1 MPI Reference Card

`int MPI_Abort( comm, errorcode )`
`MPI_Comm comm;`
`int errorcode;`
	Terminates MPI execution environment

`int MPI_Address( location, address)`
`void *location;`
`MPI_Aint *address;`
	Gets the address of a location in memory

`int MPI_Allgatherv ( sendbuf, sendcount,`
`sendtype, recvbuf, recvcounts,`
`displs, recvtype, comm`
`)`
`void *sendbuf;`
`int sendcount;`
`MPI_Datatype sendtype;`
`void *recvbuf;`
`int *recvcounts;`
`int *displs;`
`MPI_Datatype recvtype;`
`MPI_Comm comm;`
	Gathers data from all tasks and deliver it to all

`int MPI_Allgather ( sendbuf, sendcount,`
`sendtype, recvbuf, recvcount,`
`recvtype, comm`
`)`
`void *sendbuf;`
`int sendcount;`
`MPI_Datatype sendtype;`
`void *recvbuf;`
`int recvcount;`
`MPI_Datatype recvtype;`
`MPI_Comm comm;`
	Gathers data from all tasks and distribute it to all

`int MPI_Allreduce ( sendbuf, recvbuf,`
`count, datatype, op, comm )`
`void *sendbuf;`
`void *recvbuf;`
`int count;`
`MPI_Datatype datatype;`
`MPI_Op op;`
`MPI_Comm comm;`
	Combines values from all processes and distribute the result back to all processes

`int MPI_Alltoallv ( sendbuf, sendcnts,`
`sdispls, sendtype, recvbuf,`
`recvcnts, rdispls, recvtype,`
`comm`
`)`
`void *sendbuf;`
`int *sendcnts;`
`int *sdispls;`
`MPI_Datatype sendtype;`
`void *recvbuf;`
`int *recvcnts;`
`int *rdispls;`
`MPI_Datatype recvtype;`
`MPI_Comm comm;`
	Sends data from all to all processes, with a displacement

int MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, comm) void *sendbuf; int sendcount; MPI_Datatype sendtype; void *recvbuf; int recvcnt; MPI_Datatype recvtype; MPI_Comm comm;	Sends data from all to all processes
int MPI_Attr_delete (comm, keyval) MPI_Comm comm; int keyval;	Deletes attribute value associated with a key
int MPI_Attr_get (comm, keyval, attr_value, flag) MPI_Comm comm; int keyval; void **attr_value; int *flag;	Retrieves attribute value by key
int MPI_Attr_put (comm, keyval, attr_value) MPI_Comm comm; int keyval; void *attr_value;	Stores attribute value associated with a key
int MPI_Barrier (comm) MPI_Comm comm;	Blocks until all process have reached this routine.
int MPI_Bcast (buffer, count, datatype, root, comm) void *buffer; int count; MPI_Datatype datatype; int root; MPI_Comm comm;	Broadcasts a message from the process with rank "root" to all other processes of the group.
int MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int dest; int tag; MPI_Comm comm; MPI_Request *request;	Builds a handle for a buffered send
int MPI_Bsend(buf, count, datatype, dest, tag, comm) void *buf; int count, dest, tag; MPI_Datatype datatype; MPI_Comm comm;	Basic send with user-specified buffering
int MPI_Buffer_attach(buffer, size) void *buffer; int size;	Attaches a user-defined buffer for sending

`int MPI_Buffer_detach( buffer, size )`
`void **buffer;`
`int *size;`
	Removes an existing buffer (for use in MPI_Bsend etc)

`int MPI_Cancel( request )` `MPI_Request *request;`	Cancels a communication request

`int MPI_Cart_coords ( comm, rank,`
`maxdims, coords )`
`MPI_Comm comm;`
`int rank;`
`int maxdims;`
`int *coords;`
	Determines process coords in cartesian topology given rank in group

`int MPI_Cart_create ( comm_old, ndims,`
`dims, periods, reorder,`
`comm_cart )`
`MPI_Comm comm_old;`
`int ndims;`
`int *dims;`
`int *periods;`
`int reorder;`
`MPI_Comm *comm_cart;`
	Makes a new communicator to which topology information has been attached

`int MPI_Cart_get ( comm, maxdims, dims,`
`periods, coords )`
`MPI_Comm comm;`
`int maxdims;`
`int *dims, *periods, *coords;`
	Retrieves Cartesian topology information associated with a communicator

`int MPI_Cart_map ( comm_old, ndims, dims,`
`periods, newrank )`
`MPI_Comm comm_old;`
`int ndims;`
`int *dims;`
`int *periods;`
`int *newrank;`
	Maps process to Cartesian topology information

`int MPI_Cart_rank ( comm, coords, rank )`
`MPI_Comm comm;`
`int *coords;`
`int *rank;`
	Determines process rank in communicator given Cartesian location

`int MPI_Cart_shift ( comm, direction,`
`displ, source, dest )`
`MPI_Comm comm;`
`int direction;`
`int displ;`
`int *source;`
`int *dest;`
	Returns the shifted source and destination ranks, given a shift direction and amount

`int MPI_Cart_sub ( comm, remain_dims,`
`comm_new )`
`MPI_Comm comm;`
`int *remain_dims;`
`MPI_Comm *comm_new;`
	Partitions a communicator into subgroups which form lower-dimensional cartesian subgrids

`int MPI_Cartdim_get ( comm, ndims )`
`MPI_Comm comm;`
`int *ndims;`
	Retrieves Cartesian topology information associated with a communicator

`int MPI_Comm_compare ( comm1, comm2, result )`
`MPI_Comm comm1;`
`MPI_Comm comm2;`
`int *result;`
	Compares two communicators

`int MPI_Comm_create ( comm, group, comm_out )`
`MPI_Comm comm;`
`MPI_Group group;`
`MPI_Comm *comm_out;`
	Creates a new communicator

`int MPI_Comm_dup ( comm, comm_out )` `MPI_Comm comm, *comm_out;`	Duplicates an existing communicator with all its cached information
`int MPI_Comm_free ( comm )` `MPI_Comm *comm;`	Marks the communicator object for deallocation

`int MPI_Comm_group ( comm, group )`
`MPI_Comm comm;`
`MPI_Group *group;`
	Accesses the group associated with given communicator

`int MPI_Comm_rank ( comm, rank )`
`MPI_Comm comm;`
`int *rank;`
	Determines the rank of the calling process in the communicator

`int MPI_Comm_remote_group ( comm, group )`
`MPI_Comm comm;`
`MPI_Group *group;`
	Accesses the remote group associated with the given inter-communicator

`int MPI_Comm_remote_size ( comm, size )`
`MPI_Comm comm;`
`int *size;`
	Determines the size of the remote group associated with an inter-communictor

`int MPI_Comm_size ( comm, size )`
`MPI_Comm comm;`
`int *size;`
	Determines the size of the group associated with a communictor

`int MPI_Comm_split ( comm, color, key, comm_out )`
`MPI_Comm comm;`
`int color, key;`
`MPI_Comm *comm_out;`
	Creates new communicators based on colors and keys

`int MPI_Comm_test_inter ( comm, flag )`
`MPI_Comm comm;`
`int *flag;`
	Tests to see if a comm is an inter-communicator

`int MPIR_dup_fn ( comm, keyval, extra_state, attr_in, attr_out, flag )`
`MPI_Comm *comm;`
`int *keyval;`
`void *extra_state;`
`void *attr_in;`
`void **attr_out;`
`int *flag;`
	A function to simple-mindedly copy attributes

`int MPI_Dims_create(nnodes, ndims, dims)`
`int nnodes;`
`int ndims;`
`int *dims;`
	Creates a division of processors in a cartesian grid

`int MPI_Errhandler_create( function, errhandler )`
`MPI_Handler_function *function;`
`MPI_Errhandler *errhandler;`
	Creates an MPI-style errorhandler

`int MPI_Errhandler_free( errhandler )` `MPI_Errhandler *errhandler;`	Frees an MPI-style errorhandler

`int MPI_Errhandler_get( comm, errhandler )`
`MPI_Comm comm;`
`MPI_Errhandler *errhandler;`
	Gets the error handler for a communicator

`int MPI_Errhandler_set( comm, errhandler )`
`MPI_Comm comm;`
`MPI_Errhandler errhandler;`
	Sets the error handler for a communicator

`int MPI_Error_class( errorcode, errorclass )` `int errorcode, *errorclass;`	Converts an error code into an error class

`int MPI_Error_string( errorcode, string, resultlen )`
`int errorcode, *resultlen;`
`char *string;`
	Return a string for a given error code

`int MPI_Finalize()`	Terminates MPI execution environment

`int MPI_Gatherv ( sendbuf, sendcnt, sendtype, recvbuf, recvcnts, displs, recvtype, root, comm )`
`void *sendbuf;`
`int sendcnt;`
`MPI_Datatype sendtype;`
`void *recvbuf;`
`int *recvcnts;`
`int *displs;`
`MPI_Datatype recvtype;`
`int root;`
`MPI_Comm comm;`
	Gathers into specified locations from all processes in a group

`int MPI_Gather ( sendbuf, sendcnt, sendtype, recvbuf, recvcount, recvtype, root, comm )`
`void *sendbuf;`
`int sendcnt;`
`MPI_Datatype sendtype;`
`void *recvbuf;`
`int recvcount;`
`MPI_Datatype recvtype;`
`int root;`
`MPI_Comm comm;`
	Gathers together values from a group of processes

`int MPI_Get_count( status, datatype, count )`
`MPI_Status *status;`
`MPI_Datatype datatype;`
`int *count;`
	Gets the number of "top level" elements

`int MPI_Get_elements ( status, datatype, elements )`
`MPI_Status *status;`
`MPI_Datatype datatype;`
`int *elements;`
	Returns the number of basic elements in a datatype

`int MPI_Get_processor_name( name, resultlen )`
`char *name;`
`int *resultlen;`
	Gets the name of the processor

`int MPI_Graph_create ( comm_old, nnodes, index, edges, reorder, comm_graph )`
`MPI_Comm comm_old;`
`int nnodes;`
`int *index;`
`int *edges;`
`int reorder;`
`MPI_Comm *comm_graph;`
	Makes a new communicator to which topology information has been attached

`int MPI_Graph_get ( comm, maxindex, maxedges, index, edges )`
`MPI_Comm comm;`
`int maxindex, maxedges;`
`int *index, *edges;`
	Retrieves graph topology information associated with a communicator

`int MPI_Graph_map ( comm_old, nnodes, index, edges, newrank )`
`MPI_Comm comm_old;`
`int nnodes;`
`int *index;`
`int *edges;`
`int *newrank;`
	Maps process to graph topology information

`int MPI_Graph_neighbors_count ( comm, rank, nneighbors )`
`MPI_Comm comm;`
`int rank;`
`int *nneighbors;`
	Returns the number of neighbors of a node associated with a graph topology

`int MPI_Graph_neighbors ( comm, rank, maxneighbors, neighbors )`
`MPI_Comm comm;`
`int rank;`
`int maxneighbors;`
`int *neighbors;`
	Returns the neighbors of a node associated with a graph topology

`int MPI_Graphdims_get ( comm, nnodes, nedges )`
`MPI_Comm comm;`
`int *nnodes;`
`int *nedges;`
	Retrieves graph topology information associated with a communicator

`int MPI_Group_compare ( group1, group2, result )`
`MPI_Group group1;`
`MPI_Group group2;`
`int *result;`
	Compares two groups

`int MPI_Group_difference ( group1, group2, group_out )` `MPI_Group group1, group2, *group_out;`	Makes a group from the difference of two groups

`int MPI_Group_excl ( group, n, ranks, newgroup )`
`MPI_Group group, *newgroup;`
`int n, *ranks;`
	Produces a group by reordering an existing group and taking only unlisted members

`int MPI_Group_free ( group )` `MPI_Group *group;`	Frees a group

`int MPI_Group_incl ( group, n, ranks, group_out )`
`MPI_Group group, *group_out;`
`int n, *ranks;`
	Produces a group by reordering an existing group and taking only listed members

`int MPI_Group_intersection ( group1, group2, group_out )` `MPI_Group group1, group2, *group_out;`	Produces a group as the intersection of two existing groups

`int MPI_Group_range_excl ( group, n, ranges, newgroup )`
`MPI_Group group, *newgroup;`
`int n, ranges[][3];`
	Produces a group by excluding ranges of processes from an existing group

`int MPI_Group_range_incl ( group, n, ranges, newgroup )`
`MPI_Group group, *newgroup;`
`int n, ranges[][3];`
	Creates a new group from ranges of ranks in an existing group

`int MPI_Group_rank ( group, rank )`
`MPI_Group group;`
`int *rank;`
	Returns the rank of this process in the given group

`int MPI_Group_size ( group, size )`
`MPI_Group group;`
`int *size;`
	Returns the size of a group

`int MPI_Group_translate_ranks ( group_a, n, ranks_a, group_b, ranks_b )`
`MPI_Group group_a;`
`int n;`
`int *ranks_a;`
`MPI_Group group_b;`
`int *ranks_b;`
	Translates the ranks of processes in one group to those in another group

`int MPI_Group_union ( group1, group2, group_out )` `MPI_Group group1, group2, *group_out;`	Produces a group by combining two groups

`int MPI_Ibsend( buf, count, datatype, dest, tag, comm, request )`
`void *buf;`
`int count;`
`MPI_Datatype datatype;`
`int dest;`
`int tag;`
`MPI_Comm comm;`
`MPI_Request *request;`
	Starts a nonblocking buffered send

`int MPI_Initialized( flag )` `int *flag;`	Indicates whether MPI_Init has been called.

`int MPI_Init(argc,argv)`
`int *argc;`
`char ***argv;`
	Initialize the MPI execution environment

`int MPI_Intercomm_create ( local_comm,`
`  local_leader, peer_comm,`
`  remote_leader, tag, comm_out`
`)`
`MPI_Comm local_comm;`
`int local_leader;`
`MPI_Comm peer_comm;`
`int remote_leader;`
`int tag;`
`MPI_Comm *comm_out;`
	Creates an intercommuncator from two intracommunicators

`int MPI_Intercomm_merge ( comm, high,`
`  comm_out )`
`MPI_Comm comm;`
`int high;`
`MPI_Comm *comm_out;`
	Creates an intracommuncator from an intercommunicator

`int MPI_Iprobe( source, tag, comm, flag,`
`  status )`
`int source;`
`int tag;`
`int *flag;`
`MPI_Comm comm;`
`MPI_Status *status;`
	Nonblocking test for a message

`int MPI_Irecv( buf, count, datatype,`
`  source, tag, comm, request )`
`void *buf;`
`int count;`
`MPI_Datatype datatype;`
`int source;`
`int tag;`
`MPI_Comm comm;`
`MPI_Request *request;`
	Begins a nonblocking receive

`int MPI_Irsend( buf, count, datatype,`
`  dest, tag, comm, request )`
`void *buf;`
`int count;`
`MPI_Datatype datatype;`
`int dest;`
`int tag;`
`MPI_Comm comm;`
`MPI_Request *request;`
	Starts a nonblocking ready send

`int MPI_Isend( buf, count, datatype,`
`  dest, tag, comm, request )`
`void *buf;`
`int count;`
`MPI_Datatype datatype;`
`int dest;`
`int tag;`
`MPI_Comm comm;`
`MPI_Request *request;`
	Begins a nonblocking send

`int MPI_Issend( buf, count, datatype, dest, tag, comm, request )`
`void *buf;`
`int count;`
`MPI_Datatype datatype;`
`int dest;`
`int tag;`
`MPI_Comm comm;`
`MPI_Request *request;`
	Starts a nonblocking synchronous send

`int MPI_Keyval_create ( copy_fn, delete_fn, keyval, extra_state )`
`MPI_Copy_function *copy_fn;`
`MPI_Delete_function *delete_fn;`
`int *keyval;`
`void *extra_state;`
	Generates a new attribute key

`int MPI_Keyval_free ( keyval )` `int *keyval;`	Frees attribute key for communicator cache attribute

`int MPI_Op_create( function, commute, op )`
`MPI_User_function *function;`
`int commute;`
`MPI_Op *op;`
	Creates a user-defined combination function handle

`int MPI_Op_free( op )` `MPI_Op *op;`	Frees a user-defined combination function handle

`int MPI_Pack_size ( incount, datatype, comm, size )`
`int incount;`
`MPI_Datatype datatype;`
`MPI_Comm comm;`
`int *size;`
	Returns the upper bound on the amount of space needed to pack a message

`int MPI_Pack ( inbuf, incount, type, outbuf, outcount, position, comm )`
`void *inbuf;`
`int incount;`
`MPI_Datatype type;`
`void *outbuf;`
`int outcount;`
`int *position;`
`MPI_Comm comm;`
	Packs a datatype into contiguous memory

`int MPI_Pcontrol( level )` `int level;`	Controls profiling

`int MPI_Probe( source, tag, comm, status )`
`int source;`
`int tag;`
`MPI_Comm comm;`
`MPI_Status *status;`
	Blocking test for a message

`int MPI_Recv_init( buf, count, datatype,`
`  source, tag, comm, request )`
`void *buf;`
`int count;`
`MPI_Request *request;`
`MPI_Datatype datatype;`
`int source;`
`int tag;`
`MPI_Comm comm;`
	Builds a handle for a receive

`int MPI_Recv( buf, count, datatype,`
`  source, tag, comm, status )`
`void *buf;`
`int count, source, tag;`
`MPI_Datatype datatype;`
`MPI_Comm comm;`
`MPI_Status *status;`
	Basic receive

`int MPI_Reduce_scatter ( sendbuf,`
`  recvbuf, recvcnts, datatype,`
`  op, comm )`
`void *sendbuf;`
`void *recvbuf;`
`int *recvcnts;`
`MPI_Datatype datatype;`
`MPI_Op op;`
`MPI_Comm comm;`
	Combines values and scatters the results

`int MPI_Reduce ( sendbuf, recvbuf, count,`
`  datatype, op, root, comm )`
`void *sendbuf;`
`void *recvbuf;`
`int count;`
`MPI_Datatype datatype;`
`MPI_Op op;`
`int root;`
`MPI_Comm comm;`
	Reduces values on all processes to a single value

`int MPI_Request_free( request )` `MPI_Request *request;`	Frees a communication request object

`int MPI_Rsend_init( buf, count, datatype,`
`  dest, tag, comm, request )`
`void *buf;`
`int count;`
`MPI_Datatype datatype;`
`int dest;`
`int tag;`
`MPI_Comm comm;`
`MPI_Request *request;`
	Builds a handle for a ready send

`int MPI_Rsend( buf, count, datatype,`
`  dest, tag, comm )`
`void *buf;`
`int count, dest, tag;`
`MPI_Datatype datatype;`
`MPI_Comm comm;`
	Basic ready send

`int MPI_Scan ( sendbuf, recvbuf, count, datatype, op, comm )`
`void *sendbuf;`
`void *recvbuf;`
`int count;`
`MPI_Datatype datatype;`
`MPI_Op op;`
`MPI_Comm comm;`
	Computes the scan (partial reductions) of data on a collection of processes

`int MPI_Scatterv ( sendbuf, sendcnts, displs, sendtype, recvbuf, recvcnt, recvtype, root, comm )`
`void *sendbuf;`
`int *sendcnts;`
`int *displs;`
`MPI_Datatype sendtype;`
`void *recvbuf;`
`int recvcnt;`
`MPI_Datatype recvtype;`
`int root;`
`MPI_Comm comm;`
	Scatters a buffer in parts to all tasks in a group

`int MPI_Scatter ( sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm )`
`void *sendbuf;`
`int sendcnt;`
`MPI_Datatype sendtype;`
`void *recvbuf;`
`int recvcnt;`
`MPI_Datatype recvtype;`
`int root;`
`MPI_Comm comm;`
	Sends data from one task to all other tasks in a group

`int MPI_Send_init( buf, count, datatype, dest, tag, comm, request )`
`void *buf;`
`int count;`
`MPI_Datatype datatype;`
`int dest;`
`int tag;`
`MPI_Comm comm;`
`MPI_Request *request;`
	Builds a handle for a standard send

`int MPI_Sendrecv_replace( buf, count, datatype, dest, sendtag, source, recvtag, comm, status )`
`void *buf;`
`int count, dest, sendtag, source, recvtag;`
`MPI_Datatype datatype;`
`MPI_Comm comm;`
`MPI_Status *status;`
	Sends and receives using a single buffer

`int MPI_Sendrecv( sendbuf, sendcount,`
`  sendtype, dest, sendtag,`
`  recvbuf, recvcount, recvtype,`
`  source, recvtag, comm, status`
`)`
`void *sendbuf;`
`int sendcount;`
`MPI_Datatype sendtype;`
`int dest, sendtag;`
`void *recvbuf;`
`int recvcount;`
`MPI_Datatype recvtype;`
`int source, recvtag;`
`MPI_Comm comm;`
`MPI_Status *status;`
	Sends and receives a message

`int MPI_Send( buf, count, datatype, dest,`
`  tag, comm )`
`void *buf;`
`int count, dest, tag;`
`MPI_Datatype datatype;`
`MPI_Comm comm;`
	Basic send

`int MPI_Ssend_init( buf, count, datatype,`
`  dest, tag, comm, request )`
`void *buf;`
`int count;`
`MPI_Datatype datatype;`
`int dest;`
`int tag;`
`MPI_Comm comm;`
`MPI_Request *request;`
	Builds a handle for a synchronous send

`int MPI_Ssend( buf, count, datatype,`
`  dest, tag, comm )`
`void *buf;`
`int count, dest, tag;`
`MPI_Datatype datatype;`
`MPI_Comm comm;`
	Basic synchronous send

`int MPI_Startall( count,`
`  array_of_requests )`
`int count;`
`MPI_Request array_of_requests[];`
	Starts a collection of requests

`int MPI_Start( request )` `MPI_Request *request;`	Initiates a communication with a persistent request handle

`int MPI_Test_cancelled( status, flag )`
`MPI_Status *status;`
`int *flag;`
	Tests to see if a request was canceled

`int MPI_Testall( count,`
`  array_of_requests, flag,`
`  array_of_statuses )`
`int count;`
`MPI_Request array_of_requests[];`
`int *flag;`
`MPI_Status *array_of_statuses;`
	Tests for the completion of all previously initiated communications

`int MPI_Testany( count,`
`  array_of_requests, index, flag,`
`  status )`
`int count;`
`MPI_Request array_of_requests[];`
`int *index, *flag;`
`MPI_Status *status;`
	Tests for completion of any previdously initiated communication

`int MPI_Testsome( incount,`
`  array_of_requests, outcount,`
`  array_of_indices,`
`  array_of_statuses`
`)`
`int incount, *outcount,`
`array_of_indices[];`
`MPI_Request array_of_requests[];`
`MPI_Status array_of_statuses[];`
	Tests for some given communications to complete

`int MPI_Test ( request, flag, status )`
`MPI_Request *request;`
`int *flag;`
`MPI_Status *status;`
	Tests for the completion of a send or receive

`int MPI_Topo_test ( comm, top_type )`
`MPI_Comm comm;`
`int *top_type;`
	Determines the type of topology (if any) associated with a communicator

`int MPI_Type_commit ( datatype )` `MPI_Datatype *datatype;`	Commits the datatype

`int MPI_Type_contiguous( count, old_type,`
`  newtype )`
`int count;`
`MPI_Datatype old_type;`
`MPI_Datatype *newtype;`
	Creates a contiguous datatype

`int MPI_Type_count ( datatype, count )`
`MPI_Datatype datatype;`
`int *count;`
	Returns the number of "top-level" entries in the datatype

`int MPI_Type_extent( datatype, extent )`
`MPI_Datatype datatype;`
`MPI_Aint *extent;`
	Returns the extent of a datatype

`int MPI_Type_free ( datatype )` `MPI_Datatype *datatype;`	Frees the datatype

`int MPI_Type_hindexed( count, blocklens,`
`  indices, old_type, newtype )`
`int count;`
`int blocklens[];`
`MPI_Aint indices[];`
`MPI_Datatype old_type;`
`MPI_Datatype *newtype;`
	Creates an indexed datatype with offsets in bytes

`int MPI_Type_hvector( count, blocklen, stride, old_type, newtype )`
`int count;`
`int blocklen;`
`MPI_Aint stride;`
`MPI_Datatype old_type;`
`MPI_Datatype *newtype;`
	Creates a vector (strided) datatype with offset in bytes

`int MPI_Type_indexed( count, blocklens, indices, old_type, newtype )`
`int count;`
`int blocklens[];`
`int indices[];`
`MPI_Datatype old_type;`
`MPI_Datatype *newtype;`
	Creates an indexed datatype

`int MPI_Type_lb ( datatype, displacement )`
`MPI_Datatype datatype;`
`MPI_Aint *displacement;`
	Returns the lower-bound of a datatype

`int MPI_Type_size ( datatype, size )`
`MPI_Datatype datatype;`
`MPI_Aint *size;`
	Return the number of bytes occupied by entries in the datatype

`int MPI_Type_struct( count, blocklens, indices, old_types, newtype )`
`int count;`
`int blocklens[];`
`MPI_Aint indices[];`
`MPI_Datatype old_types[];`
`MPI_Datatype *newtype;`
	Creates a struct datatype

`int MPI_Type_ub ( datatype, displacement )`
`MPI_Datatype datatype;`
`MPI_Aint *displacement;`
	Returns the upper bound of a datatype

`int MPI_Type_vector( count, blocklen, stride, old_type, newtype )`
`int count;`
`int blocklen;`
`int stride;`
`MPI_Datatype old_type;`
`MPI_Datatype *newtype;`
	Creates a vector (strided) datatype

`int MPI_Unpack ( inbuf, insize, position, outbuf, outcount, type, comm )`
`void *inbuf;`
`int insize;`
`int *position;`
`void *outbuf;`
`int outcount;`
`MPI_Datatype type;`
`MPI_Comm comm;`
	Unpack a datatype into contiguous memory

`int MPI_Waitall(count, array_of_requests, array_of_statuses )`
`int count;`
`MPI_Request array_of_requests[];`
`MPI_Status array_of_statuses[];`
	Waits for all given communications to complete

`int MPI_Waitany(count, array_of_requests, index, status )`
`int count;`
`MPI_Request array_of_requests[];`
`int *index;`
`MPI_Status *status;`
	Waits for any specified send or receive to complete

`int MPI_Waitsome( incount, array_of_requests, outcount, array_of_indices, array_of_statuses )`
`int incount, *outcount, array_of_indices[];`
`MPI_Request array_of_requests[];`
`MPI_Status array_of_statuses[];`
	Waits for some given communications to complete

`int MPI_Wait ( request, status )`
`MPI_Request *request;`
`MPI_Status *status;`
	Waits for an MPI send or receive to complete

`double MPI_Wtick()`	Returns the resolution of MPI_Wtime
`double MPI_Wtime()`	Returns an elapsed time on the calling processor