**MPI\_Allgather**

Gathers data from all tasks and distribute the combined data to all tasks

**int MPI\_Allgather(**

**void** \**sendbuf***,**

**int** *sendcount***,**

**MPI\_Datatype** *sendtype***,**

**void** \**recvbuf***,**

**int** *recvcount***,**

**MPI\_Datatype** *recvtype***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*sendbuf*

[in] starting address of send buffer (choice)

*sendcount*

[in] number of elements in send buffer (integer)

*sendtype*

[in] data type of send buffer elements (handle)

*recvbuf*

[out] address of receive buffer (choice)

*recvcount*

[in] number of elements received from any process (integer)

*recvtype*

[in] data type of receive buffer elements (handle)

*comm*

[in] communicator (handle)

**Remarks**

The MPI standard (1.0 and 1.1) says that

    The jth block of data sent from each proess is received by every process and placed in the jth block of the buffer recvbuf.

This is misleading; a better description is

    The block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf.

MPI\_ALLGATHER can be thought of as MPI\_GATHER, but where all processes receive the result, instead of just the root. The jth block of data sent from each process is received by every process and placed in the jth block of the buffer recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process.

The outcome of a call to MPI\_ALLGATHER(...) is as if all processes executed n calls to

MPI\_GATHER(sendbuf,sendcount,sendtype,recvbuf,recvcount, recvtype,root,comm),

for root = 0 , ..., n-1. The rules for correct usage of MPI\_ALLGATHER are easily found from the corresponding rules for MPI\_GATHER.

The "in place" option for intracommunicators is specified by passing the value MPI\_IN\_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer. Specifically, the outcome of a call to MPI\_ALLGATHER in the "in place" case is as if all processes executed *n* calls to

MPI\_GATHER( MPI\_IN\_PLACE, 0, MPI\_DATATYPE\_NULL, recvbuf, recvcount, recvtype, root, comm )

for root = 0, ..., n - 1.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

*Advice to users.*

The communication pattern of MPI\_ALLGATHER executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group A and the arguments recvcount, recvtype in group B), need not equal the number of items sent by processes in group B (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction by specifying sendcount = 0 for the communication in the reverse direction.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Allgather](http://mpi.deino.net/mpi_functions/MPI_Allgather.html).

#include "mpi.h"  
#include <stdio.h>  
  
#define MAX\_PROCESSES 10  
  
int main( int argc, char \*\*argv )  
{  
    int rank, size, i,j;  
    int table[MAX\_PROCESSES][MAX\_PROCESSES];  
    int errors=0;  
    int participants;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( MPI\_COMM\_WORLD, &rank );  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( MPI\_COMM\_WORLD, &size );  
    /\* Exactly MAX\_PROCESSES processes can participate \*/  
    if ( size >= MAX\_PROCESSES ) participants = MAX\_PROCESSES;  
    else  
    {  
        fprintf( stderr, "Number of processors must be at least %d\n", MAX\_PROCESSES );fflush(stderr);  
        [MPI\_Abort](http://mpi.deino.net/mpi_functions/MPI_Abort.html)( MPI\_COMM\_WORLD, 1 );  
    }  
    if ( (rank < participants) ) {  
        /\* Determine what rows are my responsibility \*/  
        int block\_size = MAX\_PROCESSES / participants;  
        int begin\_row = rank \* block\_size;  
        int end\_row = (rank+1) \* block\_size;  
        int send\_count = block\_size \* MAX\_PROCESSES;  
        int recv\_count = send\_count;  
        /\* Paint my rows my color \*/  
        for (i=begin\_row; i<end\_row ;i++)  
            for (j=0; j<MAX\_PROCESSES; j++)  
                table[i][j] = rank + 10;  
        /\* Everybody gets the gathered table \*/  
        [MPI\_Allgather](http://mpi.deino.net/mpi_functions/MPI_Allgather.html)(&table[begin\_row][0], send\_count, MPI\_INT,   
                           &table[0][0], recv\_count, MPI\_INT, MPI\_COMM\_WORLD);  
        /\* Everybody should have the same table now, \*/  
        for (i=0; i<MAX\_PROCESSES;i++) {  
            if ( (table[i][0] - table[i][MAX\_PROCESSES-1] !=0) )   
                errors++;  
        }  
    }   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    if (errors)  
    {  
        printf( "[%d] done with ERRORS(%d)!\n", rank, errors );fflush(stdout);  
    }  
    return errors;  
}

**MPI\_Allreduce**

Combines values from all processes and distributes the result back to all processes

**int MPI\_Allreduce(**

**void** \**sendbuf***,**

**void** \**recvbuf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**MPI\_Op** *op***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*sendbuf*

[in] starting address of send buffer (choice)

*recvbuf*

[out] starting address of receive buffer (choice)

*count*

[in] number of elements in send buffer (integer)

*datatype*

[in] data type of elements of send buffer (handle)

*op*

[in] operation (handle)

*comm*

[in] communicator (handle)

**Remarks**

MPI includes variants of each of the reduce operations where the result is returned to all processes in the group. MPI requires that all processes participating in these operations receive identical results.

Same as MPI\_REDUCE except that the result appears in the receive buffer of all the group members.

The "in place" option for intracommunicators is specified by passing the value MPI\_IN\_PLACE to the argument sendbuf at the root. In such case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide the same count value.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Notes on collective operations**

The reduction functions (MPI\_Op) do not return an error value. As a result, if the functions detect an error, all they can do is either call MPI\_Abort or silently skip the problem. Thus, if you change the error handler from MPI\_ERRORS\_ARE\_FATAL to something else, for example, MPI\_ERRORS\_RETURN, then no error may be indicated.

The reason for this is the performance problems in ensuring that all collective routines return the same error value.

**Example Code**

The following sample code illustrates [MPI\_Allreduce](http://mpi.deino.net/mpi_functions/MPI_Allreduce.html).

#include "mpi.h"  
#include <stdio.h>  
  
int main(int argc, char \*argv[])  
{  
    int count = 1000;  
    int \*in, \*out, \*sol;  
    int i, fnderr=0;  
    int rank, size;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc, &argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &size);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &rank);  
    in = (int \*)malloc( count \* sizeof(int) );  
    out = (int \*)malloc( count \* sizeof(int) );  
    sol = (int \*)malloc( count \* sizeof(int) );  
    for (i=0; i<count; i++)  
    {  
        \*(in + i) = i;  
        \*(sol + i) = i\*size;  
        \*(out + i) = 0;  
    }  
    [MPI\_Allreduce](http://mpi.deino.net/mpi_functions/MPI_Allreduce.html)( in, out, count, MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD );  
    for (i=0; i<count; i++)  
    {  
        if (\*(out + i) != \*(sol + i))  
        {  
            fnderr++;  
        }  
    }  
    if (fnderr)  
    {  
        fprintf( stderr, "(%d) Error for type MPI\_INT and op MPI\_SUM\n", rank );  
        fflush(stderr);  
    }  
    free( in );  
    free( out );  
    free( sol );  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return fnderr;  
}

**MPI\_Alltoall**

Sends data from all to all processes

**int MPI\_Alltoall(**

**void** \**sendbuf***,**

**int** *sendcount***,**

**MPI\_Datatype** *sendtype***,**

**void** \**recvbuf***,**

**int** *recvcount***,**

**MPI\_Datatype** *recvtype***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*sendbuf*

[in] starting address of send buffer (choice)

*sendcount*

[in] number of elements to send to each process (integer)

*sendtype*

[in] data type of send buffer elements (handle)

*recvbuf*

[out] address of receive buffer (choice)

*recvcount*

[in] number of elements received from any process (integer)

*recvtype*

[in] data type of receive buffer elements (handle)

*comm*

[in] communicator (handle)

**Remarks**

MPI\_ALLTOALL is an extension of MPI\_ALLGATHER to the case where each process sends distinct data to each of the receivers. The jth block sent from process i is received by process j and is placed in the ith block of recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

The outcome is as if each process executed a send to each process (itself included) with a call to,

http://mpi.deino.net/mpi_functions/img119.gif

and a receive from every other process with a call to,

http://mpi.deino.net/mpi_functions/img120.gif

All arguments on all processes are significant. The argument comm must have identical values on all processes.

No "in place" option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The *j*-th send buffer of process *i* in group A should be consistent with the *i*-th receive buffer of process *j* in group B, and vice versa.

*Advice to users.*

When all-to-all is executed on an intercommunication domain, then the number of data items sent from processes in group A to processes in group B need not equal the number of items sent in the reverse direction. In particular, one can have unidirectional communication by specifying sendcount = 0 in the reverse direction.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Alltoall](http://mpi.deino.net/mpi_functions/MPI_Alltoall.html).

#include "mpi.h"  
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
#include <errno.h>  
  
#ifndef EXIT\_SUCCESS  
#define EXIT\_SUCCESS 0  
#define EXIT\_FAILURE 1  
#endif  
  
int main( int argc, char \*argv[] )  
{  
    int rank, size;  
    int chunk = 128;  
    int i;  
    int \*sb;  
    int \*rb;  
    int status, gstatus;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD,&rank);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD,&size);  
    for ( i=1 ; i < argc ; ++i ) {  
        if ( argv[i][0] != '-' )  
            continue;  
        switch(argv[i][1]) {  
            case 'm':  
                chunk = atoi(argv[++i]);  
                break;  
            default:  
                fprintf(stderr, "Unrecognized argument %s\n", argv[i]);fflush(stderr);  
                [MPI\_Abort](http://mpi.deino.net/mpi_functions/MPI_Abort.html)(MPI\_COMM\_WORLD,EXIT\_FAILURE);  
        }  
    }  
    sb = (int \*)malloc(size\*chunk\*sizeof(int));  
    if ( !sb ) {  
        perror( "can't allocate send buffer" );fflush(stderr);  
        [MPI\_Abort](http://mpi.deino.net/mpi_functions/MPI_Abort.html)(MPI\_COMM\_WORLD,EXIT\_FAILURE);  
    }  
    rb = (int \*)malloc(size\*chunk\*sizeof(int));  
    if ( !rb ) {  
        perror( "can't allocate recv buffer");fflush(stderr);  
        free(sb);  
        [MPI\_Abort](http://mpi.deino.net/mpi_functions/MPI_Abort.html)(MPI\_COMM\_WORLD, EXIT\_FAILURE);  
    }  
    for ( i=0 ; i < size\*chunk ; ++i ) {  
        sb[i] = rank + 1;  
        rb[i] = 0;  
    }  
    status = [MPI\_Alltoall](http://mpi.deino.net/mpi_functions/MPI_Alltoall.html)(sb, chunk, MPI\_INT, rb, chunk, MPI\_INT, MPI\_COMM\_WORLD);  
    [MPI\_Allreduce](http://mpi.deino.net/mpi_functions/MPI_Allreduce.html)( &status, &gstatus, 1, MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD );  
    if (rank == 0) {  
        if (gstatus != 0) {  
            printf("all\_to\_all returned %d\n",gstatus);fflush(stdout);  
        }  
    }  
    free(sb);  
    free(rb);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return(EXIT\_SUCCESS);  
}

**MPI\_Barrier**

Blocks until all processes in the communicator have reached this routine.

**int MPI\_Barrier(**

**MPI\_Comm** *comm*

**);**

**Input Parameter**

*comm*

[in] communicator (handle)

**Remarks**

Blocks the caller until all processes in the communicator have called it; that is, the call returns at any process only after all members of the communicator have entered the call.

For MPI-2, comm may be an intercommunicator or an intracommunicator. If comm is an intercommunicator, the barrier is performed across all processes in the intercommunicator. In this case, all processes in the local group of the intercommunicator may exit the barrier when all of the processes in the remote group have entered the barrier.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Barrier](http://mpi.deino.net/mpi_functions/MPI_Barrier.html).

#include "mpi.h"   
#include <stdio.h>   
int main(int argc, char \*argv[])   
{   
    int rank, nprocs;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);   
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD,&nprocs);   
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD,&rank);   
    [MPI\_Barrier](http://mpi.deino.net/mpi_functions/MPI_Barrier.html)(MPI\_COMM\_WORLD);  
    printf("Hello, world.  I am %d of %d\n", rank, nprocs);fflush(stdout);   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();   
    return 0;   
}

**MPI\_Bcast**

Broadcasts a message from the process with rank "root" to all other processes of the communicator

**int MPI\_Bcast(**

**void** \**buffer***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *root***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*buffer*

[in/out] starting address of buffer (choice)

*count*

[in] number of entries in buffer (integer)

*datatype*

[in] data type of buffer (handle)

*root*

[in] rank of broadcast root (integer)

*comm*

[in] communicator (handle)

**Remarks**

MPI\_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of group using the same arguments for comm, root. On return, the contents of root's communication buffer has been copied to all processes.

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI\_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful here.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI\_ROOT in root. All other processes in group A pass the value MPI\_PROC\_NULL in root. Data is broadcast from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Bcast](http://mpi.deino.net/mpi_functions/MPI_Bcast.html).

#include "mpi.h"  
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
  
#define ROOT 0  
#define NUM\_REPS 5  
#define NUM\_SIZES 3  
  
int main( int argc, char \*\*argv)  
{  
    int \*buf;  
    int i, rank, reps, n;  
    int bVerify = 1;  
    int sizes[NUM\_SIZES] = { 100, 64\*1024, 128\*1024 };  
    int num\_errors=0, tot\_errors;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc, &argv);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &rank);  
    if (argc > 1)  
    {  
        if (strcmp(argv[1], "-novalidate") == 0 || strcmp(argv[1], "-noverify") == 0)  
            bVerify = 0;  
    }  
    buf = (int \*) malloc(sizes[NUM\_SIZES-1]\*sizeof(int));  
    memset(buf, 0, sizes[NUM\_SIZES-1]\*sizeof(int));  
    for (n=0; n<NUM\_SIZES; n++)  
    {  
        if (rank == ROOT)  
        {  
            printf("bcasting %d MPI\_INTs %d times\n", sizes[n], NUM\_REPS);  
            fflush(stdout);  
        }  
        for (reps=0; reps < NUM\_REPS; reps++)  
        {  
            if (bVerify)  
            {  
                if (rank == ROOT)  
                {  
                    for (i=0; i<sizes[n]; i++)  
                    {  
                        buf[i] = 1000000 \* (n \* NUM\_REPS + reps) + i;  
                    }  
                }  
                else  
                {  
                    for (i=0; i<sizes[n]; i++)  
                    {  
                        buf[i] = -1 - (n \* NUM\_REPS + reps);  
                    }  
                }  
            }  
            [MPI\_Bcast](http://mpi.deino.net/mpi_functions/MPI_Bcast.html)(buf, sizes[n], MPI\_INT, ROOT, MPI\_COMM\_WORLD);  
            if (bVerify)  
            {  
                num\_errors = 0;  
                for (i=0; i<sizes[n]; i++)  
                {  
                    if (buf[i] != 1000000 \* (n \* NUM\_REPS + reps) + i)  
                    {  
                        num\_errors++;  
                        if (num\_errors < 10)  
                        {  
                            printf("Error: Rank=%d, n=%d, reps=%d, i=%d, buf[i]=%d expected=%d\n", rank, n, reps, i, buf[i],  
                                    1000000 \* (n \* NUM\_REPS + reps) +i);  
                            fflush(stdout);  
                        }  
                    }  
                }  
                if (num\_errors >= 10)  
                {  
                    printf("Error: Rank=%d, num\_errors = %d\n", rank, num\_errors);  
                    fflush(stdout);  
                }  
            }  
        }  
    }  
  
    [MPI\_Reduce](http://mpi.deino.net/mpi_functions/MPI_Reduce.html)( &num\_errors, &tot\_errors, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD );  
    if (rank == 0 && tot\_errors == 0)   
        printf(" No Errors\n");  
    fflush(stdout);  
    free(buf);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Bsend**

Basic send with user-provided buffering

**int MPI\_Bsend(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements in send buffer (nonnegative integer)

*datatype*

[in] datatype of each send buffer element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

**Remarks**

This send is provided as a convenience function; it allows the user to send messages without worring about where they are buffered (because the user *must* have provided buffer space with [MPI\_Buffer\_attach](http://mpi.deino.net/mpi_functions/MPI_Buffer_attach.html)).

In deciding how much buffer space to allocate, remember that the buffer space is not available for reuse by subsequent [MPI\_Bsend](http://mpi.deino.net/mpi_functions/MPI_Bsend.html)s unless you are certain that the message has been received (not just that it should have been received). For example, this code does not allocate enough buffer space

[MPI\_Buffer\_attach](http://mpi.deino.net/mpi_functions/MPI_Buffer_attach.html)( b, n\*sizeof(double) + MPI\_BSEND\_OVERHEAD );

for (i=0; i<m; i++) {

[MPI\_Bsend](http://mpi.deino.net/mpi_functions/MPI_Bsend.html)( buf, n, MPI\_DOUBLE, ... );

}

because only enough buffer space is provided for a single send, and the loop may start a second [MPI\_Bsend](http://mpi.deino.net/mpi_functions/MPI_Bsend.html) before the first is done making use of the buffer.

In C, you can force the messages to be delivered by

[MPI\_Buffer\_detach](http://mpi.deino.net/mpi_functions/MPI_Buffer_detach.html)( &b, &n );

[MPI\_Buffer\_attach](http://mpi.deino.net/mpi_functions/MPI_Buffer_attach.html)( b, n );

(The [MPI\_Buffer\_detach](http://mpi.deino.net/mpi_functions/MPI_Buffer_detach.html) will not complete until all buffered messages are delivered.)

A **buffered** mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is **local**, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. An error will occur if there is insufficient buffer space. The amount of available buffer space is controlled by the user. Buffer allocation by the user may be required for the buffered mode to be effective.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Bsend](http://mpi.deino.net/mpi_functions/MPI_Bsend.html).

#include "mpi.h"  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
/\*   
\* This is a simple program that tests bsend. It may be run as a single  
\* process; in addition, bsend allows send-to-self programs.  
\*/  
int main( int argc, char \*argv[] )  
{  
    MPI\_Comm comm = MPI\_COMM\_WORLD;  
    int dest = 0, src = 0, tag = 1;  
    int s1, s2, s3;  
    char \*buf, \*bbuf;  
    char msg1[7], msg3[17];  
    double msg2[2];  
    char rmsg1[64], rmsg3[64];  
    double rmsg2[64];  
    int errs = 0, rank;  
    int bufsize, bsize;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( MPI\_COMM\_WORLD, &rank );  
  
    /\* According to the standard, we must use the PACK\_SIZE length of each  
        message in the computation of the message buffer size \*/  
    [MPI\_Pack\_size](http://mpi.deino.net/mpi_functions/MPI_Pack_size.html)( 7, MPI\_CHAR, comm, &s1 );  
    [MPI\_Pack\_size](http://mpi.deino.net/mpi_functions/MPI_Pack_size.html)( 2, MPI\_DOUBLE, comm, &s2 );  
    [MPI\_Pack\_size](http://mpi.deino.net/mpi_functions/MPI_Pack_size.html)( 17, MPI\_CHAR, comm, &s3 );  
    bufsize = 3 \* MPI\_BSEND\_OVERHEAD + s1 + s2 + s3;  
    buf = (char \*)malloc( bufsize );  
    [MPI\_Buffer\_attach](http://mpi.deino.net/mpi_functions/MPI_Buffer_attach.html)( buf, bufsize );  
    strncpy( msg1, "012345", 7 );  
    strncpy( msg3, "0123401234012341", 17 );  
    msg2[0] = 1.23; msg2[1] = 3.21;  
  
    if (rank == src) {  
        /\* These message sizes are chosen to expose any alignment problems \*/  
        [MPI\_Bsend](http://mpi.deino.net/mpi_functions/MPI_Bsend.html)( msg1, 7, MPI\_CHAR, dest, tag, comm );  
        [MPI\_Bsend](http://mpi.deino.net/mpi_functions/MPI_Bsend.html)( msg2, 2, MPI\_DOUBLE, dest, tag, comm );  
        [MPI\_Bsend](http://mpi.deino.net/mpi_functions/MPI_Bsend.html)( msg3, 17, MPI\_CHAR, dest, tag, comm );  
    }  
  
    if (rank == dest) {  
        [MPI\_Recv](http://mpi.deino.net/mpi_functions/MPI_Recv.html)( rmsg1, 7, MPI\_CHAR, src, tag, comm, MPI\_STATUS\_IGNORE );  
        [MPI\_Recv](http://mpi.deino.net/mpi_functions/MPI_Recv.html)( rmsg2, 10, MPI\_DOUBLE, src, tag, comm, MPI\_STATUS\_IGNORE );  
        [MPI\_Recv](http://mpi.deino.net/mpi_functions/MPI_Recv.html)( rmsg3, 17, MPI\_CHAR, src, tag, comm, MPI\_STATUS\_IGNORE );  
        if (strcmp( rmsg1, msg1 ) != 0) {  
            errs++;  
            fprintf( stderr, "message 1 (%s) should be %s\n", rmsg1, msg1 );fflush(stderr);  
        }  
        if (rmsg2[0] != msg2[0] || rmsg2[1] != msg2[1]) {  
            errs++;  
            fprintf( stderr,   
                        "message 2 incorrect, values are (%f,%f) but should be (%f,%f)\n",  
                        rmsg2[0], rmsg2[1], msg2[0], msg2[1] );fflush(stderr);  
        }  
        if (strcmp( rmsg3, msg3 ) != 0) {  
            errs++;  
            fprintf( stderr, "message 3 (%s) should be %s\n", rmsg3, msg3 );fflush(stderr);  
        }  
    }  
  
    /\* We can't guarantee that messages arrive until the detach \*/  
    [MPI\_Buffer\_detach](http://mpi.deino.net/mpi_functions/MPI_Buffer_detach.html)( &bbuf, &bsize );  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)( errs );  
  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Comm\_create**

Creates a new communicator

**int MPI\_Comm\_create(**

**MPI\_Comm** *comm***,**

**MPI\_Group** *group***,**

**MPI\_Comm** \**newcomm*

**);**

**Parameters**

*comm*

[in] communicator (handle)

*group*

[in] group, which is a subset of the group of comm (handle)

*newcomm*

[out] new communicator (handle)

**Remarks**

This function creates a new communicator newcomm with communication group defined by group and a new context. No cached information propagates from comm to newcomm. The function returns MPI\_COMM\_NULL to processes that are not in group. The call is erroneous if not all group arguments have the same value, or if group is not a subset of the group associated with comm. Note that the call is to be executed by all processes in comm, even if they do not belong to the new group. This call applies only to intra-communicators.

*Rationale.*

The requirement that the entire group of comm participate in the call stems from the following considerations:

* It allows the implementation to layer MPI\_COMM\_CREATE on top of regular collective communications.
* It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.
* It permits implementations sometimes to avoid communication related to context creation.

*Advice to users.*

MPI\_COMM\_CREATE provides a means to subset a group of processes for the purpose of separate MIMD computation, with separate communication space. newcomm, which emerges from MPI\_COMM\_CREATE can be used in subsequent calls to MPI\_COMM\_CREATE (or other communicator constructors) further to subdivide a computation into parallel sub-computations. A more general service is provided by MPI\_COMM\_SPLIT.

If comm\_in is an intercommunicator, then the output communicator is also an intercommunicator where the local group consists only of those processes contained in group. The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of comm\_out. If either group does not specify at least one process in the local group of the intercommunicator, or if the calling process is not included in the group, MPI\_COMM\_NULL is returned.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**See Also**

[MPI\_Comm\_free](http://mpi.deino.net/mpi_functions/MPI_Comm_free.html)

**Example Code**

The following sample code illustrates [MPI\_Comm\_create](http://mpi.deino.net/mpi_functions/MPI_Comm_create.html).

#include <mpi.h>  
#include <stdio.h>  
  
int main(int argc, char\* argv[] )  
{  
    MPI\_Comm dup\_comm\_world, world\_comm;  
    MPI\_Group world\_group;  
    int world\_rank, world\_size, rank, size;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc, &argv);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( MPI\_COMM\_WORLD, &world\_rank );  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( MPI\_COMM\_WORLD, &world\_size );  
    [MPI\_Comm\_dup](http://mpi.deino.net/mpi_functions/MPI_Comm_dup.html)( MPI\_COMM\_WORLD, &dup\_comm\_world );  
    /\* Exercise Comm\_create by creating an equivalent to dup\_comm\_world (sans attributes) \*/  
    [MPI\_Comm\_group](http://mpi.deino.net/mpi_functions/MPI_Comm_group.html)( dup\_comm\_world, &world\_group );  
    [MPI\_Comm\_create](http://mpi.deino.net/mpi_functions/MPI_Comm_create.html)( dup\_comm\_world, world\_group, &world\_comm );  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( world\_comm, &rank );  
    if (rank != world\_rank) {  
        printf( "incorrect rank in world comm: %d\n", rank );fflush(stdout);  
        [MPI\_Abort](http://mpi.deino.net/mpi_functions/MPI_Abort.html)(MPI\_COMM\_WORLD, 3001 );  
    }  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Comm\_size**

Determines the size of the group associated with a communicator

**int MPI\_Comm\_size(**

**MPI\_Comm** *comm***,**

**int** \**size*

**);**

**Parameters**

*comm*

[in] communicator (handle)

*size*

[out] number of processes in the group of comm (integer)

**Remarks**

*Rationale.*

This function is equivalent to accessing the communicator's group with MPI\_COMM\_GROUP (see above), computing the size using MPI\_GROUP\_SIZE, and then freeing the temporary group via MPI\_GROUP\_FREE. However, this function is so commonly used, that this shortcut was introduced. ( *End of rationale.*)   
  
*Advice to users.*

This function indicates the number of processes involved in a communicator. For MPI\_COMM\_WORLD, it indicates the total number of processes available (for this version of MPI, there is no standard way to change the number of processes once initialization has taken place).

**Thread and Interrupt Safety**

This routine is both thread- and interrupt-safe. This means that this routine may safely be used by multiple threads and from within a signal handler.

**Example Code**

The following sample code illustrates [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html).

#include "mpi.h"   
#include <stdio.h>   
int main(int argc, char \*argv[])   
{   
    int rank, nprocs;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);   
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD,&nprocs);   
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD,&rank);   
    printf("Hello, world.  I am %d of %d\n", rank, nprocs);fflush(stdout);   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();   
    return 0;   
}

**MPI\_Comm\_spawn**

Spawn up to maxprocs instances of a single MPI application

**int MPI\_Comm\_spawn(**

**char** \**command***,**

**char** \**argv*[]**,**

**int** *maxprocs***,**

**MPI\_Info** *info***,**

**int** *root***,**

**MPI\_Comm** *comm***,**

**MPI\_Comm** \**intercomm***,**

**int** *array\_of\_errcodes*[]

**);**

**int MPI\_Comm\_spawn(**

**wchar\_t** \**command***,**

**wchar\_t** \**argv*[]**,**

**int** *maxprocs***,**

**MPI\_Info** *info***,**

**int** *root***,**

**MPI\_Comm** *comm***,**

**MPI\_Comm** \**intercomm***,**

**int** *array\_of\_errcodes*[]

**);**

**Parameters**

*command*

[in] name of program to be spawned (string, significant only at root)

*argv*

[in] arguments to command (array of strings, significant only at root)

*maxprocs*

[in] maximum number of processes to start (integer, significant only at root)

*info*

[in] a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root)

*root*

[in] rank of process in which previous arguments are examined (integer)

*comm*

[in] intracommunicator containing group of spawning processes (handle)

*intercomm*

[out] intercommunicator between original group and the newly spawned group (handle)

*array\_of\_errcodes*

[out] one code per process (array of integer)

**Info argument**

DeinoMPI uses the info argument to specify additional control parameters to the spawn command.  Here are the supported keys:

* path - search path to locate the executable. Use semicolons (;) to separate paths.  Example: c:\temp;c:\bin
* host - host name to launch the processes on.  All processes will be launched on this host.
* wdir - working directory
* env - list of environment variables in the form: env=var env2=var2 ...  Variables with spaces or = characters in them should be quoted.  The quote and escape characters need to be escaped within quoted strings.  For example: name="John Doe" random="He said, \"put it in c:\\temp\""
* hosts - list of hosts where to deposit processes.  The form is: hostA[:n] hostB[:m] where :n is an optional way to deposit more than one process per host.  Example: hostA hostB:2 hostC.  If count = 5 the processes will be deposited as follows: hostA hostB hostB hostC hostA
* machinefile - file to be read to create a host list.  The format of the file is one host per line with blank lines and lines beginning with # ignored.  Multiple processes per host can be specified by specifying the host name as follows: hostA:n.  Hosts are selected round robin from the list until count number of processes are launched.
* map - list of network drives to map before launching the processes.  The format is: drive:\\share;drive2:\\share2.  Example: z:\\myserver\home\userA;y:\\myserver\data
* localonly - launch the processes only on the host that performs the spawn.  This is the host that executes mpiexec, not the host of the root process that called spawn.
* priority - set the priority of the launched processes.  Format: priority class[:thread priority].  You can specify the process priority and optionally the thread priority.  The classes are: 0,1,2,3,4 (idle, below, normal, above, high).  The thread priorities are: 0,1,2,3,4,5 (idle, lowest, below, normal, above, highest).  The default is 2:3
* exitcodes - print the exit codes of the spawned processes when the group exits.
* log - shortcut to enable MPE logging of the spawned group.

**Remarks**

MPI\_COMM\_SPAWN tries to start maxprocs identical copies of the MPI program specified by command, establishing communication with them and returning an intercommunicator. The spawned processes are referred to as children. The children have their own MPI\_COMM\_WORLD, which is separate from that of the parents. MPI\_COMM\_SPAWN is collective over comm, and also may not return until MPI\_INIT has been called in the children. Similarly, MPI\_INIT in the children may not return until all parents have called MPI\_COMM\_SPAWN. In this sense, MPI\_COMM\_SPAWN in the parents and MPI\_INIT in the children form a collective operation over the union of parent and child processes. The intercommunicator returned by MPI\_COMM\_SPAWN contains the parent processes in the local group and the child processes in the remote group. The ordering of processes in the local and remote groups is the same as the as the ordering of the group of the comm in the parents and of MPI\_COMM\_WORLD of the children, respectively. This intercommunicator can be obtained in the children through the function MPI\_COMM\_GET\_PARENT.

*Advice to users.*

An implementation may automatically establish communication before MPI\_INIT is called by the children. Thus, completion of MPI\_COMM\_SPAWN in the parent does not necessarily mean that MPI\_INIT has been called in the children (although the returned intercommunicator can be used immediately). ( *End of advice to users.*)

The command argument The command argument is a string containing the name of a program to be spawned. The string is null-terminated in C. In Fortran, leading and trailing spaces are stripped. MPI does not specify how to find the executable or how the working directory is determined. These rules are implementation-dependent and should be appropriate for the runtime environment.

*Advice to users.*

MPI does not say what happens if the program you start is a shell script and that shell script starts a program that calls MPI\_INIT. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. ( *End of advice to users.*)

The argv argument argv is an array of strings containing arguments that are passed to the program. The first element of argv is the first argument passed to command, not, as is conventional in some contexts, the command itself. The argument list is terminated by NULL in C and C++ and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI\_ARGV\_NULL may be used in C, C++ and Fortran to indicate an empty argument list. In C and C++, this constant is the same as NULL.

Examples of argv in C and Fortran

To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:

char command[] = "ocean";

char \*argv[] = {"-gridfile", "ocean1.grd", NULL};

MPI\_Comm\_spawn(command, argv, ...);

or, if not everything is known at compile time:

char \*command;

char \*\*argv;

command = "ocean";

argv=(char \*\*)malloc(3 \* sizeof(char \*));

argv[0] = "-gridfile";

argv[1] = "ocean1.grd";

argv[2] = NULL;

MPI\_Comm\_spawn(command, argv, ...);

In Fortran:

CHARACTER\*25 command, argv(3)

command = ' ocean '

argv(1) = ' -gridfile '

argv(2) = ' ocean1.grd'

argv(3) = ' '

call MPI\_COMM\_SPAWN(command, argv, ...)

Arguments are supplied to the program if this is allowed by the operating system. In C, the MPI\_COMM\_SPAWN argument argv differs from the argv argument of main in two respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the implementation and conventionally contains the name of the program (given by command). argv[1] of main corresponds to argv[0] in MPI\_COMM\_SPAWN, argv[2] of main to argv[1] of MPI\_COMM\_SPAWN, etc. Second, argv of MPI\_COMM\_SPAWN must be null-terminated, so that its length can be determined. Passing an argv of MPI\_ARGV\_NULL to MPI\_COMM\_SPAWN results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the name of the program.

If a Fortran implementation supplies routines that allow a program to obtain its arguments, the arguments may be available through that mechanism. In C, if the operating system does not support arguments appearing in argv of main(), the MPI implementation may add the arguments to the argv that is passed to MPI\_INIT.

The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn maxprocs processes, it raises an error of class MPI\_ERR\_SPAWN.

An implementation may allow the info argument to change the default behavior, such that if the implementation is unable to spawn all maxprocs processes, it may spawn a smaller number of processes instead of raising an error. In principle, the info argument may specify an arbitrary set http://mpi.deino.net/mpi_functions/img7.gifof allowed values for the number of processes spawned. The set *{mi}* does not necessarily include the value maxprocs. If an implementation is able to spawn one of these allowed numbers of processes, MPI\_COMM\_SPAWN returns successfully and the number of spawned processes, *m*, is given by the size of the remote group of intercomm. If *m* is less than maxproc, reasons why the other processes were not spawned are given in array\_of\_errcodes as described below. If it is not possible to spawn one of the allowed numbers of processes, MPI\_COMM\_SPAWN raises an error of class MPI\_ERR\_SPAWN.

A spawn call with the default behavior is called *hard*. A spawn call for which fewer than maxprocs processes may be returned is called soft.

*Advice to users.*

By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior "spawn as many processes as possible, up to *N*," you should do a soft spawn, where the set of allowed values *{mi}* is *{0 ... N}*. However, this is not completely portable, as implementations are not required to support soft spawning. ( *End of advice to users.*)

The info argument The info argument to all of the routines in this chapter is an opaque handle of type MPI\_Info in C, MPI::Info in C++ and INTEGER in Fortran. It is a container for a number of user-specified ( key, value) pairs. key and value are strings (null-terminated char\* in C, character\*(\*) in Fortran).

For the SPAWN calls, info provides additional (and possibly implementation-dependent) instructions to MPI and the runtime system on how to start processes. An application may pass MPI\_INFO\_NULL in C or Fortran. Portable programs not requiring detailed control over process locations should use MPI\_INFO\_NULL.

MPI does not specify the content of the info argument, except to reserve a number of special key values. The info argument is quite flexible and could even be used, for example, to specify the executable and its command-line arguments. In this case the command argument to MPI\_COMM\_SPAWN could be empty. The ability to do this follows from the fact that MPI does not specify how an executable is found, and the info argument can tell the runtime system where to "find" the executable "" (empty string). Of course a program that does this will not be portable across MPI implementations.

The root argument All arguments before the root argument are examined only on the process whose rank in comm is equal to root. The value of these arguments on other processes is ignored.

The array\_of\_errcodes argument The array\_of\_errcodes is an array of length maxprocs in which MPI reports the status of each process that MPI was requested to start. If all maxprocs processes were spawned, array\_of\_errcodes is filled in with the value MPI\_SUCCESS. If only *m* (http://mpi.deino.net/mpi_functions/img8.gif ) processes are spawned, *m* of the entries will contain MPI\_SUCCESS and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class MPI\_ERR\_SPAWN if there was no error in the argument list. In C or Fortran, an application may pass MPI\_ERRCODES\_IGNORE if it is not interested in the error codes. In C++ this constant does not exist, and the array\_of\_errcodes argument may be omitted from the argument list.

*Advice to users.*

It is possible in MPI to start a static SPMD or MPMD application by starting first one process and having that process start its siblings with MPI\_COMM\_SPAWN. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI-1 application.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Comm\_spawn](http://mpi.deino.net/mpi_functions/MPI_Comm_spawn.html).

#include "mpi.h"  
#include <stdio.h>  
#include <stdlib.h>  
  
#define NUM\_SPAWNS 2  
  
int main( int argc, char \*argv[] )  
{  
    int np = NUM\_SPAWNS;  
    int errcodes[NUM\_SPAWNS];  
    MPI\_Comm parentcomm, intercomm;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
    [MPI\_Comm\_get\_parent](http://mpi.deino.net/mpi_functions/MPI_Comm_get_parent.html)( &parentcomm );  
    if (parentcomm == MPI\_COMM\_NULL)  
    {  
        /\* Create 2 more processes - this example must be called spawn\_example.exe for this to work. \*/  
        [MPI\_Comm\_spawn](http://mpi.deino.net/mpi_functions/MPI_Comm_spawn.html)( "spawn\_example", MPI\_ARGV\_NULL, np, MPI\_INFO\_NULL, 0, MPI\_COMM\_WORLD, &intercomm, errcodes );  
        printf("I'm the parent.\n");  
    }  
    else  
    {  
        printf("I'm the spawned.\n");  
    }  
    fflush(stdout);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Finalize**

Terminates MPI execution environment

int MPI\_Finalize( void );

**Remarks**

This routines cleans up all MPI state. Once this routine is called, no MPI routine (even MPI\_INIT) may be called. The user must ensure that all pending communications involving a process completes before the process calls MPI\_FINALIZE.

All processes must call this routine before exiting. The number of processes running *after* this routine is called is undefined; it is best not to perform much more than a return rc after calling [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html).

**Thread and Signal Safety**

The MPI standard requires that [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html) be called *only* by the same thread that initialized MPI with either [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html) or [MPI\_Init\_thread](http://mpi.deino.net/mpi_functions/MPI_Init_thread.html).

**Example Code**

The following sample code illustrates [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html).

#include "mpi.h"   
#include <stdio.h>   
int main(int argc, char \*argv[])   
{   
    int rank, nprocs;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);   
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD,&nprocs);   
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD,&rank);   
    printf("Hello, world.  I am %d of %d\n", rank, nprocs);fflush(stdout);   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();   
    return 0;   
}

# MPI\_Gather

Gathers together values from a group of processes

**int MPI\_Gather(**

**void** \**sendbuf***,**

**int** *sendcnt***,**

**MPI\_Datatype** *sendtype***,**

**void** \**recvbuf***,**

**int** *recvcnt***,**

**MPI\_Datatype** *recvtype***,**

**int** *root***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*sendbuf*

[in] starting address of send buffer (choice)

*sendcount*

[in] number of elements in send buffer (integer)

*sendtype*

[in] data type of send buffer elements (handle)

*recvbuf*

[out] address of receive buffer (choice, significant only at root)

*recvcount*

[in] number of elements for any single receive (integer, significant only at root)

*recvtype*

[in] data type of recv buffer elements (significant only at root) (handle)

*root*

[in] rank of receiving process (integer)

*comm*

[in] communicator (handle)

**Remarks**

Each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is *as if* each of the n processes in the group (including the root process) had executed a call to

http://mpi.deino.net/mpi_functions/img103.gif

and the root had executed n calls to

http://mpi.deino.net/mpi_functions/img104.gif

where extent(recvtype) is the type extent obtained from a call to MPI\_Type\_extent().

An alternative description is that the n messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to MPI\_RECV(recvbuf, recvcounthttp://mpi.deino.net/mpi_functions/img105.gif n, recvtype, ...).

The receive buffer is ignored for all non-root processes.

General, derived datatypes are allowed for both sendtype and recvtype. The type signature of sendcount, sendtype on process i must be equal to the type signature of recvcount, recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the recvcount argument at the root indicates the number of items it receives from *each* process, not the total number of items it receives.

The "in place" option for intracommunicators is specified by passing MPI\_IN\_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI\_ROOT in root. All other processes in group A pass the value MPI\_PROC\_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Gather](http://mpi.deino.net/mpi_functions/MPI_Gather.html).

#include "mpi.h"  
#include <stdlib.h>  
#include <stdio.h>  
  
/\* Gather data from a vector to contiguous \*/  
  
int main( int argc, char \*\*argv )  
{  
    MPI\_Datatype vec;  
    MPI\_Comm comm;  
    double \*vecin, \*vecout;  
    int minsize = 2, count;  
    int root, i, n, stride, errs = 0;  
    int rank, size;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
    comm = MPI\_COMM\_WORLD;  
    /\* Determine the sender and receiver \*/  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( comm, &rank );  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( comm, &size );  
   
    for (root=0; root<size; root++) {  
        for (count = 1; count < 65000; count = count \* 2) {  
            n = 12;  
            stride = 10;  
            vecin = (double \*)malloc( n \* stride \* size \* sizeof(double) );  
            vecout = (double \*)malloc( size \* n \* sizeof(double) );  
   
            [MPI\_Type\_vector](http://mpi.deino.net/mpi_functions/MPI_Type_vector.html)( n, 1, stride, MPI\_DOUBLE, &vec );  
            [MPI\_Type\_commit](http://mpi.deino.net/mpi_functions/MPI_Type_commit.html)( &vec );  
  
            for (i=0; i<n\*stride; i++) vecin[i] =-2;  
            for (i=0; i<n; i++) vecin[i\*stride] = rank \* n + i;  
  
            [MPI\_Gather](http://mpi.deino.net/mpi_functions/MPI_Gather.html)( vecin, 1, vec, vecout, n, MPI\_DOUBLE, root, comm );  
  
            if (rank == root) {  
                for (i=0; i<n\*size; i++) {  
                    if (vecout[i] != i) {  
                        errs++;  
                        if (errs < 10) {  
                            fprintf( stderr, "vecout[%d]=%d\n", i, (int)vecout[i] );fflush(stderr);  
                        }  
                    }  
                }  
            }  
            [MPI\_Type\_free](http://mpi.deino.net/mpi_functions/MPI_Type_free.html)( &vec );  
            free( vecin );  
            free( vecout );  
        }  
    }  
   
    /\* do a zero length gather \*/  
    [MPI\_Gather](http://mpi.deino.net/mpi_functions/MPI_Gather.html)( NULL, 0, MPI\_BYTE, NULL, 0, MPI\_BYTE, 0, MPI\_COMM\_WORLD );  
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Init**

Initialize the MPI execution environment

**int MPI\_Init(**

**int** \**argc***,**

**char** \*\*\**argv*

**);**

**int MPI\_Init(**

**int** \**argc***,**

**wchar\_t** \*\*\**argv*

**);**

**Parameters**

*argc*

[in] Pointer to the number of arguments

*argv*

[in] Pointer to the argument vector

**Remarks**

This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous (see MPI\_INITIALIZED).

The MPI standard does not say what a program can do before an MPI\_INIT or after an MPI\_FINALIZE. In the MPICH implementation, you should do as little as possible. In particular, avoid anything that changes the external state of the program, such as opening files, reading standard input or writing to standard output.

**Thread and Signal Safety**

This routine must be called by one thread only. That thread is called the *main thread* and must be the thread that calls [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html).

**Example Code**

The following sample code illustrates [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html).

#include "mpi.h"   
#include <stdio.h>   
int main(int argc, char \*argv[])   
{   
    int rank, nprocs;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);   
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD,&nprocs);   
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD,&rank);   
    printf("Hello, world.  I am %d of %d\n", rank, nprocs);fflush(stdout);   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();   
    return 0;   
}

# MPI\_Irecv

Begins a nonblocking receive

**int MPI\_Irecv(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *source***,**

**int** *tag***,**

**MPI\_Comm** *comm***,**

**MPI\_Request** \**request*

**);**

**Parameters**

*buf*

[in] initial address of receive buffer (choice)

*count*

[in] number of elements in receive buffer (integer)

*datatype*

[in] datatype of each receive buffer element (handle)

*source*

[in] rank of source (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

*request*

[out] communication request (handle)

**Remarks**

Start a nonblocking receive.

These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Irecv](http://mpi.deino.net/mpi_functions/MPI_Irecv.html).

#include "mpi.h"  
#include <stdio.h>  
   
int main(int argc, char \*argv[])  
{  
    int myid, numprocs, left, right;  
    int buffer[10], buffer2[10];  
    MPI\_Request request;  
    MPI\_Status status;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &numprocs);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &myid);  
   
    right = (myid + 1) % numprocs;  
    left = myid - 1;  
    if (left < 0)  
        left = numprocs - 1;  
   
    [MPI\_Irecv](http://mpi.deino.net/mpi_functions/MPI_Irecv.html)(buffer, 10, MPI\_INT, left, 123, MPI\_COMM\_WORLD, &request);  
    [MPI\_Send](http://mpi.deino.net/mpi_functions/MPI_Send.html)(buffer2, 10, MPI\_INT, right, 123, MPI\_COMM\_WORLD);  
    [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)(&request, &status);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Irsend**

Starts a nonblocking ready send

**int MPI\_Irsend(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm***,**

**MPI\_Request** \**request*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements in send buffer (integer)

*datatype*

[in] datatype of each send buffer element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

*request*

[out] communication request (handle)

**Remarks**

Start a ready mode nonblocking send.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Irsend](http://mpi.deino.net/mpi_functions/MPI_Irsend.html).

#include "mpi.h"  
#include <stdio.h>  
int main(int argc, char \*argv[])  
{  
    int myid, numprocs, left, right;  
    int buffer[10], buffer2[10];  
    MPI\_Request request, request2;  
    MPI\_Status status;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &numprocs);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &myid);  
    right = (myid + 1) % numprocs;  
    left = myid - 1;  
    if (left < 0)  
        left = numprocs - 1;  
    [MPI\_Irecv](http://mpi.deino.net/mpi_functions/MPI_Irecv.html)(buffer, 10, MPI\_INT, left, 123, MPI\_COMM\_WORLD, &request);  
    [MPI\_Barrier](http://mpi.deino.net/mpi_functions/MPI_Barrier.html)(MPI\_COMM\_WORLD); /\* Ready sends require that the receive buffer be ready before the send call initiates so use a barrier to ensure this is true \*/  
    [MPI\_Irsend](http://mpi.deino.net/mpi_functions/MPI_Irsend.html)(buffer2, 10, MPI\_INT, right, 123, MPI\_COMM\_WORLD, &request2);  
    [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)(&request, &status);  
    [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)(&request2, &status);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Isend**

Begins a nonblocking send

**int MPI\_Isend(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm***,**

**MPI\_Request** \**request*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements in send buffer (integer)

*datatype*

[in] datatype of each send buffer element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

*request*

[out] communication request (handle)

**Remarks**

Start a standard mode, nonblocking send.

**Example Code**

The following sample code illustrates [MPI\_Isend](http://mpi.deino.net/mpi_functions/MPI_Isend.html).

#include "mpi.h"  
#include <stdio.h>  
int main(int argc, char \*argv[])  
{  
    int myid, numprocs, left, right;  
    int buffer[10], buffer2[10];  
    MPI\_Request request, request2;  
    MPI\_Status status;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &numprocs);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &myid);  
    right = (myid + 1) % numprocs;  
    left = myid - 1;  
    if (left < 0)  
        left = numprocs - 1;  
    [MPI\_Irecv](http://mpi.deino.net/mpi_functions/MPI_Irecv.html)(buffer, 10, MPI\_INT, left, 123, MPI\_COMM\_WORLD, &request);  
    [MPI\_Isend](http://mpi.deino.net/mpi_functions/MPI_Isend.html)(buffer2, 10, MPI\_INT, right, 123, MPI\_COMM\_WORLD, &request2);  
    [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)(&request, &status);  
    [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)(&request2, &status);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Issend**

Starts a nonblocking synchronous send

**int MPI\_Issend(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm***,**

**MPI\_Request** \**request*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements in send buffer (integer)

*datatype*

[in] datatype of each send buffer element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

*request*

[out] communication request (handle)

**Remarks**

Start a synchronous mode, nonblocking send.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Issend](http://mpi.deino.net/mpi_functions/MPI_Issend.html).

#include "mpi.h"  
#include <stdio.h>  
int main(int argc, char \*argv[])  
{  
    int myid, numprocs, left, right;  
    int buffer[10], buffer2[10];  
    MPI\_Request request, request2;  
    MPI\_Status status;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &numprocs);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &myid);  
    right = (myid + 1) % numprocs;  
    left = myid - 1;  
    if (left < 0)  
        left = numprocs - 1;  
    [MPI\_Irecv](http://mpi.deino.net/mpi_functions/MPI_Irecv.html)(buffer, 10, MPI\_INT, left, 123, MPI\_COMM\_WORLD, &request);  
    [MPI\_Issend](http://mpi.deino.net/mpi_functions/MPI_Issend.html)(buffer2, 10, MPI\_INT, right, 123, MPI\_COMM\_WORLD, &request2);  
    [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)(&request, &status);  
    [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)(&request2, &status);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Ibsend**

Starts a nonblocking buffered send

**int MPI\_Ibsend(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm***,**

**MPI\_Request** \**request*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements in send buffer (integer)

*datatype*

[in] datatype of each send buffer element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

*request*

[out] communication request (handle)

**Remarks**

Start a buffered mode, nonblocking send.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Ibsend](http://mpi.deino.net/mpi_functions/MPI_Ibsend.html).

#include <stdio.h>  
#include "mpi.h"  
   
#define BUFSIZE 2000  
int main( int argc, char \*argv[] )  
{  
    MPI\_Status status;  
    MPI\_Request request;  
    int a[10], b[10];  
    int buf[BUFSIZE], \*bptr, bl, i, j, rank, size, errs=0;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( 0, 0 );  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( MPI\_COMM\_WORLD, &rank );  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( MPI\_COMM\_WORLD, &size );  
    [MPI\_Buffer\_attach](http://mpi.deino.net/mpi_functions/MPI_Buffer_attach.html)( buf, BUFSIZE );  
   
    for (j=0; j<10; j++) {  
        for (i=0; i<10; i++) {  
            a[i] = (rank + 10 \* j) \* size + i;  
        }  
        [MPI\_Ibsend](http://mpi.deino.net/mpi_functions/MPI_Ibsend.html)( a, 10, MPI\_INT, 0, 27+j, MPI\_COMM\_WORLD, &request );  
        [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)( &request, &status );  
    }  
    if (rank == 0) {  
        for (i=0; i<size; i++) {  
            for (j=0; j<10; j++) {  
                int k;  
                status.MPI\_TAG = -10;  
                status.MPI\_SOURCE = -20;  
                [MPI\_Recv](http://mpi.deino.net/mpi_functions/MPI_Recv.html)( b, 10, MPI\_INT, i, 27+j, MPI\_COMM\_WORLD, &status );  
  
                if (status.MPI\_TAG != 27+j) {  
                    errs++;  
                    printf( "Wrong tag = %d\n", status.MPI\_TAG );fflush(stdout);  
                }  
                if (status.MPI\_SOURCE != i) {  
                    errs++;  
                    printf( "Wrong source = %d\n", status.MPI\_SOURCE );fflush(stdout);  
                }  
                for (k=0; k<10; k++) {  
                    if (b[k] != (i + 10 \* j) \* size + k) {  
                        errs ++;  
                        printf( "received b[%d] = %d from %d tag %d\n", k, b[k], i, 27+j );fflush(stdout);  
                    }  
                }  
            }  
        }  
    }  
    [MPI\_Buffer\_detach](http://mpi.deino.net/mpi_functions/MPI_Buffer_detach.html)( &bptr, &bl );  
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return errs;  
}

**MPI\_Reduce**

Reduces values on all processes to a single value

**int MPI\_Reduce(**

**void** \**sendbuf***,**

**void** \**recvbuf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**MPI\_Op** *op***,**

**int** *root***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*sendbuf*

[in] address of send buffer (choice)

*recvbuf*

[out] address of receive buffer (choice, significant only at root)

*count*

[in] number of elements in send buffer (integer)

*datatype*

[in] data type of elements of send buffer (handle)

*op*

[in] reduce operation (handle)

*root*

[in] rank of root process (integer)

*comm*

[in] communicator (handle)

**Remarks**

MPI\_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. 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 MPI\_MAX and the send buffer contains two elements that are floating point numbers ( count = 2 and datatype = MPI\_FLOAT), then http://mpi.deino.net/mpi_functions/img123.gifand http://mpi.deino.net/mpi_functions/img124.gif.

The operation op 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.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Notes on collective operations**

The reduction functions (MPI\_Op) do not return an error value. As a result, if the functions detect an error, all they can do is either call MPI\_Abort or silently skip the problem. Thus, if you change the error handler from MPI\_ERRORS\_ARE\_FATAL to something else, for example, MPI\_ERRORS\_RETURN, then no error may be indicated.

The reason for this is the performance problems in ensuring that all collective routines return the same error value.

**Example Code**

The following sample code illustrates [MPI\_Reduce](http://mpi.deino.net/mpi_functions/MPI_Reduce.html).

#include "mpi.h"  
#include <stdio.h>  
#include <stdlib.h>  
   
/\* A simple test of Reduce with all choices of root process \*/  
int main( int argc, char \*argv[] )  
{  
    int errs = 0;  
    int rank, size, root;  
    int \*sendbuf, \*recvbuf, i;  
    int minsize = 2, count;   
    MPI\_Comm comm;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
   
    comm = MPI\_COMM\_WORLD;  
    /\* Determine the sender and receiver \*/  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( comm, &rank );  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( comm, &size );  
   
    for (count = 1; count < 130000; count = count \* 2) {  
        sendbuf = (int \*)malloc( count \* sizeof(int) );  
        recvbuf = (int \*)malloc( count \* sizeof(int) );  
        for (root = 0; root < size; root ++) {  
            for (i=0; i<count; i++) sendbuf[i] = i;  
            for (i=0; i<count; i++) recvbuf[i] = -1;  
            [MPI\_Reduce](http://mpi.deino.net/mpi_functions/MPI_Reduce.html)( sendbuf, recvbuf, count, MPI\_INT, MPI\_SUM, root, comm );  
            if (rank == root) {  
                for (i=0; i<count; i++) {  
                    if (recvbuf[i] != i \* size) {  
                        errs++;  
                    }  
                }  
            }  
        }  
        free( sendbuf );  
        free( recvbuf );  
    }  
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return errs;  
}

**MPI\_Reduce\_scatter**

Combines values and scatters the results

**int MPI\_Reduce\_scatter(**

**void** \**sendbuf***,**

**void** \**recvbuf***,**

**int** \**recvcnts***,**

**MPI\_Datatype** *datatype***,**

**MPI\_Op** *op***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*sendbuf*

[in] starting address of send buffer (choice)

*recvbuf*

[out] starting address of receive buffer (choice)

*recvcounts*

[in] integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.

*datatype*

[in] data type of elements of input buffer (handle)

*op*

[in] operation (handle)

*comm*

[in] communicator (handle)

**Remarks**

MPI\_REDUCE\_SCATTER first does an element-wise reduction on vector of http://mpi.deino.net/mpi_functions/img138.gifelements in the send buffer defined by sendbuf, count and datatype. Next, the resulting vector of results is split into n disjoint segments, where n is the number of members in the group. Segment i contains recvcounts[i] elements. The ith segment is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Notes on collective operations**

The reduction functions (MPI\_Op) do not return an error value. As a result, if the functions detect an error, all they can do is either call MPI\_Abort or silently skip the problem. Thus, if you change the error handler from MPI\_ERRORS\_ARE\_FATAL to something else, for example, MPI\_ERRORS\_RETURN, then no error may be indicated.

The reason for this is the performance problems in ensuring that all collective routines return the same error value.

**Example Code**

The following sample code illustrates [MPI\_Reduce\_scatter](http://mpi.deino.net/mpi_functions/MPI_Reduce_scatter.html).

#include "mpi.h"  
#include <stdio.h>  
#include <stdlib.h>  
   
int main( int argc, char \*\*argv )  
{  
    int err = 0;  
    int \*sendbuf, recvbuf, \*recvcounts;  
    int size, rank, i, sumval;  
    MPI\_Comm comm;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
    comm = MPI\_COMM\_WORLD;  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( comm, &size );  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( comm, &rank );  
    sendbuf = (int \*) malloc( size \* sizeof(int) );  
    for (i=0; i<size; i++)   
        sendbuf[i] = rank + i;  
    recvcounts = (int \*)malloc( size \* sizeof(int) );  
    for (i=0; i<size; i++)   
        recvcounts[i] = 1;  
   
    [MPI\_Reduce\_scatter](http://mpi.deino.net/mpi_functions/MPI_Reduce_scatter.html)( sendbuf, &recvbuf, recvcounts, MPI\_INT, MPI\_SUM, comm );  
   
    sumval = size \* rank + ((size - 1) \* size)/2;  
    /\* recvbuf should be size \* (rank + i) \*/  
    if (recvbuf != sumval) {  
        err++;  
        fprintf( stdout, "Did not get expected value for reduce scatter\n" );  
        fprintf( stdout, "[%d] Got %d expected %d\n", rank, recvbuf, sumval );fflush(stdout);  
    }  
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)( );  
    return err;  
}

**MPI\_Rsend**

Blocking ready send

**int MPI\_Rsend(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements in send buffer (nonnegative integer)

*datatype*

[in] datatype of each send buffer element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

**Remarks**

Send in ready mode.

A send that uses the **ready** communication mode may be started *only* if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**MPI\_Rsend\_init**

Creates a persistent request for a ready send

**int MPI\_Rsend\_init(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm***,**

**MPI\_Request** \**request*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements sent (integer)

*datatype*

[in] type of each element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

*request*

[out] communication request (handle)

**Remarks**

Creates a persistent communication object for a ready mode send operation.

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a **persistent** communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a "half-channel." It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another. It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**MPI\_Scatter**

Sends data from one process to all other processes in a communicator

**int MPI\_Scatter(**

**void** \**sendbuf***,**

**int** *sendcnt***,**

**MPI\_Datatype** *sendtype***,**

**void** \**recvbuf***,**

**int** *recvcnt***,**

**MPI\_Datatype** *recvtype***,**

**int** *root***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*sendbuf*

[in] address of send buffer (choice, significant only at root)

*sendcount*

[in] number of elements sent to each process (integer, significant only at root)

*sendtype*

[in] data type of send buffer elements (significant only at root) (handle)

*recvbuf*

[out] address of receive buffer (choice)

*recvcount*

[in] number of elements in receive buffer (integer)

*recvtype*

[in] data type of receive buffer elements (handle)

*root*

[in] rank of sending process (integer)

*comm*

[in] communicator (handle)

**Remarks**

MPI\_SCATTER is the inverse operation to MPI\_GATHER.

The outcome is *as if* the root executed n send operations,

http://mpi.deino.net/mpi_functions/img112.gif

and each process executed a receive,

http://mpi.deino.net/mpi_functions/img113.gif

An alternative description is that the root sends a message with MPI\_Send(sendbuf, sendcounthttp://mpi.deino.net/mpi_functions/img114.gif n, sendtype, ...). This message is split into n equal segments, the *i*th segment is sent to the *i*th process in the group, and each process receives this message as above.

The send buffer is ignored for all non-root processes.

The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

The "in place" option for intracommunicators is specified by passing MPI\_IN\_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain *n* segments, where *n* is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI\_ROOT in root. All other processes in group A pass the value MPI\_PROC\_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Scatter](http://mpi.deino.net/mpi_functions/MPI_Scatter.html).

#include "mpi.h"  
#include <stdio.h>  
   
#define MAX\_PROCESSES 10  
   
int main( int argc, char \*\*argv )  
{  
    int rank, size, i,j;  
    int table[MAX\_PROCESSES][MAX\_PROCESSES];  
    int row[MAX\_PROCESSES];  
    int errors=0;  
    int participants;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( MPI\_COMM\_WORLD, &rank );  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( MPI\_COMM\_WORLD, &size );  
   
    /\* A maximum of MAX\_PROCESSES processes can participate \*/  
    if ( size > MAX\_PROCESSES ) participants = MAX\_PROCESSES;  
    else participants = size;  
    if ( (rank < participants) ) {  
        int send\_count = MAX\_PROCESSES;  
        int recv\_count = MAX\_PROCESSES;  
   
        /\* If I'm the root (process 0), then fill out the big table \*/  
        if (rank == 0)   
            for ( i=0; i<participants; i++)   
                for ( j=0; j<MAX\_PROCESSES; j++ )   
                    table[i][j] = i+j;  
  
        /\* Scatter the big table to everybody's little table \*/  
        [MPI\_Scatter](http://mpi.deino.net/mpi_functions/MPI_Scatter.html)(&table[0][0], send\_count, MPI\_INT,   
                         &row[0] , recv\_count, MPI\_INT, 0, MPI\_COMM\_WORLD);  
   
        /\* Now see if our row looks right \*/  
        for (i=0; i<MAX\_PROCESSES; i++)   
            if ( row[i] != i+rank ) errors++;  
    }   
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return errors;  
}

**MPI\_Scatterv**

Scatters a buffer in parts to all processes in a communicator

**int MPI\_Scatterv(**

**void** \**sendbuf***,**

**int** \**sendcnts***,**

**int** \**displs***,**

**MPI\_Datatype** *sendtype***,**

**void** \**recvbuf***,**

**int** *recvcnt***,**

**MPI\_Datatype** *recvtype***,**

**int** *root***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*sendbuf*

[in] address of send buffer (choice, significant only at root)

*sendcounts*

[in] integer array (of length group size) specifying the number of elements to send to each processor

*displs*

[in] integer array (of length group size). Entry i specifies the displacement (relative to sendbuf from which to take the outgoing data to process i

*sendtype*

[in] data type of send buffer elements (handle)

*recvbuf*

[out] address of receive buffer (choice)

*recvcount*

[in] number of elements in receive buffer (integer)

*recvtype*

[in] data type of receive buffer elements (handle)

*root*

[in] rank of sending process (integer)

*comm*

[in] communicator (handle)

**Remarks**

MPI\_SCATTERV is the inverse operation to MPI\_GATHERV.

MPI\_SCATTERV extends the functionality of MPI\_SCATTER by allowing a varying count of data to be sent to each process, since sendcounts is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing the new argument, displs.

The outcome is as if the root executed n send operations,

http://mpi.deino.net/mpi_functions/img115.gif

and each process executed a receive,

http://mpi.deino.net/mpi_functions/img116.gif

The send buffer is ignored for all non-root processes.

The type signature implied by sendcount[i], sendtype at the root must be equal to the type signature implied by recvcount, recvtype at process i (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts, types, and displacements should not cause any location on the root to be read more than once.

The "in place" option for intracommunicators is specified by passing MPI\_IN\_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain *n* segments, where *n* is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI\_ROOT in root. All other processes in group A pass the value MPI\_PROC\_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Scatterv](http://mpi.deino.net/mpi_functions/MPI_Scatterv.html).

#include "mpi.h"  
#include <stdio.h>  
   
#define MAX\_PROCESSES 10  
   
int main( int argc, char \*\*argv )  
{  
    int rank, size, i,j;  
    int table[MAX\_PROCESSES][MAX\_PROCESSES];  
    int row[MAX\_PROCESSES];  
    int errors=0;  
    int participants;  
    int displs[MAX\_PROCESSES];  
    int send\_counts[MAX\_PROCESSES];  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( MPI\_COMM\_WORLD, &rank );  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( MPI\_COMM\_WORLD, &size );  
   
    /\* A maximum of MAX\_PROCESSES processes can participate \*/  
    if ( size > MAX\_PROCESSES ) participants = MAX\_PROCESSES;  
    else participants = size;  
    if ( (rank < participants) ) {  
        int recv\_count = MAX\_PROCESSES;  
  
        /\* If I'm the root (process 0), then fill out the big table and setup send\_counts and displs arrays \*/  
        if (rank == 0)   
            for ( i=0; i<participants; i++) {  
                send\_counts[i] = recv\_count;  
                displs[i] = i \* MAX\_PROCESSES;  
                for ( j=0; j<MAX\_PROCESSES; j++ )   
                    table[i][j] = i+j;  
            }  
   
        /\* Scatter the big table to everybody's little table \*/  
        [MPI\_Scatterv](http://mpi.deino.net/mpi_functions/MPI_Scatterv.html)(&table[0][0], send\_counts, displs, MPI\_INT,   
                           &row[0] , recv\_count, MPI\_INT, 0, MPI\_COMM\_WORLD);  
   
        /\* Now see if our row looks right \*/  
        for (i=0; i<MAX\_PROCESSES; i++)   
            if ( row[i] != i+rank ) errors++;  
    }   
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return errors;  
}

**MPI\_Send**

Performs a blocking send

**int MPI\_Send(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements in send buffer (nonnegative integer)

*datatype*

[in] datatype of each send buffer element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

**Remarks**

This routine may block until the message is received by the destination process.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Send](http://mpi.deino.net/mpi_functions/MPI_Send.html).

#include "mpi.h"  
#include <stdio.h>  
   
int main(int argc, char \*argv[])  
{  
    int rank, size, i;  
    int buffer[10];  
    MPI\_Status status;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc, &argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &size);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &rank);  
    if (size < 2)  
    {  
        printf("Please run with two processes.\n");fflush(stdout);  
        [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
        return 0;  
    }  
    if (rank == 0)  
    {  
        for (i=0; i<10; i++)  
            buffer[i] = i;  
        [MPI\_Send](http://mpi.deino.net/mpi_functions/MPI_Send.html)(buffer, 10, MPI\_INT, 1, 123, MPI\_COMM\_WORLD);  
    }  
    if (rank == 1)  
    {  
        for (i=0; i<10; i++)  
            buffer[i] = -1;  
        [MPI\_Recv](http://mpi.deino.net/mpi_functions/MPI_Recv.html)(buffer, 10, MPI\_INT, 0, 123, MPI\_COMM\_WORLD, &status);  
        for (i=0; i<10; i++)  
        {  
            if (buffer[i] != i)  
                printf("Error: buffer[%d] = %d but is expected to be %d\n", i, buffer[i], i);  
        }  
        fflush(stdout);  
    }  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Send\_init**

Create a persistent request for a standard send

**int MPI\_Send\_init(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm***,**

**MPI\_Request** \**request*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements sent (integer)

*datatype*

[in] type of each element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

*request*

[out] communication request (handle)

**Remarks**

Creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a **persistent** communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a "half-channel." It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another. It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Send\_init](http://mpi.deino.net/mpi_functions/MPI_Send_init.html).

#include "mpi.h"  
#include <stdlib.h>  
   
int main(int argc, char \*argv[])  
{  
    MPI\_Request r;  
    MPI\_Status s;  
    int flag;  
    int buf[10];  
    int rbuf[10];  
    int tag = 27;  
    int dest = 0;  
    int rank, size, i;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( MPI\_COMM\_WORLD, &size );  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( MPI\_COMM\_WORLD, &rank );  
   
    /\* Create a persistent send request \*/  
    [MPI\_Send\_init](http://mpi.deino.net/mpi_functions/MPI_Send_init.html)( buf, 10, MPI\_INT, dest, tag, MPI\_COMM\_WORLD, &r );  
   
    /\* Use that request \*/  
    if (rank == 0) {  
        int i;  
        MPI\_Request \*rr = (MPI\_Request \*)malloc(size \* sizeof(MPI\_Request));  
        for (i=0; i<size; i++) {  
            [MPI\_Irecv](http://mpi.deino.net/mpi_functions/MPI_Irecv.html)( rbuf, 10, MPI\_INT, i, tag, MPI\_COMM\_WORLD, &rr[i] );  
        }  
        [MPI\_Start](http://mpi.deino.net/mpi_functions/MPI_Start.html)( &r );  
        [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)( &r, &s );  
        [MPI\_Waitall](http://mpi.deino.net/mpi_functions/MPI_Waitall.html)( size, rr, MPI\_STATUSES\_IGNORE );  
        free(rr);  
    }  
    else {  
        [MPI\_Start](http://mpi.deino.net/mpi_functions/MPI_Start.html)( &r );  
        [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)( &r, &s );  
    }  
   
    [MPI\_Request\_free](http://mpi.deino.net/mpi_functions/MPI_Request_free.html)( &r );  
   
   
    if (rank == 0)  
    {  
        MPI\_Request sr;  
        /\* Create a persistent receive request \*/  
        [MPI\_Recv\_init](http://mpi.deino.net/mpi_functions/MPI_Recv_init.html)( rbuf, 10, MPI\_INT, MPI\_ANY\_SOURCE, tag, MPI\_COMM\_WORLD, &r );  
        [MPI\_Isend](http://mpi.deino.net/mpi_functions/MPI_Isend.html)( buf, 10, MPI\_INT, 0, tag, MPI\_COMM\_WORLD, &sr );  
        for (i=0; i<size; i++) {  
            [MPI\_Start](http://mpi.deino.net/mpi_functions/MPI_Start.html)( &r );  
            [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)( &r, &s );  
        }  
        [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)( &sr, &s );  
        [MPI\_Request\_free](http://mpi.deino.net/mpi_functions/MPI_Request_free.html)( &r );  
    }  
    else {  
        [MPI\_Send](http://mpi.deino.net/mpi_functions/MPI_Send.html)( buf, 10, MPI\_INT, 0, tag, MPI\_COMM\_WORLD );  
    }  
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Sendrecv**

Sends and receives a message

**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*

**);**

**Parameters**

*sendbuf*

[in] initial address of send buffer (choice)

*sendcount*

[in] number of elements in send buffer (integer)

*sendtype*

[in] type of elements in send buffer (handle)

*dest*

[in] rank of destination (integer)

*sendtag*

[in] send tag (integer)

*recvbuf*

[out] initial address of receive buffer (choice)

*recvcount*

[in] number of elements in receive buffer (integer)

*recvtype*

[in] type of elements in receive buffer (handle)

*source*

[in] rank of source (integer)

*recvtag*

[in] receive tag (integer)

*comm*

[in] communicator (handle)

*status*

[out] status object (Status). This refers to the receive operation.

**Remarks**

Execute a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Sendrecv](http://mpi.deino.net/mpi_functions/MPI_Sendrecv.html).

#include "mpi.h"  
#include <stdio.h>  
   
int main(int argc, char \*argv[])  
{  
    int myid, numprocs, left, right;  
    int buffer[10], buffer2[10];  
    MPI\_Request request;  
    MPI\_Status status;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &numprocs);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &myid);  
   
    right = (myid + 1) % numprocs;  
    left = myid - 1;  
    if (left < 0)  
        left = numprocs - 1;  
   
    [MPI\_Sendrecv](http://mpi.deino.net/mpi_functions/MPI_Sendrecv.html)(buffer, 10, MPI\_INT, left, 123, buffer2, 10, MPI\_INT, right, 123, MPI\_COMM\_WORLD, &status);  
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Sendrecv\_replace**

Sends and receives using a single buffer

**int MPI\_Sendrecv\_replace(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *sendtag***,**

**int** *source***,**

**int** *recvtag***,**

**MPI\_Comm** *comm***,**

**MPI\_Status** \**status*

**);**

**Parameters**

*buf*

[in/out] initial address of send and receive buffer (choice)

*count*

[in] number of elements in send and receive buffer (integer)

*datatype*

[in] type of elements in send and receive buffer (handle)

*dest*

[in] rank of destination (integer)

*sendtag*

[in] send message tag (integer)

*source*

[in] rank of source (integer)

*recvtag*

[in] receive message tag (integer)

*comm*

[in] communicator (handle)

*status*

[out] status object (Status)

**Remarks**

Execute a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Sendrecv\_replace](http://mpi.deino.net/mpi_functions/MPI_Sendrecv_replace.html).

#include "mpi.h"  
#include <stdio.h>  
   
int main(int argc, char \*argv[])  
{  
    int myid, numprocs, left, right;  
    int buffer[10];  
    MPI\_Request request;  
    MPI\_Status status;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &numprocs);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &myid);  
   
    right = (myid + 1) % numprocs;  
    left = myid - 1;  
    if (left < 0)  
        left = numprocs - 1;  
   
    [MPI\_Sendrecv\_replace](http://mpi.deino.net/mpi_functions/MPI_Sendrecv_replace.html)(buffer, 10, MPI\_INT, left, 123, right, 123, MPI\_COMM\_WORLD, &status);  
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Ssend**

Blocking synchronous send

**int MPI\_Ssend(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements in send buffer (nonnegative integer)

*datatype*

[in] datatype of each send buffer element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

**Remarks**

Send in synchronous mode.

A send that uses the **synchronous** mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is **non-local**.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Ssend](http://mpi.deino.net/mpi_functions/MPI_Ssend.html).

#include "mpi.h"  
#include <stdio.h>  
   
int main(int argc, char \*argv[])  
{  
    int rank, size, i;  
    int buffer[10];  
    MPI\_Status status;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc, &argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &size);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &rank);  
    if (size < 2)  
    {  
        printf("Please run with two processes.\n");fflush(stdout);  
        [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
        return 0;  
    }  
    if (rank == 0)  
    {  
        for (i=0; i<10; i++)  
            buffer[i] = i;  
        [MPI\_Ssend](http://mpi.deino.net/mpi_functions/MPI_Ssend.html)(buffer, 10, MPI\_INT, 1, 123, MPI\_COMM\_WORLD);  
    }  
    if (rank == 1)  
    {  
        for (i=0; i<10; i++)  
            buffer[i] = -1;  
        [MPI\_Recv](http://mpi.deino.net/mpi_functions/MPI_Recv.html)(buffer, 10, MPI\_INT, 0, 123, MPI\_COMM\_WORLD, &status);  
        for (i=0; i<10; i++)  
        {  
            if (buffer[i] != i)  
                printf("Error: buffer[%d] = %d but is expected to be %d\n", i, buffer[i], i);  
        }  
        fflush(stdout);  
    }  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Ssend\_init**

Creates a persistent request for a synchronous send

**int MPI\_Ssend\_init(**

**void** \**buf***,**

**int** *count***,**

**MPI\_Datatype** *datatype***,**

**int** *dest***,**

**int** *tag***,**

**MPI\_Comm** *comm***,**

**MPI\_Request** \**request*

**);**

**Parameters**

*buf*

[in] initial address of send buffer (choice)

*count*

[in] number of elements sent (integer)

*datatype*

[in] type of each element (handle)

*dest*

[in] rank of destination (integer)

*tag*

[in] message tag (integer)

*comm*

[in] communicator (handle)

*request*

[out] communication request (handle)

**Remarks**

Creates a persistent communication object for a synchronous mode send operation.

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a **persistent** communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a "half-channel." It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another. It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Ssend\_init](http://mpi.deino.net/mpi_functions/MPI_Ssend_init.html).

#include "mpi.h"  
#include <stdlib.h>  
   
int main(int argc, char \*argv[])  
{  
    MPI\_Request r;  
    MPI\_Status s;  
    int flag;  
    int buf[10];  
    int rbuf[10];  
    int tag = 27;  
    int dest = 0;  
    int rank, size, i;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( &argc, &argv );  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)( MPI\_COMM\_WORLD, &size );  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)( MPI\_COMM\_WORLD, &rank );  
   
    /\* Create a persistent synchronous send request \*/  
    [MPI\_Ssend\_init](http://mpi.deino.net/mpi_functions/MPI_Ssend_init.html)( buf, 10, MPI\_INT, dest, tag, MPI\_COMM\_WORLD, &r );  
   
    /\* Use that request \*/  
    if (rank == 0) {  
        int i;  
        MPI\_Request \*rr = (MPI\_Request \*)malloc(size \* sizeof(MPI\_Request));  
        for (i=0; i<size; i++) {  
            [MPI\_Irecv](http://mpi.deino.net/mpi_functions/MPI_Irecv.html)( rbuf, 10, MPI\_INT, i, tag, MPI\_COMM\_WORLD, &rr[i] );  
        }  
        [MPI\_Start](http://mpi.deino.net/mpi_functions/MPI_Start.html)( &r );  
        [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)( &r, &s );  
        [MPI\_Waitall](http://mpi.deino.net/mpi_functions/MPI_Waitall.html)( size, rr, MPI\_STATUSES\_IGNORE );  
        free(rr);  
    }  
    else {  
        [MPI\_Start](http://mpi.deino.net/mpi_functions/MPI_Start.html)( &r );  
        [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)( &r, &s );  
    }  
   
    [MPI\_Request\_free](http://mpi.deino.net/mpi_functions/MPI_Request_free.html)( &r );  
   
   
    if (rank == 0)  
    {  
        MPI\_Request sr;  
        /\* Create a persistent receive request \*/  
        [MPI\_Recv\_init](http://mpi.deino.net/mpi_functions/MPI_Recv_init.html)( rbuf, 10, MPI\_INT, MPI\_ANY\_SOURCE, tag, MPI\_COMM\_WORLD, &r );  
        [MPI\_Isend](http://mpi.deino.net/mpi_functions/MPI_Isend.html)( buf, 10, MPI\_INT, 0, tag, MPI\_COMM\_WORLD, &sr );  
        for (i=0; i<size; i++) {  
            [MPI\_Start](http://mpi.deino.net/mpi_functions/MPI_Start.html)( &r );  
            [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)( &r, &s );  
        }  
        [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)( &sr, &s );  
        [MPI\_Request\_free](http://mpi.deino.net/mpi_functions/MPI_Request_free.html)( &r );  
    }  
    else {  
        [MPI\_Send](http://mpi.deino.net/mpi_functions/MPI_Send.html)( buf, 10, MPI\_INT, 0, tag, MPI\_COMM\_WORLD );  
    }  
   
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Wait**

Waits for an MPI request to complete

**int MPI\_Wait(**

**MPI\_Request** \**request***,**

**MPI\_Status** \**status*

**);**

**Parameters**

*request*

[in] request (handle)

*status*

[out] status object (Status). May be MPI\_STATUS\_IGNORE.

**Remarks**

A call to MPI\_WAIT returns when the operation identified by request is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI\_WAIT and the request handle is set to MPI\_REQUEST\_NULL. MPI\_WAIT is a non-local operation.

The call returns, in status, information on the completed operation. The content of the status object for a receive operation can be accessed. The status object for a send operation may be queried by a call to MPI\_TEST\_CANCELLED.

One is allowed to call MPI\_WAIT with a null or inactive request argument. In this case the operation returns immediately with empty status.

*Advice to users.*

Successful return of MPI\_WAIT after a MPI\_IBSEND implies that the user send buffer can be reused --- i.e., data has been sent out or copied into a buffer attached with MPI\_BUFFER\_ATTACH. Note that, at this point, we can no longer cancel the send. If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of MPI\_CANCEL (always being able to free program space that was committed to the communication subsystem).

**Note on status for send operations**

For send operations, the only use of status is for [MPI\_Test\_cancelled](http://mpi.deino.net/mpi_functions/MPI_Test_cancelled.html) or in the case that there is an error, in which case the MPI\_ERROR field of status will be set.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html).

#include "mpi.h"  
#include <stdio.h>  
   
int main(int argc, char \*argv[])  
{  
    int myid, numprocs, left, right;  
    int buffer[10], buffer2[10];  
    MPI\_Request request;  
    MPI\_Status status;  
   
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc,&argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &numprocs);  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &myid);  
   
    right = (myid + 1) % numprocs;  
    left = myid - 1;  
    if (left < 0)  
        left = numprocs - 1;  
   
    [MPI\_Irecv](http://mpi.deino.net/mpi_functions/MPI_Irecv.html)(buffer, 10, MPI\_INT, left, 123, MPI\_COMM\_WORLD, &request);  
    [MPI\_Send](http://mpi.deino.net/mpi_functions/MPI_Send.html)(buffer2, 10, MPI\_INT, right, 123, MPI\_COMM\_WORLD);  
    [MPI\_Wait](http://mpi.deino.net/mpi_functions/MPI_Wait.html)(&request, &status);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Waitall**

Waits for all given MPI Requests to complete

**int MPI\_Waitall(**

**int** *count***,**

**MPI\_Request** *array\_of\_requests*[]**,**

**MPI\_Status** *array\_of\_statuses*[]

**);**

**Parameters**

*count*

[in] list length (integer)

*array\_of\_requests*

[in] array of request handles (array of handles)

*array\_of\_statuses*

[out] array of status objects (array of Statuses). May be MPI\_STATUSES\_IGNORE.

**Remarks**

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The i-th entry in array\_of\_statuses is set to the return status of the i-th operation. Requests that were created by nonblocking communication operations are deallocated and the corresponding handles in the array are set to MPI\_REQUEST\_NULL. The list may contain null or inactive handles. The call sets to empty the status of each such entry.

The error-free execution of MPI\_WAITALL(count, array\_of\_requests, array\_of\_statuses) has the same effect as the execution of   
MPI\_WAIT(&array\_of\_request[i], &array\_of\_statuses[i]), for i=0 ,..., count-1, in some arbitrary order. MPI\_WAITALL with an array of length one is equivalent to MPI\_WAIT.

When one or more of the communications completed by a call to MPI\_WAITALL fail, it is desireable to return specific information on each communication. The function MPI\_WAITALL will return in such case the error code MPI\_ERR\_IN\_STATUS and will set the error field of each status to a specific error code. This code will be MPI\_SUCCESS, if the specific communication completed; it will be another specific error code, if it failed; or it can be MPI\_ERR\_PENDING if it has neither failed nor completed. The function MPI\_WAITALL will return MPI\_SUCCESS if no request had an error, or will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

*Rationale.*

This design streamlines error handling in the application. The application code need only test the (single) function result to determine if an error has occurred. It needs to check each individual status only when an error occurred. (*End Rationale)*

If one or more of the requests completes with an error, MPI\_ERR\_IN\_STATUS is returned. An error value will be present is elements of array\_of\_status associated with the requests. Likewise, the MPI\_ERROR field in the status elements associated with requests that have successfully completed will be MPI\_SUCCESS. Finally, those requests that have not completed will have a value of MPI\_ERR\_PENDING.

While it is possible to list a request handle more than once in the array\_of\_requests, such an action is considered erroneous and may cause the program to unexecpectedly terminate or produce incorrect results.

**Note on status for send operations**

For send operations, the only use of status is for [MPI\_Test\_cancelled](http://mpi.deino.net/mpi_functions/MPI_Test_cancelled.html) or in the case that there is an error, in which case the MPI\_ERROR field of status will be set.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Waitall](http://mpi.deino.net/mpi_functions/MPI_Waitall.html).

#include "mpi.h"  
#include <stdio.h>  
  
int main(int argc, char \*argv[])  
{  
    int rank, size;  
    int i;  
    int buffer[400];  
    MPI\_Request request[4];  
    MPI\_Status status[4];  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc, &argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &size);  
    if (size != 4)  
    {  
        printf("Please run with 4 processes.\n");fflush(stdout);  
        [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
        return 1;  
    }  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &rank);  
  
    if (rank == 0)  
    {  
        for (i=0; i<size \* 100; i++)  
            buffer[i] = i/100;  
        for (i=0; i<size-1; i++)  
        {  
            [MPI\_Isend](http://mpi.deino.net/mpi_functions/MPI_Isend.html)(&buffer[i\*100], 100, MPI\_INT, i+1, 123, MPI\_COMM\_WORLD, &request[i]);  
        }  
        [MPI\_Waitall](http://mpi.deino.net/mpi_functions/MPI_Waitall.html)(size-1, request, status);  
    }  
    else  
    {  
        [MPI\_Recv](http://mpi.deino.net/mpi_functions/MPI_Recv.html)(buffer, 100, MPI\_INT, 0, 123, MPI\_COMM\_WORLD, &status[0]);  
        printf("%d: buffer[0] = %d\n", rank, buffer[0]);fflush(stdout);  
    }  
  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Waitany**

Waits for any specified MPI Request to complete

**int MPI\_Waitany(**

**int** *count***,**

**MPI\_Request** *array\_of\_requests*[]**,**

**int** \**index***,**

**MPI\_Status** \**status*

**);**

**Parameters**

*count*

[in] list length (integer)

*array\_of\_requests*

[in/out] array of requests (array of handles)

*index*

[out] index of handle for operation that completed (integer). In the range 0 to count-1. In Fortran, the range is 1 to count.

*status*

[out] status object (Status). May be MPI\_STATUS\_IGNORE.

**Remarks**

Blocks until one of the operations associated with the active requests in the array has completed. If more then one operation is enabled and can terminate, one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing communication. (The array is indexed from zero in C, and from one in Fortran.) If the request was allocated by a nonblocking communication operation, then it is deallocated and the request handle is set to MPI\_REQUEST\_NULL.

The array\_of\_requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns immediately with index = MPI\_UNDEFINED, and a empty status.

The execution of MPI\_WAITANY(count, array\_of\_requests, index, status) has the same effect as the execution of MPI\_WAIT(&array\_of\_requests[i], status), where i is the value returned by index (unless the value of index is MPI\_UNDEFINED). MPI\_WAITANY with an array containing one active entry is equivalent to MPI\_WAIT.

If all of the requests are MPI\_REQUEST\_NULL, then index is returned as MPI\_UNDEFINED, and status is returned as an empty status.

While it is possible to list a request handle more than once in the array\_of\_requests, such an action is considered erroneous and may cause the program to unexecpectedly terminate or produce incorrect results.

**Note on status for send operations**

For send operations, the only use of status is for [MPI\_Test\_cancelled](http://mpi.deino.net/mpi_functions/MPI_Test_cancelled.html) or in the case that there is an error, in which case the MPI\_ERROR field of status will be set.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Waitany](http://mpi.deino.net/mpi_functions/MPI_Waitany.html).

#include "mpi.h"  
#include <stdio.h>  
  
int main(int argc, char \*argv[])  
{  
    int rank, size;  
    int i, index;  
    int buffer[400];  
    MPI\_Request request[4];  
    MPI\_Status status[4];  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc, &argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &size);  
    if (size != 4)  
    {  
        printf("Please run with 4 processes.\n");fflush(stdout);  
        [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
        return 1;  
    }  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &rank);  
  
    if (rank == 0)  
    {  
        for (i=0; i<size \* 100; i++)  
            buffer[i] = i/100;  
        for (i=0; i<size-1; i++)  
        {  
            [MPI\_Isend](http://mpi.deino.net/mpi_functions/MPI_Isend.html)(&buffer[i\*100], 100, MPI\_INT, i+1, 123, MPI\_COMM\_WORLD, &request[i]);  
        }  
        for (i=0; i<size-1; i++)  
        {  
            [MPI\_Waitany](http://mpi.deino.net/mpi_functions/MPI_Waitany.html)(size-1, request, &index, status);  
        }  
    }  
    else  
    {  
        [MPI\_Recv](http://mpi.deino.net/mpi_functions/MPI_Recv.html)(buffer, 100, MPI\_INT, 0, 123, MPI\_COMM\_WORLD, &status[0]);  
        printf("%d: buffer[0] = %d\n", rank, buffer[0]);fflush(stdout);  
    }  
  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Waitsome**

Waits for some given MPI Requests to complete

**int MPI\_Waitsome(**

**int** *incount***,**

**MPI\_Request** *array\_of\_requests*[]**,**

**int** \**outcount***,**

**int** *array\_of\_indices*[]**,**

**MPI\_Status** *array\_of\_statuses*[]

**);**

**Parameters**

*incount*

[in] length of array\_of\_requests (integer)

*array\_of\_requests*

[in] array of requests (array of handles)

*outcount*

[out] number of completed requests (integer)

*array\_of\_indices*

[out] array of indices of operations that completed (array of integers)

*array\_of\_statuses*

[out] array of status objects for operations that completed (array of Status). May be MPI\_STATUSES\_IGNORE.

**Remarks**

Waits until at least one of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list array\_of\_requests that have completed. Returns in the first outcount locations of the array array\_of\_indices the indices of these operations (index within the array array\_of\_requests; the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array\_of\_status the status for these completed operations. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to MPI\_REQUEST\_NULL.

If the list contains no active handles, then the call returns immediately with outcount = MPI\_UNDEFINED.

When one or more of the communications completed by MPI\_WAITSOME fails, then it is desirable to return specific information on each communication. The arguments outcount, array\_of\_indices and array\_of\_statuses will be adjusted to indicate completion of all communications that have succeeded or failed. The call will return the error code MPI\_ERR\_IN\_STATUS and the error field of each status returned will be set to indicate success or to indicate the specific error that occurred. The call will return MPI\_SUCCESS if no request resulted in an error, and will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

The array of indicies are in the range 0 to incount - 1 for C and in the range 1 to incount for Fortran.

Null requests are ignored; if all requests are null, then the routine returns with outcount set to MPI\_UNDEFINED.

While it is possible to list a request handle more than once in the array\_of\_requests, such an action is considered erroneous and may cause the program to unexecpectedly terminate or produce incorrect results.

[MPI\_Waitsome](http://mpi.deino.net/mpi_functions/MPI_Waitsome.html) provides an interface much like the Unix select or poll calls and, in a high qualilty implementation, indicates all of the requests that have completed when [MPI\_Waitsome](http://mpi.deino.net/mpi_functions/MPI_Waitsome.html) is called. However, [MPI\_Waitsome](http://mpi.deino.net/mpi_functions/MPI_Waitsome.html) only guarantees that at least one request has completed; there is no guarantee that *all* completed requests will be returned, or that the entries in array\_of\_indices will be in increasing order. Also, requests that are completed while [MPI\_Waitsome](http://mpi.deino.net/mpi_functions/MPI_Waitsome.html) is executing may or may not be returned, depending on the timing of the completion of the message.

**Note on status for send operations**

For send operations, the only use of status is for [MPI\_Test\_cancelled](http://mpi.deino.net/mpi_functions/MPI_Test_cancelled.html) or in the case that there is an error, in which case the MPI\_ERROR field of status will be set.

**Thread and Interrupt Safety**

This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe.

**Example Code**

The following sample code illustrates [MPI\_Waitsome](http://mpi.deino.net/mpi_functions/MPI_Waitsome.html).

#include "mpi.h"  
#include <stdio.h>  
  
int main(int argc, char \*argv[])  
{  
    int rank, size;  
    int i, index[4], count, remaining;  
    int buffer[400];  
    MPI\_Request request[4];  
    MPI\_Status status[4];  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc, &argv);  
    [MPI\_Comm\_size](http://mpi.deino.net/mpi_functions/MPI_Comm_size.html)(MPI\_COMM\_WORLD, &size);  
    if (size > 4)  
    {  
        printf("Please run with 4 processes.\n");fflush(stdout);  
        [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
        return 1;  
    }  
    [MPI\_Comm\_rank](http://mpi.deino.net/mpi_functions/MPI_Comm_rank.html)(MPI\_COMM\_WORLD, &rank);  
  
    if (rank == 0)  
    {  
        for (i=0; i<size \* 100; i++)  
            buffer[i] = i/100;  
        for (i=0; i<size-1; i++)  
        {  
            [MPI\_Isend](http://mpi.deino.net/mpi_functions/MPI_Isend.html)(&buffer[i\*100], 100, MPI\_INT, i+1, 123, MPI\_COMM\_WORLD, &request[i]);  
        }  
        remaining = size-1;  
        while (remaining > 0)  
        {  
            [MPI\_Waitsome](http://mpi.deino.net/mpi_functions/MPI_Waitsome.html)(size-1, request, &count, index, status);  
            if (count > 0)  
            {  
                printf("%d sends completed\n", count);fflush(stdout);  
                remaining = remaining - count;  
            }  
        }  
    }  
    else  
    {  
        [MPI\_Recv](http://mpi.deino.net/mpi_functions/MPI_Recv.html)(buffer, 100, MPI\_INT, 0, 123, MPI\_COMM\_WORLD, &status[0]);  
        printf("%d: buffer[0] = %d\n", rank, buffer[0]);fflush(stdout);  
    }  
  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return 0;  
}

**MPI\_Wtick**

Returns the resolution of [MPI\_Wtime](http://mpi.deino.net/mpi_functions/MPI_Wtime.html)

double MPI\_Wtick( void );

**Return value**

Time in seconds of resolution of MPI\_Wtime.

**Remarks**

MPI\_WTICK returns the resolution of MPI\_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI\_WTICK should be http://mpi.deino.net/mpi_functions/img196.gif

**Example Code**

The following sample code illustrates [MPI\_Wtick](http://mpi.deino.net/mpi_functions/MPI_Wtick.html).

#include "mpi.h"  
#include <stdio.h>  
  
int main( int argc, char \*argv[] )  
{  
    double tick;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( 0, 0 );  
    tick = [MPI\_Wtick](http://mpi.deino.net/mpi_functions/MPI_Wtick.html)();  
    printf("A single MPI tick is %0.9f seconds\n", tick);  
    fflush(stdout);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)( );  
    return 0;  
}

**MPI\_Wtime**

Returns an elapsed time on the calling processor

double MPI\_Wtime( void );

**Return value**

Time in seconds since an arbitrary time in the past.

**Remarks**

MPI\_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.

The "time in the past" is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

The times returned are local to the node that called them. There is no requirement that different nodes return "the same time."

This is intended to be a high-resolution, elapsed (or wall) clock. See MPI\_WTICK to determine the resolution of MPI\_WTIME. If the attribute MPI\_WTIME\_IS\_GLOBAL is defined and true, then the value is synchronized across all processes in MPI\_COMM\_WORLD.

**Example Code**

The following sample code illustrates [MPI\_Wtime](http://mpi.deino.net/mpi_functions/MPI_Wtime.html).

#include "mpi.h"  
#include <windows.h>  
#include <stdio.h>  
  
int main( int argc, char \*argv[] )  
{  
    double t1, t2;  
  
    [MPI\_Init](http://mpi.deino.net/mpi_functions/MPI_Init.html)( 0, 0 );  
    t1 = [MPI\_Wtime](http://mpi.deino.net/mpi_functions/MPI_Wtime.html)();  
    Sleep(1000);  
    t2 = [MPI\_Wtime](http://mpi.deino.net/mpi_functions/MPI_Wtime.html)();  
    printf("[MPI\_Wtime](http://mpi.deino.net/mpi_functions/MPI_Wtime.html) measured a 1 second sleep to be: %1.2f\n", t2-t1);fflush(stdout);  
    [MPI\_Finalize](http://mpi.deino.net/mpi_functions/MPI_Finalize.html)( );  
    return 0;  
}