JÜLICH FORSCHUNGSZENTRUM

Parallel programming

with MPI and OpenMP

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Learning objectives

At the end of this course, you will be able to

- Explain the main architectures in HPC today
- Discuss domain decomposition techniques
- Write parallel applications using the Message Passing Interface
- Use any of the three communication paradigms of MPI
- Use the interface for parallel I/O provided by MPI
- Parallelize applications using OpenMP constructs
- Use MPI and OpenMP parallelization in the same application

Acknowledgments

This course is influenced by courses and input of a number of people over the years. I wish to thank some of them explicitly for their input.

- Rolf Rabenseifner for his comprehensive course on MPI and OpenMP
- Bernd Mohr for his input on parallel architectures and OpenMP
- Boris Orth for his work on our concerted MPI courses of the past

JÜLICH
FORSCHUNGSZENTRUM

Parallel programming

Parallel architectures and programming models

Course GSTU2009 | Marc-André Hermanns

Learning objectives

After this lesson, you will be able to

- List the most common computing architectures in HPC
- List the most common interconnect topologies in HPC
- Select a programming model for a given computing architecture
- Discuss the advantages and disadvantages of the MPI, OpenMP and PGAS programming model
- Evaluate how a parallelization scheme is influenced by the computing platform

Supercomputing drives computational science

- Three pillars of scientific research
  - Theoretical science
  - Experimental science
  - Computational science
- Computational science is used wherever phenomena are
  - too complex to be reliably predicted by theory
  - too expensive or too dangerous to be investigated by experiments
- High-performance computing (HPC) has become an indispensable tool for the solution of many fundamental problems in science and engineering

Why go parallel?

- Physical limits of sequential architectures are mostly reached
- The clock cycle time cannot be made arbitrarily small
  - High clock frequencies need a small physical size of the system due to the finite speed of light
  - High clock frequencies and high integration density lead to a high heat density, which must be dissipated out of the system
- The memory-to-CPU bandwidth is limited even stronger
  - Increasing gap between processor clock cycle and memory access times
  - Processors deliver a continuously decreasing fraction of their peak performance
- Your application requires huge amounts of memory
- You need to cut down total time to solution
- You want to solve a more complex problem

Parallel architecture concepts

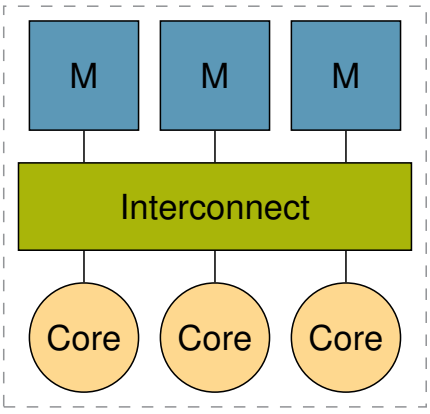
- Parallel processing concepts:
  - Pipelines, vector registers and instructions
    → vector computing
  - Functional parallelism
    → superscalar processors, very long instruction word (VLIW)
  - Multithreading
  - Simultaneous Multithreading (SMT), Hyperthreading
  - Shared-memory and distributed-memory parallel computers
  - Hybrid systems
- Memory access concepts:
  - Cache-based
  - Vector access via several memory banks
  - Pre-loading, pre-fetching

Shared and distributed memory systems

- Shared-memory systems
  - Single address space shared by multiple processors
  - SMP = symmetric multi-processing
- Distributed-memory systems
  - Separate nodes have separate address spaces
  - Cluster
  - MPP = massively parallel processing systems
- Hybrid systems
  - Combining both concepts
  - Cluster - Network of Workstations
    - #nodes $>$ #processors/node
  - Constellation
    - #processors/node $\geq$ #nodes

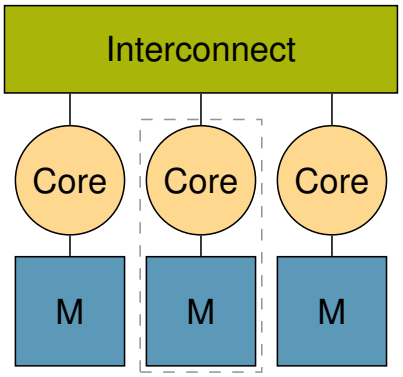
Shared-memory systems

- All cores have access to all memory banks (shared address space)
- Symmetric multi-processing (SMP)
- Concepts:
  - Uniform memory access (UMA)
  - Distributed shared memory (DSM)
  - Cache-coherent NUMA (ccNUMA)
- Programming Models:
  - Implicit communication via shared data
  - Explicit synchronization
  - Pthreads, OpenMP, PGAS, [MPI]

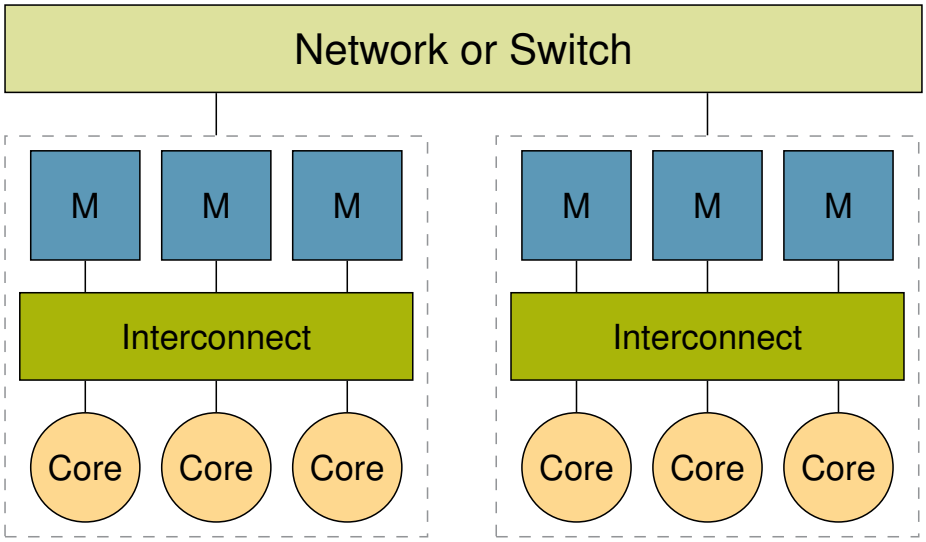


Distributed-memory systems

- All nodes/PEs
  - own local memory
  - interconnected by a communication network
- Concepts:
  - Non-uniform memory access (NUMA)
  - Remote direct memory access (RDMA)
  - No remote memory access (NORMA)
- Programming Models:
  - Explicit data distribution, communication, synchronization
  - MPI, PGAS, PVM



Hybrid systems I



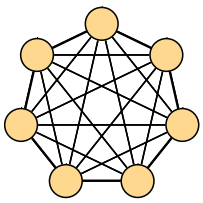
Hybrid systems II

- Due to the multi-core development almost all modern HPC systems are clusters of SMPs
  - Non-Uniform Memory Access (NUMA)
  - Shared-memory inside each node
  - Distributed-memory between nodes
- Programming models
  - Message-passing between all cores
  - Message-passing between nodes, multi-threading inside nodes
  - Partitioned global address space (PGAS) between all cores

Interconnection topologies I

Complete communication graph

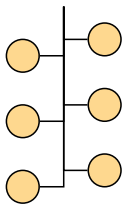
- Each node is directly connected to every other node
- Dedicated link between each pair of nodes
- Grows expensive with a large number of nodes



Interconnection topologies II

Bus communication

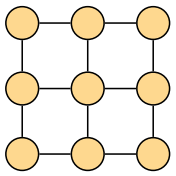
- Nodes communicate over a common bus, either directly or via a shared memory attached to the bus
- Concurrent communication may interfere
- Grows inefficient with a large number of nodes



Interconnection topologies III

Regular n-dimensional grids

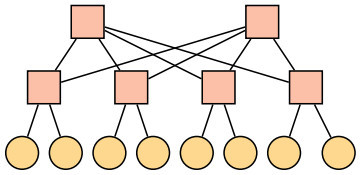
- Direct link to neighbors
- Grid
  - Each node has 0,1, or 2 neighbors per dimension
- Torus
  - Each node has exactly 2 neighbors per dimension
- Efficient nearest neighbor communication
- Suitable for a large number of nodes



Interconnection topologies IV

Trees

- Nodes on leaves of the tree
- Special form: fat tree
- Cost effective
- Suitable for large number of nodes



Parallelization

- Two major computation resources:
  - Processor
  - Memory
- Parallelization means
  - Distributing work among processors
  - Synchronization of the distributed work
- If memory is distributed it also means
  - Distributing data
  - Communicating data between local and remote processors

Basic parallel programming paradigm: SPMD

- SPMD = Single Program, Multiple Data
- Programmer writes *one* program which is executed on all processors (contrary *e.g.* to a client-server model)
- Basic paradigm for implementation of parallel programs
- MPMD (= Multiple Programs, Multiple Data) can be emulated with SPMD, *e.g.*

```
if (my_id == 42) then
  do_something()
else
  do_something_else()
endif
```


Parallel programming model

- Abstract model for the programming of (classes of) parallel computer systems
- Offers combined methods for
  - Distribution of work & data
  - Communication and synchronization
- Independent of a specific programming language

Parallel programming models

- Message Passing Interface (MPI)
  - Distributed-memory parallelization
  - User specifies how data and work are distributed
  - User specifies how and when data is communicated
  - By explicit calls to MPI library functions
- OpenMP
  - Shared-memory parallelization
  - Automatic or directed work distribution (*e.g.* loop parallelization)
  - No explicit data distribution and communication
  - Synchronization is implicit (can also be user-defined)
- Partitioned Global Address Space (PGAS)
  - Data-parallel programming (single address space)
  - Distribution of data and work is done by the compiler
  - Possibly array statements expressing data-parallel operations
  - *e.g.*, HPF, CAF, UPC, X10, Chapel, Fortress, Titanium

Summary

- The main concepts are shared-memory and distributed-memory with some high-speed network
- There is a large variety of possible network topologies
  - Fat-tree, multi-dimensional torus, hyper-cube, . . .
  - Low-end platforms usually have less network complexity
  - High-end platforms often have more complex network topologies.
- Architectures and topologies underlie constant evolution
- Application development and optimization needs to be balanced between portability and efficiency

JÜLICH FORSCHUNGSZENTRUM

Parallel programming

Workload distribution and domain decomposition

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Learning objectives

At the end of this lesson, you will be able to

- Classify different decomposition techniques
- Evaluate the cost of different decomposition schemes
- Evaluate parallelization parameters
  - data locality, dimensionality of data decomposition, data placement, etc.
- Create an ($\leq n$)-dimensional block decomposition for n-dimensional problems

Dependencies in parallelization

- Time-like variables cannot be parallelized
  - Dependency between step N and $N + 1$
  - *e.g.* Signal processing, non-commutative operations ...
- Space-like variables can be parallelized
  - Complete information used in a partial step is available
  - *e.g.* Objects, ...
- Some variables have both characters
  - Work with time-like character needs to be serialized

Load balancing

- Parallelized parts of a program only perform well, if load between processes is balanced
- Load balancing has more than one variable
  - Time to process data
  - Time to communicate data
  - Time to calculate load balance
  - Local information on load distribution
- Different factors determine the *what* and *how* in balancing the load
- Different types of data may need to be handled differently

Work distribution

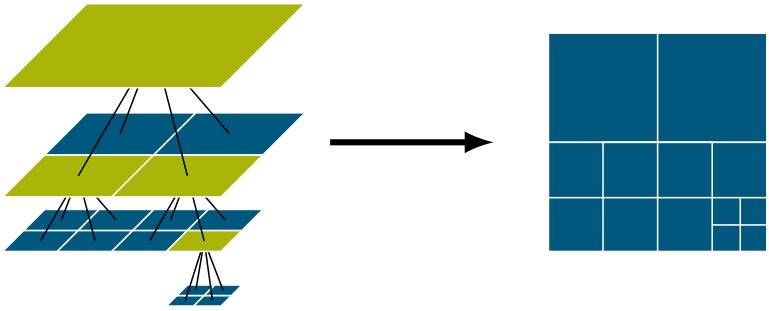
- Domain decomposition
  - Workload is evenly distributed over processes
  - Common case for homogeneous platforms
- Functional decomposition
  - Different functional parts are distributed to different groups of processes
  - Can be advantageous and/or necessary on heterogeneous platforms

Considerations on work distribution

- Partitioning costs
  - Calculation of partitions
  - Possible exchange of halo data
- Data-locality in partitions
- Stability of partitions
  - Static partitioning
  - Dynamic partitioning
- Shape of data and decomposition
  - Geometric decompositions
  - Graph decompositions
- Dimensionality of the problem

Geometric decomposition

- Utilize any information about the shape of the data to optimize decomposition
- When objects have an underlying geometrical shape, then geometric decomposition techniques are useful
  - finite elements, particles in a 3D volume, cells on a 2D game-plan, . . .
- Example: quad-tree (2D) or oct-tree (3D) decompositions
  - Divide space into quadrants (2D)/octants (3D)
  - If a quadrant contains too much load, refine successively



Graph decompositions

- If communication pattern is less regular, decomposition needs to be more general
  - Objects as vertices of a graph
  - Communication and adjacency information as edges
- Goal: Find optimal decomposition and mapping
  - No efficient algorithm known for exact solution
  - Heuristics to approximate work quite well
  - Good starting point: (Par)Metis

Dynamic decomposition and load balancing

- Complexity of partition may not be known in advance
- Complexity of partition may change over time

- Adaptive mesh refinement (AMR)
- Re-partitioning (run partitioner and re-distribute data)
- Overdecomposition
  - Create much more partitions than processes
  - Schedule new partition to process as it completed a partition
  - Better load balance in case of different partition complexities
  - More communication and synchronization

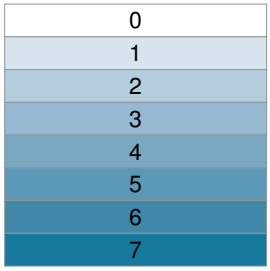
Border exchange

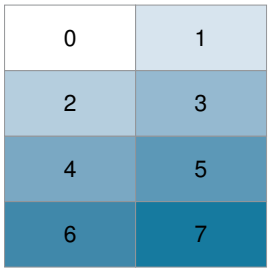
Objects needed in several partitions

- One process is owner of the object
- Other processes keep a copy
  - ghost cell, halo cell, shadow cell, . . .
- Usually local ghost cells are integrated in the local data structures
- Synchronization between processes needed between updates of cells
- Additional data exchange after computation phase

Partitioning example

n-dimensional block data decomposition





- less neighbors (≤ 2)
- fits any process count
- larger halo exchanges

- more neighbors (≤ 8)
- process count restrictions for balanced grid
- smaller halo exchanges

Cost factors

- halo width in each dimension
- matrix width in each dimension
- number of direct and indirect neighbors
  - cyclic boundaries equal number of neighbors for all partitions
  - direct neighbors have full halo exchange
  - indirect neighbors have corner halo exchange
- number of partitions in each dimension

Summary

- Parallelization is hard when too many dependencies exist between partitions
- Partitioning with distributed data induces communication and synchronization time
- Domain decomposition depends on shape and behavior of simulated data
- Work distribution costs depend on
  - Platform independent factors
    - halo size, message size, . . .
  - Platform dependent factors
    - process placement, memory requirements, . . .

Parallel programming

Introduction to MPI

Course GSTU2009 | Marc-André Hermanns

Learning objectives

At the end of this lesson, you will be able to

- explain the message-passing paradigm
- relate message-passing functionality to every-day communication
- create a minimal message-passing program with MPI
- query the most important communicator information
- define the blocking and non-blocking semantic in MPI
- list different communication paradigms and other functionality offered by MPI

What is MPI?

- Message-Passing Interface
- Industry standard for message passing systems
  - Manufacturers of parallel computers and researchers from universities, laboratories and industry are involved in its development
  - `http://www.mpi-forum.org`
- Implementation of the standard is a library of subroutines
  - Can be used with Fortran, C, and C++ programs
  - Contains more than 300 functions, but:
  - Only 6 are sufficient to write complete message-passing programs!
  - Open source implementations available
- MPI is widely used
- MPI-parallelized programs are *portable*

MPI history I

- Version 1.0 (1994)
  - Fortran77 and C language bindings
  - 129 functions
- Version 1.1 (1995)
  - Corrections and clarifications, minor changes
  - No additional functions
- Version 1.2 (1997)
  - Further corrections and clarifications for MPI-1
  - 1 additional function
- Version 2.0 (1997)
  - MPI-2 with new functionalities
  - 193 additional functions

MPI history II

- Version 2.1 (2008)
  - Corrections and clarifications
  - Unification of MPI 1.2 and 2.0 into a single document
- Version 2.2 (scheduled for 2009)
  - Further corrections and clarifications
  - New functionality with minor impact on implementations
- Version 3.0 (in discussion)
  - New functionality with major impact on implementations

Message-passing programming paradigm

- Each processor has its own private address space
  - Can physically be distributed or shared
- The variables of each sub-program have
  - the same name
  - but possibly different locations and different data contents
- Data exchange is explicit by passing a message

Messages

- Messages are packets of data exchanged between processes
- Necessary information for the message-passing system:
  - Sending process
  - Receiving process *i.e.*, the ranks
  - Source location
  - Destination location
  - Source data type
  - Destination data type
  - Source data size
  - Destination data size

Messages in MPI

- A message is an array of elements of a particular MPI datatype,

  described by a 3-tuple consisting of
  - Position in memory (buffer address)
  - Number of elements
  - MPI datatype
- MPI datatypes
  - Basic datatypes
  - Derived datatypes
- Derived datatypes can be built from basic or derived datatypes
- Basic MPI datatypes are different for C and Fortran

The message-passing infrastructure

- Every sub-program needs to be connected to a message-passing infrastructure
- Access to a message-passing infrastructure is comparable to a
  - Mailbox
  - Phone
  - Fax

MPI

- Program must be linked against an MPI library
- Program must be started with the MPI start-up mechanism
- MPI must be initialized and finalized in the program

Header files

C/C++

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

- Contain definition of
  - constants
  - functions
  - subroutines

Header files

Fortran

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

or

```
use mpi          ! Fortran 90
```

- Contain definition of
  - constants
  - functions
  - subroutines

Generic MPI function format

C/C++

```
error = MPI_Function(parameter,...);
```

- Error code is integer return value

MPI namespace

The `MPI_` and `PMPI_` prefix is reserved for MPI constants and routines, *i.e.* application variables and functions must not begin with `MPI_` or `PMPI_`

Generic MPI function format

Fortran

```
call MPI_FUNCTION(parameter,...,ierror)
```

- Error code is additional integer parameter

MPI namespace

The `MPI_` and `PMPI_` prefix is reserved for MPI constants and routines, *i.e.* application variables and functions must not begin with `MPI_` or `PMPI_`

Course GSTU2009 Introduction to MPI Slide 49

MPI initialization and finalization

C/C++

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

- Must be called as the first MPI function
  - Only exception: `MPI_Initialized`

```
int MPI_Finalize();
```

- Must be called as the last MPI function
  - Only exception: `MPI_Finalized`

MPI initialization and finalization

Fortran

```
MPI_INIT(ierr)
  INTEGER ierr
```

- Must be called as the first MPI function
  - Only exception: `MPI_Initialized`

```
MPI_FINALIZE(ierr)
  INTEGER ierr
```

- Must be called as the last MPI function
  - Only exception: `MPI_Finalized`

Addressing

- Messages need addresses that they are sent to
- Addresses are similar to
  - Mail addresses
  - Phone numbers
  - Fax numbers

MPI

- Communication within a group of processes is handled via a communicator
- Each process possesses a unique ID (rank) within each communicator
- The ranks of the MPI processes are used as addresses

Communicator basics

- Two predefined communicators
  - `MPI_COMM_WORLD` – contains all MPI processes
  - `MPI_COMM_SELF` – contains only the local process
- Communicators are defined by a process group and a specific context
  - Different communicators can have the same process group
- New communicators are created
  - by derivation from existing ones
  - from a process group by a collective operation
- Collective operations on a communicator
  - have to be called by all processes of the communicator
  - need to be called in the same order on every participating process

Handles

- MPI maintains internal data structures
- These are referenced by the user through handles
  - Example: `MPI_COMM_WORLD`
- Some MPI routines return handles to the user
- These can be used in subsequent MPI calls
- Handles are of a specific language dependent datatype
  - C special MPI `typedefs`
  - Fortran `INTEGER`

Accessing communicator information: rank

C/C++

- The rank identifies each process within a communicator

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

```
int myrank;
...
MPI_Init(&argc, &argv);
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
```

Accessing communicator information: rank

Fortran

- The rank identifies each process within a communicator

```
MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR
```

```
INTEGER :: myrank, ierr
...
call MPI_INIT(ierr)
...
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
...
```


Accessing communicator information: size

C/C++

- The number of processes in a communicator is its size

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

```
int size;
...
MPI_Init(&argc, &argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
...
```

Accessing communicator information: size

Fortran

- The number of processes in a communicator is its size

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

```
INTEGER :: size, ierr
...
call MPI_INIT(ierr)
...
call MPI_COMM_SIZE(MPI_COMM_WOLD, size, ierr)
...
```


Basic datatypes in MPI: C

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Basic datatypes in MPI: Fortran

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

MPI datatypes for non-standard C and Fortran datatypes

An MPI implementation may also support non-standard datatypes, if the host language supports these.

MPI datatype	C datatype
MPI_LONG_LONG_INT	longlong int (64 bit Integer)

MPI datatype	Fortran datatype
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_REAL<n>	REAL*<n>
MPI_INTEGER<n>	INTEGER*<n>
MPI_COMPLEX<n>	COMPLEX*<n>

JÜLICH FORSCHUNGSZENTRUM

Communication paradigms and parallel I/O

- Point-to-point communication
  - 1:1 communication, with explicit calls on two processes of a communicator
- Collective communication
  - n:m communication, with explicit calls on all processes of a communicator.
- One-sided communication
  - 1:1 communication, with explicit call on one process of a communicator
- Parallel I/O
  - Communication with the I/O subsystem (disk)

JÜLICH FORSCHUNGSZENTRUM

Blocking vs. non-blocking semantics

Blocking communication

The call to the MPI function will return to the calling function, when the overall call has completed in a sense that the provided user buffer is again free to use.

Non-blocking communication

The call to the MPI function will return as soon as possible to the calling function. The user-provided communication buffer must not be touched, before the communication has been completed by an appropriate call at the calling process.

Summary

- MPI is a settled industry standard on one hand, and still actively developed on the other
- MPI can be used with shared and distributed memory architectures
- MPI has language bindings for C, C++, and Fortran
- All communication in MPI is done with communicators
- Addressing of processes in MPI is done with a so-called rank that is unique for a process in a communicator
- MPI has multiple communication paradigms available, as well as an interface for structured parallel I/O

Parallel programming

Blocking point-to-point communication

Course GSTU2009 | Marc-André Hermanns

Learning objectives

After this lesson, you will be able to

- write message passing programs with MPI
- asses the up- and downsides of blocking point-to-point communication
- list different communication modes for MPI point-to-point communication
- evaluate use cases for different types of point-to-point communication modes

Point-to-point communication

- Communication between two processes
  - Note: A process can send messages to itself
- A source process sends a message to a destination process by a call to an MPI send routine
- A destination process needs to post a receive by a call to an MPI receive routine
- The destination process is specified by its rank in the communicator, e.g. `MPI_COMM_WORLD`
- Every message sent with a point-to-point call, needs to be matched by a receive.

Sending a message

C/C++

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

- BUF is the address of the message to be sent, with COUNT elements of type DATATYPE
- DEST is the rank of the destination process within the communicator COMM
- TAG is a marker used to distinguish between different messages

Sending a message

Fortran

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type>  BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

- BUF is the address of the message to be sent, with COUNT elements of type DATATYPE
- DEST is the rank of the destination process within the communicator COMM
- TAG is a marker used to distinguish between different messages

Receiving a message

C/C++

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

- BUF, COUNT and DATATYPE refer to the receive buffer
- SOURCE is the rank of the sending source process within the communicator COMM (can be MPI_ANY_SOURCE)
- TAG is a marker used to prescribe that only a message with the specified tag should be received (can be MPI_ANY_TAG)
- STATUS (output) contains information about the received message

Receiving a message

Fortran

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS,
        IERROR)
    <type>  BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
        STATUS(MPI_STATUS_SIZE), IERROR
```

- BUF, COUNT and DATATYPE refer to the receive buffer
- SOURCE is the rank of the sending source process within the communicator COMM (can be MPI_ANY_SOURCE)
- TAG is a marker used to prescribe that only a message with the specified tag should be received (can be MPI_ANY_TAG)
- STATUS (output) contains information about the received message

Probing a message

C/C++

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

- On unknown message size with no guaranteed upper bound
- Query of communication envelope
- SOURCE is the rank of the sending source process within the communicator COMM (can be MPI_ANY_SOURCE)
- TAG is a marker used to prescribe that only a message with the specified tag should be received (can be MPI_ANY_TAG)
- STATUS (output) contains information about the received message

Probing a message

Fortran

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
            IERROR
```

- On unknown message size with no guaranteed upper bound
- Query of communication envelope
- SOURCE is the rank of the sending source process within the communicator COMM (can be MPI_ANY_SOURCE)
- TAG is a marker used to prescribe that only a message with the specified tag should be received (can be MPI_ANY_TAG)
- STATUS (output) contains information about the received message

Course GSTU2009 Blocking point-to-point communication Slide 67

Communication envelope

C/C++

- All communication parameters, but the actual message payload is accounted to the message envelope
  - source rank, tag, message size, ...
- Envelope information is returned in `STATUS` variable

```
status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
```


Communication envelope

Fortran

- All communication parameters, but the actual message payload is accounted to the message envelope
  - source rank, tag, message size, ...
- Envelope information is returned in `STATUS` variable

```
status(MPI_SOURCE)
status(MPI_TAG)
status(MPI_ERROR)
```

Information on received message size

C/C++

- Message received doesn't need to fill the receive buffer
- Number of elements actually received can be found by querying the communication envelope (STATUS)

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


Information on received message size

Fortran

- Message received doesn't need to fill the receive buffer
- Number of elements actually received can be found by querying the communication envelope (STATUS)

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT,
        IERROR
```

Communication modes

Send modes

- Synchronous send ⇒ `MPI_Ssend`
- Buffered send ⇒ `MPI_Bsend`
- Standard send ⇒ `MPI_Send`
- Ready send ⇒ `MPI_Rsend`

Receive all modes

- Receive ⇒ `MPI_Recv`
- Probe ⇒ `MPI_Probe`

Completion conditions

Communication mode	**Completion condition**	**Remark**
Synchronous send `MPI_SSEND`	Only completes when the receive has started	
Buffered send `MPI_BSEND`	Always completes (unless an error occurs) irrespective of whether a receive has been posted	Needs a user-defined buffer to be attached with `MPI_BUFFER_ATTACH`
Standard Send `MPI_SEND`	Either synchronous or buffered	Uses an internal buffer
Ready Send `MPI_RSEND`	Always completes (unless an error occurs) irrespective of whether a receive has been posted	May be started only if the matching receive is already posted - not recommended
Receive `MPI_RECV`	Completes when a message has arrived	Same routine for all communication modes

Point-to-point communication requirements

- Communicator must be the same
- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
  - `MPI_ANY_SOURCE` also valid
- Tags must match
  - `MPI_ANY_TAG` also valid
- Message datatypes must match
- Receive buffer must be large enough to hold the message
  - If it is not, behavior is undefined
  - Can be larger than the data received

Wildcards

- Receiver can use wildcards for `SOURCE` and `TAG`
- To receive a message from any source use `MPI_ANY_SOURCE`
- To receive a message with any tag use `MPI_ANY_TAG`
- Actual source and tag are returned in the `STATUS` parameter

Rules I

Standard send (MPI_SEND)

- Minimal transfer time
- May be implemented as buffered or synchronous send, possibly depending on the message size - do not assume either case

Synchronous send (MPI_SSEND)

- High latency, best bandwidth
- Risk of idle times, serialization, deadlocks

Rules II

Buffered send (MPI_BSEND)

- Low latency, low bandwidth

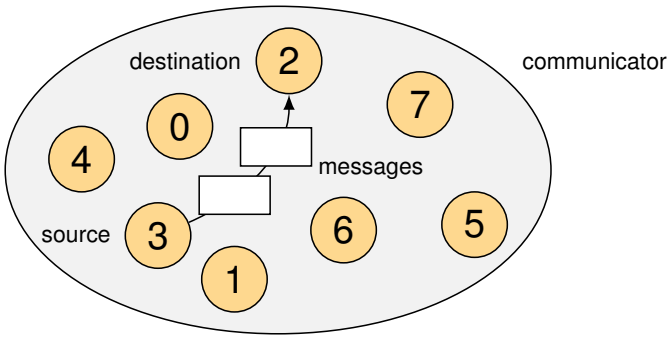
Ready send (MPI_RSEND)

- Use only if the logical flow of your parallel program permits it, *i.e.* if it is guaranteed that the receiving process is ready for the message

Point-to-point communication semantics

General properties of point-to-point communication

- Message order preservation
  - Messages sent from the same sender which match the same receive are received in the order they were sent
  - Messages do not overtake each other



- Progress
  - It is not possible for a matching send and receive pair to remain permanently outstanding

Point-to-point communication semantics II

General properties of point-to-point communication

- Fairness
  - Not guaranteed!
  - It is the programmers responsibility to prevent starvation of send or receive operations
- Resource limitations
  - Any pending communication operation consumes system resources (*e.g.* buffer space) that are limited
  - Errors may occur when a lack of resources prevents the execution of an MPI call

Pitfalls

- Deadlocks with *standard* sends
  - Overall application semantic causes deadlock when implementation chooses *synchronous* mode
  - Do not have all processes send or receive at the same time with blocking calls
  - Use special communication patterns
    - checked, odd-even, ...
- Performance penalties
  - Late Receiver – in *synchronous* mode, the sender waits for the receiver to post the receive call
  - Late Sender – receiving process blocks in call until the sender starts to send the message

Summary

- Point-to-point communication has several modes with different semantics
- A blocking call will return, when the user-provided buffer can be reused
- Non-determinism in message flow can be expressed with wildcards for source process and message tag on receiver side
- Message with same envelope will not overtake each other
  - Messages with different tags may overtake each other when received/probed explicitly (not recommended)
- Blocking point-to-point communication is straight-forward to implement and usually has a very low internal overhead
- Precautions have to be taken not to induce deadlocks or performance breakdowns into the communication

JÜLICH FORSCHUNGSZENTRUM

Parallel programming

Non-blocking point-to-point communication

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Learning objectives

At the end of this lesson, you will be able to

- differentiate between asynchronous and non-blocking communication
- use the non-blocking point-to-point communication interface to overlap communication with computation
- decide when to use the blocking or non-blocking communication calls
- circumnavigate typical pitfalls with non-blocking communication

Non-blocking communication

Non-blocking communication uses three phases:

1. Initiate communication
   - 'I' for 'immediate' (or 'initiate', or 'incomplete', or . . .)
   - Non-blocking routines return before the communication has completed
   - Non-blocking routines have the same arguments as their blocking counterparts except for an extra request argument
2. User-application can attend other work
   - Computation, communication, . . .
   - Goal: overlapping communication and computation
3. Complete communication
   - waiting for the communication request to finish

Request handles

- Used for non-blocking communication
- Request handles *must* be stored in local variables
  - C `MPI_Request`
  - Fortran `INTEGER`
- A non-blocking communication routine returns a value for the request handle
- This value is used by `MPI_WAIT` or `MPI_TEST` to test when the communication has completed
- If the communication has completed the request handle is set to `MPI_REQUEST_NULL`

Communication modes

Send modes

- Synchronous send ⇒ `MPI_Issend`
- Buffered send ⇒ `MPI_Ibsend`
- Standard send ⇒ `MPI_Isend`
- Ready send ⇒ `MPI_Irsend`

Receive all modes

- Receive ⇒ `MPI_Irecv`
- Probe ⇒ `MPI_Iprobe`

Non-blocking synchronous send

C/C++

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

- Non-blocking send routines have the same arguments as their blocking counterparts except for the extra `REQUEST` argument
- Send buffer `BUF` must not be accessed before the send has been successfully tested for completion with `MPI_WAIT` or `MPI_TEST`

Non-blocking synchronous send

Fortran

```
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
        IERROR)
    <type>  BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
        IERROR
```

- Non-blocking send routines have the same arguments as their blocking counterparts except for the extra `REQUEST` argument
- Send buffer `BUF` must not be accessed before the send has been successfully tested for completion with `MPI_WAIT` or `MPI_TEST`

Non-blocking receive

C/C++

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

- Non-blocking receive routine has the same arguments as its blocking counterpart, except the last parameter is not a `STATUS` but a `REQUEST` argument
- Receive buffer `BUF` must not be accessed before the receive has been successfully tested for completion with `MPI_WAIT` or `MPI_TEST`

Non-blocking receive

Fortran

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
        REQUEST, IERROR)
    <type>  BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,
        IERROR
```

- Non-blocking receive routine has the same arguments as its blocking counterpart, except the last parameter is not a `STATUS` but a `REQUEST` argument
- Receive buffer `BUF` must not be accessed before the receive has been successfully tested for completion with `MPI_WAIT` or `MPI_TEST`

Blocking vs non-blocking operations

- A blocking send can be used with a non-blocking receive, and vice versa
- Non-blocking sends can use any mode, just like the blocking counterparts
- Synchronous mode refers to the completion of the send (tested with `MPI_WAIT`/`MPI_TEST`), not to the initiation (`MPI_ISSEND` returns immediately!)
- A non-blocking operation immediately followed by a matching `MPI_WAIT` is equivalent to the blocking operation
- Fortran problems (see MPI, Chapter 16.2.2, pp. 461)

Communication completion

The user has two variants for checking communication completion

- Blocking
  - The call returns when communication is completed
  - The `MPI_Wait*` family of calls
- Non-blocking
  - The call returns immediately with a flag indicating completion
  - The `MPI_Test*` family of calls
  - If the flag is indicating a completed request, the communication needs no further completion
- Both calls return when `MPI_REQUEST_NULL` is tested for completion

Waiting for completion

- Used where synchronization required
- `MPI_Wait` – wait for a single request
- `MPI_Waitany` – wait for a single request out of multiple requests given
- `MPI_Waitsome` – wait for one or more requests out of multiple requests given
- `MPI_Waitall` – wait for all given requests to complete

Testing for completion

- Used where synchronization required
- `MPI_Test` – testing for a single request
- `MPI_Testany` – testing for a single request out of multiple requests given
- `MPI_Testsome` – testing for one or more requests out of multiple requests given
- `MPI_Testall` – testing for all given requests to be completed

Overlapping communication and computation

- The implementation may choose not to overlap communication and computation
- Progress may only be done inside of MPI calls

⇒ Not all platforms perform significantly better than well placed blocking communication

  - More internal overhead for communication handling

- If hardware support is present, application performance may significantly improve due to overlap
- Initiation of communication should be placed as early as possible
- Synchronization/completion should be placed as late as possible

Summary

- Non-blocking refers to the initiation of the communication
- Calls need to be completed by a separate call
- Progress is only guaranteed within MPI calls
- User can test or wait for completion
- Buffers may only be accessed after successful completion
- Handles refer to pending communication requests
- Single and multiple requests can be tested or waited for

Parallel programming

Derived Datatypes

Course GSTU2009 | Marc-André Hermanns

Learning Objectives

At the end of this lesson, you will be able to

- Define what MPI datatypes are
- Derive MPI datatypes for arbitrary memory layouts
- Use MPI derived datatypes in MPI communication calls
- Evaluate where MPI datatypes can be helpful
- Differentiate between size and extent of a datatype
- Resize datatypes by manipulating lower and upper bound markers

Motivation

- With MPI communication calls only multiple consecutive elements of the same type can be sent
- Buffers may be non-contiguous in memory
  - Sending only the real/imaginary part of a buffer of complex doubles
  - Sending sub-blocks of matrices
- Buffers may be of mixed type
  - User defined data structures

Solutions without MPI derived datatypes

- Non-contiguous data of a single type
  - Consecutive MPI calls to send and receive each element in turn
    - Additional latency costs due to multiple calls
  - Copy data to a single buffer before sending it
    - Additional latency costs due to memory copy
- Contiguous data of mixed types
  - Consecutive MPI calls to send and receive each element in turn
    - Additional latency costs due to multiple calls
  - Use `MPI_BYTE` and `sizeof()` to avoid the type-matching rules
    - Not portable to a heterogeneous system

Derived datatypes

- General MPI datatypes describe a buffer layout in memory by specifying
  - A sequence of basic datatypes
  - A sequence of integer (byte) displacements
- Derived datatypes are derived from basic datatypes using constructors
- MPI datatypes are referenced by an opaque handle

MPI datatypes are opaque objects! Using the `sizeof()` operator on an MPI datatype handle will return the size of the handle, neither the size nor the extent of an MPI datatype.

Example: Data layout and datatype handle

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

Compiler

```
types[0] = MPI_INT;
blocklengths[0] = 3;
displacements[0] = 0;
types[1] = MPI_DOUBLE;
blocklengths[1] = 5;
displacements[1] = ...;
MPI_Type_create_struct(2,
    blocklengths, displacements,
    types, &buff_datatype);
MPI_Type_commit(&buff_datatype);
```

```
MPI_Send(&buffer, 1, buff_datatype, ...);
```

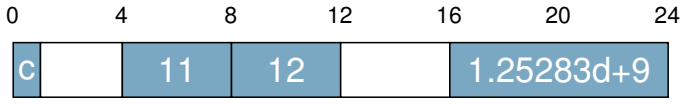
Creating a derived datatype: Type map

- Any derived datatype is defined by its type map
  - A list of basic datatypes
  - A list of displacements (positive, zero, or negative)
  - Any type matching is done by comparing type maps

basic datatype 0	disp. of datatype 0
basic datatype 1	disp. of datatype 1
...	...
basic datatype n	disp. of datatype n

Example of a type map

- A datatype with padding and holes



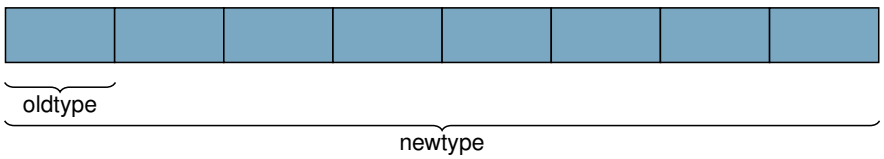
Basic datatype	Displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

Contiguous data

C/C++

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

- Simplest derived datatype
- Consists of a number of contiguous items of the same datatype

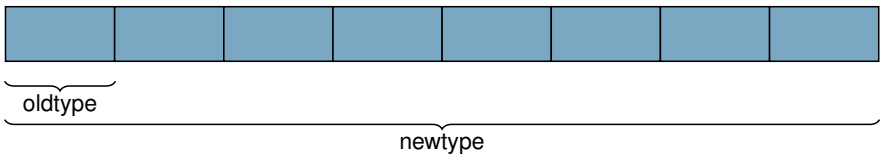


Contiguous data

Fortran

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

- Simplest derived datatype
- Consists of a number of contiguous items of the same datatype

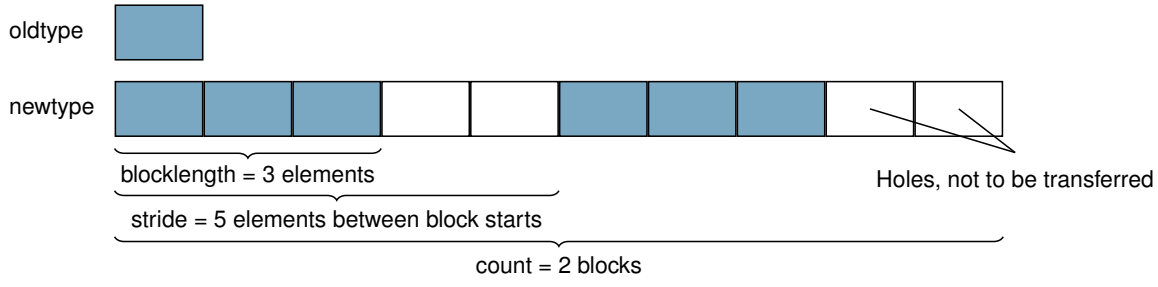


Vector data

C/C++

```
int MPI_Type_vector(int count, int blocklength,
        int stride, MPI_Datatype oldtype,
        MPI_Datatype *newtype)
```

- Consists of a number of elements of the same datatype repeated with a certain stride

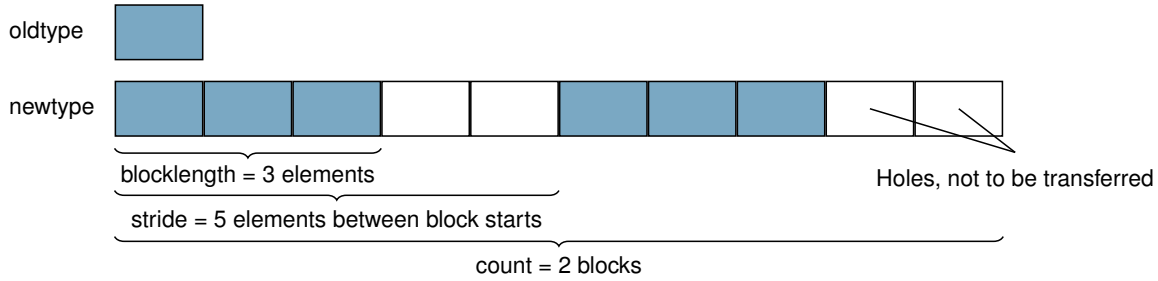


Vector data

Fortran

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
        NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
        NEWTYPE, IERROR
```

- Consists of a number of elements of the same datatype repeated with a certain stride

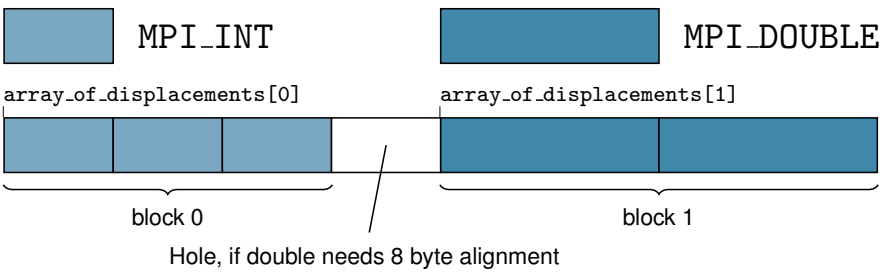


Struct data

C/C++

```
int MPI_Type_create_struct(int count,
        int *array_of_blocklengths,
        MPI_Aint *array_of_displacements,
        MPI_Datatype *array_of_types,
        MPI_Datatype *newtype)
```

- Data structure with different types

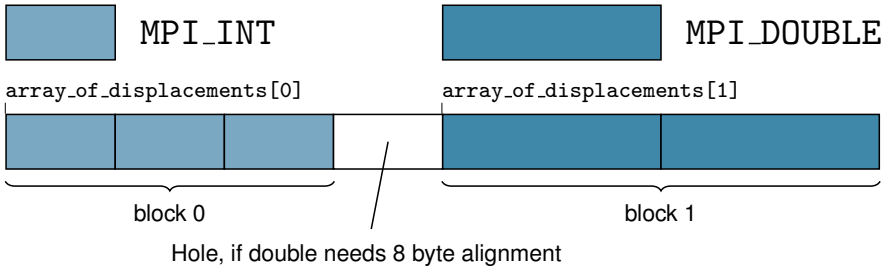


Struct data

Fortran

```
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
        ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES,
        NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
        ARRAY_OF_TYPES(*), NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

- Data structure with different types

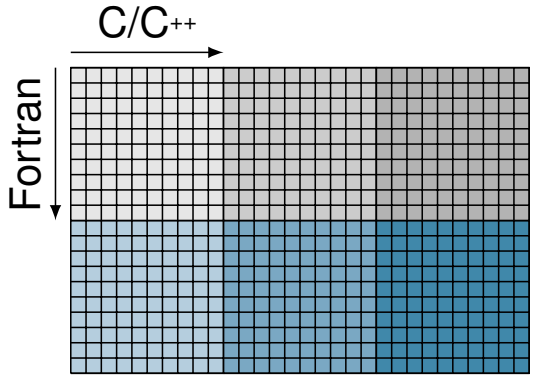


Sub-array data

C/C++

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

- N-dimensional sub-array of an N-dimensional array
- Fortran and C order allowed
- Fortran *and* C calls expect indices starting from 0

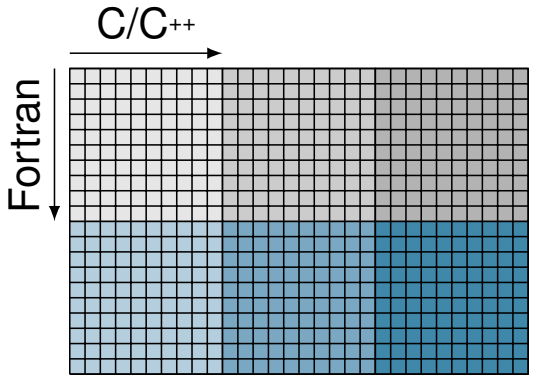


Sub-array data

Fortran

```
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES,
        ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER,
        OLDTYPE, NEWTYPE, IERROR)
    INTEGER NDIMS, ARRAY_OF_SIZES(*),
        ARRAY_OF_SUBSIZES(*), ARRAY_OF_STARTS(*), ORDER,
        OLDTYPE, NEWTYPE, IERROR
```

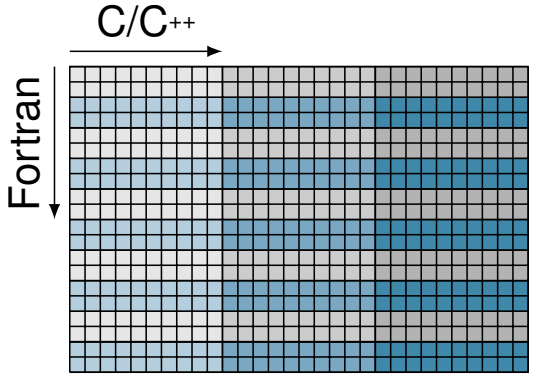
- N-dimensional sub-array of an N-dimensional array
- Fortran and C order allowed
- Fortran *and* C calls expect indices starting from 0



Distributed array data

```
int MPI_Type_create_darray(int size, int rank, int ndims,
        int array_of_gsizes[], int array_of_distribs[],
        int array_of_dargs[], int array_of_psizes[],
        int order, MPI_Datatype oldtype,
        MPI_Datatype *newtype)
```

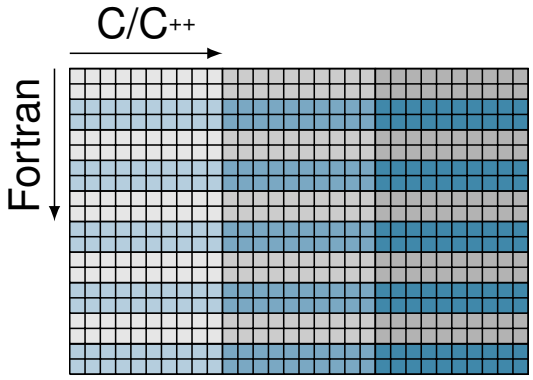
- N-dimensional distributed/strided sub-array of an N-dimensional array
- Fortran and C order allowed
- Fortran *and* C calls expect indices starting from 0



Distributed array data

```
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS,
        ARRAY_OF_GSIZES[], ARRAY_OF_DISTRIBS[],
        ARRAY_OF_DARGS[], ARRAY_OF_PSIZES[],
        ORDER, MPI_DATATYPE OLDTYPE,
        MPI_DATATYPE *NEWTYPE)
```

- N-dimensional distributed/strided sub-array of an N-dimensional array
- Fortran and C order allowed
- Fortran *and* C calls expect indices starting from 0



Finding the address of a memory location

C/C++

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

```
MPI_Aint addr_block_0, addr_block_i;

MPI_Get_address(&block_0, &addr_block_0);
MPI_Get_address(&block_i, &addr_block_i);

displacement_i = addr_block_i - addr_block_0;
```

Do not rely on C's address operator `&`, as ANSI C does not guarantee pointer values to be absolute addresses. Furthermore, address space may be segmented. Always use `MPI_GET_ADDRESS`, which also guarantees portability.

Finding the address of a memory location

Fortran

```
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type>  LOCATION(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
    INTEGER IERROR
```

```
MPI_Aint addr_block_0 = MPI_Aint addr_block_i = 0;

MPI_Get_address(&block_0, &addr_block_0);
MPI_Get_address(&block_i, &addr_block_i);

displacement_i = addr_block_i - addr_block_0;
```


Fixed memory layout of struct datatypes

C/C++

- C, struct

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

Fixed memory layout of struct datatypes

Fortran

- Fortran common block

```
INTEGER I(3)
DOUBLE PRECISION D(5)
COMMON /BCOMM/ I, D
```

- Fortran derived types

```
TYPE buff_type
  SEQUENCE
  INTEGER, DIMENSION(3)::i
  DOUBLE PRECISION, DIMENSION(5)::d
END TYPE buff_type
TYPE(buff_type)::buff_var
```


Alternative: Arbitrary memory layout

- Each array is allocated independently
- Each buffer is a pair of a 3-int and a 5-double array
- The size of the hole may be positive, zero or even negative
- For every buffer one needs a dedicated datatype handle, *e.g.*

inbuf_datatype

outbuf_datatype

Committing and freeing derived datatypes

C/C++

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

- Before it can be used in a communication or I/O call, each derived datatype has to be committed

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

- Mark a datatype for deallocation
- Datatype will be deallocated when all pending operations are finished

Committing and freeing derived datatypes

Fortran

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

- Before it can be used in a communication or I/O call, each derived datatype has to be committed

```
MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

- Mark a datatype for deallocation
- Datatype will be deallocated when all pending operations are finished

Size vs. extent of a datatype I

Size

- The size of a datatype is the net number of bytes to be transferred (without “holes”).

Extent

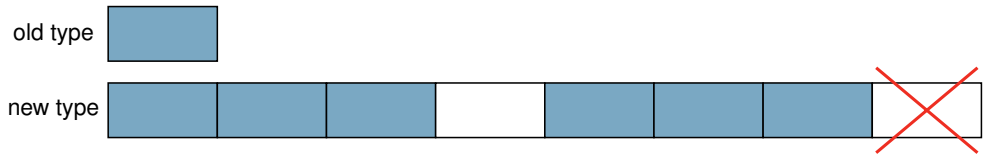
- The extent of a datatype is the span from the lower to the upper bound (including inner “holes”). When creating new types, holes at the end of the new type are not counted to the extent.

Size vs. extent of a datatype II

Basic datatypes

- size = extent = number of bytes used by the compiler

Derived datatypes (example)



- size = 6 × size of old type
- extent = 7 × extent of old type

Query size and extent of a datatype

C/C++

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

- Returns the total number of bytes of the entries in `DATATYPE`

```
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint* lb,
        MPI_Aint* extent)
```

- The extent is the number of bytes between the lower and the upper bound markers

Query size and extent of a datatype

Fortran

```
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

- Returns the total number of bytes of the entries in `DATATYPE`

```
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
  INTEGER DATATYPE, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
```

- The extent is the number of bytes between the lower and the upper bound markers

Resizing datatypes

C/C++

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

- Sets new lower and upper bound markers
- Allows for correct stride in creation of new derived datatypes
  - Holes at the end of datatypes do not initially count to the extent
  - Successive datatypes (e.g. contiguous, vector) would not be defined as intended

Resizing datatypes

Fortran

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
  INTEGER OLDTYPE, NEWTYPE, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
```

- Sets new lower and upper bound markers
- Allows for correct stride in creation of new derived datatypes
  - Holes at the end of datatypes do not initially count to the extent
  - Successive datatypes (e.g. contiguous, vector) would not be defined as intended

Summary

- MPI datatypes are defined by their type map
- Derived datatypes are created by defining a type map with existing datatypes
- Complex patterns can be defined
- Datatypes are *local* to a process
- Datatypes can have holes
- Size is the number of bytes of the entries of a datatype
- Extent is defined by the number of bytes between lower and upper bound

Parallel programming

Collective communication

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Learning objectives

At the end of this lesson, you will be able to

- use collective communication for communication patterns involving multiple processes
- combine computation and communication using reduction calls
- associate different communication calls with communication patterns

JÜLICH FORSCHUNGSZENTRUM

Collective communication

Communication involving a group of processes

Examples

- Barrier synchronization
- Broadcast, scatter, gather
- Global reductions (sum, maximum, logical operations, . . .)

Characteristics of collective operations I

- Collective operations are associated with a communicator
  - All processes within the communicator must participate in the collective communication
- Synchronization may or may not occur
  - Depends on algorithm and implementation used to communicate data
- Collective operations are blocking
  - Buffers can be accessed after call returns
  - Non-blocking API is currently under discussion in the MPI Forum and targeted for MPI 3.0

Characteristics of collective operations II

- No interference with point-to-point communication
  - Be careful when overlapping blocking point-to-point with collective communication
- No tags
  - Communications occur in the order they are issued by the user
- Receive buffers must have exactly the same size as the corresponding send buffers

Barrier synchronization

C/C++

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

- Explicit synchronization between processes
  - Remember: Time spent in `MPI_BARRIER` or any other kind of synchronization is always non-productive
  - Use only where global synchronization (over a communicator) is needed
  - Synchronization is implicitly done by communication routines
- A process cannot leave the function call before all participating processes have entered the function
- Global synchronization always includes inter-process communication
- If you use it for the synchronization of external communication (*e.g.* I/O), consider exchanging tokens
  - May be more efficient and scalable

Barrier synchronization

Fortran

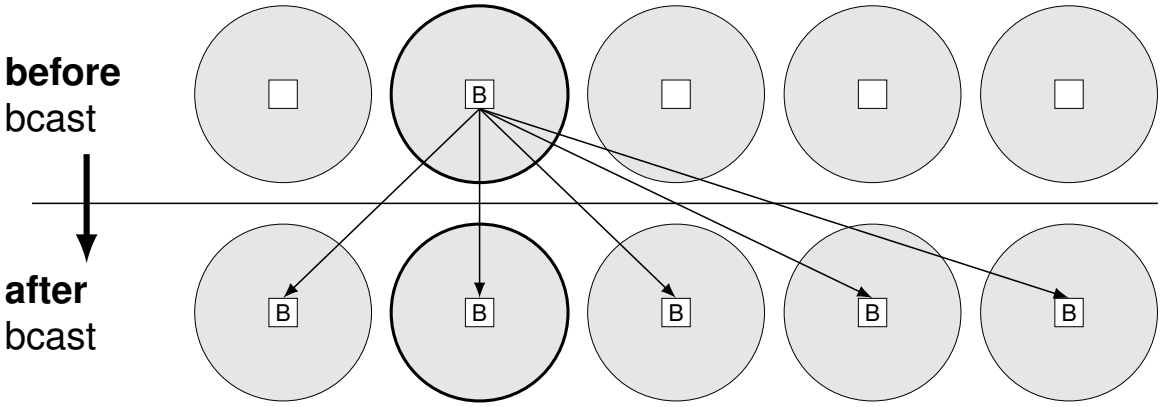
```
MPI_BARRIER(MPI_COMM COMM, IERROR)
    INTERGER COMM, IERROR
```

- Explicit synchronization between processes
  - Remember: Time spent in `MPI_BARRIER` or any other kind of synchronization is always non-productive
  - Use only where global synchronization (over a communicator) is needed
  - Synchronization is implicitly done by communication routines
- A process cannot leave the function call before all participating processes have entered the function
- Global synchronization always includes inter-process communication
- If you use it for the synchronization of external communication (*e.g.* I/O), consider exchanging tokens
  - May be more efficient and scalable

Broadcast

C/C++

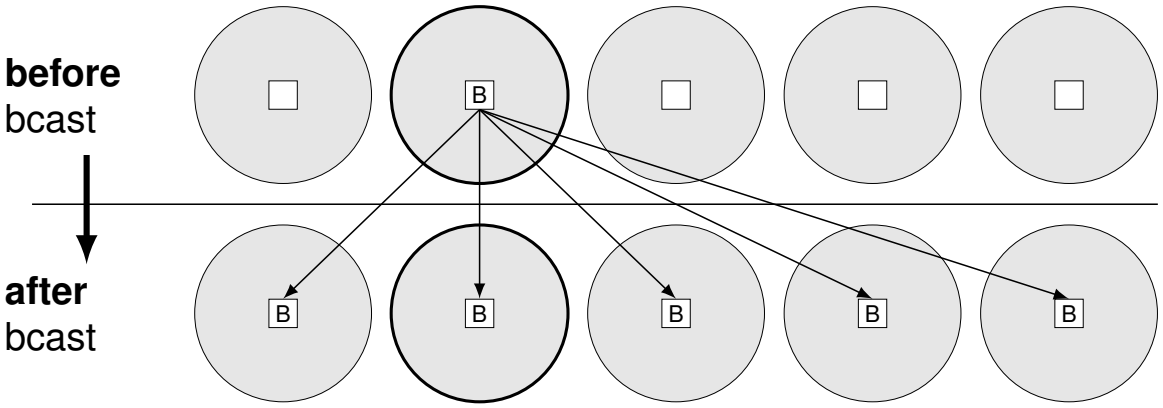
```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int root, MPI_Comm comm)
```



Broadcast

Fortran

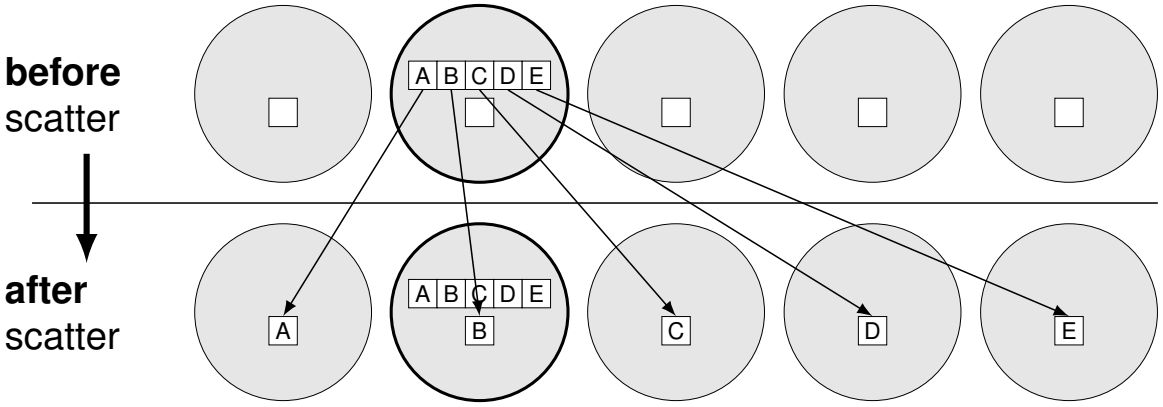
```
MPI_BCAST(BUF, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type>  BUF(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```



Scatter

C/C++

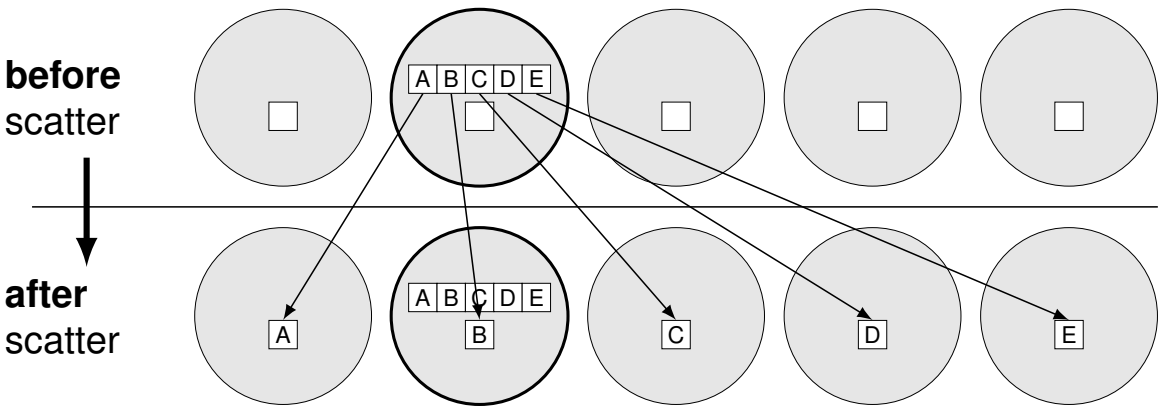
```
int MPI_Scatter(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```



Scatter

Fortran

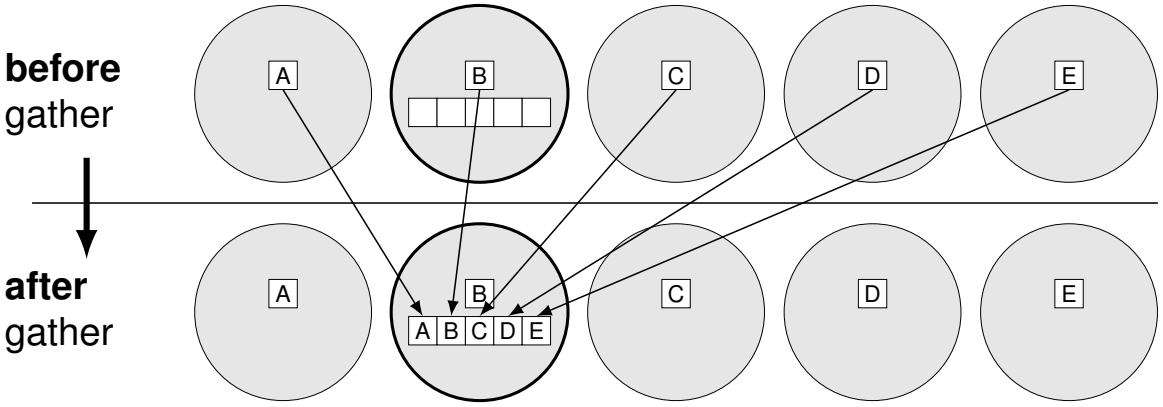
```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
        RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
    <type>  SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
            COMM, IERROR
```



Gather

C/C++

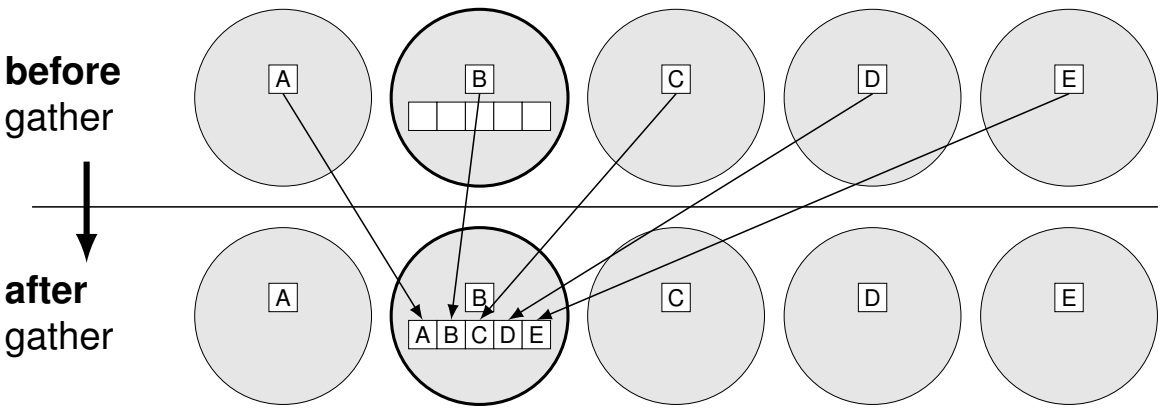
```
int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```



Gather

Fortran

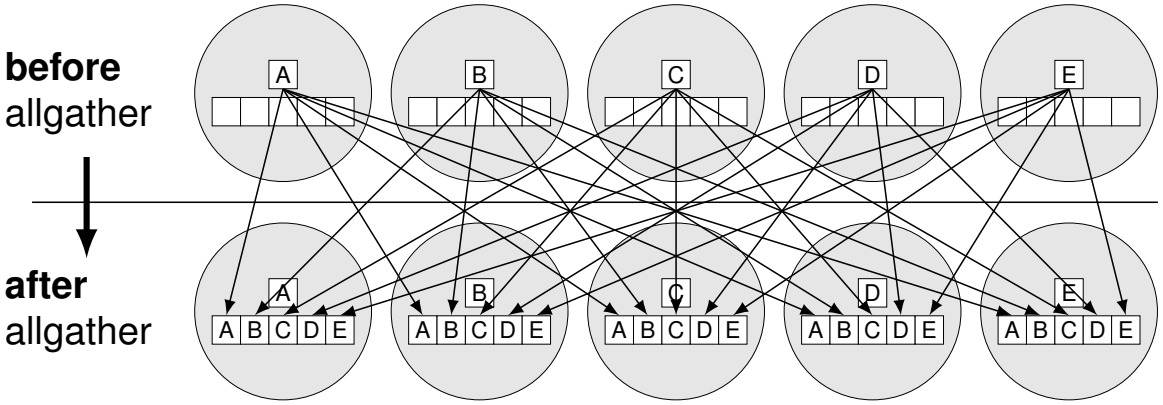
```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
         RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
        COMM, IERROR
```



Gather-to-all

C/C++

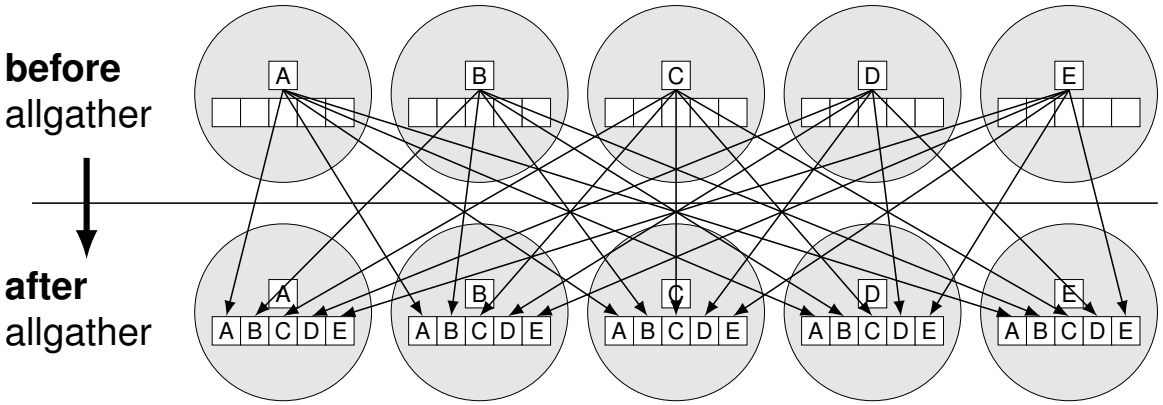
```
int MPI_Allgather(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype,
        MPI_Comm comm)
```



Gather-to-all

Fortran

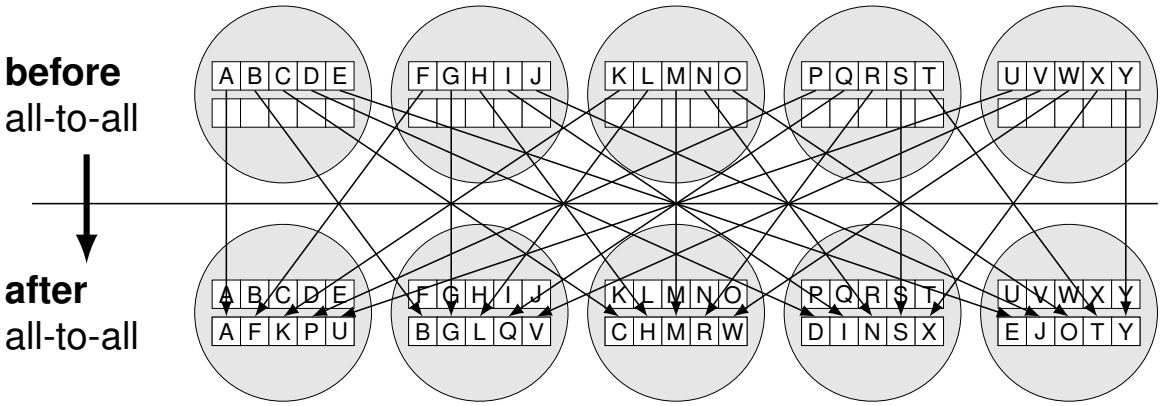
```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
        RECVCOUNT, RECVTYPE, COMM, IERROR)
    <type>  SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
            COMM, IERROR
```



All-to-all

C/C++

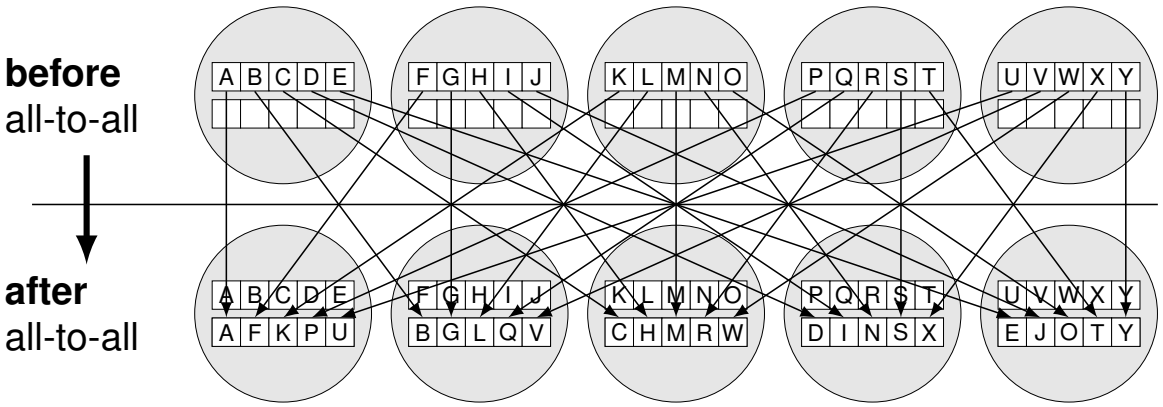
```
int MPI_All_to_all(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype,
        MPI_Comm comm)
```



All-to-all

Fortran

```
MPI_ALL_TO_ALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
        RECVCOUNT, RECVTYPE, COMM, IERROR)
    <type>  SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
            COMM, IERROR
```



Global reduction operators I

Perform a global reduction operation across all members of a group

- Assosiative operation over distributed data
  - $d_0 \circ d_1 \circ d_2 \circ \ldots \circ d_{n-1}$
  - d_i Data of process with rank i
  - $\circ$ Associative operation
- Examples
  - Global sum or product
  - Global maximum or minimum
  - Global user-defined operation
- Order in which sub-reductions are performed is not defined
  - Floating point rounding may depend on associativity
  - Behavior may be non-deterministic

Example: Global reduction

C/C++

```
root = 0;
MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM,
           root, MPI_COMM_WORLD);
```

- Global integer sum
- Sum of all `inbuf` values is to be returned in `resultbuf`
- The result is written to `resultbuf` of the root process only

Example: Global reduction

Fortran

```
root = 0
MPI_REDUCE(inbuf, resultbuf, 1, MPI_INTEGER, MPI_SUM,
           root, MPI_COMM_WORLD, ierror)
```

- Global integer sum
- Sum of all `inbuf` values is to be returned in `resultbuf`
- The result is written to `resultbuf` of the root process only

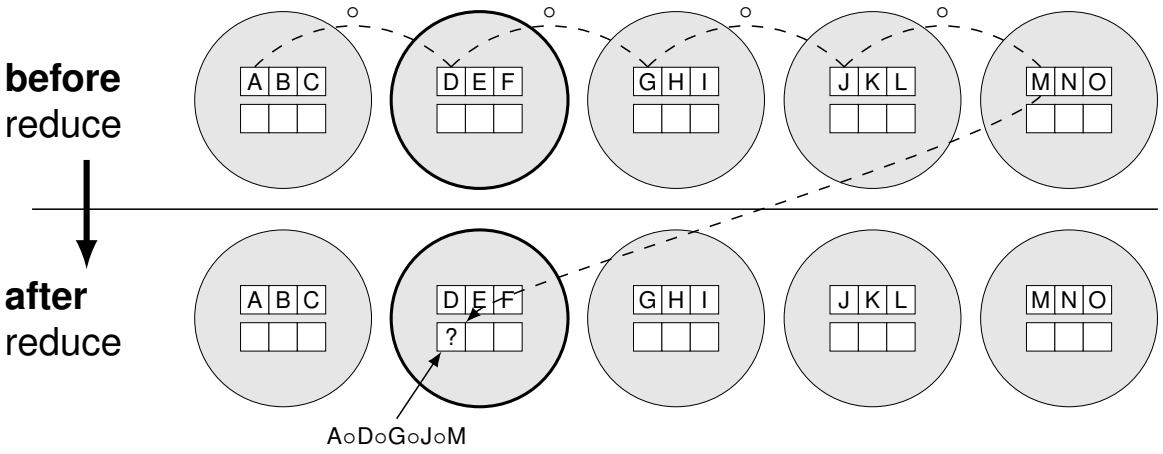
Predefined reduction operation handles

Operation handle	Function
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 its location
MPI_MINLOC	Minimum and its location

Reduce

C/C++

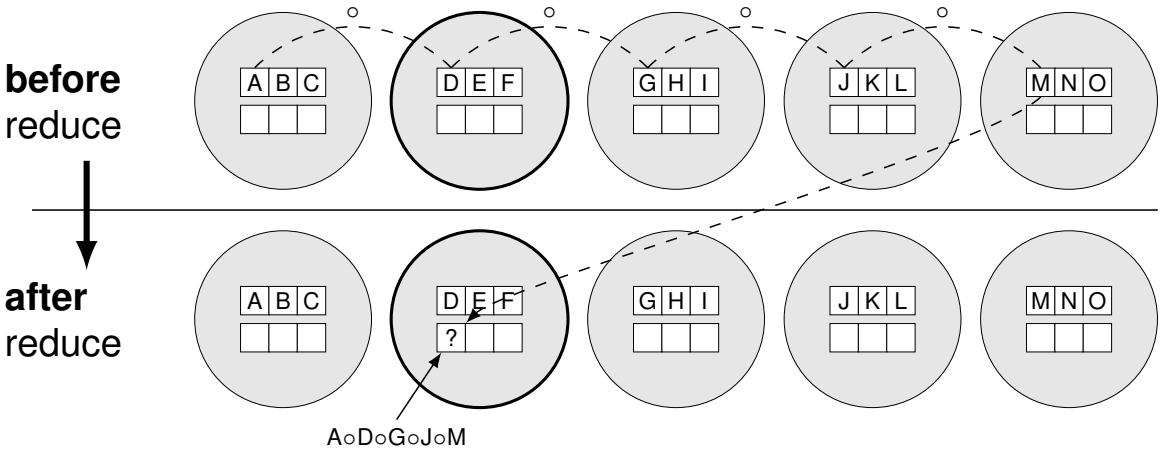
```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int root,
        MPI_Comm comm)
```



Reduce

Fortran

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT,
        COMM, IERROR)
    <type>  SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```



User-defined reduction operators

C/C++

- Operator handles
  - Predefined - see table on previous slide
  - User-defined
- User-defined operation $\square$
  - Associative
  - Must perform the operation $\vec{A} \square \vec{B}$
  - syntax $\Rightarrow$ MPI standard

```
int MPI_Op_create(MPI_User_function *func,
        int commute, MPI_Op *op)
```

- The `COMMUTE` flag tells MPI whether `FUNC` is commutative

User-defined reduction operators

Fortran

- Operator handles
  - Predefined - see table on previous slide
  - User-defined
- User-defined operation $\square$
  - Associative
  - Must perform the operation $\vec{A} \square \vec{B}$
  - syntax $\Rightarrow$ MPI standard


```
MPI_OP_CREATE(FUNC, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```


- The `COMMUTE` flag tells MPI whether `FUNC` is commutative

Variants of reduction operators

- `MPI_ALLREDUCE`
  - No root
  - All processes receive the result
- `MPI_REDUCE_SCATTER`
  - Similar to `MPI_ALLREDUCE`, but:
  - Processes can choose to receive certain-size segments of the result vector
- `MPI_[EX]SCAN`
  - ”Parallel prefix” operation
  - Result in `RECVBUF` of process with rank *i* is the reduction of the `SENDBUF`-values of ranks $0, \ldots, i$ (inclusive)
  - Either including or excluding the local value

Summary

- Collective communication provides interfaces to common communication patterns involving multiple processes
- It hides low-level communication algorithms behind high-level interfaces
- Different MPI implementations can optimize for a desired architecture
- Reduction functions provide the possibility of combining communication with computation
- Support for user-defined reduction operators
- All collective communication functions available in MPI 2.x are blocking

JÜLICH FORSCHUNGSZENTRUM

Parallel programming

One-sided communication

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Learning objectives

At the end of this lesson, you will be able to

- use one-sided communication as the third communication paradigm provided by MPI
- differentiate between active and passive target synchronization
- use active and passive target synchronization calls to enable fine-grained synchronization
- understand where the one-sided communication paradigm ease programming effort

Motivation

- Not all communication patterns can be efficiently solved with the existing MPI communication paradigms
- Collective and point-to-point communication partially need tight synchronization of processes
- Two-sided communication needs two explicit communication calls:
  - Sender has explicit function call to send(...)
  - Receiver has explicit function call to recv(...)
- Problem:
  - The receiver of the data cannot initiate the data transfer
- Solution:
  - A data transfer from the callee to the caller

Amenities of one-sided communication

- Receiver can initiate data transfer
  - Ease of programming where communication partner does not know it needs to participate in a data transfer in advance
- Synchronization is detached from data transfer
- Reduction operations available (only predefined)

Terms and definitions I

Target

The process providing access to its memory through a defined window. The target does not explicitly participate in the data transfer.

Origin

The process triggering the one-sided operation, specifying all needed parameters.

Terms and definitions II

Window

A defined block of memory opened for remote access through MPI RMA operations. Its definition is collective on all processes using this window. Only designated targets have to specify a valid buffer, origins can use a special placeholder to obtain a handle without opening memory for remote access.

Exposure epoch

An exposure epoch is the time interval some defined data access is allowed on a window.

Terms and definitions III

Access epoch

An access epoch is the time interval from the origin process' start signal of data access to its end signal of data access on a window.

Registering memory for MPI-RMA

C/C++

```
int MPI_Win_create(void* base, MPI_Aint* size, int disp_unit,
        MPI_Info info, MPI_Comm comm, MPI_Win* win)
```

- Needed to initialize a memory buffer for use with MPI RMA
- Can be any part in memory
  - Memory allocated `MPI_ALLOC_MEM` may perform better
  - Memory allocated at natural boundaries may perform better
- `DISP_UNIT` sets offset handling for window
  - 1 for no scaling (each byte is addressable)
  - `sizeof(type)` for array like indexing on non-byte arrays
- Window is declared on all processes of a communicator
  - Base address, size, displacement unit, and info argument may differ on the processes

Registering memory for MPI-RMA

Fortran

```
MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
  <type> BASE(*)
  INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
  INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
```

- Needed to initialize a memory buffer for use with MPI RMA
- Can be any part in memory
  - Memory allocated `MPI_ALLOC_MEM` may perform better
  - Memory allocated at natural boundaries may perform better
- `DISP_UNIT` sets offset handling for window
  - 1 for no scaling (each byte is addressable)
  - `sizeof(type)` for array like indexing on non-byte arrays
- Window is declared on all processes of a communicator
  - Base address, size, displacement unit, and info argument may differ on the processes

De-registering memory for MPI-RMA

C/C++

```
int MPI_Win_free(MPI_Win* win)
```

- Frees the handle (not the memory) and returns a null handle
- Can only be called after all RMA operations have been handled
  - RMA operations must be completed by a proper synchronization call
- Memory can be freed by the user after the call returns

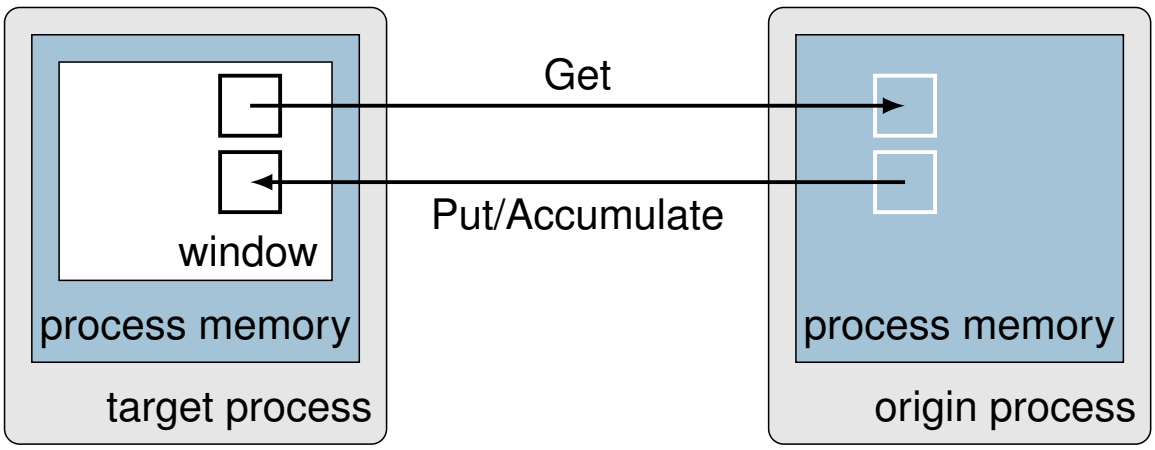
De-registering memory for MPI-RMA

Fortran

```
MPI_WIN_FREE(WIN, IERROR)
  INTEGER WIN, IERROR
```

- Frees the handle (not the memory) and returns a null handle
- Can only be called after all RMA operations have been handled
  - RMA operations must be completed by a proper synchronization call
- Memory can be freed by the user after the call returns

RMA operations in MPI-2



Put operation

C/C++

```
int MPI_Put(void* origin_addr, int origin_count,
        MPI_Datatype origin_type, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_type, MPI_Win win)
```

- Data transfer from origin to target
- No matching call on target side
- Communication parameters all on one side
  - Origin must know the correct index to the window on target process

Put operation

Fortran

```
MPI_PUT(ORIGIN_ADDRE, ORIGIN_COUNT, ORIGIN_TYPE, TARGET_RANK,
    TARGET_DISP, TARGET_COUNT, TARGET_TYPE, WIN, IERROR)
  <type> ORIGIN_ADDR(*)
  INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
  INTEGER ORIGIN_COUNT, ORIGIN_TYPE, TARGET_RANK,
          TARGET_COUNT, TARGET_TYPE, WIN, IERROR
```

- Data transfer from origin to target
- No matching call on target side
- Communication parameters all on one side
  - Origin must know the correct index to the window on target process

Accumulate operation

C/C++

```
int MPI_Accumulate(void* origin_addr, int origin_count,
        MPI_Datatype origin_type, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_type, MPI_Op op, MPI_Win win)
```

- Like PUT operation
- Buffer elements on target side are combined with operation `OP`
  - `OP` can only be a predefined reduction operator
  - Only with predefined datatypes
  - `MPI_Put` is actually an accumulate with the `MPI_REPLACE` operator

Accumulate operation

Fortran

```
MPI_ACCUMULATE(ORIGIN_ADDRE, ORIGIN_COUNT, ORIGIN_TYPE,
    TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_TYPE,
    OP, WIN, IERROR)
  <type> ORIGIN_ADDR(*)
  INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
  INTEGER ORIGIN_COUNT, ORIGIN_TYPE, TARGET_RANK,
          TARGET_COUNT, TARGET_TYPE, OP, WIN, IERROR
```

- Like PUT operation
- Buffer elements on target side are combined with operation `OP`
  - `OP` can only be a predefined reduction operator
  - Only with predefined datatypes
  - `MPI_Put` is actually an accumulate with the `MPI_REPLACE` operator

Course GSTU2009 One-sided communication Slide 144

Get operation

C/C++

```
int MPI_Get(void* origin_addr, int origin_count,
        MPI_Datatype origin_type, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_type, MPI_Win win)
```

- Data transfer from target to origin
- No matching call on target side
- Communication parameters all on one side
  - Origin must know the correct index to the window on target process

Get operation

Fortran

```
MPI_GET(ORIGIN_ADDRE, ORIGIN_COUNT, ORIGIN_TYPE, TARGET_RANK,
    TARGET_DISP, TARGET_COUNT, TARGET_TYPE, WIN, IERROR)
  <type> ORIGIN_ADDR(*)
  INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
  INTEGER ORIGIN_COUNT, ORIGIN_TYPE, TARGET_RANK,
          TARGET_COUNT, TARGET_TYPE, WIN, IERROR
```

- Data transfer from target to origin
- No matching call on target side
- Communication parameters all on one side
  - Origin must know the correct index to the window on target process

Synchronization schemes for MPI-2 RMA operations

Active target synchronization

Origin *and* target participate equally in synchronizing the RMA operations.

- Collective synchronization with fence
- General active target synchronization (GATS)

Passive target synchronization

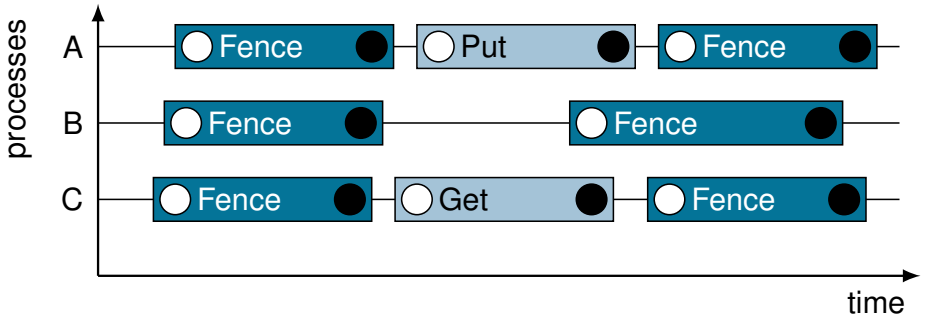
Target process is not explicitly taking part in the synchronization of the accessing RMA operation.

- Synchronization with locks

Synchronization with fence

- Collective call on communicator used for window creation
- Contains an implicit barrier
- Data access has to occur between two fence calls
- Written and read data is only accessible after completing fence
- Local and remote accesses must not occur between the same fence calls
- Access and exposure epoch matching is done automatically

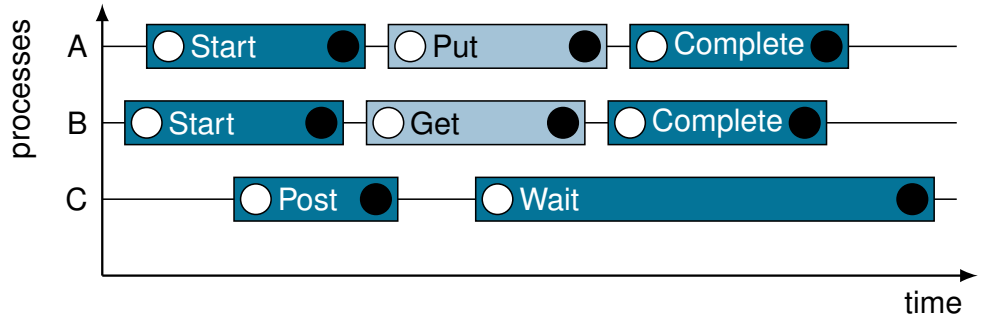
Example: Synchronization with fence



General active target synchronization

- Pairwise synchronization of processes on subgroups of communicator used for window definition
- Individual calls for access and exposure epochs
  - `MPI_Win_start`/`MPI_Win_complete` for access epoch
  - `MPI_Win_post`/`MPI_Win_wait` for exposure epoch
- Accesses to local data only after epoch is closed
  - Data read from remote processes (access epoch) is accessible after `MPI_Win_complete`
  - Data written by remote processes (exposure epoch) is accessible after `MPI_Win_wait`
- Mind the order of calls with process-local access *and* exposure epochs
  - Exposure epoch must contain access epoch

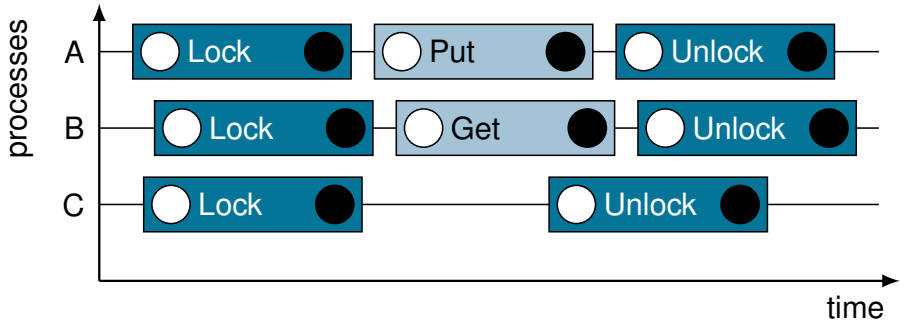
Example: General active target synchronization



Passive target synchronization

- Explicit synchronization and RMA operations *only* on the origin process
- Local *and* remote accesses need to be embraced by calls to `MPI_Win_lock` and `MPI_Win_unlock`
  - Needed to ensure serial consistency of memory updates
- Shared and exclusive locks available
  - `MPI_LOCK_SHARED` and `MPI_LOCK_EXCLUSIVE`
- Order of accesses is not guaranteed and has to be handled otherwise
  - Be aware of possible race-conditions in your code
- Lock and any number of following RMA operations are allowed to be non-blocking
  - Operations may be scheduled and executed within the `MPI_Win_Unlock`

Example: Synchronization with locks



Assertions on MPI one-sided calls

- Assertions may help optimize the synchronization process
- The implementation may ignore the assertions
- User-provided assertions *must* be correct
- Assertion code 0 (no assertions) is always valid.
- Assertion code is a binary-or'ed integer
  - Full list of assertions available in MPI 2.1 standard pp. 343

Pitfalls with one-sided communication

- Exposed memory needs to reside in a common block in Fortran
- Order of RMA operations needs to be managed by the user in passive mode
- Memory exposed for passive target synchronization needs to be allocated with `MPI_Alloc_mem`
  - Not portable in Fortran

Summary

- One-sided communication is the third major communication paradigm available in MPI
- All communication parameters are defined by the origin process
- RMA operations comprise Put, Accumulate and Get
  - Accumulate only works with predefined operators on predefined datatypes
- Synchronization is separated from the data movement
- Three synchronization modes are available
  - Fence (collective on the window communicator)
  - GATS (pair-wise synchronization on subgroups)
  - Lock/Unlock (passive target synchronization)

JÜLICH FORSCHUNGSZENTRUM

Parallel programming

Virtual Topologies

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Learning Objectives

After this lesson, you will be able to

- List available virtual topology types in MPI
- Create multi-dimensional Cartesian communicators
- Create sub-communicators of existing Cartesian communicators
- Use MPI to determine optimal Cartesian decomposition
- Query neighbors on Cartesian communicators

Motivation

- Problem domain is often multi-dimensional
- Optimal mapping to one dimension often complex and not portable
  - Network topology may largely influences application performance if not adhered properly
- Neighbor determination may be cumbersome
- Naming in multi-dimensional space is often more intuitive

Virtual Topology Use Case

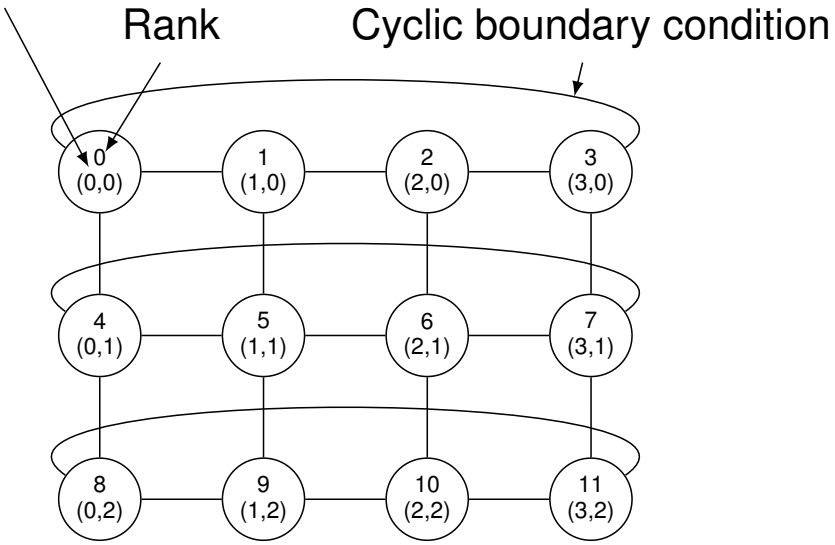
- Array $A(1:3000, 1:4000, 1:500) = 6 \times 10^9$ words distributed on $3 \times 4 \times 5 = 60$ processors with coordinates $(0..2, 0..3, 0..4)$
- Processor $(2, 0, 3)$, *i.e.* rank 43 gets *e.g.* $A(2001:3000, 1:1000, 301:400) = 0.1 \times 10^9$ words
- Process coordinates: Handled by virtual Cartesian topology
- Array decomposition: Handled by the application, *i.e.* the user

Virtual topologies

- Convenient process naming mechanism
- Allow to name the processes in a communicator in a way that fits the communication pattern better
- Make subsequent code simpler
- May provide hints to the run-time system that allow for optimization of the communication
- Creating a virtual topology produces a new communicator
- MPI provides mapping functions
  - To compute ranks from virtual coordinates
  - And vice versa

Example: 2-dimensional cylinder

Cartesian process coordinate



Topologies

- Cartesian topology
  - Each process is connected to its neighbor process in a virtual grid
  - Boundaries can be cyclic, or not (torus vs. grid)
  - Processes are identified by their Cartesian coordinates
  - Any process within the communicator can of course still communicate with any other
- Graph topologies (not covered in this course)
  - Generalized graphs
  - A process can have an arbitrary number of neighbors
  - Edges can be directed (asymmetric adjacency matrix)
  - Nodes can the same neighbor process multiple times

Creating a Cartesian virtual topology

C/C++

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
        int *dims, int *periods, int reorder,
        MPI_Comm *comm_cart)
```

- Example

```
comm_old = MPI_COMM_WORLD
   ndims = 2
    dims = (4, 3)
 periods = (1true, 0false)
 reorder = see next slide
```

Creating a Cartesian virtual topology

Fortran

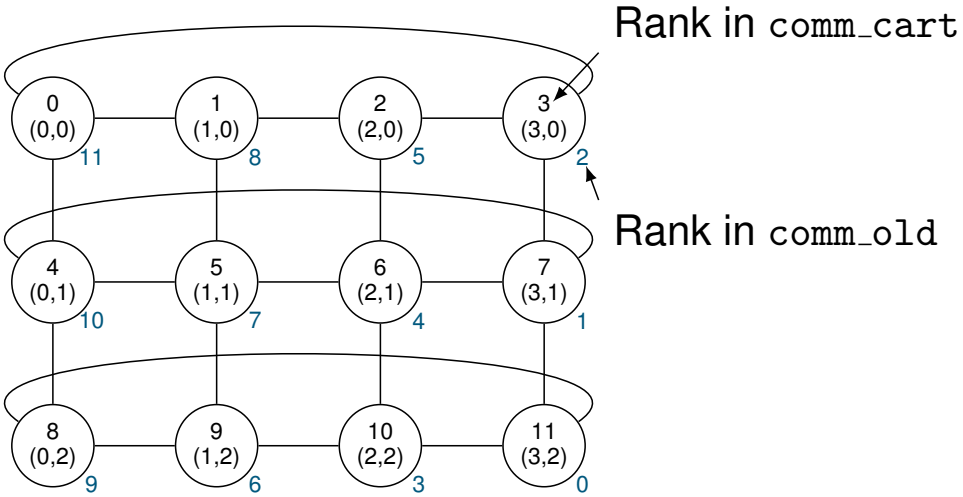
```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS,
        REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

- Example

```
comm_old = MPI_COMM_WORLD
   ndims = 2
    dims = (4, 3)
 periods = (1true, 0false)
 reorder = see next slide
```


Example: 2-dimensional cylinder

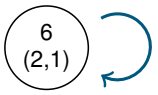
- If `reorder == true`, Process ranks in `comm_old` and `comm_cart` may differ



Mapping functions

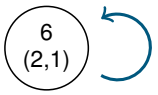
C/C++

- Mapping ranks to process grid coordinates




```
int MPI_Cart_coords(MPI_Comm comm_cart, int rank, int maxdims,
        int *coords)
```

- Mapping process grid coordinates to ranks

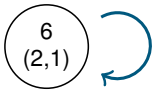



```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
```


Mapping functions

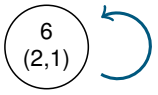
Fortran

- Mapping ranks to process grid coordinates



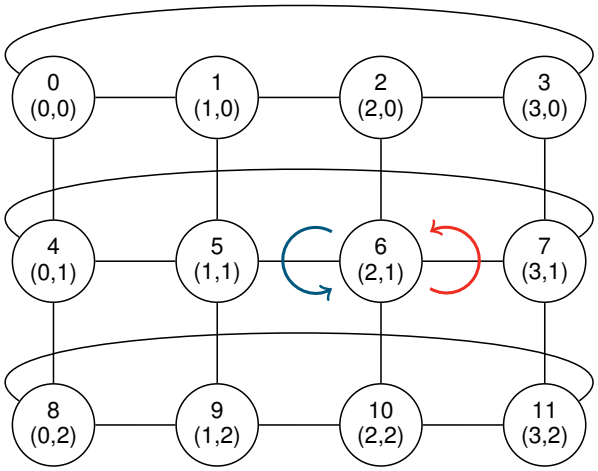

```
MPI_CART_COORDS(COMM_CART, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM_CART, RANK, MAXDIMS, COORDS(*), IERROR
```

- Mapping process grid coordinates to ranks




```
MPI_CART_RANK(COMM_CART, COORDS, RANK, IERROR)
    INTEGER COMM_CART, COORDS(*), RANK, IERROR
```

Local process coordinates and rank



- Each process obtains its own local values using
  - `MPI_Comm_rank(comm_cart, &myrank)`
  - `MPI_Cart_coords(comm_cart, myrank, maxdims, &mycoords)`

Computing the ranks of neighbor processes

C/C++

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

- Returns `MPI_PROC_NULL` if there is no neighbor
- `MPI_PROC_NULL` can be used as source or destination rank in each communication
  - This communication will then be a no-op!

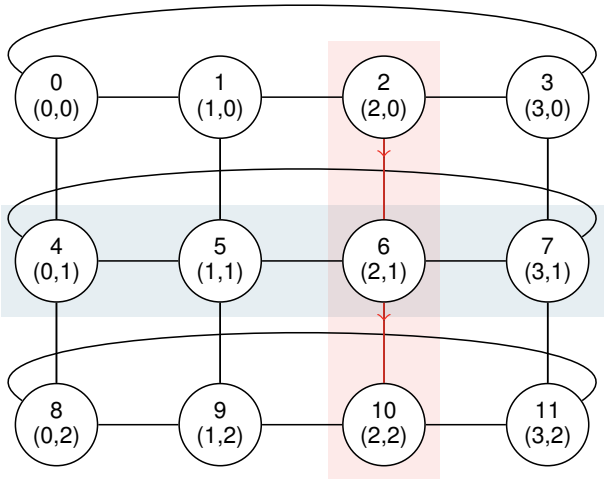
Computing the ranks of neighbor processes

Fortran

```
MPI_CART_SHIFT(COMM_CART, DIRECTION, DISP, RANK_SOURCE,
        RANK_DEST, IERROR)
    INTEGER COMM_CART, DIRECTION, DISP, RANK_SOURCE,
        RANK_DEST, IERROR
```

- Returns `MPI_PROC_NULL` if there is no neighbor
- `MPI_PROC_NULL` can be used as source or destination rank in each communication
  - This communication will then be a no-op!

Example: Computing a neighbor rank




```
MPI_Cart_shift(comm_cart,dir,displacement,&source,&dest);
```

Example for myrank=6: 0 or +1 5 7

1 +1 2 10

Cartesian partitioning

C/C++

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *remain_dims,
        MPI_Comm *comm_slice)
```

- Cut a grid into sub-grids (“slices”)
- A new communicator is created for each slice

JÜLICH FORSCHUNGSZENTRUM

Cartesian partitioning

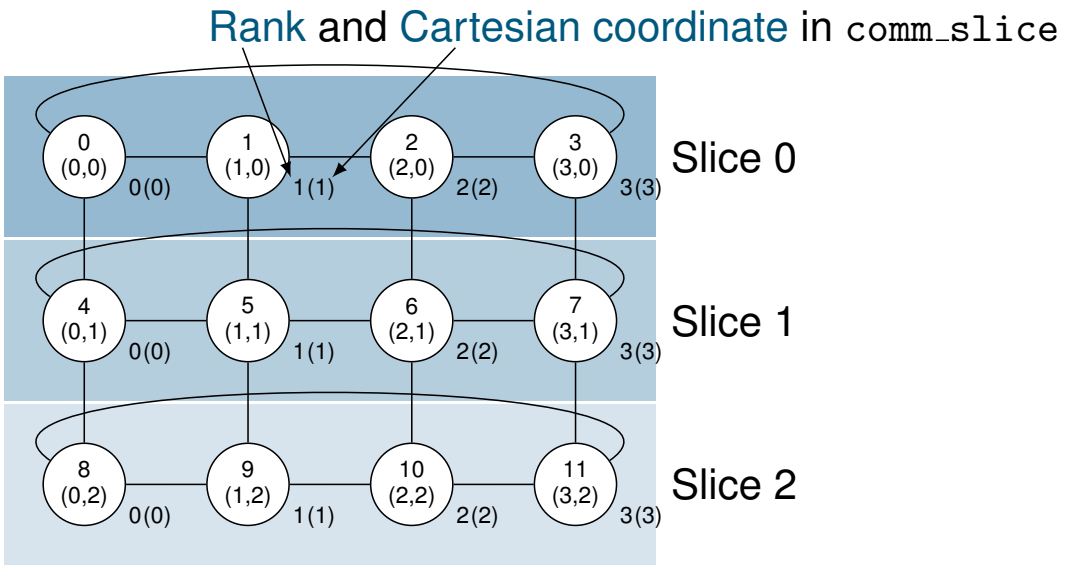
Fortran

```
MPI_CART_SUB(COMM_CART, REMAIN_DIMS, COMM_SLICE, IERROR)
    INTEGER COMM_CART
    LOGICAL REMAIN_DIMS(*)
    INTEGER COMM_SLICE, IERROR
```

- Cut a grid into sub-grids (“slices”)
- A new communicator is created for each slice

Course GSTU2009 Virtual Topologies Slide 169

Example: Cartesian sub-communicators




```
int remain_dims[2] = {1,0};
MPI_Cart_sub(comm_cart, remain_dims, &comm_slice);
```


Summary

- MPI provides virtual topologies to enable the user to describe the application topology
- MPI libraries may optimize process placement for the virtual topology
- Query functions ease handling of neighbor communication
- Multi-dimensional Cartesian grids and generalized graphs can be created
- Cartesian topologies can be sliced into sub-communicators

JÜLICH FORSCHUNGSZENTRUM

Parallel programming

Basic parallel I/O

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Learning Objectives

At the end of this lesson, you will be able to

- Motivate the need for parallel I/O
- Discuss the advantages of MPI file I/O
- Explain the common terms of MPI I/O
- Create, open and close files with parallel access
- Read from and write to a file
- Move the file pointer to an arbitrary position in the file

Motivation for parallel I/O

- Current scientific applications work on larger datasets that need to be read from and written to persistent storage.
- With CPU and interconnect speed increasing faster than persistent I/O speed, I/O might be the bottleneck of your application in the future.
- Workload is distributed to processes, while a lot of applications still serialize the I/O
- Process local files have serious scalability issues
  - Too many files
  - Manipulation of file system meta-data may be serialized
- Efficient parallel I/O to a minimal set of files is needed

Amenities of MPI-I/O I

- Portability
  - Standardized in 1997 and widespread support among vendors.
  - Open Source implementation ROMIO is publicly available.
- Ease of use
  - It blends into syntax and semantic scheme of point-to-point and collective communication of MPI.
  - Writing to a file is like sending data to another process.
- Efficiency
  - MPI implementers can transparently choose the best performing implementation for a specific platform.

Amenities of MPI-I/O II

- High level Interface
  - It provides coordinated and structured access to a file for multiple processes.
  - Distributed I/O to the same file through collective operations.
- Handling of heterogeneous environments
  - Automatic data conversion in heterogeneous systems.
  - File interoperability between systems via external representation.

MPI-I/O requirements

- Understanding of collective communication
  - A file handle works like a communicator for file I/O.
  - Coordination of file accesses can be of collective nature.
- Handling of Immediate Operations
  - Non-blocking calls may overlap computation and I/O.
- Derived Datatypes
  - Non-contiguous file access is defined with MPI's derived datatypes.

Terms and Definitions I

File

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

Displacement

Displacement is an absolute byte position relative to the beginning of a file.

Offset

Offset is a position in the file relative to the current view. It is expressed as a count of elementary types.

Terms and Definitions II

Elementary type

The elementary type is the basic entity of a file. It must be the same on all processes with the same file handle.

File type

The file type describes the access pattern of the processes on the file. It defines what parts of the file are accessible by a specific process. The processes may have different file types to access different parts of a file.

Terms and Definitions III

File view

A view defines the file data visible to a process. Each process has an individual view of a file defined by a displacement, an elementary type and a file type. The pattern is the same that `MPI_TYPE_CONTIGUOUS` would produced if it were passed the file type.

File pointer

A file pointer is an explicit offset maintained by MPI.

Opening a file

C/C++

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

- Filename's namespace is implementation dependent
- Call is collective on `comm`
- Process-local files can be opened with `MPI_COMM_SELF`
- Filename must reference the same file on all processes
- Additional information can be passed to MPI environment via the `MPI_Info` handle.

Opening a file

Fortran

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    CHARACTER*(*) FILENAME
    INTERGER COMM, AMODE, INFO, FH, IERROR
```

- Filename's namespace is implementation dependent
- Call is collective on `comm`
- Process-local files can be opened with `MPI_COMM_SELF`
- Filename must reference the same file on all processes
- Additional information can be passed to MPI environment via the `MPI_Info` handle.

Access modes

- Access mode is a bit-vector, which is modified with
  - | (Bitwise OR) in C
  - `IOR` (IOR Operator) in FORTRAN 90
  - + (Addition Operator) in FORTRAN 77
- One and only one of the following modes is mandatory:
  - `MPI_MODE_RDONLY` – read only
  - `MPI_MODE_RDWR` – read and write access
  - `MPI_MODE_WRONLY` – write only
- The following modes are optional:
  - `MPI_MODE_CREATE` – create file if it doesn't 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 can not be opened elsewhere
  - `MPI_MODE_SEQUENTIAL` – sequential file access (e.g tapes)
  - `MPI_MODE_APPEND` – all file pointers are set to end of file

Reserved info keys for MPI-I/O I

- Argument of `MPI_FILE_OPEN`, `MPI_FILE_DELETE`, `MPI_FILE_SET_VIEW`, `MPI_FILE_SET_INFO`
- Examples of reserved key values:
  - Collective buffering
    - `collective_buffering` (boolean): specifies whether the application may benefit from collective buffering
    - `cb_block_size` (integer): data access in chunks of this size
    - `cb_buffer_size` (integer): on each node; usually a multiple of block size
    - `cb_nodes` (integer): number of nodes used for collective buffering
  - Disk striping (only relevant in `MPI_FILE_OPEN`)
    - `striping_factor` (integer): number of I/O devices used for striping
    - `striping_unit` (integer): amount of consecutive data assigned to one I/O device before progressing to the next device

Reserved info keys for MPI-I/O II

- `MPI_INFO_NULL` may be passed

Closing a file

C/C++

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

- Collective operation
- If `MPI_MODE_DELETE_ON_CLOSE` was specified on opening, the file is deleted after closing

Closing a file

Fortran

```
MPI_FILE_CLOSE(FH, IERROR)
    INTERGER FH, IERROR
```

- Collective operation
- If `MPI_MODE_DELETE_ON_CLOSE` was specified on opening, the file is deleted after closing

Deleting a file

C/C++

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

- May be used to delete a file that is not currently opened
- Call is not collective, if called by multiple processes on the same file, all but one will return an error code $\neq$ MPI_SUCCESS
- A file is deleted automatically by MPI_FILE_CLOSE if MPI_DELETE_ON_CLOSE was specified in amode parameter of MPI_FILE_OPEN

Deleting a file

Fortran

```
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER INFO, IERROR
```

- May be used to delete a file that is not currently opened
- Call is not collective, if called by multiple processes on the same file, all but one will return an error code $\neq$ MPI_SUCCESS
- A file is deleted automatically by MPI_FILE_CLOSE if MPI_DELETE_ON_CLOSE was specified in amode parameter of MPI_FILE_OPEN

Writing to a file

C/C++

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

- Writes `COUNT` elements of `DATATYPE` from memory starting at `BUF` to the file
- Starts writing at the current position of the file pointer
- `STATUS` will indicate how many bytes have been written

Writing to a file

Fortran

```
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS,
    IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE),
      IERROR
```

- Writes `COUNT` elements of `DATATYPE` from memory starting at `BUF` to the file
- Starts writing at the current position of the file pointer
- `STATUS` will indicate how many bytes have been written

Reading from a file

C/C++

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

- Reads COUNT elements of DATATYPE from the file to memory starting at BUF
- Starts reading at the current position of the file pointer
- STATUS will indicate how many bytes have been read

Reading from a file

Fortran

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS,
    IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE),
      IERROR
```

- Reads COUNT elements of DATATYPE from the file to memory starting at BUF
- Starts reading at the current position of the file pointer
- STATUS will indicate how many bytes have been read

Seeking to a file position

C/C++

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

- Updates the individual file pointer according to WHENCE, which can have the following values:
  - MPI_SEEK_SET: pointer is set to OFFSET
  - MPI_SEEK_CUR: pointer is set to the current position plus OFFSET
  - MPI_SEEK_END: pointer is set to the end of file plus OFFSET
- OFFSET can be negative, which allows seeking backwards
- It is erroneous to seek to a negative position in the view

Seeking to a file position

Fortran

```
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

- Updates the individual file pointer according to WHENCE, which can have the following values:
  - MPI_SEEK_SET: pointer is set to OFFSET
  - MPI_SEEK_CUR: pointer is set to the current position plus OFFSET
  - MPI_SEEK_END: pointer is set to the end of file plus OFFSET
- OFFSET can be negative, which allows seeking backwards
- It is erroneous to seek to a negative position in the view

Querying the position of the file pointer

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

- Returns the current position of the individual file pointer in `OFFSET`
- The value can be used to return to this position or calculate a displacement
  - Do not forget to convert from offset to byte displacement if needed

Querying the position of the file pointer

```
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
  INTEGER FH, IERROR
  INTEGER(KIND=MPI_OFFSET_KIND)
```

- Returns the current position of the individual file pointer in `OFFSET`
- The value can be used to return to this position or calculate a displacement
  - Do not forget to convert from offset to byte displacement if needed

Error handling

- Default error handler is `MPI_ERRORS_RETURN`
  - Return values can be evaluated by the user application
- I/O errors are usually less catastrophic than communication errors
- Error handlers are set on a per-file-handle basis
- Default error handler is set by using `MPI_FILE_NULL` as the file handle

Summary

- MPI offers an easy to use interface to parallel I/O
- Opening and closing a file is collective
- Interface blends into the general look-and-feel
- MPI datatypes are used to read from and write to files
- MPI I/O sets `MPI_ERRORS_RETURN` as default handler

JÜLICH FORSCHUNGSZENTRUM

Parallel programming

File pointers and views

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Learning Objectives

At the end of this lesson, you will be able to

- Use individual and shared file pointers
- Create a special view for a file handle
- Use consecutive views for multipart files
- Query the view set on a file handle

File pointers

- Individual file pointers
  - Each process has its own file pointer that is only altered on accesses of that specific process
- Shared file pointers
  - This file pointer is shared among all processes in the communicator used to open the file
  - It is modified by any shared file pointer access of any process
  - Shared file pointers can only be used if file type gives each process access to the whole file!
- Explicit offset
  - No file pointer is used or modified
  - An explicit offset is given to determine access position
  - This can not be used with `MPI_MODE_SEQUENTIAL`!

Writing to a file using the shared file pointer

C/C++

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

- Blocking, individual write using the shared file pointer
- Only the shared file pointer will be advanced accordingly
- `DATATYPE` is used as the access pattern to `BUF`
- Middleware will serialize accesses to the shared file pointer to ensure collision-free file access

Writing to a file using the shared file pointer

Fortran

```
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

- Blocking, individual write using the shared file pointer
- Only the shared file pointer will be advanced accordingly
- `DATATYPE` is used as the access pattern to `BUF`
- Middleware will serialize accesses to the shared file pointer to ensure collision-free file access

Reading from a file using the shared file pointer

C/C++

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

- Blocking, individual read using the shared file pointer
- Only the shared file pointer will be advanced accordingly
- `DATATYPE` is used as the access pattern to `BUF`
- Middleware will serialize accesses to the shared file pointer to ensure collision-free file access

Reading from a file using the shared file pointer

Fortran

```
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

- Blocking, individual read using the shared file pointer
- Only the shared file pointer will be advanced accordingly
- `DATATYPE` is used as the access pattern to `BUF`
- Middleware will serialize accesses to the shared file pointer to ensure collision-free file access

Seeking with the shared file pointer

C/C++

```
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset,
        int whence)
```

- Updates the individual file pointer according to `WHENCE`, which can have the following values:
  - `MPI_SEEK_SET`: pointer is set to `OFFSET`
  - `MPI_SEEK_CUR`: pointer is set to the current position plus `OFFSET`
  - `MPI_SEEK_END`: pointer is set to the end of file plus `OFFSET`
- `OFFSET` can be negative, which allows seeking backwards
- It is erroneous to seek to a negative position in the view
- The call is collective
  - *all* processes with the file handle have to participate

Seeking with the shared file pointer

Fortran

```
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

- Updates the individual file pointer according to `WHENCE`, which can have the following values:
  - `MPI_SEEK_SET`: pointer is set to `OFFSET`
  - `MPI_SEEK_CUR`: pointer is set to the current position plus `OFFSET`
  - `MPI_SEEK_END`: pointer is set to the end of file plus `OFFSET`
- `OFFSET` can be negative, which allows seeking backwards
- It is erroneous to seek to a negative position in the view
- The call is collective
  - *all* processes with the file handle have to participate

Querying the shared file pointer position

C/C++

```
int MPI_File_get_position_shared(MPI_File fh,
        MPI_Offset* offset)
```

- Returns the current position of the individual file pointer in `OFFSET`
- The value can be used to return to this position or calculate a displacement
  - Do not forget to convert from offset to byte displacement if needed
- Call is *not* collective

Querying the shared file pointer position

Fortran

```
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
  INTEGER FH, IERROR
  INTEGER(KIND=MPI_OFFSET_KIND)
```

- Returns the current position of the individual file pointer in `OFFSET`
- The value can be used to return to this position or calculate a displacement
  - Do not forget to convert from offset to byte displacement if needed
- Call is *not* collective

Writing to an explicit offset in a file

C/C++

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

- Writes `COUNT` elements of `DATATYPE` from memory `BUF` to the file
- Starts writing at OFFSET units of etype from begin of view
- The sequence of basic datatypes of `DATATYPE` (= signature of `DATATYPE`) must match contiguous copies of the etype of the current view

Writing to an explicit offset in a file

Fortran

```
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE,
    STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE),
          IERROR
  INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

- Writes COUNT elements of DATATYPE from memory BUF to the file
- Starts writing at OFFSET units of etype from begin of view
- The sequence of basic datatypes of DATATYPE (= signature of DATATYPE) must match contiguous copies of the etype of the current view

Reading at an explicit offset in a file

C/C++

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

- Reads COUNT elements of DATATYPE from the file into memory
- DATATYPE defines where the data is placed in memory
- EOF is reaches when elements read $\neq$ COUNT
- The sequence of basic datatypes of DATATYPE (= signature of DATATYPE) must match contiguous copies of the etype of the current view

Reading at an explicit offset in a file

Fortran

```
MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS,
    IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
  INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

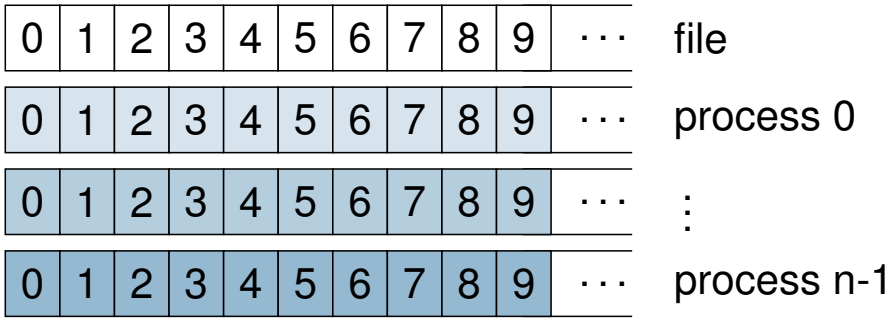
- Reads `COUNT` elements of `DATATYPE` from the file into memory
- `DATATYPE` defines where the data is placed in memory
- `EOF` is reaches when elements read $\neq$ `COUNT`
- The sequence of basic datatypes of `DATATYPE` (= signature of `DATATYPE`) must match contiguous copies of the etype of the current view

File views

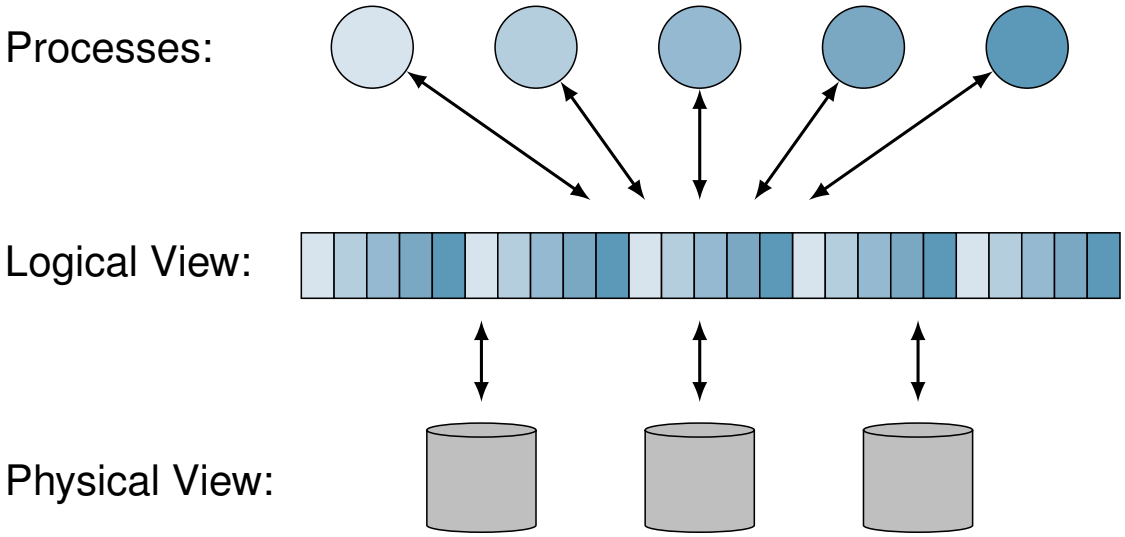
- Each process has a view connected to each handle
- The view determines the process' access to regions of a file
- A view is defined by a displacement, and elementary type and a file type
- The user can set a new elementary and file type via `MPI_File_set_view`
- Elementary type can be any valid MPI datatype
- File type must be a collection of the elementary typed items

Default view

- A default view for each participating process is defined implicitly while opening the file
  - No displacement
  - The file has no specific structure
    - The elementary type is `MPI_BYTE`
  - All processes have access to the complete file
    - The file type is `MPI_BYTE`

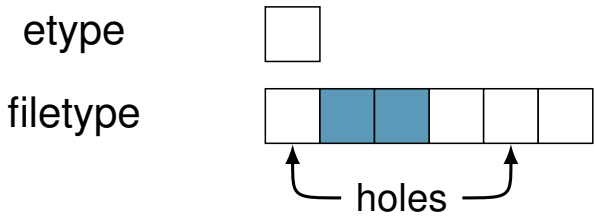


Different views on a file

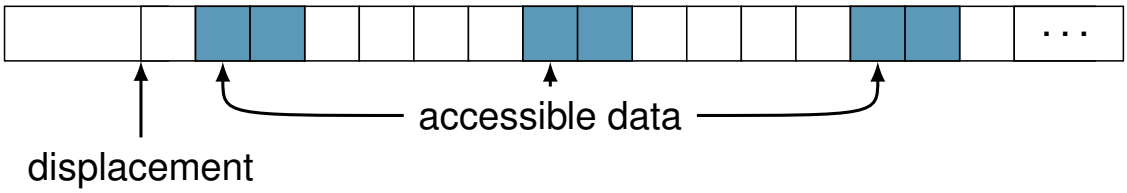


File layout

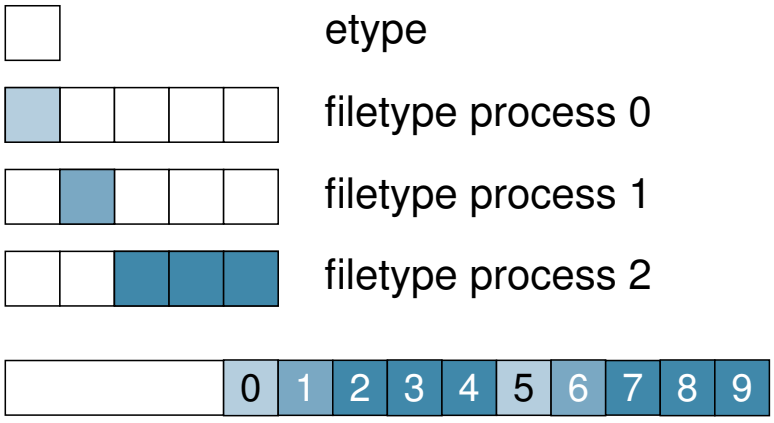
The filetype has to be constructed from one or multiple elementary types



The layout then repeats the filetype like `MPI_TYPE_CONTIGUOUS` would



Example of a file view



Set file view

C/C++

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

- Changes the process's view of the data
- Local and shared file pointers are reset to zero
- Collective operation
- ETYPE and FILETYPE must be committed
- DATAREP is a string specifying the data format
  native, internal, external32, or user-defined

Set file view

Fortran

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP,
        INFO, IERROR)
    INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

- Changes the process's view of the data
- Local and shared file pointers are reset to zero
- Collective operation
- ETYPE and FILETYPE must be committed
- DATAREP is a string specifying the data format
  native, internal, external32, or user-defined

Course GSTU2009 File pointers and views Slide 207

Query the file view

C/C++

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp,
          MPI_Datatype *etype, MPI_Datatype *filetype,
          char *datarep)
```

- Returns the process's view of the data

Query the file view

Fortran

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP,
      IERROR)
  INTEGER FH, ETYPE, FILETYPE, IERROR
  CHARACTER*(*) DATAREP
  INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

- Returns the process's view of the data

Data Representations I

- native
  - Data is stored in the file exactly as it is in memory
  - On homogeneous system no loss in precision or I/O performance due to type conversions
  - On heterogeneous systems loss of transparent interoperability
  - No guarantee that MPI files are accessible from C/Fortran
- internal
  - Data is stored in implementation-specific format
  - Can be used in a homogeneous or heterogeneous environment
  - Implementation will perform file conversions if necessary
  - No guarantee that MPI files are accessible from C/Fortran

Data Representations II

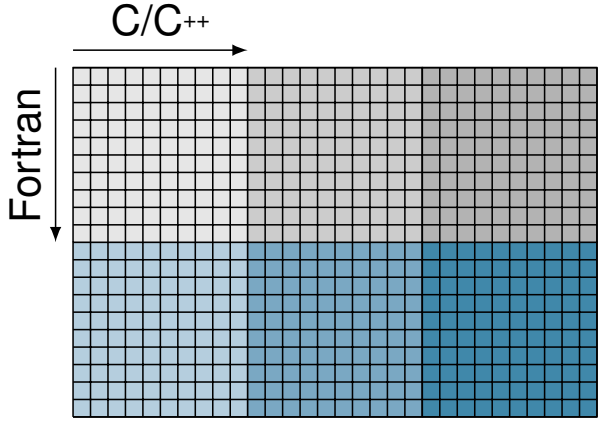
- external32
  - Standardized data representation (big-endian IEEE)
  - Read/write operations convert all data from/to this representation
  - Files can be exported/imported to/from different MPI environments
  - Precision and I/O performance may be lost due to type conversions between
  - native and external32 representations
  - internal may be implemented as external32
  - Can be read/written also by non-MPI programs
- User-defined
  - Allow the user to insert a third party converter into the I/O stream to do the data representation conversion

Parallel I/O to a single file

- A global matrix of values can be stored in a single file
  - The file can be used on an arbitrary number of processes
    - File holds global data that is distributed to the processes via the file view
  - Easier load distribution between several runs
    - Just reset the view
- Using the same ordering in file and local memory structures

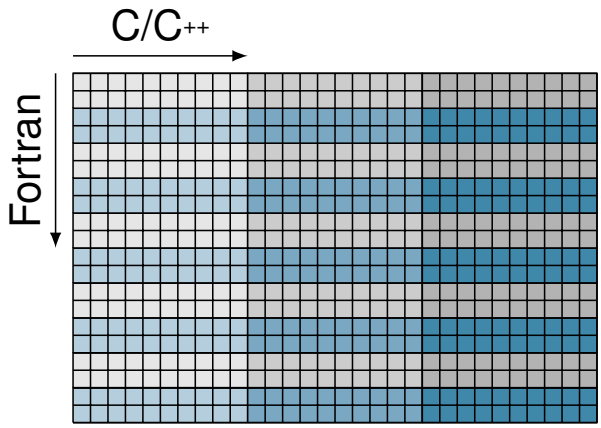
Example with sub-matrices

- Filetype: SUBARRAY
- Process topology: 2x3
- Global array on file: 20x30
- Local array on process: 10x10



Example with the distributed submatrices

- Filetype: DARRAY
- Process topology: 2x3
- Cyclic distribution in first direction in strips of length 2
- Block distribution in second direction



Conversion from offset to displacement

C/C++

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

- Converts a view-relative offset into an absolute byte position (e.g. for use as `DISP` parameter for a new view)

Conversion from offset to displacement

Fortran

```
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
  INTEGER FH, IERROR
  INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

- Converts a view-relative offset into an absolute byte position (e.g. for use as `DISP` parameter for a new view)

Summary

- MPI file handles support
  - individual file pointer access
  - shared file pointer access
  - file access with explicit offsets
- MPI file views define the logical view on a file
  - The file appears to the process like a contiguous stream
- File views determine where the data is located in the file
- Datatype used in file access calls is only specifying the the in-memory layout
- Displacement in file view is used to skip header sections
- File views are valid from the current displacement to the end of the file

JÜLICH FORSCHUNGSZENTRUM

Parallel programming

Collective and non-blocking I/O

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Learning Objectives

At the end of this lesson, you will be able to

- Use collective calls to coordinate file access
- Use non-blocking calls to overlap file access with computation
- Evaluate which type of call to use in a given scenario

Benefits of coordinated Access

- Explicit offsets / individual file pointers:
  - MPI implementation may internally communicate data to avoid serialization of file access
  - MPI implementation may internally communicate data to avoid redundant file access
  - Chance of best performance
- Shared file pointer
  - Data accesses do not have to be serialized by the MPI-implementation
  - First, locations for all accesses can be computed, then accesses can proceed independently (possibly in parallel)
  - Also here: Chance of good performance

Collective file access functions

- Special suffix on function name
  - `_all` with individual file pointer and explicit offset
  - `_ordered` with shared file pointer
- With shared file pointers data is written in the order of process ranks
  - Deterministic outcome as opposed to individual writes with the shared file pointer
- All processes sharing the file handle have to participate

Characteristics of non-blocking I/O

- If supported by hardware, I/O can complete without intervention of the CPU
  - Overlap of computation and I/O
- I/O calls have two parts
  - Initialization
  - Completion
- Implementations may perform all I/O in either part *e.g.* when I/O progress is not supported by the operating system

Non-blocking file access functions

- Individual function calls
  - Initialized by call to `MPI_File_i[...]`
  - Completed by call to `MPI_Wait` or `MPI_Test`
- Collective function calls
  - Also called *split-collective*
  - Initialized by call to `[...]_begin`
  - Completed by call to `[...]_end`
- `STATUS` parameter is replaced by `REQUEST` parameter
  - `STATUS` is parameter on completion call
- File pointers are updated to the new position by the end of the initialization call

Non-blocking write with individual file pointer

C/C++

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

- Same semantics to buffer access as non-blocking point-to-point communication
- Completed by a call to `MPI_Wait` or `MPI_Test`
- Other individual calls analogous

Non-standard interface, if ROMIO is used and not integrated

```
MPIO_Request request
MPI_FILE_IREAD(fh, buf, count, datatype, request)
MPIO_WAIT(request, status)
MPIO_TEST(request, flag, status)
```


Non-blocking write with individual file pointer

Fortran

```
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

- Same semantics to buffer access as non-blocking point-to-point communication
- Completed by a call to `MPI_Wait` or `MPI_Test`
- Other individual calls analogous

Non-standard interface, if ROMIO is used and not integrated

```
MPIO_Request request
MPI_FILE_IREAD(fh, buf, count, datatype, request)
MPIO_WAIT(request, status)
MPIO_TEST(request, flag, status)
```

Split collective file access

C/C++

```
int MPI_File_read_at_all_begin(MPI_File fh,
        MPI_Offset offset, void *buf, int count,
        MPI_Datatype datatype)
int MPI_File_read_at_all_end(MPI_File fh,
        void *buf, MPI_Status *status)
```

- Collective operations may be split into two parts
- Rules and restrictions:
  - Only one active (pending) split or regular collective operation per file handle at any time
  - Split collective operations do not match the corresponding regular collective operation
  - Same BUF argument in `_begin` and `_end` calls

Split collective file access

Fortran

```
MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT,
    DATATYPE, IERROR)
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE),
        IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

- Collective operations may be split into two parts
- Rules and restrictions:
  - Only one active (pending) split or regular collective operation per file handle at any time
  - Split collective operations do not match the corresponding regular collective operation
  - Same BUF argument in `_begin` and `_end` calls

Course GSTU2009 Collective and non-blocking I/O Slide 223

Use cases I

Each process has to read in the complete file

- Solution: `MPI_FILE_READ_ALL`
  - collective with individual file pointers, same view (displacement, etype, filetype) on all processes
  - internally: read in once from disk by several processes (striped), then distributed by broadcast

Use cases II

The file contains a list of tasks, each task requires a different amount of computing time

- Solution: `MPI_FILE_READ_SHARED`
  - non-collective with a shared file pointer
  - same view on all processes (mandatory)

Use cases III

The file contains a list of tasks, each task requires the same amount of computing time

- Solution: `MPI_FILE_READ_ORDERED`
  - collective with a shared file pointer
  - same view on all processes (mandatory)
- or: `MPI_FILE_READ_ALL`
  - collective with individual file pointers
  - different views: filetype with `MPI_TYPE_CREATE_SUBARRAY(...)`
- internally: both may be implemented in the same way, see also the following use case

Use cases IV

The file contains a matrix, distributed block partitioning, each process reads a block

- Solution: generate different filetypes with `MPI_TYPE_CREATE_DARRAY`
  - the view of each process represents the block that is to be read by this process
  - `MPI_FILE_READ_AT_ALL` with `OFFSET=0`
  - collective with explicit offsets
  - reads the whole matrix collectively
  - internally: contiguous blocks read in by several processes (striped), then distributed with all-to-all

Use cases V

Each process has to read the complete file

- Solution: `MPI_FILE_READ_ALL_BEGIN/END`
  - collective with individual file pointers
  - same view (displacement, etype, filetype) on all processes
  - internally: asynchronous read by several processes (striped) started, data distributed with bcast when striped reading has finished

Naming conventions in MPI I/O

- Data access functions
  `MPI_File_write.../MPI_File_read...`
- Positioning
  - Explicit offset: `..._at...`
  - Individual Filepointer: no special qualifier.
  - Shared Filepointer: `..._[shared|ordered]...`
- Synchronism
  - Blocking: no special qualifier
  - Non-Blocking: either `MPI_File_i...` (for individual access) or `..._[begin|end]` for split collective
- Process coordination
  - Individual: no special qualifier
  - Collective: `..._all...`

Other functions

- Pre-allocating space for a file [may be expensive]
  - `MPI_FILE_PREALLOCATE(fh, size)`
- Resizing a file [may speed up first write access to a file]
  - `MPI_FILE_SET_SIZE(fh, size)`
- Querying file size
  - `MPI_FILE_GET_SIZE(filename, size)`
- Querying file parameters
  - `MPI_FILE_GET_GROUP(fh, group)`
  - `MPI_FILE_GET_AMODE(fh, amode)`
- File info object
  - `MPI_FILE_SET_INFO(fh, info)`
  - `MPI_FILE_GET_INFO(fh, info_used)`

Summary

- Collective calls may help the MPI implementation to optimize file access
- Non-blocking calls may allow MPI implementation to overlap file access with computation
- File access can be categorized in positioning, synchronization, and blocking semantics

JÜLICH FORSCHUNGSZENTRUM

Parallel programming

MPI Summary

Course GSTU2009 | Marc-André Hermanns

JÜLICH FORSCHUNGSZENTRUM

Summary

- MPI is an interface for message passing applications
  - Data is explicitly transferred by sending and receiving
- MPI is portable, efficient, and flexible
- MPI provides three main communication paradigms
  - Point-to-point, collective and one-sided
- MPI I/O provides access to portable and scalable I/O
- MPI is the de-facto standard for parallelization in HPC
- MPI is still developed further by the MPI Forum
- Manufacturers of parallel computers and researchers from universities, laboratories and industry are involved in its development