# MPI Summary

Generated by Doxygen 1.8.9.1

Thu Jul 9 2015 14:49:44

# Contents

# Chapter 1

# File Documentation

## 1.1 summary.h File Reference

```
#include "mpi.h"
```

### Functions

- int MPI_Init (int ∗argc, char ∗∗∗argv)

  *Initialize the MPI library.*
- int MPI_Comm_size (MPI_Comm comm, int ∗size)

  *Obtaining the total number of processes of the program.*
- int MPI_Comm_rank (MPI_Comm comm, int ∗rank)
- int MPI_Send (const void ∗buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

  *Sending Data using MPI Point-to-Point Communication.*
- int MPI_Recv (void ∗buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status ∗status)

  *Receiving Data with MPI Point-to-Point Communication.*
- int MPI_Get_count (const MPI_Status ∗status, MPI_Datatype datatype, int ∗count)

  *The number of data elements transmitted to the receiver can be obtained from the data structure status.*
- int MPI_Sendrecv (const 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)

  *Data Exchange with MPI Sendrecv()*
- 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)

  *Like MPI_Sendrecv.*
- int MPI_Isend (const void ∗buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI↩ _Request ∗request)

  *Non-blocking send operation.*
- int MPI_Irecv (void ∗buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_↩ Request ∗request)

  *Non-blocking receive operation.*
- int MPI_Test (MPI_Request ∗request, int ∗flag, MPI_Status ∗status)

  *Querying the status of a non-blocking communication operation.*
- int MPI_Wait (MPI_Request ∗request, MPI_Status ∗status)

  *Waiting for the completion of a communication operation.*
- int MPI_Bsend (const void ∗buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

*Blocking send operation in buffered mode.*

- int MPI_Ibsend (const void ∗buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, M←PI_Request ∗request)

  *Non-blocking send operation in buffered mode.*
- int MPI_Buffer_attach (void ∗buffer, int size)

  *Provision of a buffer.*
- int MPI_Buffer_detach (void ∗buffer, int ∗size)

  *Detaching a buffer previously provided.*
- int MPI_Bcast (void ∗buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

  *Broadcast Operation.*
- int MPI_Reduce (const void ∗sendbuf, void ∗recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

  *Accumulation Operation.*
- int MPI_Op_create (MPI_User_function ∗function, int commute, MPI_Op ∗op)

  *User defined accumulation operation.*
- int MPI_Gather (const void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

  *Each of the participating n processes provides a block of data that is collected at the root process.*
- int MPI_Gatherv (const void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, const int recvcounts[ ], const int displs[ ], MPI_Datatype recvtype, int root, MPI_Comm comm)

  *More general vector-based MPI Gatherv operation.*
- int MPI_Scatter (const void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

  *Scatter: The root process provides a data block (with the same size but possibly different elements) for each participating process.*
- int MPI_Allgather (const void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

  *Multi-broadcast operation: Each process sends the same block of data to each other process.*
- int MPI_Allgatherv (const void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, const int recvcounts[ ], const int displs[ ], MPI_Datatype recvtype, MPI_Comm comm)

  *Syntax of the vector-based MPI operation MPI_Allgatherv()*
- int MPI_Allreduce (const void ∗sendbuf, void ∗recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI←_Comm comm)

  *Multi-accumulation Operation.*
- int MPI_Alltoall (const void ∗sendbuf, int sendcount, MPI_Datatype sendtype, void ∗recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

  *Total Exchange.*
- int MPI_Alltoallv (const void ∗sendbuf, const int sendcounts[ ], const int sdispls[ ], MPI_Datatype sendtype, void ∗recvbuf, const int recvcounts[ ], const int rdispls[ ], MPI_Datatype recvtype, MPI_Comm comm)

  *Syntax of the more general vector-based version for data blocks of different sizes.*
- int MPI_Comm_group (MPI_Comm comm, MPI_Group ∗group)

  *The corresponding process group to a given communicator comm can be obtained by calling.*
- int MPI_Group_union (MPI_Group group1, MPI_Group group2, MPI_Group ∗newgroup)

  *Union of two existing groups group1 and group2.*
- int MPI_Group_intersection (MPI_Group group1, MPI_Group group2, MPI_Group ∗newgroup)

  *The intersection of two groups is obtained by calling this function.*
- int MPI_Group_difference (MPI_Group group1, MPI_Group group2, MPI_Group ∗newgroup)

  *The set difference of two groups.*
- int MPI_Group_incl (MPI_Group group, int n, const int ranks[ ], MPI_Group ∗newgroup)

  *Construction of a subset of an existing group.*
- int MPI_Group_excl (MPI_Group group, int n, const int ranks[ ], MPI_Group ∗newgroup)

  *Deletion of processes from a group.*

- int MPI_Group_size (MPI_Group group, int ∗size)

  *The size of a process group* `group`
- int MPI_Group_rank (MPI_Group group, int ∗rank)

  *The rank of the calling process in a group.*
- int MPI_Group_compare (MPI_Group group1, MPI_Group group2, int ∗result)

  *To check whether two process groups describe the same process group.*
- int MPI_Comm_create (MPI_Comm comm, MPI_Group group, MPI_Comm ∗newcomm)

  *Generation of a new intra-communicator to a given group of processes.*
- int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm ∗newcomm)

  *Splitting of a communicator.*
- int MPI_Cart_create (MPI_Comm old_comm, int ndims, const int dims[ ], const int periods[ ], int reorder, M←PI_Comm ∗comm_cart)

  *Definition of a virtual Cartesian grid structure of arbitrary dimension.*
- int MPI_Dims_create (int nnodes, int ndims, int dims[ ])

  *Select a balanced distribution of the processes for the different dimensions.*
- int MPI_Cart_rank (MPI_Comm comm, const int coords[ ], int ∗rank)
- int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int coords[ ])

  *Translation of group ranks into Cartesian coordinates.*
- int MPI_Cart_shift (MPI_Comm comm, int direction, int disp, int ∗rank_source, int ∗rank_dest)

  *Determining the neighboring processes in each dimension of the grid.*
- int MPI_Cart_sub (MPI_Comm comm, const int remain_dims[ ], MPI_Comm ∗new_comm)

  *A virtual topology can be partitioned into subgrids.*
- int MPI_Cartdim_get (MPI_Comm comm, int ∗ndims)

  *Number of dimensions of the virtual grid.*
- int MPI_Cart_get (MPI_Comm comm, int maxdims, int dims[ ], int periods[ ], int coords[ ])

  *Cartesian coordinates of the calling process within the virtual grid associated with communicator comm.*
- double MPI_Wtime (void)

  *Returns an elapsed time on the calling processor.*
- int MPI_Abort (MPI_Comm comm, int errorcode)

  *Abortion of the execution of all processes of a communicator.*
- int MPI_Info_create (MPI_Info ∗info)

  *a new structure of type* `MPI_Info` *is created*
- int MPI_Info_set (MPI_Info info, const char ∗key, const char ∗value)

  *Adds a key/value pair to info.*
- int MPI_Info_get (MPI_Info info, const char ∗key, int valuelen, char ∗value, int ∗flag)

  *Retrieves the value associated with a* `key` *in an info object.*
- int MPI_Info_delete (MPI_Info info, const char ∗key)

  *Pair* `(key, value)` *can be removed by this function.*
- int MPI_Comm_spawn (const char ∗command, char ∗argv[ ], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm ∗intercomm, int array_of_errcodes[ ])

  *Spawns a number of identical binaries.*
- int MPI_Comm_spawn_multiple (int count, char ∗array_of_commands[ ], char ∗∗array_of_argv[ ], const int array_of_maxprocs[ ], const MPI_Info array_of_info[ ], int root, MPI_Comm comm, MPI_Comm ∗intercomm, int array_of_errcodes[ ])

  *Spawns multiple binaries, or the same binary with multiple sets of arguments.*
- int MPI_Win_fence (int assert, MPI_Win win)

  *Global synchronization of a process group of a window.*
- int MPI_Win_start (MPI_Group group, int assert, MPI_Win win)

  *Starts an RMA access epoch for win.*
- int MPI_Win_complete (MPI_Win win)

  *Completes an RMA access epoch on win started by a call to MPI_Win_start.*

- int MPI_Win_post (MPI_Group group, int assert, MPI_Win win)

    *Starts an RMA exposure epoch for the local window associated with win.*
- int MPI_Win_wait (MPI_Win win)

    *Completes an RMA exposure epoch started by a call to MPI_Win_post on win.*
- int MPI_Win_test (MPI_Win win, int ∗flag)

    *Attempts to complete an RMA exposure epoch; a nonblocking version of MPI_Win_wait.*
- int MPI_Win_lock (int lock_type, int rank, int assert, MPI_Win win)

    *Setting a lock before accessing.*
- int MPI_Win_unlock (int rank, MPI_Win win)

    *Releasing a lock after access.*
- int MPI_Accumulate (const void ∗origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

    *Accumulation of data in the memory of another process.*
- int MPI_Get (void ∗origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)

    *Reading a data block from the memory of another process.*
- int MPI_Put (const void ∗origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI↩ _Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)

    *Copies data from the origin memory to the target.*
- int MPI_Win_create (void ∗base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win ∗win)

    *One-sided MPI call that returns a window object for RMA operations.*
- int MPI_Win_free (MPI_Win ∗win)

    *Frees the window object and returns a null handle.*
- int main (int argc, char ∗∗argv)

### 1.1.1 Function Documentation

#### 1.1.1.1 int main ( int *argc,* char ∗∗ *argv* )

#### 1.1.1.2 int MPI_Abort ( MPI_Comm *comm,* int *errorcode* )

Abortion of the execution of all processes of a communicator.

#### 1.1.1.3 int MPI_Accumulate ( const void ∗ *origin_addr,* int *origin_count,* MPI_Datatype *origin_datatype,* int *target_rank,* MPI_Aint *target_disp,* int *target_count,* MPI_Datatype *target_datatype,* MPI_Op *op,* MPI_Win *win* )

Accumulation of data in the memory of another process.

Combines the contents of the origin buffer with that of a target buffer.

**See also**

MPI_Win_create

#### 1.1.1.4 int MPI_Allgather ( const void ∗ *sendbuf,* int *sendcount,* MPI_Datatype *sendtype,* void ∗ *recvbuf,* int *recvcount,* MPI_Datatype *recvtype,* MPI_Comm *comm* )

Multi-broadcast operation: Each process sends the same block of data to each other process.

Each process performs a single-broadcast operation.

Each process provides a receive buffer recvbuf in which all received data blocks are collected in rank order of the sending processes.

A multi-broadcast does not have a distinguished root process.

Example: each process contributes a send buffer with 100 integer values which are made available by a multi-broadcast operation to all processes:

int sbuf[100], gsize, *rbuf; MPI Comm size (comm, &gsize); rbuf = (int*) malloc (gsize∗100∗sizeof(int)); MPI Allgather (sbuf, 100, MPI INT, rbuf, 100, MPI INT, comm);

**Parameters**

| | |
|---|---|
| *sendbuf* | `send buffer` proviced by each of the participating processes. |

#### 1.1.1.5 int MPI_Allgatherv ( const void ∗ *sendbuf,* int *sendcount,* MPI_Datatype *sendtype,* void ∗ *recvbuf,* const int *recvcounts[ ],* const int *displs[ ],* MPI_Datatype *recvtype,* MPI_Comm *comm* )

Syntax of the vector-based MPI operation MPI_Allgatherv()

Each process provides a receive buffer recvbuf in which all received data blocks are collected in rank order of the sending processes.

A multi-broadcast does not have a distinguished root process.

#### 1.1.1.6 int MPI_Allreduce ( const void ∗ *sendbuf,* void ∗ *recvbuf,* int *count,* MPI_Datatype *datatype,* MPI_Op *op,* MPI_Comm *comm* )

Multi-accumulation Operation.

Each process provides a data block of the same size.

The data blocks are accumulated with a reduction operation -> multi-accumulation equals a single-accumulation with a subsequent broadcast.

**Parameters**

| | |
|---|---|
| *sendbuf* | is the **local buffer** in which each process provices its local data |
| *recvbuf* | is the local buffer of each process in which the accumulated result is **collected**. |

#### 1.1.1.7 int MPI_Alltoall ( const void ∗ *sendbuf,* int *sendcount,* MPI_Datatype *sendtype,* void ∗ *recvbuf,* int *recvcount,* MPI_Datatype *recvtype,* MPI_Comm *comm* )

Total Exchange.

Each process provides a different block of data for each other process Each process collects the data blocks provided for this particular process.

The blocks are arranged in rank order of the target processes.

**Parameters**

| | |
|---|---|
| *sendbuf* | is the **send buffer** in which each process provides for each process a block of data with **sendcount** elements of type **sendtype** |

#### 1.1.1.8 int MPI_Alltoallv ( const void ∗ *sendbuf,* const int *sendcounts[ ],* const int *sdispls[ ],* MPI_Datatype *sendtype,* void ∗ *recvbuf,* const int *recvcounts[ ],* const int *rdispls[ ],* MPI_Datatype *recvtype,* MPI_Comm *comm* )

Syntax of the more general vector-based version for data blocks of different sizes.

#### 1.1.1.9 int MPI_Bcast ( void ∗ *buffer,* int *count,* MPI_Datatype *datatype,* int *root,* MPI_Comm *comm* )

Broadcast Operation.

The root process root sends the same data block to all other processes of the group.

All global communication operations are blocking in MPI.

#### 1.1.1.10 int MPI_Bsend ( const void ∗ *buf,* int *count,* MPI_Datatype *datatype,* int *dest,* int *tag,* MPI_Comm *comm* )

Blocking send operation in buffered mode.

#### 1.1.1.11 int MPI_Buffer_attach ( void ∗ *buffer,* int *size* )

Provision of a buffer.

The buffer space to be used by the runtime system must be provided by the programmer

**Parameters**

| | |
|---|---|
| *size* | is the size of the buffer buffer in bytes. |

#### 1.1.1.12 int MPI_Buffer_detach ( void ∗ *buffer,* int ∗ *size* )

Detaching a buffer previously provided.

#### 1.1.1.13 int MPI_Cart_coords ( MPI_Comm *comm,* int *rank,* int *maxdims,* int *coords[ ]* )

Translation of group ranks into Cartesian coordinates.

The cartesian coordinates of the process are returned in the array `coords`.

**Parameters**

| | |
|---|---|
| *rank* | Contains the process number |
| *dims* | Denotes the number of dimensions in the virtual grid defined for communicator `comm`. |

#### 1.1.1.14 int MPI_Cart_create ( MPI_Comm *old_comm,* int *ndims,* const int *dims[ ],* const int *periods[ ],* int *reorder,* MPI_Comm ∗ *comm_cart* )

Definition of a virtual Cartesian grid structure of arbitrary dimension.

The array `periods` of size `ndims` specifies for each dimension whether the grid is periodic (entry 1) or not (entry 0) in this dimension.

**Parameters**

| | |
|---|---|
| *old_comm* | Is the original communicator **without topology** |
| *ndims* | Specifies the **number** of dimensions of hte grid to be created |
| *dims* | Is an integer array with `ndims` elements where `dims[i]` denotes the **total number of processes in dimension `i`** |
| *reorder* | For `reorder = false`, the processes in `comm_cart` have the same rank as in `old←_comm` |

#### 1.1.1.15 int MPI_Cart_get ( MPI_Comm *comm,* int *maxdims,* int *dims[ ],* int *periods[ ],* int *coords[ ]* )

Cartesian coordinates of the calling process within the virtual grid associated with communicator comm.

Where `maxdims` is the number of dimensions of the virtual topology, and `dims`, `periods`, and `coords` are arrays of size `maxdims`.

The arrays `dims` and periods have the same meaning as for MPI_Cart_create().

The array `coords` is used to return the coordinates.

#### 1.1.1.16 int MPI_Cart_rank ( MPI_Comm *comm,* const int *coords[ ],* int ∗ *rank* )

**Parameters**

| | |
|---:|---|
| *Translation* | of Cartesian coordinates into group ranks |

The call translates the Cartesian coordinates of a process provided in the array `coords` into the group rank according to the virtual grid associated with `comm`.

#### 1.1.1.17 int MPI_Cart_shift ( MPI_Comm *comm,* int *direction,* int *disp,* int ∗ *rank_source,* int ∗ *rank_dest* )

Determining the neighboring processes in each dimension of the grid.

The result of the call is that `rank_dest` contains the group rank of the neighboring process in the specified dimension and distance; `rank_source` returns the rank of the process for which the calling process is the neighbor in the specified dimension and distance.

- positive value: request neighbors in upward direction;
- negative value: request neighbors in downward direction.

**Parameters**

| | |
|---:|---|
| *direction* | Specifies the dimension for which the neighboring process should be determined. |
| *disp* | Specifies the displacement desired. |

#### 1.1.1.18 int MPI_Cart_sub ( MPI_Comm *comm,* const int *remain_dims[ ],* MPI_Comm ∗ *new_comm* )

A virtual topology can be partitioned into subgrids.

The subgrid selection is controlled by the array `remain_dims` which contains an entry for each dimension of the original grid.

Setting `remain_dims[i]=1` means that the ith dimension is kept in the subgrid;

`remain_dims[i]=0` means that the ith dimension is dropped in the subgrid.

If a dimension `i` does not exist in the subgrid, the size of dimension `i` defines the number of subgrids that have been generated for this dimension.

**Parameters**

| | |
|---:|---|
| *comm* | Is the communicator for which the virtual topology has been defined; |
| *new_comm* | Denotes the new communicator for which the new topology as a subgrid of the original grid is defined. |

#### 1.1.1.19 int MPI_Cartdim_get ( MPI_Comm *comm,* int ∗ *ndims* )

Number of dimensions of the virtual grid.

#### 1.1.1.20 int MPI_Comm_create ( MPI_Comm *comm,* MPI_Group *group,* MPI_Comm ∗ *newcomm* )

Generation of a new intra-communicator to a given group of processes.

**All** processes of comm must call MPI_Comm_create() with **the same group** as an argument.

Result of the call: each calling process which is a member of group `group` obtains a pointer to the new communicator `newcomm`.

Processes not belonging to group get `MPI_COMM_NULL` as return value in `new_comm`.

**Parameters**

| | |
|---|---|
| *group* | Must specify a process gorup which is a subset of the process `group` associated with communicator `comm`. |

#### 1.1.1.21 int MPI_Comm_group ( MPI_Comm *comm,* MPI_Group ∗ *group* )

The corresponding process group to a given communicator comm can be obtained by calling.

`MPI_GROUP_EMPTY` denotes the empty process group.

#### 1.1.1.22 int MPI_Comm_rank ( MPI_Comm *comm,* int ∗ *rank* )

Obtaining the local process number

**Parameters**

| | |
|---|---|
| *comm* | The communicator (e.g. MPI_COMM_WORLD) |
| *rank* | The variable for my rank |

#### 1.1.1.23 int MPI_Comm_size ( MPI_Comm *comm,* int ∗ *size* )

Obtaining the total number of processes of the program.

**Parameters**

| | |
|---|---|
| *comm* | The communicator of the process group |
| *size* | The output variable for the total number of processes |

#### 1.1.1.24 int MPI_Comm_spawn ( const char ∗ *command,* char ∗ *argv[ ],* int *maxprocs,* MPI_Info *info,* int *root,* MPI_Comm *comm,* MPI_Comm ∗ *intercomm,* int *array_of_errcodes[ ]* )

Spawns a number of identical binaries.

New processes can be created in MPI-2 by this function

#### 1.1.1.25 int MPI_Comm_spawn_multiple ( int *count,* char ∗ *array_of_commands[ ],* char ∗∗ *array_of_argv[ ],* const int *array_of_maxprocs[ ],* const MPI_Info *array_of_info[ ],* int *root,* MPI_Comm *comm,* MPI_Comm ∗ *intercomm,* int *array_of_errcodes[ ]* )

Spawns multiple binaries, or the same binary with multiple sets of arguments.

Several different MPI programs with possibly different command line arguments can be split off as new processes by this function

#### 1.1.1.26 int MPI_Comm_split ( MPI_Comm *comm,* int *color,* int *key,* MPI_Comm ∗ *newcomm* )

Splitting of a communicator.

The process group associated with communicator `comm` is partitioned into a number of disjoint subgroups that equals the number of different values specified in `color`.

Each subgroup contains all processes that specify the same value for `color`.

The rank order of the processes within a subgroup is defined by the argument `key`.

If two processes specify the same value for `key` the order of the original group is used.

If a process specifies `color = MPI_UNDEFINED`, it is not a member of any of the subgroups generated.

Each participating process gets a pointer `new_comm` to the communicator of that subgroup which the process belongs to.

#### 1.1.1.27 int MPI_Dims_create ( int *nnodes,* int *ndims,* int *dims[ ]* )

Select a balanced distribution of the processes for the different dimensions.

In the case `dims[i] = 0` is specified for the call, `dims[i]` contains the number of processes in dimension `i` after the call.

The function tries to assign the same number of processes to each dimension.

The number of processes in a dimension `i` can be fixed by setting `dims[i]` to the desired number of processes before the call. The MPI runtime system sets the entries of the other, non-initialized entries of `dims` accordingly.

**Parameters**

| | |
|---|---|
| *nnodes* | Is the total number of processes in the grid |
| *ndims* | Is the number of dimensions in the grid to be defined |
| *dims* | Is an integer array of size `ndims`. |

#### 1.1.1.28 int MPI_Gather ( const void ∗ *sendbuf,* int *sendcount,* MPI_Datatype *sendtype,* void ∗ *recvbuf,* int *recvcount,* MPI_Datatype *recvtype,* int *root,* MPI_Comm *comm* )

Each of the participating n processes provides a block of data that is collected at the root process.

**Parameters**

| | |
|---|---|
| *sendbuf* | Send buffer that is provided by each participating process |
| *sendcount* | Number of data elements with data type `sendtype` |
| *recvbuf* | Receive buffer provided by the root process `root` that is large enough to hold all data elements sent. |

#### 1.1.1.29 int MPI_Gatherv ( const void ∗ *sendbuf,* int *sendcount,* MPI_Datatype *sendtype,* void ∗ *recvbuf,* const int *recvcounts[ ],* const int *displs[ ],* MPI_Datatype *recvtype,* int *root,* MPI_Comm *comm* )

More general vector-based MPI Gatherv operation.

Each process can provide a different number of elements.

Overlaps in the receive buffer must not occur; −> displs_`root` [i + 1] >= displs_`root` [i] + sendcount_`i` with recvcounts_`root`[i] = sendcount_`i`

**Parameters**

| | |
|---:|---|
| *sendcount* | Number of data elements to be sent |
| *recvcounts* | Array, where recvcounts[i] denotes the number of elements proviced by process i |
| *displs* | Array that specifies the positions of the data blocks in `recvbuf` |

#### 1.1.1.30 int MPI_Get ( void ∗ *origin_addr,* int *origin_count,* MPI_Datatype *origin_datatype,* int *target_rank,* MPI_Aint *target_disp,* int *target_count,* MPI_Datatype *target_datatype,* MPI_Win *win* )

Reading a data block from the memory of another process.

Copies data from the target memory to the origin.

**Parameters**

| | |
|---:|---|
| *origin_addr* | is the starting address of the receive buffer in the local memory of the calling process, |
| *origin_count* | specifies the number of elements from type `origin_type`, transferred to the receiving buffer. |
| *target_rank* | is the rank of the target process, i.e. the process to be read from |
| *win* | is the window object |

**See also**

MPI_Win_create

#### 1.1.1.31 int MPI_Get_count ( const MPI_Status ∗ *status,* MPI_Datatype *datatype,* int ∗ *count* )

The number of data elements transmitted to the receiver can be obtained from the data structure status.

**Parameters**

| | |
|---:|---|
| *status* | Pointer to the data structure returned by the corresponding call to MPI_Recv() |
| *datatype* | Data type of the elements |
| *count* | Address of a variable wich the number of elements received are returned |

#### 1.1.1.32 int MPI_Group_compare ( MPI_Group *group1,* MPI_Group *group2,* int ∗ *result* )

To check whether two process groups describe the same process group.

`res`:

- `MPI_IDENT` The groups `group1` and `group2` contain the **same processes in the same order**
- `MPI_SIMILAR` Both groups contain the **same processes** but different order
- `MPI_UNEQUAL` The two groups contain **different processes**

#### 1.1.1.33 int MPI_Group_difference ( MPI_Group *group1,* MPI_Group *group2,* MPI_Group ∗ *newgroup* )

The set difference of two groups.

Where the process order from `group1` is kept as well.

#### 1.1.1.34 int MPI_Group_excl ( MPI_Group *group,* int *n,* const int *ranks[ ],* MPI_Group ∗ *newgroup* )

Deletion of processes from a group.

The new group `new_group` is generated by deleting the processes with ranks `ranks[0]`, ..., `ranks[p-1]` from `group`.

#### 1.1.1.35 int MPI_Group_incl ( MPI_Group *group,* int *n,* const int *ranks[ ],* MPI_Group ∗ *newgroup* )

Construction of a subset of an existing group.

The call creates a new group new group with p processes which have ranks from 0 to p-1.

Process i is the process which has rank `ranks[i]` in group group.

The group group must contain at least p processes and the values `ranks[i]` must be valid process ranks in group which are different from each other.

**Parameters**

| | |
|---|---|
| *ranks* | is an integer array with p entries. |

#### 1.1.1.36 int MPI_Group_intersection ( MPI_Group *group1,* MPI_Group *group2,* MPI_Group ∗ *newgroup* )

The intersection of two groups is obtained by calling this function.

Where the process order from `group1` is kept for new group. The processes in `new_group` get successive ranks starting from 0.

#### 1.1.1.37 int MPI_Group_rank ( MPI_Group *group,* int ∗ *rank* )

The rank of the calling process in a group.

#### 1.1.1.38 int MPI_Group_size ( MPI_Group *group,* int ∗ *size* )

The size of a process group `group`

#### 1.1.1.39 int MPI_Group_union ( MPI_Group *group1,* MPI_Group *group2,* MPI_Group ∗ *newgroup* )

Union of two existing groups group1 and group2.

The processes in group1 keep their ranks from group1 and the processes in group2 which are not in group1 get subsequent ranks in consecutive order.

#### 1.1.1.40 int MPI_Ibsend ( const void ∗ *buf,* int *count,* MPI_Datatype *datatype,* int *dest,* int *tag,* MPI_Comm *comm,* MPI_Request ∗ *request* )

Non-blocking send operation in buffered mode.

#### 1.1.1.41 int MPI_Info_create ( MPI_Info ∗ *info* )

a new structure of type `MPI_Info` is created

Creates a new info object.

#### 1.1.1.42 int MPI_Info_delete ( MPI_Info *info,* const char ∗ *key* )

Pair (`key`, `value`) can be removed by this function.

#### 1.1.1.43 int MPI_Info_get ( MPI_Info *info,* const char ∗ *key,* int *valuelen,* char ∗ *value,* int ∗ *flag* )

Retrieves the value associated with a `key` in an info object.

searches in info for a pair with the provided `key` and writes in `value` the respective value with a max. length of `valuelen`. Value of `flag` is set to false if no matching pair was found, otherwise it is set to true

#### 1.1.1.44 int MPI_Info_set ( MPI_Info *info,* const char ∗ *key,* const char ∗ *value* )

Adds a key/value pair to info.

adds a new pair (`key`, `value`) to `info`, or overwrites an already existing pair by with the same content of key

#### 1.1.1.45 int MPI_Init ( int ∗ *argc,* char ∗∗∗ *argv* )

Initialize the MPI library.

**Parameters**

| | |
|---|---|
| *argc* | The argc argument from the main function |
| *argv* | The argv argument from the main function |

#### 1.1.1.46 int MPI_Irecv ( void ∗ *buf,* int *count,* MPI_Datatype *datatype,* int *source,* int *tag,* MPI_Comm *comm,* MPI_Request ∗ *request* )

Non-blocking receive operation.

**Parameters**

| | |
|---|---|
| *buf* | Buffer of adequate size to receive the message |
| *count* | Upper limit of the number of elements to accept |
| *datatype* | Data type of the elements to be received |
| *source* | Rank of the process from which to receive a message |
| *tag* | Message tag of the message to be received |
| *comm* | Communicator of the underlying processor group |
| *request* | Communication request |

**See also**

MPI_Recv()

#### 1.1.1.47 int MPI_Isend ( const void ∗ *buf,* int *count,* MPI_Datatype *datatype,* int *dest,* int *tag,* MPI_Comm *comm,* MPI_Request ∗ *request* )

Non-blocking send operation.

The same as MPI_Send() but non-blocking.

**Parameters**

| | |
|---|---|
| *buf* | Send buffer containing the elements to be sent successively |
| *count* | Number of elements to be sent |
| *datatype* | Data type common to all elements to be sent |
| *dest* | Rank of the target process that should receive the data |
| *tag* | Additional message tag (between 0 and 32767) to distinguish different messages of the same sender |
| *comm* | Communicator of the underlying processor group |

**See also**

MPI_Send

#### 1.1.1.48 int MPI_Op_create ( MPI_User_function ∗ *function,* int *commute,* MPI_Op ∗ *op* )

User defined accumulation operation.

The call of MPI Op create() returns a reduction operation op which can then be used as parameter of MPI Reduce().

**Parameters**

| | |
|---|---|
| *function* | The argument function specifies a user-defined function which must define the following four parameters: void ∗in, void ∗out, int ∗len, MPI Datatype ∗type. |
| *commute* | The parameter commute specifies whether the function is commutative (commute = 1) or not (commute = 0). |

#### 1.1.1.49 int MPI_Put ( const void ∗ *origin_addr,* int *origin_count,* MPI_Datatype *origin_datatype,* int *target_rank,* MPI_Aint *target_disp,* int *target_count,* MPI_Datatype *target_datatype,* MPI_Win *win* )

Copies data from the origin memory to the target.

**See also**

MPI_Win_create

#### 1.1.1.50 int MPI_Recv ( void ∗ *buf,* int *count,* MPI_Datatype *datatype,* int *source,* int *tag,* MPI_Comm *comm,* MPI_Status ∗ *status* )

Receiving Data with MPI Point-to-Point Communication.

MPI_Send() and MPI_Recv() are *blocking* and *asynchronous* operations!

**Parameters**

| | |
|---|---|
| *buf* | Buffer of adequate size to receive the message |
| *count* | Upper limit of the number of elements to accept |
| *datatype* | Data type of the elements to be received |
| *source* | Rank of the process from which to receive a message |
| *tag* | Message tag of the message to be received |
| *comm* | Communicator of the underlying processor group |
| *status* | Data structure to be filled with information on the message received |

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

Accumulation Operation.

MPI provides the following predefined reduction operations:

- arithmetical: MPI_{MAX, MIN, SUM, PROD, MINLOC, MAXLOC};
- logical: MPI_{LAND, BAND, LOR, BOR, LXOR, BXOR};
- MPI_{MAXLOC, MINLOC} additionally return the index attached by the process with the maximum or minimum value respectively

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

Scatter: The root process provides a data block (with the same size but possibly different elements) for each participating process.

**Parameters**

| | |
|---|---|
| *sendbuf* | Is the send buffer provided by the root process `root` which contains a data block with `sendcount` elements of data type `sendtype` for each process of communicator `comm`. |

#### 1.1.1.53 int MPI_Send ( const void ∗ *buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm* )

Sending Data using MPI Point-to-Point Communication.

This function is blocking.

**Parameters**

| | |
|---|---|
| *buf* | Send buffer containing the elements to be sent successively |
| *count* | Number of elements to be sent |
| *datatype* | Data type common to all elements to be sent |
| *dest* | Rank of the target process that should receive the data |
| *tag* | Additional message tag (between 0 and 32767) to distinguish different messages of the same sender |
| *comm* | Communicator of the underlying processor group |

#### 1.1.1.54 int MPI_Sendrecv ( const 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* )

Data Exchange with MPI Sendrecv()

Advantage of MPI Sendrecv(): The runtime system guarantees deadlock freedom.

Prerequisit: sendbuf and recvbuf must be disjoint, non-overlapping memory locations.

Messages of different lengths and different data types may be exchanged.

If send and receive buffers are identical, the MPI operation MPI_Sendrecv_replace() may be used.

**Parameters**

| | |
|---|---|
| *sendbuf* | Send buffer in which the data elements to be sent are stored |
| *sendcount* | Number of data elements to be sent |
| *sendtype* | Data type of the elements to be sent |
| *dest* | Rank of the target process to which the data elements are sent |

| | |
|---|---|
| *sendtag* | Tag for the message to be send |
| *recvbuf* | Receive buffer for the message to be received |
| *recvcount* | Maximum number of data elements to be received |
| *recvtype* | Data type of the data elements to be received |
| *source* | Rank of the process from which the message is expected |
| *recvtag* | Tag of the message to be received |
| *comm* | Communicator used for the communication |
| *status* | Data structure to store information on the message received |

### 1.1.1.55 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* )

Like MPI_Sendrecv.

**Parameters**

| | |
|---|---|
| *buf* | Buffer that is used as both send and receive buffer |

**See also**

MPI_Sendrecv

### 1.1.1.56 int MPI_Test ( MPI_Request ∗ *request,* int ∗ *flag,* MPI_Status ∗ *status* )

Querying the status of a non-blocking communication operation.

If MPI Test() is called for a receive operation that is completed the parameter status contains

**Parameters**

| | |
|---|---|
| *request* | The request |
| *flag* | 1 if the send or receive communication operation specified by request has been completed, 0 denotes that the operation is still in progress. |
| *status* | |

### 1.1.1.57 int MPI_Wait ( MPI_Request ∗ *request,* MPI_Status ∗ *status* )

Waiting for the completion of a communication operation.

This MPI operation blocks the calling process until the send or receive operation specified by request is completed.

**Parameters**

| | |
|---|---|
| *request* | |
| *status* | |

### 1.1.1.58 int MPI_Win_complete ( MPI_Win *win* )

Completes an RMA access epoch on win started by a call to MPI_Win_start.

**See also**

MPI_Win_create

#### 1.1.1.59 int MPI_Win_create ( void ∗ *base,* MPI_Aint *size,* int *disp_unit,* MPI_Info *info,* MPI_Comm *comm,* MPI_Win ∗ *win* )

One-sided MPI call that returns a window object for RMA operations.

each process from the communicator comm has to execute that operation

#### 1.1.1.60 int MPI_Win_fence ( int *assert,* MPI_Win *win* )

Global synchronization of a process group of a window.

Suitable for regular applications with alternating

- global computation phases and
- global communication phases

**See also**

MPI_Win_create

#### 1.1.1.61 int MPI_Win_free ( MPI_Win ∗ *win* )

Frees the window object and returns a null handle.

All operations of a participating processes have to be finished

#### 1.1.1.62 int MPI_Win_lock ( int *lock_type,* int *rank,* int *assert,* MPI_Win *win* )

Setting a lock before accessing.

#### 1.1.1.63 int MPI_Win_post ( MPI_Group *group,* int *assert,* MPI_Win *win* )

Starts an RMA exposure epoch for the local window associated with win.

**See also**

MPI_Win_create

#### 1.1.1.64 int MPI_Win_start ( MPI_Group *group,* int *assert,* MPI_Win *win* )

Starts an RMA access epoch for win.

**See also**

MPI_Win_create

#### 1.1.1.65 int MPI_Win_test ( MPI_Win *win,* int ∗ *flag* )

Attempts to complete an RMA exposure epoch; a nonblocking version of MPI_Win_wait.

**See also**

MPI_Win_create

#### 1.1.1.66 int MPI_Win_unlock ( int *rank,* MPI_Win *win* )

Releasing a lock after access.

**See also**

MPI_Win_create

#### 1.1.1.67 int MPI_Win_wait ( MPI_Win *win* )

Completes an RMA exposure epoch started by a call to MPI_Win_post on win.

**See also**

MPI_Win_create

#### 1.1.1.68 double MPI_Wtime ( void )

Returns an elapsed time on the calling processor.

start = MPI Wtime(); part to measure(); end = MPI Wtime(); time = end - start;

# Index