A SHORT INTRODUCTION TO PARALLEL PROGRAMMING WITH MPI AND OPENMP

20–21 November 2018 | Benedikt Steinbusch | Jülich Supercomputing Centre

Member of the Helmholtz Association

JÜLICH Forschungszentrum

A SHORT INTRODUCTION TO PARALLEL PROGRAMMING WITH MPI AND OPENMP

Benedikt Steinbusch | Jülich Supercomputing Centre | 20–21 November 2018

CONTENTS

TIMETABLE

	Day 1	Day 2
09:00 10:30	Fundamentals of Parallel Computing	First Steps with OpenMP
	☕	
11:00 12:30	First Steps with MPI	Low-Level Constructs
	🍴	
13:30 15:00	Point-to-Point Communication	Loop Worksharing
	☕	
15:30 17:00	Collective Communication & Thread Compliance	Task Worksharing

PART I

FUNDAMENTALS OF PARALLEL COMPUTING

1 MOTIVATION

PARALLEL COMPUTING

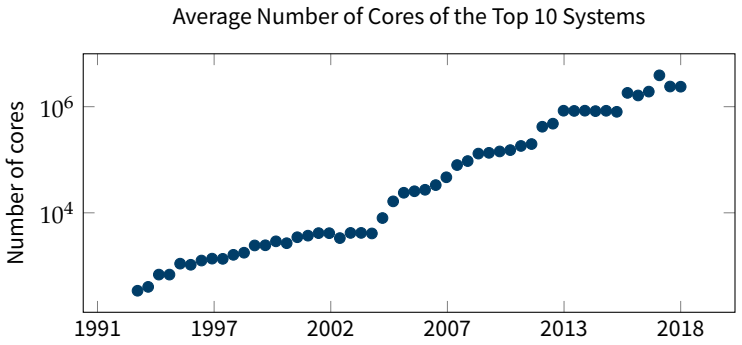
Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. (Wikipedia[1])

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

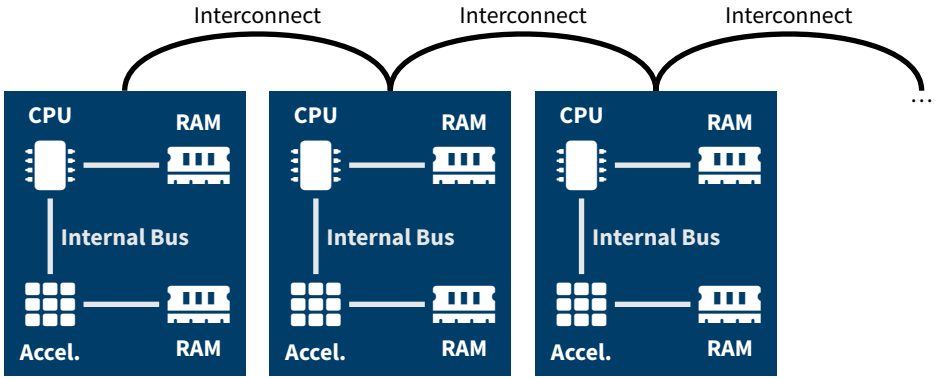
PARALLELISM IN THE TOP 500 LIST



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

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)

PART II

FIRST STEPS WITH MPI

4 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[2])

- 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
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

5 TERMINOLOGY

PROCESS ORGANIZATION [MPI-3.1, 6.2]

[2] 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`.

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.

6 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
$ bgxlc foo.c -o foo
```

Fortran Compiler Wrappers

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

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

PROCESS STARTUP [MPI-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

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

Fortran Language Bindings

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

F08
```
use mpi_f08
```

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

F90
```
use mpi
```

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

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

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

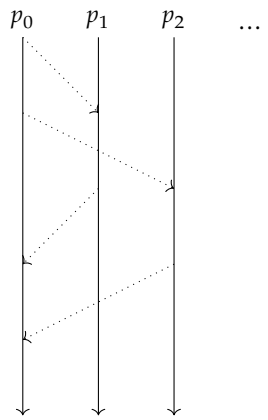
Java mpiJava, MPJ, MPJ Express

And many others, ask your favorite search engine.

7 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).



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

Exception (can be used before initialization)

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

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

FINALIZATION [MPI-3.1, 8.7]

Finalize MPI library when you are done using its functions

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

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

Exception (can be used after finalization)

```c
int MPI_Finalized(int *flag)
```

```f08
MPI_Finalized(flag, ierror)
logical, intent(out) :: flag
integer, optional, intent(out) :: ierror
```

PREDEFINED COMMUNICATORS

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

```c
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 (not part of this course)
  - 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`

8 FIRST STEPS ON A SUPERCOMPUTER

LOGIN & PROGRAMMING ENVIRONMENT

Compilers

C

```
$ mpicc
```

Fortran

```
$ mpif90
```

C++

```
$ mpicxx
```

JURECA Login

1. Open a terminal on your PC

   ```
   $ ssh jureca
   ```

   (YMMV)

2. Load modules

   ```
   $ module load intel-para
   ```

Course Material

```
/homea/hpclab/trainXXX/mpi-omp/
|--exercises/{C|C++|Fortran}
|--tutorial/{C|C++|Fortran}
\--mpi-omp-article.pdf
```

RUNNING PARALLEL PROGRAMS ON JURECA

Interactive Mode

1. Start an interactive session

   ```
   $ salloc --reservation=ompi --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=ompi <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

9 EXERCISES

Exercise 1 – First Steps

1.1 Output of Ranks

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

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

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

1.2 Output of ranks and total number of processes

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

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

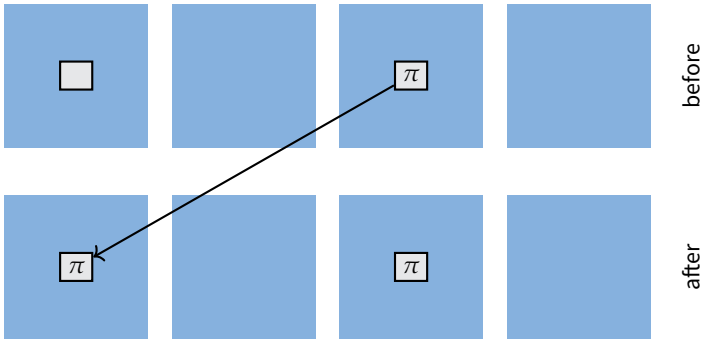
Use: `MPI_Comm_size`

PART III

BLOCKING POINT-TO-POINT COMMUNICATION

10 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

11 SENDING

SENDING MESSAGES [MPI-3.1, 3.2.1]

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

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

```
F08 MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    type(*), dimension(..), intent(in) :: buf
    integer, intent(in) :: count, dest, tag
    type(MPI_Datatype), intent(in) :: datatype
    type(MPI_Comm), intent(in) :: comm
    integer, optional, intent(out) :: ierror
```

MESSAGES [MPI-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
`signed int`	`MPI_INT`
`float`	`MPI_FLOAT`
`char`	`MPI_CHAR`
…	

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

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]

12 RECEIVING

RECEIVING MESSAGES [MPI-3.1, 3.2.4]

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

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

```
F08  MPI_Recv(buf, count, datatype, source, tag, comm, status,
     ↪  ierror)
     type(*), dimension(..) :: buf
     integer, intent(in) :: count, source, tag
     type(MPI_Datatype), intent(in) :: datatype
     type(MPI_Comm), intent(in) :: comm
     type(MPI_Status) :: status
     integer, optional, intent(out) :: ierror
```

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

THE MPI_STATUS TYPE [MPI-3.1, 3.2.5]

Contains information about received messages

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

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

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

```
F08  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>
```

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

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

Returns after a matching message is ready to be received.

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

13 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.

14 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.

15 PITFALLS

DEADLOCK

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

Process 0
`call MPI_Ssend(..., 1, ...)`
`call MPI_Recv(..., 1, ...)`

Process 1
`call MPI_Ssend(..., 0, ...)`
`call MPI_Recv(..., 0, ...)`

Mitigation Strategies

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

16 EXERCISES

Exercise 2 – Point-to-Point Communication
2.1 Global Summation – Sequential

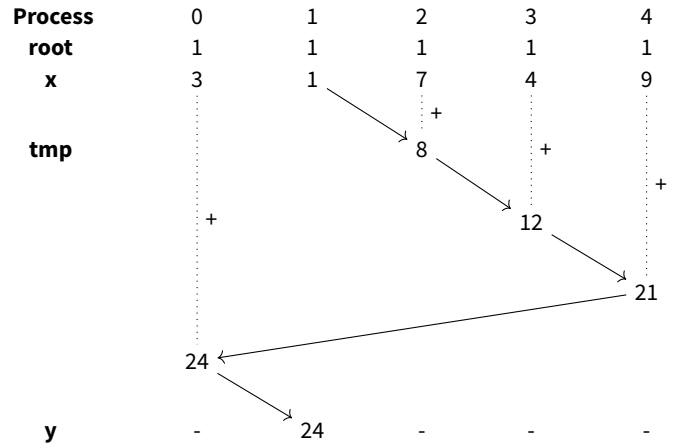
In the file `global_sum.{c|c++|f90}` implement the function/subroutine `global_sum_sequential(x, y, root, comm)`. It will be called by all processes on the communicator `comm` and on each one accepts an integer `x`. It should compute the global sum of all values of `x` across all processes and return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

BLOCKING COLLECTIVE COMMUNICATION

17 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

18 SYNCHRONIZATION

BARRIER [MPI-3.1, 5.3]

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

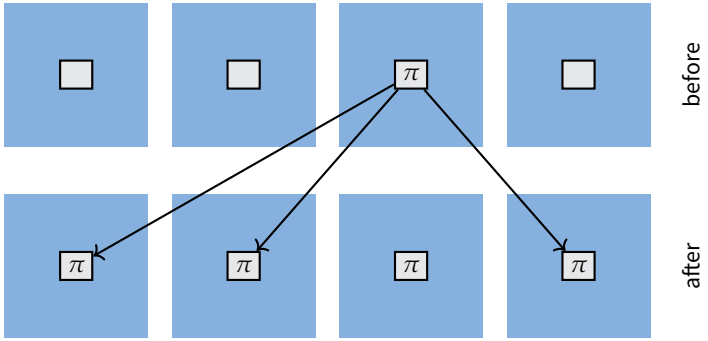
```
C  int MPI_Barrier(MPI_Comm comm)
```

```
F08 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.

19 DATA MOVEMENT

BROADCAST [MPI-3.1, 5.4]

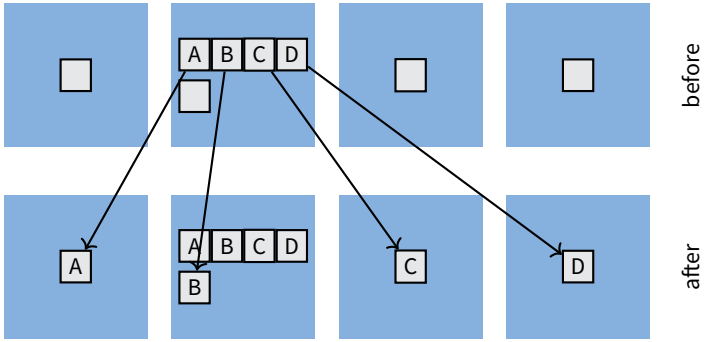



```
*  MPI_Bcast( <buffer>, <root> )
```

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

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

SCATTER [MPI-3.1, 5.6]

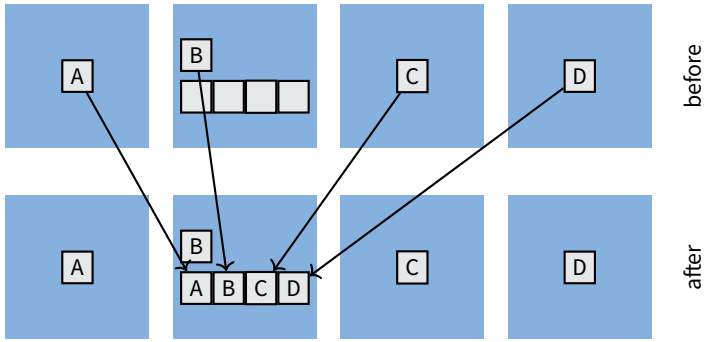



```
*  MPI_Scatter( <send buffer>, <receive buffer>, <root> )
```

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

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

GATHER [MPI-3.1, 5.5]

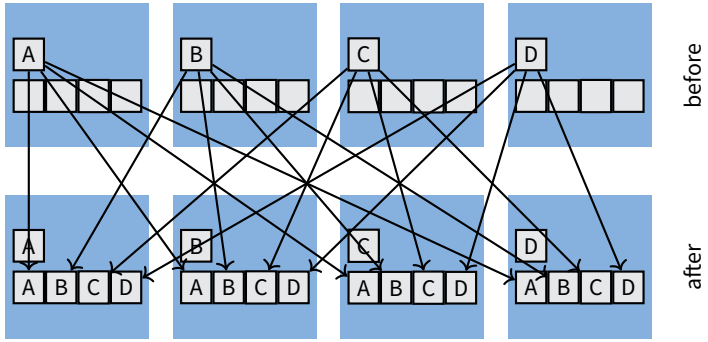



```
*  MPI_Gather( <send buffer>, <receive buffer>, <root> )
```

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

```
F08 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]

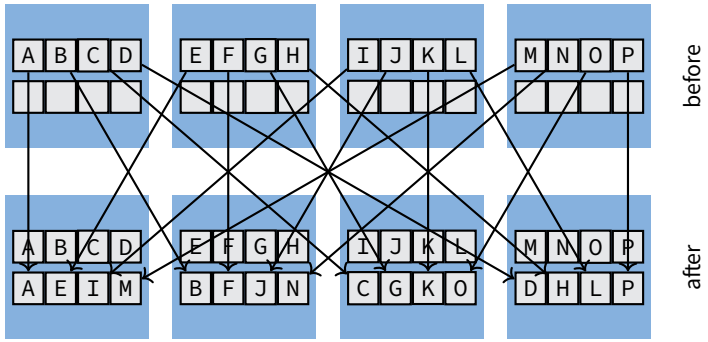



```
*  MPI_Allgather( <send buffer>, <receive buffer>, <communicator>
     ↪ )
```

```
C  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>
     ↪ )
```

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

```
F08  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
```

20 REDUCTIONS

GLOBAL REDUCTION OPERATIONS [MPI-3.1, 5.9]

Associative operations over distributed data

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

d_i, data of process with rank i

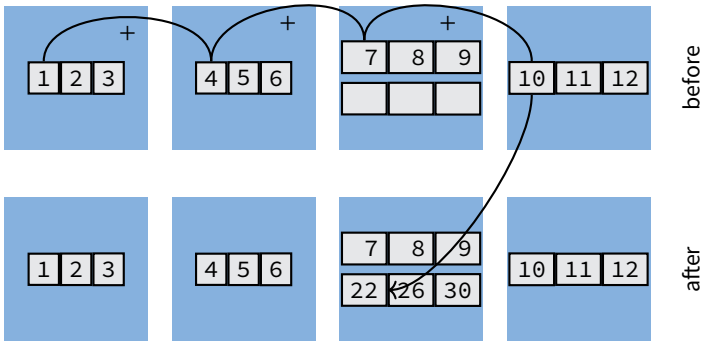
$\oplus$, associative operation

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]

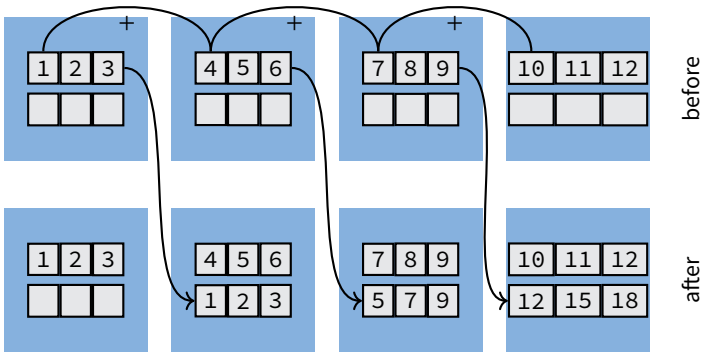



```
*    MPI_Reduce( <send buffer>, <receive buffer>, <operation>,
       ↪ <root> )
```

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

```
F08  MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm,
       ↪ ierror)
     type(*), dimension(..), intent(in) :: sendbuf
     type(*), dimension(..) :: recvbuf
     integer, intent(in) :: count, root
     type(MPI_Datatype), intent(in) :: datatype
     type(MPI_Op), intent(in) :: op
     type(MPI_Comm), intent(in) :: comm
     integer, optional, intent(out) :: ierror
```

EXCLUSIVE SCAN [MPI-3.1, 5.11.2]




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

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

```
F08 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
```

PREDEFINED OPERATIONS [MPI-3.1, 5.9.2]

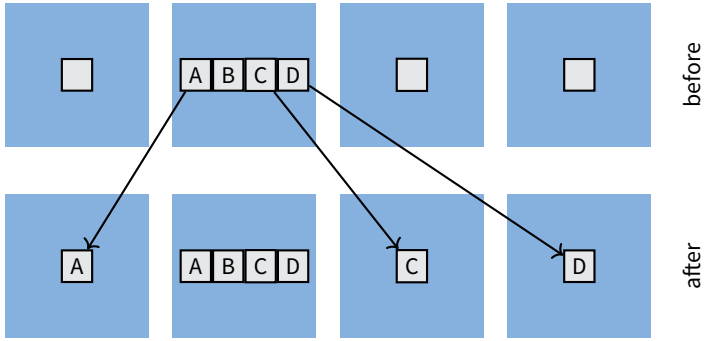
Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and the first rank that holds it [MPI-3.1, 5.9.4]
MPI_MINLOC	Minimum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

21 IN PLACE MODE

IN PLACE MODE

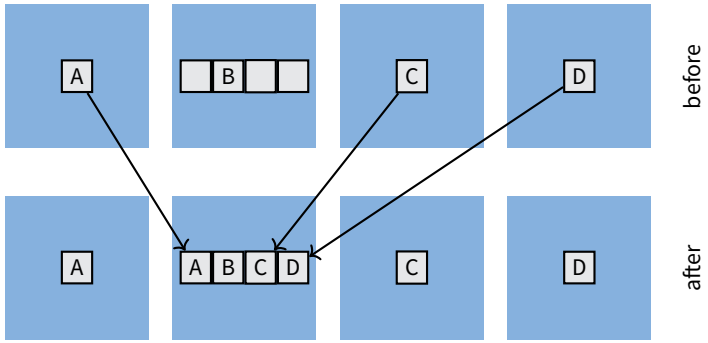
- Collectives can be used in *in place mode* with only one buffer to conserve memory
- The special value `MPI_IN_PLACE` is used in place of either the send or receive buffer address
- `count` and `datatype` of that buffer are ignored

IN PLACE SCATTER



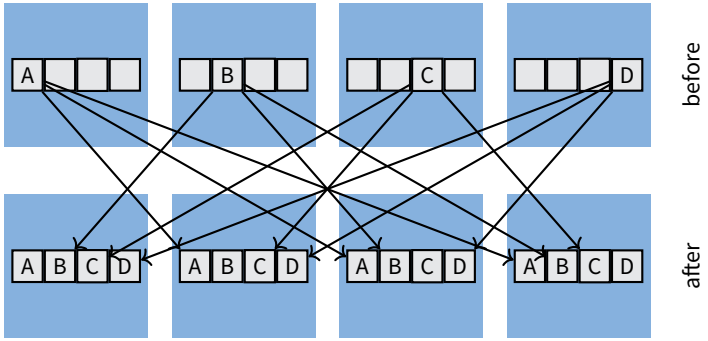
If `MPI_IN_PLACE` is used for `recvbuf` on the root process, `recvcount` and `recvtype` are ignored and the root process does not send data to itself

IN PLACE GATHER



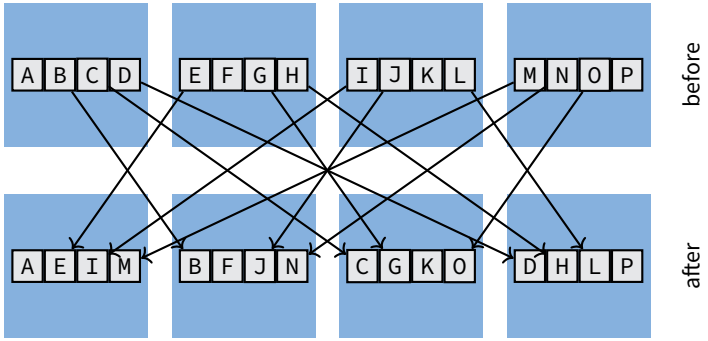
If `MPI_IN_PLACE` is used for `sendbuf` on the root process, `sendcount` and `sendtype` are ignored on the root process and the root process will not send data to itself.

IN PLACE GATHER-TO-ALL



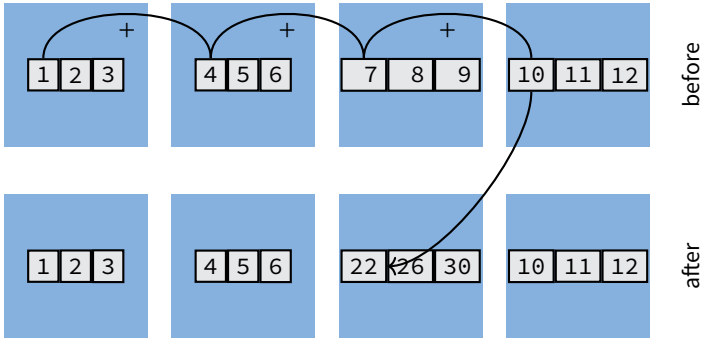
If `MPI_IN_PLACE` is used for `sendbuf` on all processes, `sendcount` and `sendtype` are ignored and the input data is assumed to already be in the correct position in `recvbuf`.

IN PLACE ALL-TO-ALL SCATTER/GATHER



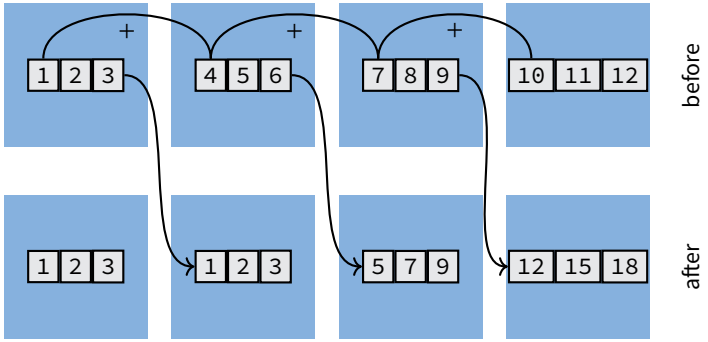
If `MPI_IN_PLACE` is used for `sendbuf` on all processes, `sendcount` and `sendtype` are ignored and the input data is assumed to already be in the correct position in `recvbuf`.

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.

22 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

23 EXERCISES

Exercise 3 – Collective Communication

3.1 Monte Carlo Calculation of π

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

Hints:

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

Some geometry:

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

Random numbers:

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

F08
```f90
real :: x
call random_number(x)
```

Use: `MPI_Wtime`, `MPI_Reduce`

3.2 Redistribution of Points with Collectives

Write a program `redistribute.{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 `nranks` processes: process i gets partition $[i/nranks, (i+1)/nranks)$.
3. Redistribute the points, so that every process is left with only those particles that lie inside its partition.

Guidelines:

- Use collectives, either `MPI_Gather` and `MPI_Scatter` or `MPI_Alltoall(v)` (see 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

C
```c
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)
```

F08
```f90
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 V

THREAD COMPLIANCE

24 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`

25 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]

C
```
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:

C
```
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*:

C
```
int MPI_Is_thread_main(int* flag)
```

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

26 MATCHING PROBE AND RECEIVE

MATCHING PROBE [MPI-3.1, 3.8.2]

C
```
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]

```
C   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

27 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 VI

FIRST STEPS WITH OPENMP

28 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*[3])

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

Current version of the specification: 4.5 (November 2015)

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

2017 Version 5.0 *Preview 2* (not standardized yet), memory model, base language compatibility, allocators, extended task and devices facilities

COVERAGE

- Directives
  - Directive Format ✓

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

  - Conditional Compilation ✓
  - Internal Control Variables ✓
  - Array Sections
  - `parallel` Construct ✓
  - Canonical Loop Form ✓
  - Worksharing Constructs ✓
  - SIMD Constructs
  - Tasking Constructs ✓
  - Device Constructs
  - Combined Constructs (✓)
  - `if` Clause ✓
  - Master and Synchronization Constructs and Clauses ✓
  - Cancellation Constructs
  - Data Environment ✓
  - `declare reduction` Directive
  - Nesting of Regions
- Runtime Library Routines
  - Runtime Library Definitions (✓)
  - Execution Environment Routines (✓)
  - Lock Routines ✓
  - Timing Routines
  - Device Memory Routines
  - …
- Environment Variables (✓)
- …

LITERATURE

Official Resources

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

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

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

Additional Literature

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

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

29 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*, generated when a *thread* encounters a `task`, `taskloop`, `parallel`, `target` or `teams` *construct*.

LANGUAGE

Terminology: Base Language

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

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

Terminology: Base Program

A program written in the *base language*.

Terminology: OpenMP Program

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

Terminology: Directive

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

30 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

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

GNU Fortran Compiler OpenMP Command Line Switch

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

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

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

Compiling Programs with Hybrid Parallelization

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

RUNTIME LIBRARY DEFINITIONS [OpenMP-4.5, 3.1]

C/C++ Runtime Library Definitions

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

C

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

Fortran Runtime Library Definitions

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

F77

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

or a Fortran 90 module

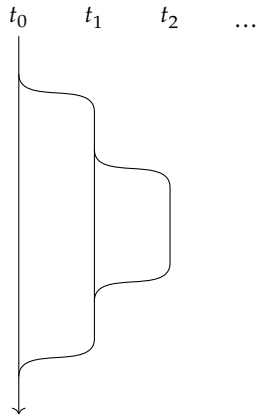
F08

```
use omp_lib
```

31 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-4.5, 2.1]

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

C

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

- 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-4.5, 2.1.1, 2.1.2]

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

- Directives are case-insensitive

Fixed Form Sentinels

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

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

Free Form Sentinel

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

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

CONDITIONAL COMPILATION [OpenMP-4.5, 2.2]

C Preprocessor Macro

C
```
#define _OPENMP yyyymm
```

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

Fortran Fixed Form Sentinels

F08
```
!$ | *$ | c$
```

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

Fortran Free Form Sentinel

F08
```
!$
```

- 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-4.5, 2.5]

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

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

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

THREAD COORDINATES [OpenMP-4.5, 3.2.2, 3.2.4]

Team size

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

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

Returns the number of threads in the current team

Thread number

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

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

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

A FIRST OPENMP PROGRAM

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

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

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

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

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.
```

```
F08
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
```

32 EXERCISES

Exercise 4 – Warm Up

4.1 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.

4.2 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):

$$\mathtt{dot}(\mathbf{x}, \mathbf{y}) = \sum_i x_i y_i.$$

Compile the file into a program and run it to test your implementation.

PART VII

LOW-LEVEL OPENMP CONCEPTS

33 INTRODUCTION

MAGIC

Any sufficiently advanced technology is indistinguishable from magic.
(Arthur C. Clarke[4])

INTERNAL CONTROL VARIABLES [OpenMP-4.5, 2.3]

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()`)
- 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-4.5, 4.12]

PARALLELISM CLAUSES [OpenMP-4.5, 2.5, 2.12]

`if` Clause

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

F08
```
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)
```

F08
```
num_threads(scalar-integer-expression)
```

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

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

EXAMPLE

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

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

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

F08
```
!$omp parallel num_threads( 64 )
statement1
statement2
statement3
!$omp end parallel
```

CONTROLLING THE *nthreads-var* ICV

`omp_set_num_threads` API Routine [OpenMP-4.5, 3.2.1]

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

F08
```
subroutine omp_set_num_threads(num_threads)
integer num_threads
```

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

`omp_get_max_threads` API Routine [OpenMP-4.5, 3.2.3]

C
```
int omp_get_max_threads(void);
```

F08
```
integer function omp_get_max_threads()
```

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

THREAD LIMIT & DYNAMIC ADJUSTMENT

`omp_get_thread_limit` API Routine [OpenMP-4.5, 3.2.14]

C
```
int omp_get_thread_limit(void);
```

F08
```
integer function omp_get_thread_limit()
```

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

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

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

F08
```
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-4.5, 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?

34 EXERCISES

Exercise 5 – Controlling `parallel`
5.1 Controlling the Number of Threads

Use `hello_openmp.{c|c++|f90}` 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`.

5.2 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.

35 DATA ENVIRONMENT

DATA-SHARING ATTRIBUTES [OpenMP-4.5, 2.15.1]

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-4.5, 2.15.1]

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

Pre-determined

- Variables with automatic storage duration declared inside the construct are private (C and C++)
- 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.

```
C++
struct foo {
  static int m;
  static void bar() {
    int *p = new int(4);
    m = 8;
    #pragma omp parallel
    {
      int t; static int s; // t is private, s is shared
      t = omp_get_thread_num(); assert(t ==
        ↪ omp_get_thread_num());
      t = *p; assert(t == 4); // p and *p are shared
      t = m; assert(t == 8); // m is shared
    }
    delete p;
  }
};
```

```
F08
subroutine foo(x)
  integer :: x(0:), i

  !$omp parallel
    do i = 0, 2 ! i is private
      if (3 * omp_get_thread_num() + i <= ubound(x, 1)) &
        ! x is shared
        x(3 * omp_get_thread_num() + i) = omp_get_thread_num()
    end do
  !$omp end parallel

  ! assertion: x == (/ 0, 0, 0, 1, 1, 1, 2, ... /)
end subroutine
```

DATA-SHARING ATTRIBUTE RULES II [OpenMP-4.5, 2.15.2]

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

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

```
C++
void foo() {


}

#pragma omp parallel
  foo()
```

```
F08
```

THE SHARED CLAUSE [OpenMP-4.5, 2.15.3.2]

```
*  shared(list)
```

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

THE PRIVATE CLAUSE [OpenMP-4.5, 2.15.3.3]

```
*  private(list)
```

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

FIRSTPRIVATE CLAUSE [OpenMP-4.5, 2.15.3.4]

```
*  firstprivate(list)
```

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

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

DEFAULT CLAUSE [OpenMP-4.5, 2.15.3.1]

C and C++

```
C  default(shared | none)
```

Fortran

```
F08  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-4.5, 2.15.3.6]

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

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

36 EXERCISES

Exercise 6 – Data-sharing Attributes

6.1 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.

37 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-4.5, 2.13.3]

```
C  #pragma omp barrier
```

F08
```
!$omp barrier
```

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

THE CRITICAL CONSTRUCT [OpenMP-4.5, 2.13.2]

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

F08
```
!$omp critical [(name)]
  structured-block
!$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, *name*s live in their own name space.
- In Fortran, *name*s of critical regions can collide with other identifiers.

LOCK ROUTINES [OpenMP-4.5, 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);
```

F08
```
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
- Initialize a lock before first use
- Destroy a lock when it is no longer needed
- Lock and unlock using the `set` and `unset` routines
- `set` blocks if lock is already set

THE ATOMIC AND FLUSH CONSTRUCTS [OpenMP-4.5, 2.13.6, 2.13.7]

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

38 EXERCISES

Exercise 7 – Thread Synchronization
7.1 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 VIII
WORKSHARING

39 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

40 THE SINGLE CONSTRUCT

THE SINGLE CONSTRUCT [OpenMP-4.5, 2.7.3]

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

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

41 SINGLE CLAUSES

IMPLICIT BARRIERS & THE NOWAIT CLAUSE [OpenMP-4.5, 2.7]

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

THE COPYPRIVATE CLAUSE [OpenMP-4.5, 2.15.4.2]

*
```
copyprivate(list)
```

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

42 THE LOOP CONSTRUCT

LOOP CONSTRUCT [OpenMP-4.5, 2.7.1]

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

F08
```
!$omp do [clause[[,] clause]...]
  do-loops
[!$omp end do [nowait]]
```

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

Data-environment clauses

- `private`
- `firstprivate`
- `lastprivate`
- `reduction`

Loop-specific clauses

- `schedule`
- `collapse`

CANONICAL LOOP FORM [OpenMP-4.5, 2.6]

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

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

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

- `var` must not be modified by the loop body
- The number of iterations of the loop must be known beforehand

43 LOOP CLAUSES

THE COLLAPSE CLAUSE [OpenMP-4.5, 2.7.1]

```
* collapse(n)
```

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

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

THE SCHEDULE CLAUSE [OpenMP-4.5, 2.7.1]

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

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

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

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

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

`auto` Let the compiler and runtime decide.

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

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

44 EXERCISES

Exercise 8 – Loop Worksharing

8.1 Generalized Vector Addition (`axpy`)

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

8.2 Dot Product

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

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

45 WORKSHARE CONSTRUCT

WORKSHARE (FORTRAN ONLY) [OpenMP-4.5, 2.7.4]

```
!$omp workshare
  structured-block
!$omp end workshare [nowait]
```
F08

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.

46 EXERCISES

Exercise 9 – `workshare` Construct

9.1 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.

9.2 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.

47 COMBINED CONSTRUCTS

COMBINED CONSTRUCTS [OpenMP-4.5, 2.11]

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

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

can be turned into

```
C   #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 IX

TASK WORKSHARING

48 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

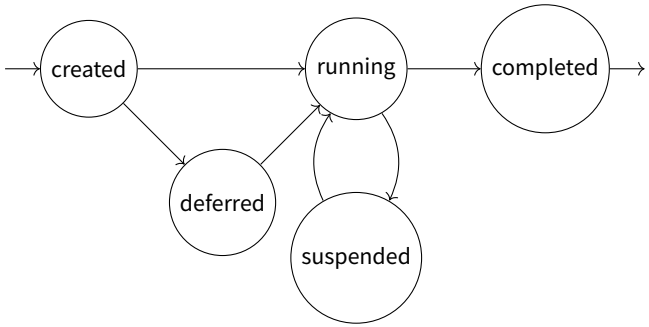
A *task* that is the *child task* of a *task region* or of one of its *descendent task regions*.

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*



49 THE TASK CONSTRUCT

THE TASK CONSTRUCT [OpenMP-4.5, 2.9.1]

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

```
F08 !$omp task [clause[[,] clause]...]
      structured-block
    !$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-4.5, 2.15.1.1]

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

If no `default` clause is present and

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

50 TASK CLAUSES

THE IF CLAUSE [OpenMP-4.5, 2.9.1]

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

If the scalar expression evaluates to *false*:

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

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-4.5, 2.9.1]

```
* 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-4.5, 2.9.1]

```
* untied
```

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

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-4.5, 2.9.1]

```
* priority(priority-value)
```

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

THE DEPEND CLAUSE [OpenMP-4.5, 2.13.9]

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

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

Terminology: Dependent Task

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

51 TASK SCHEDULING

TASK SCHEDULING POLICY [OpenMP-4.5, 2.9.5]

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

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

Task scheduling points

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

THE TASKYIELD CONSTRUCT [OpenMP-4.5, 2.9.4]

C
```
#pragma omp taskyield
```

F08
```
!$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

52 TASK SYNCHRONIZATION

THE TASKWAIT & TASKGROUP CONSTRUCTS [OpenMP-4.5, 2.13.4, 2.13.5]

C
```
#pragma omp taskwait
```

F08
```
!$omp taskwait
```

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

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

F08
```
!$omp taskgroup
  structured-block
!$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.

53 EXERCISES

Exercise 10 – Task worksharing

10.1 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.

10.2 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.

10.3 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 X

WRAP-UP

ALTERNATIVES

Horizontal Alternatives

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

Parallel frameworks Charm++, HPX, StarPU

Shared memory tasking Cilk, TBB

Accelerators CUDA, OpenCL, OpenACC, OmpSs

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

Vertical Alternatives

Applications Gromacs, CP2K, ANSYS, OpenFOAM

Numerics libraries PETSc, Trilinos, DUNE, FEniCS

Big data frameworks Hadoop, Spark

JSC COURSE PROGRAMME

- Introduction to the usage and programming of supercomputer resources in Jülich, **22 – 23 November**
- Advanced Parallel Programming with MPI and OpenMP, **26 – 28 November**
- Introduction to GPU programming using OpenACC, **29 – 30 October**
- And more, see `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Workshops/Courses/courses_node.html`

COLOPHON

This document was typeset using

- LuaLaTeX 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 .