Introduction to the Message Passing Interface (MPI)

Rolf Rabenseifner
rabenseifner@hlrs.de

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Acknowledgments

• This course is partially based on the MPI course developed by the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh.
• Thanks to the EPCC, especially to Neil MacDonald, Elspeth Minty, Tim Harding, and Simon Brown.
• Course Notes and exercises of the EPCC course can be used together with this slides.
Outline

1. MPI Overview
   - one program on several processors
   - work and data distribution

2. Process model and language bindings
   - starting several MPI processes
   - MPI_Init(), MPI_Comm_rank()

3. Messages and point-to-point communication
   - the MPI processes can communicate

4. Non-blocking communication
   - to avoid idle time and deadlocks

5. Derived datatypes
   - transfer any combination of typed data

6. Virtual topologies
   - a multi-dimensional process naming scheme

7. Collective communication
   - e.g., broadcast

8. All other MPI-1 features

[...] = MPI 1.1 chapter

---

Outline

5. Derived datatypes
   - transfer any combination of typed data

6. Virtual topologies
   - a multi-dimensional process naming scheme

7. Collective communication
   - e.g., broadcast

8. All other MPI-1 features

[...] = MPI 1.1 chapter

---

3. — Introduction to the Message Passing Interface (MPI) — 3.
3. — Introduction to the Message Passing Interface (MPI) — 3.3

### Information about MPI

- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18, 1997)
- Marc Snir and William Gropp et al.:  
  (excellent catching up of the standard MPI-1.2 and MPI-2 in a readable form)
- William Gropp, Ewing Lusk and Rajeev Thakur:  
  (or both in one volume, 725 pages, ISBN 026257134X)
- Peter S. Pacheco:  
  **Parallel Programming with MPI.**  
  (very good introduction, can be used as accompanying text for MPI lectures)
- [http://www.hlrs.de/mpi/](http://www.hlrs.de/mpi/)

### Compilation and Parallel Start

- **Your working directory:** ~/MPI/#nr
  
  with #nr = number of your PC
- **Initialization:**  
  in .profile: `USE_MPI=1` (on many systems)
- **Compilation in C:**  
  `cc -o prg prg.c` (on T3E)
  `cc -o prg prg.c -lmpi` (on IRIX)
  `cc -nx -o prg prg.c -lmpi` (on Paragon)
  `mpicc -o prg prg.c` (Hitachi, HP, NEC)
  `f90 -o prg prg.f` (on T3E)
  `f90 -o prg prg.f` (on IRIX)
  `f90 -o prg prg.f -lmpi` (on Paragon)
  `mpif90 -o prg prg.f` (Hitachi, HP, NEC)
- **Compilation in Fortran:**  
  `f77 -nx -o prg prg.f -lmpi` (on Paragon)
  `mpif90 -o prg prg.f` (Hitachi, HP, NEC)
- **Program start on num PEs:**  
  `mpirun -np num /prg` (all, except ...)
  `isub -sz num /prg` (Paragon)
  
  (standard MPI-2)
- **Empty and used partitions:**  
  `fpart; grmview -rw` (on T3E)
  `freepart` (Hitachi, Paragon)
- **MPI Profiling:**  
  `export MPIPROFOUT=stdout` (on T3E)
  `freepart`  
  (Hitachi, Paragon)
- **C examples**  
  `~/MPI/course/C/Ch[2-8]*.c` (Hitachi, HP, NEC)
- **Fortran examples**  
  `~/MPI/course/F/Ch[2-8]*.f .../F/heat/* .../F/mpi_io/*`  
  (the examples of a chapter are only readable after the end of the practical of that chapter)
Chap. 1 MPI Overview

1. **MPI Overview**
   - one program on several processors
   - work and data distribution
   - the communication network

2. Process model and language bindings

3. Messages and point-to-point communication

4. Non-blocking communication

5. Derived datatypes

6. Virtual topologies

7. Collective communication

8. All other MPI-1 features

---

The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**
  - data
  - memory
  - processor

- **Message-Passing Programming Paradigm**
  - data
  - distributed memory
  - parallel processors
  - communication network

---

3. Introduction to the Message Passing Interface (MPI) — 3.
The Message-Passing Programming Paradigm

- Each processor in a message passing program runs a **sub-program**:  
  - written in a conventional sequential language, e.g., C or Fortran,  
  - typically the same on each processor (SPMD),  
  - the variables of each sub-program have  
    • the same name  
    • but different locations (distributed memory) and different data!  
    • i.e., all variables are private  
  - communicate via special send & receive routines (**message passing**)  

Data and Work Distribution

- the value of **myrank** is returned by special library routine  
- the system of **size** processes is started by special MPI initialization program (**mpirun** or **mpiexec**)  
- all distribution decisions are based on **myrank**  
  - i.e., which process works on which data
**Analogy: Electric Installations in Parallel**

- MPI sub-program = work of one electrician on one floor
- data = the electric installation
- MPI communication = real communication to guarantee that the wires are coming at the same position through the floor

**What is SPMD?**

- **Single Program, Multiple Data**
- Same (sub-)program runs on each processor

- MPI allows also MPMD, i.e., Multiple Program, ...
- but some vendors may be restricted to SPMD
- MPMD can be emulated with SPMD
Emulation of Multiple Program (MPMD), Example

- main(int argc, char **argv)
  
  if (myrank < ... /* process should run the ocean model */)  
  
  ocean /* arguments */  
  
  else  
  
  weather /* arguments */  
  
- PROGRAM
  
  IF (myrank < ...) THEN  
  
  CALL ocean (some arguments)
  
  ELSE
  CALL weather (some arguments)
  
ENDIF
END

Messages

- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:
  - sending process
  - receiving process
  - source location
  - destination location
  - source data type
  - destination data type
  - source data size
  - destination buffer size

i.e., the ranks
Access

- A sub-program needs to be connected to a message passing system.
- A message passing system is similar to:
  - mail box
  - phone line
  - fax machine
  - etc.
- MPI:
  - sub-program must be linked with an MPI library
  - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool

Addressing

- Messages need to have addresses to be sent to.
- Addresses are similar to:
  - mail addresses
  - phone number
  - fax number
  - etc.
- MPI: addresses are ranks of the MPI processes (sub-programs)
3. — Introduction to the Message Passing Interface (MPI) —

Slide 17

Reception

- All messages must be received.

Slide 18

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
  - synchronous send
  - buffered = asynchronous send
Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.

Buffered = Asynchronous Sends

- Only know when the message has left.
### Blocking Operations

- Operations are local activities, e.g.,
  - sending (a message)
  - receiving (a message)
- Some operations may **block** until another process acts:
  - synchronous send operation **blocks until** receive is posted;
  - receive operation **blocks until** message is sent.
- Relates to the completion of an operation.
- Blocking subroutine returns only when the operation has completed.

### Non-Blocking Operations

- Non-blocking operation: returns immediately and allow the sub-program to perform other work.
- At some later time the sub-program must **test** or **wait** for the completion of the non-blocking operation.
Non-Blocking Operations (cont’d)

- All non-blocking operations must have matching wait (or test) operations. (Some system or application resources can be freed only when the non-blocking operation is completed.)
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.
- Non-blocking operations are not the same as sequential subroutine calls:
  - the operation may continue while the application executes the next statements!

Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms
- Can be built out of point-to-point communications.
Broadcast

- A one-to-many communication.

Reduction Operations

- Combine data from several processes to produce a single result.

\[ \text{sum} = ? \]
3. — Introduction to the Message Passing Interface (MPI) — 3.

**Barriers**

- Synchronize processes.

**MPI Forum**

- MPI-1 Forum
  - First message-passing interface standard.
  - Sixty people from forty different organizations.
  - Users and vendors represented, from US and Europe.
  - Two-year process of proposals, meetings and review.
  - MPI 1.0 — June, 1994.
  - MPI 1.1 — June 12, 1995.
MPI-2 Forum

- MPI-2 Forum
  - Same procedure.
  - MPI-2: *Extensions to the Message-Passing Interface* document (July 18, 1997).
  - MPI 1.2 — mainly clarifications.
  - MPI 2.0 — extensions to MPI 1.2.

Goals and Scope of MPI

- MPI's prime goals
  - To provide a message-passing interface.
  - To provide source-code portability.
  - To allow efficient implementations.
- It also offers:
  - A great deal of functionality.
  - Support for heterogeneous parallel architectures.
- With MPI-2:
  - Important additional functionality.
  - No changes to MPI-1.
Chap. 2 Process Model and Language Bindings

1. MPI Overview

2. Process model and language bindings
   - starting several MPI processes
   - MPI_Init()
   - MPI_Comm_rank()

3. Messages and point-to-point communication

4. Non-blocking communication

5. Derived datatypes

6. Virtual topologies

7. Collective communication

8. All other MPI-1 features

Header files

- C
  
  \#include <mpi.h>

- Fortran

  include 'mpif.h'
MPI Function Format

- C: error = MPI_Xxxxxx( parameter, ... );
  MPI_Xxxxxx( parameter, ... );
- Fortran: CALL MPI_XXXXXX( parameter, ..., IERROR )

forget absolutely never!

MPI Function Format Details

- Have a look into the MPI standard, e.g., MPI 1.1, page 20. Each MPI routine is defined:
  - language independent,
  - in several programming languages (C, Fortran, C++ [in MPI-2]).

  Output arguments in C:
  - definition in the standard
  - usage in your code:

  ```c
  MPI_Comm_rank( ..., int *rank)
  MPI_Recv(..., MPI_Status *status)
  main...
  {  int myrank;  MPI_Status rcv_status;
    MPI Comm_rank(..., &myrank);  
    MPI_Recv(..., &rcv_status);
  }
  ```

- Last two pages of the standard is the MPI function index,
  - it is ±1 page inexact — test it, e.g., find MPI_Init!
- MPI_...... namespace is reserved for MPI constants and routines, i.e. application routines and variable names must not begin with MPI_.

Initializing MPI

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

```c
#include <mpi.h>
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ....
}
```

- Fortran: `MPI_INIT( IERROR )`

```fortran
INTEGER IERROR
program xxxx
implicit none
include 'mpif.h'
integer ierror
call MPI_Init(ierr)
....
```

- Must be first MPI routine that is called.

Starting the MPI Program

- Start mechanism is implementation dependent
- `mpirun -np number_of_processes /executable` (most implementations)
- `mpiexec -n number_of_processes /executable` (with MPI-2 standard)

- The parallel MPI processes exist at least after MPI_Init was called.
Communicator MPI_COMM_WORLD

- All processes (= sub-programs) of one MPI program are combined in the communicator MPI_COMM_WORLD.
- MPI_COMM_WORLD is a predefined handle in mpi.h and mpif.h.
- Each process has its own rank in a communicator:
  - starting with 0
  - ending with (size-1)

Handles

- Handles identify MPI objects.
- For the programmer, handles are
  - predefined constants in mpi.h or mpif.h
    - example: MPI_COMM_WORLD
    - predefined values exist only after MPI_Init was called
  - values returned by some MPI routines, to be stored in variables, that are defined as
    - in Fortran: INTEGER
    - in C: special MPI typedefs
- Handles refer to internal MPI data structures
Rank

- The rank identifies different processes.
- The rank is the basis for any work and data distribution.

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

Fortran: `MPI_COMM_RANK( comm, rank, ierror)`

```c
INTEGER comm, rank, ierror

myrank = 0
myrank = 1
myrank = 2
myrank = (size - 1)

CALL MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierror)
```

Size

- How many processes are contained within a communicator?

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

Fortran: `MPI_COMM_SIZE( comm, size, ierror)`

```c
INTEGER comm, size, ierror
```
Exiting MPI

- C: int MPI_Finalize()
- Fortran: MPI_FINALIZE(ierr)
  INTEGER ierr

- Must be called last by all processes.

Exercise: Hello World

- Write a minimal MPI program which prints "hello world" by each MPI process.
- Compile and run it on a single processor.
- Run it on several processors in parallel.
- Modify your program so that
  - every process writes its rank and the size of MPI_COMM_WORLD,
  - only process ranked 0 in MPI_COMM_WORLD prints "hello world".
- Why is the sequence of the output non-deterministic?
Advanced Exercises: Hello World with deterministic output

- Discuss with your neighbor, what must be done, that the output of all MPI processes on the terminal window is in the sequence of the ranks.
- Or is there no chance to guarantee this.

Chap.3 Messages and Point-to-Point Communication

1. MPI Overview
2. Process model and language bindings
   - `MPI_Init()`, `MPI_Comm_rank()`
3. Messages and point-to-point communication
   - the MPI processes can communicate
4. Non-blocking communication
5. Derived datatypes
6. Virtual topologies
7. Collective communication
8. All other MPI-1 features
Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic datatype.
  - Derived datatypes.
- Derived datatypes can be built up from basic or derived datatypes.
- C types are different from Fortran types.
- Datatype handles are used to describe the type of the data in the memory.

Example: message with 5 integers

2345  654  96574  -12  7676

MPI Basic Datatypes — C

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
### MPI Basic Datatypes — Fortran

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Fortran datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

- `count=5` INTEGER arr(5)  

#### Point-to-Point Communication
- Communication between two processes.  
- Source process sends message to destination process.  
- Communication takes place within a communicator, e.g., MPI_COMM_WORLD.  
- Processes are identified by their ranks in the communicator.
Sending a Message

- C: `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- Fortran: `MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)`
  `<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR`

- `buf` is the starting point of the message with `count` elements, each described with `datatype`.
- `dest` is the rank of the destination process within the communicator `comm`.
- `tag` is an additional nonnegative integer piggyback information, additionally transferred with the message.
- The tag can be used by the program to distinguish different types of messages.

Receiving a Message

- C: `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- Fortran: `MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)`
  `<type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM INTEGER STATUS(MPI_STATUS_SIZE), IERROR`

- `buf/count/datatype` describe the receive buffer.
- Receiving the message sent by process with rank `source` in `comm`.
- Envelope information is returned in `status`.
- Output arguments are printed `blue-cursive`.
- Only messages with matching `tag` are received.
Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message datatypes must match.
- Receiver’s buffer must be large enough.

Wildcarding

- Receiver can wildcard.
- To receive from any source — source = MPI_ANY_SOURCE
- To receive from any tag — tag = MPI_ANY_TAG
- Actual source and tag are returned in the receiver’s status parameter.
Communication Envelope

- Envelope information is returned from MPI_RECV in `status`.
- C: `status.MPI_SOURCE` `status.MPI_TAG` `count` via `MPI_Get_count()`
- Fortran: `status(MPI_SOURCE)` `status(MPI_TAG)` `count` via `MPI_GET_COUNT()`

Receive Message Count

- C: `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`
- Fortran: `MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)`
  `INTEGER STATUS(MPI_STATUS_SIZE)`
  `INTEGER DATATYPE, COUNT, IERROR`
Communication Modes

- Send communication modes:
  - synchronous send \( \rightarrow \text{MPI\_SSEND} \)
  - buffered [asynchronous] send \( \rightarrow \text{MPI\_BSEND} \)
  - standard send \( \rightarrow \text{MPI\_SEND} \)
  - Ready send \( \rightarrow \text{MPI\_RSEND} \)

- Receiving all modes \( \rightarrow \text{MPI\_RECV} \)

---

Communication Modes — Definitions

<table>
<thead>
<tr>
<th>Sender mode</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous send</td>
<td>Only completes when the receive has started</td>
<td></td>
</tr>
<tr>
<td>\text{MPI_SSEND}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffered send</td>
<td>Always completes (unless an error occurs), irrespective of receiver</td>
<td>needs application-defined buffer to be declared with MPI_BUFFER_ATTACH</td>
</tr>
<tr>
<td>\text{MPI_BSEND}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard send</td>
<td>Either synchronous or buffered</td>
<td>uses an internal buffer</td>
</tr>
<tr>
<td>\text{MPI_SEND}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ready send</td>
<td>May be started only if the matching receive is already posted!</td>
<td>highly dangerous!</td>
</tr>
<tr>
<td>\text{MPI_RSEND}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when a message has arrived</td>
<td>same routine for all communication modes</td>
</tr>
<tr>
<td>\text{MPI_RECV}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Rules for the communication modes

- **Standard send (MPI_SEND)**
  - minimal transfer time
  - may block due to synchronous mode
  - risks with synchronous send
- **Synchronous send (MPI_SSEND)**
  - risk of deadlock
  - risk of serialization
  - risk of waiting → idle time
  - high latency / best bandwidth
- **Buffered send (MPI_BSEND)**
  - low latency / bad bandwidth
- **Ready send (MPI_RSEND)**
  - use never, except you have a 200% guarantee that Recv is already called in the current version and all future versions of your code

Message Order Preservation

- Rule for messages on the same connection, i.e., same communicator, source, and destination rank:
- **Messages do not overtake each other.**
- This is true even for non-synchronous sends.

- If both receives match both messages, then the order is preserved.
Exercise — Ping pong

- Write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message, process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Add timing calls before and after the loop:
  - C: `double MPI_Wtime(void);`
  - Fortran: `DOUBLE PRECISION FUNCTION MPI_WTIME()`
  - MPI_WTIME returns a wall-clock time in seconds.
- At process 0, print out the transfer time of one message
  - in seconds
  - in µs.

```c
if (my_rank==0)                /* i.e., emulated multiple program */
    MPI_Send( ... dest=1 ...)
    MPI_Recv( ... source=1 ...)
else
    MPI_Recv( ... source=0 ...)
    MPI_Send( ... dest=0 ...)
fi
```

see also login-slides
Advanced Exercises — Ping pong latency and bandwidth

- latency = transfer time for short messages
- bandwidth = message size (in bytes) / transfer time

- Print out message transfer time and bandwidth
  - for following send modes:
    - for standard send (MPI_Send)
    - for synchronous send (MPI_Ssend)
  - for following message sizes:
    - 8 bytes (e.g., one double or double precision value)
    - 512 B (= 8*64 bytes)
    - 32 kB (= 8*64**2 bytes)
    - 2 MB (= 8*64**3 bytes)

Chap.4 Non-Blocking Communication

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Non-blocking communication — to avoid idle time and deadlocks
5. Derived datatypes
6. Virtual topologies
7. Collective communication
8. All other MPI-1 features
### Deadlock

- Code in each MPI process:
  
  ```
  MPI_Ssend(..., right_rank, ...)
  MPI_Recv(..., left_rank, ...)
  ```

  Will block and never return, because `MPI_Recv` cannot be called in the right-hand MPI process.

- Same problem with standard send mode (`MPI_Send`), if MPI implementation chooses synchronous protocol.

### Non-Blocking Communications

- Separate communication into three phases:
  - Initiate non-blocking communication
    - returns immediately
    - routine name starting with `MPI_I...`
  - Do some work (perhaps involving other communications?)
  - Wait for non-blocking communication to complete
Non-Blocking Examples

- Non-blocking send
  - MPI_Isend(...) doing some other work
  - MPI_Wait(...)

- Non-blocking receive
  - MPI_Irecv(...) doing some other work
  - MPI_Wait(...)

= waiting until operation locally completed

Non-Blocking Send

- Initiate non-blocking send
  - in the ring example: Initiate non-blocking send to the right neighbor
- Do some work:
  - in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for non-blocking send to complete
Non-Blocking Receive

- Initiate non-blocking receive
  - in the ring example: Initiate non-blocking receive from left neighbor
- Do some work:
  - in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for non-blocking receive to complete

Handles, already known

- Predefined handles
  - defined in mpi.h / mpif.h
  - communicator, e.g., MPI_COMM_WORLD
  - datatype, e.g., MPI_INT, MPI_INTEGER, ...
- Handles can also be stored in local variables
  - memory for datatype handles
    - in C: MPI_Datatype
    - in Fortran: INTEGER
  - memory for communicator handles
    - in C: MPI_Comm
    - in Fortran: INTEGER
Request Handles

Request handles
- are used for non-blocking communication
- **must** be stored in local variables – in C: `MPI_Request`
  - in Fortran: `INTEGER`
- the value
  - **is generated** by a non-blocking communication routine
  - **is used** (and freed) in the `MPI_WAIT` routine

Non-blocking Synchronous Send

- **C**:
  ```c
  MPI_Issend( buf, count, datatype, dest, tag, comm,
              OUT &request_handle);
  MPI_Wait( INOUT &request_handle, &status);
  ```
- **Fortran**:
  ```fortran
  CALL MPI_ISSEND( buf, count, datatype, dest, tag, comm,
                   OUT request_handle, ierror)
  CALL MPI_WAIT( INOUT request_handle, status, ierror)
  ```
- `buf` must not be used between `Issend` and `Wait` (in all progr. languages)
- "`Issend` + `Wait` directly after `Issend" is equivalent to blocking call (`Ssend`)
- `status` is not used in `Issend`, but in `Wait` (with send: nothing returned)
- Fortran problems, see MPI-2, Chap. 10.2.2, pp 284-290

3. — Introduction to the Message Passing Interface (MPI) — 3-35
Non-blocking Receive

- C:
  ```c
  MPI_Irecv (buf, count, datatype, source, tag, comm, 
  OUT &request_handle);
  MPI_Wait( INOUT &request_handle, &status);
  ```
- Fortran:
  ```fortran
  CALL MPI_IRECV (buf, count, datatype, source, tag, comm, 
  OUT request_handle, ierr)
  CALL MPI_WAIT( INOUT request_handle, status, ierr)
  ```
- buf must not be used between Irecv and Wait (in all progr. languages)
- Fortran problems, see MPI-2, Chap. 10.2.2, pp 284-290
- e.g., compiler does not see modifications in buf in MPI_WAIT,
  workaround: call `MPI_ADDRESS(buf, iaddrdummy, ierr)` after MPI_WAIT

Non-blocking Receive and Register Optimization

- Fortran:
  ```fortran
  MPI_IRECV (buf, ..., request_handle, ierr)
  MPI_WAIT(request_handle, status, ierr)
  write (*,*) buf
  ```
  may be compiled as
  ```fortran
  MPI_IRECV (buf, ..., request_handle, ierr)
  registerA = buf
  MPI_WAIT(request_handle, status, ierr) may receive data into buf
  write ('*',*) registerA
  ```
  i.e. old data is written instead of received data!
- Workarounds:
  - buf may be allocated in a common block, or
  - calling `MPI_ADDRESS(buf, iaddrdummy, ierr)` after MPI_WAIT
Non-blocking MPI routines and strided sub-arrays

- **Fortran:**
  
  `MPI_ISEND (buf(7,:,:), ..., request_handle, ierror)`

- **Do not use non-contiguous sub-arrays in non-blocking calls!!!**
- **Use first sub-array element (buf(1,1,9)) instead of whole sub-array (buf(:,:,9:13))**
- **Call by reference necessary → Call by in-and-out-copy forbidden**
  - **use the correct compiler flags!**

**Blocking and Non-Blocking**

- **Send and receive can be blocking or non-blocking.**
- **A blocking send can be used with a non-blocking receive, and vice-versa.**
- **Non-blocking sends can use any mode**
  - standard → `MPI_ISEND`
  - synchronous → `MPI_ISSEND`
  - buffered → `MPI_IBSEND`
  - ready → `MPI_IRSEND`
- **Synchronous mode affects completion, i.e. MPI_Wait / MPI_Test, not initiation, i.e., MPI_I...**
- **The non-blocking operation immediately followed by a matching wait is equivalent to the blocking operation, except the Fortran problems.**
Completion

- C:
  
  ```c
  MPI_Wait( request_handle, &status);
  MPI_Test( request_handle, &flag, &status);
  ```
- Fortran:
  
  ```fortran
  CALL MPI_WAIT( request_handle, status, ierror)
  CALL MPI_TEST( request_handle, flag, status, ierror)
  ```

- one must
  - WAIT or
  - loop with TEST until request is completed, i.e., flag == 1 or .TRUE.

Multiple Non-Blocking Communications

You have several request handles:

- Wait or test for completion of one message
  - MPI_Waitany / MPI_Testany
- Wait or test for completion of all messages
  - MPI_Waitall / MPI_Testall
- Wait or test for completion of as many messages as possible
  - MPI_Waitsome / MPI_Testsome
Exercise — Rotating information around a ring

- A set of processes are arranged in a ring.
- Each process stores its rank in MPI_COMM_WORLD into an integer variable `snd_buf`.
- Each process passes this on to its neighbor on the right.
- Each processor calculates the sum of all values.
- Keep passing it around the ring until the value is back where it started, i.e.
- Each process calculates sum of all ranks.
- Use non-blocking MPI_Isend
  - to avoid deadlocks
  - to verify the correctness, because blocking synchronous send will cause a deadlock

Initialization:
Each iteration:

Fortran:
dest = mod(my_rank+1, size)
source = mod(my_rank-1+size, size)

C:
dest = (my_rank+1) % size;
source = (my_rank-1+size) % size;

Single Program !!!

see also login-slides
**Advanced Exercises — Irecv instead of Issend**


- Or


---

**Chap.5 Derived Datatypes**

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Non-blocking communication
5. Derived datatypes — transfer of any combination of typed data
6. Virtual topologies
7. Collective communication
8. All other MPI-1 features
MPI Datatypes

- Description of the memory layout of the buffer
  - for sending
  - for receiving
- Basic types
- Derived types
  - vectors
  - structs
  - others

Data Layout and the Describing Datatype Handle

```c
struct buff_layout
{
    int i_val[3];
    double d_val[5];
} buffer;

array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=0;
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=...;
MPI_Type_struct(2, array_of_blocklengths,
    array_of_displacements, array_of_types,
    &buff_datatype);
MPI_Type_commit(&buff_datatype);

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

`&buffer = the start address of the data`
`the datatype handle describes the data layout`
Derived Datatypes — Type Maps

- A derived datatype is logically a pointer to a list of entries:
  - **basic datatype at displacement**

<table>
<thead>
<tr>
<th>basic datatype 0</th>
<th>displacement of datatype 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic datatype 1</td>
<td>displacement of datatype 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>basic datatype n-1</td>
<td>displacement of datatype n-1</td>
</tr>
</tbody>
</table>

Example:

```
11 6.36324d+107
22 0 4 8 12 16 20 24
```

A derived datatype describes the memory layout of, e.g., structures, common blocks, subarrays, some variables in the memory.
Contiguous Data

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype

```
oldtype

newtype
```

- C: `int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)`
- Fortran: `MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)`

Vector Datatype

```
oldtype

newtype
```

- holes, that should not be transferred
- `blocklength = 3` elements per block
- `stride = 5` (element stride between blocks)
- `count = 2` blocks

- C: `int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`
- Fortran: `MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)`

INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
Struct Datatype

oldtypes

<table>
<thead>
<tr>
<th>MPI_INT</th>
<th>MPI_DOUBLE</th>
</tr>
</thead>
</table>

newtype

<table>
<thead>
<tr>
<th>addr_0</th>
<th>addr_1</th>
</tr>
</thead>
</table>

block 0 holes, if double needs an 8 byte alignment block 1

- C: int MPI_Type_struct(int count, int *array_of_blocklengths,
  MPI_Aint *array_of_displacements,
  MPI_Datatype *array_of_types, MPI_Datatype *newtype)
- Fortran: MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
  ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES,
  NEWTYPE, IERROR)

Memory Layout of Struct Datatypes

buf_datatype int double

Fixed memory layout:
- C
  struct buff
  { int i_val[3];
   double d_val[5];
  }
- Fortran, common block
  integer i_val(3)
  double precision d_val(5)
  common /bcomm/ i_val, d_val
- Fortran, derived types
  TYPE buff_type
  SEQUENCE
  INTEGER DIMENSION(3): i_val
  DOUBLE PRECISION, &
  DIMENSION(5): d_val
  END TYPE buff_type
  TYPE (buff_type) :: buff_variable

Alternatively, arbitrary memory layout:
- Each array is allocated independently.
- Each buffer is a pair of a 3-int-array and a 5-double-array.
- The length of the hole may be any arbitrary positive or negative value!
- For each buffer, one needs a specific datatype handle
- CAUTION – Fortran register optimi.: MPI_Send & _Recv of ...d_val is invi-
  sible for the compiler → add MPI_Address

in_buf_datatype

in_i_val in_d_val

out_buf_datatype

out_i_val out_d_val
How to compute the displacement

- array_of_displacements[i] := address(block_i) – address(block_0)

- MPI-1
  - C: int MPI_Address(void* location, MPI_Aint *address)
  - Fortran: MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR

- MPI-2
  - C: int MPI_Get_address(void* location, MPI_Aint *address)
  - Fortran: MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
    INTEGER IERROR

Committing a Datatype

- Before a datatype handle is used in message passing communication, it needs to be committed with MPI_TYPE_COMMIT.

- This must be done only once.

- C: int MPI_Type_commit(MPI_Datatype *datatype);

- Fortran: MPI_TYPE_COMMIT(DATATYPE, IERROR)
  INTEGER DATATYPE, IERROR
Size and Extent of a Datatype, I.

- Size := number of bytes that have to be transferred.
- Extent := spans from first to last byte.
- Basic datatypes: Size = Extent = number of bytes used by the compiler.
- Derived datatypes, an example:

```
oldtype
newtype
```

\[
\begin{align*}
\text{size} & := 6 \times \text{size(oldtype)} \\
\text{extent} & := 8 \times \text{extent(oldtype)}
\end{align*}
\]

better visualization of newtype:

Size and Extent of a Datatype, II.

- MPI-1:
  - C: int MPI_Type_size(MPI_Datatype datatype, int *size)
  - C: int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
  - Fortran: MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
    MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
    INTEGER DATATYPE, EXTENT, IERROR

- MPI-2:
  - C: int MPI_Type_get_extent(MPI_Datatype datatype,
    MPI_Aint *lb, MPI_Aint *extent)
  - Fortran: MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
Exercise — Derived Datatypes

• Modify the pass-around-the-ring exercise.
• Use your own result from Chap. 4 or copy our solution:
  cp ~/MPI/course/F/Ch4/ring.f
  cp ~/MPI/course/C/Ch4/ring.c
• Calculate two separate sums:
  – rank integer sum (as before)
  – rank floating point sum
• Use a struct datatype for this
  • with same fixed memory layout for send and receive buffer.

Initialization:

Each iteration:

1. Initialization:
2. Each iteration:
3. 2 3 4 5
4. 2 3 4 5
5. 2 3 4 5

see also login-slides
Advanced Exercises — Sendrecv & Sendrecv_replace

- Substitute your Issend–Recv–Wait method by \texttt{MPI\_Sendrecv} in your ring-with-datatype program:
  - \texttt{MPI\_Sendrecv} is a deadlock-free combination of \texttt{MPI\_Send} and \texttt{MPI\_Recv}.
  - \texttt{MPI\_Sendrecv} is described in the MPI-1 standard. (You can find \texttt{MPI\_Sendrecv} by looking at the function index on the last page of the standard document.)

- Substitute \texttt{MPI\_Sendrecv} by \texttt{MPI\_Sendrecv\_replace}:
  - Three steps are now combined.
  - The receive buffer (rcv\_buf) must be removed.
  - The iteration is now reduced to three statements:
    - \texttt{MPI\_Sendrecv\_replace} to pass the ranks around the ring,
    - computing the integer sum,
    - computing the floating point sum.

Chap.6 Virtual Topologies

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Non-blocking communication
5. Derived datatypes
6. \textbf{Virtual topologies} — a multi-dimensional process naming scheme
7. Collective communication
8. All other MPI-1 features
Example

- Global array \( A(1:3000, 1:4000, 1:500) = 6 \times 10^9 \) words
- on \( 3 \times 4 \times 5 = 60 \) processors
- process coordinates \( 0.2, 0.3, 0.4 \)

- example: on process \( i_{c_2}=2, i_{c_1}=0, i_{c_2}=3 \) (rank=43)
  decomposition, e.g., \( A(2001:3000, 1:1000, 301:400) = 0.1 \times 10^9 \) words

- process coordinates: handled with virtual Cartesian topologies
- Array decomposition: handled by the application program directly

Virtual Topologies

- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimize communications.
How to use a Virtual Topology

- Creating a topology produces a new communicator.
- MPI provides mapping functions:
  - to compute process ranks, based on the topology naming scheme,
  - and vice versa.

Example – A 2-dimensional Cylinder

- Ranks and Cartesian process coordinates

```
  0  |  3  |  6  |  9
  (0,0) | (1,0) | (2,0) | (3,0)

  1  |  4  |  7  | 10
  (0,1) | (1,1) | (2,1) | (3,1)

  2  |  5  |  8  | 11
  (0,2) | (1,2) | (2,2) | (3,2)
```
Topology Types

- Cartesian Topologies
  - each process is connected to its neighbor in a virtual grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course, communication between any two processes is still allowed.

- Graph Topologies
  - general graphs,
  - not covered here.

Creating a Cartesian Virtual Topology

- C: int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

- Fortran: MPI_CART_CREATE( COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)

\[
\begin{align*}
\text{comm_old} & = \text{MPI_COMM_WORLD} \\
\text{ndims} & = 2 \\
\text{dims} & = (4, 3) \\
\text{periods} & = (1/\text{true}, 0/\text{false}) \\
\text{reorder} & = \text{see next slide}
\end{align*}
\]
Example – A 2-dimensional Cylinder

- Ranks and Cartesian process coordinates in `comm_cart`

```
0 (0,0)
3 (1,0)
6 (2,0)
9 (3,0)
1 (0,1)
4 (1,1)
7 (2,1)
10 (3,1)
2 (0,2)
5 (1,2)
8 (2,2)
11 (3,2)
```

- Ranks in `comm` and `comm_cart` may differ, if `reorder = 1` or `.TRUE.
- This reordering can allow MPI to optimize communications

Cartesian Mapping Functions

- Mapping ranks to process grid coordinates

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

```
Fortran: MPI_CART_COORDS(COMM_CART, RANK, MAXDIMS, COORDS, IERROR)
```

- INTEGER COMM_CART, RANK
- INTEGER MAXDIMS, COORDS(*), IERROR
Cartesian Mapping Functions

• Mapping process grid coordinates to ranks

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

Fortran:
```
MPI_CART_RANK(COMM_CART, COORDS, RANK, IERROR)
```

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

Own coordinates

• Each process gets its own coordinates with

```
MPI_Comm_rank(comm_cart, my_rank, ierror)
MPI_Cart_coords(comm_cart, my_rank, maxdims, my_coords, ierror)
```

0 (0,0) 3 (1,0) 6 (2,0) 9 (3,0)
1 (0,1) 4 (1,1) 7 (2,1) 10 (3,1)
2 (0,2) 5 (1,2) 8 (2,2) 11 (3,2)
Cartesian Mapping Functions

- Computing ranks of neighboring processes

- C:
  ```c
  int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int disp,
                   int *rank_source, int *rank_dest)
  ```

- Fortran:
  ```fortran
  INTEGER COMM_CART, DIRECTION
  INTEGER DISP, RANK_SOURCE
  INTEGER RANK_DISP, IERROR
  ```

- Returns MPI_PROC_NULL if there is no neighbor.
- MPI_PROC_NULL can be used as source or destination rank in each communication ➔ Then, this communication will be a noop!

**MPI_Cart_shift – Example**

<table>
<thead>
<tr>
<th>Process</th>
<th>Rank (Cart)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
</tr>
<tr>
<td>1</td>
<td>(0,1)</td>
</tr>
<tr>
<td>2</td>
<td>(0,2)</td>
</tr>
<tr>
<td>3</td>
<td>(1,0)</td>
</tr>
<tr>
<td>4</td>
<td>(1,1)</td>
</tr>
<tr>
<td>5</td>
<td>(1,2)</td>
</tr>
<tr>
<td>6</td>
<td>(2,0)</td>
</tr>
<tr>
<td>7</td>
<td>(2,1)</td>
</tr>
<tr>
<td>8</td>
<td>(2,2)</td>
</tr>
<tr>
<td>9</td>
<td>(3,0)</td>
</tr>
<tr>
<td>10</td>
<td>(3,1)</td>
</tr>
<tr>
<td>11</td>
<td>(3,2)</td>
</tr>
</tbody>
</table>

 invisible input argument: my_rank in cart

- `MPI_Cart_shift(cart, direction, disp, rank_source, rank_dest, ierror)`
- Example on process rank=7:
  - 0 or +1: 4
  - 1 or +1: 6
  - 4 or 10: 8
**Cartesian Partitioning**

- Cut a grid up into *slices*.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.

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

**Fortran:**
```
MPI_CART_SUB( COMM_CART, REMAIN_DIMS, COMM_SLICE, IERROR)
```

**Example**
```
MPI_Cart_sub( comm_cart, remain_dims, comm_sub, ierror)
```

**Ranks and Cartesian process coordinates in comm_sub**

- `true`, `false`
Exercise — One-dimensional ring topology

- Rewrite the pass-around-the-ring program using a one-dimensional ring topology.
- Use the results from Chap. 4 (non-blocking, without derived datatype):
  ~/MPI/course/F/Ch4/ring.f
  ~/MPI/course/C/Ch4/ring.c
- Hints:
  - After calling MPI_Cart_create,
    - there should be no further usage of MPI_COMM_WORLD, and
    - the my_rank must be recomputed on the base of comm_cart.
  - the cryptic way to compute the neighbor ranks should be substituted by one call to MPI_Cart_shift, that should be before starting the loop.
  - Only one-dimensional:
    - direction=0
    - dims and period as normal variables, i.e., no arrays
    - coordinates are not necessary, because coord=rank

Advanced Exercises — Two-dimensional topology

- Rewrite the exercise in two dimensions, as a cylinder.
- Each row of the cylinder, i.e. each ring, should compute its own separate sum of the original ranks in the two dimensional comm_cart.
- Compute the two dimensional factorization with MPI_Dims_create().

```
C:    int MPI_Dims_create(int nnodes, int ndims, int *dims)
Fortran: MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
       INTEGER NNODES, NDIMS, DIMS(*)
       INTEGER IERROR
```

Array dims must be initialized with (0,0)
Chap. 7 Collective Communication

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Non-blocking communication
5. Derived datatypes
6. Virtual topologies
7. Collective communication
   - e.g., broadcast
8. All other MPI-1 features

Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.
Characteristics of Collective Communication

- Collective action over a communicator.
- All processes of the communicator must communicate, i.e. must call the collective routine.
- Synchronization may or may not occur, therefore all processes must be able to start the collective routine.
- All collective operations are blocking.
- No tags.
- Receive buffers must have exactly the same size as send buffers.

Barrier Synchronization

- C: `int MPI_BARRIER(MPI_Comm comm)`
- Fortran: `MPI_BARRIER(COMM, IERROR)`

- MPI_BARRIER is normally never needed:
  - all synchronization is done automatically by the data communication:
    - a process cannot continue before it has the data that it needs.
  - if used for debugging:
    - please guarantee, that it is removed in production.
  - for profiling: to separate time measurement of
    - load imbalance of computation `[MPI_Wtime(); MPI_BARRIER(); MPI_Wtime()]`
    - communication epochs `[MPI_Wtime(); MPI_Allreduce(); ...; MPI_Wtime()]`
  - if used for synchronizing external communication (e.g. I/O):
    - exchanging tokens may be more efficient and scalable than a barrier on MPI_COMM_WORLD.
    - see also advanced exercise of this chapter.
3. Introduction to the Message Passing Interface (MPI)

**Broadcast**

- C: `int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
- Fortran: `MPI_Bcast(BUF, COUNT, DATATYPE, ROOT, COMM, IERROR)`

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

  e.g., root=1

**Scatter**

- C: `int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
- Fortran: `MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)`

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

Example:

`MPI_Scatter(sbuf, 1, MPI_CHAR, rbuf, 1, MPI_CHAR, 1, MPI_COMM_WORLD)`
3. — Introduction to the Message Passing Interface (MPI) — 3. 3-60

### 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)
  ```

- **Fortran:**
  ```fortran
  MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 
  RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
  ```

- **Before gather:**
  - A
  - B
  - C
  - D
  - E

- **After gather:**
  - A
  - B
  - A
  - C
  - D
  - E

### Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- For example:
  - Global sum or product
  - Global maximum or minimum
  - Global user-defined operation

- Floating point rounding may depend on usage of associative law:
  - 
    - `[(d_0 \odot d_1) \odot (d_2 \odot d_3) \odot \ldots \odot (d_{s-2} \odot d_{s-1})]`
    - `((((d_0 \odot d_1) \odot d_2) \odot d_3) \odot \ldots \odot d_{s-2}) \odot d_{s-1})`
Example of Global Reduction

- Global integer sum.
- Sum of all inbuf values should be returned in resultbuf.
- C:
  ```c
  root=0;
  MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
  ```
- Fortran:
  ```fortran
  root=0
  MPI_REDUCE(inbuf, resultbuf, 1, MPI_INTEGER, MPI_SUM, root, MPI_COMM_WORLD, IERROR)
  ```
- The result is only placed in resultbuf at the root process.

Predefined Reduction Operation Handles

<table>
<thead>
<tr>
<th>Predefined operation handle</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of the maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of the minimum</td>
</tr>
</tbody>
</table>
3. — Introduction to the Message Passing Interface (MPI) —

3-62

User-Defined Reduction Operations

- Operator handles
  - predefined – see table above
  - user-defined

- User-defined operation:
  - associative
  - user-defined function must perform the operation vector_A \oplus vector_B
  - syntax of the user-defined function → MPI-1 standard

- Registering a user-defined reduction function:
  - C: `MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *op)`
  - Fortran: `MPI_OP_CREATE(FUNC, COMMUTE, OP, IERROR)`

- COMMUTE tells the MPI library whether FUNC is commutative.
Variants of Reduction Operations

- **MPI_ALLREDUCE**
  - no root,
  - returns the result in all processes
- **MPI_REDUCE_SCATTER**
  - result vector of the reduction operation is scattered to the processes into the real result buffers
- **MPI_SCAN**
  - prefix reduction
  - result at process with rank $i :=$ reduction of inbuf-values from rank 0 to rank $i$

**MPI_ALLREDUCE**

*before* MPI_ALLREDUCE

*after* MPI_ALLREDUCE

- inbuf
- result

AoDoGoJoM
Exercise — Global reduction

- Rewrite the pass-around-the-ring program to use the MPI global reduction to perform the global sum of all ranks of the processes in the ring.

- Use the results from Chap. 4:
  ```
  ~/MPI/course/F/Ch4/ring.f
  ~/MPI/course/C/Ch4/ring.c
  ```

- I.e., the pass-around-the-ring communication loop must be totally substituted by one call to the MPI collective reduction routine.

see also login-slides

---

3. — Introduction to the Message Passing Interface (MPI) — 3. 3-64
Advanced Exercises — Global scan and sub-groups

- Global scan:
  - Rewrite the last program so that each process computes a partial sum.
  - Rewrite in a way that each process prints out its partial result in the correct order:
    - rank=0  →  sum=0
    - rank=1  →  sum=1
    - rank=2  →  sum=3
    - rank=3  →  sum=6
    - rank=4  →  sum=10
  - This can be done, e.g., by sending a token (empty message) from process 0 to process 1, from 1 to 2, and so on (expecting that all MPI-processes’ stdout are synchronously merged to the program’s stdout).

- Global sum in sub-groups:
  - Rewrite the result of the advanced exercise of chapter 6.
  - Compute the sum in each slice with the global reduction.

Chap.8 All Other MPI-1 Features

1. MPI Overview
2. Process model and language bindings
3. Messages and point-to-point communication
4. Non-blocking communication
5. Derived datatypes
6. Virtual topologies
7. Collective communication
8. All other MPI-1 features
Other MPI features (1)

- **Point-to-point**
  - MPI_Sendrecv & MPI_Sendrecv_replace (see advanced exercise of Chap. 5)
  - Null processes, MPI_PROC_NULL (see Chap. 7, slide on MPI_Cart_shift)
  - MPI_Pack & MPI_Unpack
  - MPI_Probe: check length (tag, source rank) before calling MPI_Recv
  - MPI_Iprobe: check whether a message is available
  - MPI_Request_free, MPI_Cancel
  - Persistent requests
  - MPI_BOTTOM (in point-to-point and collective communication)

- **Collective Operations**
  - MPI_Allgather
  - MPI_Alltoall
  - MPI_Reduce_scatter
  - MPI_Gatherv, MPI_Scatterv, MPI_Allgatherv, MPI_Alltoallv

- **Topologies**
  - MPI_DIMS_CREATE (see advanced exercise of Chap. 7)

Other MPI features (2)

- **Groups of processes and their communicators**
  - subgroups / subcommunicators
  - intracommunicator / intercommunicator

- **Attribute caching**

- **Environmental management**
  - inquire MPI_TAG_UB, MPI_HOST, MPI_IO, MPI_WTIME_IS_GLOBAL
    (tag = 0…32767 always possible)

- **Profiling Interface**
  - Each generated handle can be freed.
  - Lower and upper bound marker in derived datatypes:
    - reviewed and modified in MPI-2 — MPI_Type_create_resized()
Other MPI features (3)

- Error Handling
  - the communication should be reliable
  - if the MPI program is erroneous:
    - by default: abort, if error detected by MPI library
    - otherwise, unpredictable behavior
  - Fortran: call MPI_Errhandler_set (comm, MPI_ERRORS_RETURN, ierr)
  - C: MPI_Errhandler_set (comm, MPI_ERRORS_RETURN);
    - ierror returned by each MPI routine
    - undefined state after an erroneous MPI call has occurred
      (only MPI_ABORT(...) should be still callable)

MPI provider

- The vendor of your computers
- The network provider (e.g. with MYRINET)
- MPICH – the public domain MPI library from Argonne
  - for all UNIX platforms
  - for Windows NT, ...
- LAM – another public domain MPI library
- see also at www.lam-mpi.org/mpi/implementation/
  - list of MPI implementations
- other info at www.hlrs.de/mpi/
Summary

MPI-1
- Parallel MPI process model
- Message passing
  - blocking → several modes (standard, buffered, synchronous, ready)
  - non-blocking → to allow message passing from all processes in parallel
    → to avoid deadlocks
  - derived datatypes → to transfer any combination of data in one message
- Virtual topologies → a convenient processes naming scheme
- Collective communications → a major chance for optimization
- Overview on other MPI-1 features