



# INTRODUCTION TO PARALLEL PROGRAMMING WITH MPI AND OPENMP

August 12–16 2019 | Benedikt Steinbusch | Jülich Supercomputing Centre



# INTRODUCTION TO PARALLEL PROGRAMMING WITH MPI AND OPENMP

Benedikt Steinbusch | Jülich Supercomputing Centre | August 12–16 2019

## CONTENTS

### I Fundamentals of Parallel Computing

- 1 Motivation
- 2 Hardware
- 3 Software
- 4 Exercises

### II First Steps with MPI

- 5 What is MPI?
- 6 Terminology
- 7 Infrastructure
- 8 Basic Program Structure
- 9 First Steps on a Supercomputer
- 10 Exercises

### III Blocking Point-to-Point Communication

- 11 Introduction
- 12 Sending
- 13 Receiving
- 14 Communication Modes
- 15 Semantics
- 16 Pitfalls
- 17 Exercises

### IV Nonblocking Point-to-Point Communication

- 18 Introduction
- 19 Start
- 20 Completion
- 21 Remarks
- 22 Exercises

### V Blocking Collective Communication

- 23 Introduction
- 24 Synchronization
- 25 Data Movement
- 26 Reductions
- 27 In Place Mode
- 28 Variants

### VI Nonblocking Collective Communication

- 3 Introduction
- 3 Examples
- 3 Exercises

### VII Derived Datatypes

- 5 Introduction
- 5 Constructors
- 6 Address Calculation
- 6 Padding
- 7 Exercises

### VIII Input/Output

- 9 Introduction
- 10 File Manipulation
- 11 File Views
- 11 Data Access
- 12 Consistency
- 12 Exercises

### IX Tools

- 14 MUST
- 14 Exercises

### X Communicators

41

45	Introduction	41	75	The task Construct	58
46	Constructors	41	76	task Clauses	58
47	Accessors	42	77	Task Scheduling	59
48	Destructors	44	78	Task Synchronization	60
49	Exercises	44	79	Exercises	60
<b>XI</b>	<b>Thread Compliance</b>	<b>44</b>	<b>XVI</b>	<b>Wrap-up</b>	<b>61</b>
50	Introduction	44	<b>XVII</b>	<b>Tutorial</b>	<b>61</b>
51	Enabling Thread Support	44			
52	Matching Probe and Receive	45			
53	Remarks	45			
<b>XII</b>	<b>First Steps with OpenMP</b>	<b>46</b>			
54	What is OpenMP?	46			
55	Terminology	46			
56	Infrastructure	47			
57	Basic Program Structure	47			
58	Exercises	48			
<b>XIII</b>	<b>Low-Level OpenMP Concepts</b>	50			
59	Introduction	50			
60	Exercises	50			
61	Data Environment	52			
62	Exercises	52			
63	Thread Synchronization	53			
64	Exercises	54			
<b>XIV</b>	<b>Worksharing</b>	55			
65	Introduction	55			
66	The single construct	55			
67	single Clauses	56			
68	The loop construct	56			
69	loop Clauses	57			
70	Exercises	57			
71	workshare Construct	57			
72	Exercises	57			
73	Combined Constructs	57			
<b>XV</b>	<b>Task Worksharing</b>	<b>57</b>			
74	Introduction	58			

## PART I

# FUNDAMENTALS OF PARALLEL COMPUTING

## 1 MOTIVATION

### PARALLEL COMPUTING

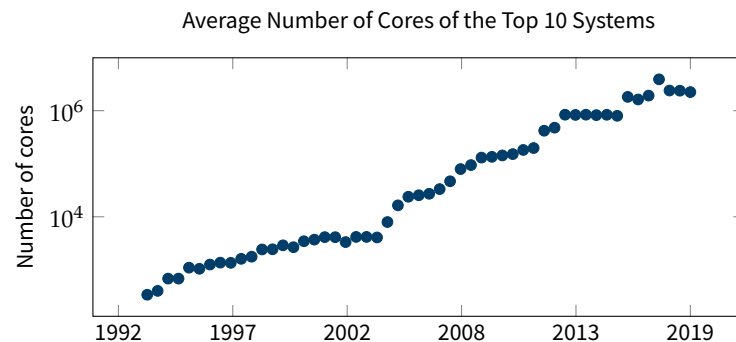
Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. (Wikipedia<sup>1</sup>)

### 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

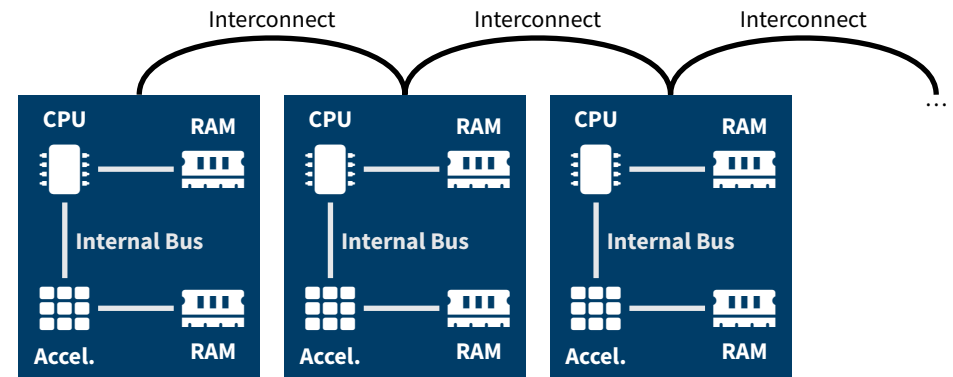
### PARALLELISM IN THE TOP 500 LIST



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

## 2 HARDWARE

### A MODERN SUPERCOMPUTER



### 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

#### **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

#### *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

## **3 SOFTWARE**

### **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 active send and active 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)

## 4 EXERCISES

### Exercise 1 – Access to JURECA

#### 1.1 Generate an SSH authentication key

1. Log in to the PC in front of you (see paper slip for credentials)
2. Open a terminal and enter the following command into your shell:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key
  ( /home/trainXXX/.ssh/id_rsa ):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in
  ( /home/trainXXX/.ssh/id_rsa ).
Your public key has been saved in
  ( /home/trainXXX/.ssh/id_rsa.pub ).
The key fingerprint is:
SHA256:7JG38qQM5cFtJ9LCdbZxhYtKZAFfFZ4+AjVqTzmR+s
  ( trainXXX@zamYYYY )
```

#### 1.2 Upload the key to JuDoor

1. Print your public key to the terminal then select it and copy it to the clipboard (Ctrl-C):

```
$ cat ~/.ssh/id_rsa.pub
```

2. Log in to JuDoor (<https://judoor.fz-juelich.de/>).
3. Click “Manage SSH-keys” next to the entry “jureca” in the list of systems.
4. Paste your public key into the text field and submit (Ctrl-V).

#### 1.3 Add the key to the key agent

Enter the following command into your shell:

```
$ ssh-add
Enter passphrase for /home/trainXXX/.ssh/id_rsa:
Identity added: /home/trainXXX/.ssh/id_rsa (trainXXX@zamYYYY)
```

## PART II

# FIRST STEPS WITH MPI

## 5 WHAT IS MPI?

MPI (**M**essage-**P**assing **I**nterface) is a message-passing library interface specification. [...] MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. (MPI Forum<sup>2</sup>)

- 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)

### 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** ✓

<sup>2</sup>Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 3.1. June 4, 2015. URL: <https://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.

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: <https://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
- <https://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

## 6 TERMINOLOGY

### PROCESS ORGANIZATION [MPI-3.1, 6.2]



#### Terminology: Process

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

#### Terminology: Rank

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

#### Terminology: Group

An ordered set of process identifiers

#### Terminology: Context

A property that allows the partitioning of the communication space

#### Terminology: Communicator

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

### OBJECTS [MPI-3.1, 2.5.1]

#### Terminology: 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.

#### Terminology: 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.

#### Terminology: 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.

## 7 INFRASTRUCTURE

### COMPILING & LINKING [MPI-3.1, 17.1.7]

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

#### C Compiler Wrappers

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

#### Fortran Compiler Wrappers

```
$ # Generic compiler wrapper shipped with e.g. OpenMPI
$ mpifort foo.f90 -o foo
$ # Vendor specific wrapper for IBM's XL Fortran compiler on BG/Q
$ bxcclf90 foo.f90 -o foo
```

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

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

### 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

```
FO8 use mpi_f08
```

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

```
FO use mpi
```

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

```
F77 include 'mpi.f.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 `ierror` 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

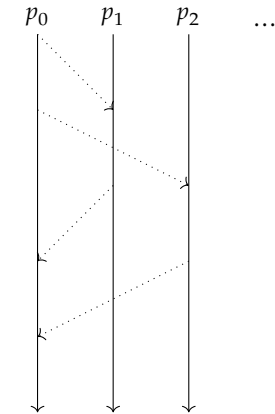
**Java** mpiJava, MPJ, MPJ Express

And many others, ask your favorite search engine.

# 8 BASIC PROGRAM STRUCTURE

## 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

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

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

Exception (can be used before initialization)

```
C int MPI_Initialized(int* flag)
```

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

## PREDEFINED COMMUNICATORS

After MPI\_Init has been called, MPI\_COMM\_WORLD is a valid handle to a predefined communicator that includes all processes available for communication. Additionally, the handle MPI\_COMM\_SELF is a communicator that is valid on each process and contains only the process itself.

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

```
F08 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

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

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

### Examples

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

```
F08 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

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

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

### Examples

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

```
F08 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

# 9 FIRST STEPS ON A SUPERCOMPUTER

## LOGIN & PROGRAMMING ENVIRONMENT

### JURECA Login

1. In a terminal on your PC, enter:

```
$ ssh name1@jureca.fz-juelich.de
```

2. Load modules and activate the training project:

```
$ module load intel-para
$ jutil env activate -p training1918 -A training1918
```

### Course Material

Copy the course material by running:

```
$ $PROJECT/material/copy.sh
```

### MPI Infrastructure

```
C
```

```
$ mpicc
```

```
Fortran
```

```
$ mpif90
```

```
C++
```

```
$ mpicxx
```

```
Process startup
```

```
$ srun -n <numprocs> <program>
```

## 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. Start applications with n processes

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

### Batch Mode

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 --time=00:05:00
module load intel-para
srun <application>
```

## USEFUL COMMANDS ON JURECA

Command	Description
squeue -u <user-id>	Shows the status of jobs
scancel <job-id>	Aborts the job with ID <job-id>
scontrol show job <job-id>	Show detailed information about a pending, running or recently completed job
watch <command>	Executes <command> every 2 seconds and shows the output

## 10 EXERCISES

### EXERCISE STRATEGIES

#### Solving

- Do not have to solve all exercises, one per section would be good
- Exercise description tells you what MPI functions/OpenMP directives to use
- Work in pairs on harder exercises
- If you get stuck
  - ask us
  - work on the half solution
  - peek at solution
- Makefile is included

## Solutions

- Skeletons in exercises/{C|C++|Fortran}/: Almost empty, you add MPI/OpenMP and most of the algorithm
- Half solutions in exercises/{C|C++|Fortran}/half-solutions: Most of the solution is there, you add MPI/OpenMP
- Solutions in exercises/{C|C++|Fortran}/solutions: Fully solved exercises, if you are completely stuck or for comparison

## EXERCISES

### Exercise 2 – First Steps

#### 2.1 Output of Ranks

Write a program `print_rank`. {c|c++|f90} in C/C++ or Fortran that has each process printing its rank.

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

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

#### 2.2 Output of ranks and total number of processes

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

```
I am process 0 of 4
I am process 1
I am process 2
I am process 3
```

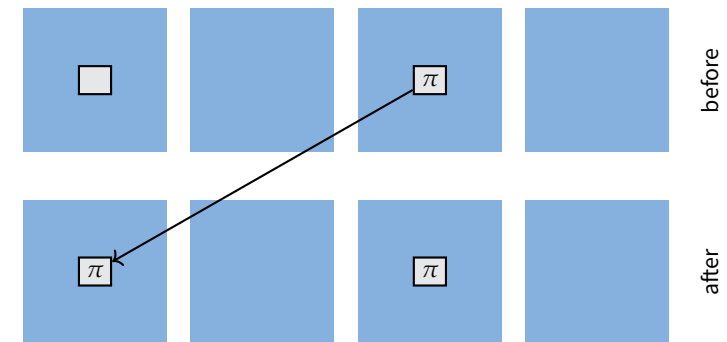
**Use:** `MPI_Comm_size`

## PART III

# BLOCKING POINT-TO-POINT COMMUNICATION

## 11 INTRODUCTION

### MESSAGE PASSING



### BLOCKING & NONBLOCKING PROCEDURES

#### Terminology: 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.

#### Terminology: 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
  - Caution: 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

## 12 SENDING

### 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)
```

```

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:

Terminology: Envelope

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

Terminology: 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]

Terminology: Data Type

Describes the structure of a piece of data

Terminology: Basic Data Types

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

C type	MPI basic data type
<b>signed int</b>	MPI_INT
<b>float</b>	MPI_FLOAT
<b>char</b>	MPI_CHAR
...	
Fortran type	MPI basic data type
<b>integer</b>	MPI_INTEGER
<b>real</b>	MPI_REAL
<b>character</b>	MPI_CHARACTER
...	

Terminology: 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]

Terminology: 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.

Terminology: 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.

Terminology: Packed data

See [MPI-3.1, 4.2]

## 13 RECEIVING

### 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
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
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
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)
```

```
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)

## 14 COMMUNICATION MODES

#### 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.

## 15 SEMANTICS

### 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.

## 16 PITFALLS

### DEADLOCK

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

Process 0
<b>call</b> MPI_Ssend(..., 1, ...)
<b>call</b> MPI_Recv(..., 1, ...)
Process 1
<b>call</b> MPI_Ssend(..., 0, ...)
<b>call</b> MPI_Recv(..., 0, ...)

#### Mitigation Strategies

- Changing communication structure (e.g. checkerboard)
- Using MPI\_Sendrecv
- Using nonblocking routines

## 17 EXERCISES

### Exercise 3 – Point-to-Point Communication

#### 3.1 Global Summation – Sequential

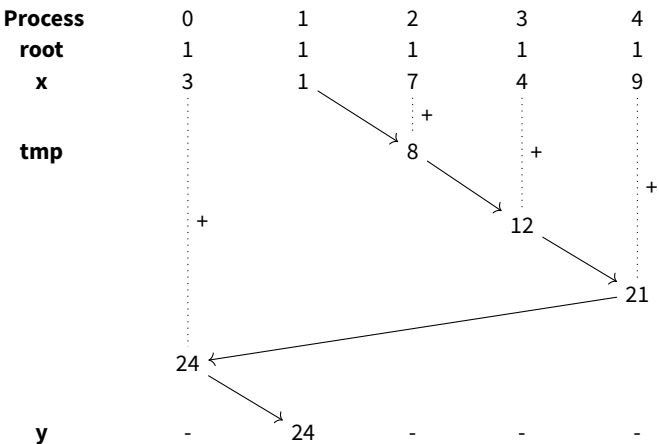
In the file `global_sum.{c | c++ | f90}` implement the function/subroutine `global_sum_sequential(x, y, root, comm)`. It will be called by all processes on the communicator `comm` and on each one accepts an integer `x`. It should compute the global sum of all values of `x` across all processes and return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

Use the following strategy:

1. The process with rank `root` starts by sending its value of `x` to the process with the next higher rank (wrap around to rank 0 on the process with the highest rank).
2. All other processes start by receiving the partial sum from the process with the next lower rank (or from the process with the highest rank on process 0)
3. Next, they add their value of `x` to the partial result and send it to the next process.
4. The `root` process eventually receives 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.

**Use:** MPI\_Send, MPI\_Recv (and maybe MPI\_Sendrecv)



## PART IV



# NONBLOCKING POINT-TO-POINT COMMUNICATION

## 18 INTRODUCTION

### 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

## 19 START

### INITIATION ROUTINES [MPI-3.1, 3.7.2]

#### Send

**Synchronous** MPI\_Issend

**Standard** MPI\_Isend

**Buffered** MPI\_Ibsend

**Ready** MPI\_Irsend

#### Receive

MPI\_Irecv

#### Probe

MPI\_Iprobe

- “I” is for immediate.
- Signature is similar to blocking counterparts with additional *request* object.

- Initiate operations and relinquish access rights to any buffer involved.

### NONBLOCKING SEND [MPI-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>
```

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

```
F08 MPI_Irecv(buf, count, datatype, source, tag, comm, request,
    ierror)
type(*), dimension(..), asynchronous :: buf
integer, intent(in) :: count, source, tag
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Comm), intent(in) :: comm
type(MPI_Request), intent(out) :: request
integer, optional, intent(out) :: ierror
```

### NONBLOCKING PROBE [MPI-3.1, 3.8.1]

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

```

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

```

```

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.

## 20 COMPLETION

### WAIT [MPI-3.1, 3.7.3]

```

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

```

```

int MPI_Wait(MPI_Request *request, MPI_Status *status)

```

```

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)

```

```

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)

```

```

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)

```

```

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)

## 21 REMARKS

### 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_Irsend` must hold at start of operation
- A nonblocking operation immediately followed by a matching wait is equivalent to the blocking operation

*The Fortran Language Bindings and nonblocking operations*

- Arrays with subscript triplets (e.g. `a(1:100:5)`) can only be reliably used as buffers if the compile time constant `MPI_SUBARRAYS_SUPPORTED` equals `.true.` [MPI-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 *computation*
- 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

## 22 EXERCISES

### Exercise 4 – Nonblocking P2P Communication

#### 4.1 Global Summation – Tree

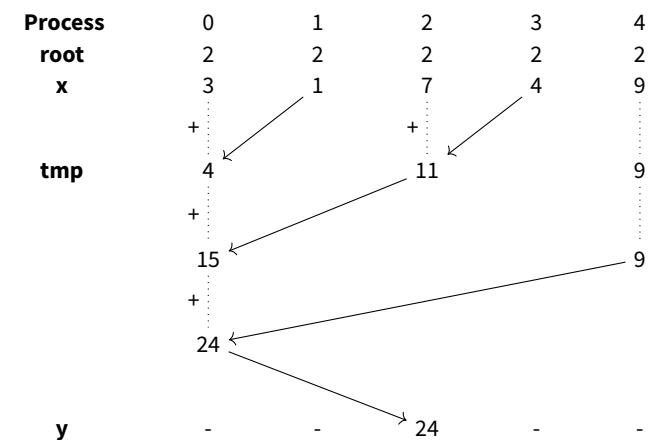
In the file `global_sum.{c|c++|f90}`, implement a function/subroutine `global_sum_tree(x, y, root, comm)` that performs the same operation as the solution to exercise 3.1.

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:
  - (a) Group processes into pairs – let us call them the left and the right process.
  - (b) The right process sends its partial result to the left process.
  - (c) The left process receives the partial result and adds it to its own one.
  - (d) 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.

**Use:** `MPI_Irecv`, `MPI_Wait`



## PART V

# BLOCKING COLLECTIVE COMMUNICATION

## 23 INTRODUCTION

### COLLECTIVE [MPI-3.1, 2.4, 5.1]

*Terminology: 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

## 24 SYNCHRONIZATION

### BARRIER [MPI-3.1, 5.3]

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

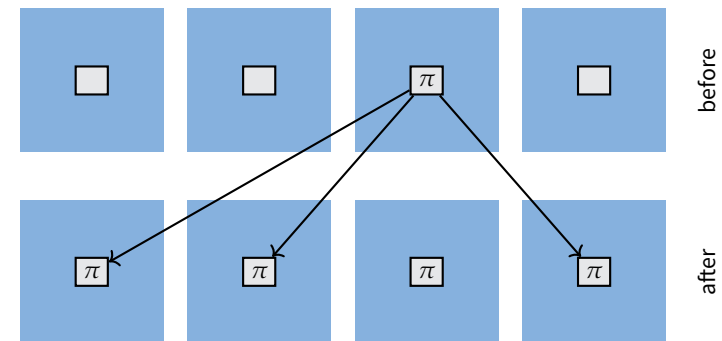
```
⌋ int MPI_Barrier(MPI_Comm comm)
```

```
⌋ 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.

## 25 DATA MOVEMENT

### BROADCAST [MPI-3.1, 5.4]

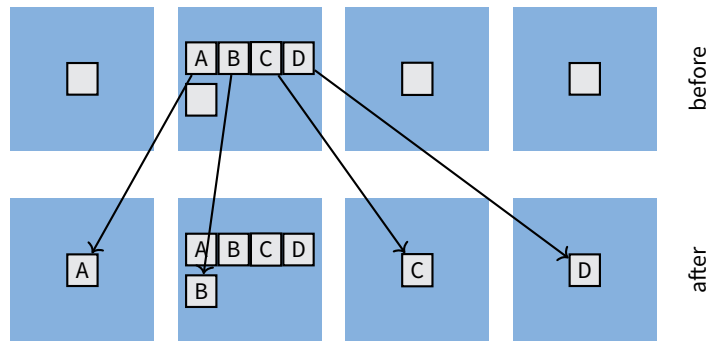


```
* MPI_Bcast( <buffer>, <root> )
```

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

```
⌋ 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]

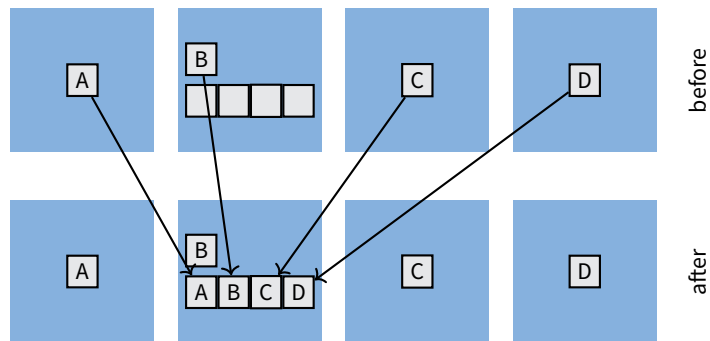


```
* MPI_Gather( <send buffer>, <receive buffer>, <root> )
```

```
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
```

#### GATHER [MPI-3.1, 5.5]

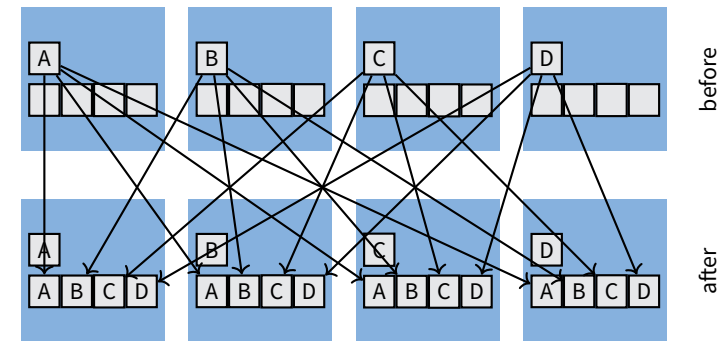


```
* MPI_Gather( <send buffer>, <receive buffer>, <root> )
```

```
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
```

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



```
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)
```

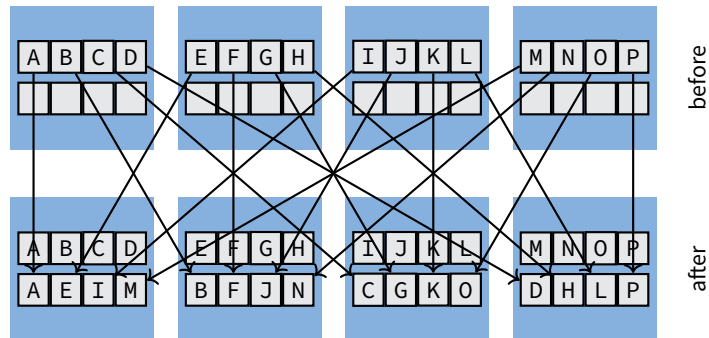
F08

```

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
integer, optional, intent(out) :: ierror

```

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



```

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
integer, optional, intent(out) :: ierror

```

F08

## 26 REDUCTIONS

### GLOBAL REDUCTION OPERATIONS [MPI-3.1, 5.9]

Associative operations over distributed data

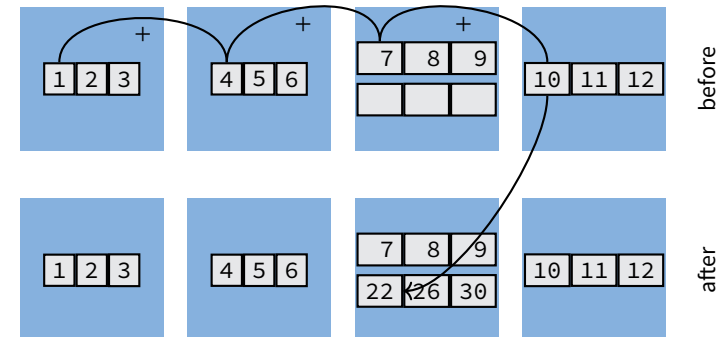
$d_0 \oplus d_1 \oplus d_2 \oplus \dots \oplus d_{n-1}$ , where  
 $d_i$ , data of process with rank  $i$   
 $\oplus$ , associative operation

Examples for  $\oplus$ :

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

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

### REDUCE [MPI-3.1, 5.9.1]



```

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)

```

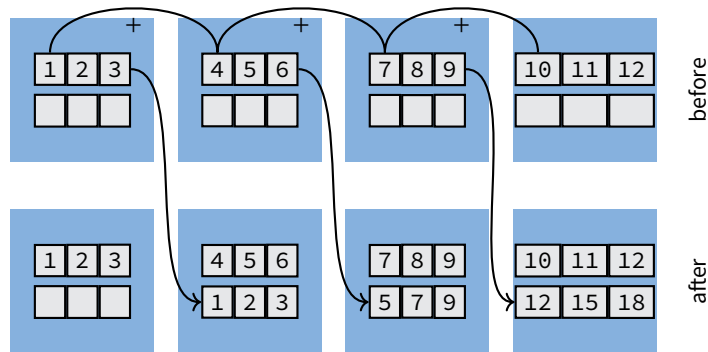
```

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

```

F08

## EXCLUSIVE SCAN [MPI-3.1, 5.11.2]



```

MPI_Exscan( <send buffer>, <receive buffer>, <operation>,
            & <communicator> )

```

\*

```

int MPI_Exscan(const void* sendbuf, void* recvbuf, int count,
                & MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

```

⌋

```

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

```

F08

## PREDEFINED OPERATIONS [MPI-3.1, 5.9.2]

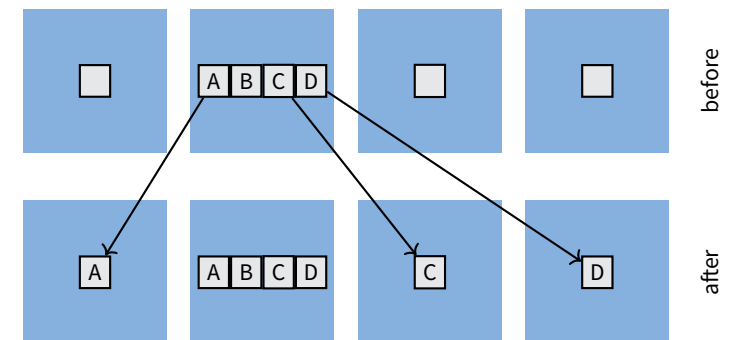
Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and the first rank that holds it [MPI-3.1, 5.9.4]
MPI_MINLOC	Minimum and the first rank that holds it [MPI-3.1, 5.9.4]

## 27 IN PLACE MODE

### IN PLACE MODE

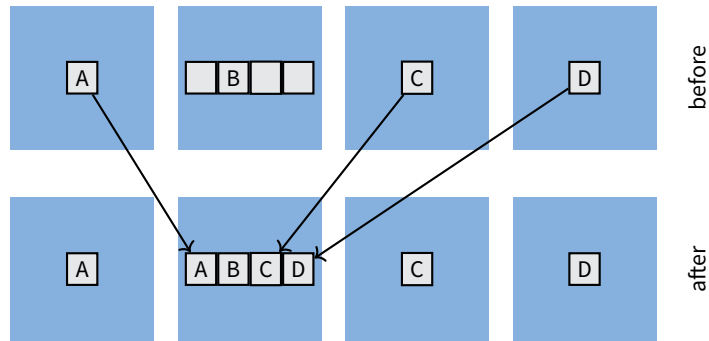
- Collectives can be used in *in place mode* with only one buffer to conserve memory
- The special value MPI\_IN\_PLACE is used in place of either the send or receive buffer address
- count and datatype of that buffer are ignored

### IN PLACE SCATTER



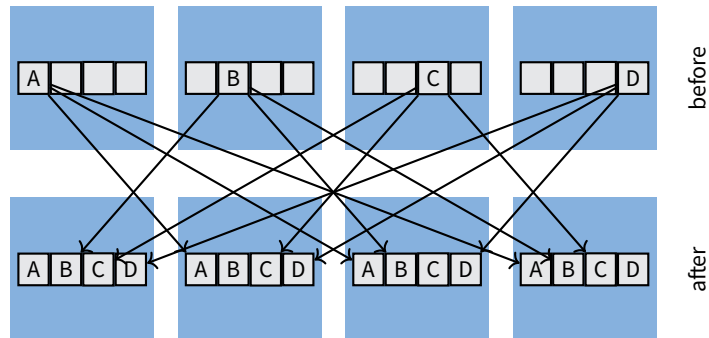
If MPI\_IN\_PLACE is used for recvbuf on the root process, recvcount and recvttype are ignored and the root process does not send data to itself

### IN PLACE GATHER



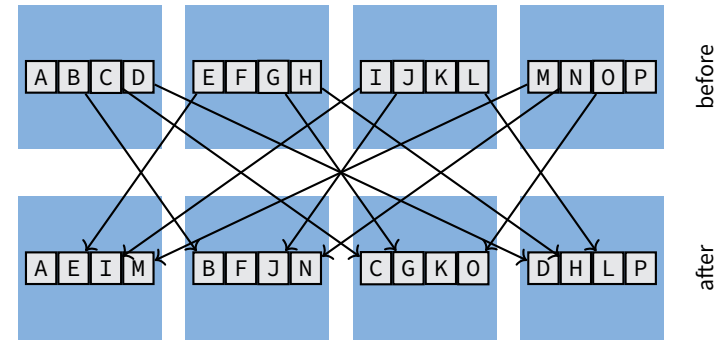
If MPI\_IN\_PLACE is used for sendbuf on the root process, sendcount and sendtype are ignored on the root process and the root process will not send data to itself.

### IN PLACE GATHER-TO-ALL



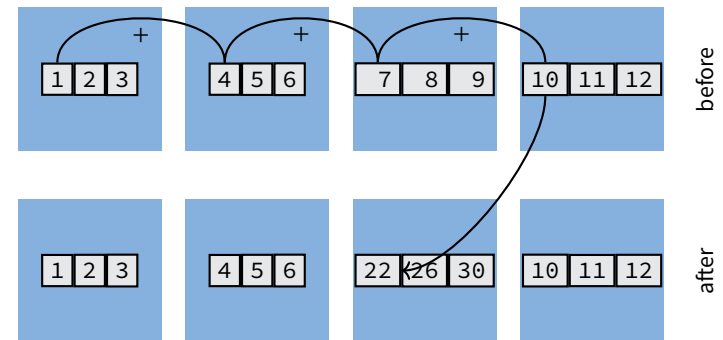
If MPI\_IN\_PLACE is used for sendbuf on all processes, sendcount and sendtype are ignored and the input data is assumed to already be in the correct position in recvbuf.

### IN PLACE ALL-TO-ALL SCATTER/GATHER



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.

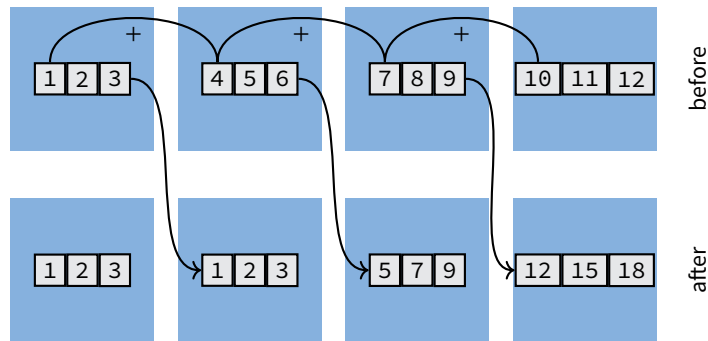
### IN PLACE REDUCE



If MPI\_IN\_PLACE is used for sendbuf on the root process, the input data for the root process is taken from recvbuf.

### IN PLACE EXCLUSIVE SCAN





If `MPI_IN_PLACE` is used for sendbuf on all the processes, the input data is taken from recvbuf and replaced by the results.

## 28 VARIANTS

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

## PART VI

# NONBLOCKING COLLECTIVE COMMUNICATION

## 29 INTRODUCTION

### 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`, ...)

Caution: Nonblocking collective operations cannot be matched with blocking collective operations.

*Nonblocking Barrier*

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

## 30 EXAMPLES

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)
```

```
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
FOR
integer, optional, intent(out) :: ierror
```

```

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

```

## 31 EXERCISES

### EXERCISES

#### Exercise 5 – Collective Communication

##### 5.1 Global Summation – MPI\_Reduce

In the file `global_sum.f90` {c | c++ | f90} implement the function/subroutine `global_sum_reduce(x, y, root, comm)` that performs the same operation as the solution to exercise 3.1.

Since `global_sum...` is a specialization of `MPI_Reduce`, it can be implemented by calling `MPI_Reduce`, passing on the function arguments in the correct way and selecting the correct MPI datatype and reduction operation.

**Use:** `MPI_Reduce`

##### 5.2 Monte Carlo Calculation of $\pi$

Write a program `monte_carlo.f90` {c | c++ | f90} that calculates  $\pi$  using the Monte Carlo method. The program should print the number of processes used, the difference to the exact value of  $\pi$  (can be computed as  $\pi = 2 \arccos(0)$ ) and the time needed for the calculation in seconds (use `MPI_Wtime()` [MPI-3.1, 8.6]).

#### Hints:

- Inscribe a circle in a square.
- Sample a number of points  $n_{\text{total}}$  from a random uniform distribution inside the square.
- Count the number of points that lie inside the circle  $n_{\text{circle}}$ .
- 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>
int rand(void);

```

```

real :: x
call random_number(x)

```

**Use:** `MPI_Wtime`, `MPI_Reduce`

##### 5.3 Redistribution of Points with Collectives

Write a program `redistribute.f90` {c | c++ | f90} 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 `nrank` 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 below)
- 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

**Use:** `MPI_Alltoall`, `MPI_Alltoallv`

## ALL-TO-ALL WITH VARYING COUNTS

```

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

```

```

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

```

## PART VII

# DERIVED DATATYPES

## 32 INTRODUCTION

### 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]

Terminology: Type map

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

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

Terminology: 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];
}

```

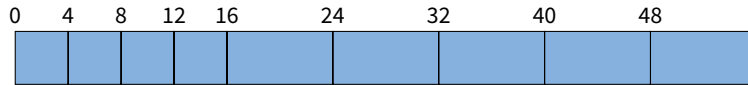
```

type, bind(C) :: heterogeneous
  integer :: i(4)
  real(real64) :: d(5)
end type

```

#### Basic Datatype

0	MPI_INT	MPI_INTEGER
4	MPI_INT	MPI_INTEGER
8	MPI_INT	MPI_INTEGER
12	MPI_INT	MPI_INTEGER
16	MPI_DOUBLE	MPI_REAL8
24	MPI_DOUBLE	MPI_REAL8
32	MPI_DOUBLE	MPI_REAL8
40	MPI_DOUBLE	MPI_REAL8
48	MPI_DOUBLE	MPI_REAL8



## 33 CONSTRUCTORS

### 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:

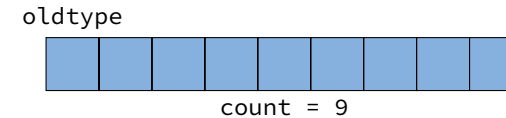
1. `MPI_Type_contiguous`  $n$  consecutive instances of `oldtype`
2. `MPI_Type_vector`  $n$  blocks of  $m$  instances of `oldtype` with stride  $s$
3. `MPI_Type_indexed`  $n$  blocks of  $m_i$  instances of `oldtype` with displacement  $d_i$  for each  $i = 1, \dots, n$
4. `MPI_Type_indexed_block`  $n$  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 `oldtype_i` 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
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.



### 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
```

Caution: Fortran derived data types must be declared **sequence** or **bind**(C), see [MPI-3.1, 17.1.15].

### 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;
```

```

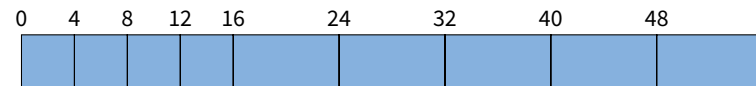
type, bind(C) :: heterogeneous
  integer :: i(4)
  real(real64) :: d(5)
end type

```

```

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
array_of_types(2) = MPI_REAL8

```



## 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

```

## 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;

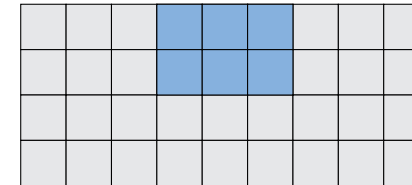
```

```

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  $4 \times 9$  containing a subarray of size  $2 \times 3$  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)

```

```

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)

```

```

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

```

## 34 ADDRESS CALCULATION

### 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;

```

```

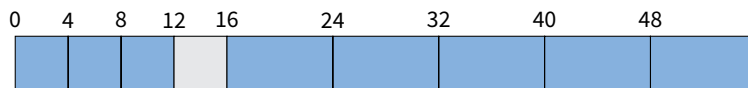
type, bind(C) :: heterogeneous
    integer :: i(3)
    real(real64) :: d(5)
end type

```

```

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
⌋ array_of_types(2) = MPI_REAL8

```



## 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
⌋ 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)

```

```

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

```

Subtraction

```

⌋ MPI_Aint MPI_Aint_diff(MPI_Aint a, MPI_Aint b)

```

```

⌋ integer(kind=MPI_ADDRESS_KIND) MPI_Aint_diff(a, b)
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);

```

```

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

types = (/ MPI_INTEGER, MPI_REAL8 /)
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)

```

## 35 PADDING

### TYPE EXTENT [MPI-3.1, 4.1]

*Terminology: Extent*

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

$$\begin{aligned}
 \text{Typemap} &= \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\} \\
 lb \text{ Typemap} &= \min_j disp_j \\
 ub \text{ Typemap} &= \max_j (disp_j + \text{sizeof } type_j) + \epsilon \\
 \text{extent Typemap} &= ub \text{ Typemap} - lb \text{ Typemap}
 \end{aligned}$$

*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, ...)` 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
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`.

## 36 EXERCISES

### Exercise 6 – Derived Datatypes

#### 6.1 Structs

Given a definition of a datatype that represents a point in three-dimensional space with additional properties:

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

write a program `struct.{c|c++|f90}` in which 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?

**Use:** `MPI_Get_address`, `MPI_Aint_diff`, `MPI_Type_create_struct`, `MPI_Type_commit`, `MPI_Type_free`, `MPI_Bcast`

#### 6.2 Matrix Access – Diagonal

In the file `matrix_access.{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:

$$vector_i = matrix_{i,i}, \quad i = 1 \dots N.$$

Do not access the elements of either the matrix or the vector directly. Rather, use MPI datatypes for accessing your data. Assume that the matrix elements are stored in row-major order in C (all elements of the first row, followed by all elements of the second row, etc.), column-major order in Fortran.

**Hint:** `MPI_Sendrecv` on the `MPI_COMM_SELF` communicator can be used for copying the data.

**Use:** MPI\_Type\_vector

### 6.3 Matrix Access – Upper Triangle

In the file `matrix_access.{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 and assume row-major layout of the matrices in C, column-major order in Fortran. 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

## PART VIII

# INPUT/OUTPUT

## 37 INTRODUCTION

### 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

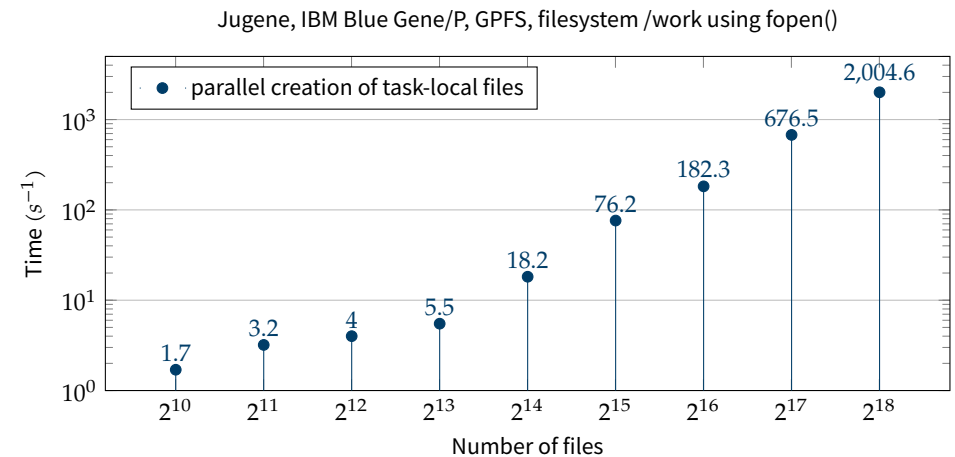
*All or several processes use one file*

- + 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

*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





## 38 FILE MANIPULATION

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

*Terminology: File*

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

*Terminology: File Pointer*

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

*Terminology: 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
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, ior intrinsic in Fortran).

### CLOSING A FILE [MPI-3.1, 13.2.2]

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

```
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)
```

```
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]

Caution: 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)
```

```
EOS MPI_File_set_errhandler(file, errhandler, ierror)
type(MPI_File), intent(in) :: file
type(MPI_Errhandler), intent(in) :: errhandler
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 using MPI\_File\_set\_errhandler on a special file handle, MPI\_FILE\_NULL

## 39 FILE VIEWS

### FILE VIEW [MPI-3.1, 13.3]

#### Terminology: 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]

#### Terminology: 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]

Terminology: 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.  $Filetype_0 = \{(int, 0), (hole, 4), (hole, 8)\}$ ,  $Filetype_1 = \{(hole, 0), (int, 4), (hole, 8)\}, \dots$ :

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
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

"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

## 40 DATA ACCESS

Three orthogonal aspects

1. Synchronism
  - (a) Blocking
  - (b) Nonblocking
  - (c) Split collective

## 2. Coordination

- (a) Noncollective
- (b) Collective

## 3. Positioning

- (a) Explicit offsets
- (b) Individual file pointers
- (c) Shared file pointers

*POSIX read() and write()*

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

Combine the prefix `MPI_File_` with any of the following suffixes:

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	read_at, write_at	read_at_all, write_at_all
	nonblocking	iread_at, iwrite_at	iread_at_all, iwrite_at_all
	split collective	N/A	read_at_all_begin, read_at_all_end, write_at_all_begin, write_at_all_end
individual file pointers	blocking	read, write	read_all, write_all
	nonblocking	iread, iwrite	iread_all, iwrite_all
	split collective	N/A	read_all_begin, read_all_end, write_all_begin, write_all_end
shared file pointers	blocking	read_shared, write_shared	read_ordered, write_ordered
	nonblocking	iread_shared, iwrite_shared	N/A
	split collective	N/A	read_ordered_begin, read_ordered_end, write_ordered_begin, 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
integer, optional, intent(out) :: ierror
```

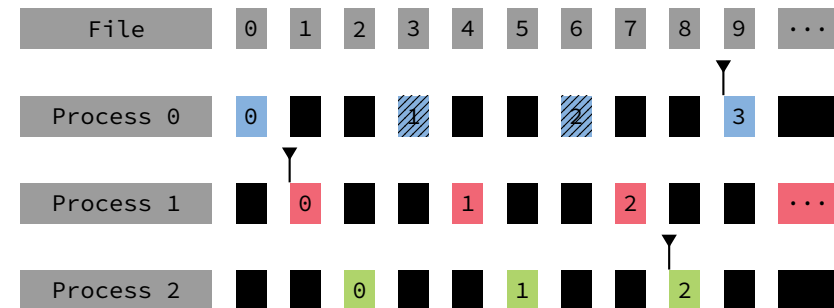
- Starting offset for access is explicitly given

- No file pointer is updated
- Writes count elements of datatype from memory starting at buf
- Typesig *datatype* = Typesig *etype* ... 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)
```

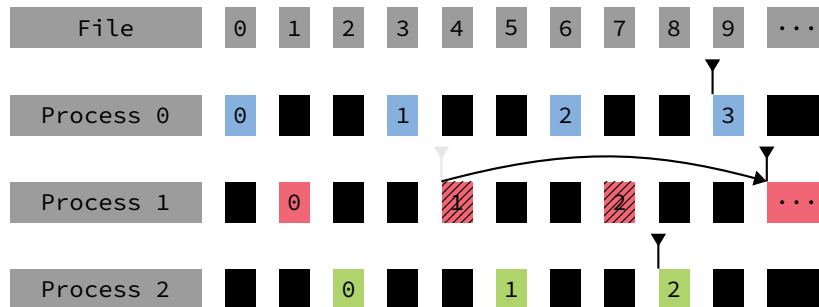
```
MPI_File_write(fh, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- 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 calls `MPI_File_write(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
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)
```

F08

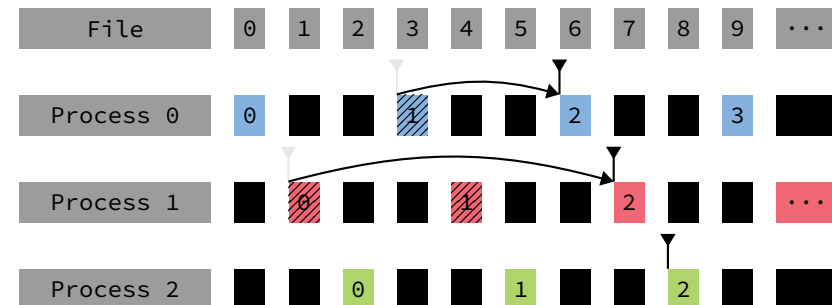
```
MPI_File_write_all(fh, buf, count, datatype, status, ierror)
type(MPI_File), intent(in) :: fh
type(*), dimension(..), intent(in) :: buf
integer, intent(in) :: count
type(MPI_Datatype), intent(in) :: datatype
type(MPI_Status) :: status
integer, optional, intent(out) :: ierror
```

- 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 calls `MPI_File_write_all(count = 1)`,
- With its file pointer at element 0, process 1 calls `MPI_File_write_all(count = 2)`,
- With its file pointer at element 2, process 2 calls `MPI_File_write_all(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)
```

F08

```

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
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
integer, optional, intent(out) :: ierror

```

F08

- buf argument must match corresponding begin routine

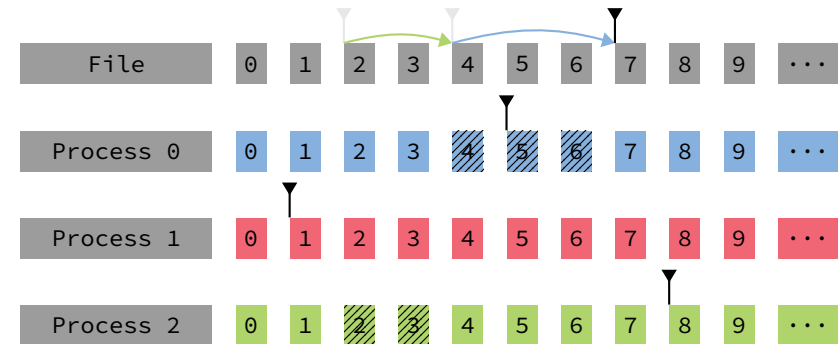
## EXAMPLE

blocking, noncollective, shared [\[MPI-3.1, 13.4.4\]](#)

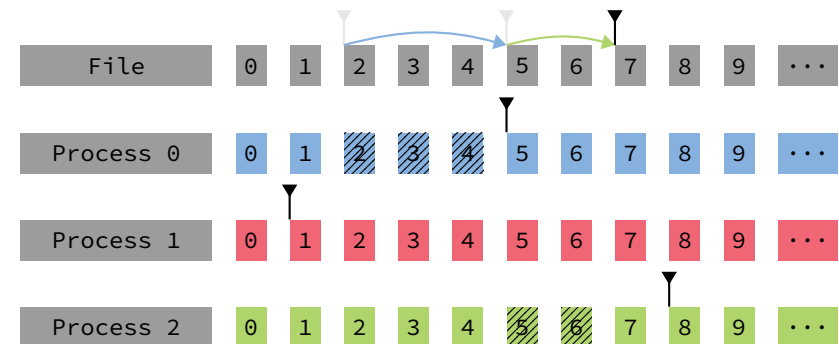
With the shared pointer at element 2,

- process 0 calls MPI\_File\_write\_shared(count = 3),
- process 2 calls MPI\_File\_write\_shared(count = 2):

### Scenario 1:



### Scenario 2:

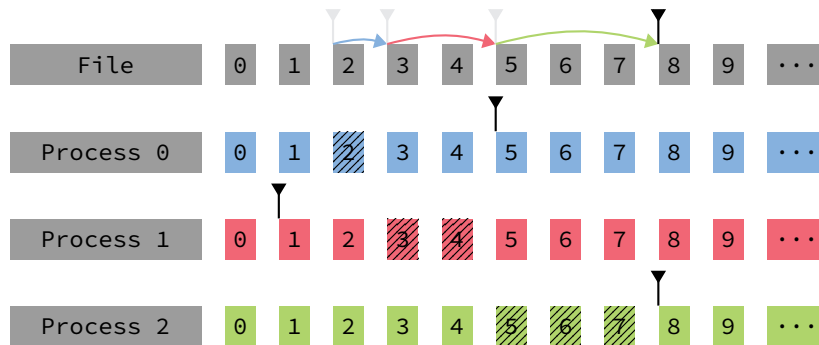


## EXAMPLE

blocking, collective, shared [\[MPI-3.1, 13.4.4\]](#)

With the shared pointer at element 2,

- process 0 calls MPI\_File\_write\_ordered(count = 1),
- process 1 calls MPI\_File\_write\_ordered(count = 2),
- process 2 calls MPI\_File\_write\_ordered(count = 3):



## READING

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

```

C  int MPI_File_read(MPI_File fh, void* buf, int count,
  ~ MPI_Datatype datatype, MPI_Status* status)

```

```

F08 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
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]

```

C  int MPI_File_get_position(MPI_File fh, MPI_Offset* offset)

```

```

F08 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]

```

C  int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)

```

```

F08 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
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]

```

C  int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
  ~ MPI_Offset* disp)

```

```

F08 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
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



## 41 CONSISTENCY

### CONSISTENCY [MPI-3.1, 13.6.1]

*Terminology: 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

Caution: 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)
```

```
⌋ 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

*Terminology: 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)
```

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

### 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

## 42 EXERCISES

*Exercise 7 – Data Access*

7.1 Writing and Reading Data

Write a program `write_rank.{c|c++|f90}`:

- Each process writes its own rank to the common file `rank.dat`
- 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

**Use:** MPI\_File\_open, MPI\_File\_set\_errhandler, MPI\_File\_set\_view, MPI\_File\_write\_ordered, MPI\_File\_sync, MPI\_File\_read, MPI\_File\_close

7.2 Accessing Parts of Files

Take the file `rank.dat` from the previous exercise and write a program `read_rank.{c|c++|f90}` where:

- 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

**Use:** `MPI_File_seek`, `MPI_File_get_position`

### 7.3 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

```

Write a program `phonebook.{c|c++|f90}` and use MPI I/O to 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.

**Use:** `MPI_File_read`

## PART IX

# TOOLS

## 43 MUST

### 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.

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

## 44 EXERCISES

### Exercise 8 – MPI Tools

#### 8.1 Must

Have a look at the file `must.c`. 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:** `must.f90` uses the MPI Fortran 90 interface.

## PART X

# COMMUNICATORS

## 45 INTRODUCTION

### 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.

## 46 CONSTRUCTORS

### DUPLICATE [MPI-3.1, 6.4.2]

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

F08

```
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]

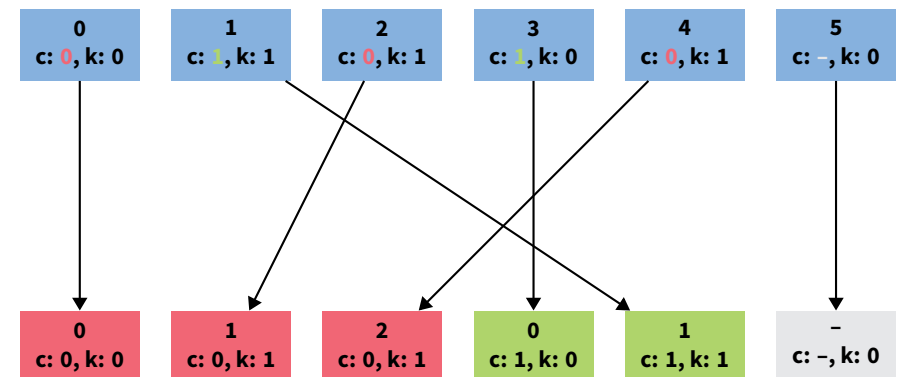
C

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

F08

```
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
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

#### Input:

`comm_old` contains 12 processes (or more)

`ndims = 2, dims = [ 4, 3 ], periods = [ .false., .false. ] reorder = .false.`

#### Output:

process 0–11: new communicator with topology as shown

process 12–: MPI\_COMM\_NULL

2	<b>2</b> (0, 2)	<b>5</b> (1, 2)	<b>8</b> (2, 2)	<b>11</b> (3, 2)
1	<b>1</b> (0, 1)	<b>4</b> (1, 1)	<b>7</b> (2, 1)	<b>10</b> (3, 1)
0	<b>0</b> (0, 0)	<b>3</b> (1, 0)	<b>6</b> (2, 0)	<b>9</b> (3, 0)
	0	1	2	3

## 47 ACCESSORS

### 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)
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
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
integer, optional, intent(out) :: ierror

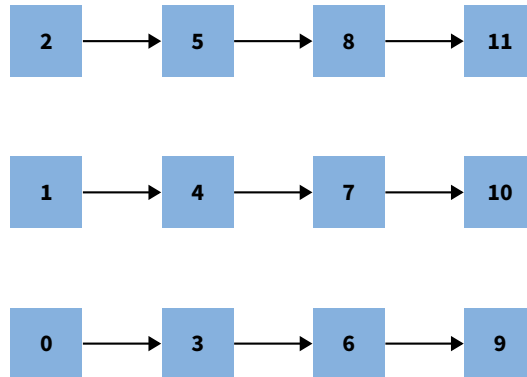
```

- 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

**Input:** direction = 0, disp = 1, not periodic

**Output:**

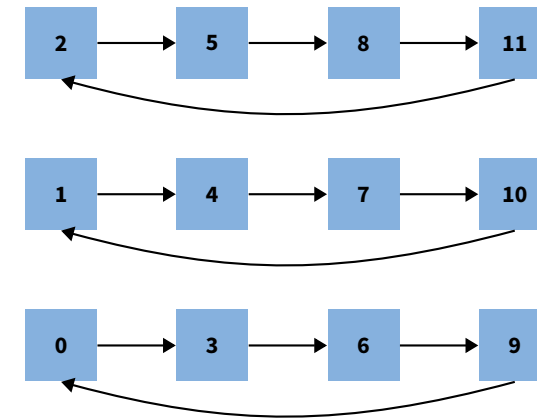
process 0: rank\_source = MPI\_PROC\_NULL, rank\_dest = 3  
...  
process 3: rank\_source = 0, rank\_dest = 6  
...  
process 9: rank\_source = 6, rank\_dest = MPI\_PROC\_NULL  
...



**Input:** direction = 0, disp = 1, periodic

**Output:**

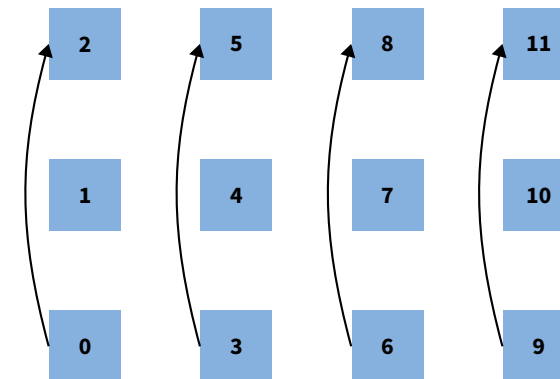
process 0: rank\_source = 9, rank\_dest = 3  
...  
process 3: rank\_source = 0, rank\_dest = 6  
...  
process 9: rank\_source = 6, rank\_dest = 0  
...



**Input:** direction = 1, disp = 2, not periodic

**Output:**

process 0: rank\_source = MPI\_PROC\_NULL, rank\_dest = 2  
process 1: rank\_source = MPI\_PROC\_NULL, rank\_dest = MPI\_PROC\_NULL  
process 2: rank\_source = 0, rank\_dest = MPI\_PROC\_NULL  
...



**NULL PROCESSES [MPI-3.1, 3.11]**

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

```
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

#### COMPARISON [MPI-3.1, 6.4.1]

```

C  int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int
   ~ *result)

```

```

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

## 48 DESTRUCTORS

#### FREE [MPI-3.1, 6.4.3]

```

C  int MPI_Comm_free(MPI_Comm *comm)

```

```

FO8 MPI_Comm_free(comm, ierror)
     type(MPI_Comm), intent(inout) :: comm
     integer, optional, intent(out) :: ierror

```

Marks a communicator for deallocation.

## 49 EXERCISES

### Exercise 9 – Communicators

#### 9.1 Cartesian Topology

In `global_sum_with_communicators.{c|c++|f90}`, redo exercise 3.1 using a Cartesian communicator.

**Use:** `MPI_Cart_create`, `MPI_Cart_shift`, `MPI_Comm_free`

#### 9.2 Split

In `global_sum_with_communicators.{c|c++|f90}`, redo exercise 4.1 using a new split communicator per communication round.

**Use:** `MPI_Comm_split`

## PART XI

# THREAD COMPLIANCE

## 50 INTRODUCTION

#### 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`

## 51 ENABLING THREAD SUPPORT

#### 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

MPI\_THREAD\_SINGLE < MPI\_THREAD\_FUNNELED < MPI\_THREAD\_SERIALIZED < MPI\_THREAD\_MULTIPLE

## INITIALIZATION [MPI-3.1, 12.4.3]

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

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

```
F08 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)
```

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

## 52 MATCHING PROBE AND RECEIVE

### MATCHING PROBE [MPI-3.1, 3.8.2]

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

```
F08 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  
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)
```

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

- Receives the previously probed message message
- Sets the message handle to MPI\_MESSAGE\_NULL
- Nonblocking variant MPI\_Imrecv exists

## 53 REMARKS

### 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

## 54 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 FAQ*<sup>3</sup>)

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

Current version of the specification: 5.0 (November 2018)

### BRIEF HISTORY

**1997** FORTRAN version 1.0

**1998** C/C++ version 1.0

**1999** FORTRAN version 1.1

**2000** FORTRAN version 2.0

**2002** C/C++ version 2.0

**2005** First combined version 2.5, memory model, internal control variables, clarifications

**2008** Version 3.0, tasks

**2011** Version 3.1, extended task facilities

**2013** Version 4.0, thread affinity, SIMD, devices, tasks (dependencies, groups, and cancellation), improved Fortran 2003 compatibility

**2015** Version 4.5, extended SIMD and devices facilities, task priorities

<sup>3</sup>Matthijs van Waveren et al. *OpenMP FAQ*. version 3.0. June 6, 2018. URL: <http://www.openmp.org/about/openmp-faq/> (visited on 01/30/2019).

**2018** Version 5.0, memory model, base language compatibility, allocators, extended task and devices facilities

### COVERAGE

#### • Directives

- Directive Format ✓
- Conditional Compilation ✓
- Variant Directives
- `requires` Directive
- Internal Control Variables ✓
- `parallel` Construct ✓
- `teams` Construct
- Worksharing Constructs ✓
- Loop-Related Directives (✓)
- Tasking Constructs ✓
- Memory Management Directives
- Device Directives
- Combined Constructs & Clauses on Combined and Composite Constructs (✓)
- `if` Clause ✓
- `master` Construct
- Synchronization Constructs and Clauses ✓
- Cancellation Constructs
- Data Environment ✓
- Nesting of Regions

#### • Runtime Library Routines

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



- Memory Management Routines
- Tool Control Routine
- ...
- Environment Variables (✓)
- ...

## LITERATURE

### Official Resources

- OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Version 5.0. Nov. 2018. URL: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.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

### 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

## 55 TERMINOLOGY

### THREADS & TASKS

*Terminology: Thread*

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

*Terminology: OpenMP Thread*

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

*Terminology: Team*

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

*Terminology: Task*

A specific instance of executable code and its data environment that the OpenMP implementation can schedule for execution by threads.

### LANGUAGE

*Terminology: Base Language*

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

The following base languages are given in [OpenMP-5.0, 1.7]: C90, C99, C++98, Fortran 77, Fortran 90, Fortran 95, Fortran 2003, and subsets of C11, C++11, C++14, C++17, and Fortran 2008

*Terminology: Base Program*

A program written in the *base language*.

*Terminology: OpenMP Program*

A program that consists of a *base program* that is annotated with OpenMP *directives* or that calls OpenMP API runtime library routines.

*Terminology: Directive*

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

## 56 INFRASTRUCTURE

### 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
---

\$ <code>icc -qopenmp ...</code>
----------------------------------

GNU Fortran Compiler OpenMP Command Line Switch
---

\$ <code>gfortran -fopenmp ...</code>
---------------------------------------

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
--

\$ <code>mpicc -qopenmp ...</code>
------------------------------------

## RUNTIME LIBRARY DEFINITIONS [OpenMP-5.0, 3.1]

### C/C++ Runtime Library Definitions

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

<code>#include &lt;omp.h&gt;</code>
-------------------------------------

### Fortran Runtime Library Definitions

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

<code>include "omp_lib.h"</code>
----------------------------------

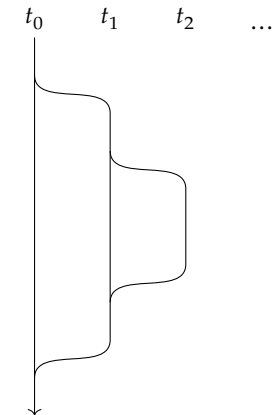
or a Fortran 90 module

<code>use omp_lib</code>
--------------------------

## 57 BASIC PROGRAM STRUCTURE

### 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.



## C AND C++ DIRECTIVE FORMAT [OpenMP-5.0, 2.1]

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

<code>#pragma omp directive-name [clause[[,] clause]...]</code>
---

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

### Terminology: 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*.

## FORTRAN DIRECTIVE FORMAT [OpenMP-5.0, 2.1.1, 2.1.2]

<code>sentinel directive-name [clause[[,] clause]...]</code>
--

- Directives are case-insensitive

### Fixed Form Sentinels

<code>sentinel = !\$omp   c\$omp   *\$omp</code>
--

- 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

```
FO8 sentinel = !$omp
```

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

### CONDITIONAL COMPILATION [OpenMP-5.0, 2.2]

#### C Preprocessor Macro

```
⌋ #define _OPENMP yyyyymm
```

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

#### Terminology: 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*.

#### Terminology: Region

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

#### Terminology: Executable Directive

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

### THE PARALLEL CONSTRUCT [OpenMP-5.0, 2.6]

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

```
FO8 !$omp parallel [clause[[],] clause]...  
    structured-block  
FO8 !$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-5.0, 3.2.2, 3.2.4]

#### Team size

```
⌋ int omp_get_num_threads(void);
```

```
FO8 integer function omp_get_num_threads()
```

Returns the number of threads in the current team

#### Thread number

```
⌋ int omp_get_thread_num(void);
```

```
FO8 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 <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");
}

```

#### 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.

```

```

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

```

E08

## 58 EXERCISES

### Exercise 10 – Warm Up

#### 10.1 Generalized Vector Addition (axpy)

In the file `axpy.{c|c++|f90}`, fill in the missing body of the function/subroutine `axpy_serial(a, x, y, z[, n])` 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.

#### 10.2 Dot Product

In the file `dot.{c|c++|f90}`, fill in the missing body of the function/subroutine `dot_serial(x, y[, n])` 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

## 59 INTRODUCTION

### MAGIC

Any sufficiently advanced technology is indistinguishable from magic.  
(Arthur C. Clarke<sup>4</sup>)

### INTERNAL CONTROL VARIABLES [OpenMP-5.0, 2.5]

*Terminology: 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()`)

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

- 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)
- Use `OMP_DISPLAY_ENV=TRUE` to inspect the value of ICVs that correspond to environment variables [\[OpenMP-5.0, 6.12\]](#)

## PARALLELISM CLAUSES [\[OpenMP-5.0, 2.6, 2.15\]](#)

### if Clause

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

```
FO8 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

```
C num_threads(integer-expression)
```

```
FO8 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;
C }
```

A parallel directive with a num\_threads clause and associated structured block in Fortran:

```
!$omp parallel num_threads( 64 )
statement1
statement2
statement3
FO8 !$omp end parallel
```

## CONTROLLING THE *nthreads-var* ICV

### omp\_set\_num\_threads API Routine [\[OpenMP-5.0, 3.2.1\]](#)

```
C 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 [\[OpenMP-5.0, 3.2.3\]](#)

```
C 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 [\[OpenMP-5.0, 3.2.14\]](#)

```
C 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 [\[OpenMP-5.0, 3.2.7, 3.2.8\]](#)

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

FOR

```

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

```

Enable or disable dynamic adjustment of the number of threads.

## INSIDE OF A PARALLEL REGION?

**omp\_in\_parallel** API Routine [OpenMP-5.0, 3.2.6]

```

C int omp_in_parallel(void);

```

```

F08 logical function omp_in_parallel()

```

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

## 60 EXERCISES

### Exercise 11 – Controlling parallel

#### 11.1 Controlling the Number of Threads

Use `hello_omp.c` to play around with the various ways to set the number of threads forked for a parallel region:

- The `OMP_NUM_THREADS` environment variable
- The `omp_set_num_threads` API routine
- The `num_threads` clause
- The `if` clause

Inspect the number of threads that are actually forked using `omp_get_num_threads`.

#### 11.2 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.

## 61 DATA ENVIRONMENT

### DATA-SHARING ATTRIBUTES [OpenMP-5.0, 2.19.1]

#### Terminology: Variable

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

#### Terminology: 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.

#### Terminology: 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 I [OpenMP-5.0, 2.19.1.1]

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++)
- Objects with dynamic storage duration are shared (C and C++)
- Variables with static storage duration declared in the construct are shared (C and C++)
- Static data members are shared (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 II [OpenMP-5.0, 2.19.1.2]

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-5.0, 2.19.4.2]

```
* 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-5.0, 2.19.4.3]

```
* 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-5.0, 2.19.4.4]

```
* 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-5.0, 2.19.4.1]

##### 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.

Caution: `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-5.0, 2.19.5.4]

```
* 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`, `iand`, `ior` or `ieor` (Fortran)
- Private versions of the variable are initialized with appropriate values

## 62 EXERCISES

### Exercise 12 – Data-sharing Attributes

#### 12.1 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.

## 63 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.

#### Terminology: 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-5.0, 2.17.2]

```
C #pragma omp barrier
```

```
FOR !$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-5.0, 2.17.1]

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

```
FOR !$omp critical [(name)]
   structured-block
!$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-5.0, 3.3]

```
C 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);
```

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

- Like `critical` sections, but identified by runtime value rather than global name
- Locks must be shared between threads
- 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-5.0, 2.17.7, 2.17.8]

- `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 [\[OpenMP-5.0, 1.4\]](#)
- See also: C11 and C++11 Memory Models

## 64 EXERCISES

### Exercise 13 – Thread Synchronization

#### 13.1 Dot Product

In the file `dot.{c|c++|f90}` add a new function/subroutine `dot_parallel(x, y[, n])` that uses multiple threads to perform the dot product. Do not use the `reduction` 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 `reduction` clause to simplify your program.

## PART XIV

# WORKSHARING

## 65 INTRODUCTION

### 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`

## 66 THE SINGLE CONSTRUCT

### THE SINGLE CONSTRUCT [\[OpenMP-5.0, 2.8.2\]](#)

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

```
!$omp single [clause[[,] clause]...]
└─ structured-block
!$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.

## 67 SINGLE CLAUSES

### IMPLICIT BARRIERS & THE NOWAIT CLAUSE [\[OpenMP-5.0, 2.8\]](#)

- 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.

### THE COPYPRIVATE CLAUSE [\[OpenMP-5.0, 2.19.6.2\]](#)

```
* 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 `nowait`.

## 68 THE LOOP CONSTRUCT

### WORKSHARING-LOOP CONSTRUCT [OpenMP-5.0, 2.9.2]

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

```
!$omp do [clause[,] clause]...  
do-loops  
$OMP [!$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

Worksharing-Loop-specific clauses

- schedule
- collapse

### CANONICAL LOOP FORM [OpenMP-5.0, 2.9.1]

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

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

```
⧈ for (range-decl: range-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
- For nested loops, the bounds of an inner loop (b and lb) may depend at most linearly on the iteration variable of an outer loop, i.e.  $a_0 + a_1 * \text{var-outer}$
- var must not be modified by the loop body
- The beginning of the range has to be a random access iterator

- The number of iterations of the loop must be known beforehand

In Fortran the do-loops must have the following form:

```
DO  
do [label] var = lb, b[, incr]
```

- var must be of integer type
- incr must be invariant with respect to the outermost loop
- The loop bounds b and lb of an inner loop may depend at most linearly on the iteration variable of an outer loop, i.e.  $a_0 + a_1 * \text{var-outer}$
- The number of iterations of the loop must be known beforehand

## 69 LOOP CLAUSES

### THE COLLAPSE CLAUSE [OpenMP-5.0, 2.9.2]

```
* 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-5.0, 2.9.2]

```
* 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`.

If no `schedule` clause is present, the default schedule is implementation defined.

## 70 EXERCISES

*Exercise 14 – Loop Worksharing*

14.1 Generalized Vector Addition (axpy)

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

14.2 Dot Product

In the file `dot.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.

## 71 WORKSHARE CONSTRUCT

**WORKSHARE (FORTRAN ONLY) [OpenMP-5.0, 2.8.3]**

```
!$omp workshare
  structured-block
 $\infty$  !$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.

## 72 EXERCISES

*Exercise 15 – workshare Construct*

15.1 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.

15.2 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.

## 73 COMBINED CONSTRUCTS

**COMBINED CONSTRUCTS [OpenMP-5.0, 2.13]**

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

## 74 INTRODUCTION

### TASK TERMINOLOGY

*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.

*Terminology: 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.

*Terminology: Descendent Task*

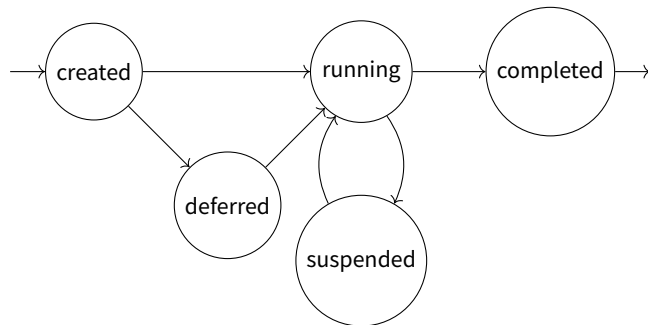
A task that is the *child task* of a task region or of one of its *descendent task regions*.

*Terminology: 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*



## 75 THE TASK CONSTRUCT

### THE TASK CONSTRUCT [OpenMP-5.0, 2.10.1]

```
#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
- untied
- mergeable
- depend
- priority

### TASK DATA-ENVIRONMENT [OpenMP-5.0, 2.19.1.1]

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`.

## 76 TASK CLAUSES

### THE IF CLAUSE [OpenMP-5.0, 2.10.1]

```
* 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

*Terminology: 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-5.0, 2.10.1]

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

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

- *undeferred* and
- executed immediately.

*Terminology: Final Task*

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

*Terminology: Included Task*

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

## THE UNTIED CLAUSE [OpenMP-5.0, 2.10.1]

```
* 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.

*Terminology: 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*.

*Terminology: 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 PRIORITY CLAUSE [OpenMP-5.0, 2.10.1]

```
* 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-5.0, 2.17.11]

```
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.

*Terminology: Dependent Task*

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

# 77 TASK SCHEDULING

## TASK SCHEDULING POLICY [OpenMP-5.0, 2.10.6]

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-5.0, 2.10.4]

```
┌ #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

## 78 TASK SYNCHRONIZATION

### THE TASKWAIT & TASKGROUP CONSTRUCTS [OpenMP-5.0, 2.17.5, 2.17.6]

```
┌ #pragma omp taskwait
```

```
└ !$omp taskwait
```

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

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

```
└ !$omp taskgroup  
  structured-block  
└ !$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.

## 79 EXERCISES

### Exercise 16 – Task worksharing

#### 16.1 Generalized Vector Addition (axpy)

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

#### 16.2 Dot Product

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

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

#### 16.3 Bitonic Sort

The file `bsort.{c|c++|f90}` contains a serial implementation of the bitonic sort algorithm. Use OpenMP task worksharing to parallelize it.

## 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 GPU programming using OpenACC, **28 – 29 October**
- Introduction to the usage and programming of supercomputer resources in Jülich, **28 – 29 November**
- Advanced Parallel Programming with MPI and OpenMP, **2 – 4 December**
- And more, see [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Workshops/Courses/courses\\_node.html](http://www.fz-juelich.de/ias/jsc/EN/Expertise/Workshops/Courses/courses_node.html)

## PART XVII

# TUTORIAL

### N-BODY SIMULATIONS

Dynamics of the N-body problem:

$$\mathbf{a}_{i,j} = \frac{q_i q_j}{\sqrt{(\mathbf{x}_i - \mathbf{x}_j) \cdot (\mathbf{x}_i - \mathbf{x}_j)}}^3 (\mathbf{x}_i - \mathbf{x}_j)$$
$$\ddot{\mathbf{x}}_i = \mathbf{a}_i = \sum_{j \neq i} \mathbf{a}_{i,j}$$

Velocity Verlet integration:

$$\mathbf{v}^* \left( t + \frac{\Delta t}{2} \right) = \mathbf{v}(t) + \frac{\Delta t}{2} \mathbf{a}(t)$$
$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}^* \left( t + \frac{\Delta t}{2} \right) \Delta t$$
$$\mathbf{v}(t + \Delta t) = \mathbf{v}^* \left( t + \frac{\Delta t}{2} \right) + \frac{\Delta t}{2} \mathbf{a}(t + \Delta t)$$

Program structure:

```
read initial state from file
calculate accelerations
for number of time steps:
    write state to file
    calculate helper velocities v*
    calculate new positions
    calculate new accelerations
    calculate new velocities
* write final state to file
```

### A SERIAL N-BODY SIMULATION PROGRAM

Compiling nbody.x

```
$ make nbody.x
...
```

Invoking nbody.x

```
$ ./nbody.x
Usage: nbody <input file>
$ ./nbody.x ../input/kaplan_10000.bin
Working on step 1...
```

Visualizing the results

```
$ paraview --state=kaplan.pvsm
```

Initial conditions based on: A. E. Kaplan, B. Y. Dubetsky, and P. L. Shkolnikov. “Shock Shells in Coulomb Explosions of Nanoclusters”. In: *Physical Review Letters* 91 (14 Oct. 3, 2003), p. 143401. doi: 10.1103/PhysRevLett.91.143401

## SOME SUGGESTIONS

### *Distribution of work*

Look for loops with a number of iterations that scales with the problem size  $N$ . If the individual loop iterations are independent, they can be run in parallel. Try to distribute iterations evenly among the threads / processes.

### *Distribution of data*

What data needs to be available to which process at what time? Having the entire problem in the memory of every process will not scale. Think of particles as having two roles: targets ( $i$  index) that experience acceleration due to sources ( $j$  index). Make every process responsible for a group of either target particles or source particles and communicate the same particles in the other role to other processes. What particle properties are important for targets and sources?

### *Input / Output*

You have heard about different I/O strategies during the MPI I/O part of the course. Possible solutions include:

- Funneled I/O: one process reads then scatters or gathers then writes
- MPI I/O: every process reads or writes the particles it is responsible for

### *Scalability*

Keep an eye on resource consumption. Ideally, the time it takes for your program to finish should be inversely proportional to the number of threads or processes running it  $O(N^2/p)$ . Similarly, the amount of memory consumed by your program should be independent of the number of processes  $O(N)$ .

## EXERCISES

### *Exercise 17 – N-body simulation program*

#### 17.1 OpenMP parallel version

Write a version of `nbody.x` that is parallelized using OpenMP. Look for suitable parts of the program to annotate with OpenMP directives.

#### 17.2 MPI parallel version

Write a version of `nbody.x` that is parallelized using MPI. The distribution of work might be similar to the previous exercise. Ideally, the entire system state is not stored on every process, thus particle data has to be communicated. Communication could be point-to-point or collective. Input and output functions might have to be adapted as well.

#### 17.3 Hybrid parallel version


Write a version of `nbody.x` that is parallelized using both MPI and OpenMP. This might just be a combination of the previous two versions.

## Bonus

A clever solution is described in: M. Driscoll et al. “A Communication-Optimal N-Body Algorithm for Direct Interactions”. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013, pp. 1075–1084. doi: 10.1109/IPDPS.2013.108. Implement Algorithm 1 from the paper.

## COLOPHON

This document was typeset using

- Lua<sup>®</sup>TeX and a host of macro packages,
- Adobe Source Sans Pro for body text and headings,
- Adobe Source Code Pro for listings,
- TeX Gyre Pagella Math for mathematical formulae,
- icons from Font Awesome .