An Introduction to Message Passing and Parallel Programming With MPI

Georg Hager, Thomas Zeiser, Gerhard Wellein, Markus Wittmann, Ayesha Afzal (RRZE)
hpc@rrze.fau.de

A. Skjellum, P. Bangalore, S. Herbert, R. Rabenseifner

Message Passing Interface
Introduction to MPI: Agenda

- Blocking point to point communication
- Helper functions
- Nonblocking point to point communication
- Collectives
- Derived data types
- Virtual Topologies
Point-to-Point Communication Blocking
MPI_Send/MPI_Recv

// two process only example
int dst;
if (rank == 0) { dst = 1; } else { dst = 0; }

char * buffer = malloc(count * sizeof(char));

MPI_Send(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD);
MPI_Recv(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);

$ # tested on supermic
$ mpiexec -n 2 ./send 10      # OK
$ mpiexec -n 2 ./send 100     # OK
$ mpiexec -n 2 ./send 1000    # OK
$ mpiexec -n 2 ./send 10000   # OK
$ mpiexec -n 2 ./send 100000  # OK
$ mpiexec -n 2 ./send 1000000 # DEAD LOCK
Sending

Buffered
- Always successful
- Do not care of time of delivery
- Completion does not involve action of other side

Synchronous
- Completion is successful arrival of message
- Completion involves action of other side

MPI_Bsend

MPI_Ssend
Sending Modes

- **Completion**
  - When function call returns (for *blocking* p2p communication)
  - Buffer can safely be reused

<table>
<thead>
<tr>
<th>MPI function</th>
<th>type</th>
<th>completes when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>synchronous or buffered</td>
<td>depends on type</td>
</tr>
<tr>
<td>MPI_Bsend</td>
<td>buffered</td>
<td>buffer has been copied</td>
</tr>
<tr>
<td>MPI_Ssend</td>
<td>synchronous</td>
<td>remote starts receive</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>--</td>
<td>message was received</td>
</tr>
</tbody>
</table>
Point-to-Point Communication:(MPI _Bsend (optional…))

Syntax (C):

\[
\text{MPI\_Bsend}\left(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm}\right)
\]

\textbf{buf: buffer to send}

- **Bsend** completes when message has been copied
- Predictable & no synchronization

User has to attach extra buffer (see next slide)

Problems: comes at the cost of additional copy operations

Only one buffer can be attached to the application at the same time

(C) 2010 - 2019 RRZE, LRZ
Point-to-Point Communication: MPI_Buffer_attach/MPI_Buffer_detach (optional…)

- **Syntax (C):**
  ```c
  MPI_Buffer_attach(void * buffer, int size);
  buffer: address of buffer
  size: buffer size in bytes
  ```

  ```c
  MPI_Buffer_detach(void ** buffer, int * size);
  buffer: returns addr. of detached buffer, defined as void *, but actually expects void **
  size: returns size of the detached buffer
  ```

- **Fortran:** with mpi module or mpi.h buffer argument is not used, with mpi_f08 module buffer is of type TYPE(C_PTR)

- **Size of buffer = (size of all outstanding BSENDs) + (number of intended BSENDs * MPI_BSEND_OVERHEAD)**

- **Best way to get required size for one message:**
  ```c
  MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int * s)
  size = s + MPI_BSEND_OVERHEAD
  ```
Point-to-Point Communication: MPI_Ssend

Syntax (C):

```
MPI_Ssend(buf, count, datatype, dest, tag, comm)
```

Problems:
- Performance: high latency, risk of serialization
- Source for potential deadlocks

But: should be used for debugging

Ssend completes after message has been accepted by destination

predictable & safe behavior

Blocking synchronization of source and destination
### Possible solutions for deadlock example

#### MPI_Bsend

// two process only example
```c
int dst; if (rank == 0) { dst = 1; } else { dst = 0; }
char * buffer = malloc(count * sizeof(char));
```

// assuming buffer has been attached
```c
MPI_Bsend(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD);
MPI_Recv(buffer, count, MPI_CHAR, dst, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

#### MPI_Ssend

// two process only example
```c
int dst; if (rank == 0) { dst = 1; } else { dst = 0; }
char * buffer = malloc(count * sizeof(char));
```

```c
if (rank == 0) {
    MPI_Ssend(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else if (rank == 1) {
    MPI_Recv(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Ssend(buffer, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
}
```
Point-to-Point Communication
MPI_SENDRECV

Sending/Receiving at the same time is a common use case
- e.g.: shift messages, ring topologies, ghost cell exchange

MPI_Send/MPI_Recv pairs are not reliable:

```c
// Rank left from myself.
left = (rank - 1 + size) % size;
// Rank right from myself.
right = (rank + 1) % size;

MPI_Send(buffer_send, n, MPI_INT, right, 1, MPI_COMM_WORLD);
MPI_Recv(buffer_recv, n, MPI_INT, left, 1, MPI_COMM_WORLD, status);
```

How to avoid potential deadlock?
Point-to-Point Communication

MPI_SENDRECV

- Syntax: simple combination of send and receive arguments:
  ```c
  MPI_Sendrecv(
      buffer_send, sendcount, sendtype, dest, sendtag,
      buffer_recv, recvcount, recvtype, source, recvtag,
      comm, MPI_Status * status)
  ```

- MPI takes care no deadlocks occur

```c
// Rank left from myself.
left = (rank - 1 + size) % size;
// Rank right from myself.
right = (rank + 1) % size;

MPI_Sendrecv(
    buffer_send, n, MPI_INT, right, 0,
    buffer_recv, n, MPI_INT, left, 0, MPI_COMM_WORLD, status);
```

- disjoint send/receive buffers
- can have different count & data type
MPI_Sendrecv matches with simple *send/*recv point-to-point calls

MPI_PROC_NULL as source/destination acts as no-op
• send/recv with MPI_PROC_NULL return as soon as possible buffers are not altered

useful for open chains/non-circular shifts:

```c
// Rank left from myself.
left = rank - 1; if (left < 0) { left = MPI_PROC_NULL; }

// Rank right from myself.
right = rank + 1; if (right >= size) { right = MPI_PROC_NULL; }

MPI_Sendrecv(
    buffer_send, n, MPI_INT, right, 0,  
    buffer_recv, n, MPI_INT, left,  0, MPI_COMM_WORLD, &status);
```
Domain distributed to ranks
here 4 x 3 ranks
each rank gets one tile

Each rank's tile is
surrounded by ghost cells,
representing the cells of the neighbors

After each sweep over a tile perform **ghost cell exchange**, i.e. update ghost cells with new values of neighbor cells

Possible implementation:
1. copy new data into contiguous send buffer
2. send to corresponding neighbor receive new data from same neighbor
3. copy new data into ghost cells

```
MPI_Sendrecv(sb, ..., j, rb, ..., j, ...
```

```
MPI_Sendrecv(sb, ..., i, rb, ..., i, ...
```
When only one single buffer is required:

```c
MPI_Sendrecv_replace(
    buf, count, datatype,
    dest, sendtag,
    source, recvtag,
    comm, MPI_Status * status)
```

MPI ensures no deadlocks occur

```c
// Rank left from myself.
left = (rank - 1 + size) % size;
// Rank right from myself.
right = (rank + 1) % size;

MPI_Sendrecv_replace(
    buf, n, MPI_INT, right, 0, left, 0, MPI_COMM_WORLD, &status);
```
Blocking MPI communication calls
- Operation completes when call returns
- After completion: send/receive buffer can safely be reused

Available Send communication modes:
- Synchronous -- MPI_Ssend:
  - Guarantee receiving has started
  - Performance drawbacks, deadlock dangers
- Buffered -- MPI_Bsend:
  - Completes after buffer is copied
  - User-provided buffer to save messages
  - Additional copy operations
- Standard -- MPI_Send:
  - Behavior can be synchronous or buffered or depending on message length, no guarantee about that
Message order preservation (guaranteed inside a communicator)

same communicator (e.g. MPI_COMM_WORLD)
Useful MPI Calls:

**MPI_GET_PROCESSOR_NAME**

- Return a string to identify the hardware the process is running on
  
  ```c
  MPI_Get_processor_name(char * name, int * rlen);
  ```

- Typically the hostname of the compute node, but any arbitrary string is possible

```c
char name[MPI_MAX_PROCESSOR_NAME];
int rlen;

MPI_Get_processor_name(name, &rlen);
printf("rank %d runs on %s\n", rank, name);
```

```plaintext
# SuperMIC Output from mpiexec -n 2./a.out
rank 0 runs on i01r13a06.
rank 1 runs on i01r13a06.
```
Useful MPI Calls:

**MPI_WTIME**

- Returns seconds since one point in past time

  ```c
  double MPI_Wtime()
  ```

- Use only for computation of time differences

  ```c
  time_start = MPI_Wtime();
  // ...working...
  duration = MPI_Wtime() - time_start
  ```

- Returns time resolution in seconds,

  ```c
  double MPI_Wtick()
  ```

  - e.g. if resolution is 1ms `MPI_Wtick()` returns `1e-3`

- **No ierror** argument in Fortran

- Typically clocks from different ranks are not synchronized
**Useful MPI Calls: MPI_ABORT**

- **MPI_ABORT** forces an MPI program to terminate:

  ```c
  int MPI_Abort(MPI_Comm comm, int errorcode)
  ```

- Aborts all processes in communicator
- `errorcode` will be handed as exit value to calling environment
- Safe and well-defined way of terminating an MPI program (if implemented correctly)

- In general, if something unexpected happens, try to shut down your MPI program the standard way (**MPI_Finalize()**)
Point-to-Point Communication
Nonblocking
Nonblocking Point-to-Point Communication

**Advantages**
- Avoid deadlocks
- Possibility for overlapping communication with useful work
  - Best case: hide communication cost
  - Not guaranteed by the standard

```c
MPI_Request request;
MPI_Status status;

MPI_Isend(
    send_buffer, count, MPI_CHAR, dst, 0,
    MPI_COMM_WORLD, &request);

// do some work...
// do not use send_buffer

MPI_Wait(&request, &status)
```

**Best case scenario**
- Avoid idle time
- Avoid synchronization
Nonblocking communication:
- Return from function != completion
- Each initiated operation must have a matching wait/test!

```c
// two process only example
int dst; if (rank == 0) { dst = 1; } else { dst = 0; }

MPI_Request requests[2];
MPI_Status statuses[2]

MPI_Isend(send_buffer, count, MPI_CHAR, dst, 0,
          MPI_COMM_WORLD, &requests[0]);
MPI_Irecv(recv_buffer, count, MPI_CHAR, dst, 0,
          MPI_COMM_WORLD, &requests[1]);

// do some work...
// using send_buffer/rec_buffer is prohibited

MPI_Waitall(2, requests, statuses)
```

1. start operation
2. obtain request handle (new parameter)
3. wait/test for completion

buffer is only allowed to be reused after completion!
Nonblocking Point-to-Point Communication: MPI_ISEND / MPI_IRECV

- **Standard nonblocking send/receive:**

  ```c
  MPI_Isend(sendbuf, count, datatype, dest, tag, comm, MPI_Request * request)
  ```

  ```c
  MPI_Irecv(recvbuf, count, datatype, source, tag, comm, MPI_Request * request)
  ```

  **request:** variable of type `MPI_Request`, will be associated with the corresponding operation

- **Do not reuse `sendbuf(recvbuf)` before MPI_Isend/MPI_Irecv has been completed**

- **MPI_Irecv has no status argument**
  - obtained later during completion via `MPI_Wait*/MPI_Test*`
Blocking and Nonblocking Point-To-Point Communication

- Blocking `send/recv` can be used with nonblocking ones

- Type synchronous/buffered affects completion
  - Meaning: when `MPI_Wait / MPI_Test` return
  - Not when initiation, i.e. `MPI_I…`, returns

- Nonblocking operation immediately followed by a matching wait is equivalent to the blocking operation
  - Except for some compiler problems (see later slides)
  - Emulate blocking call via nonblocking operation:

    ```c
    MPI_Request request;
    MPI_Status status;

    MPI_Send(buf, ...);
    MPI_Isend(buf, ..., &request);
    MPI_Wait(&request, &status);
    ```
MPI provides two test modes:

- **MPI_Wait**: Wait until the communication has been completed and buffer can safely be reused: Blocking

- **MPI_Test**: Return TRUE (FALSE) if the communication has (not) completed: Nonblocking
Test one communication handle for completion:

```c
MPI_Wait(MPI_Request * request,
         MPI_Status * status);
```

```c
MPI_Test(MPI_Request * request, int * flag,
         MPI_Status * status);
```

- request: request handle of type `MPI_Request`
- status: status object of type `MPI_Status` (cf. `MPI_Recv`)
- flag: variable of type `int` to test for success
Nonblocking Point-to-Point Communication:
Test for Completion

MPI_Wait

MPI_Request request;
MPI_Status status;

MPI_Isend(
    send_buffer, count, MPI_CHAR,
dst, 0, MPI_COMM_WORLD, &request);

// do some work...
// do not use send_buffer

MPI_Wait(&request, &status)

// use send_buffer

MPI_Test

MPI_Request request;
MPI_Status status;
int flag;

MPI_Isend(
    send_buffer, count, MPI_CHAR,
dst, 0, MPI_COMM_WORLD, &request);

do {
    // do some work...
    // do not use send_buffer
    MPI_Test(&request, &flag, &status);
} while (!flag);

// use send_buffer
Nonblocking Point-to-Point Communication: Test for Completion

- MPI can handle multiple communication requests
- \textbf{Wait/Test} for completion of \textit{multiple} requests:
  \begin{verbatim}
  MPI_Waitall(int count, MPI_Request requests[],
              MPI_Status statuses[]);
  
  MPI_Testall(int count, MPI_Request requests[],
              int *flag, MPI_Status statuses[]);
  \end{verbatim}

- \textbf{Waits for/Tests if all} provided requests have been completed

  \begin{verbatim}
  MPI_Request requests[2];
  MPI_Status statuses[2];

  MPI_Isend(send_buffer, ..., &requests[0]);
  MPI_Irecv(recv_buffer, ..., &requests[1]);
  // do some work...
  MPI_Waitall(2, requests, statuses)
  // Isend & Irecv have been completed
  \end{verbatim}
ghost cell exchange, with nonblocking send/recv with all neighbors at once

**Possible implementation:**
1. Copy new data into contiguous send buffers
2. Start nonblocking receives/sends from/to corresponding neighbors
3. Wait with MPI_Waitall for all obtained requests to complete
4. Copy new data into ghost cells
Nonblocking Point-to-Point Communication: Test for Completion

- Wait/Test for completion of multiple requests:
  
  ```c
  MPI_Waitany(int count, MPI_Request requests[],
              int * idx, MPI_Status * status);
  
  MPI_Testany(int count, MPI_Request requests[],
              int * idx, int * flag,
              MPI_Status * status);
  ```

- Waits for/Tests if one request has been completed
  
  ```c
  MPI_Request requests[2];
  MPI_Status status;
  int finished = 0;

  MPI_Isend(send_buffer, ..., &(requests[0]));
  MPI_Irecv(recv_buffer, ..., &(requests[1]));
  do {
    // do some work...
    MPI_Testany(2, requests, &idx, &flag, &status);
    if (flag) { ++finished; }
  } while (finished < 2)
  ```

- completed requests are automatically set to MPI_REQUEST_NULL
- completed request requests[idx]
Nonblocking Point-to-Point Communication: Pitfalls due to compiler optimization

- **Fortran:**

  ```fortran
  MPI_Irecv(recvbuf, ..., request, ierror)
  MPI_Wait(request, status, ierror)
  write (*,*)) recvbuf
  ```

  may be compiled as

  ```fortran
  MPI_Irecv(recvbuf, ..., request, ierror)
  registerA = recvbuf
  MPI_Wait(request, status, ierror)
  write (*,*)) registerA
  ```

  i.e. old data is written instead of received data!

- **Workarounds:**
  - `recvbuf` may be allocated in a common block, or
  - calling `MPI_Get_Address(recvbuf, iaddr_dummy, ierror)` after `MPI_Wait`

  MPI might modify `recvbuf` after function return, but compiler is not aware of this.
Nonblocking Point-to-Point Communication and strided sub-arrays

- Fortran:
  ```fortran
  MPI_ISEND(buf(7,:,:,:), ..., request, ierror)
  ! other work
  MPI_WAIT(request, status, ierror)
  ```

- Do not use non-contiguous sub-arrays in nonblocking calls!
- Use first sub-array element: `buf(1,1,9)` instead of whole sub-array: `buf(:, :, 9:13)`
- Call by reference necessary
- Call by in-and-out-copy forbidden

- Specified array is non-contiguous
- Compiler generates a temporary array for the function all
- Temp. array is destroyed after MPI_ISEND returns

Data is send in this time frame, but source array is already lost.
Collective Communication in MPI
Operations including all ranks of a communicator

**ALL RANKS MUST CALL THE FUNCTION**

- Blocking calls: buffer can be reused after return
- Nonblocking calls with MPI-3.0: buffer can be used after completion
  
  ```
  MPI_Wait*/MPI_Test*
  ```
- May or may not synchronize the processes
- Cannot interfere with point-to-point communication
- Data type matching
- No tags
- Sent message must fill receive buffer (count is exact)
- Typically MPI libraries provide optimized implementations for operations

**Types:**
- Synchronization (barrier)
- Data movement (broadcast, scatter, gather, all to all)
- Collective computation (reduction)
Collective Communication
Synchronization

- Explicit synchronization of all ranks from specified communicator

  \texttt{MPI	extunderscore Barrier(comm)}

- Ranks only return from call after every rank has called the function

- \texttt{MPI	extunderscore Barrier} rarely needed, most of the time for debugging, e.g. to make sure every rank has reached a certain point in the application
Collective Communication

Broadcast

- send buffer from one to all ranks

\[
\text{MPI\_Bcast}(buf, \text{count}, \text{datatype}, \text{int root}, \text{comm})
\]

**root:** rank from which data should be taken, typically 0, but everyone is allowed

```
buffer
count = 3

MPI\_Bcast(buffer, 3, MPI\_INT, 1, MPI\_COMM\_WORLD)
```

```
buffer
1 2 3
```

(C) 2010 - 2019 RRZE, LRZ
Collective Communication
Scatter

- Send the \(i\)th chunk to the \(i\)th rank

\[
\text{MPI\_Scatter}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \\
\quad \text{recvbuf}, \text{recvcount}, \text{recvtype}, \\
\quad \text{root}, \text{comm})
\]

- In general \(\text{sendcount} = \text{recvcount}\)
- \text{sendbuf} is ignored on non-root ranks

<table>
<thead>
<tr>
<th>rank</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>recvbuf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{MPI\_Scatter}(\text{sendbuf}, 1, \text{MPI\_INT}, \text{recvbuf}, 1, \text{MPI\_INT}, \\
\quad \text{root}, \text{MPI\_COMM\_WORLD})
\]

| sendbuf | 1 | 2 | 3 | 4 |
| recvbuf | 1 | 2 | 3 | 4 |
Collective Communication

**Gather**

- Receive a from each rank and place \( i \text{th} \) rank’s msg at \( i \text{th} \) position in receive buffer

\[
\text{MPI\_Gather} (\text{sendbuf}, \text{sendcount}, \text{sendtype}, \\
\quad \text{recvbuf}, \text{recvcount}, \text{recvtype}, \\
\quad \text{root}, \text{comm})
\]

- In general \( \text{sendcount} = \text{recvcount} \)
- \( \text{recvbuf} \) is ignored on non-root ranks

<table>
<thead>
<tr>
<th>rank</th>
<th>0</th>
<th>1 (root)</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>recvbuf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{MPI\_Gather} (\text{sendbuf}, 1, \text{MPI\_INT}, \text{recvbuf}, 1, \text{MPI\_INT}, \\
\quad \text{root}, \text{MPI\_COMM\_WORLD})
\]

| sendbuf | 1 | 2 | 3 | 4 |
| recvbuf | 1 | 2 | 3 | 4 |
Send chunks of different sizes to different ranks

\[
\text{MPI\_Scatterv}(\text{sendbuf}, \text{int sendcounts[]}[], \text{int displs[]}[], \text{sendtype}, \text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{root}, \text{comm})
\]

**sendcounts**: array specifying the number of elements to send to each rank: send \(\text{sendcounts}[i]\) elements to rank \(i\)

**displs**: integer array specifying the displacements in \text{sendbuf} from which to take the outgoing data to each rank, specified in number of elements
### Collective Communication

#### Scatterv Example (optional …)

<table>
<thead>
<tr>
<th>rank</th>
<th>sendbuf</th>
<th>sendcounts</th>
<th>displs</th>
<th>recvbuf</th>
<th>recvcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>5 4 1 0</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1 2 3 4</td>
<td>2 1 3 1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5 6 7</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**MPI_Scatterv() with root = 1**

<table>
<thead>
<tr>
<th>recvbuf</th>
<th>recvcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 7</td>
<td>2 3 4 1</td>
</tr>
</tbody>
</table>
Collective Communication
Gatherv (optional …)

- Receive segments of different sizes from different ranks

```c
MPI_Gatherv(
    sendbuf, sendcount, sendtype,
    recvbuf, int recvcounts[], int displs[], recvtype, 
    root, comm)
```

- **recvcounts**: array specifying the number of elements to receive from each rank:
  - receive `recvcounts[i]` elements from rank `i`

- **displs**: integer array specifying the displacements where received data from specific rank is put in `recvbuf`, in units of elements:
Gather data from all ranks and broadcast it

\[
\text{MPI\_Allgather}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{comm})
\]

- In general \(\text{sendcount} = \text{recvcount}\)
- Also available: \text{MPI\_Allgatherv} (cf. \text{MPI\_Gatherv})

- No \text{MPI\_Allscatter}

- MPI library has more possibilities for optimization than manual gather/bcast:

\[
\text{MPI\_Gather()}\text{ with root} = i
\]
\[
\text{MPI\_Bcast()}\text{ with root} = i
\]
**MPI_Allgather**: Gather data from all ranks and broadcast it

<table>
<thead>
<tr>
<th>rank</th>
<th>sendbuf</th>
<th>sendcount</th>
<th>recvbuf</th>
<th>recvcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MPI_Allgather()** *(no root required)*

<table>
<thead>
<tr>
<th>recvbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>
**MPI_Alltoall:** For all ranks, send ith chunk to ith rank

\[ \text{MPI\_Alltoall}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{comm}) \]

**MPI\_Alltoallv:** Allows different number of elements to be send/received by each rank

**MPI\_Alltoallw:** Allows also different data types and displacements in bytes
**MPI_Alltoall**: For all ranks, send ith chunk to ith rank

<table>
<thead>
<tr>
<th>rank</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>0 1 2 3</td>
<td>4 5 6 7</td>
<td>8 9 10 11</td>
<td>12 13 14 15</td>
</tr>
<tr>
<td>sendcount</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>recvbuf</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recvcount</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

`MPI_Alltoall()` *(no root required)*

| recvbuf | 0 4 8 12 | 1 5 9 13 | 2 6 10 14 | 3 7 11 15 |
Global Operations
Syntax

- Compute results over distributed data

\[
\text{MPI\_Reduce}(\text{sendbuf}, \text{recvbuf}, \text{count}, \text{datatype}, \text{MPI\_Op op}, \text{root}, \text{comm})
\]

- Result in \text{recvbuf} only on \text{root} process available
- Perform operation on all \text{count} elements of an array with \text{count} > 1
- If all ranks require result use \text{MPI\_Allreduce}
- If the 12 predefined ops are not enough use \text{MPI\_Op\_create}/\text{MPI\_Op\_free} to create own ones

```
rank | sendbuf
-----|--------
  0   | 0 9 2 6
  1   | 5 1 0 4
  2   | 8 3 4 5
  3   | 1 0 6 8
```

\[
\text{MPI\_Reduce()}\quad \text{count} = 4
\text{op} = \text{MPI\_MAX}
\]

```
recvbuf on root
\text{8 9 6 8}
```
### Global Operations

#### Predefined Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>MPI_BAND</td>
<td>Bit-AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
<td>MPI_BOR</td>
<td>Bit-OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical XOR</td>
<td>MPI_BXOR</td>
<td>Bit-XOR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum+ Position</td>
<td>MPI_MINLOC</td>
<td>Minimum+ Position</td>
</tr>
</tbody>
</table>

- Define own operations with `MPI_Op_create/MPI_Op_free`
- MPI assumes that the operations are associative
- Be careful with floating point operations, as they may be not associative because of rounding errors
Avoid local copy operations, e.g. from send to receive buffers:

### Gather

```c
int value = ...;
MPI_Gather(&value, 1, MPI_INT,
            recv_buf, 1, MPI_INT,
            root, comm);
```

```c
int value = ...;
if (rank == root) {
    recv_buf[root] = value;
    MPI_Gather(MPI_IN_PLACE, 1, MPI_INT,
               recv_buf, 1, MPI_INT,
               root, comm);
}
else {
    MPI_Gather(&value, 1, MPI_INT,
               recv_buf, 1, MPI_INT,
               root, comm);
}
```

### Allgather

```c
int value = ...;
MPI_AllGather(&value, 1, MPI_INT,
              recv_buf, 1, MPI_INT,
              root, comm);
```

```c
int value = ...;
recv_buf[rank] = value;
MPI_Allgather(MPI_IN_PLACE, 1, MPI_INT,
              recv_buf, 1, MPI_INT,
              root, comm);
```

requires distinction between root and non-root ranks
Extended Collective Operations:
“In place” Buffer Specification

Override input buffer with result

**Reduce**

```c
int partial_sum = ..., total_sum;
MPI_Reduce(&partial_sum, &total_sum,
    1, MPI_INT,
    MPI_SUM, root, comm);
```

```c
int partial_sum = ..., total_sum;
if (rank == root) {
    total_sum = partial_sum;
    MPI_Reduce(MPI_IN_PLACE, &total_sum,
        1, MPI_INT,
        MPI_SUM, root, comm);
}
else {
    MPI_Reduce(&partial_sum, &total_sum,
        1, MPI_INT,
        MPI_SUM, root, comm);
}
```

**Allreduce**

```c
int partial_sum = ..., total_sum;
MPI_AllReduce(&partial_sum, &total_sum,
    1, MPI_INT,
    MPI_SUM, comm);
```

```c
int partial_sum = ..., total_sum;
total_sum = partial_sum;
MPI_AllReduce(MPI_IN_PLACE, &total_sum,
    1, MPI_INT,
    MPI_SUM, comm);
```
## MPI_IN_PLACE Cheat Sheet

<table>
<thead>
<tr>
<th>Function</th>
<th>MPI_IN_PLACE argument</th>
<th>At which rank(s)</th>
<th>Comment [MPI 3.0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_GATHER</td>
<td>send buffer</td>
<td>root</td>
<td>Recv value at root already in the correct place in receive buffer.</td>
</tr>
<tr>
<td>MPI_GATHERV</td>
<td>send buffer</td>
<td>root</td>
<td>Recv value at root already in the correct place in receive buffer.</td>
</tr>
<tr>
<td>MPI_SCATTER</td>
<td>receive buffer</td>
<td>root</td>
<td>Root-th segment of send buffer is not moved.</td>
</tr>
<tr>
<td>MPI_SCATTERV</td>
<td>receive buffer</td>
<td>root</td>
<td>Root-th segment of send buffer is not moved.</td>
</tr>
<tr>
<td>MPI_ALLGATHER</td>
<td>send buffer</td>
<td>all</td>
<td>Input data at the correct place were process would receive its own contribution.</td>
</tr>
<tr>
<td>MPI_ALLGATHERV</td>
<td>send buffer</td>
<td>all</td>
<td>Input data at the correct place were process would receive its own contribution.</td>
</tr>
<tr>
<td>MPI_ALLTOALL</td>
<td>send buffer</td>
<td>all</td>
<td>Data to be send is taken from receive buffer and replaced by received data, data send/received must be of the same type map specified in receive count/receive type.</td>
</tr>
<tr>
<td>MPI_ALLTOALLV</td>
<td>send buffer</td>
<td>all</td>
<td>Data to be send is taken from receive buffer and replaced by received data. Data send/received must be of the same type map specified in receive count/receive type. The same amount of data and data type is exchanged between two processes.</td>
</tr>
<tr>
<td>MPI_REDUCE</td>
<td>send buffer</td>
<td>root</td>
<td>Data taken from receive buffer, replaced with output data.</td>
</tr>
<tr>
<td>MPI_ALLREDUCE</td>
<td>send buffer</td>
<td>all</td>
<td>Data taken from receive buffer, replaced with output data.</td>
</tr>
</tbody>
</table>
Derived Data Types in MPI
Derived Datatypes in MPI: Why Do We Need Them?

Root reads configuration and broadcasts it to all others

```c
// root: read configuration from file into struct config
MPI_Bcast(&cfg.nx, 1, MPI_INT, ...);
MPI_Bcast(&cfg.ny, 1, MPI_INT, ...);
MPI_Bcast(&cfg.du, 1, MPI_DOUBLE, ...);
MPI_Bcast(&cfg.it, 1, MPI_INT, ...);
```

Send column of matrix (noncontiguous in C):
- Send each element alone?
- Manually copy elements out into a contiguous buffer and send it?

MPI_Bcast(&cfg, sizeof(cfg), MPI_BYTE, ...)

is not a solution. It's not portable as no data conversion can take place.
Derived Data Types in MPI: Construction

- Create in three steps

- Generate with

  \[ \text{MPI\_Type}^* \]

- Commit new data type with

  \[ \text{MPI\_Type\_commit} (\text{MPI\_Datatype}^* \ nt) \]

- After use, deallocate the data type with

  \[ \text{MPI\_Type\_free} (\text{MPI\_Datatype}^* \ nt) \]
Derived Data Types in MPI:
MPI_TYPE_VECTOR

- Create vector-like data type

\[
\text{MPI\_Type\_vector}(\text{int } \text{count}, \text{ int } \text{blocklength}, \text{ int } \text{stride}, \\
\quad \text{MPI\_Datatype } \text{oldtype}, \ \\
\quad \text{MPI\_Datatype} * \text{newtype})
\]

- **count**: 2 (no. of blocks)
- **blocklength**: 3 (no. of elements in each block)
- **stride**: 5 (no. of elements b/w start of each block)
- **oldtype**: MPI_INT

\[
\text{MPI\_Datatype } nt; \\
\text{MPI\_Type\_vector}(2, 3, 5, \text{MPI\_INT}, &nt); \\
\text{MPI\_Type\_commit}(nt); \\
// use nt...
\]

\[
\text{MPI\_Type\_free}(nt);
\]

**Caution:** Concatenating such types in a SEND operation can lead to unexpected results!

See Sec. 3.12.3 and 3.12.5 of the MPI 1.1 Standard for details.
count argument to send and others must be handled with care:

`MPI_Send(buf, 2, nt, ...)` with `nt` (newtype from prev. slide)
- Get the total size (in bytes) of datatype in a message

```c
int MPI_type_size(MPI_Datatype newtype, int *size)
```

- Get the lower bound and the extent (span from the first byte to the last byte) of datatype

```c
int MPI_type_get_extent(MPI_Datatype newtype,
                        MPI_Aint *lb,
                        MPI_Aint *extent)
```

- MPI allows to change the extent of a datatype
  - using lb_marker and ub_marker
  - do not affect the size or count of a datatype, and the message content
  - do affect the outcome of a replication of this datatype
Derived Data Types in MPI:
Example for MPI_TYPE_VECTOR

- Create data type describing one column of a matrix
  - assuming row-major layout like in C

```c
double matrix[30]
MPI_Datatype nt;

// count = nrows, blocklength = 1,
// stride = ncols
MPI_Type_vector(nrows, 1, ncols,
                MPI_DOUBLE, &nt);
MPI_Type_commit(&nt);

// send column
MPI_Send(&matrix[1], 1, nt, ...,);

MPI_Type_free(&nt);
```

```
&matrix[1]
```
Derived Data Types in MPI:(MPI_Type_create_subarray

- Create sub array data type
  
  ```c
  MPI_Type_create_subarray(int dims,
  int ar_sizes[], int ar_subsizes[], int ar_starts[],
  int order, MPI_Datatype oldtype, MPI_Datatype * newtype)
  ```

- **dims:** dimension of the array
- **ar_sizes:** array with sizes of array (dims entries)
- **ar_subsizes:** array with sizes of subarray (dims entries)
- **ar_starts:** start indices of the subarray inside array (dims entries), start at 0 (also in Fortran)
- **order**
  - **row-major:** MPI_ORDER_C
  - **column-major:** MPI_ORDER_FORTRAN
- **oldtype:** data type the array consist of
- **newtype:** data type describing a subarray
Derived Data Types in MPI:
MPI_Type_create_subarray

```c
MPI_Type_create_subarray(dims, ar_sizes, ar_subsizes,
ar_starts, order, oldtype, &nt);

MPI_Type_commit(&nt);
// use nt...
MPI_Type_free(&nt);
```
Derived Data Types in MPI: MPI_TYPE_CREATE_STRUCT

- **Most general type constructor**
  - Describe blocks with arbitrary data types and arbitrary displacements

```c
MPI_Type_create_struct(int count, 
  int block_lengths[],
  MPI_Aint displs[],
  MPI_Datatype types[],
  MPI_Datatype *newtype)
```

- **Example**
  - `count = 2`
  - `types[0]` = [green]
  - `types[1]` = [blue]
  - `block_lengths[0] = 1`
  - `block_lengths[1] = 3`

The contents of `displs` are either the displacements in **bytes** of the block bases or **MPI addresses**.
What about displacements?

```fortran
MPI_GET_ADDRESS(location, address, ierror)
   <type> location
   INTEGER(KIND=MPI_ADDRESS_KIND) address
```

Example:

```fortran
double precision a(100)
integer a1, a2, disp
call MPI_GET_ADDRESS(a(1), a1, ierror)
call MPI_GET_ADDRESS(a(50), a2, ierror)
disp=a2-a1
```

Result would usually be \( \text{disp} = 392 \) (49 x 8)

When using absolute addresses, set buffer address = \texttt{MPI.BOTTOM}
Derived Data Types in MPI: Example for MPI_TYPE_CREATE_STRUCT

- **Task:** Communicate array of real values together with a supplementary real number

```
integer abl(2), adp(2), atp(2), newtype

double precision a(1001), eps

abl(1) = 1000
abl(2) = 1

atp(1) = MPI_DOUBLE_PRECISION
atp(2) = MPI_DOUBLE_PRECISION

call MPI_GET_ADDRESS(a, adp(1), ierror)
call MPI_GET_ADDRESS(eps, adp(2), ierror)
call MPI_TYPE_CREATE_STRUCT(2, abl, adp, atp, newtype, ierror)
call MPI_TYPE_COMMIT(newtype, ierror)
...
call MPI_SEND(MPI_BOTTOM, 1, newtype,...)
```
Derived data types provide a flexible tool to communicate complex data structures in an MPI environment.

**Most important calls:**
- `MPI_Type_vector` (second simplest)
- `MPI_Type_create_subarray`
- `MPI_Type_create_struct` (most advanced)
- `MPI_Type_commit`/`MPI_Type_free`
- `MPI_GET_ADDRESS`
- `MPI_Type_get_extent`
- `MPI_Type_size`

**Other useful features:**
- `MPI_Type_contiguous`, `MPI_Type_indexed`

**Matching rule:** send and receive match if specified basic datatypes match one by one, regardless of displacements.
- Correct displacements at receiver side are automatically matched to the corresponding data items.
Virtual Topologies
a multi-dimensional process naming scheme
Virtual Topologies

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

Let MPI map ranks to coordinates
User: map array segments to ranks
Create new communicator accompanied by Cartesian topology

\[
\text{MPI\_Cart\_create(MPI\_Comm oldcomm,}\n\text{ ndims, int dims[], int periods[],}\n\text{ int reorder, MPI\_Comm * cart\_comm)}
\]

**ndims**: number of dimensions

**dims**: array with \text{ndims} elements,
\text{dims}[i] specifies the number of ranks in dimension \text{i}

**periods**: array with \text{ndims} elements,
\text{periods}[i] specifies if dimension \text{i} is periodic

**reorder**: allow rank of \text{oldcomm} to have a different rank in \text{cart\_comm}

\[
\begin{array}{cccc}
\text{dim 0} & & & \\
0 & 3 & 6 & 9 \\
(0,0) & (0,1) & (0,2) & (0,3) \\
\text{dim 1} & & & \\
1 & 4 & 7 & 10 \\
(1,0) & (1,1) & (1,2) & (1,3) \\
2 & 5 & 8 & 11 \\
(2,0) & (2,1) & (2,2) & (2,3)
\end{array}
\]

\text{ndims} = 2
\text{dims} = \{4, 3\}
\text{periods} = \{0, 0\}
\text{reorder} = 0
Create new communicator accompanied by Cartesian topology

\[ \text{MPI\_Cart\_create}(\text{MPI\_Comm oldcomm, ndims, int dims[], int periods[], int reorder, MPI\_Comm * cart\_comm}) \]

- **ndims**: number of dimensions
- **dims**: array with `ndims` elements,
  - `dims[i]` specifies the number of ranks in dimension \(i\)
- **periods**: array with `ndims` elements,
  - `periods[i]` specifies if dimension \(i\) is periodic
- **reorder**: allow rank of `oldcomm` to have a different rank in `cart_comm`

\[ \text{ndims} = 2 \]
\[ \text{dims} = \{4, 3\} \]
\[ \text{periods} = \{1, 0\} \]
\[ \text{reorder} = 0 \]
• Retrieve rank in new Cartesian communicator
  
  ```c
  MPI_Comm_rank(cart_comm, &cart_rank)
  ```

• Map rank → coordinates
  
  ```c
  MPI_Cart_coords(comm, rank, int maxdims, int coords[])
  ```
  
  - `rank`: any rank which is part of Cartesian communicator `comm`
  - `coords`: array of `maxdims` elements, receives the coordinates for `rank`

• Map coordinates → rank
  
  ```c
  MPI_Cart_rank(comm, int coords[], int * rank)
  ```
  
  - `coords`: coordinates; if periodic in direction `i`, `coords[i]` are automatically mapped into the valid range, else they are erroneous

• Where am I inside the grid?
  
  ```c
  int coords[ndims];
  MPI_Comm_rank(cart_comm, &cart_rank);
  MPI_Cart_coords(cart_comm, cart_rank, ndims, coords);
  ```
Example: 12 processes arranged on a 4 x 3 grid
- Column-major numbering
- Process coordinates begin with 0
### Sending/receiving from neighbors typical task in Cartesian topologies

**MPI_Cart_shift**

```c
MPI_Cart_shift(cart, direction, disp,
               int * source_rank, int * dest_rank)
```

- **direction**: dimension to shift
- **disp**: offset to shift: > 0 shift in positive direction,
  < 0 shift in negative direction
- **src/dst**: returned ranks as input into `MPI_Sendrecv*` calls

Example: 4x3 process grid, periodic in 1\(^{st}\) dimension, each process has an int value, which gets shifted

- Shift in 1\(^{st}\) dimension, which is periodic
- Shift in 2\(^{nd}\) dimension, which is non-periodic

For non-periodic dimensions `MPI_PROC_NULL` is returned on boundaries.