Dan’s Small MPI Reference

abridged from:
https://computing.llnl.gov/tutorials/mpi/

Environment Management Routines

This group of routines is used for interrogating and setting the MPI execution environment, and covers an assortment of purposes, such as initializing and terminating the MPI environment, querying a rank's identity, querying the MPI library's version, etc. Most of the commonly used ones are described below.

MPI_Init

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

```
MPI_Init (&argc,&argv)
MPI_INIT (ierr)
```

MPI_Comm_size

Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

```
MPI_Comm_size (comm,&size)
MPI_COMM_SIZE (comm,size,ierr)
```

MPI_Comm_rank

Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

```
MPI_Comm_rank (comm,&rank)
MPI_COMM_RANK (comm,rank,ierr)
```

MPI_Abort

Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

```
MPI_Abort (comm,errorcode)
MPI_ABORT (comm,errorcode,ierr)
```

MPI_Get_processor_name

Returns the processor name. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands.

```
MPI_Get_processor_name (&name,&resultlength)
MPI_GET_PROCESSOR_NAME (name,resultlength,ierr)
```

MPI_Get_version

Returns the version and subversion of the MPI standard that's implemented by the library.

```
MPI_Get_version (&version,&subversion)
MPI_GET_VERSION (version,subversion,ierr)
```

MPI_Initialized

Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary. MPI_Initialized solves this problem.

```
MPI_Initialized (&flag)
MPI_INITIALIZED (flag,ierr)
```

MPI_Wtime

Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

```
MPI_Wtime ()
MPI_WTIME ()
```

MPI_Wtick

Returns the resolution in seconds (double precision) of MPI_Wtime.

```
MPI_Wtick ()
MPI_WTICK ()
```

MPI_Finalize

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

```
MPI_Finalize ()
MPI_FINALIZE (ierr)
```

Point to Point Communication Routines

MPI Message Passing Routine Arguments

MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

Buffer

Program (application) address space that references the data that is to be sent or received. In most cases, this is simply the variable name that is be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: `&var1`

Data Count

Indicates the number of data elements of a particular type to be sent.

Data Type

For reasons of portability, MPI predefines its elementary data types. The table below lists those required by the standard.

C Data Types		Fortran Data Types	
`MPI_CHAR`	char	`MPI_CHARACTER`	character(1)
`MPI_WCHAR`	wchar_t - wide character		
`MPI_SHORT`	signed short int		

`MPI_INT`	signed int	`MPI_INTEGER`
		`MPI_INTEGER1`
		`MPI_INTEGER2`
		`MPI_INTEGER4`
			integer
			integer*1
			integer*2
			integer*4

MPI_LONG	signed long int		
MPI_LONG_LONG_INT **MPI_LONG_LONG**	signed long long int		
MPI_SIGNED_CHAR	signed char		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long int		
MPI_UNSIGNED_LONG_LONG	unsigned long long int		
MPI_FLOAT	float	**MPI_REAL** **MPI_REAL2** **MPI_REAL4** **MPI_REAL8**	real real*2 real*4 real*8
MPI_DOUBLE	double	**MPI_DOUBLE_PRECISION**	double precision
MPI_LONG_DOUBLE	long double		
MPI_C_COMPLEX **MPI_C_FLOAT_COMPLEX**	float _Complex	**MPI_COMPLEX**	complex
MPI_C_DOUBLE_COMPLEX	double _Complex	**MPI_DOUBLE_COMPLEX**	double complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex		
MPI_C_BOOL	_Bool	**MPI_LOGICAL**	logical
MPI_INT8_T **MPI_INT16_T** **MPI_INT32_T** **MPI_INT64_T**	int8_t int16_t int32_t int64_t		

MPI_UINT8_T **MPI_UINT16_T** **MPI_UINT32_T** **MPI_UINT64_T**	uint8_t uint16_t uint32_t uint64_t		
MPI_BYTE	8 binary digits	**MPI_BYTE**	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	**MPI_PACKED**	data packed or unpacked with MPI_Pack()/ MPI_Unpack

Notes:

- Programmers may also create their own data types (see Derived Data Types).
- MPI_BYTE and MPI_PACKED do not correspond to standard C or Fortran types.
- Types shown in **GRAY FONT** are recommended if possible.
- Some implementations may include additional elementary data types (MPI_LOGICAL2, MPI_COMPLEX32, etc.). Check the MPI header file.

Destination

An argument to send routines that indicates the process where a message should be delivered. Specified as the rank of the receiving process.

Source

An argument to receive routines that indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card MPI_ANY_SOURCE to receive a message from any task.

Tag

Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag. The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.

Communicator

Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator MPI_COMM_WORLD is usually used.

Status

For a receive operation, indicates the source of the message and the tag of the message. In C, this argument is a pointer to a predefined structure MPI_Status (ex. **`stat.MPI_SOURCE stat.MPI_TAG`**). In Fortran, it is an integer array of size MPI_STATUS_SIZE (ex. **`stat(MPI_SOURCE) stat(MPI_TAG)`**). Additionally, the actual number of bytes received is obtainable from Status via the MPI_Get_count routine. The constants MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE can be substituted if a message's source, tag or size will be be queried later.

Request

Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation. In C, this argument is a pointer to a predefined structure MPI_Request. In Fortran, it is an integer.

Blocking Message Passing Routines

The more commonly used MPI blocking message passing routines are described below.

MPI_Send

Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. Note that this routine may be implemented differently on different systems. The MPI standard permits the use of a system buffer but does not require it. Some implementations may actually use a synchronous send (discussed below) to implement the basic blocking send.

```
MPI_Send (&buf,count,datatype,dest,tag,comm)
MPI_SEND (buf,count,datatype,dest,tag,comm,ierr)
```

MPI_Recv

Receive a message and block until the requested data is available in the application buffer in the receiving task.

```
MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
MPI_RECV
(buf,count,datatype,source,tag,comm,status,ierr)
```

MPI_Ssend

Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

```
MPI_Ssend (&buf,count,datatype,dest,tag,comm)
MPI_SSEND (buf,count,datatype,dest,tag,comm,ierr)
```

MPI_Sendrecv

Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.

```
MPI_Sendrecv
(&sendbuf,sendcount,sendtype,dest,sendtag,
       &recvbuf,recvcount,recvtype,source,recvtag,
       comm,&status)
MPI_SENDRECV (sendbuf,sendcount,sendtype,dest,sendtag,
       recvbuf,recvcount,recvtype,source,recvtag,
       comm,status,ierr)
```

MPI_Wait
MPI_Waitany
MPI_Waitall
MPI_Waitsome

MPI_Wait blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify any, all or some completions.

```
MPI_Wait (&request,&status)
MPI_Waitany (count,&array_of_requests,&index,&status)
MPI_Waitall
(count,&array_of_requests,&array_of_statuses)
MPI_Waitsome (incount,&array_of_requests,&outcount,
       &array_of_offsets, &array_of_statuses)
MPI_WAIT (request,status,ierr)
MPI_WAITANY
(count,array_of_requests,index,status,ierr)
MPI_WAITALL
(count,array_of_requests,array_of_statuses,
       ierr)
MPI_WAITSOME (incount,array_of_requests,outcount,
       array_of_offsets, array_of_statuses,ierr)
```

MPI_Probe

Performs a blocking test for a message. The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag. For the C routine, the actual source and tag will be returned in the status structure as **status.MPI_SOURCE** and **status.MPI_TAG**. For the Fortran routine, they will be returned in the integer array **status(MPI_SOURCE)** and **status(MPI_TAG)**.

```
MPI_Probe (source,tag,comm,&status)
MPI_PROBE (source,tag,comm,status,ierr)
```

MPI_Get_count

Returns the source, tag and number of elements of datatype received. Can be used with both blocking and non-blocking receive operations. For the C routine, the actual source and tag will be returned in the status structure as **status.MPI_SOURCE** and **status.MPI_TAG**. For the Fortran routine, they will be returned in the integer array **status(MPI_SOURCE)** and **status(MPI_TAG)**.

```
MPI_Get_count (&status,datatype,&count)
MPI_GET_COUNT (status,datatype,count,ierr)
```

Non-blocking Message Passing Routines

The more commonly used MPI non-blocking message passing routines are described below.

MPI_Isend

Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the non-blocking send has completed.

```
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
MPI_ISEND
(buf,count,datatype,dest,tag,comm,request,ierr)
```

MPI_Irecv

Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to MPI_Wait or MPI_Test to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

```
MPI_Irecv
(&buf,count,datatype,source,tag,comm,&request)
MPI_IRECV
(buf,count,datatype,source,tag,comm,request,ierr)
```

MPI_Issend

Non-blocking synchronous send. Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.

```
MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
MPI_ISSEND
(buf,count,datatype,dest,tag,comm,request,ierr)
```

MPI_Test
MPI_Testany
MPI_Testall
MPI_Testsome

MPI_Test checks the status of a specified non-blocking send or receive operation. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.

```
MPI_Test (&request,&flag,&status)
MPI_Testany
(count,&array_of_requests,&index,&flag,&status)
MPI_Testall
(count,&array_of_requests,&flag,&array_of_statuses)
MPI_Testsome (incount,&array_of_requests,&outcount,
```

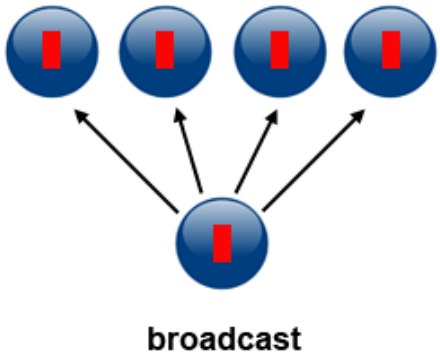
```
       &array_of_offsets, &array_of_statuses)
MPI_TEST (request,flag,status,ierr)
MPI_TESTANY
(count,array_of_requests,index,flag,status,ierr)
MPI_TESTALL
(count,array_of_requests,flag,array_of_statuses,ierr)
MPI_TESTSOME (incount,array_of_requests,outcount,
       array_of_offsets, array_of_statuses,ierr)
```

MPI_Iprobe

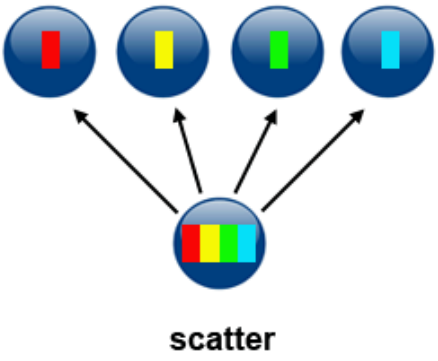
Performs a non-blocking test for a message. The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag. The integer "flag" parameter is returned logical true (1) if a message has arrived, and logical false (0) if not. For the C routine, the actual source and tag will be returned in the status structure as **status.MPI_SOURCE** and **status.MPI_TAG**. For the Fortran routine, they will be returned in the integer array **status(MPI_SOURCE)** and **status(MPI_TAG)**.

```
MPI_Iprobe (source,tag,comm,&flag,&status)
MPI_IPROBE (source,tag,comm,flag,status,ierr)
```

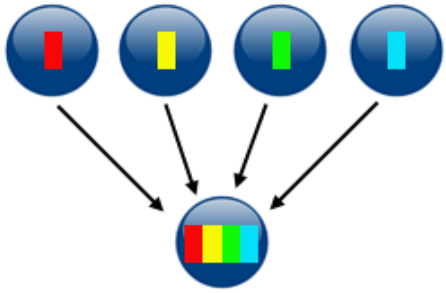
Collective Communication Routines



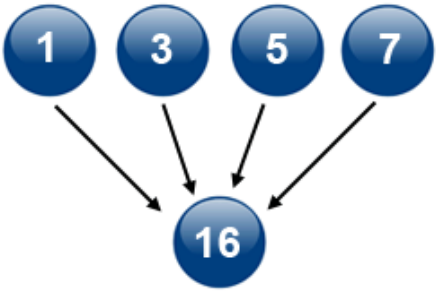
broadcast



scatter



gather



reduction

Types of Collective Operations:

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.
- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective Computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

Scope:

- Collective communication routines must involve **all** processes within the scope of a communicator.
  - All processes are by default, members in the communicator MPI_COMM_WORLD.
  - Additional communicators can be defined by the programmer. See the Group and Communicator Management Routines section for details.

- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Programming Considerations and Restrictions:

- Collective communication routines do not take message tag arguments.
- Collective operations within subsets of processes are accomplished by first partitioning the subsets into new groups and then attaching the new groups to new communicators (discussed in the Group and Communicator Management Routines section).
- Can only be used with MPI predefined datatypes - not with MPI Derived Data Types.
- MPI-2 extended most collective operations to allow data movement between intercommunicators (not covered here).
- With MPI-3, collective operations can be blocking or non-blocking. Only blocking operations are covered in this tutorial.

Collective Communication Routines

MPI_Barrier

Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed.

```
MPI_Barrier (comm)
MPI_BARRIER (comm,ierr)
```

MPI_Bcast

Data movement operation. Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

```
MPI_Bcast (&buffer,count,datatype,root,comm)
MPI_BCAST (buffer,count,datatype,root,comm,ierr)
```

MPI_Scatter

Data movement operation. Distributes distinct messages from a single source task to each task in the group.

```
MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,
       recvcnt,recvtype,root,comm)
MPI_SCATTER (sendbuf,sendcnt,sendtype,recvbuf,
       recvcnt,recvtype,root,comm,ierr)
```

MPI_Gather

Data movement operation. Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

```
MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,
       recvcount,recvtype,root,comm)
MPI_GATHER (sendbuf,sendcnt,sendtype,recvbuf,
       recvcount,recvtype,root,comm,ierr)
```

MPI_Allgather

Data movement operation. Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.

```
MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf,
       recvcount,recvtype,comm)
MPI_ALLGATHER (sendbuf,sendcount,sendtype,recvbuf,
       recvcount,recvtype,comm,info)
```

MPI_Reduce

Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task.

```
MPI_Reduce
(&sendbuf,&recvbuf,count,datatype,op,root,comm)
MPI_REDUCE
(sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```

The predefined MPI reduction operations appear below. Users can also define their own reduction functions by using the MPI_Op_create routine.

MPI Reduction Operation		C Data Types	Fortran Data Type
MPI_MAX	maximum	integer, float	integer, real, complex
MPI_MIN	minimum	integer, float	integer, real, complex
MPI_SUM	sum	integer, float	integer, real, complex
MPI_PROD	product	integer, float	integer, real, complex
MPI_LAND	logical AND	integer	logical
MPI_BAND	bit-wise AND	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	logical OR	integer	logical
MPI_BOR	bit-wise OR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer	logical
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double	real, complex,double precision
MPI_MINLOC	min value and location	float, double and long double	real, complex, double precision

- Note from the MPI_Reduce man page: The operation is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition. [Advice to implementors] It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors. [End of advice to implementors]

MPI_Allreduce

Collective computation operation + data movement. Applies a reduction operation and places the result in all tasks in the group. This is equivalent to an MPI_Reduce followed by an MPI_Bcast.

```
MPI_Allreduce
(&sendbuf,&recvbuf,count,datatype,op,comm)
MPI_ALLREDUCE
(sendbuf,recvbuf,count,datatype,op,comm,ierr)
```

MPI_Reduce_scatter

Collective computation operation + data movement. First does an element-wise reduction on a vector across all tasks in the group. Next, the result vector is split into disjoint segments and distributed across the tasks. This is equivalent to an MPI_Reduce followed by an MPI_Scatter operation.

```
MPI_Reduce_scatter
(&sendbuf,&recvbuf,recvcount,datatype,
        op,comm)
MPI_REDUCE_SCATTER (sendbuf,recvbuf,recvcount,datatype,
        op,comm,ierr)
```

MPI_Alltoall

Data movement operation. Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.

```
MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf,
        recvcnt,recvtype,comm)
MPI_ALLTOALL (sendbuf,sendcount,sendtype,recvbuf,
        recvcnt,recvtype,comm,ierr)
```

MPI_Scan

Performs a scan operation with respect to a reduction operation across a task group.

```
MPI_Scan (&sendbuf,&recvbuf,count,datatype,op,comm)
MPI_SCAN (sendbuf,recvbuf,count,datatype,op,comm,ierr)
```