Standard MPI - Message Passing Interface

The message-passing paradigm is one of the oldest and most widely used approaches for programming parallel machines, especially those with distributed memory. There are two key attributes characterizing the message-passing paradigm:

- it assumes a partitioned address space,
- it supports only explicit parallelization.

The logical view of a machine supporting the message-passing paradigm consists of $P$ processes, each with its own exclusive address space.

MPI was developed by the MPI working group. In 1993 the MPI Forum (http://www.mpi-forum.org/) was constituted and MPI became a widely used standard for writing message-passing programs.

The interface is suitable for use by fully general MIMD programs, as well as those written in style of SPMD.

MPI includes:

- communication contexts,
- process groups,
- point-to-point communication,
- synchronization mechanisms,
- collective operations,
- process topologies.

From the user point of view MPI is a library of about 250 routines that can be used in programs written in C, C++ and Fortran. All MPI routines, data types, constants are prefixed by "MPI". The return code for successful completion is MPI_SUCCESS; otherwise an implementation-defined error code is returned. All MPI constants and data structures are defined for C in the file "mpi.h". This header file must be included in the MPI program.

The concepts that MPI provides, especially to support robust libraries, are as follows:

- Contexts of communication;
- Communicators;
- Groups of processes;
- Virtual topologies;
- Attribute caching.

COMMUNICATOR specifies the communication context for the communication operation. Each communication context provides a separate communication universe. Messages are always received within the context they were sent, and messages sent in different contexts do not interfere. The communicator also specifies the set of processes that share this communication context. Every process that belongs to a communicator is uniquely identified by its rank. The
rank of a process is an integer with values $0, 1, \ldots, P - 1$, where $P$ denotes the number of processes.

The communicators are identified by the handles with type MPI_Comm. A predefined communicator MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and all processes are identified by their rank in the group of MPI_COMM_WORLD.

**MPI routines**

1. Starting MPI library

   ```c
   int MPI_Init( int *argc, 
                 **argv[] )
   ```

   The arguments of MPI_Init are the command-line arguments of function main. MPI_Init can change these arguments.

2. Terminating MPI library

   ```c
   int MPI_Finalize( void )
   ```

3. Getting information (the number of processes determination)

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

   This function indicates the number of processes involved in the communicator.

4. Getting information (ranks determination)

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

   This function gives the rank of the calling process in the group of `comm`.

5. Communicator destruction

   ```c
   int MPI_Comm_free( MPI_Comm *comm )
   ```

   This collective operation marks the communication object for deallocation.

**Group constructors**

6. The handle to the group

   ```c
   int MPI_Comm_group( MPI_Comm comm, 
                       MPI_Group *group )
   ```

   This function returns a handle to the group of the communicator `comm`.

7. Selection to a new group

   ```c
   int MPI_Group_incl( MPI_Group group, 
                       int n, 
                       int *ranks, 
                       MPI_Group *newgroup )
   ```
This function creates a group `newgroup` that consists of the `n` processes in group `group` with ranks `ranks[0],...,ranks[n-1]`; the process with rank `i` in `newgroup` is the process with rank `ranks[i]` in `group`. In the case of `n=0` `newgroup` is `MPI_GROUP_EMPTY`.

8. Communicator construction to a new group

```c
int MPI_Comm_create ( MPI_Comm comm ,
                    MPI_Group newgroup ,
                    MPI_Comm *newcomm )
```

This function creates a new communicator `newcomm` with communication group defined by `newgroup` and a new context. The function returns `MPI_COMM_NULL` to processes that are not in `newgroup`.

9. The group of processes partitioning

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

This function partitions the group associated with the communicator `comm` into disjoint subgroups, one for each value of `color` (`color > 0`). Each subgroup contains all processes of the same `color`. Within each subgroup, the processes are ranked, in the order defined by the value of `key`, with ties broken according to their rank in the old group. A new communicator `newcomm` is created for each subgroup.

**Processes synchronization**

10. Synchronization (barrier)

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

**Sending and receiving messages (point-to-point communication)**

Message passing programs may be written using **asynchronous** or **synchronous** paradigm. In the asynchronous approach, all concurrent tasks execute asynchronously. Such programs have non-deterministic behavior due to a race conditions. In the case of synchronous implementation, tasks synchronize to perform interactions.

The basic operations in the message-passing programming paradigm are **send** and **receive**. The message transfer consists of the following three phases:

- data is pulled out of the send buffer and a message is assembled,
- a message is transferred to receiver,
- data is pulled from the incoming message and disassembled into the receive buffer.

MPI library offers several types of send/receive operations. We can distinguish:

- **synchronous and asynchronous**
  - **synchronous** - send operation returns only after the receiver has received the data.
asynchronous - send operation returns just after sending the message (the sender doesn’t wait for the acknowledge receipt of data).

- blocking and non-blocking:
  - blocking - send/receive operation blocks until it can guarantee that the semantics won’t be violated on return irrespective of what happens in the program. The sending process sends a request to communicate to the receiver. When the receiver encounters the target receive, it responds to the request. After receiving this response, the sender initiates the transfer operation.
  - non-blocking - send/receive operation returns before it is semantically safe to do so.

Non-blocking send can be matched with blocking receive, and vice-versa.

In the message-passing paradigm the communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication.

**Point-to-point communication arguments:**

- tag message tag (the range of valid values is 0, ..., MPI_TAG_UB).
- datatype ∈ {MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED}

**Blocking send and receive operations**

MPI provides four modes for blocking communication – asynchronous (standard) and three additional indicated by a one letter prefix: B for buffered, S for synchronous and R for ready.

11. Asynchronous send (**standard**)

```c
int MPI_Send( void *sendbuf,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm )
```

The function sends `count` data of the type specified by the parameter `datatype` stored in the send buffer, starting with the entry at address `sendbuf`. The destination of the message is uniquely specified by the `dest` and `comm` arguments. The `dest` is the rank of the destination process in the communicator domain specified by `comm`. The argument `tag` with values from the range 0, ..., MPI_TAG_UB is used to distinguish different types of messages. MPI_TAG_UB is defined MPI constant.

12. Buffered send

```c
int MPI_Bsend( void *sendbuf,
               int count,
               MPI_Datatype datatype,
               int dest,
```
The sender copies the data into the designed buffer and then sends it. Buffer allocation by the user may be required for the buffered mode to be effective. Buffering is done by the sender:

13. Send buffer allocation

    int MPI_Buffer_attach( void *buffer, int size)

This function provides to MPI a buffer in the user’s memory to be used for buffering outgoing messages. The buffer size is size. Only one buffer can be attached to a process at a time.

14. Synchronous send

    int MPI_Ssend( void *sendbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )

The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.

15. Ready send

    int MPI_Rsend( void *sendbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )

A send can be started only if the matching receive is already posted (the message is sent as soon as possible).

16. Blocking receive (standard)

    int MPI_Recv( void *recvbuf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status )

The function waits for the message with tag tag which should be sent by the process with rank source in the communicator domain specified by comm. The receiver may specify MPI_ANY_SOURCE value for source, and/or MPI_ANY_TAG value for tag, indicating that any source and/or any tag are acceptable. Data is pulled from the incoming message and
disassembled into the receive buffer with the initial address recvbuf. This buffer consists of count entities of the type specified by the parameter datatype (the length of the received message must be less than or equal to the length of the receive buffer). The source or tag if wildcard values were used in the receive operation, and other additional information concerned with the message may be returned by the status object. The type of status is MPI-defined.

17. Non-blocking testing

\[
\text{int MPI_Iprobe( int source, } \text{ int tag, } \\
\text{MPI_Comm comm, } \text{ int *flag, } \\
\text{MPI_Status *status })
\]

Returns flag=true if there is a message that can be received and that matches the arguments source, comm and tag. It is not necessary to receive a message immediately after it has been probed for (the same message may be probed for several times before it is received).

18. Blocking testing

\[
\text{int MPI_Probe( int source, } \text{ int tag, } \\
\text{MPI_Comm comm, } \text{ MPI_Status *status })
\]

Behaves like MPI_Iprobe except that it is a blocking call that returns only after a matching message has been found.

19. \[
\text{int MPI_Get_count( MPI_Status *status, } \\
\text{MPI_Datatype datatype, } \text{ int *count )}
\]

This function returns the number of entries received (count denotes number of received entries).

**Non-blocking send and receive operations**

Upon return from the non-blocking send or receive operation, the process is free to perform any computation that doesn’t depend on the completion of the operation. An alternative mechanism that often leads to better performance is to use non-blocking communication. A non-blocking send call will return before the message was copied out of the send buffer. A non-blocking receive call will return before the message is stored into the receive buffer. MPI provides the same four modes for non-blocking communication: standard, buffered, synchronous and ready (the same naming conventions are used). The prefix of I (for Immediate) indicates that the call is non-blocking. The additional argument request is used to identify whether the operation is terminated.

20. Non-blocking send (standard)

\[
\text{int MPI_Isend( void *sendbuf,}
\]
int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm,
    MPI_Request *request )

21. Non-blocking buffered send

    int MPI_Ibsend( void *sendbuf,
                    int count,
                    MPI_Datatype datatype,
                    int dest,
                    int tag,
                    MPI_Comm comm,
                    MPI_Request *request )

22. Non-blocking synchronous send

    int MPI_Issend( void *sendbuf,
                    int count,
                    MPI_Datatype datatype,
                    int dest,
                    int tag,
                    MPI_Comm comm,
                    MPI_Request *request )

23. Non-blocking receive

    int MPI_Irecv( void *recvbuf,
                    int count,
                    MPI_Datatype datatype,
                    int source,
                    int tag,
                    MPI_Comm comm,
                    MPI_Request *request )

24. Waiting for operation completion

    int MPI_Wait( MPI_Request *request,
                 MPI_Status *status )

    A call to MPI_Wait returns when the operation identified by request is completed. It returns
    information on the completed operation in status. The request is set to MPI_REQUEST_NULL.

25. Waiting for any operation completion

    int MPI_Waitany( int count,
                     MPI_Request *array_of_requests,
                     int *index,
                     MPI_Status *status )
The function blocks until one of the operations associated with the active request in the array array_of_requests has completed.

26. Testing

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

A call to MPI_Test returns flag = true if the operation identified by request is completed.

**Communication - recommendations**

Blocking operations facilitate safe and easier programming. Non-blocking operations are useful for speed up the computation by masking communication overhead. A non-blocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of non-blocking sends is advantageous only if data coping can be concurrent with computation. Additionally, one must be careful using non-blocking operations since errors can result from unsafe access to data that is the process of being communicated.

On the other hand the blocking synchronous protocol may lead to deadlock. Consider the following example. Example: The values x1 and x2 are calculated, and messages with the results of calculations are exchanged using synchronous send protocol.

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1=f1(x)</td>
<td>x2=f2(x)</td>
</tr>
<tr>
<td>MPI_Ssend</td>
<td>MPI_Ssend</td>
</tr>
<tr>
<td>x1 to Process P2</td>
<td>x2 to Process P1</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>MPI_Recv</td>
</tr>
</tbody>
</table>

In this simple example processes P1 waits for the matching receive at P2, and vice-versa, resulting in an infinite wait (deadlock situation).

**Advice to users:**
- Asynchronous blocking send (standard) MPI_Send; blocking buffered send MPI_Bsend for large amount of transmitted data, or in situation where the programmer wants more control.
- Asynchronous blocking receive MPI_Recv, preceded by non-blocking testing MPI_Iprobe.
- Synchronization - using barrier function MPI_Barrier.

**Alternative proposition:**

27. Combined blocking send and receive

```c
int MPI_Sendrecv( void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, int dest, int sendtag,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, int source, int recvtag,
                  MPI_Comm comm, MPI_Status *status )
```
This function combines in one call the sending of a message to one destination and the receiving of another message, from another process. Both source and destination are possibly the same. The arguments are the same like in \texttt{MPI\_Send} and \texttt{MPI\_Recv}. Both send and receive use the same communicator \texttt{comm}, but possibly different tags \texttt{sendtag} and \texttt{recvtag}. The send and receive buffers must be disjoint, and may have different lengths and datatypes.

**Collective communication**

A collective operation is executed by having all processes in the group call the communication routine, with matching arguments. The syntax and semantics are consistent with the syntax and semantics of the point-to-point communication.

28. Broadcast from one member to all members of a group

\begin{verbatim}
int MPI_Bcast( void *buffer,
               int count,
               MPI_Datatype datatype,
               int root,
               MPI_Comm comm )
\end{verbatim}

This function broadcasts a message from the process with rank \texttt{root} to all processes of the group. It is called by all members of the group using the same \texttt{comm} and \texttt{root}, and the contents of \texttt{root}’s buffer is copied to all processes. Arguments: \texttt{count} – number of entries in buffer, \texttt{datatype} – data type of buffer, \texttt{buffer} starting address of buffer.

29. Scatter data from one member to all members of a group (the same amount of data)

\begin{verbatim}
int MPI_Scatter( void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm )
\end{verbatim}

The process \texttt{root} splits data buffered in \texttt{sendbuf} into equal segments of size \texttt{sendcount} and sends the \(i\)-th segment to the \(i\)-th process in the group (\texttt{root} process included). Each
process receives the message (recvcount – size, recvtype – data type) and copy it into the buffer recvbuf. The arguments root and comm must have identical values on all processes. The amount of data sent must be equal to the amount of data received, pairwise between each process and the root.

![Diagram of data distribution](diagram.png)

30. Scatter data from one member to all members of a group (different amounts of data)

```c
int MPI_Scatterv( void *sendbuf,
                   int *sendcounts,
                   int *displs,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int recvcount,
                   MPI_Datatype recvtype,
                   int root,
                   MPI_Comm comm )
```

A vector variant of the scatter operation that allows different amounts of data to be sent to different processors. The parameter sendcount is replaced by the array sendcounts that determines the number of elements to be sent to each process. In particular, the target process sends sendcounts[i] elements to process i. Also, the array displs is used to determine where in sendbuf these elements will be sent from. In particular if, sendbuf is of the same type sendtype, the data sent to process i start at location displs[i] of array sendbuf. Both the sendcounts and displs arrays are of size equal to the number of processes.
31. Gather data from all members of a group to one member (the same amount of data)

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

The inverse operation to `MPI_Scatter`. Each process (root process included) within the domain of communicator `comm` sends the contents of its send buffer to the root process. The type signature of `sendcount` and `sendtype` on all processes must be equal to the type signature of `recvcount` and `recvtype` at the root process. So, the amount of data sent must be equal to the amount of data received, pairwise between each process and the root.

32. Gather data from all members of a group to one member (different amounts of data)

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

A vector variant of the gather operation that allows different amounts of data to be sent. The parameter `recvcount` is replaced by the array `recvcounts` that determines the number of elements to be sent to each process. The amount of data sent by processor `i` is equal to `recvcounts[i]`. The array `displs` is used to determine where in `recvbuf` the data sent by each process will be stored.

33. Scatter/Gather data from all members to all members of a group
int MPI_Alltoall( void *sendbuf, int sendcount,
    MPI_Datatype sendtype,
    void *recvbuf, int recvcount,
    MPI_Datatype recvtype,
    MPI_Comm comm )

Each process sends distinct data \texttt{sendbuf} to all processes of a group (included itself). Each process sends to the \textit{i}-th process \texttt{sendcount} elements with data type \texttt{sendtype}, starting from the element number \textit{i}*\texttt{sendcount} of the table \texttt{sendbuf}. The received data are placed in the table \texttt{recvbuf}. The block of data (\texttt{recvcount} – number of elements, \texttt{recvtype} – data type) sent from process \textit{i}-th is placed in the table \texttt{recvbuf} starting from the element number \textit{i}*\texttt{recvcount}. All processes in the domain within the communicator \texttt{comm} must be call with the same values of arguments: \texttt{sendcount}, \texttt{sendtype}, \texttt{recvcount}, \texttt{recvtype}.

![Diagram]

34. Global reduction operations

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

This function combines the elements provided in the input buffer \texttt{sendbuf} (\texttt{count} – number of elements, \texttt{datatype} – data type) of each process in the group, using the operation \texttt{op}:

\texttt{op} \in \{ \texttt{MPI\_MAX}, \texttt{MPI\_MIN}, \texttt{MPI\_SUM}, \texttt{MPI\_PROD}, \texttt{MPI\_MAXLOC}, \texttt{MPI\_MINLOC}, \texttt{MPI\_LAND}, \texttt{MPI\_BAND}, \texttt{MPI\_LOR}, \texttt{MPI\_BOR}, \texttt{MPI\_LXOR}, \texttt{MPI\_BXOR} \}.

The results are returned in the output buffer \texttt{recvbuf} (\texttt{count} – number of elements, \texttt{datatype} – data type) of the process with rank \texttt{root}. Both, input and output buffers have the same number of elements, with the same type. The routine is called by all group members using the same arguments.

The operations are performed independently for each element from \texttt{sendbuf}. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is \texttt{MPI\_MAX} and the send buffer contains \text{n} elements, the \textit{i}-th element of \texttt{recvbuf} will be the
maximal value of \( i \)-th elements of all buffers \texttt{sendbuf} of all processes in the considered group.

\texttt{MPI\_Allreduce(...) -} variant of the reduce operation where the result is returned to all processes in the group.