90 module

```
F08 use omp_lib
```

WORLD ORDER IN OPENMP

- Program starts as one single-threaded process.
- Forks into teams of multiple threads when appropriate.
- Stream of instructions might be different for each thread.
- Information is exchanged via shared parts of memory.
- OpenMP threads may be nested inside MPI processes.



DIRECTIVE FORMAT

C and C++ [OpenMP-4.5, 2.1]

In C and C++, OpenMP directives are written using the *#pragma* method:

```
#pragma omp directive-name [clause[,] clause]...
```

- Directives are case-sensitive
- Applies to the next statement which must be a *structured block*

Structured Block

An executable statement, possibly compound, with a single entry at the top and a single exit at the bottom, or an OpenMP *construct*.

DIRECTIVE FORMAT

Fortran [OpenMP-4.5, 2.1.1, 2.1.2]

```
F08 sentinel directive-name [clause[[,] clause]...]
```

- Directives are case-insensitive

Fixed Form Sentinels

```
F08 sentinel = !$omp | c$omp | *$omp
```

- Must start in column 1
- The usual line length, white space, continuation and column rules apply
- Column 6 is blank for first line of directive, non-blank and non-zero for continuation

Free Form Sentinel

```
F08 sentinel = !$omp
```

- The usual line length, white space and continuation rules apply

CONDITIONAL COMPILATION [OpenMP-4.5, 2.2]

C Preprocessor Macro

```
⌋ #define _OPENMP yyymm
```

yyyy and mm are the year and month the OpenMP specification supported by the compiler was published.

Fortran Fixed Form Sentinels

```
FO8 !$ | *$ | c$
```

- Must start in column 1
- Only numbers or white space in columns 3–5
- Column 6 marks continuation lines

Fortran Free Form Sentinel

```
FO8 !$
```

- Must only be preceded by white space
- Can be continued with ampersand

CONSTRUCTS & REGIONS

Construct

An OpenMP *executable directive* (and for Fortran, the paired end *directive*, if any) and the associated statement, loop or *structured block*, if any, not including the code in any called routines. That is, the lexical extent of an *executable directive*.

Region

All code encountered during a specific instance of the execution of a given *construct* or of an OpenMP library routine.

Executable Directive

An OpenMP *directive* that is not declarative. That is, it may be placed in an executable context.

THE PARALLEL CONSTRUCT [OpenMP-4.5, 2.5]

```
C    #pragma omp parallel [clause[[,] clause]...]  
    structured-block
```

```
FOR !$omp parallel [clause[[,] clause]...]  
    structured-block  
!$omp end parallel
```

- Creates a team of threads to execute the `parallel` region
- Each thread executes the code contained in the structured block
- Inside the region threads are identified by consecutive numbers starting at zero
- Optional clauses (explained later) can be used to modify behavior and data environment of the `parallel` region

THREAD COORDINATES [OpenMP-4.5, 3.2.2, 3.2.4]

```
C int omp_get_num_threads(void);
```

```
F08 integer function omp_get_num_threads()
```

Returns the number of threads in the current team

```
C int omp_get_thread_num(void);
```

```
F08 integer function omp_get_thread_num()
```

Returns the number that identifies the calling thread within the current team (between zero and `omp_get_num_threads()`)

A FIRST OPENMP PROGRAM

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(void) {
    printf("Hello from your main thread.\n");

    #pragma omp parallel
        printf("Hello from thread %d of %d.\n",
            ↪ omp_get_thread_num(), omp_get_num_threads());

    printf("Hello again from your main thread.\n");
    return EXIT_SUCCESS;
}
```

A FIRST OPENMP PROGRAM

Program Output

```
$ gcc -fopenmp -o hello_openmp.x hello_openmp.c
$ ./hello_openmp.x
Hello from your main thread.
Hello from thread 1 of 8.
Hello from thread 0 of 8.
Hello from thread 3 of 8.
Hello from thread 4 of 8.
Hello from thread 6 of 8.
Hello from thread 7 of 8.
Hello from thread 2 of 8.
Hello from thread 5 of 8.
Hello again from your main thread.
```

A FIRST OPENMP PROGRAM

```
program hello_openmp
  use omp_lib
  implicit none

  print *, "Hello from your main thread."

  !$omp parallel
  print *, "Hello from thread ", omp_get_thread_num(), "
    ↪ of ", omp_get_num_threads(), "."
  !$omp end parallel

  print *, "Hello again from your main thread."
end program
```

F08

EXERCISES

Exercise 9 – Warm Up	9.1 Generalized Vector Addition (axpy)
	In the file <code>axpy.{c c++ f90}</code> , fill in the missing body of the function/subroutine <code>axpy_serial(a, x, y, z[, n])</code> so that it implements the generalized vector addition (in serial, without making use of OpenMP): $\mathbf{z} = a\mathbf{x} + \mathbf{y}.$ Compile the file into a program and run it to test your implementation.
	9.2 Dot Product
	In the file <code>dot.{c c++ f90}</code> , fill in the missing body of the function/subroutine <code>dot_serial(x, y[, n])</code> so that it implements the dot product (in serial, without making use of OpenMP): $\text{dot}(\mathbf{x}, \mathbf{y}) = \sum_i x_i y_i.$ Compile the file into a program and run it to test your implementation.

Part XIII: Low-Level OpenMP Concepts

MAGIC

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke⁴

⁴Arthur C. Clarke. *Profiles of the future : an inquiry into the limits of the possible*. London: Pan Books, 1973. ISBN: 9780330236195.

INTERNAL CONTROL VARIABLES

[OpenMP-4.5, 2.3]

Internal Control Variable (ICV)

A conceptual variable that specifies runtime behavior of a set of *threads* or *tasks* in an *OpenMP* program.

- Set to an initial value by the OpenMP implementation
- Some can be modified through either environment variables (e.g. OMP_NUM_THREADS) or API routines (e.g. `omp_set_num_threads()`)
- Some can be read through API routines (e.g. `omp_get_max_threads()`)
- Some are inaccessible to the user
- Might have different values in different scopes (e.g. data environment, device, global)
- Some can be overridden by clauses (e.g. the `num_threads()` clause)

PARALLELISM CLAUSES [OpenMP-4.5, 2.5, 2.12]

if Clause

```
└ if([parallel :] scalar-expression)
```

```
└ for if([parallel :] scalar-logical-expression)
```

If *false*, the region is executed only by the encountering thread(s) and no additional threads are forked.

num_threads Clause

```
└ num_threads(integer-expression)
```

```
└ for num_threads(scalar-integer-expression)
```

Requests a team size equal to the value of the expression (overrides the *nthreads-var* ICV)

EXAMPLE

A `parallel` directive with an `if` clause and associated structured block in C:

```
#pragma omp parallel if( length > threshold )  
{  
    statement0;  
    statement1;  
    statement2;  
}
```

A `parallel` directive with a `num_threads` clause and associated structured block in Fortran:

```
!$omp parallel num_threads( 64 )  
statement1  
statement2  
statement3  
!$omp end parallel
```

CONTROLLING THE *nthreads-var* ICV

omp_set_num_threads API Routine [OpenMP-4.5, 3.2.1]

```
⌋ void omp_set_num_threads(int num_threads);
```

```
FO8 subroutine omp_set_num_threads(num_threads)  
integer num_threads
```

Sets the ICV that controls the number of threads to fork for parallel regions (without num_threads clause) encountered subsequently.

omp_get_max_threads API Routine [OpenMP-4.5, 3.2.3]

```
⌋ int omp_get_max_threads(void);
```

```
FO8 integer function omp_get_max_threads()
```

Queries the ICV that controls the number of threads to fork.

THREAD LIMIT & DYNAMIC ADJUSTMENT

omp_get_thread_limit API Routine [OpenMP-4.5, 3.2.14]

```
⌋ int omp_get_thread_limit(void);
```

```
FO8 integer function omp_get_thread_limit()
```

Upper bound on the number of threads used in a program.

omp_get_dynamic and omp_set_dynamic API Routines [OpenMP-4.5, 3.2.7, 3.2.8]

```
⌋ int omp_get_dynamic(void);  
⌋ void omp_set_dynamic(int dynamic);
```

```
FO8 logical function omp_get_dynamic()  
      subroutine omp_set_dynamic(dynamic)  
FO8 logical dynamic
```

Enable or disable dynamic adjustment of the number of threads.

INSIDE OF A PARALLEL REGION?

omp_in_parallel API Routine [OpenMP-4.5, 3.2.6]

```
⌋ int omp_in_parallel(void);
```

```
⌋ logical function omp_in_parallel()
```

Is this code being executed as part of a parallel region?

EXERCISES

Exercise 10 – Controlling parallel	10.1 Controlling the Number of Threads
	Use <code>hello_omp.{c c++ f90}</code> to play around with the various ways to set the number of threads forked for a <code>parallel</code> region: • The <code>OMP_NUM_THREADS</code> environment variable • The <code>omp_set_num_threads</code> API routine • The <code>num_threads</code> clause • The <code>if</code> clause Inspect the number of threads that are actually forked using <code>omp_get_num_threads</code>.
	10.2 Limits of the OpenMP Implementation
	Determine the maximum number of threads allowed by the OpenMP implementation you are using and check whether it supports dynamic adjustment of the number of threads.

DATA-SHARING ATTRIBUTES [OpenMP-4.5, 2.15.1]

Variable

A named data storage block, for which the value can be defined and redefined during the execution of a program.

Private Variable

With respect to a given set of *task regions* that bind to the same *parallel region*, a *variable* for which the name provides access to a **different** block of storage for each *task region*.

Shared Variable

With respect to a given set of *task regions* that bind to the same *parallel region*, a *variable* for which the name provides access to the **same** block of storage for each *task region*.

DATA-SHARING ATTRIBUTE RULES

Inside a construct [OpenMP-4.5, 2.15.1]

The rules that determine the data-sharing attributes of variables referenced from the inside of a construct fall into one of the following categories:

Pre-determined

- Variables with automatic storage duration declared inside the construct are private (C and C++)
- Variables with dynamic and static storage duration are shared (C and C++)
- Loop iteration variables are private (Fortran)
- Implied-do indices and **forall** indices are private (Fortran)
- Assumed-size arrays are shared (Fortran)

Explicit

Data-sharing attributes are determined by explicit clauses on the respective constructs.

Implicit

If the data-sharing attributes are neither pre-determined nor explicitly determined, they fall back to the attribute determined by the default clause, or shared if no default clause is present.

DATA-SHARING ATTRIBUTE RULES

Inside a region, but not a construct [OpenMP-4.5, 2.15.2]

The data-sharing attributes of variables inside regions, not constructs, are governed by simpler rules:

- Static variables (C and C++) and variables with the **save** attribute (Fortran) are shared
- File-scope (C and C++) or namespace-scope (C++) variables and common blocks or variables accessed through use or host association (Fortran) are shared
- Objects with dynamic storage duration are shared (C and C++)
- Static data members are shared (C++)
- Arguments passed by reference have the same data-sharing attributes as the variable they are referencing (C++ and Fortran)
- Implied-do indices, **forall** indices are private (Fortran)
- Local variables are private

THE SHARED CLAUSE [OpenMP-4.5, 2.15.3.2]

```
* shared(list)
```

- Declares the listed variables to be shared.
- The programmer must ensure that shared variables are alive while they are shared.
- Shared variables must not be part of another variable (i.e. array or structure elements).

THE PRIVATE CLAUSE [OpenMP-4.5, 2.15.3.3]

```
* private(list)
```

- Declares the listed variables to be private.
- All threads have their own new versions of these variables.
- Private variables must not be part of another variable.
- If private variables are of class type, a default constructor must be accessible. (C++)
- The type of a private variable must not be **const**-qualified, incomplete or reference to incomplete. (C and C++)
- Private variables must either be definable or allocatable. (Fortran)
- Private variables must not appear in **namelist** statements, variable format expressions or expressions for statement function definitions. (Fortran)
- Private variables must not be pointers with **intent** (in). (Fortran)

FIRSTPRIVATE CLAUSE [OpenMP-4.5, 2.15.3.4]

```
* firstprivate(list)
```

Like private, but initialize the new versions of the variables to have the same value as the variable that exists before the construct.

- Non-array variables are initialized by copy assignment (C and C++)
- Arrays are initialize by element-wise assignment (C and C++)
- Copy constructors are invoked if present (C++)
- Non-**pointer** variables are initialized by assignment or not associated if the original variable is not associated (Fortran)
- **pointer** variables are initialized by pointer assignment (Fortran)

DEFAULT CLAUSE [OpenMP-4.5, 2.15.3.1]

C and C++

```
↪ default(shared | none)
```

Fortran

```
↪ default(private | firstprivate | shared | none)
```

Determines the data-sharing attributes for all variables referenced from inside of a region that have neither pre-determined nor explicit data-sharing attributes.

`default(none)` forces the programmer to make data-sharing attributes explicit if they are not pre-determined. This can help clarify the programmer's intentions to someone who does not have the implicit data-sharing rules in mind.

REDUCTION CLAUSE [OpenMP-4.5, 2.15.3.6]

```
* reduction(reduction-identifier : list)
```

- Listed variables are declared private.
- At the end of the construct, the original variable is updated by combining the private copies using the operation given by `reduction-identifier`.
- `reduction-identifier` may be `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, `min` or `max` (C and C++) or an *identifier* (C) or an *id-expression* (C++)
- `reduction-identifier` may be a base language identifier, a user-defined operator, or one of `+`, `-`, `*`, `.` and `.`, `.or.`, `.eqv.`, `.neqv.`, `max`, `min`, `and`, `ior` or `ieor` (Fortran)
- Private versions of the variable are initialized with appropriate values

EXERCISES

Exercise 11 – Data-sharing Attributes

11.1 Generalized Vector Addition (axpy)

In the file `axpy.{c | c++ | f90}` add a new function/subroutine `axpy_parallel(a, x, y, z[, n])` that uses multiple threads to perform a generalized vector addition. Modify the main part of the program to have your function/subroutine tested.

Hints:

- Use the `parallel` construct and the necessary clauses to define an appropriate data environment.
- Use `omp_get_thread_num()` and `omp_get_num_threads()` to decompose the work.

THREAD SYNCHRONIZATION

- In MPI, exchange of data between processes implies synchronization through the message metaphor.
- In OpenMP, threads exchange data through shared parts of memory.
- Explicit synchronization is needed to coordinate access to shared memory.

Data Race

A data race occurs when

- multiple threads write to the same memory unit without synchronization or
 - at least one thread writes to and at least one thread reads from the same memory unit without synchronization.
- Data races result in unspecified program behavior.
 - OpenMP offers several synchronization mechanism which range from high-level/general to low-level/specialized.

THE BARRIER CONSTRUCT [OpenMP-4.5, 2.13.3]

```
C #pragma omp barrier
```

```
FO8 !$omp barrier
```

- Threads are only allowed to continue execution of code after the barrier once all threads in the current team have reached the barrier.
- A barrier region must be executed by all threads in the current team or none.

THE CRITICAL CONSTRUCT [OpenMP-4.5, 2.13.2]

```
C    #pragma omp critical [(name)]  
    structured-block
```

```
FO8  !$omp critical [(name)]  
    structured-block  
FO8  !$omp end critical [(name)]
```

- Execution of `critical` regions with the same *name* are restricted to one thread at a time.
- *name* is a compile time constant.
- In C, *names* live in their own name space.
- In Fortran, *names* of critical regions can collide with other identifiers.

LOCK ROUTINES [OpenMP-4.5, 3.3]

```
void omp_init_lock(omp_lock_t* lock);  
void omp_destroy_lock(omp_lock_t* lock);  
void omp_set_lock(omp_lock_t* lock);  
  void omp_unset_lock(omp_lock_t* lock);
```

```
subroutine omp_init_lock(svar)  
subroutine omp_destroy_lock(svar)  
subroutine omp_set_lock(svar)  
subroutine omp_unset_lock(svar)  
FO8 integer(kind = omp_lock_kind) :: svar
```

- Like critical sections, but identified by runtime value rather than global name
- Initialize a lock before first use
- Destroy a lock when it is no longer needed
- Lock and unlock using the set and unset routines
- set blocks if lock is already set

THE ATOMIC AND FLUSH CONSTRUCTS

[OpenMP-4.5, 2.13.6, 2.13.7]

- `barrier`, `critical`, and `locks` implement synchronization between general blocks of code
- If blocks become very small, synchronization overhead could become an issue
- The `atomic` and `flush` constructs implement low-level, fine grained synchronization for certain limited operations on scalar variables:
 - `read`
 - `write`
 - `update`, writing a new value based on the old value
 - `capture`, like `update` and the old or new value is available in the subsequent code
- Correct use requires knowledge of the OpenMP Memory Model [OpenMP-4.5, 1.4]
- See also: C11 and C++11 Memory Models

EXERCISES

Exercise 12 – Thread Synchronization	12.1 Dot Product
	In the file <code>dot.{c c++ f90}</code> add a new function/subroutine <code>dot_parallel(x, y[, n])</code> that uses multiple threads to perform the dot product. Do not use the <code>reduction</code> clause. Modify the main part of the program to have your function/subroutine tested. Hint: • Decomposition of the work load should be similar to the last exercise • Partial results of different threads should be combined in a shared variable • Use a suitable synchronization mechanism to coordinate access
	Bonus
	Use the <code>reduction</code> clause to simplify your program.

Part XIV: Worksharing

WORKSHARING CONSTRUCTS

- Decompose work for concurrent execution by multiple threads
- Used inside parallel regions
- Available worksharing constructs:
 - single and sections construct
 - loop construct
 - workshare construct
 - task worksharing

THE SINGLE CONSTRUCT [OpenMP-4.5, 2.7.3]

```
C  #pragma omp single [clause[[,] clause]...]  
    structured-block
```

```
F08 !$omp single [clause[[,] clause]...]  
    structured-block  
F08 !$omp end single [end_clause[[,] end_clause]...]
```

- The *structured block* is executed by a single thread in the encountering team.
- Permissible clauses are `firstprivate`, `private`, `copyprivate` and `nowait`.
- `nowait` and `copyprivate` are `end_clauses` in Fortran.

IMPLICIT BARRIERS & THE NOWAIT CLAUSE

[OpenMP-4.5, 2.7]

- Worksharing constructs (and the `parallel` construct) contain an implied barrier at their exit.
- The `nowait` clause can be used on worksharing constructs to disable this implicit barrier.

```
#pragma omp single
    work(); // work() is executed on a single thread
// All other threads wait until execution of work() is
↳ completed
```

```
#pragma omp single nowait
    work(); // work() is executed on a single thread
// All other threads are free to continue with something
↳ else
```

THE COPYPRIVATE CLAUSE

[OpenMP-4.5, 2.15.4.2]

```
* copyprivate(list)
```

- *list* contains variables that are *private* in the enclosing parallel region.
- At the end of the *single* construct, the values of all *list* items on the single thread are copied to all other threads.
- E.g. serial initialization
- *copyprivate* cannot be combined with *wait*.

LOOP CONSTRUCT [OpenMP-4.5, 2.7.1]

```
#pragma omp for [clause[[,] clause]...]
    for-loops
```

```
!$omp do [clause[[,] clause]...]
    do-loops
FOR [!$omp end do [nowait]]
```

Declares the iterations of a loop to be suitable for concurrent execution on multiple threads.
Data-environment clauses

- private
- firstprivate
- lastprivate
- reduction

Loop-specific clauses

- schedule
- collapse

CANONICAL LOOP FORM [OpenMP-4.5, 2.6]

In C and C++ the for-loops must have the following form:

```
for ([type] var = lb; var relational-op b; incr-expr)  
    ↪ structured-block
```

- var can be an integer, a pointer, or a random access iterator
- incr-expr increments (or decrements) var, e.g. var = var + incr
- The increment incr must not change during execution of the loop
- var must not be modified by the loop body
- The number of iterations of the loop must be known beforehand

THE COLLAPSE CLAUSE [OpenMP-4.5, 2.7.1]

* collapse(n)

- The loop directive applies to the outermost loop of a set of nested loops, by default
- collapse(n) extends the scope of the loop directive to the n outer loops
- All associated loops must be perfectly nested, i.e.:

```
for (int i = 0; i < N; ++i) {  
    for (int j = 0; j < M; ++j) {  
        // ...  
    }  
}
```

THE SCHEDULE CLAUSE [OpenMP-4.5, 2.7.1]

```
* schedule(kind[, chunk_size])
```

Determines how the iteration space is divided into chunks and how these chunks are distributed among threads.

static Divide iteration space into chunks of `chunk_size` iterations and distribute them in a round-robin fashion among threads. If `chunk_size` is not specified, chunk size is chosen such that each thread gets at most one chunk.

dynamic Divide into chunks of size `chunk_size` (defaults to 1). When a thread is done processing a chunk it acquires a new one.

guided Like dynamic but chunk size is adjusted, starting with large sizes for the first chunks and decreasing to `chunk_size` (default 1).

auto Let the compiler and runtime decide.

runtime Schedule is chosen based on ICV *run-sched-var*.

EXERCISES

Exercise 13 – Loop Worksharing

13.1 Generalized Vector Addition (axpy)

In the file `axpy.{c|c++|f90}` add a new function/subroutine `axpy_parallel_for(a, x, y, z[, n])` that uses loop worksharing to perform the generalised vector addition.

13.2 Dot Product

In the file `dot.{c|c++|f90}` add a new function/subroutine `dot_parallel_for(x, y[, n])` that uses loop worksharing to perform the dot product.

Caveat: Make sure to correctly synchronize access to the accumulator variable.

WORKSHARE [OpenMP-4.5, 2.7.4]

Fortran only

```
!$omp workshare  
    structured-block  
!$omp end workshare [nowait]
```

The structured block may contain:

- array assignments
- scalar assignments
- **forall** constructs
- **where** statements and constructs
- `atomic`, `critical` and `parallel` constructs

Where possible, these are decomposed into independent units of work and executed in parallel.

EXERCISES

Exercise 14 – workshare Construct

14.1 Generalized Vector Addition (axpy)

In the file `axpy.f90` add a new subroutine `axpy_parallel_workshare(a, x, y, z)` that uses the `workshare` construct to perform the generalized vector addition.

14.2 Dot Product

In the file `dot.f90` add a new function `dot_parallel_workshare(x, y)` that uses the `workshare` construct to perform the dot product.

Caveat: Make sure to correctly synchronize access to the accumulator variable.

COMBINED CONSTRUCTS [OpenMP-4.5, 2.11]

Some constructs that often appear as nested pairs can be combined into one construct, e.g.

```
#pragma omp parallel
#pragma omp for
for (...; ...; ...) {
    ...
}
```

can be turned into

```
#pragma omp parallel for
for (...; ...; ...) {
    ...
}
```

Similarly, `parallel` and `workshare` can be combined.
Combined constructs usually accept the clauses of either of the base constructs.

Part XV: Task Worksharing

TASK TERMINOLOGY

Task

A specific instance of executable code and its *data environment*, generated when a *thread* encounters a task, taskloop, parallel, target or teams *construct*.

Child Task

A *task* is a *child task* of its generating *task region*. A *child task region* is not part of its generating *task region*.

Descendent Task

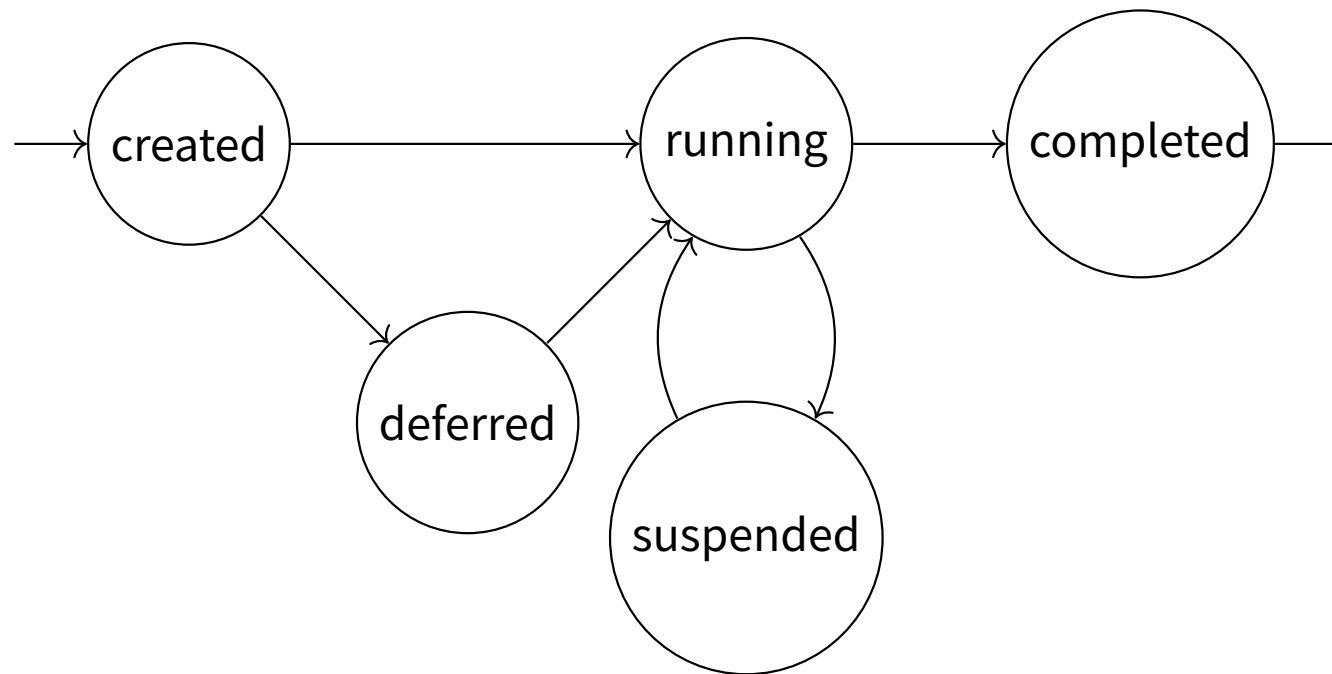
A *task* that is the *child task* of a *task region* or of one of its *descendent task regions*.

Sibling Task

Tasks that are *child tasks* of the same *task region*.

TASK LIFE-CYCLE

- Execution of tasks can be *deferred* and *suspended*
- Scheduling is done by the OpenMP runtime system at *scheduling points*
- Scheduling decisions can be influenced by e.g. *task dependencies* and *task priorities*



THE TASK CONSTRUCT [OpenMP-4.5, 2.9.1]

```
#pragma omp task [clause[[,] clause]...]  
    structured-block
```

```
!$omp task [clause[[,] clause]...]  
    structured-block  
!$omp end task
```

Creates a task. Execution of the task may commence immediately or be deferred.

Data-environment clauses

- private
- firstprivate
- shared

Task-specific clauses

- if
- final
- tied
- mergeable
- depend
- priority

TASK DATA-ENVIRONMENT [OpenMP-4.5, 2.15.1.1]

The rules for implicitly determined data-sharing attributes of variables referenced in task generating constructs are slightly different from other constructs:

If no default clause is present and

- the variable is shared by all implicit tasks in the enclosing context, it is also shared by the generated task,
- otherwise, the variable is firstprivate.

THE IF CLAUSE [OpenMP-4.5, 2.9.1]

```
* if([task: ] scalar-expression)
```

If the scalar expression evaluates to *false*:

- Execution of the current task
 - is suspended and
 - may only be resumed once the generated task is complete
- Execution of the generated task may commence immediately

Undeferred Task

A *task* for which execution is not deferred with respect to its generating *task region*. That is, its generating *task region* is suspended until execution of the *undeferred task* is completed.

THE FINAL CLAUSE [OpenMP-4.5, 2.9.1]

```
* final(scalar-expression)
```

If the scalar expression evaluates to *true* all *descendent tasks* of the generated task are

- *undelayed* and
- executed immediately.

Final Task

A *task* that forces all of its *child tasks* to become *final* and *included tasks*.

Included Task

A *task* for which execution is sequentially included in the generating *task region*. That is, an *included task* is *undelayed* and executed immediately by the *encountering thread*.

THE UNTIED CLAUSE [OpenMP-4.5, 2.9.1]

* untied

- The generated task is *untied* meaning it can be suspended by one thread and resume execution on another.
- By default, tasks are generated as *tied* tasks.

Untied Task

A *task* that, when its *task region* is suspended, can be resumed by any *thread* in the team. That is, the *task* is not tied to any *thread*.

Tied Task

A *task* that, when its *task region* is suspended, can be resumed only by the same *thread* that suspended it. That is, the *task* is tied to that *thread*.

THE MERGEABLE CLAUSE [OpenMP-4.5, 2.9.1]

* mergeable

If a *mergeable task* is also either *undelayed* or *included* it is *merged* into the current task, i.e. no *data environment* and set of ICVs is created for it.

Mergeable Task

A task that may be a *merged task* if it is an *undelayed task* or an *included task*.

Merged Task

A task for which the *data environment*, inclusive of ICVs, is the same as that of its generating *task region*.

THE PRIORITY CLAUSE [OpenMP-4.5, 2.9.1]

* `priority(priority-value)`

- *priority-value* is a scalar non-negative numerical value
- Priority influences the order of task execution
- Among tasks that are ready for execution, those with a higher priority are more likely to be executed next

THE DEPEND CLAUSE [OpenMP-4.5, 2.13.9]

```
depend(in: list)
depend(out: list)
* depend(inout: list)
```

- *list* contains storage locations
- A task with a dependence on *x*, `depend(in: x)`, has to wait for completion of previously generated *sibling tasks* with `depend(out: x)` or `depend(inout: x)`
- A task with a dependence `depend(out: x)` or `depend(inout: x)` has to wait for completion of previously generated *sibling tasks* with any kind of dependence on *x*
- *in*, *out* and *inout* correspond to intended read and/or write operations to the listed variables.

Dependent Task

A *task* that because of a *task dependence* cannot be executed until its *predecessor tasks* have completed.

TASK SCHEDULING POLICY [OpenMP-4.5, 2.9.5]

The task scheduler of the OpenMP runtime environment becomes active at *task scheduling points*. It may then

- begin execution of a task or
- resume execution of untied tasks or tasks tied to the current thread.

Task scheduling points

- generation of an explicit task
- task completion
- `taskyield` regions
- `taskwait` regions
- the end of `taskgroup` regions
- implicit and explicit `barrier` regions

THE TASKYIELD CONSTRUCT [OpenMP-4.5, 2.9.4]

```
┌ #pragma omp taskyield
```

```
└ !$omp taskyield
```

- Notifies the scheduler that execution of the current task may be suspended at this point in favor of another task
- Inserts an explicit scheduling point

THE TASKWAIT & TASKGROUP CONSTRUCTS

[OpenMP-4.5, 2.13.4, 2.13.5]

```
┌ #pragma omp taskwait
```

```
FO8 !$omp taskwait
```

Suspends the current task until all *child tasks* are completed.

```
┌ #pragma omp taskgroup  
  structured-block
```

```
FO8 !$omp taskgroup  
    structured-block  
FO8 !$omp end taskgroup
```

The current task is suspended at the end of the taskgroup region until all *descendent tasks* generated within the region are completed.

EXERCISES

Exercise 15 – Task worksharing	15.1 Generalized Vector Addition (axpy)
	In the file <code>axpy.{c c++ f90}</code> add a new function/subroutine <code>axpy_parallel_task(a, x, y, z[, n])</code> that uses task worksharing to perform the generalized vector addition.
	15.2 Dot Product
	In the file <code>dot.{c c++ f90}</code> add a new function/subroutine <code>dot_parallel_task(x, y[, n])</code> that uses task worksharing to perform the dot product. Caveat: Make sure to correctly synchronize access to the accumulator variable.

Part XVI: Wrap-up

ALTERNATIVES

Horizontal Alternatives

Parallel languages Fortran Coarrays, UPC; Chapel, Fortress, X10

Parallel frameworks Charm++, HPX, StarPU

Shared Memory Tasking Cilk, TBB

Accelerators CUDA, OpenCL, OpenACC, OmpSs

Platform Solutions PLINQ, GCD, `java.util.concurrent`

Vertical Alternatives

Applications Gromacs, CP2K, ANSYS, OpenFOAM

Numerics libraries PETSc, Trilinos, DUNE, FEniCS

Big data frameworks Hadoop, Spark

JSC COURSE PROGRAMME

- Introduction to the usage and programming of supercomputer resources in Jülich, **28 – 29 May**
- Advanced Parallel Programming with MPI and OpenMP, **26 – 28 November**
- Introduction to GPU programming using OpenACC, **Fall 2018**
- GPU Programming with CUDA, **23 – 25 April**
- From zero to hero: Understanding and fixing intra-node performance bottlenecks, **11 – 12 April**
- And more, see http://www.fz-juelich.de/ias/jsc/EN/Expertise/Workshops/Courses/courses_node.html