Parallel Programming with MPI

Basic and Advanced Features of the Message Passing Interface
Introduction

• Aim of this session
  – Overview on most important MPI Feature sets
    • Focusing on 20% out of 300+ MPI functions
    • Covering hopefully 80% of your use-cases
  – Knowing where to start and where to find documentation
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
Overview

• MPI, the Message Passing Interface,
  – is a standardized comprehensive library interface specification
  – targeted primarily at High Performance Computing (HPC).

• In addition to the classical message passing model, various extensions apart from message passing have been added to the MPI standard, e.g.
  – Dynamic Process Creation
  – Remote Memory Access (RMA)
  – Parallel I/O (MPI I/O)
Overview

• The MPI standard is defined through
  – an open process
  – by a community of
    • Parallel computing vendors
    • Computer scientists and
    • Application developers

• The main advantages the MPI standard brings are portability and ease-of-use.
Overview

- Multiple implementations of MPI exist.
  - The most predominant implementations are
    - OpenMPI (https://www.open-mpi.org/)
    - MPICH (http://www.mpich.org/)
  - Vendor-specific implementations are often based on one of these two.
History of MPI

• 1994:
  – MPI 1.0 was defined, covering
    • Point-to-Point communications
    • Collective operations
    • Process topologies (Cartesian and general graph)

• 1995:
  – MPI 1.1 brings minor changes.

• 1997:
  – MPI 1.2 brings additional clarifications and corrections
  – MPI 2.0 defines
    • Interoperability with threads
    • Dynamic process creation
    • Remote Memory Access (RMA)

• 2008:
  – MPI 1.3 (now MPI-1) brings further minor changes.
  – MPI 2.1 combines MPI 2.0 and MPI 1.3 in one single document.

• 2009:
  – MPI 2.2 (now MPI-2) adds
    • In-Place option for collective routines
    • Deprecation of some functions
    • Deprecation of the C++ language binding

• 2012
  – MPI 3.0 is a major update and introduces
    • Non-blocking collective communications
    • Neighborhood collective communications
    • Fortran 2008 bindings
    • Removal of some deprecated functions
    • Removal of C++ language binding

• 2015
  – MPI 3.1 (now MPI-3) is a minor update, mostly
    • Corrections and clarifications.
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
MPI Basics

Semantic Terms and Basic Usage
MPI Execution

• MPI execution
  – Several processes communicating with each other

  – **Typical:** All Processes execute the same program but operate on different data (SPMD), e.g.
    > mpirun -n 5 ./myprogram

  – **Also possible:** Processes executing different programs (MPMD), e.g.
    > mpirun -n 1 ./master : -n 4 ./worker
(Intra-)Communicator

- **Communicator** *(also Intra-Communicator)*
  - A subset of processes
    - Processes within a communicator can communicate with each other
  - A context for communication
    - Messages within different communicators will never be mixed up

- Predefined Communicators
  - **MPI_COMM_WORLD** *(Containing all processes)*
  - **MPI_COMM_SELF** *(Containing only the current process)*
Inter-Communicator

• **Inter-Communicator**
  
  – A pair of disjoint subsets of processes forming two groups, say $A$ and $B$.

  – Processes of group $A$ can communicate with processes of group $B$ and vice versa.

  • But not for communication within group $A$ or $B$. 
Process Rank

• Rank
  – The unique numerical id of a process within a Communicator
  
  – Strictly speaking “rank” is always with respect to a given Communicator, however often “rank” is used synonymous with “rank within MPI_COMM_WORLD”.

03/10/2016

PRACE Autumn School 2016
Using MPI in programs

- C/C++
  
  `#include <mpi.h>`

- Fortran 2008 + TS 29113 and later (recommended):
  
  `USE mpi_f08`

- Earlier Fortran versions
  
  `USE mpi`
Compiling MPI Programs

- Most MPI Implementations provide wrapper compilers for convenience, e.g.
  - C:
    - mpicc, mpiicc
  - C++:
    - mpicxx, mpiCC, mpic++, mpiicpc
  - Fortran:
    - mpifc, mpif77, mpif90, mpiifort
MPI Language Bindings

• **C binding**
  – All MPI functions return an error code (int)
    • MPI_SUCCESS on success or otherwise implementation-defined.
  – Function Naming Convention: `MPI_Xxx_xxx`

• **Fortran binding**
  – All MPI Operations are procedures
    • The last argument is always the error code:
      `INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

In the following, we will stick to C binding and C examples.
MPI Initialization

• **Single-Threaded execution**
  
  ```c
  int MPI_Init(int *argc, char ***argv)
  ```

• **Multi-Threaded execution** (Hybrid MPI programs, e.g. MPI+OpenMP)
  
  ```c
  int MPI_Init_thread(int *argc, char ***argv,
                     int required, int *provided)
  ```

  required may be one of:
  – **MPI_THREAD_SINGLE**: Only one thread will execute.
  – **MPI_THREAD_FUNNELED**: Only main thread makes MPI calls.
  – **MPI_THREAD_SERIALIZED**: Only one threat at a time makes MPI calls.
  – **MPI_THREAD_MULTIPLE**: No restrictions.

  Actually supported threading mode is returned as *provided.
MPI Basics

Hello World Code Example
Hello World Example (1/2)

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

int main(int argc, char* argv[]) {
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello from rank %d of %d!\n", rank, size);

    MPI_Finalize();
    return 0;
}
```
Hello World Example (2/2)

# Compilation
> mpicc -o hello hello.c

# Invocation
> mpirun -n 5 ./hello

# Possible Output
Hello from rank 1 of 5!
Hello from rank 2 of 5!
Hello from rank 4 of 5!
Hello from rank 0 of 5!
Hello from rank 3 of 5!
MPI Basics

Semantic Terms (continued)
Blocking / Non-blocking operations

• **Blocking**
  – Function return indicates completion of associated operation.
  – E.g. MPI_Send, MPI_Recv

• **Non-blocking**
  – Function may return before associated operation has completed.
  – E.g. MPI_Isend, MPI_Irecv
    • Note: “I” stands for “immediate”, i.e. the function will return “immediately”.
  – Resources (e.g. message buffers) passed to function must not be reused until the operation has completed.
  – Non-blocking functions return an MPI_Request handle for querying the completion status of the operation.
Local / Non-local operations

• **Local**
  – The completion of the operation depends only on the local executing process.
  – E.g. MPI_Comm_size, MPI_Comm_rank, MPI_Bsend

• **Non-local**
  – The completion of the operation *may* require the execution of some MPI operation on another process (*may* involve communication).
  – E.g. MPI_Send
Collective / Non-collective operations

• Collective
  – An operation is collective on a communicator $C$, when all processes in $C$ are required to take part in the operation (all processes need to call the same MPI function on $C$).
  – Multiple collective operations on $C$ must be executed in the same order by all members of $C$.
  – E.g. MPI_Bcast, MPI_Gather, MPI_Scatter, etc.

• Non-collective
  – E.g. MPI_Send, MPI_Recv, etc.
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
Point-to-Point Communication

Communication Protocols and Send Modes
Communication Protocols

- **Eager protocol:**
  - Message is sent instantaneously.
  - Message may have to be buffered on the receiver side (if matching receive has not yet been posted).
  - Typically used for small messages.

- **Rendezvous protocol**
  - 3-way handshake:
    - Sender sends “Request to send” control message.
    - Receiver sends “Ready to send” (when matching receive is posted).
    - Sender sends actual message to receiver.
  - Avoids message buffering on the receiver side.
  - Typically used for large messages.
### MPI Send Operations

<table>
<thead>
<tr>
<th></th>
<th>Blocking Variant</th>
<th>Non-Blocking Variant</th>
<th>Protocol</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Send:</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
<td></td>
<td>Non-local</td>
</tr>
<tr>
<td>Synchronous Send:</td>
<td>MPI_Ssend</td>
<td>MPI_Is send</td>
<td>Rendezvous</td>
<td>Non-local</td>
</tr>
<tr>
<td>Ready Send:</td>
<td>MPI_Rsend</td>
<td>MPI_Ir send</td>
<td>Eager *</td>
<td>Non-local</td>
</tr>
<tr>
<td>Buffered Send:</td>
<td>MPI_Bsend</td>
<td>MPI_Ib send</td>
<td></td>
<td>Local</td>
</tr>
</tbody>
</table>

* Not explicitly stated by the standard, but typical for most MPI implementations;
### MPI Send Operations

<table>
<thead>
<tr>
<th></th>
<th>Blocking Variant</th>
<th>Non-Blocking Variant</th>
<th>Protocol</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard Send:</strong></td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
<td></td>
<td>Non-local</td>
</tr>
<tr>
<td><strong>Synchronous Send:</strong></td>
<td>MPI_Ssend</td>
<td>MPI_Isend</td>
<td>Rendezvous</td>
<td>Non-local</td>
</tr>
<tr>
<td><strong>Ready Send:</strong></td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
<td>Eager</td>
<td>Non-local</td>
</tr>
<tr>
<td><strong>Buffered Send:</strong></td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
<td></td>
<td>Local</td>
</tr>
</tbody>
</table>

- **Completion Semantics:**
  - Send buffer can be reused.
  - Message may have been sent or buffered locally (Most MPI implementations use buffering for small messages).
### MPI Send Operations

<table>
<thead>
<tr>
<th></th>
<th>Blocking Variant</th>
<th>Non-Blocking Variant</th>
<th>Protocol</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard Send:</strong></td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
<td>Non-local</td>
<td></td>
</tr>
<tr>
<td><strong>Synchronous Send:</strong></td>
<td>MPI_Ssend</td>
<td>MPI_Isend</td>
<td>Rendezvous</td>
<td>Non-local</td>
</tr>
<tr>
<td><strong>Ready Send:</strong></td>
<td>MPI_Rsend</td>
<td>MPI_Insend</td>
<td>Eager</td>
<td>Non-local</td>
</tr>
<tr>
<td><strong>Buffered Send:</strong></td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
<td>Local</td>
<td></td>
</tr>
</tbody>
</table>

- **Completion Semantics:**
  - A matching receive operation on the remote process has been posted.
  - Message transmission has started (may have finished).
  - Send buffer can be reused.
**MPI Send Operations**

<table>
<thead>
<tr>
<th></th>
<th>Blocking Variant</th>
<th>Non-Blocking Variant</th>
<th>Protocol</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Send:</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
<td></td>
<td>Non-local</td>
</tr>
<tr>
<td>Synchronous Send:</td>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
<td>Rendezvous</td>
<td>Non-local</td>
</tr>
<tr>
<td>Ready Send:</td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
<td>Eager</td>
<td>Non-local</td>
</tr>
<tr>
<td>Buffered Send:</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
<td></td>
<td>Local</td>
</tr>
</tbody>
</table>

- **Prerequisite:**
  - The receiver has to be “ready” (matching receive has been posted).
  - No buffering will occur on the receiver side.
**MPI Send Operations**

<table>
<thead>
<tr>
<th></th>
<th>Blocking Variant</th>
<th>Non-Blocking Variant</th>
<th>Protocol</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Send:</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
<td></td>
<td>Non-local</td>
</tr>
<tr>
<td>Synchronous Send:</td>
<td>MPI_Ssend</td>
<td>MPI_IsSend</td>
<td>Rendezvous</td>
<td>Non-local</td>
</tr>
<tr>
<td>Ready Send:</td>
<td>MPI_Rsend</td>
<td>MPI_IsSend</td>
<td>Eager</td>
<td>Non-local</td>
</tr>
<tr>
<td>Buffered Send:</td>
<td>MPI_Bsend</td>
<td>MPI_IbSend</td>
<td></td>
<td>Local</td>
</tr>
</tbody>
</table>

- **Completion Semantics:**
  - Same as Standard send.
  - Send buffer can be reused.
  - Message has been sent (typical case) or buffered.
# MPI Send Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Blocking Variant</th>
<th>Non-Blocking Variant</th>
<th>Protocol</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Send:</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
<td></td>
<td>Non-local</td>
</tr>
<tr>
<td>Synchronous Send:</td>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
<td>Rendezvous</td>
<td>Non-local</td>
</tr>
<tr>
<td>Ready Send:</td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
<td>Eager</td>
<td>Non-local</td>
</tr>
<tr>
<td>Buffered Send:</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
<td></td>
<td>Local</td>
</tr>
</tbody>
</table>

- **Completion Semantics:**
  - Send buffer can be reused.
  - Message may have been sent (if matching receive operation has been posted) or buffered.
### MPI Send Operations

<table>
<thead>
<tr>
<th></th>
<th>Blocking Variant</th>
<th>Non-Blocking Variant</th>
<th>Protocol</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Send:</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
<td></td>
<td>Non-local</td>
</tr>
<tr>
<td>Synchronous Send:</td>
<td>MPI_Ssend</td>
<td>MPI_IsSsend</td>
<td>Rendezvous</td>
<td>Non-local</td>
</tr>
<tr>
<td>Ready Send:</td>
<td>MPI_Rsend</td>
<td>MPI_IsRsend</td>
<td>Eager</td>
<td>Non-local</td>
</tr>
<tr>
<td>Buffered Send:</td>
<td>MPI_Bsend</td>
<td>MPI_IsBsend</td>
<td></td>
<td>Local</td>
</tr>
</tbody>
</table>

- **Advice:**
  - **Standard Send** is most appropriate in most cases
    - Most freedom for choosing best protocol and whether to use buffering or not.
  - **Synchronous send** can be helpful for debugging (deadlock detection).
MPI Receive Operations

• Much simpler than Send:

<table>
<thead>
<tr>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Recv</td>
<td>MPI_Irecv</td>
</tr>
</tbody>
</table>

• Any pairing of the different send and receive operations is legal.
Point-to-Point Communication

Message Envelope and Message Matching
Message Envelope

- **Message Envelope**
  - *C*: Communicator
  - *src*: Source rank of sending process w.r.t. *C*
  - *dst*: Destination rank of receiving process w.r.t. *C*
  - *tag*: Message Tag (arbitrary integer between 0 and at least $2^{15}-1 = 32767$)
  - *type*: Type signature (sequence of basic data types occurring in the message)
  - *len*: Length of the message

- **For send operations:**
  - (*C, dst, tag, type, len*) are specified explicitly via arguments
  - *src* is given implicitly (rank of the local process w.r.t. *C*)

- **For receive operations:**
  - (*C, src, tag, type, len*) are specified explicitly (wildcards MPI_ANY_SOURCE and MPI_ANY_TAG allowed)
  - *dst* is given implicitly (rank of local process w.r.t. *C*)
Message Matching

• Send operation $S$
  – Envelope $E_1$
  – Message $M$

• Receive operation $R$
  – Envelope $E_2$

• The operation $R$ will receive the message $M$ if and only if:
  – $E_1.C = E_2.C$ (communicators match)
  – $E_1.dst = E_2.dst$ (message goes to specified destination)
  – $E_1.src = E_2.src$ or $E_2.src = MPI\_ANY\_SOURCE$ (message comes from specified source)
  – $E_1.tag = E_2.tag$ or $E_2.tag = MPI\_ANY\_TAG$ (message has correct tag)

• Required for correctness (Runtime-checks can typically be turned on/off):
  – $E_1.type \approx E_2.type$ (sequence of basic types in the message is as expected, more on this later)
  – $E_1.len \leq E_2.len$ (message may be shorter then specified, but not longer)
Point-to-Point Communication

A simple example: 1D Halo exchange
1D Halo-Exchange Example
Variant 1 (1/2)

```c
int rank, size;
int data[3] = {-1, -1, -1};

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

data[1] = rank;
exchange_left();
exchange_right();

    printf("Data of rank %d: {%d %d %d}\n", rank, data[0], data[1], data[2]);

    MPI_Finalize();
    return 0;
}
```
void exchange_left() {
    int left = rank-1 >= 0 ? rank-1 : MPI_PROC_NULL;
    MPI_Send(&data[1], 1, MPI_INT, left, 0, MPI_COMM_WORLD);
    MPI_Recv(&data[0], 1, MPI_INT, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

void exchange_right() {
    int right = rank+1 < size ? rank+1 : MPI_PROC_NULL;
    MPI_Send(&data[1], 1, MPI_INT, right, 0, MPI_COMM_WORLD);
    MPI_Recv(&data[2], 1, MPI_INT, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
What is the (biggest) problem here?

```
void exchange_left() {
    int left = rank-1 >= 0 ? rank-1 : MPI_PROC_NULL;
    MPI_Send(&data[1], 1, MPI_INT, left, 0, MPI_COMM_WORLD);
    MPI_Recv(&data[0], 1, MPI_INT, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

void exchange_right() {
    int right = rank+1 < size ? rank+1 : MPI_PROC_NULL;
    MPI_Send(&data[1], 1, MPI_INT, right, 0, MPI_COMM_WORLD);
    MPI_Recv(&data[2], 1, MPI_INT, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```
1D Halo-Exchange Example

Variant 1 - Problems

• Deadlock
  – When **rendezvous protocol** is used for sending, all processes hang in first **MPI_Send** operation (no matching **MPI_Recv** at this stage).
  – **Debugging Tip**: Replacing **MPI_Send** with **MPI_Ssend** (“synchronous send”) can help for detecting deadlocks (forces rendezvous protocol).

• Serialization
  – When eager protocol is used for sending, Process $N$ cannot proceed until Process $N-1$ sends data (**exchange_left** is always done first).
1D Halo-Exchange Example
Fixing the Deadlock

- **Solution 1:**
  - Using `MPI_Sendrecv`
    - Combined `MPI_Send` and `MPI_Recv`, but user does not need to worry on correct order.

- **Solution 2:**
  - Using Non-blocking communication;
    - Discussed later (Variant 2)
1D Halo-Exchange Example
Fixing the Deadlock

```c
void exchange_left() {
    int left = rank-1 >= 0 ? rank-1 : MPI_PROC_NULL;
    MPI_Sendrecv(
        &data[1], 1, MPI_INT, left, 0,
        &data[0], 1, MPI_INT, left, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );
}

void exchange_right() {
    int right = rank+1 < size ? rank+1 : MPI_PROC_NULL;
    MPI_Sendrecv(
        &data[1], 1, MPI_INT, right, 0,
        &data[2], 1, MPI_INT, right, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );
}
```
1D Halo-Exchange Example
Fixing Serialization

• Solution 1
  – Processes with even rank communicate first with (1) right neighbor, then with (2) left neighbor.
  – Processes with odd rank do it the other way round.
1D Halo-Exchange Example

Fixing Serialization – Solution 1

```c
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    data[1] = rank;

    if(rank % 1 == 0) {
        exchange_right();
        exchange_left();
    } else {
        exchange_left();
        exchange_right();
    }

    printf("Data of rank %d: {%d %d %d}\n", rank, data[0], data[1], data[2]);
    ...
```
1D Halo-Exchange Example
Fixing Serialization – Solution 2

• **Solution 2** (more elegant)
  – Let data “flow” in one direction (e.g. right)
  – And then let data “flow” in other direction.
1D Halo-Exchange Example
Fixing Serialization – Solution 2

```c
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    data[1] = rank;
    shift_left();
    shift_right();

    printf("Data of rank %d: { %d %d %d}\n", rank, data[0], data[1], data[2]);

    MPI_Finalize();
    return 0;
}
```
1D Halo-Exchange Example
Fixing Serialization – Solution 2

```c
void shift_left() {
    int left = ..., right = ...;
    MPI_Sendrecv(
        &data[1], 1, MPI_INT, left, 0,
        &data[2], 1, MPI_INT, right, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );
}

void shift_right() {
    ...
    MPI_Sendrecv(
        &data[1], 1, MPI_INT, right, 0,
        &data[0], 1, MPI_INT, left, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );
}
```

03/10/2016
1D Halo-Exchange Example
Variant 2 (1/2)

```c
int rank, size;
int data[3] = {-1, -1, -1};

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    data[1] = rank;

    exchange();

    printf("Data of rank %d: {%d %d %d}\n", rank, data[0], data[1], data[2]);

    MPI_Finalize();
    return 0;
}
```
void exchange() {
    int left = rank - 1 >= 0 ? rank - 1 : MPI_PROC_NULL;
    int right = rank + 1 < size ? rank + 1 : MPI_PROC_NULL;
    MPI_Request requests[4];

    MPI_Isend(&data[1], 1, MPI_INT, left, 0, MPI_COMM_WORLD, &requests[0]);
    MPI_Isend(&data[1], 1, MPI_INT, right, 0, MPI_COMM_WORLD, &requests[1]);
    MPI_Irecv(&data[0], 1, MPI_INT, left, 0, MPI_COMM_WORLD, &requests[2]);
    MPI_Irecv(&data[2], 1, MPI_INT, right, 0, MPI_COMM_WORLD, &requests[3]);

    MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
}
Point-to-Point Communication

Receiving Messages of unknown length
Receiving messages of known maximum length

```c
#define MAXLEN 15

int recvbuf[MAXLEN];
int actual_length;

void receive() {
    MPI_Status status;

    MPI_Recv(recvbuf, MAXLEN, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_INT, &actual_length);

    printf("Recieved %d integers from process 0.\n", actual_length);
}
```
Receiving messages of arbitrary length

```
int *recvbuf;
int length;

void receive() {
  MPI_Status status;
  MPI_Message message;

  // Blocking check for message, do not yet receive
  MPI_Mprobe(0, 0, MPI_COMM_WORLD, &message, &status);

  // Get message length, allocate buffer and receive
  MPI_Get_count(&status, MPI_INT, &length);
  recvbuf = malloc(length * sizeof(int));
  MPI_Mrecv(recvbuf, length, MPI_INT, &message, &status);

  printf("Recieved %d integers from process 0.\n", length);
}
```
Point-to-Point Communication

Summary and Advice
Summary

- **Standard Send and Receive** (typical case):
  - Use `MPI_Send / MPI_Isend` for sending
  - Use `MPI_Recv / MPI_Irecv` for receiving
  - Alternatively also `MPI_Sendrecv` (there is no `MPI_Isendrecv`).
    - There is also `MPI_Sendrecv_replace` (in case send and receive buffer are the same)

- **Other Send Modes**
  - **Synchronous Send** forces rendezvous protocol (3-way handshake).
  - **Ready Send** avoids handshake, receiver has to be ready.
  - **Buffered Send** forces sender-side buffering (local operation).
Summary

• **Request Completion** (non-blocking operations)
  – Single request:
    • MPI_Wait / MPI_Test: Wait is blocking; Test is non-blocking
  – Array of requests:
    • MPI_Waitall / MPI_Testall: Check for completion of all requests
    • MPI_Waitany / MPI_Testany: Check for next completed request (exactly one)
    • MPI_Waitsome / MPI_Testsome: Check for next completed requests (maybe more than one at once)

• **Messages of unknown length**
  – MPI_Mprobe / MPI_Improbe:
    • Check for a pending message without receiving
    • Returns message handle and status object
  – MPI_Get_count: Get message length from status object
  – MPI_Mrecv / MPI_Imrecv: Receive message (by message handle)
Advice

• **Standard Send** is sufficient in most cases
  – i.e. `MPI_Send` or `MPI_Isend` (paired with `MPI_Recv` or `MPI_Irecv`)
  – Gives MPI most freedom in choosing protocol and adequate buffering

• **Synchronous send** can be helpful for debugging (deadlock detection).

• **Prefer higher-level communication constructs** to tedious point-to-point communication whenever possible, e.g.
  – Collective operations
    • on `MPI_COMM_WORLD` or tailored intra- or inter-communicators
  – MPI-3 Sparse collective operations,
    • on Cartesian Communicators or MPI-3 Distributed Graph communicator.
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
Remote Memory Access

a.k.a. One-sided communications
RMA – Basic Workflow

• Create **Memory window** object for remote access
  – Makes a given memory region (buffer) usable for RMA

• Use **MPI_Win_fence** for synchronization

• Use **MPI_Put / MPI_Get / MPI_Accumulate** for accessing remote memory.
RMA – Memory Window Creation

- **MPI_Win_create:**
  - Create Memory window from given (user-allocated) buffer.

- **MPI_Win_allocate:**
  - Create Memory window with memory allocated by MPI.

- **MPI_Win_free:**
  - Release Memory window
  - When window was created with MPI_Win_allocate also associated buffer is released.
RMA - Synchronization

- **Active target synchronization (collective)**
  - Use `MPI_Win_fence` to separate distinct “access epochs”

- Access epochs:
  - Local reads and writes (from/to local buffer)
  - Local and remote reads (from local buffer)
  - Remote writes or updates (to local buffer) – “exposure epoch”
  - Also `MPI_Put` and `MPI_Accumulate` to the same remote memory region cannot be intermixed within one access epoch.
RMA - Synchronization

• Active target synchronization (advanced)
  – Reduce synchronization overhead by specifying explicitly participating processes (non-collective synchronization)
    – MPI_Win_post / MPI_Win_wait
      • Start / end exposure epoch (target process)
    – MPI_Win_start / MPI_Win_complete
      • Start / end remote access epoch (origin process)
RMA - Synchronization

• Passive target synchronization
  – MPI_Win_lock / MPI_Win_lock_all
    • Acquire shared lock (for reads) or exclusive lock (for writes) for accessing remote memory at specified rank (MPI_Win_lock) or all participating ranks (MPI_Win_lock)
  – MPI_Win_unlock / MPI_Win_unlock_all
    • Release shared or exclusive lock after accessing remote memory.
RMA Operations

- **Supported operations:**
  - **Get:** read from remote memory.
  - **Put:** write to remote memory.
  - **Accumulate:** update remote memory with reduction operation.
  - **Get-and-accumulate:** get old value and update.
  - **Compare-and-swap:** get old value and overwrite only on successful comparison.

- **Non-blocking ("Request-based")** variants available
  - Only usable with passive target synchronization.
RMA Operations

- **MPI_Get / MPI_Rget**
  - Blocking / non-blocking ("Request based") read from remote memory.

- **MPI_Put / MPI_Rput**
  - Blocking / non-blocking write to remote memory.

- **MPI_Accumulate / MPI_Raccumulate**
  - Blocking / non-blocking update of remote memory.
  - All predefined reduction operations can be used.
    - MPI_MAX, MPI_MIN, MPI_SUM, MPI_LAND, etc.
    - See also MPI 3.1 Standard, Section 5.9.2
  - Additionally MPI_REPLACE can be used.
RMA Operations

- **MPI_Get_accumulate / MPI_Rget_accumulate**
  - Blocking / Non-blocking update of remote memory, returning original value.

- **MPI_Fetch_and_op**
  - Similar to **MPI_Get_accumulate**
    - Less generic (fewer arguments), potentially faster (hardware-support)

- **MPI_Compare_and_swap**
  - Atomic compare-and-swap of remote memory
    - If current value in target buffer equals given value, it is overwritten.
    - Original value in target buffer is returned.
Remote Memory Access

Summary and Advice
Summary

- Remote Memory Access (RMA) is versatile and powerful (but hard to get right).

- RMA can bring performance advantages in comparison to Point-to-Point communications.

- Three synchronization modes provided
  - MPI_Win_fence
    - Active target synchronization (collective)
  - MPI_Win_[post|wait|start|complete]
    - Active target synchronization (fine-grained)
  - MPI_Win_[un]lock[_all]
    - Passive target synchronization (shared or exclusive locks)
Advice

• Using collective active target synchronization, i.e. using `MPI_Win_fence` for separating distinct access epochs is the easiest variant to implement.
  – Still more than enough opportunities to get things wrong (e.g. race conditions)

• **Performance**
  – Pass `MPI_Info` dictionary when creating the memory window
    • E.g. when *not* using passive target synchronization, pass "no_locks": "true"
    • See also MPI Standard 3.1, Section 11.2.1
  – Use assertions (additional hints for MPI) when using synchronization calls
    • See also MPI Standard 3.1, Section 11.5.5

• **Good luck!**
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
MPI Datatypes

Motivation and Definition
MPI Datatypes

- **MPI Datatypes**
  - A way to describe the exact data layout of send and receive buffers.

- **Data layout**
  - Location and type of primitive data elements in buffer

- **Primitive data elements**
  - E.g. integers, single/double precision numbers, characters, etc.
MPI Datatypes

**Main advantage**
- MPI operations can directly operate on application data structures.
  - No manual buffer packing and unpacking by the user

**Further advantages**
- Runtime type checking (to some degree)
  - Error when data elements in received message are not as expected.
- Data-conversion
  - E.g. conversion from little to big endian
  - This aspect is in fact more relevant in the context of MPI-IO
MPI Datatypes

- **Type Signature**
  - Data layout of a message
  - Given by a *sequence of primitive datatypes*

- **MPI Datatype**
  - Data layout of a memory buffer
  - Given by
    - *Sequence of types* of primitive data elements in buffer (= Type Signature)
    - *Sequence of displacements* of data elements w.r.t. buffer start address
MPI Datatypes – Type Matching

• Let \( S = (s_1, s_2, s_3, \ldots) \) be the Type Signature of a message \( M \),
  – E.g. \( S = (\text{int}, \text{int}, \text{double}) \)

• Let \( T = (t_1, t_2, t_2, \ldots) \) be the Type Signature of the MPI Datatype used in a receive operation \( R \),
  – E.g. \( T = (\text{int}, \text{int}, \text{double}, \text{int}) \)

• For successful receipt of \( M \) by \( R \), \( S \) must be a prefix of \( T \)

• In other words:
  – The **primitive data types have to match** one-to-one.
  – The **actual message might be shorter** than the receive buffer.
MPI Datatypes

Predefined and Derived Types
## Predefined (primitive) Datatypes

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Corresponding C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char</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_BYTE</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

For a full list, see [http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node48.htm](http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node48.htm)
Creating Derived Datatypes

1) Use one or more “Type Constructors”

   MPI_Type_contiguous, MPI_Type_vector, MPI_Type_struct, etc.

2) Commit final type
   (those ones used in communications or I/O)

   MPI_Type_commit

3) Free all types
   (The freed type may still be in use by pending send and receive operations or in other derived types)

   MPI_Type_free
MPI Datatypes

Type Constructors

(A selection)
Contiguous Type (Array)

```c
int MPI_Type_contiguous(
    int count, MPI_Datatype oldtype,
    MPI_Datatype *newtype
)
```
Vector Type

```
int MPI_Type_vector(
   int count, int blocklength, int stride,
   MPI_Datatype oldtype, MPI_Datatype *newtype
)
```
Indexed Type (1)

```c
int MPI_Type_indexed(
    int count,
    const int blocklengths[], const int displacements[],
    MPI_Datatype oldtype, MPI_Datatype *newtype
)
```

count: 3
blocklengths: {1, 2, 1}
displacements: {0, 2, 6}

oldtype

newtype

0 1 2 3 4 5 6
Indexed Type (2)

```c
int MPI_Type_create_indexed_block(
    int count,
    int blocklength, const int displacements[],
    MPI_Datatype oldtype, MPI_Datatype *newtype
)
```

count: 4
blocklength: 1
displacements: {0, 2, 3, 6}
Structured Type

```c
int MPI_Type_create_struct(
    int count,
    const int blocklengths[], const MPI_Aint displacements[],
    const MPI_Datatype types[], MPI_Datatype *newtype
)
```

- Count: 3
- Blocklengths: {2, 1, 3}
- Displacements: {0, 8, 16}
- Types: {type1, type2, type3}

**Diagram:**
- Newtype: 8 bytes + 8 bytes
Subarray Type

```c
int MPI_Type_create_subarray(
    int ndims,
    const int sizes[], const int subsizes[], const int starts[],
    int order,
    MPI_Datatype oldtype, MPI_Datatype *newtype
)
```

oldtype

```
oldtype buffer[4][5];
```

```
ndims: 2
sizes: {5, 4}
subsizes: {2, 3}
starts: {2, 1}
order: MPI_ORDER_FORTRAN
```
Subarray Type

```c
int MPI_Type_create_subarray(
    int ndims,
    const int sizes[], const int subsizes[], const int starts[],
    int order,
    MPI_Datatype oldtype, MPI_Datatype *newtype
)
```

oldtype

newtype

```
oldtype buffer[5][4];
```

ndims: 2
sizes: {5, 4}
subsizes: {2, 3}
starts: {2, 1}
order: MPI_ORDER_C
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
Collective Communication

Motivation and Overview
Motivation

• Thinking in terms of global communication patterns (of groups of processes) is in general easier than thinking in terms of Point-to-Point messages (which message has to go where?).

• Advantages of Collective Communications (compared to Point-to-Point)
  – Performance
    • MPI can choose the best from multiple communication algorithms
  – Maintainability
    • Code is often easier to read and understand
Overview

• MPI defines **9 types of collective operations:**
  
  - Barrier
  - Broadcast
  - Gather
  - All-Gather
  - Scatter
  - All-to-all
  - Reduce
  - All-Reduce
  - Reduce-Scatter
  - Scan

• **New in MPI-3:**
  
  - Non-blocking variants for all collective operations.
Collective Communication

Collective Operation Semantics
Barrier

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

- **Synchronizes** all processes within the Communicator `comm`.

- **Completion Semantics**
  - All processes in `comm` have entered the barrier.
    - MPI makes no guarantees on how long it will take other processes to leave the barrier.
    - MPI Barriers are not appropriate for highly accurate time measurements.
Broadcast

```c
int MPI_Bcast(
    void* buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm
)
```
Gather

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

sendcount = recvcount: 1
sendtype = recvtype: ?
root: 2
All-Gather

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

![Diagram showing the All-Gather operation between processes]

- Proc. 0: A
- Proc. 1: B
- Proc. 2: C
- Proc. 3: D

After the operation:
- Proc. 0: A B C D
- Proc. 1: A B C D
- Proc. 2: A B C D
- Proc. 3: A B C D

03/10/2016  PRACE Autumn School 2016
int MPI_Scatter(
    const void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm
)

sendcount = recvcount: 1
sendtype = recvtype: ?
root: 2
All-to-all

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

<table>
<thead>
<tr>
<th>Proc. 0</th>
<th>Proc. 1</th>
<th>Proc. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>G</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>H</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>I</td>
</tr>
</tbody>
</table>

03/10/2016 PRACE Autumn School 2016
Reduce

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

<table>
<thead>
<tr>
<th>Proc. 0</th>
<th>Proc. 1</th>
<th>Proc. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proc. 0</th>
<th>Proc. 1</th>
<th>Proc. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>F</td>
</tr>
</tbody>
</table>

```plaintext
count: 2
op: MPI_SUM
root: 2
```
Reduce

• Supported Reduction operations:
  – Predefined
    • MPI_MIN, MPI_MAX, MPI_SUM, MPI_LAND, etc.
    • Full list see MPI 3.1 Standard, Section 5.9.2
    • [http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node112.htm](http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node112.htm)

  – User-defined
    • Use MPI_Op_create (register callback function) and MPI_Op_free.
All-Reduce

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

<table>
<thead>
<tr>
<th>Proc. 0</th>
<th>Proc. 1</th>
<th>Proc. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>F</td>
</tr>
</tbody>
</table>

**Before**

**After**

<table>
<thead>
<tr>
<th>Proc. 0</th>
<th>Proc. 1</th>
<th>Proc. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+C+E</td>
<td>B+D+F</td>
<td>A+C+E</td>
</tr>
<tr>
<td>A+C+E</td>
<td>B+D+F</td>
<td>A+C+E</td>
</tr>
<tr>
<td>A+C+E</td>
<td>B+D+F</td>
<td>A+C+E</td>
</tr>
<tr>
<td>A+C+E</td>
<td>B+D+F</td>
<td>A+C+E</td>
</tr>
</tbody>
</table>
Reduce-Scatter

\[
\text{int MPI\_Reduce\_scatter\_block(}\\ 
\text{const void* sendbuf, void* recvbuf,}\\ 
\text{int recvcount, MPI\_Datatype datatype,}\\ 
\text{MPI\_Op op, MPI\_Comm comm})
\]

recvcount: 2
sendcount: 2*\text{nprocs} = 6
op: MPI\_SUM
Scan (inclusive)

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

- **Proc. 0**:
  - Send buffer: A
  - Receive buffer: A

- **Proc. 1**:
  - Send buffer: C
  - Receive buffer: A+C

- **Proc. 2**:
  - Send buffer: E
  - Receive buffer: A+C+E

**Count**: 2

**Op**: MPI_SUM
Scan (exclusive)

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

count: 2
op: MPI_SUM

<table>
<thead>
<tr>
<th>Proc. 0</th>
<th>Proc. 1</th>
<th>Proc. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>F</td>
</tr>
</tbody>
</table>

A + C
B + D
Collective Communication

Variants
Variants

• For all operations where it makes sense, variants for **varying send/receive counts** (varying block-lengths) are defined.
  – For all-to-all, even a variant with **varying send/receive types** is provided.

• For many operations, **MPI_IN_PLACE** (predefined constant) can be used as send buffer argument
  – Send data (read from receive buffer) is overwritten “in-place”.

• For all operations, **non-blocking variants** are available.
  – Following the already known naming scheme:
    • Prefix “I” for “immediate” return, e.g. **MPI_Ibcast**
  – Checking for completion:
    • **MPI_Request** as output argument
    • Same *Wait* and *Test* routines as for non-blocking P2P
## Variants

<table>
<thead>
<tr>
<th>Fixed Block-length</th>
<th>Varying Block-lengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Gather</td>
<td>MPI_Gather\textsuperscript{v}</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>MPI_Allgather\textsuperscript{v}</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>MPI_Scatter\textsuperscript{v}</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td>MPI_Alltoall\textsuperscript{v}</td>
</tr>
<tr>
<td>MPI_Reduce_scatter_block</td>
<td>MPI_Reduce_scatter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fixed Block-length &amp; Type</th>
<th>Varying Block-lengths and Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Alltoall</td>
<td>MPI_Alltoall\textsuperscript{w}</td>
</tr>
</tbody>
</table>
Collective Communication

Correctness Issues
Correctness Issues (1/2)

• Amount of data sent **must exactly match** amount of data specified by the receiver.
  – More strict than in P2P communications, where actual message may be shorter than receive buffer

• Collective operations have to be issued in the **same order** by all participating processes.

• Except for **MPI_Barrier**, users should **not rely on any synchronization effects** other than given by data-dependencies.
Correctness Issues (2/2)

• **Multiple active non-blocking** collective operations are **allowed**.
  – E.g. `MPI_Ibcast – MPI_Ibcast – MPI_Waitall` is correct.

• **Blocking and non-blocking** collective operations **do not match**.
  – E.g. `MPI_Ibcast` on process A will not match `MPI_Bcast` on process B.

• **Blocking and non-blocking** collective operations **can be interleaved**.
  – E.g. `MPI_Ibarrier – MPI_Bcast – MPI_Wait` is correct.
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
Communicator Creation

Basic Communicator Constructors
Communicator Creation

- **MPI_Comm_dup**
  - Creates a new Communicator containing the same processes with unchanged ordering of ranks but provides a different context for message transfer.

- **MPI_Comm_split**
  - Partitions a given Communicator into subgroups of processes.
  - Useful for collective operations on subgroups of processes.
Duplication

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

color: rank < 2 ? 0 : 1
key: 0

Partition
Partition

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

<table>
<thead>
<tr>
<th>color: rank &lt; 2 ? 0 : 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>key: -rank</td>
</tr>
</tbody>
</table>

Diagram:

- **comm**: 0 1 1 0 2 2 3 1 4 0
- **new_comm (1)**: 0 1
- **new_comm (2)**: 2 3 4
Inter-Communicator

```c
int MPI_Intercomm_create(
    MPI_Comm local_comm, int local_leader,
    MPI_Comm peer_comm, int remote_leader,
    int tag, MPI_Comm *intercomm
)
```

![Diagram of inter-communicator](image)
Inter-Communicator

• For exact semantics of **collective operations on Inter-Communicators**, see
  – MPI 3.1 Standard, Section 6.6
  – or [http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node166.htm](http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node166.htm)
Communicator Creation

Process Topologies
Process Topologies

• Equipping Communicators with additional “virtual topology” (neighborhood relations between processes)

• Advantages
  – Certain communication patterns can be “naturally” expressed in terms of Neighborhood collective operations
  – MPI can select a good embedding of the virtual topology to physical machine (if user allows reordering of ranks)
Cartesian Communicator

```c
int MPI_Cart_create(MPI_Comm comm,
    int ndims, const int dims[], const int periods[],
    int reorder, MPI_Comm *newcomm)
```

- `ndims`: 2
- `dims`: `{3, 2}`
- `reorder`: 0 (false)
Cartesian Communicator

```c
int MPI_Cart_create(MPI_Comm comm,
    int ndims, const int dims[], const int periods[],
    int reorder, MPI_Comm *newcomm
)
```

- **ndims**: 2
- **dims**: `{3, 2}`
- **reorder**: 1 (true)
Convenience functions

• **MPI_Dims_create**:
  – Helps selecting a balanced distribution of processes per coordinate direction
    • Inputs: nnodes, ndims, constraints (optional)

• **MPI_Cart_rank**:
  – Get rank by coordinates

• **MPI_Cart_coords**:
  – Get coordinates by rank

• **MPI_Cart_shift**:
  – Compute source and destination ranks for **MPI_Sendrecv**
    • Inputs: index of dimension, shift offset
Graph Communicator

```c
int MPI_Dist_graph_create_adjacent(MPI_Comm comm,
int indegree, const int srcs[], const int srcweights[],
int outdegree, const int dsts[], const int dstweights[],
MPI_Info info, int reorder, MPI_Comm *newcomm)
```

- **indegree**: 3
  - **srcs**: {0, 2, 4}
  - **srcweights**: MPI_UNWEIGHTED

- **outdegree**: 2
  - **dsts**: {1, 4}
  - **dstweights**: MPI_UNWEIGHTED
Graph Communicator

```c
int MPI_Dist_graph_create_adjacent(MPI_Comm comm,
    int indegree, const int srcs[], const int srcweights[],
    int outdegree, const int dsts[], const int dstweights[],
    MPI_Info info, int reorder, MPI_Comm *newcomm)
```

- **indegree**: 3
  - `srcs`: {0, 2, 4}
  - `srcweights`: {2, 1, 1}

- **outdegree**: 2
  - `dsts`: {1, 4}
  - `dstweights`: {1, 2}
Graph Communicator

• More functions for Creating Graph Topologies
  – See MPI 3.1 Standard, Section 7.5.3
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
Neighborhood Collectives

Gather and All-to-all
Neighborhood Gather

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

Typical case:
- `sendcount = recvcount`
- `sendtype = recvtype`

Notes:
- Length of `sendbuf = sendcount`
- Length of `recvbuf = recvcount * indegree`
Neighborhood All-to-all

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

**Typical case:**
- `sendcount = recvcount`
- `sendtype = recvtype`

**Notes:**
- Length of `sendbuf = sendcount * outdegree`
- Length of `recvbuf = recvcount * indegree`
## Variants

<table>
<thead>
<tr>
<th>Fixed Block-length</th>
<th>Varying Block-lengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Neighbor_allgather</td>
<td>MPI_Neighbor_allgather\text{v}</td>
</tr>
<tr>
<td>MPI_Neighbor_alltoall</td>
<td>MPI_Neighbor_alltoall\text{v}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fixed Block-length &amp; Type</th>
<th>Varying Block-lengths and Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Neighbor_alltoall</td>
<td>MPI_Neighbor_alltoall\text{w}</td>
</tr>
</tbody>
</table>
Variants

• Also **non-blocking Variants** provided

  \[
  \text{MPI}\_\text{Ineighbor}_* \\
  \]

• **Completion checking** and correctness rules same as for “normal” collective communication operations
Collective Communication

Summary and Advice
Summary

• MPI provides powerful high-level abstractions for a large variety of communication patterns.
  – MPI may use heuristics to dynamically choose a good algorithm for given operation.

• Process topologies allow a good mapping of the virtual topology of a given problem to the physical machine.
  – Currently most MPI Implementations do not yet exploit this optimization potential.
  – This will change in future.
Advice

• **Prefer Collective operations** (on Topology-, Intra- or Inter-Communicators) to Point-to-Point communications when possible.
  – In many cases you’ll get better (or at least similar) performance.
  – In some cases however, P2P and RMA are the right choices:
    • E.g. Dynamic Sparse Data Exchange

• **Experiment** with different collective operations
  – Consult current MPI Standard documents for description of the operations
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
Shared Memory Windows

Creation and Access
Shared Memory Window Creation

- **MPI_Comm_split_type**
  - Create a Communicator \( C \) containing all processes “living” on the same node
  - Pass \( \text{split\_type} = \text{MPI\_COMM\_TYPE\_SHARED} \).
    - Other values for \( \text{split\_type} \) may exist (e.g. processes “living” on one socket), but are implementation dependent.

- **MPI_Win_allocate_shared**
  - Create Shared Memory Window on the “intra-node” Communicator \( C \)
  - Can be used for RMA calls
Shared Memory Window Creation

```c
int MPI_Win_allocate_shared(
    MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm,
    void *baseptr, MPI_Win *win
)
```

![Diagram showing the allocation of shared memory windows for different ranks.](image)
Shared Memory Window Creation

```c
int MPI_Win_allocate_shared(
    MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm,
    void *baseptr, MPI_Win *win
)
```

```
info: {
    "alloc_shared_noncontig"="true",
    ...
}
```
Shared Memory Window Access

• 3 Possibilities:
  – Direct address arithmetic based on baseptr
    • Works only if memory is contiguous
  – Use MPI_Win_shared_query for getting base pointers to memory chunks of other ranks.
  – Use MPI_Put / MPI_Get
Shared Memory Windows

Summary and Advice
Summary

• Shared Memory windows can be useful in many scenarios.
  – E.g. large replicated read-only data structures can be stored once per node.
  • Memory saving / Alternative to threading
Advice

• **Allowing for non-contiguous layout** of memory window may improve performance
  – Especially on non-uniform-memory-access (NUMA) machines
    • Local memory chunks can be allocated near to the “owning” process

• Thus consider specifying in `MPI_Info` object the entry
  • "alloc_shared_noncontig"="true"
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
MPI I/O

Just one Remark
MPI I/O

• For full reference see
  – MPI 3.1 Standard, Chapter 13

• Consider also using higher-level libraries for I/O (often based on MPI-I/O)
  – E.g. Parallel HDF5, Parallel NetCDF, etc.
Outline

1. Overview
2. MPI Basics
3. P2P Communication
4. RMA
5. Datatypes
6. Collective Communication
7. Communicators
8. Sparse Collectives
9. Shared Memory
10. Parallel I/O
11. Conclusion
Conclusion

Overall Summary and General Advice
Conclusion

• MPI is much **more than message passing**.
  – Remote and Shared Memory Access
  – Collective operations on Topology-, Intra- or Inter-Communicators
  – Scalable I/O
  – and more...

• **Topics not covered** in this talk (with reference to MPI 3.1 Standard):
  – Persistent communication requests (Section 3.9)
  – Process groups (Chapter 6)
  – MPI Environmental Management, in particular Error Handling (Chapter 8)
  – (Dynamic) Process Creation and Management (Chapter 9)
  – MPI I/O (Chapter 13)
Alternatives to MPI

• **Partitioned Global Address Space** (PGAS) languages and libraries, e.g.
  – Chapel (Cray)
  – Co-Array Fortran
  – UPC (Unified Parallel C)
  – ...

• Promising approaches, might be worth a try.
  – (but MPI is still predominant in HPC and it looks like this won’t change in the nearer future).
General Advice

• **Before rushing into programming with MPI,**
  – Consider using higher-level libraries (built on top of MPI) which potentially could serve your use-case.
  – Examples:
    • libMesh ([http://libmesh.github.io/](http://libmesh.github.io/))
    • And many many many more...
      – E.g. for Linear Algebra: [http://www.netlib.org/utk/people/JackDongarra/la-sw.html](http://www.netlib.org/utk/people/JackDongarra/la-sw.html)

• **Documentation:**
  – Use MPI Standard documents as complementary (if not primary) source.
  – Good books are out there (e.g. see next slide).
References

- **MPI Standard Documents:** [http://mpi-forum.org/docs/docs.html](http://mpi-forum.org/docs/docs.html)
- In particular **MPI-3.1 Standard:** [http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf](http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf)
Advertisement

• Have a look at the PRACE Code Vault:
  – http://www.prace-ri.eu/prace-codevault/
  – Feedback and even Code Contributions are warmly welcome!