Message Passing with MPI

PPCES 2016

Hristo Iliev
IT Center / JARA-HPC
Agenda

- Motivation
- Part 1
  - Concepts
  - Point-to-point communication
  - Non-blocking operations
- Part 2
  - Collective operations
  - Communicators
  - User datatypes
- Part 3
  - Hybrid parallelisation
  - Common parallel patterns
Collective Operations

- MPI collective operations involve all ranks in a given communication context (communicator) at the same time.

- All ranks in the communicator must make the same MPI call for the operation to succeed.
  
  NB: There should be only one call per MPI rank (i.e. not per thread).

- Some collective operations are globally synchronous.
  
  → The MPI standard allows for early return in some ranks.

- Collective operations are provided as convenience to the end user and can be (and often are) implemented with basic point-to-point communication primitives.
  
  → But they are usually tuned to deliver the best system performance.
The only explicit synchronisation operation in MPI:

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

\[ \begin{align*}
\text{max}(t_{S,0} ; t_{S,1} ; t_{S,2}) < \min(t_{E,0} ; t_{E,1} ; t_{E,2})
\end{align*} \]
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 is usually quite low
Data Replication (Broadcast)

Replicate data from one rank to all other ranks:

```
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**: communication context

Notes:

- in all ranks but root, **data** is an output argument
- in rank root, **data** is an input argument
- **MPI_Bcast** completes only after all ranks in **comm** have made the call
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)
  ```

```
Ranks
```
```
data
```
```
A_0
```
```
Ranks
```
```
data
```
```
Broadcast
```
```
Ranks
```
```
data
```
```
A_0
```
```
A_0
```
```
A_0
```
```
A_0
```
Data Replication (Broadcast)

Replicate data from one rank to all other ranks:

**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
```
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:

- **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**: communication context

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

Significant at root rank only
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)
  ```

- 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
  - for each chunk the amount of data sent must match the receive size, i.e.
    ```c
    if sendtype == recvtype holds, then sendcount == recvcount must hold too
    ```
Data Scatter

Distribute chunks of data from one rank to all ranks:

\[
\text{MPI Scatter} \left( \text{void *sendbuf, int sendcount, MPI Datatype sendtype,} \\
\text{void *recvbuf, int recvcount, MPI Datatype recvtype,} \\
\text{int root, MPI Comm comm} \right)
\]
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** is only accessed in the root rank
- **recvbuf** is written into in all ranks
- example use:

```
// 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:

- **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
Data Gather

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

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

![Diagram showing the process of data gathering](image)
Gather-to-All

Collect chunks of data from all ranks in all ranks:

\[
\text{MPI\_Allgather \ (void *sendbuf, int sendcount, MPI\_Datatype sendtype,}
\text{  void *recvbuf, int recvcount, MPI\_Datatype recvtype, MPI\_Comm comm)}
\]

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
→ for each chunk the receive size must match the amount of data sent
→ equivalent to \text{MPI\_Gather + MPI\_Bcast}, but possibly more efficient
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)
All-to-All

- Combined scatter and gather operation:

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

- Notes:
  
  → no root rank – 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)
All-to-All

- Combined scatter and gather operation:
  
  \[
  \text{MPI\textunderscore Alltoall (void *sendbuf, int sendcount, MPI\_Datatype sendtype,} \\
  \text{void *recvbuf, int recvcount, MPI\_Datatype recvtype, MPI\_Comm comm)}
  \]

- Note: a kind of global chunked transpose
### Varying Counts

- Position and length of each chunk can be explicitly specified with the so-called varying count (-v) versions
  - Displacement and count in units of data elements specified for each chunk

![Diagram of varying counts](image)

- Useful when the problem size is not divisible by the number of MPI processes or when dealing with irregular domain decomposition
Varying Counts

- Most collectives have varying count versions

MPI_Scatterv (void *sbuf, int *scnts, int *sdispls, MPI_Datatype stype, 
void *rbuf, int rcount, MPI_Datatype rtype, 
int root, MPI_Comm comm)

MPI_Gatherv (void *sbuf, int scount, MPI_Datatype stype, 
void *rbuf, int *rcnts, int *rdispls, MPI_Datatype rtype, 
int root, MPI_Comm comm)

MPI_Allgatherv (void *sbuf, int *snts, int *sdispls, MPI_Datatype stype, 
void *rbuf, int rcount, MPI_Datatype rtype, 
MPI_Comm comm)

MPI_Alltoallv (void *sbuf, int *scnts, int *sdispls, MPI_Datatype stype, 
void *rbuf, int *rcnts, int *rdispls, MPI_Datatype rtype, 
MPI_Comm comm)
Global Reduction

Perform an arithmetic reduction operation while gathering data

\[
\text{MPIReduce} (\text{void } \ast \text{sendbuf, void } \ast \text{recvbuf, int count,} \\
\qquad \text{MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm})
\]

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

Result is computed in- or out-of-order depending on the operation:

- All predefined operations are \textit{associative} and \textit{commutative}
- Beware of non-commutative effects on floats
### Global reduction

#### Element-wise and cross-rank operation

\[ rbuf[i] = sbuf_0[i] \odot sbuf_1[i] \odot sbuf_2[i] \odot \ldots \odot sbuf_{nranks-1}[i] \]

<table>
<thead>
<tr>
<th></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>sbuf_0[]</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>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>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>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>

\( \odot = \text{MPI}_\text{SUM} \)
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:

\[
\text{MPI\_Allreduce (void \*sendbuf, void \*recvbuf, int count,}
\text{ MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm)}
\]

Notes:

- no root rank – every rank receives the result of the reduction operation
- equivalent to \texttt{MPI\_Reduce + MPI\_Bcast} with the same root
- can be slower with non-commutative operations because of the forced inorder execution (the same applies to \texttt{MPI\_Reduce})
  - concerns non-commutative user-defined operations only
Collective Calls

- 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 true synchronising MPI

- One cannot use MPI_Recv to receive data sent by MPI_Scatter / MPI_Alltoall
- One cannot use MPI_Send to send data to MPI_Gather / MPI_Allgather / MPI_Alltoall
Advantages

- Collective operations implement portably common SPMD patterns
- Implementation-specific magic, 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, $O(1)$
Collective Operations
Agenda

- Motivation
- Part 1
  - Concepts
  - Point-to-point communication
  - Non-blocking operations
- Part 2
  - Collective operations
  - Communicators
  - User datatypes
- Part 3
  - Hybrid parallelisation
  - Common parallel patterns
Communication Contexts

- Each communication operation in MPI happens in a certain context:
  - Group of participating peers (process group)
  - Error handlers for communication and I/O operations
  - Local key/value store
  - Optional virtual topology

- MPI always provides two predefined contexts:
  - MPI_COMM_WORLD
    - contains all processes launched initially as part of the MPI program
  - MPI_COMM_SELF
    - contains only the current process

- A unique communication endpoints consists of a communicator handle and a rank from that communicator
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 (<strong>MPI_ANY_SOURCE</strong>)</td>
</tr>
<tr>
<td>Destination</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Tag</td>
<td>Explicit</td>
<td>Explicit, wildcard possible (<strong>MPI_ANY_TAG</strong>)</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

- **Simple flat addressing using MPI_COMM_WORLD often suffices**
Virtual Topologies

- Each communicator can have an associated topology
  - Mapping between ranks and abstract addresses
  - Virtual connectivity (neighbour links) information

- Three different topology kinds:
  - no topology – e.g. MPI_COMM_WORLD
  - Cartesian topology – regular $n$-dimensional grid
  - graph topology – general connectivity graph

- Not having a neighbour link in the topology does not prevent ranks from communicating with each other
Cartesian Topology

- **Regular n-dimensional grid**

  - Dimensions are numbered starting from 0
### Cartesian Topology

- **Construct a Cartesian topology**

  ```c
  MPI_Cart_create (MPI_Comm old_comm, int ndims, int dims[], int periods[],
  int reorder, MPI_Comm *comm_cart)
  ```

  - Creates a new communicator `comm_cart` from the process group of `old_comm` with an `ndims`-dimensional Cartesian topology attached
  - `dims[]` – specifies the number of nodes in each dimension
  - `periods[]` – specifies the periodicity of each dimension (boolean array)
  - `reorder` – if set to true (non-zero), hints the MPI runtime to reorder the ranks in the new communicator so that their virtual connectivity matches as closely as possible the physical one; otherwise ranks are kept
Balanced Cartesian Distribution

Create a balance distribution of a number of processes

\[
\text{MPI\_Dims\_create (int nnodes, int ndims, int dims[])}
\]

→ Computes the most balanced way to arrange \textit{nnodes} ranks into an \textit{ndims}-dimensional grid

→ Non-zero elements of \textit{dims} specify the number of nodes in the corresponding dimension

→ Zero elements are filled with the optimal number of nodes in the corresponding dimension

→ Error if the product of non-zero elements of \textit{dims} does not divide \textit{nnodes}
Balanced Cartesian Distribution

- Create a balance distribution of a number of processes

\[
\text{MPI\_Dims\_create (int nnodes, int ndims, int dims[])}
\]

- Factors \text{nnodes} / \text{product}\{\{1\} \cup \{\text{non-zero elements of dims}\}\}
- The computed sizes are set in non-increasing order
  - the lowest-numbered dimension receives the biggest size
- Example (taken from the MPI standard):

<table>
<thead>
<tr>
<th>dims before call</th>
<th>function call</th>
<th>dims on return</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>MPI_Dims_create(6, 2, dims)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>MPI_Dims_create(7, 2, dims)</td>
<td>(7,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI_Dims_create(6, 3, dims)</td>
<td>(2,3,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI_Dims_create(7, 3, dims)</td>
<td>erroneous call</td>
</tr>
</tbody>
</table>
Coordinate Conversion

- Translate Cartesian coordinate tuples into ranks

```c
MPI_Cart_rank (MPI_Comm comm, int coords[], int *rank)
```

→ `comm` – Cartesian communicator
→ `coords` – an array of at least `ndims` elements – Cartesian coordinates
→ `rank` – corresponding process rank in `comm`

- Translate ranks into Cartesian coordinate tuples

```c
MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int coords[])```

→ `coords` – an array of `maxdims` elements to receive the coordinates
→ `maxdims` should be equal to or larger than `ndims`
**Cartesian Shift**

- **Find ranks of neighbour processes**

  ```c
  MPI_Cart_shift (MPI_Comm comm, int dir, int disp, int *source, int *dest)
  ```

  → Computes the ranks of neighbours to communicate with in order to perform data shift (e.g. using `MPI_Sendrecv`) at distance of `disp` in direction `dir`

  → Equivalent to:

  → obtain the Cartesian coordinates of the calling process

  → translate (..., coord\textsubscript{dir} + disp, ...) into rank `dest`

  → translate (..., coord\textsubscript{dir} – disp, ...) into rank `source`

  → If the source or the destination lies beyond a non-periodic boundary, the corresponding rank is returned as `MPI_PROC_NULL`
Cartesian Shift

- Example: periodic boundary (dir = 1, disp = 2)
Example: periodic boundary (dir = 1, disp = 2)
Cartesian Shift

Example: non-periodic boundary (dir = 1, disp = 2)

(source ranks) ➔ (calling ranks) ➔ (dest. ranks)
Example: non-periodic boundary (dir = 1, disp = 2)

Cartesian Shift

- Source ranks: (0, 0) 0, (1, 0) 5
- Calling ranks: (0, 1) 1, (0, 2) 2, (1, 1) 6, (1, 2) 7
- Destination ranks: (0, 3) 3, (0, 4) 4, (1, 3) 8, (1, 4) 9

Send:
- (0, 0) 0 to (0, 1) 1
- (0, 1) 1 to (0, 2) 2
- (1, 0) 5 to (1, 1) 6
- (1, 1) 6 to (1, 2) 7

Null messages: (0, 4) 4 to (0, 3) 3, (1, 4) 9 to (1, 3) 8
Cartesian Partitioning

- **Split a Cartesian communicator along some dimensions**

  ```c
  MPI_Cart_sub (MPI_Comm comm, int remain_dims[], MPI_Comm *newcomm)
  ```

  - `remain_dims` – boolean array; a true value flags particular dimension as being preserved by the operation (i.e. no splitting along that dimension)
  - Creates a new Cartesian subcommunicator for each node in non-preserved dimensions
  - Nodes with the same coordinate along non-preserved dimensions become members of the same subcommunicator
  - Periodicity of the preserved dimensions is carried on into the newly created subcommunicators
Cartesian Partitioning

- Example: initial 2x5 Cartesian topology
Cartesian Partitioning

- Example: remain_dims = { true, false }

→ Five one-dimensional subcommunicators created as a result

→ Each subcommunicator contains 2 processes
Cartesian Partitioning

- Example: remain_dims = \{\text{false, true}\}

→ Two one-dimensional subcommunicators created as a result

→ Each subcommunicator contains 5 processes


Cartesian Partitioning

- Example: `remain_dims = { false, false }`

→ Ten zero-dimensional subcommunicators created as a result
→ Each sub communicator contains only one process
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

- Do not try to free predefined communicators such as MPI_COMM_WORLD, MPI_COMM_SELF or MPI_COMM_NULL
Communicators
Agenda

- **Motivation**
- **Part 1**
  - Concepts
  - Point-to-point communication
  - Non-blocking operations
- **Part 2**
  - Collective operations
  - Communicators
  - User datatypes
- **Part 3**
  - Hybrid parallelisation
  - Common parallel patterns
User Datatypes

- Basic MPI datatypes can be combined into complex user datatypes
  - User (derived) datatypes can be further combined into even more complex derived datatypes

- MPI datatypes are essentially instructions for accessing the binary content of the buffer
  - type sequence – \((\text{basic data type}, \text{displacement})\)
    - displacements are relative to the beginning of the memory buffer and can be positive or negative
  - type map – \(\{ (\text{type}_0, \text{disp}_0), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1}) \} \)
  - type signature – \(\{ \text{type}_0, \ldots, \text{type}_{n-1} \} \)

- The type signature at the sender **must** match that at the receiver
Datatype Internals

- **Lower and upper bound:**
  - $lb(\text{datatype}) = \min \; disp_j$
  - $ub(\text{datatype}) = \max (disp_j + \text{sizeof}(type_j)) + \text{padding}$

- **Extent**
  - $extent(\text{datatype}) = ub(\text{datatype}) - lb(\text{datatype})$
  - The span in memory from the first to the last basic element
  - The size of the step when accessing consecutive elements of that type

- **Size**
  - $size(\text{datatype}) = \sum \text{sizeof}(type_j)$
  - The total amount of bytes taken by the datatype, not counting any gaps in it
Datatype Internals

- **Example: MPI_INT**
  - `type map = { (int, 0) }
  - `lb = 0`
  - `ub = 4`
  - `extent = 4 bytes`
  - `size = 4 bytes`

- **All predefined basic MPI datatypes have lower bound 0, i.e. data is flush with the buffer start**

- **Platform-specific alignment rules are taken into account**
  - The upper bound is therefore adjusted if necessary
Datatype Internals

```c
MPI_Send(sbuf, 2, stype, dest, 0, MPI_COMM_WORLD);
```

```c
MPI_Recv(rbuf, 2, rtype, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

![Diagram of datatype internals]

```c
MPI_Recv(rbuf, 2, rtype, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```
Datatype Internals

MPI_Send(sbuf, 2, stype, dest, 0, MPI_COMM_WORLD);

MPI_Recv(rbuf, 1, rtype, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Recv(rbuf, 1, rtype, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
Datatype Internals

MPI_Send(sbuf, 2, stype, dest, 0, MPI_COMM_WORLD);

**sbuf**

**message**

**rbuf**

extent of stype

extent of rtype

type map of stype

type map of rtype

MPI_Recv(rbuf, 1, rtype, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

**types not congruent**
Contiguous Datatypes

- Create a sequence of elements of an existing datatype

\[
\text{MPI\_Type\_contiguous (int count, MPI\_Type oldtype, MPI\_Type *newtype)}
\]

- The new datatype represents a contiguous sequence of \text{count} elements of \text{oldtype}
- The elements are separated from each other by the extent of \text{oldtype}
- A send/receive of one element of \text{newtype} is congruent with a receive/send of \text{count} elements of \text{oldtype}

- Useful for sending entire matrix rows (C/C++) or columns (Fortran)
Vector Datatypes

- Create a sequence of equally spaced blocks of elements

```c
MPI_Type_vector (count, blen, stride, oldtype, newtype)
```

- The new datatype represents a sequence of `count` blocks, each containing `blen` elements of the old datatype
- Every two consecutive blocks are separated by `stride` elements each

- Useful for sending matrix columns (C/C++) or rows (Fortran)
  - `stride` = row (C/C++) | column (Fortran) length (in number of elements)
  - `blen` = 1 (or the number of consecutive rows/columns)
  - `count` = number of rows (C/C++) | columns (Fortran)
Vector Datatypes

Example: single column of a C/C++ matrix

→ mat[3][6]

- **blen** = 1
- **stride** = 6
- **count** = 3

consecutive memory cells
Vector Datatypes

- Example: two consecutive columns of a C/C++ matrix

  \[ \text{mat}[3][6] \]
Structure Datatypes

- **The most generic datatype**
  
  - Useful for C/C++ structures and Fortran derived data type / COMMON blocks

  ```cpp
  MPI_Type_create_struct (int count, int blens[], MPI_Aint displs[],
                          MPI_Datatype types[], MPI_Datatype *datatype)
  ```

  - **count**: number of blocks in the datatype
  - **blens[]**: number of elements in each block
  - **displs[]**: displacement in bytes from the start of each block
  - **types[]**: datatype of the elements in each block
  - **datatype**: handle of the new datatype
Structure Datatypes

The most generic datatype

→ Useful for C/C++ structures and Fortran derived data type / COMMON blocks

```
MPI_Type_create_struct (int count, int blens[], MPI_Aint displs[],
                        MPI_Datatype types[], MPI_Datatype *datatype)
```

```
|-----------|-----------|-----------|-----------|
```

```
```

memory
Structure Datatypes

- The most generic datatype

  → Corresponds to C/C++ struct

```c
typedef struct {
    float mass;
    double pos[3];
    char sym;
} Particle;

int blens[] = { 1, 3, 1 };
MPI_Aint displs[] = {
    offsetof(Particle, mass),
    offsetof(Particle, pos),
    offsetof(Particle, sym) };
MPI_Type types[] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };

MPI_Type particle_type;
MPI_Type_create_struct(3, blens, displs, types, &particle_type);
```
Using Derived Datatypes

- Register a datatype for use with communication operations:
  
  ```
  MPI_Type_commit (MPI_Datatype *datatype)
  ```
  
  - A datatype must be committed before it can be used in communications
  - All predefined datatypes are already committed
  - Intermediate datatypes, i.e., ones used for building more complex datatypes but not used in communication, can be left uncommitted

- Deregister and free a datatype:
  
  ```
  MPI_Type_free (MPI_Datatype *datatype)
  ```
  
  - Derived datatypes, build from the freed datatype, are not affected
  - `datatype` set to `MPI_TYPE_NULL` upon successful return
Structure Datatypes

The most generic datatype

typedef struct {
    float mass;
    double pos[3];
    char sym;
} Particle;

int blens[] = { 1, 3, 1 };
MPI_Aint displs[] = { offsetof(Particle, mass),
                        offsetof(Particle, pos),
                        offsetof(Particle, sym) };
MPI_Type types[] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };

MPI_Type particle_struct;
MPI_Type_create_struct(3, blens, displs, types, &particle_struct);
MPI_Type_commit(&particle_struct);

particle_struct can now be used to send a scalar of type Particle
Structure Datatypes

- Resize to the true size of the structure

```c
int blens[] = { 1, 3, 1 };
MPI_Aint displs[] = { offsetof(Particle, mass),
                     offsetof(Particle, pos),
                     offsetof(Particle, sym) };
MPI_Type types[] = { MPI_FLOAT, MPI_DOUBLE, MPI_CHAR };

MPI_Type particle_struct;
MPI_Type_create_struct(3, blens, displs, types, &particle_struct);
// No need to commit particle_struct - not used in communication

MPI_Aint true_size = sizeof(Particle);
MPI_Type_create_resized(particle_struct, 0, true_size, &particle_type);
MPI_Type_commit(&particle_type);
```

- `MPI_Type_create_resized` takes an existing datatype and creates a new one with modified lower bound and extent
Using Derived Datatypes

- Datatypes can be mixed and matched on both sides of a communication operation as long as their type signatures match.
  - E.g. one can send 10 `MPI_INT` elements and receive them as a single element of a contiguous datatype with `count = 10` and `oldtype = MPI_INT`.
  - Extra care should be taken when using derived datatypes in collective operations.

- If the amount of data in the received message is not enough to build an integral number of elements of a derived datatype, a count of `MPI_UNDEFINED` is returned by `MPI_Get_count`.

Communicators