Message Passing with MPI

PPCES 2019

Joachim Protze / Marc-André Hermanns
IT Center / JARA-HPC

Slides by Hristo Iliev
Agenda: MPI Basics

- **Motivation**

- **Part 1**
  - MPI - Concepts
  - Point-to-Point Communication
  - Non-Blocking Communication

- **Part 2**
  - Collective Communication
  - Communicator Basics
Motivation
Clustering

- HPC market is at large dominated by distributed memory *multicomputers*: *clusters* and specialised *supercomputers*
- Nodes have no direct access to other nodes’ memory and run a separate copy of the (possibly stripped down) OS
Parallel Architectures

- **Shared Memory**

  → All processing elements (P) have direct access to the main memory block (M)
Parallel Architectures

- **Shared Memory**
  - All processing elements (P) have direct access to the main memory block (M).

- Data exchange is achieved through read/write operations on shared variables located in the global address space.
Parallel Architectures

- **Distributed Memory**

  ➔ Each processing element (P) has its separate main memory block (M)
Parallel Architectures

- Distributed Memory

- Each processing element (P) has its separate main memory block (M)

- Data exchange is achieved through message passing over the network

- Data exchange is achieved through message passing over the network
Parallel Architectures

- **Distributed Memory**

  - Each processing element (P) has its separate main memory block (M)
  - Data exchange is achieved through message passing over the network
  - Message passing can be either explicit (MPI) or implicit (PGAS)
  - Programs typically implemented as a set of OS entities with own (virtual) address spaces – processes
  - No shared variables
    - No data races
    - Explicit synchronisation mostly unneeded
      - Results as side effect of the send-receive semantics
Processes

- A process is a running in-memory instance of an executable file
  - Executable code, e.g., binary machine instructions
  - One or more threads of execution sharing memory address space
  - Memory: data, heap, stack, processor state (CPU registers and flags)
  - Operating system context (e.g. signals, I/O handles, etc.)
  - PID

- Isolation and protection
  - A process cannot interoperate with other processes or access their context (even on the same node) without the help of the operating system
  - No direct inter-process data exchange (isolated/virtual address spaces)
  - No direct inter-process synchronisation
SPMD Model

- Abstractions make programming and understanding easier

**Single Program Multiple Data**

→ Multiple instruction flows (instances) from a Single Program working on Multiple (different parts of) Data

→ Instances could be threads (OpenMP) and/or processes (MPI)

→ Each instance receives a unique ID – can be used for flow control

```c
if (myID == specificID) {
    do_something();
} else {
    do_something_different();
}
```
SPMD Model

- **SPMD Program Lifecycle** – multiple processes (e.g. MPI)

  ![Diagram showing the SPMD program lifecycle with the following steps:
  - **Source Code**
  - **Compile & Link** to create an **Executable**
  - **SPMD Launch** to spawn multiple processes (e.g., MPI processes)
  - **Parallel Execution** of the processes
  - **Result**]

Message Passing with MPI (PPCES 2019)
Joachim Protze / Marc-André Hermanns | IT Center der RWTH Aachen University
SPMD Environments

- Provide dynamic identification of all peers
  → Who else is also working on this problem?

- Provide robust mechanisms to exchange data
  → Whom to send data to / From whom to receive the data?
  → How much data?
  → What kind of data?
  → Has the data arrived?

- Provide synchronisation mechanisms
  → Have all processes reached same point in the program execution flow?

- Provide methods to launch and control a set of processes
  → How do we start multiple processes and get them to work together?

- Portability
MPI - Concepts
Message Passing Interface

- The de-facto standard API for explicit message passing nowadays
- A moderately large standard (v3.1 is a 868 pages long)
- Maintained by the non-profit Message Passing Interface Forum
  

- Many concrete implementations of the MPI standard
  - Open MPI, MPICH, Intel MPI, MVAPICH, MS-MPI, etc.

- MPI is used to express the explicit interaction (communication) in programs for computers with distributed memory

- MPI provides source level portability of parallel applications between different implementations and hardware platforms
MPI

A language-independent specification (LIS) of a set of communication and I/O operations

→ Standard bindings for C and Fortran
  → Concrete function prototypes / interfaces

→ Non-standard bindings for other languages exist:
  → C++ Boost.MPI
  → Java Open MPI, MPJ Express
  → Python mpi4py

Unlike OpenMP and PGAS languages, MPI does not extend the base language, but provides a set of library functions (+ specialised runtime) and makes use of existing compilers
More Information & Documentation

- The MPI Forum document archive (free standards for everyone!)
  → http://www.mpi-forum.org/docs/

- The MPI home page at Argonne National Lab
  → http://www-unix.mcs.anl.gov/mpi/

- Open MPI (default MPI implementation on the RWTH cluster)
  → http://www.open-mpi.org/

- Manual pages
  → man MPI
  → man MPI_Xxx_yyy_zzz (for all MPI calls)
Literature about MPI

- **MPI: The Complete Reference Vol. 1**
  The MPI Core
  by Marc Snir, Steve Otto, Steven Huss-Lederman,
  David Walker, Jack Dongarra

- **MPI: The Complete Reference Vol. 2**
  The MPI Extensions
  by William Gropp, Steven Huss-Lederman,
  Andrew Lumsdain, Ewing Lusk, Bill Nitzberg,
  William Saphir, Marc Snir
Literature about MPI

- **Using MPI**
  by William Gropp, Ewing Lusk, Anthony Skjellum

- **Using MPI-2**
  by William Gropp, Ewing Lusk, Rajeev Thakur

- **Parallel Programming with MPI**
  by Peter Pacheco
  Morgan Kaufmann Publishers, 1996
Literature about MPI

- Using MPI
  by William Gropp, Ewing Lusk, Anthony Skjellum

- Using Advanced MPI
  by William Gropp, Torsten Höfler, Ewing Lusk, Rajeev Thakur
MPI Basics – Agenda

- **MPI Basics**
  - Start-up, initialisation, finalisation, and shutdown

- **Point-to-Point Communication**
  - Send and receive
  - Basic MPI data types
  - Message envelope
  - Combined send and receive
  - Send modes
  - Non-blocking operations
  - Common pitfalls
General Structure of an MPI Program

Start-up, initialisation, finalisation, and shutdown – C

1. Inclusion of the MPI header file
   
2. Pre-initialisation mode: uncoordinated
   • No MPI function calls allowed with few exceptions
   • All program instances run exactly the same code

3. Initialisation of the MPI environment
   Implicit synchronisation

4. Parallel MPI code
   Typically computation and communication

5. Finalisation of the MPI environment
   Internal buffers are flushed

6. Post-finalisation mode: uncoordinated
   • No MPI function calls allowed with few exceptions

C

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    ... some code ...
    MPI_Init(&argc, &argv);

    ... computation & communication ...

    MPI_Finalize();
    ... wrap-up ...
    return 0;
}
```
General Structure of an MPI Program

Start-up, initialisation, finalisation, and shutdown – Fortran

1. Inclusion of the MPI module
2. Pre-initialisation mode: uncoordinated
   - No MPI function calls allowed with few exceptions
   - All program instances run exactly the same code
3. Initialisation of the MPI environment
   Implicit synchronisation
4. Parallel MPI code
   Typically computation and communication
5. Finalisation of the MPI environment
   Internal buffers are flushed
6. Post-finalisation mode: uncoordinated
   - No MPI function calls allowed with few exceptions

PROGRAM example
USE mpi_f08 ! USE mpi

... some code ...

CALL MPI_Init(ierr)

... computation & communication ...

CALL MPI_Finalize(ierr)

... wrap-up ...
END
General Structure of an MPI Program

- How many processes are there in total?
- Who am I?

```
#include <mpi.h>

int main(int argc, char **argv)
{
    ... some code ...
    int ierr = MPI_Init(&argc, &argv), numberOfProcs, rank;
    ... other code ...
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &numberOfProcs);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ... computation & communication ...
    ierr = MPI_Finalize();
    ... wrap-up ...
    return 0;
}
```

1. Obtains the number of processes (ranks) in the MPI program

   Example: if the job was started with 4 processes, then `numberOfProcs` will be set to 4 by the call

2. Obtains the identity of the calling process within the MPI program

   NB: MPI processes are numbered starting from 0

   Example: if there are 4 processes in the job, then `rank` receive value of 0 in the first process, 1 in the second process, and so on
General Structure of an MPI Program

- How many processes are there in total?
- Who am I?

```fortran
PROGRAM example
  USE mpi_f08  ! USE mpi
  INTEGER :: rank, numberOfProcs, ierr
  ... some code ...
  CALL MPI_Init(ierr)
  ... other code ...
  CALL MPI_Comm_size(MPI_COMM_WORLD,&
                    numberOfProcs, ierr)
  CALL MPI_Comm_rank(MPI_COMM_WORLD,&
                    rank, ierr)
  ... computation & communication ...
  CALL MPI_Finalize(ierr)
  ... wrap-up ...
END PROGRAM example
```

1. Obtains the number of processes (ranks) in the MPI program

   Example: if the job was started with 4 processes, then `numberOfProcs` will be set to 4 by the call

2. Obtains the identity of the calling process within the MPI program

   NB: MPI processes are numbered starting from 0

   Example: if there are 4 processes in the job, then `rank` receive value of 0 in the first process, 1 in the second process, and so on
- The processes in any MPI program are initially indistinguishable
- MPI_Init assigns each process a unique identity – rank
Ranks

- The processes in any MPI program are initially indistinguishable
- **MPI_Init** assigns each process a unique identity – rank
  - Without personality, the started MPI processes cannot do coordinated parallel work in the pre-initialisation mode
  - Ranks range from 0 up to the total number of processes minus 1
- **Ranks are associated with the so-called communicators**
  - Logical contexts where communication takes place
  - Represent groups of MPI processes with some additional information
  - The most important one is the world communicator **MPI_COMM_WORLD**
    - Contains all processes launched *initially* as part of the MPI program
  - Ranks are always provided in MPI calls in combination with the corresponding communicator
Basic MPI Use

- **Initialisation:**
  
  C: `ierr = MPI_Init(&argc, &argv);`
  
  Fortran: `CALL MPI_Init(ierr)`
  
  → Initialises the MPI library and makes the process member of the world communicator
  
  → [C] Modern MPI implementations allow both arguments to be NULL, otherwise they must point to the arguments of `main()`
  
  → May not be called more than once for the duration of the program execution
  
  → Error code as return value in [C] and additional parameter in [F]

- **Finalisation:**
  
  C: `ierr = MPI_Finalize();`
  
  Fortran: `CALL MPI_Finalize(ierr)`
  
  → Cleans up the MPI library and prepares the process for termination
  
  → Must be called once before the process terminates
  
  → Having other code after the finalisation call is not recommended
Basic MPI Use

■ Number of processes in the MPI program:

C:   ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
Fortran: CALL MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

→ Obtains the number of processes initially started in the MPI program
   (the size of the world communicator)
→ size is an integer variable
→ MPI_COMM_WORLD is a predefined constant MPI handle that represents
   the world communicator

■ Process identification:

C:   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
Fortran: CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

→ Determines the rank (unique ID) of the process within the world communicator
→ rank is an integer variable; receives value between 0 and #processes - 1
Lab time: Exercise 0

Message Passing with MPI (PPCES 2019)
Joachim Protze / Marc-André Hermanns | IT Center der RWTH Aachen University

(cc) by dongpeiyen1000
CC BY-NC-SA 3.0
Basic MPI Use

- **Error codes indicate the success of the operation:**
  - Failure is indicated by error codes other than **MPI_SUCCESS**
  - C: \( \text{if (MPI_SUCCESS != MPI_Init(NULL, NULL))} \ldots \)
  - Fortran: \( \text{CALL MPI_Init(ierr)} \)
  - \( \text{IF (ierr /= MPI_SUCCESS)} \ldots \)

- **An MPI error handler is called first before the call returns**
  - The default error handler for non-I/O calls aborts the entire MPI program!
  - Error checking in simple programs is redundant

- **NB: Actual MPI error code values are implementation specific**
  - Use **MPI_Error_string** to derive human readable information
Basic MPI Use

- **MPI objects (e.g. communicators) are referenced via handles**
  - Opaque process-local values
  - Cannot be passed from one process to another

- **C (mpi.h)**
  - typedef'd handle types: MPI_Comm, MPI_Datatype, MPI_File, etc.

- **Fortran (USE mpi)**
  - All handles are INTEGER values
  - Easy to pass the wrong handle type

- **Fortran 2008 (USE mpi_f08)**
  - Wrapped INTEGER values: TYPE(MPI_Comm), TYPE(MPI_File), etc.
  - The INTEGER handle is still available: comm%MPI_VAL
MPI as an SPMD Environment

- Provide dynamic identification of all peers
  → Who am I and who else is also working on this problem?

- Provide robust mechanisms to exchange data
  → Whom to send data to / From whom to receive the data?
  → How much data?
  → What kind of data?
  → Has the data arrived?

- Provide synchronisation mechanisms
  → Have all processes reached same point in the program execution flow?

- Provide methods to launch and control a set of processes
  → How do we start multiple processes and get them to work together?

- Portability
Point-to-Point Communication
Message Passing

The goal is to enable communication between processes that share no memory space

Explicit message passing requires:

→ Send and receive primitives (operations)
→ Known addresses of both the sender and the receiver
→ Specification of what has to be sent/received
Sending Data

### Sending a message:

- **data:** location in memory of the data to be sent
- **count:** number of data elements to be sent (MPI is array-oriented)
- **type:** Handle of the *MPI datatype* of the buffer content
- **dest:** rank of the receiver
- **tag:** additional identification of the message
  - ranges from 0 to UB (impl. dependant but not less than 32767)
- **comm:** communication context (communicator handle)

```c
MPI_Send (data, count, type, dest, tag, comm, ierr)
```

```fortran
MPI_Send (data, count, type, dest, tag, comm, ierr)
```
Receiving Data

Receiving a message:

- **data**: location of the receive buffer
- **count**: size of the receive buffer in data elements
- **type**: Handle of the MPI datatype of the data elements
- **source**: rank of the sender or **MPI_ANY_SOURCE** (wildcard)
- **tag**: message tag or **MPI_ANY_TAG** (wildcard)
- **comm**: communication context
- **status**: status of the receive operation or **MPI_STATUS_IGNORE**

**C**

```c
MPI_Recv (data, count, type, source, tag, comm, status, ierr)
```

**Fortran**

```fortran
MPI_Recv (data, count, type, src, tag, comm, status, ierr)
```
MPI Datatypes

- MPI is a library – it cannot infer the type of elements in the supplied buffer at run time and that’s why it has to be told what it is

- MPI datatypes tell MPI how to:
  - read binary values from the send buffer
  - write binary values into the receive buffer
  - correctly apply value alignments
  - convert between machine representations in heterogeneous environments

- MPI datatype must match the language type(s) in the data buffer
- MPI datatype must match between sender and receiver
- MPI datatypes are handles and cannot be used to declare variables

- Type signatures of MPI datatypes
  - Sequence of basic MPI datatypes
  - Signature of basic MPI datatypes is sequence with 1 element: the type itself
**MPI Datatypes**

MPI provides many predefined datatypes for each language binding:

→ C

<table>
<thead>
<tr>
<th>MPI data type</th>
<th>C data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>short</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_UNSIGNED_INT</td>
<td>unsigned int</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>-</td>
</tr>
</tbody>
</table>

8 binary digits no conversion used for untyped data
MPI Datatypes

- MPI provides many predefined datatypes for each language binding:
  - C
  - Fortran

<table>
<thead>
<tr>
<th>MPI data type</th>
<th>Fortran data type</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_REAL8</td>
<td>REAL(KIND=8)</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>...</td>
<td>...</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>-</td>
</tr>
</tbody>
</table>
Message Passing as Assignment

- Message passing in MPI is explicit:

- The value of variable $b$ in rank 1 is copied into variable $a$ in rank 0

- For now, assume that $comm$ is always MPI_COMM_WORLD
  
  $\rightarrow$ We will talk about other communicators later on
Message Passing as Assignment

Message passing in MPI is explicit:

```c
if (rank == 0) {
    MPI_Recv(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
    a = b;
}
else if (rank == 1) {
    MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```
MPI as an SPMD Environment

- **Provide dynamic identification of all peers**
  - Who am I and who else is also working on this problem?

- **Provide robust mechanisms to exchange data**
  - Whom to send data to / From whom to receive the data?
  - How much data?
  - What kind of data?
  - Has the data arrived? (only the receiver knows)

- **Provide synchronisation mechanisms**
  - Have all processes reached same point in the program execution flow?

- **Provide methods to launch and control a set of processes**
  - How do we start multiple processes and get them to work together?

- **Portability**
#include <mpi.h>

int main(int argc, char **argv)
{
    int nprocs, rank, data;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)
        MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
    else if (rank == 1)
        MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
Compiling MPI Programs

- MPI is a typical library with C header files, Fortran modules, etc.
- Some MPI vendors provide convenience compiler wrappers:
  
  - cc  ➔ mpicc
  - c++ ➔ mpic++
  - f90 ➔ mpif90

- On RWTH Compute Cluster (depending on the loaded modules):
  
  - $CC  ➔ $MPICC
  - $CXX ➔ $MPICXX
  - $FC ➔ $MPIFC
Compiling MPI Programs

- MPI is a typical library with C header files, Fortran modules, etc.
- Some MPI vendors provide convenience compiler wrappers:

```
cluster:~[1]$ $MPICC --show
icc
-I/opt/MPI/openmpi-1.6.5/linux/intel/include
-I/opt/MPI/openmpi-1.6.5/linux/intel/include/openmpi
-fexceptions
-pthread
-I/opt/MPI/openmpi-1.6.5/linux/intel/lib
-Wl,-rpath,/opt/MPI/openmpi-1.6.5/linux/intel/lib
-I/opt/MPI/openmpi-1.6.5/linux/intel/lib
-L/opt/MPI/openmpi-1.6.5/linux/intel/lib
-lmpi
-ldl
-Wl,--export-dynamic
-lns1
-lutil
```
Executing MPI Programs

- Most MPI implementations provide a special launcher program:

```
mpiexec -n nprocs ... program <arg1> <arg2> <arg3> ...
```

→ launches \texttt{nprocs} instances of \texttt{program} with command-line arguments \texttt{arg1}, \texttt{arg2}, ... and provides the MPI library with enough information in order to establish network connections between the processes

- The standard specifies the \texttt{mpiexec} program but does not require it:

→ IBM BG/Q: \texttt{runjob --np 1024 ...}
→ SLURM resource manager: \texttt{srun ...}

- On RWTH Compute Cluster:

→ interactive jobs

```
$MPIEXEC -n nprocs ... program <arg1> <arg2> <arg3> ...
```

→ batch jobs

```
$MPIEXEC $FLAGS_MPI_BATCH ... program <arg1> <arg2> <arg3> ...
```
Executing MPI Programs

Most MPI implementations provide a special launcher program:

```
mpiexec -n nprocs ... program <arg1> <arg2> <arg3> ...
```

→ launches \textit{nprocs} instances of \textit{program} with command-line arguments \textit{arg1}, \textit{arg2}, \ldots\ and provides the MPI library with enough information in order to establish network connections between the processes
→ Sometimes called \textit{mpirun}

The launcher often performs more than simply launching processes:

→ Helps MPI processes find each other and establish the world communicator
→ Redirects the standard output of all ranks to the terminal
→ Redirects the terminal input to the standard input of rank 0
→ Forwards received signals (Unix-specific)
MPI as an SPMD Environment

- **Provide dynamic identification of all peers**
  → Who am I and who else is also working on this problem?

- **Provide robust mechanisms to exchange data**
  → Whom to send data to / From whom to receive the data?
  → How much data?
  → What kind of data?
  → Has the data arrived? (only the receiver knows)

- **Provide synchronisation mechanisms**
  → Have all processes reached same point in the program execution flow?

- **Provide methods to launch and control a set of processes**
  → How do we start multiple processes and get them to work together?

**Portability**

---

Message Passing with MPI (PPCES 2019)
Joachim Protze / Marc-André Hermanns | IT Center der RWTH Aachen University
Compile and Run a Simple MPI Program

```
cluster:~[1]$ vim hello.c

cluster:~[2]$ MPICC -o hello.exe hello.c

cluster:~[3]$ MPIEXEC -n 4 hello.exe
Hello world from rank 2 of 4
Hello world from rank 0 of 4
Hello world from rank 1 of 4
Hello world from rank 3 of 4
```
Message Envelope and Matching

- Reception of MPI messages is done by matching their envelope
- Send operation

\[
\text{MPI\_Send} \left( \text{void *data, int count, MPI\_Datatype type, int dest, int tag, MPI\_Comm comm} \right)
\]

- Message Envelope:

<table>
<thead>
<tr>
<th></th>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Implicit</td>
<td>Explicit, wildcard possible (MPI_ANY_SOURCE)</td>
</tr>
<tr>
<td>Destination</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Tag</td>
<td>Explicit</td>
<td>Explicit, wildcard possible (MPI_ANY_TAG)</td>
</tr>
<tr>
<td>Communicator</td>
<td>Explicit</td>
<td>Explicit</td>
</tr>
</tbody>
</table>

- Receive operation

\[
\text{MPI\_Recv} \left( \text{void *data, int count, MPI\_Datatype type, int source, int tag, MPI\_Comm comm, MPI\_Status *status} \right)
\]
Message Envelope and Matching

- Reception of MPI messages is also dependent on the data.
- Recall:

  ```
  MPI_Send (void *data, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
  ```

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

- The standard expects type signatures at both ends to match → Not verified by most implementations
- Matching sends and receives must always come in pairs
- NB: messages do not aggregate

Rank 0:  
MPI_Send(myArr,1,MPI_INT,1,0,MPI_COMM_WORLD)  
... some code ...  
MPI_Send(myArr,1,MPI_INT,1,0,MPI_COMM_WORLD)  

Rank 1:  
MPI_Recv(myArr,2,MPI_INT,0,0,MPI_COMM_WORLD,&stat)  
... some code ...  

Unmatched
Lab time: Exercise 1
Message Reception and Status

- The receive buffer must be able to fit the entire message
  - send count ≤ receive count → OK (but check status)
  - send count > receive count → ERROR (message truncation)

- The MPI status object holds information about the received message

  C: MPI_Status status;
  - status.MPI_SOURCE: message source rank
  - status.MPI_TAG: message tag
  - status.MPI_ERROR: receive status code
Message Reception and Status

- The receive buffer must be able to fit the entire message
  - send count ≤ receive count, OK (but check status)
  - send count > receive count, ERROR (message truncation)

- The MPI status object holds information about the received message

- Fortran: INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
  - status(MPI_SOURCE) message source rank
  - status(MPI_TAG) message tag
  - status(MPI_ERROR) receive status code
Message Reception and Status

- The receive buffer must be able to fit the entire message
  - send count ≤ receive count  OK (but check status)
  - send count > receive count  ERROR (message truncation)

- The MPI status object holds information about the received message

- Fortran 2008:
  - TYPE(MPI_Status) :: status
  - status%MPI_SOURCE  message source rank
  - status%MPI_TAG  message tag
  - status%MPI_ERROR  receive status code
Inquiry Operations

- **Blocks until a matching message appears:**

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

  - Message is not received, one must call **MPI_Recv** to receive it
  - Information about the message is stored in the status field

  ```c
  MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
  ```

  - Checks for any message in the given communicator

- **Message size inquiry:**

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

  - Calculates how many integral **datatype** elements can be formed from the data in the message referenced by **status**
  - If the number is not integral, **count** is set to **MPI_UNDEFINED**
  - Can be used with the status from **MPI_Recv** too
Operation Completion

- MPI operations complete once the message buffer is no longer in use by the MPI library and is thus free for reuse

- Send operations complete:
  - once the message is constructed \textit{and}
  - placed completely onto the network \textit{or}
  - buffered completely (by MPI, the OS, the network, …)

- Receive operations complete:
  - once the entire message has arrived and has been placed into the buffer

- Blocking MPI calls only return once the operation has completed
  - \texttt{MPI\_Send} and \texttt{MPI\_Recv} are blocking
## Blocking Calls

### Blocking send (w/o buffering) and receive calls:

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI_Send</strong></td>
<td><strong>MPI_Recv</strong></td>
</tr>
<tr>
<td>Data must remain constant</td>
<td>Data must not be used</td>
</tr>
</tbody>
</table>

Program

1. Send the envelope and wait
2. Acknowledge envelope match
3. First message part
4. Intermediate message part
5. Intermediate message part
6. Last message part

Message Passing with MPI (PPCES 2019)
Joachim Protze / Marc-André Hermanns | IT Center der RWTH Aachen University
Deadlocks

- Both MPI_Send and MPI_Recv calls are blocking:
  - The receive operation only returns after a matching message has arrived
  - The send operation *might* be buffered *(implementation-specific!!!)* and therefore return before the message is actually placed onto the network
  - Larger messages are usually sent only when both the send and the receive operations are active (synchronously)
  - *Never rely on any implementation-specific behaviour!!!*

- Deadlock in a typical data exchange scenario:

  ![Diagram showing a deadlock scenario](image-url)
Deadlocks

- Both MPI_Send and MPI_Recv calls are blocking:
  - The receive operation only returns after a matching message has arrived
  - The send operation *might* be buffered (*implementation-specific!!!*) and therefore return before the message is actually placed onto the network
  - Larger messages are usually sent only when both the send and the receive operations are active (synchronously)
  - *Never rely on any implementation-specific behaviour!!!*

- Deadlock prevention in a typical data exchange scenario:

```plaintext
<table>
<thead>
<tr>
<th>Time</th>
<th>Rank 0</th>
<th>Rank 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Receive from 1</td>
<td>Send to 0</td>
</tr>
<tr>
<td></td>
<td>Receive from 1</td>
<td>Receive from 0</td>
</tr>
</tbody>
</table>
```

Not symmetric / Doesn’t scale
Combined Send and Receive

MPI_Sendrecv (void *senddata, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvdata, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

- Combines message send and receive into a single call

<table>
<thead>
<tr>
<th></th>
<th>Send</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>senddata</td>
<td>recvdata</td>
</tr>
<tr>
<td>Count</td>
<td>sendcount</td>
<td>recvcount</td>
</tr>
<tr>
<td>Type</td>
<td>sendtype</td>
<td>recvtype</td>
</tr>
<tr>
<td>Destination</td>
<td>dest</td>
<td>-</td>
</tr>
<tr>
<td>Source</td>
<td>-</td>
<td>source</td>
</tr>
<tr>
<td>Tag</td>
<td>sendtag</td>
<td>recvtag</td>
</tr>
<tr>
<td>Communicator</td>
<td>comm</td>
<td>comm</td>
</tr>
<tr>
<td>Receive status</td>
<td>-</td>
<td>status</td>
</tr>
</tbody>
</table>
Combined Send and Receive

- Sends one message and receives one message (in any order) without deadlocking (unless unmatched)
- Send and receive buffers **must not** overlap!

```c
MPI_Sendrecv (void *senddata, int sendcount, MPI_Datatype sendtype,
             int dest, int sendtag, void *recvdata, int recvcount,
             MPI_Datatype recvtype, int source, int recvtag,
             MPI_Comm comm, MPI_Status *status)
```

- First sends a message to `dest`, **then** receives a message from `source`, using the same memory location, elements count and datatype for both operations
- Usually slower than `MPI_Sendrecv` and might use more memory

```c
MPI_Sendrecv_replace (void *data, int count, MPI_Datatype datatype,
                      int dest, int sendtag, int source, int recvtag,
                      MPI_Comm comm, MPI_Status *status)
```
Message Ordering

- **Order is preserved in a given communicator for point-to-point operations between any pair of processes**
  - Messages within some communicator to the same rank are non-overtaking
  - Probe/receive returns the earliest matching message

- **Order is not guaranteed for**
  - messages sent within different communicators
  - messages arriving from different senders

```c
MPI_Status status;
MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
... allocate buffer based on message size ...
MPI_Recv(buffer, size, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
```
Message Ordering

- **Order is preserved in a given communicator for point-to-point operations between any pair of processes**
  - Messages within some communicator to the same rank are non-overtaking
  - Probe/receive returns the earliest matching message

- **Order is not guaranteed for**
  - messages sent within different communicators
  - messages arriving from different senders

```c
MPI_Status status;

MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
... allocate buffer based on message size ...
MPI_Recv(buffer, size, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD, &status);
```

Also applies to sequences of wildcard receives
Non-Blocking Communication
Blocking Calls

- Blocking send (w/o buffering) and receive calls:

<table>
<thead>
<tr>
<th>Program</th>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send the envelope and wait</td>
<td>Acknowledge envelope match</td>
</tr>
<tr>
<td></td>
<td>First message part</td>
<td>Intermediate message part</td>
</tr>
<tr>
<td></td>
<td>Intermediate message part</td>
<td>Last message part</td>
</tr>
<tr>
<td>Data must remain constant</td>
<td></td>
<td>Data must not be used</td>
</tr>
</tbody>
</table>

Data must not be used.
Non-Blocking Calls

- Non-blocking MPI calls return immediately while the communication operation continues asynchronously in the background.

- Each non-blocking operation is represented by a request handle:
  - C: \texttt{MPI\_Request}
  - Fortran: \texttt{INTEGER}
  - Fortran 2008: \texttt{TYPE(MPI\_Request)}

- Non-blocking operations are progressed by certain MPI calls but most notably by the test and wait MPI calls.

- Blocking MPI calls are equivalent to making a non-blocking call and waiting immediately afterwards for the operation to complete.

- Used to overlay communication and computation and to prevent possible deadlocks.
**Non-Blocking Send and Receive**

- **Initiation of non-blocking send and receive operations:**
  
  ```c
  MPI_Isend (void *data, int count, MPI_Datatype dataType,
             int dest, int tag, MPI_Comm comm, MPI_Request *request)
  ```

  ```c
  MPI_Irecv (void *data, int count, MPI_Datatype dataType,
             int source, int tag, MPI_Comm comm, MPI_Request *request)
  ```

  → **request:** on success set to the handle of the non-blocking operation

- **Blocking wait for completion:**
  
  ```c
  MPI_Wait (MPI_Request *request, MPI_Status *status)
  ```

  → **request:** handle for an active non-blocking operation freed and set to `MPI_REQUEST_NULL` upon successful return

  → **status:** status of the completed operation
## Communication-Computation Overlay

### Blocking send (w/o buffering) and receive calls:

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program</strong></td>
<td><strong>Program</strong></td>
</tr>
<tr>
<td><strong>MPI_Send</strong></td>
<td><strong>MPI_Recv</strong></td>
</tr>
<tr>
<td><strong>First message part</strong></td>
<td><strong>Intermediate message part</strong></td>
</tr>
<tr>
<td><strong>Intermediate message part</strong></td>
<td><strong>Intermediate message part</strong></td>
</tr>
<tr>
<td><strong>Last message part</strong></td>
<td><strong>Data must not be used</strong></td>
</tr>
</tbody>
</table>

- Data must remain constant
- Data must not be used

---

Message Passing with MPI (PPCES 2019)
Joachim Protze / Marc-André Hermanns | IT Center der RWTH Aachen University
Communication-Computation Overlay

- Equivalent with non-blocking calls:

  Sender
  
  **Program**
  
  `MPI_Isend`
  
  `MPI_Wait`
  
  **Magic**
  
  **First message part**
  
  **Intermediate message part**
  
  **Last message part**

  Data must remain constant

  Receiver
  
  **Program**
  
  `MPI_Irecv`
  
  `MPI_Wait`

  Data must not be used
Communication-Computation Overlay

Other work can be done in between*:

Sender

Program

MPI_Isend

work

MPI_Wait

Receiver

Program

MPI_Irecv

work

MPI_Wait

Data must remain constant

Data must not be used

First message part

Intermediate message part

Intermediate message part

Last message part
Non-blocking operations can be used to prevent deadlocks in symmetric code:

That is how MPI_Sendrecv is usually implemented
Non-Blocking Request Testing

Test if given operation has completed:

MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)

- **flag**: true if the operation has completed, otherwise false
- **status**: status of the completed operation, only set if flag is true

- Can be (and usually is) called repeatedly inside a loop
- Upon completion of the operation (i.e. when flag is true), the operation is freed and the request handle is set to MPI_REQUEST_NULL

If called with a null request (MPI_REQUEST_NULL):

- MPI_Wait returns immediately with an empty status
- MPI_Test sets flag to true and returns an empty status
Send Modes

- **Standard mode**
  - The call blocks until the message has *either* been transferred *or* copied to an internal buffer for later delivery.

- **Synchronous mode**
  - The call blocks until a matching receive has been posted and the message reception has started.

- **Buffered mode**
  - The call blocks until the message has been copied to a user-supplied buffer. Actual transmission may happen at a later point.

- **Ready mode (don’t use!)**
  - The operation succeeds *only if a matching receive has already been posted*. Behaves as standard send in every other aspect.
MPI Lifecycle Management

- MPI can only be **initialised once** and **finalised once** for the lifetime of each MPI process
  - Multiple calls to `MPI_Init` or `MPI_Finalize` result in error

- **Determine if MPI is already initialised:**
  - `MPI_Init` (int *flag)
  - **flag** set to **true** if `MPI_Init` was called

- **Determine if MPI is already finalised:**
  - `MPI_Finalize` (int *flag)
  - **flag** set to **true** if `MPI_Finalize` was called

- Intended for use in parallel libraries built on top of MPI
Common Pitfalls – C/C++

- Do not pass pointers to pointers in MPI calls

```c
int scalar;
MPI_Send(&scalar, MPI_INT, 1, ...

int array[5];
MPI_Send(array, MPI_INT, 5, ...
... or ...
MPI_Send(&array[0], MPI_INT, 5, ...

int *pointer = new int[5];
MPI_Send(pointer, MPI_INT, 5, ...
... or ...
MPI_Send(&pointer[0], MPI_INT, 5, ...

// ERRONEOUS
MPI_Send(&pointer, MPI_INT, 5, ...
```

- **&array** will work too, but is not recommended
- Will result in the value of the pointer itself (i.e. the memory address) being sent, possibly accessing past allocated memory
Common Pitfalls – C/C++

- Do not pass pointers to pointers in MPI calls

```c
void func (int scalar)
{
    MPI_Send(&scalar, MPI_INT, 1, ...)
}

void func (int& scalar)
{
    MPI_Send(&scalar, MPI_INT, 1, ...)
}

void func (int *scalar)
{
    MPI_Send(scalar, MPI_INT, 1, ...)
}

void func (int *array)
{
    MPI_Send(array, MPI_INT, 5, ...)
    ... or ...
    MPI_Send(&array[0], MPI_INT, 5, ...)
}
Common Pitfalls – C/C++

- Use flat multidimensional arrays; **arrays of pointers do not work**

```c
// Static arrays are OK
int mat2d[10][10];
MPI_Send(&mat2d, MPI_INT, 10*10, ...)

// Flat dynamic arrays are OK
int *flat2d = new int[10*10];
MPI_Send(flat2d, MPI_INT, 10*10, ...)

// DOES NOT WORK
int **p2d[10] = new int*[10];
for (int i = 0; i < 10; i++)
    p2d[i] = new int[10];
MPI_Send(p2d, MPI_INT, 10*10, ...)
... or ...
MPI_Send(&p2d[0][0], MPI_INT, 10*10, ...)
```

MPI has no way to know that there is a hierarchy of pointers
Common Pitfalls – C/C++

- **Passing pointer values around makes little to no sense**
  - Pointer values are process-specific
  - No guarantee that memory allocations are made at the same addresses in different processes
    - Especially on heterogeneous architectures, e.g., host + co-processor
  - No guarantee that processes are laid out in memory the same way, even when they run on the same host
    - Address space layout randomisation
    - Stack and heap protection

- **Relative pointers could be passed around**
Non-contiguous array sections should not be passed to non-blocking MPI calls

```
INTEGER, DIMENSION(10,10) :: mat

! Probably OK
CALL MPI_Isend(mat(:,1:3), ...

! NOT OK
CALL MPI_Isend(mat(1:3,:), ...

! NOT OK
CALL MPI_Isend(mat(1:3,1:3), ...
```

A temporary contiguous array is created and passed to MPI. It might get destroyed on return from the call before the actual send is complete!

Solved in MPI-3.0 with the introduction of the new Fortran 2008 interface `mpi_f08`, which allows array sections to be passed
MPI Part 1
Lab time: Exercise 3

Message Passing with MPI (PPCES 2019)
Joachim Protze / Marc-André Hermanns | IT Center der RWTH Aachen University

(cc) by dongpeiyen1000
CC BY-NC-SA 3.0
Collective Communication
Collective Operations

- **Involve all ranks in a given communicator**
  - Create a smaller communicator for collective communication in a subgroup

- **All ranks must call the same MPI operation to succeed**
  - There should be only **one** call per MPI rank (i.e. not per thread)

- **Process synchronization behaviour is implementation specific**
  - The MPI standard may allow for early return on some ranks

- **Implement common group-communication patterns**
  - Usually tuned to deliver the best system performance
  - Do not reinvent the wheel!
Barrier Synchronisation

- The only explicit synchronisation operation in MPI:

\[
\text{MPI\_Barrier (MPI\_Comm comm)}
\]

\[
\begin{align*}
\max(t_S,0; t_S,1; t_S,2) & < \min(t_E,0; t_E,1; t_E,2) \\
\end{align*}
\]
Barrier Synchronisation

- Useful for benchmarking
  - Always synchronise before taking time measurements

→ Huge discrepancy between the actual work time and the measurement
Barrier Synchronisation

- **Useful for benchmarking**
  - Always synchronise before taking time measurements

Elapsed time as measured by the first rank

- Dispersion of the barrier exit times may occur, but is usually quite low
Data Replication (Broadcast)

- Replicate data from one rank to all other ranks:

  ```c
  MPI_Bcast (void *data, int count, MPI_Datatype dtype,
             int root, MPI_Comm comm)
  ```

  - **data**: data to be sent at **root**; place to put the data in all other ranks
  - **count**: number of data elements
  - **dtype**: elements’ datatype
  - **root**: source rank; **all ranks** must specify the same value
  - **comm**: communicator

- **Notes:**
  - in all ranks but **root**, **data** is an output argument
  - in rank **root**, **data** is an input argument
  - Type signatures must match across all ranks (→ User Datatypes)
Replicate data from one rank to all other ranks:

```c
MPI_Bcast (void *data, int count, MPI_Datatype dtype,
           int root, MPI_Comm comm)
```
Data Replication (Broadcast)

- Replicate data from one rank to all other ranks:

\[
\text{MPI\_Bcast (void *data, int count, MPI\_Datatype dtype, int root, MPI\_Comm comm)}
\]

→ example use:

```c
int ival;

if (rank == 0)
    ival = read_int_from_user();

MPI_Bcast(&ival, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// WRONG
if (rank == 0) {
    ival = read_int_from_user();
    MPI_Bcast(&ival, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
}
// The other ranks do not call MPI\_Bcast \rightarrow Deadlock
```
Data Replication (Broadcast)

- Naïve implementation:

```c
void broadcast (void *data, int count, MPI_Type dtype, int root, MPI_Comm comm)
{
    int rank, nprocs, i;

    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &nprocs);
    if (rank == root) {
        for (i = 0; i < nprocs; i++)
            if (i != root)
                MPI_Send(data, count, dtype, i, TAG_BCAST, comm);
    } else
        MPI_Recv(data, count, dtype, root, TAG_BCAST, comm, MPI_STATUS_IGNORE);
}
```
Data Scatter

Distribute chunks of data from one rank to all ranks:

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

- **sendbuf**: data to be distributed
- **sendcount**: size of each chunk in data elements
- **sendtype**: source datatype
- **recvbuf**: buffer for data reception
- **recvcount**: number of elements to receive
- **recvtype**: receive datatype
- **root**: source rank
- **comm**: communicator

Significant at root rank only
Data Scatter

- **Distribute chunks of data from one rank to all ranks:**

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

- **Notes:**
  - `sendbuf` must be large enough in order to supply `sendcount` elements of data to each rank in the communicator.
  - Data chunks are taken in increasing order following the receiver’s rank.
  - `root` also sends one data chunk to itself.
  - Type signatures of must match across all ranks (→ Datatypes)
Data Scatter

- Distribute chunks of data from one rank to all ranks:

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

- Distribute chunks of data from one rank to all ranks:

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

  - **sendbuf** is only accessed in the root rank
  - **recvbuf** is written into in all ranks
  - **example use:**

    ```c
    // Assume there are 10 MPI ranks
    int bigdata[100];
    int localdata[10];

    MPI_Scatter(bigdata, 10, MPI_INT,       // send buffer, root only
                localdata, 10, MPI_INT,       // receive buffer
                0, MPI_COMM_WORLD);
    ```
Data Gather

- Collect chunks of data from all ranks in one place:

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

- The opposite operation of MPI_Scatter:
  - `recvbuf` must be large enough to hold `recvcount` elements from each rank
  - `root` also receives one data chunk from itself
  - data chunks are stored in increasing order of the sender’s rank
  - for each chunk the receive size must match the amount of data sent
Collect chunks of data from all ranks in one place:

\[
\text{MPI\_Gather (void *sendbuf, int sendcount, MPI\_Datatype sendtype,}
\]
\[
\text{void *recvbuf, int recvcount, MPI\_Datatype recvtype,}
\]
\[
\text{int root, MPI\_Comm comm)}
\]
Gather-to-All

Collect chunks of data from all ranks in all ranks:

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

Note:

→ no root rank – all ranks receive a copy of the gathered data
→ each rank also receives one data chunk from itself
→ data chunks are stored in increasing order of sender’s rank
→ Type signatures of must match across all ranks (→ Datatypes)
→ equivalent to MPI_Gather + MPI_Bcast, but possibly more efficient
Gather-to-All

- Collect chunks of data from all ranks in all ranks:

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

- **Combined scatter and gather operation:**

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

- **Notes:**
  - each rank distributes its `sendbuf` to every rank in the communicator (including itself)
  - data chunks are taken in increasing order of the receiver’s rank
  - data chunks are stored in increasing order of the sender’s rank
  - almost equivalent to **MPI_Scatter + MPI_Gather**
    (one cannot mix data from separate collective operations)
**Combined scatter and gather operation:**

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

**Note: a kind of global chunked transpose**
Lab time: Exercise 4 a) – d)
Global Reduction

- Perform an arithmetic reduction operation while gathering data

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

- `sendbuf`: data to be reduced
- `recvbuf`: location for the result(s) (significant at root only)
- `count`: number of data elements
- `datatype`: element datatype
- `op`: handle of the reduction operation
- `root`: destination rank
- `comm`: communicator

- Result is computed in- or out-of-order depending on the operation:
  - All predefined operations are associative and commutative
  - Beware of non-commutative effects on floats
Global reduction

Element-wise and cross-rank operation

\[ \text{rbuf}[i] = \text{sbuf}_0[i] \circ \text{sbuf}_1[i] \circ \text{sbuf}_2[i] \circ \ldots \circ \text{sbuf}_{\text{nranks}-1}[i] \]

<table>
<thead>
<tr>
<th>sbuf_0[]</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
</tr>
<tr>
<td>sbuf_1[]</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
</tr>
<tr>
<td>sbuf_2[]</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
</tr>
<tr>
<td>sbuf_3[]</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
</tr>
<tr>
<td>rbuf[]</td>
<td>58</td>
<td>62</td>
<td>66</td>
<td>70</td>
<td>74</td>
<td>78</td>
<td>82</td>
<td>86</td>
<td>90</td>
</tr>
</tbody>
</table>

\( \circ = \text{MPI\_SUM} \)
Global Reduction

- Some predefined operation handles:

<table>
<thead>
<tr>
<th>MPI_Op</th>
<th>Result value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum value</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum value</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum of all values</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product of all values</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND of all values</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bit-wise AND of all values</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR of all values</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Users can create their own reduction operations, but that goes beyond the scope of the course.
Global Reduction

- Perform an arithmetic reduction and broadcast the result:
  
  ```c
  MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  ```

- **Notes:**
  
  - every rank receives the result of the reduction operation
  - equivalent to `MPI_Reduce + MPI_Bcast` with the same root
  - can be slower with non-commutative operations because of the forced in-order execution (the same applies to `MPI_Reduce`)
    
    - concerns non-commutative user-defined operations only
Summary: Collective Operations

- All ranks in the communicator must call the MPI collective operation for it to complete successfully:
  - both data sources (root) and data receivers have to make the same call and supply the same value for the root rank where needed
  - observe the significance of each argument
- The sequence of collective calls must be the same in all ranks
- MPI_Barrier is the only explicitly synchronising MPI collective
  - Some may synchronize implicitly (e.g., Allgather, Allreduce)
- Point-to-point and collective communication are independent of each other on the same communicator.
Advantages of Collective Operations

- Collective operations implement common SPMD patterns portably
- Platform/Vendor-specific implementation, but standard behaviour

Example: Broadcast

→ Naïve: root sends separate message to every other rank, $O(#\text{ranks})$
→ Smart: tree-based hierarchical communication, $O(\log(#\text{ranks}))$
→ Genius: pipelined segmented transport, $O(1)$
Communicator Basics
Communication Contexts

- Defines context for each communication operation in MPI
  - Group of participating peers (process group)
  - Error handlers for communication and I/O operations
  - Local key/value cache
  - Virtual topology information (optional)

- Two predefined communicators:
  - `MPI_COMM_WORLD`
    - contains all processes launched initially as part of the MPI program
  - `MPI_COMM_SELF`
    - contains only the current process
Communicators

- Communicator – process group – ranks
Query Operations

- Obtain the size of the process group of a given communicator:

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

  → ranks in the group are numbered from 0 to size-1

- Obtain the rank of the calling process in the given communicator:

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

- Special “null” rank – MPI_PROC_NULL
  
  → member of any communicator

  → can be sent messages to – results in a no-op

  → can be received messages from – zero-size message tagged MPI_ANY_TAG

  → use it to write symmetric code and handle process boundaries
Message Envelope Matching

- **Recall: message envelope**

<table>
<thead>
<tr>
<th></th>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Implicit</td>
<td>Explicit, wildcard possible (MPI_ANY_SOURCE)</td>
</tr>
<tr>
<td>Destination</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Tag</td>
<td>Explicit</td>
<td>Explicit, wildcard possible (MPI_ANY_TAG)</td>
</tr>
<tr>
<td>Communicator</td>
<td>Explicit</td>
<td>Explicit</td>
</tr>
</tbody>
</table>

- **Cross-communicator messaging is not possible**
  
  → messages sent in one communicator can only be received by ranks in the same communicator
  
  → communicators can be used to isolate communication to prevent interference and tag clashes – useful when writing parallel libraries
Communicator creation

- **Duplicate an existing communicator**
  - $\text{MPI\_Comm\_dup}$, $\text{MPI\_Comm\_dup\_with\_info}$, $\text{MPI\_Comm\_idup}$

- **Create new communicator for a subgroup of a communicator**
  - $\text{MPI\_Comm\_create}$, $\text{MPI\_Comm\_create\_group}$

- **Split an existing communicator**
  - $\text{MPI\_Comm\_split}$, $\text{MPI\_Comm\_split\_type}$
Communicator duplication

- Duplicate a given communicator:

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

  - New communication context with same ranks and ordering
  - Easy isolation of encapsulated communication
  - Libraries should never communicate on MPI_COMM_WORLD directly
Destroying Communicators

- Communicators take up memory and other precious resources
- Should be freed once no longer needed

```
MPI_Comm_free (MPI_Comm *comm)
```

→ Marks `comm` for deletion

→ `comm` is set to `MPI_COMM_NULL` on return

→ The actual communicator object is only deleted once all pending operations are completed

- It is erroneous to free predefined communicators `MPI_COMM_WORLD`, `MPI_COMM_SELF` or `MPI_COMM_NULL`
Communicators