MPI: From Fundamentals To Applications

David W. Walker
Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6067
walker@ornl.gov

WEB: http://www.ornl.gov/~walker/

April 10, 1995

This tutorial is copyright material. However, permission is granted to others to use all or part of the tutorial for non-commercial educational purposes.

© D. W. Walker 1995

MPI Information Resources

This tutorial is based on:

• the May 5, 1994 version of the MPI specification
• the October 28, 1994, errata sheet.

These may be obtained from the MPI Resource Center at ORNL:

http://www.ornl.gov/~walker/mpi/

The MPI Resource Center also contains pointers to:

• a frequently-asked-questions file;
• information about MPI implementations;
• a hypertext version of the MPI draft;
• papers and talks about MPI.

Copyright and Usage

The material in this tutorial is copyright. However, permission is granted to others to use all or part of the tutorial for non-commercial, educational purposes. Appropriate acknowledgement would be appreciated.

About this Tutorial

What this tutorial does not do:

• Give an exhaustive description of the 120 MPI calls.

What this tutorial aims to do:

• Give a good overview of the capabilities of MPI;
• Demonstrate some of these capabilities with detailed examples;
• Give the rationale behind some of the design decisions.
What Is MPI?

A proposed standard message passing interface for,

- explicit message passing in
- application programs on
- MIMD distributed memory concurrent computers.

Why Is A Standard Needed?

- Portability and ease-of-use;
- Provides hardware vendors with well-defined set of routine to implement efficiently;
- Precursor for the development of concurrent software industry;
- Will lead to more widespread use of concurrent computers.

History and Schedule

April 1992. Workshop in Williamsburg, VA.
Nov. 1992. Meeting at Supercomputing ’92
June 1993. Fifth meeting of MPIF. Core completed.
Aug. 1993. Sixth meeting of MPIF.
Sept. 1993. Seventh meeting of MPIF.
          Final readings.
          MPI draft completed.
Nov. 1993. MPI draft presented at SC ’93.
          Start of public comment period.
Feb. 1994. Final MPI Forum meeting in Knoxville

What Is In MPI?

- Point-to-point message passing
- Collective communication
- Support for process groups
- Support for communication contexts
- Support for application topologies
- Environmental inquiry routines
- Profiling interface
Process Model and Groups

- Fundamental computational unit is the process.
  Each process has:
  - an independent thread of control,
  - a separate address space
- MPI processes execute in MIMD style, but:
  - No mechanism for loading code onto processors,
    or assigning processes to processors
  - No mechanism for creating or destroying processes
- MPI supports dynamic process groups.
  - Process groups can be created and destroyed
  - Membership is static
  - Groups may overlap
- No explicit support for multithreading, but MPI is designed to be thread-safe

Communication Scope

- In MPI, a process is specified by:
  - a group
  - a rank relative to the group (0, 1, 2, ..., N – 1)
- A message label is specified by:
  - a message context
  - a message tag relative to the context
- Groups are used to partition process space
- Contexts are used to partition “message label space”
- Groups and contexts are bound together to form a communicator object. Contexts are not visible at the application level.
- A communicator defines the scope of a communication operation

Point-to-point Communication

- MPI provides for point-to-point communication between pairs of processes.
- Message selectivity is by rank and message tag.
- Rank and tag are interpreted relative to the scope of the communication.
- The scope is specified by the communicator.
- Rank and tag may be wildcarded.
- The components of a communicator may not be wildcarded.

Communication Completion

- A communication operation is locally complete on a process if the process has completed its part in the operation.
- A communication operation is globally complete if all processes involved have completed their part in the operation.
  
  A communication operation is globally complete if and only if it is locally complete for all processes.
Blocking Send

- Consider the standard blocking send routine:

```c
mpi_send (  
    IN start_of_buffer,
    IN number_of_items,
    IN datatype,
    IN destination_rank,
    IN tag,
    IN communicator,
    OUT error_code)
```

Blocking Receive

- Consider the standard blocking receive routine:

```c
mpi_recv (  
    OUT start_of_buffer,
    IN max_number_of_items,
    IN datatype,
    IN source_rank,
    IN tag,
    IN communicator,
    OUT return_status,
    OUT error_code)
```

- In a receive routine, `source_rank` and `tag` can have the values `mpi_any_source` and `mpi_any_tag`.

Return Status Objects

- The return status object is used after completion of a receive to find the actual length, source, and tag of a message.
- Return status object is MPI-defined type.
- In C status is a structure:
  - `status.source` gives source process
  - `status.tag` gives the message tag
- In Fortran `status` is an integer array
  - `status(MPI_SOURCE)` gives source process
  - `status(MPI_TAG)` gives message tag
- Number of elements in message is given by

```c
mpi_get_count (status, datatype, count, ierr)
```
Nonblocking Send

- Consider the standard nonblocking send routine:

  ```c
  mpi_isend (
    IN start_of_buffer,
    IN number_of_items,
    IN datatype,
    IN destination_rank,
    IN tag,
    IN communicator,
    OUT request_id,
    OUT error_code)
  ```

Nonblocking Receive

- Consider the standard nonblocking receive routine:

  ```c
  mpi_irecv (
    OUT start_of_buffer,
    IN max_number_of_items,
    IN datatype,
    IN source_rank,
    IN tag,
    IN communicator,
    OUT request_id,
    OUT error_code)
  ```

- Receive is not passed a return status object.

Completion Routines

- Two basic ways of checking on nonblocking sends and receives:
  - Call a wait routine that blocks until completion
  - Call a test routine that returns a flag to indicate if complete

- Use of nonblocking and completion routines allow compation and communication to be overlapped

  ```c
  mpi_wait (request_id,
            return_status, ierr)
  mpi_test (request_id, flag,
            return_status, ierr)
  ```

- **mpi_wait** blocks until the communication is complete.
- **mpi_status** returns “immediately”, and sets flag to true if the communication is complete.

Multiple Completions

- Versions of wait and status routines that act on multiple communication operations.

  ```c
  mpi_waitall (count, list_requests,
              list_status, ierr)
  ```

- Block until all communication operations in a given list have completed.

  ```c
  mpi_waitany (count, list_requests,
              index, return_status, ierr)
  ```

- Block until at least one of the communication operations in a given list has completed.

  ```c
  mpi_waitsome (count, list_requests,
                count_done, list_index,
                list_status, ierr)
  ```

- There are similar test routines for checking completion status of a list of communication operations.
Communication Modes

- The mode of a point-to-point communication operation governs when a send operation is initiated, or when it completes.
- **Standard mode:**
  - A send may be initiated even if a matching receive has not been initiated.
- **Ready mode:**
  - A send may be initiated only if a matching receive has been initiated.
- **Synchronous mode:**
  - The same as standard mode, except the send will not complete until message delivery is guaranteed.
- **Buffered mode:**
  - Similar to standard mode, but completion is always independent of matching receive, and message may be buffered to ensure this.

Buffered Mode

- In buffered mode a user-supplied buffer may be used to buffer messages so that the sending process can always return from the send before the message just been received.
- To supply the system with the user buffer:
  - `mpi_buffer_attach (buffer, size, ierr)`
- To get user buffer back from system:
  - `mpi_buffer_detach (buffer, size, ierr)`
- This will block until all communication using buffer has completed.

Flavors of Communication

- For a send operation there are:
  - 4 communication modes
  - 2 blocking modes
  - ⇒ $4 \times 2 = 8$ types of send
- For a receive operation there are:
  - 1 communication mode
  - 2 blocking modes
  - ⇒ $1 \times 2 = 2$ types of receive
- Naming convention of send routines is:
  - `mpi_i[-,i] [-,r,s,b]send [-,i]` = blocking mode
  - `[-,r,s,b] =` communication mode
- Naming convention of receive routines is:
  - `mpi_i[-,i]recv`

Naming Conventions

<table>
<thead>
<tr>
<th>SEND</th>
<th>Blocking</th>
<th>Nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td><code>mpi_send</code></td>
<td><code>mpi_isend</code></td>
</tr>
<tr>
<td>Ready</td>
<td><code>mpi_rsend</code></td>
<td><code>mpi_irsend</code></td>
</tr>
<tr>
<td>Synchronous</td>
<td><code>mpi_isend</code></td>
<td><code>mpi_isend</code></td>
</tr>
<tr>
<td>Buffered</td>
<td><code>mpi_bsend</code></td>
<td><code>mpi_bsend</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RECEIVE</th>
<th>Blocking</th>
<th>Nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td><code>mpi_recv</code></td>
<td><code>mpi_recv</code></td>
</tr>
</tbody>
</table>

- Any type of receive routine can be used to receive messages from any type of send routine.
Send/Receive Operations

- In many applications, processes send to one process while receiving from another.
- Deadlock may arise if care is not taken.
- MPI provides routines for such send/receive operations.
- For distinct send/receive buffers:
  `mpi_sendrecv`
- For identical send/receive buffers:
  `mpi_sendrecv_replace`

Communicating Non-Primitive Datatypes

- Primitive datatypes:
  - `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE`
  - `MPI_COMPLEX`, `MPI_DOUBLE_COMPLEX`
  - `MPI_LOGICAL`, `MPI_CHARACTER`, `MPI_BYTE`
- MPI also supports array sections and structures through general datatypes.
- A general datatype is a sequence of primitive types and integer byte displacements.
  `Datatype = {(type0, disp0), (type1, disp1), ...}
  ... (type_n, disp_n)}`
- Together with a base address, a datatype defines a communication buffer.

Contiguous Datatype Constructor

- A general datatype is built up hierarchically from simpler components.
  `mpi_type_contiguous (count, oldtype, newtype, ierr)`
- The above creates a new datatype made up of `count` repetitions of `oldtype`.
- For example:
  `oldtype = {(double, 0), (char, 8)}`
  then if `count = 3`,
  `newtype = {(double, 0), (char, 8),
  (double, 16), (char, 24),
  (double, 32), (char, 40)}`

Vector Datatype Constructor

- This constructor replicates a datatype, taking blocks at fixed offsets.
  `mpi_type_vector (count, blocklen, stride, oldtype, newtype, ierr)`
- The new datatype consists of:
  - `count` blocks,
  - each block is a repetition of `blocklen` items of `oldtype`,
  - the start of successive blocks is offset by `stride` items of `oldtype`.
- If `count=2, stride=4, blocklen=3`, then `newtype` is:
  `{(double, 0), (char, 8),
  (double, 16), (char, 24),
  (double, 32), (char, 40),
  (double, 64), (char, 72),
  (double, 80), (char, 88),
  (double, 96), (char, 104)}`
Indexed Datatype Constructor

- This constructor replicates a datatype, taking
  blocks at fixed offsets.

\[
\text{mpi-type-indexed} \ (\text{count}, B, I, \text{oldtype}, \text{newtype}, \text{ierr})
\]

- The new datatype consists of:
  - \text{count} blocks,
  - the \ith block is of length \text{B[i]} items of
  - \text{oldtype}.
  - the offset of the start of the \ith block is \text{I[i]}
  - items of \text{oldtype}.

- If \text{count}=2, \text{I} = \{0,4\}, and \text{B} = \{3,1\}, then
  \text{newtype} is:

\[
\begin{align*}
\{(\text{double}, 64), \text{char}, 72), \\
(\text{double}, 80), \text{char}, 88), \\
(\text{double}, 96), \text{char}, 104), \\
(\text{double}, 0), \text{char}, 8\}
\end{align*}
\]

Structure Datatype Constructor

- This constructor generalizes the indexed datatype
  by allowing each block to be of a different datatype.

\[
\text{mpi-type-struct} \ (\text{count}, B, I, T, \text{newtype}, \text{ierr})
\]

- The new datatype consists of:
  - \text{count} blocks,
  - the length of the \ith block is \text{B[i]} items of type
  - \text{T[i]}.
  - the offset of the start of the \ith block is \text{I[i]}
  - bytes.

- If \text{count}=3, \text{T} = \{\text{MPI_FLOAT}, \text{type1, MPI_CHAR}\},
  \text{I} = \{0,1,26\}, and \text{B} = \{2,1,3\}, then \text{newtype} is:

\[
\begin{align*}
\{(\text{float}, 0), \text{float}, 4), \\
(\text{double}, 0), \text{char}, 24), \\
(\text{char}, 26), \text{char}, 27), \text{char}, 28)\}
\end{align*}
\]

Other Datatype Routines

- The size of a datatype is termed the extent.
- The extent is given by:

\[
\text{mpi-type-extent} \ (\text{datatype}, \text{extent}, \text{ierr})
\]

- \text{mpi-type-bvector} is same as \text{mpi-type-vector},
  except stride is given in bytes.
- \text{mpi-type-indexed} is same as
  \text{mpi-type-indexed}, except offsets in array of
  offsets are given in bytes.
- The offsets in a datatype may be given relative to a
  “base address,” given by the MPI constant
  \text{mpi-bottom}.
- The address of a location can be found thus:

\[
\text{mpi-address} \ (\text{location}, \text{address}, \text{ierr})
\]

Example of a General Datatype

- \text{A(1:17:2, 3:11, 2:10) -> E}

```plaintext
include 'mpi.h'
real x(100,100,100), w(9,9,9)
integer oneslice, twoslice, threeslice, sizesofreal
integer rank, ierr, status(MPI_DISCONNECT)
call mpi-init(ierr)
call mpi-comm_rank (MPI_COMM_WORLD, rank, ierr)
if (rank .eq. 0) then
  call mpi-type-extent (MPI_REAL, sizesofreal, ierr)
call mpi-type-vector (9, 1, 2, MPI_REAL,
  oneslice, ierr)
call mpi-type-bvector (9, 1, 100+sizesofreal,
  oneslice, twoslice, ierr)
call mpi-type-bvector (9, 1, 100+sizesofreal+
  twoslice, threeslice, ierr)
call mpi-type-commit (threeslice, ierr)
call mpi-send (a(1:3,2), 1, threeslice, 1, 0,
  MPI_COMM_WORLD, ierr)
else if (rank .eq. 1) then
  call mpi-recv (w, 9*9*9, MPI_REAL, 0, 0,
  MPI_COMM_WORLD, status, ierr)
end if
call mpi-f finalize (ierr)
```
Persistent Communication Requests

- Can bind together arguments of a communication call to take overhead out of a loop
- `mpi_sendinit` creates a communication request that completely specifies a standard send operation. It binds:
  - start of buffer, and number of elements sent,
  - datatype
  - destination rank, message tag, and communicator
- `mpi_recvinit` creates a communication request that completely specifies a receive operation.
- Similar routines for ready, synchronous, and buffered send modes.

Communicators

- Communicators are used to create independent “message universes”.

- Communicators are used to disambiguate message selection when an application calls a library routine that performs message passing. Non-determinacy may arise
  - if processes enter the library routine asynchronously,
  - if processes enter the library routine synchronously, but there are outstanding communication operations.

- A communicator
  - binds together groups and contexts
  - defines the scope of a communication operation
  - is represented by an opaque object
  - referenced by a handle

Asynchronous Library Calls 1

- The following shows the correct sequence of communication operations
Asynchronous Library Calls 2

- The following shows an incorrect sequence of communication operations:

```
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>recv(any)</td>
<td>recv(any)</td>
<td>send(1)</td>
</tr>
<tr>
<td></td>
<td>send(0)</td>
<td>library call</td>
</tr>
<tr>
<td>recv(1)</td>
<td>recv(0)</td>
<td>send(0)</td>
</tr>
<tr>
<td></td>
<td>recv(0)</td>
<td>recv(0)</td>
</tr>
</tbody>
</table>
```

- Deadlock results in this case.
- Need to differentiate between messages sent in library routine and rest of application.

Synchronous Library Calls 1

- In this case the library call is made synchronously.
- Still have problems if there are communication operations outstanding on entry.
- Intended behavior:

```
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(1)</td>
<td>library call</td>
</tr>
<tr>
<td>library call</td>
<td>recv(0)</td>
</tr>
</tbody>
</table>
```

Synchronous Library Calls 2

- Possible behavior:

```
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(1)</td>
<td>recv(0)</td>
</tr>
<tr>
<td>send(1)</td>
<td>recv(0)</td>
</tr>
</tbody>
</table>
```

- Again we must separate communication operations inside the library from those in the rest of the application.

Libraries and Communicators

- Different phases of an application are assigned different contexts to avoid their messages being confused.
- Library routines need to be passed a communicator for use within the routine.
- Note that use of communicators in library calls does not always guarantee safety. Two successive calls using the same communicator can result in one call receiving messages from the other if the call performs wildcarded receives. This is called “back-masking” and can be prevented by inserting a barrier at the end of the library call.
Collective Communication

- Involves coordinated communication within a group of processes.
- No message tags used
- All collective routines block until they are locally complete
- Two broad classes
  - data movement routines
  - global computation routines

Collective Data Movement

- Three types of collective data movement,
  - Broadcast
  - Gather
  - Scatter
- In the “all gather” routine all processes receive the result. This concatenates data from each process.
- The “all-to-all” routine each process sends to, and receives from, all other processes

Vector Variants

- The gather, scatter, allgather, and alltoall routines all have vector variants in which the each process can send and/or receive a different number of elements.
- For example, the simple version of gather routine:

  ```
  mpi_gather (sendbuf, sendcount, sendtype, 
              recvbuf, recvcount, recvtype, 
              root, comm, ierr)
  ```

- The vector version is:

  ```
  mpi_gatherv (sendbuf, sendcount, sendtype, 
              recvbuf, recvcounts, displs, 
              recvtype, root, comm, ierr)
  ```

- `recvcounts` is an array giving number of elements received from each process
- `displs` is an array giving displacement at which to store the elements received from each process.
Global Computation Routines

- Two types of global computation routine
  - reduce
  - scan
- If \( n \) is the size of the group, and \( D_{i,j} \) is the \( j \) data item in process \( i \), then a reduce routine evaluates,
  \[
  D_j = D_{0,j} \oplus D_{1,j} \oplus \cdots \oplus D_{n-1,j}
  \]
- \( \oplus \) is the reduction function. It must be associative and commutative.
- The \( D_j \) are stored in a specified process.
- A scan routine evaluates in process
  \[
  k = 0, 1, \ldots, n-1:
  \]
  \[
  D_{k,j} = D_{k,j} \oplus D_{k+1,j} \oplus \cdots \oplus D_{n-1,j}
  \]

Application Topologies

- In many applications, processes are arranged with a particular topology, e.g., a regular grid.
- MPI supports general application topologies by a graph in which communicating processes are connected by an arc.
- MPI also provides explicit support for Cartesian grid topologies.

\[\text{mpi\_cart\_create (comm\_old, ndims, dims, period, reorder, comm\_cart, ierr)}\]

- Periodicity in each grid direction may be specified.
- Inquiry routines transform between rank in group and location in topology
- For Cartesian topologies, row-major ordering is used for processes.

Global Computation Routines

- The function passed to a global computation routine is either:
  - a predefined MPI function, e.g., \texttt{mpi\_sum}.
  - a user supplied function
- A handle to the user supplied function is created using

\[
\text{mpi\_op\_create (function, commute, op, ierr)}
\]

- Three versions of reduce that return results to:
  - a single process
  - all processes
  - scatter a vector of results across all processes
- Segmented sums can be done by creating a subgroup for each segment.
- There are 4 global computation routines, plus 2 auxiliary routines (\texttt{mpi\_op\_create} and \texttt{mpi\_op\_free}).

Topological Inquiry Routines

- \texttt{mpi\_topo\_test} returns the type of topology associated with a communicator.
- Can find number of dimensions in a Cartesian topology:

\[
\text{mpi\_cart\_dim\_get (comm, ndims, ierr)}
\]

- More information on a Cartesian topology can be obtained with:

\[
\text{mpi\_cart\_get (comm, maxdims, periods, coords, ierr)}
\]

- Mapping of coordinate position in Cartesian topology to rank:

\[
\text{mpi\_cart\_rank (comm, coords, rank, ierr)}
\]

- Mapping of rank to coordinate position:

\[
\text{mpi\_cart\_coords (comm, rank, maxdims, coords, ierr)}
\]
Uses of Topologies

- Knowledge of application topology can be used to efficiently assign processes to processors.
- Cartesian grids can be divided into hyperplanes by removing specified dimensions.
- MPI provides support for shifting data along a specified dimension of a Cartesian grid.
- MPI provides support for performing collective communication operations along a specified grid direction.

Partitioning Cartesian Topologies

- A Cartesian topology can be partitioned into a set of Cartesian topologies of lower dimension using:
  
  ```c
  mpi_cart_sub (comm, remaindims, newcomm, ierr)
  ```

  ```c
  remaindims[i] is true if the i-th dimension is retained, and is false if it is discarded.
  ```

- Example:
  If a $2 \times 3 \times 4$ Cartesian topology is associated with `comm`, and
  
  ```c
  remaindims = (true, true, false)
  ```

  then the above call to `mpi_cart_sub`
  
  - creates 4 subcommunicators, each with a $2 \times 3$ Cartesian topology.
  - the value of `newcomm` returned is that containing the calling process in its associated group.

Topologies and Data Shifts

- Consider the following two types of shift for group of size $N$:
  
  - Circular shift by $J$. Data in process $K$ is sent to process $(J + K) \mod N$.
  - End-off shift by $J$. Data in process $K$ is sent to process $J + K$ if this is between 0 and $N - 1$. Otherwise, no data are sent.
- Topological shifts are performed using `mpi_sendrecv`.
- `mpi_shift` returns the ranks of the processes that a process must send to and receive from when performing a shift on a topological group.

Topologies and Collective Communication

- Suppose we want to perform a collective communication along a dimension of an application topology, e.g., a multicast:

  ```c
  +A+ +B+ +C+ +D+ => A A A A B B B B C C C C D D D D
  ```

  ```c
  R S T U V W R S T U V W R S T U V W R S T U V W
  ```

- We can call `mpi_cart_sub` with `remaindims=(false, true)` to generate new subcommunicators for first case.
- If `remaindims=(true, false)` we get the subcommunicators for the second case.
Caching Data

We can view a communicator as containing
- a group;
- a context;
- cached data.

Manipulating Cached Data 1

- To cache data:
  - get a key;
  - put pointer to data in box that key fits;
  - close box and lock it.

- To access cached data:
  - use key for cached data required to open box;
  - remove the pointer from the box.
- To discard cached data
  - discard box

Manipulating Cached Data 2

- To create a new key:
  \[
  \text{mpi_key_create (copyfun, deletefun, keyval, extrastate, ierr)}
  \]
- `copyfun` is a callback function used to copy cached attributes when `mpi_comm_dup` is used to duplicate a communicator.
- `deletefun` is a delete function used to free cached data when a call is made to either `mpi_comm_free` or `mpi_attr_delete`.
- To free a key:
  \[
  \text{mpi_key_free (keyval, ierr)}
  \]

Associating Cached Data

- To store data:
  \[
  \text{mpi_attr_put (comm, keyval, attribute, ierr)}
  \]
  This caches an attribute referenced by `keyval` with a communicator.
- To retrieve data:
  \[
  \text{mpi_attr_get (comm, keyval, attribute, flag, ierr)}
  \]
  Retrieves the attribute value referenced by `keyval`. If no attribute is associated with the key then `flag` is returned as false.
- To delete a cached attribute:
  \[
  \text{mpi_attr_delete (comm, keyval, ierr)}
  \]
  This invokes the delete callback function specified when the key was created.
Pack and Unpack

MPI provides routines that:

- Pack data into a contiguous buffer before sending it.
- Unpack data from a contiguous buffer after receiving it.

These routines are provided:

- for compatibility with other message passing libraries,
- to allow a message to be received in parts,
- to buffer outgoing messages in user space, thereby overriding the system buffering policy.

Packing Data

To pack data:

\[ \text{mpi}_\text{pack} \left( \text{inbuf}, \text{incount}, \text{datatype}, \right. \]
\[ \left. \text{outbuf}, \text{outsize}, \text{position}, \text{comm}, \text{ierr} \right) \]

This takes \text{incount} items of specified datatype from buffer \text{inbuf} and packs them into \text{outbuf} starting at offset \text{position}.

- The offset \text{position} is specified in bytes.
- On return \text{position} is set to the next location in \text{outbuf}.

Unpacking Data

To unpack data:

\[ \text{mpi}_\text{unpack} \left( \text{inbuf}, \text{insize}, \text{position}, \right. \]
\[ \left. \text{outbuf}, \text{outcount}, \text{datatype}, \text{comm}, \text{ierr} \right) \]

This extracts \text{outcount} items of specified datatype from the buffer \text{inbuf}, starting at offset \text{position}, and stores them in the buffer \text{outbuf}.

- On return \text{position} is set to the next location in \text{inbuf} after the data just unpacked.

Example of Unpacking

A message consists of sequences of reals or integers prefixed by a type identifier and the number of items.
Code for Unpacking Example

call mpi_recv (message, size, MPI_PACKED, 
MPI_ANY_SOURCE, MPI_ANY_TAG, 
comm, status, ierr)
pos = 0
call mpi_unpack (message, size, pos, 
typeid, i, MPI_INTEGER, 
comm, ierr)
do while (typeid .gt. 0)
call mpi_unpack (message, size, pos, 
items, 1, MPI_INTEGER, 
comm, ierr)
if (typeid .eq. 1) then
  unpack integers
else
  unpack reals
end if
call mpi_unpack (message, size, pos, 
typeid, i, MPI_INTEGER, 
comm, ierr)
end do

Unpack Example (cont.)

Integers are unpacked as follows

call mpi_unpack (message, size, pos, 
items(items), items, MPI_INTEGER, 
comm, ierr)
items = items * items

Reals are unpacked as follows

call mpi_unpack (message, size, pos, 
reals(reals), items, MPI_REAL, 
comm, ierr)
reals = reals * items

Particle Migration Example

- Problem:

  Particles move on a domain that is block
  distributed. Particle data must be moved between
  neighboring process subdomains when they cross
  subdomain boundaries.

- Since each process subdomain has 6 surfaces (for
  3D), the migration can be done in 6
  communications. We shall just consider the
  migration across the right-hand boundary.


Creating a 3D Topology

- A 3D virtual topology may be created as follows:

  int dims[3] = {0, 0, 0}, periods[3] = {0, 1, 1};
  int reorder, nprocs, ierr;
  MPI_Comm comm, comm3d;
  ierr = MPI_Comm_size (comm, &nprocs);
  ierr = MPI_Dims_create (nprocs, ndims, dims);
  ierr = MPI_Cart_create (comm, ndims, dims, 
    periods, reorder, comm3d);
Skeleton Code

```c
int ierr, status, errcode=21;

ierr = MPI_Topology (comm3d, &status);
if (status != MPI_CART)
    ierr = MPI_Abort (comm3d, errcode);

CREATE PARTICLE DATATYPE

CREATE SEND/RECV DATATYPES

COMMUNICATE DATA
```

Creating the Particle Datatype

```c
struct Particle {double x[3]; double v[3]; int k;}
particle[100];

MPI_Datatype dtype[3], Ptype;
int Sblock[3] = {1, 1, 1};
int Sindex[3];
MPI_Aint sized3;

ierr = MPI_Type_contiguous (3, MPI_DOUBLE, &dtype);
ierr = MPI_Type_extent (Dtype3, &sized3);
Sindex[0] = 0;
Sindex[1] = sized3;
Sindex[2] = sized3*2;
Type[0] = Dtype3;
Type[1] = Dtype3;
Type[2] = MPI_INT;
ierr = MPI_Type_struct (3, Sblock, Sindex, Dtype3, &Ptype);
```

Creating the Send/Receive Datatypes

```c
MPI_Datatype Sendtype;
int npart, scount, i, rightedge;
int Pindex[100], Pblock[100];
scount = 0;
for (i=0;i<npart;i++)
    if (particle[i].x[0] > rightedge)
        Pindex[scount] = i; Pblock[scount] = 1;
        scount++;

ierr = MPI_Type_index (scount, Pblock, Pindex, Dtype3, &Sendtype);
ierr = MPI_Type_commit (&Sendtype);
ierr = MPI_Type_commit (&Ptype);
```

Communicating the Data

```c
int dir=0, disp=1, source, dest, rcount;
MPI_Status cstatus;

ierr = MPI_Cart_shift (comm3d, dir, disp, source, dest);

ierr = MPI_Sendrecv(
    particle, 1, Sendtype, dest, 0,
    &particle[npart], space, Ptype,
    source, 0, comm3d, &cstatus);

ierr = MPI_Get_count (&cstatus, Ptype, &rcount);
REPACK PARTICLE ARRAY
```
Scattering Particles

• Problem:
A 1D domain is block decomposed. Each domain contains a different number of particles. Initially these all lie on one process. This process scatters the particles to the other processes.

• This example uses:

```c
mpi_scatterv(
    sendbuf, sendcounts, displs, sendtype,
    recvbuf, recvcount, recvtype, root, comm, ierr)
```

![Diagram of a process division with labels for processes P0 to P5 and subdomains n1 to n4.](image)

Creating the Particle Datatype

```c
struct Pstruct {double x[3]; double v[3]; int k;}
    part[100];
MPL_Datatype Stype[3], Dtype3, Ptype;
int Sblock[3] = {1, 1, 1};
inl Sindex[3];
MPL_Aint sized3;
```

```c
ierr = MPI_Type_contiguous (3, MPI_DOUBLE, & Dtype3);
ierr = MPI_Type_extent (Dtype3, & sized3);
Sindex[0] = 0;
Sindex[1] = sized3;
Sindex[2] = sized3*2;
Type[0] = Dtype3;
Type[1] = Dtype3;
Type[2] = MPL_INT;
ierr = MPI_Type_struct (3, Sblock, Sindex, Type, & Ptype);
```

Skeleton Code

```c
struct Pstruct {double x[3]; double v[3]; int k;}
    bigbuf[1000], part[100];
ierr, nprocs, root=0, p, Sum, npart;
int displ[10], n[10];
MPL_Comm comm;
MPL_Data_type Ptype;

CREATE PARTICLE DATATYPE

ierr = MPI_Comm_size (comm, & nprocs);
Sum = 0;
for (p=0; p<nprocs; p++)
    displ[p] = Sum;
    Sum += n[p]);
ierr = MPI_Scatter (n, 1, MPI_Int, & npart,
    1, MPI_Int, root, comm);
ierr = MPI_Scatterv (bigbuf, n, displ, Ptype,
    particle, npart, Ptype, root, comm)
```

MPI on Heterogeneous Networks

• MPI provides some support for distributed computing on heterogeneous networks.
  - Data type is supplied in all communication routines, permitting conversion between different internal data representations.
  - Support for process groups allows different machine clusters to work on distinct tasks.

BUT

• MPI provides no means of specifying the physical resources to be used
• Provides limited support for assigning processes to processors.
The Future of MPI

• MPI currently has (intentional) omissions:
  - no process management
  - no F90 or C++ bindings
  - no support for fault tolerance
  - I/O not addressed
  - no support for active messages
  - etc...

• MPI Forum has begun meeting to address such advanced features.

• Several implementations of MPI exist.