MPI Programming Primer

Basic Concepts

Through Message Passing Interface (MPI) an application views its parallel environment as a static group of processes. An MPI process is born into the world with zero or more siblings. This initial collection of processes is called the world group. A unique number, called a rank, is assigned to each member process from the sequence 0 through N-1, where N is the total number of processes in the world group. A member can query its own rank and the size of the world group. Processes may all be running the same program (SPMD) or different programs (MIMD). The world group processes may subdivide, creating additional subgroups with a potentially different rank in each group.

A process sends a message to a destination rank in the desired group. A process may or may not specify a source rank when receiving a message. Messages are further filtered by an arbitrary, user specified, synchronization integer called a tag, which the receiver may also ignore.

An important feature of MPI is the ability to guarantee independent software developers that their choice of tag in a particular library will not conflict with the choice of tag by some other independent developer or by the end user of the library. A further synchronization integer called a context is allocated by MPI and is automatically attached to every message. Thus, the four main synchronization variables in MPI are the source and destination ranks, the tag and the context.

A communicator is an opaque MPI data structure that contains information on one group and that contains one context. A communicator is an argument to all MPI communication routines. After a process is created and initializes MPI, three predefined communicators are available.

MPI_COMM_WORLD	the world group
MPI_COMM_SELF	group with one member, myself
MPI_COMM_PARENT	an intercommunicator between two groups: my world group and my parent group (See *Dynamic Processes*.)

Many applications require no other communicators beyond the world communicator. If new subgroups or new contexts are needed, additional communicators must be created.

MPI constants, templates and prototypes are in the MPI header file, mpi.h.

```
#include <mpi.h>
```

`MPI_Init`	Initialize MPI state.
`MPI_Finalize`	Clean up MPI state.
`MPI_Abort`	Abnormally terminate.
`MPI_Comm_size`	Get group process count.
`MPI_Comm_rank`	Get my rank within process group.
`MPI_Initialized`	Has MPI been initialized?

Initialization

The first MPI routine called by a program must be MPI_Init(). The command line arguments are passed to MPI_Init().

```
MPI_Init(int *argc, char **argv[]);
```

A process ceases MPI operations with MPI_Finalize().

```
MPI_Finalize(void);
```

In response to an error condition, a process can terminate itself and all members of a communicator with MPI_Abort(). The implementation may report the error code argument to the user in a manner consistent with the underlying operation system.

```
MPI_Abort (MPI_Comm comm, int errcode);
```

Basic Parallel Information

Two numbers that are very useful to most parallel applications are the total number of parallel processes and self process identification. This information is learned from the MPI_COMM_WORLD communicator using the routines MPI_Comm_size() and MPI_Comm_rank().

```
MPI_Comm_size (MPI_Comm comm, int *size);
MPI_Comm_rank (MPI_Comm comm, int *rank);
```

Of course, any communicator may be used, but the world information is usually key to decomposing data across the entire parallel application.

`MPI_Send`	Send a message in standard mode.
`MPI_Recv`	Receive a message.
`MPI_Get_count`	Count the elements received.
`MPI_Probe`	Wait for message arrival.

`MPI_Bsend`	Send a message in buffered mode.
`MPI_Ssend`	Send a message in synchronous mode.
`MPI_Rsend`	Send a message in ready mode.
`MPI_Buffer_attach`	Attach a buffer for buffered sends.
`MPI_Buffer_detach`	Detach the current buffer.
`MPI_Sendrecv`	Send in standard mode, then receive.
`MPI_Sendrecv_replace`	Send and receive from/to one area.
`MPI_Get_elements`	Count the basic elements received.

Blocking Point-to-Point

This section focuses on blocking, point-to-point, message-passing routines. The term “blocking” in MPI means that the routine does not return until the associated data buffer may be reused. A point-to-point message is sent by one process and received by one process.

Send Modes

The issues of flow control and buffering present different choices in designing message-passing primitives. MPI does not impose a single choice but instead offers four transmission modes that cover the synchronization, data transfer and performance needs of most applications. The mode is selected by the sender through four different send routines, all with identical argument lists. There is only one receive routine. The four send modes are:

standard	The send completes when the system can buffer the message (it is not obligated to do so) or when the message is received.
buffered	The send completes when the message is buffered in application supplied space, or when the message is received.
synchronous	The send completes when the message is received.
ready	The send must not be started unless a matching receive has been started. The send completes immediately.

Standard Send

Standard mode serves the needs of most applications. A standard mode message is sent with MPI_Send().

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

An MPI message is not merely a raw byte array. It is a count of typed elements. The element type may be a simple raw byte or a complex data structure. See *Message Datatypes*.

The four MPI synchronization variables are indicated by the MPI_Send() parameters. The source rank is the caller's. The destination rank and message tag are explicitly given. The context is a property of the communicator.

As a blocking routine, the buffer can be overwritten when MPI_Send() returns. Although most systems will buffer some number of messages, especially short messages, without any receiver, a programmer cannot rely upon MPI_Send() to buffer even one message. Expect that the routine will not return until there is a matching receiver.

Receive

A message in any mode is received with MPI_Recv().

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

Again the four synchronization variables are indicated, with source and destination swapping places. The source rank and the tag can be ignored with the special values MPI_ANY_SOURCE and MPI_ANY_TAG. If both these wildcards are used, the next message for the given communicator is received.

Status Object

An argument not present in MPI_Send() is the status object pointer. The status object is filled with useful information when MPI_Recv() returns. If the source and/or tag wildcards were used, the actual received source rank and/or message tag are accessible directly from the status object.

`status.MPI_SOURCE`	the sender's rank
`status.MPI_TAG`	the tag given by the sender

Message Lengths

It is erroneous for an MPI program to receive a message longer than the specified receive buffer. The message might be truncated or an error condition might be raised or both. It is completely acceptable to receive a message shorter than the specified receive buffer. If a short message may arrive, the application can query the actual length of the message with MPI_Get_count().

```
MPI_Get_count (MPI_Status *status,
     MPI_Datatype dtype, int *count);
```

The status object and MPI datatype are those provided to MPI_Recv(). The count returned is the number of elements received of the given datatype. See *Message Datatypes*.

Probe Sometimes it is impractical to pre-allocate a receive buffer. MPI_Probe() synchronizes a message and returns information about it without actually receiving it. Only synchronization variables and the status object are provided as arguments. MPI_Probe() does not return until a message is synchronized.

```
MPI_Probe (in source, int tag, MPI_Comm comm,
     MPI_Status *status);
```

After a suitable message buffer has been prepared, the same message reported by MPI_Probe() can be received with MPI_Recv().

`MPI_Isend`	Begin to send a standard message.
`MPI_Irecv`	Begin to receive a message.
`MPI_Wait`	Complete a pending request.
`MPI_Test`	Check or complete a pending request.
`MPI_Iprobe`	Check message arrival.
`MPI_Ibsend`	Begin to send a buffered message.
`MPI_Issend`	Begin to send a synchronous message.
`MPI_Irsend`	Begin to send a ready message.
`MPI_Request_free`	Free a pending request.
`MPI_Waitany`	Complete any one request.
`MPI_Testany`	Check or complete any one request.
`MPI_Waitall`	Complete all requests.
`MPI_Testall`	Check or complete all requests.
`MPI_Waitsome`	Complete one or more requests.
`MPI_Testsome`	Check or complete one or more requests.
`MPI_Cancel`	Cancel a pending request.
`MPI_Test_cancelled`	Check if a pending request was cancelled.

Nonblocking Point-to-Point

The term “nonblocking” in MPI means that the routine returns immediately and may only have started the message transfer operation, not necessarily completed it. The application may not safely reuse the message buffer after a nonblocking routine returns. The four blocking send routines and one blocking receive routine all have nonblocking counterparts. The nonblocking routines have an extra output argument - a request object. The request is later passed to one of a suite of completion routines. Once an operation has completed, its message buffer can be reused.

The intent of nonblocking message-passing is to start a message transfer at the earliest possible moment, continue immediately with important computation, and then insist upon completion at the latest possible moment. When the earliest and latest moment are the same, nonblocking routines are not useful. Otherwise, a non-blocking operation on certain hardware could overlap communication and computation, thus improving performance.

MPI_Isend() begins a standard nonblocking message send.

```
MPI_Isend (void *buf, int count, MPI_Datatype
     dtype, int dest, int tag, MPI_Comm comm,
     MPI_Request *req);
```

Likewise, MPI_Irecv() begins a nonblocking message receive.

```
MPI_Irecv (void *buf, int count, MPI_Datatype
     dtype, int source, int tag, MPI_Comm comm,
     MPI_Request *req);
```

Request Completion

Both routines accept arguments with the same meaning as their blocking counterparts. When the application wishes to complete a nonblocking send or receive, a completion routine is called with the corresponding request. The Test() routine is nonblocking and the Wait() routine is blocking. Other completion routines operate on multiple requests.

```
MPI_Test (MPI_Request *req, int *flag,
     MPI_Status *status);
MPI_Wait (MPI_Request *req, MPI_Status *status);
```

MPI_Test() returns a flag in an output argument that indicates if the request completed. If true, the status object argument is filled with information. If the request was a receive operation, the status object is filled as in MPI_Recv(). Since MPI_Wait() blocks until completion, the status object argument is always filled.

Probe

MPI_Iprobe() is the nonblocking counterpart of MPI_Probe(), but it does not return a request object since it does not begin any message transfer that would need to complete. It sets the flag argument which indicates the presence of a matching message (for a subsequent receive).

```
MPI_Iprobe (int source, int tag, MPI_Comm comm,
     int *flag, MPI_Status *status);
```

Programmers should not consider the nonblocking routines as simply fast versions of the blocking calls and therefore the preferred choice in all applications. Some implementations cannot take advantage of the opportunity to optimize performance offered by the nonblocking routines. In order to preserve the semantics of the message-passing interface, some implementations may even be slower with nonblocking transfers. Programmers should have a clear and substantial computation overlap before considering nonblocking routines.

`MPI_Type_vector`	Create a strided homogeneous vector.
`MPI_Type_struct`	Create a heterogeneous structure.
`MPI_Address`	Get absolute address of memory location.
`MPI_Type_commit`	Use datatype in message transfers.
`MPI_Pack`	Pack element into contiguous buffer.
`MPI_Unpack`	Unpack element from contiguous buffer.
`MPI_Pack_size`	Get packing buffer size requirement.

`MPI_Type_continuous`	Create contiguous homogeneous array.
`MPI_Type_hvector`	Create vector with byte displacement.
`MPI_Type_indexed`	Create a homogeneous structure.
`MPI_Type_hindexed`	Create an index with byte displacements.
`MPI_Type_extent`	Get range of space occupied by a datatype.
`MPI_Type_size`	Get amount of space occupied by a datatype.
`MPI_Type_lb`	Get displacement of datatype's lower bound.
`MPI_Type_ub`	Get displacement of datatype's upper bound.
`MPI_Type_free`	Free a datatype.

Message Datatypes

Heterogeneous computing requires that message data be typed or described somehow so that its machine representation can be converted as necessary between computer architectures. MPI can thoroughly describe message datatypes, from the simple primitive machine types to complex structures, arrays and indices.

The message-passing routines all accept a datatype argument, whose C typedef is MPI_Datatype. For example, recall MPI_Send(). Message data is specified as a number of elements of a given type.

Several MPI_Datatype values, covering the basic data units on most computer architectures, are predefined:

`MPI_CHAR`	signed char
`MPI_SHORT`	signed short
`MPI_INT`	signed int
`MPI_LONG`	signed long
`MPI_UNSIGNED_CHAR`	unsigned char
`MPI_UNSIGNED_SHORT`	unsigned short
`MPI_UNSIGNED`	unsigned int
`MPI_UNSIGNED_LONG`	unsigned long
`MPI_FLOAT`	float

`MPI_DOUBLE`	double
`MPI_LONG_DOUBLE`	long double
`MPI_BYTE`	a raw byte

The number of bytes occupied by these basic datatypes follows the corresponding C definition. Thus, MPI_INT could occupy four bytes on one machine and eight bytes on another machine. A message count of one MPI_INT specified by both sender and receiver would, in one direction, require padding and always be correct. In the reverse direction, the integer may not be representable in the lesser number of bytes and the communication will fail.

Derived Datatypes

Derived datatypes are built by combining basic datatypes, or previously built derived datatypes. A derived datatype describes a memory layout which consists of multiple arrays of elements. A generalization of this capability is that the four varieties of constructor routines offer more or less control over array length, array element datatype and array displacement.

contiguous	one array length, no displacement, one datatype
vector	one array length, one displacement, one datatype
indexed	multiple array lengths, multiple displacements, one datatype
structure	multiple everything

Strided Vector Datatype

Consider a two dimensional matrix with R rows and C columns stored in row major order. The application wishes to communicate one entire column. A vector derived datatype fits the requirement.

```
MPI_Type_Vector (int count, int blocklength,
     int stride, MPI_Datatype oldtype,
     MPI_Datatype *newtype);
```

Assuming the matrix elements are of MPI_INT, the arguments for the stated requirement would be:

```
int                      R, C;
MPI_Datatype             newtype;
MPI_Type_vector(R, 1, C, MPI_INT, &newtype);
MPI_Type_commit(&newtype);
```

The count of blocks (arrays) is the number of elements in a column (R). Each block contains just one element and the elements are strided (displaced) from each other by the number of elements in a row (C).[1]

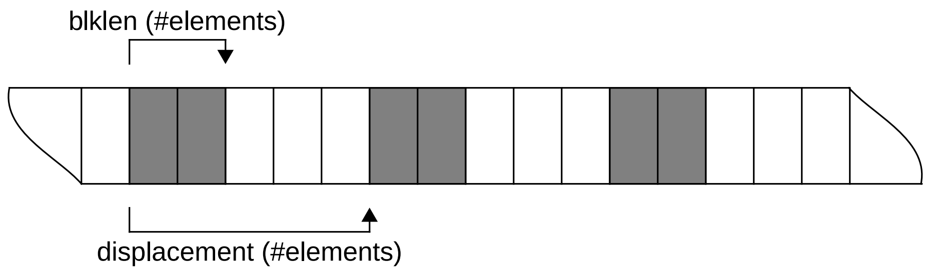

Figure 2: Strided Vector Datatype

Structure Datatype

An arbitrary record whose template is a C structure is a common message form. The most flexible MPI derived datatype, the structure, is required to describe the memory layout.

```
MPI_Type_struct (int count, int blocklengths[],
      MPI_Aint displacements[], MPI_Datatype
      dtypes[], MPI_Datatype *newtype);
```

In the following code fragment, a C struct of diverse fields is described with MPI_Type_struct() in the safest, most portable manner.

```
/*
 * non-trivial structure
 */
struct cell {
      double            energy;
      char              flags;
      float             coord[3];
};
/*
 * We want to be able to send arrays of this datatype.
 */
struct cell            cloud[2];
/*
 * new datatype for cell struct
 */
MPI_Datatype            celltype;
```

1. Note that this datatype is not sufficient to send multiple columns from the matrix, since it does not presume the final displacement between the last element of the first column and the first element of the second column. One solution is to use MPI_Type_struct() and MPI_UB. See *Structure Datatype*.

```
int                     blocklengths[4] = {1, 1, 3, 1};
MPI_Aint                base;
MPI_Aint                displacements[4];
MPI_Datatype            types[4] = {MPI_DOUBLE, MPI_CHAR,
                              MPI_FLOAT, MPI_UB};
MPI_Address(&cloud[0].energy, &displacement[0]);
MPI_Address(&cloud[0].flags, &displacement[1]);
MPI_Address(&cloud[0].coord, &displacement[2]);
MPI_Address(&cloud[1].energy, &displacement[3]);
base = displacement[0];
for (i = 0; i < 4; ++i) displacement[i] -= base;
MPI_Type_struct(4, blocklengths, displacements, types,
            &celltype);
MPI_Type_commit(&celltype);
```

The displacements in a structure datatype are byte offsets from the first storage location of the C structure. Without guessing the compiler's policy for packing and alignment in a C structure, the MPI_Address() routine and some pointer arithmetic are the best way to get the precise values. MPI_Address() simply returns the absolute address of a location in memory. The displacement of the first element within the structure is zero.

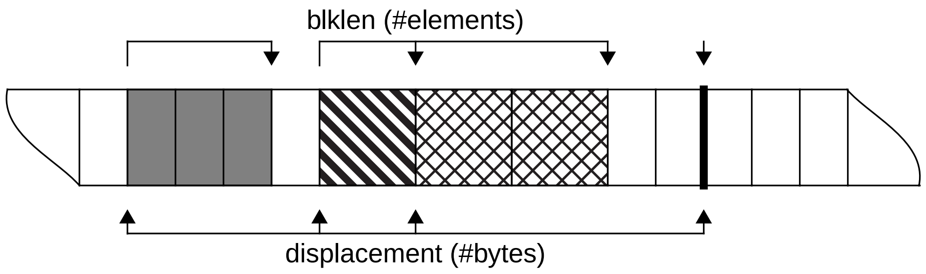

Figure 3: Struct Datatype

When transferring arrays of a given datatype (by specifying a count greater than 1 in MPI_Send(), for example), MPI assumes that the array elements are stored contiguously. If necessary, a gap can be specified at the end of the derived datatype memory layout by adding an artificial element of type MPI_UB, to the datatype description and giving it a displacement that extends to the first byte of the second element in an array.

MPI_Type_Commit() separates the datatypes that will be used to transfer messages from the intermediate ones that are scaffolded on the way to some very complicated datatype. A derived datatype must be committed before being used in communication.

Packed Datatype

The description of a derived datatype is fixed after creation at runtime. If any slight detail changes, such as the blocklength of a particular field in a structure, a new datatype is required. In addition to the tedium of creating many derived datatypes, a receiver may not know in advance which of a nearly identical suite of datatypes will arrive in the next message. MPI's solution is packing and unpacking routines that incrementally assemble and disassemble a contiguous message buffer. The packed message has the special MPI datatype, MPI_PACKED, and is transferred with a count equal to its length in bytes.

```
MPI_Pack_size (int incount, MPI_Datatype dtype,
     MPI_Comm comm, int *size);
```

MPI_Pack_size() returns the packed message buffer size requirement for a given datatype. This may be greater than one would expect from the type description due to hidden, implementation dependent packing overhead.

```
MPI_Pack (void *inbuf, int incount, MPI_Datatype
     dtype, void *outbuf, int outsize,
     int *position, MPI_Comm comm);
```

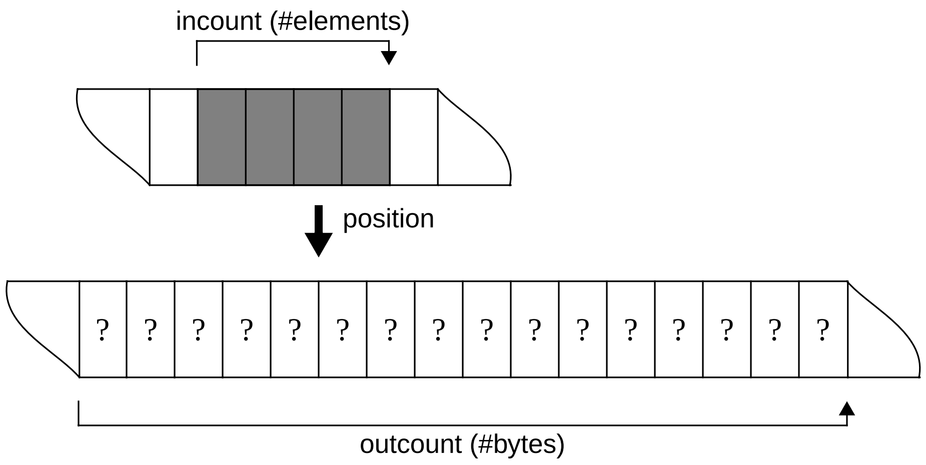

Figure 4: Packed Datatype

Contiguous blocks of homogeneous elements are packed one at a time with MPI_Pack(). After each call, the current location in the packed message buffer is updated. The "in" data are the elements to be packed and the "out" data is the packed message buffer. The outsize is always the maximum size of the packed message buffer, to guard against overflow.

```
MPI_Unpack (void *inbuf, int insize,
      int *position, void *outbuf, int outcount,
      MPI_Datatype datatype, MPI_Comm comm);
```

MPI_Unpack() is the natural reverse of MPI_Pack() where the “in” data is the packed message buffer and the “out” data are the elements to be unpacked.

Consider a networking application that is transferring a variable length message consisting of a count, several (count) Internet addresses as four byte character arrays and an equal number of port numbers as shorts.

```
#define MAXN                100
unsigned char               addrs[MAXN][4];
short                       ports[MAXN];
```

In the following code fragment, a message is packed and sent based on a given count.

```
unsigned int                membersize, maxsize;
int                         position;
int                         nhosts;
int                         dest, tag;
char                        *buffer;
/*
 * Do this once.
 */
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &membersize);
maxsize = membersize;
MPI_Pack_size(MAXN * 4, MPI_UNSIGNED_CHAR, MPI_COMM_WORLD,
            &membersize);
maxsize += membersize;
MPI_Pack_size(MAXN, MPI_SHORT, MPI_COMM_WORLD, &membersize);
maxsize += membersize;
buffer = malloc(maxsize);
/*
 * Do this for every new message.
 */
nhosts = /* some number less than MAXN */ 50;
position = 0;
MPI_Pack(nhosts, 1, MPI_INT, buffer, maxsize, &position,
            MPI_COMM_WORLD);
MPI_Pack(addrs, nhosts * 4, MPI_UNSIGNED_CHAR, buffer,
            maxsize, &position, MPI_COMM_WORLD);
MPI_Pack(ports, nhosts, MPI_SHORT, buffer, maxsize,
            &position, MPI_COMM_WORLD);
MPI_Send(buffer, position, MPI_PACKED, dest, tag,
            MPI_COMM_WORLD);
```

A buffer is allocated once to contain the maximum size of a packed message. In the following code fragment, a message is received and unpacked, based on a count packed into the beginning of the message.

```
int                     src;
int                     msgsize;
MPI_Status              status;
MPI_Recv(buffer, maxsize, MPI_PACKED, src, tag,
            MPI_COMM_WORLD, &status);
position = 0;
MPI_Get_count(&status, MPI_PACKED, &msgsize);
MPI_Unpack(buffer, msgsize, &position, &nhosts, 1, MPI_INT,
            MPI_COMM_WORLD);
MPI_Unpack(buffer, msgsize, &position, addrs, nhosts * 4,
            MPI_UNSIGNED_CHAR, MPI_COMM_WORLD);
MPI_Unpack(buffer, msgsize, &position, ports, nhosts,
            MPI_SHORT, MPI_COMM_WORLD);
```

`MPI_Bcast`	Send one message to all group members.
`MPI_Gather`	Receive and concatenate from all members.
`MPI_Scatter`	Separate and distribute data to all members.
`MPI_Reduce`	Combine messages from all members.
`MPI_Barrier`	Wait until all group members reach this point.
`MPI_Gatherv`	Vary counts and buffer displacements.
`MPI_Scatterv`	Vary counts and buffer displacements.
`MPI_Allgather`	Gather and then broadcast.
`MPI_Allgatherv`	Variably gather and then broadcast.
`MPI_Alltoall`	Gather and then scatter.
`MPI_Alltoallv`	Variably gather and then scatter.
`MPI_Op_create`	Create reduction operation.
`MPI_Allreduce`	Reduce and then broadcast.
`MPI_Reduce_scatter`	Reduce and then scatter.
`MPI_Scan`	Perform a prefix reduction.

Collective Message-Passing

Collective operations consist of many point-to-point messages which happen more or less concurrently (depending on the operation and the internal algorithm) and involve all processes in a given communicator. Every process must call the same MPI collective routine. Most of the collective operations are variations and/or combinations of four primitives: broadcast, gather, scatter and reduce.

Broadcast

```
MPI_Bcast (void *buf, int count, MPI_Datatype
     dtype, int root, MPI_Comm comm);
```

In the broadcast operation, all processes specify the same root process, whose buffer contents will be sent. Processes other than the root specify receive buffers. After the operation, all buffers contain the message from the root process.

Scatter

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

MPI_Scatter() is also a one-to-many collective operation. All processes specify the same receive count. The send arguments are only significant to the root process, whose buffer actually contains sendcount * N elements of the given datatype, where N is the number of processes in the given communicator. The send buffer will be divided equally and dispersed to all pro-

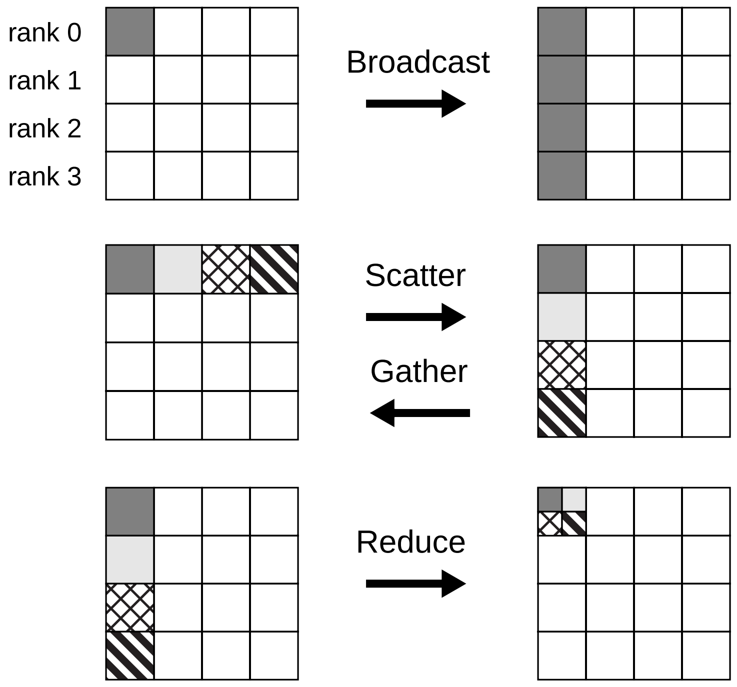

Figure 5: Primitive Collective Operations

cesses (including itself). After the operation, the root has sent sendcount elements to each process in increasing rank order. Rank 0 receives the first sendcount elements from the send buffer. Rank 1 receives the second sendcount elements from the send buffer, and so on.

Gather

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

MPI_Gather() is a many-to-one collective operation and is a complete reverse of the description of MPI_Scatter().

Reduce

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

MPI_Reduce() is also a many-to-one collective operation. All processes specify the same count and reduction operation. After the reduction, all processes have sent count elements from their send buffer to the root process.

Elements from corresponding send buffer locations are combined pair-wise to yield a single corresponding element in the root process's receive buffer. The full reduction expression over all processes is always associative and may or may not be commutative. Application specific reduction operations can be defined at runtime. MPI provides several pre-defined operations, all of which are commutative. They can be used only with sensible MPI pre-defined datatypes.

`MPI_MAX`	maximum
`MPI_MIN`	minimum
`MPI_SUM`	sum
`MPI_PROD`	product
`MPI_LAND`	logical and
`MPI_BAND`	bitwise and
`MPI_LOR`	logical or
`MPI_BOR`	bitwise or
`MPI_LXOR`	logical exclusive or
`MPI_BXOR`	bitwise exclusive or

The following code fragment illustrates the primitive collective operations together in the context of a statically partitioned regular data domain (e.g., 1-D array). The global domain information is initially obtained by the root process (e.g., rank 0) and is broadcast to all other processes. The initial dataset is also obtained by the root and is scattered to all processes. After the computation phase, a global maximum is returned to the root process followed by the new dataset itself.

```
/*
 * parallel programming with a single control process
 */
      int               root;
      int               rank, size;
      int               i;
      int               full_domain_length;
      int               sub_domain_length;
      double            *full_domain, *sub_domain;
      double            local_max, global_max;
      root = 0;
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
/*
 * Root obtains full domain and broadcasts its length.
 */
      if (rank == root) {
            get_full_domain(&full_domain,
                  &full_domain_length);
      }
      MPI_Bcast(&full_domain_length, 1 MPI_INT, root,
            MPI_COMM_WORLD);
/*
 * Distribute the initial dataset.
 */
      sub_domain_length = full_domain_length / size;
      sub_domain = (double *) malloc(sub_domain_length *
            sizeof(double));
      MPI_Scatter(full_domain, sub_domain_length,
            MPI_DOUBLE, sub_domain, sub_domain_length,
            MPI_DOUBLE, root, MPI_COMM_WORLD);
/*
 * Compute the new dataset.
 */
      compute(sub_domain, sub_domain_length, &local_max);
/*
 * Reduce the local maxima to one global maximum
 * at the root.
 */
      MPI_Reduce(&local_max, &global_max, 1, MPI_DOUBLE,
            MPI_MAX, root, MPI_COMM_WORLD);
/*
 * Collect the new dataset.
 */
      MPI_Gather(sub_domain, sub_domain_length, MPI_DOUBLE,
            full_domain, sub_domain_length, MPI_DOUBLE,
            root, MPI_COMM_WORLD);
```

`MPI_Comm_dup`	Duplicate communicator with new context.
`MPI_Comm_split`	Split into categorized sub-groups.
`MPI_Comm_free`	Release a communicator.
`MPI_Comm_remote_size`	Count intercomm. remote group members.
`MPI_Intercomm_merge`	Create an intracomm. from an intercomm.
`MPI_Comm_compare`	Compare two communicators.
`MPI_Comm_create`	Create a communicator with a given group.
`MPI_Comm_test_inter`	Test for intracommunicator or intercommunicator.
`MPI_Intercomm_create`	Create an intercommunicator.
`MPI_Group_size`	Get number of processes in group.
`MPI_Group_rank`	Get rank of calling process.
`MPI_Group_translate_ranks`	Processes in group A have what ranks in B?
`MPI_Group_compare`	Compare membership of two groups.
`MPI_Comm_group`	Get group from communicator.
`MPI_Group_union`	Create group with all members of 2 others.
`MPI_Group_intersection`	Create with common members of 2 others.
`MPI_Group_difference`	Create with the complement of intersection.
`MPI_Group_incl`	Create with specific members of old group.
`MPI_Group_excl`	Create with the complement of incl.
`MPI_Group_range_incl`	Create with ranges of old group members.
`MPI_Group_range_excl`	Create with the complement of range_incl.
`MPI_Group_free`	Release a group object.

Creating Communicators

A communicator could be described simply as a process group. Its creation is synchronized and its membership is static. There is no period in user code where a communicator is created but not all its members have joined. These qualities make communicators a solid parallel programming foundation. Three communicators are prefabricated before the user code is first called: MPI_COMM_WORLD, MPI_COMM_SELF and MPI_COMM_PARENT. See *Basic Concepts*.

Communicators carry a hidden synchronization variable called the context. If two processes agree on source rank, destination rank and message tag, but use different communicators, they will not synchronize. The extra synchronization means that the global software industry does not have to divide, allocate or reserve tag values. When writing a library or a module of an application, it is a good idea to create new communicators, and hence a private synchronization space. The simplest MPI routine for this purpose is MPI_Comm_dup(), which duplicates everything in a communicator, particularly the group membership, and allocates a new context.

```
MPI_Comm_dup (MPI_comm comm, MPI_comm *newcomm);
```

Applications may wish to split into many subgroups, sometimes for data parallel convenience (i.e. a row of a matrix), sometimes for functional grouping (i.e. multiple distinct programs in a dataflow architecture). The group membership can be extracted from the communicator and manipulated by an entire suite of MPI routines. The new group can then be used to create a new communicator. MPI also provides a powerful routine, MPI_Comm_split(), that starts with a communicator and results in one or more new communicators. It combines group splitting with communicator creation and is sufficient for many common application requirements.

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

The color and key arguments guide the group splitting. There will be one new communicator for each value of color. Processes providing the same value for color will be grouped in the same communicator. Their ranks in the new communicator are determined by sorting the key arguments. The lowest value of key will become rank 0. Ties are broken by rank in the old communicator. To preserve relative order from the old communicator, simply use the same key everywhere.

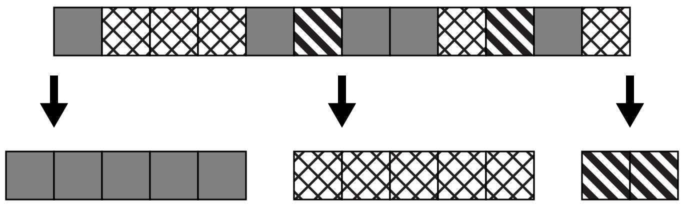
Figure 6: Communicator Split

A communicator is released by MPI_Comm_free(). Underlying system resources may be conserved by releasing unwanted communicators.

```
MPI_Comm_free (MPI_Comm *comm);
```

Inter-communicators

An intercommunicator contains two groups: a local group in which the owning process is a member and a remote group of separate processes. The remote process group has the mirror image intercommunicator - the groups are flipped. Spawning new processes creates an intercommunicator. See *Dynamic Processes*. MPI_Intercomm_merge() creates an intracommunicator (the common form with a single group) from an intercommunicator. This is often done to permit collective operations, which can only be done on intracommunicators.

```
MPI_Intercomm_merge (MPI_Comm intercomm,
     int high, MPI_Comm *newintracomm);
```

The new intracommunicator group contains the union of the two groups of the intercommunicator. The operation is collective over both groups. Rank ordering within the two founding groups is maintained. Ordering between the two founding groups is controlled by the high parameter, a boolean value. The intercommunicator group that sets this parameter true will occupy the higher ranks in the intracommunicator.

The number of members in the remote group of an intercommunicator is obtained by MPI_Comm_remote_size().

```
MPI_Comm_remote_size (MPI_Comm comm, int *size);
```

Fault Tolerance

Some MPI implementations may invalidate a communicator if a member process dies. The MPI library may raise an error condition on any attempt to use a dead communicator, including requests in progress whose communicator suddenly becomes invalid. These faults would then be detectable at the application level by setting a communicator's error handler to MPI_ERRORS_RETURN (See *Miscellaneous MPI Features*).

A crude but portable fault tolerant master/slave application can be constructed by using the following strategy:

- Spawn processes in groups of one.
- Set the error handler for the parent / child intercommunicators to MPI_ERRORS_RETURN.
- If a communication with a child returns an error, assume it is dead and free the intercommunicator.
- Spawn another process, if desired, to replace the dead process. See *Dynamic Processes*.

`MPI_Cart_create`	Create cartesian topology communicator.
`MPI_Dims_create`	Suggest balanced dimension ranges.
`MPI_Cart_rank`	Get rank from cartesian coordinates.
`MPI_Cart_coords`	Get cartesian coordinates from rank.
`MPI_Cart_shift`	Determine ranks for cartesian shift.
`MPI_Cart_sub`	Split into lower dimensional sub-grids.
`MPI_Graph_create`	Create arbitrary topology communicator.
`MPI_Topo_test`	Get type of communicator topology.
`MPI_Graphdims_get`	Get number of edges and nodes.
`MPI_Graph_get`	Get edges and nodes.
`MPI_Cartdim_get`	Get number of dimensions.
`MPI_Cart_get`	Get dimensions, periodicity and local coordinates.
`MPI_Graph_neighbors_count`	Get number of neighbors in a graph topology.
`MPI_Graph_neighbors`	Get neighbor ranks in a graph topology.
`MPI_Cart_map`	Suggest new ranks in an optimal cartesian mapping.
`MPI_Graph_map`	Suggest new ranks in an optimal graph mapping.

Process Topologies

MPI is a process oriented programming model that is independent of underlying nodes in a parallel computer. Nevertheless, to enhance performance, the data movement patterns in a parallel application should match, as closely as possible, the communication topology of the hardware. Since it is difficult for compilers and message-passing systems to guess at an application's data movement, MPI allows the application to supply a topology to a communicator, in the hope that the MPI implementation will use that information to identify processes in an optimal manner.

For example, if the application is dominated by Cartesian communication and the parallel computer has a cartesian topology, it is preferable to align the distribution of data with the machine, and not blindly place any data coordinate at any node coordinate.

MPI provides two types of topologies, the ubiquitous cartesian grid, and an arbitrary graph. Topology information is attached to a communicator by creating a new communicator. MPI_Cart_create() does this for the cartesian topology.

```
MPI_Cart_create (MPI_Comm oldcomm, int ndims,
     int *dims, int *periods, int reorder,
     MPI_Comm *newcomm);
```

The essential information for a cartesian topology is the number of dimensions, the length of each dimension and a periodicity flag (does the dimension wrap around?) for each dimension. The reorder argument is a flag that indicates if the application will allow a different ranking in the new topology communicator. Reordering may make coordinate calculation easier for the MPI implementation.

With a topology enhanced communicator, the application will use coordinates to decide source and destination ranks. Since MPI communication routines still use ranks, the coordinates must be translated into a rank and vice versa. MPI eases this translation with MPI_Cart_rank() and MPI_Cart_coords().

```
MPI_Cart_rank (MPI_comm comm, int *coords,
     int *rank);
MPI_Cart_coords (MPI_Comm comm, int rank,
     int maxdims, int *coords);
```

To further assist process identification in cartesian topology applications, MPI_Cart_shift() returns the ranks corresponding to common neighbourly shift communication. The direction (dimension) and relative distance are input arguments and two ranks are output arguments, one on each side of the calling process along the given direction. Depending on the periodicity of the cartesian topology associated with the given communicator, one or both ranks may be returned as MPI_PROC_NULL, indicating a shift off the edge of the grid.

```
MPI_Cart_shift (MPI_Comm comm, int direction,
     int distance, int *rank_source,
     *int rank_dest);
```

Consider a two dimensional cartesian dataset. The following code skeleton establishes a corresponding process topology for any number of processes, and then creates a new communicator for collective operations on the first column of processes. Finally, it obtains the ranks which hold the previous and next rows, which would lead to data exchange.

```
     int              mycoords[2];
     int              dims[2];
     int              periods[2] = {1, 0};
     int              rank_prev, rank_next;
     int              size;
     MPI_Comm         comm_cart;
     MPI_Comm         comm_col1;
```

```
/*
 * Create communicator with 2D grid topology.
 */
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Dims_create(size, 2, dims);
      MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1,
            &comm_cart);
/*
 * Get local coordinates.
 */
      MPI_Comm_rank(comm_cart, &rank);
      MPI_Cart_coords(comm_cart, rank, 2, mycoords);
/*
 * Build new communicator on first column.
 */
      if (mycoords[1] == 0) {
            MPI_Comm_split(comm_cart, 0, mycoords[0],
                  &comm_col1);
      } else {
            MPI_Comm_split(comm_cart, MPI_UNDEFINED, 0,
                  &comm_col1);
}
/*
 * Get the ranks of the next and previous rows, same column.
 */
      MPI_Cart_shift(comm_cart, 0, 1, &rank_prev,
            &rank_next);
```

MPI_Dims_create() suggests the most balanced (“square”) dimension ranges for a given number of nodes and dimensions.

A good reason for building a communicator over a subset of the grid, in this case the first column in a mesh, is to enable the use of collective operations. See *Collective Message-Passing*.

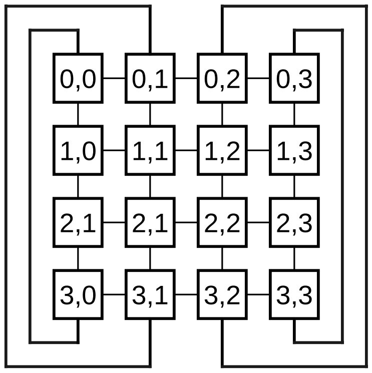

Figure 7: 2D Cartesian Topology

`MPI_Spawn`	Start copies of one program.
`MPI_Spawn_multiple`	Start multiple programs.
`MPI_Port_open`	Obtain a connection point for a server.
`MPI_Port_close`	Release a connection point.
`MPI_Accept`	Accept a connection from a client.
`MPI_Connect`	Make a connection to a server.
`MPI_Name_publish`	Publish a connection point under a service name.
`MPI_Name_unpublish`	Stop publishing a connection point.
`MPI_Name_get`	Get connection point from service name.
`MPI_Info_create`	Create a new info object.
`MPI_Info_set`	Store a key/value pair to an info object.
`MPI_Info_get`	Read the value associated with a stored key.
`MPI_Info_get_valuelen`	Get the length of a key value.
`MPI_Info_get_nkeys`	Get number of keys stored with an info object.
`MPI_Info_get_nthkey`	Get the key name in a sequence position.
`MPI_Info_dup`	Duplicate an info object.
`MPI_Info_free`	Destroy an info object.
`MPI_Info_delete`	Remove a key/value pair from an info object.

Process Creation

Due to the static nature of process groups in MPI (a virtue), process creation must be done carefully. Process creation is a collective operation over a given communicator. A group of processes are created by one call to MPI_Spawn(). The child processes start up, initialize and communicate in the traditional MPI way. They must begin by calling MPI_Init(). The child group has its own MPI_COMM_WORLD which is distinct from the world communicator of the parent group.

```
MPI_Spawn (char program[], char *argv[], int
      maxprocs, MPI_Info info, int root, MPI_Comm,
      parents, MPI_Comm *children, int errs[]);
```

How do the parents communicate with their children? The natural mechanism for communication between two groups is the intercommunicator. An intercommunicator whose remote group contains the children is returned to the parents in the second communicator argument of MPI_Spawn(). The children get the mirror communicator, whose remote group contains the parents, as the pre-defined communicator MPI_COMM_PARENT. In the application's original process world that has no parent, the remote group of MPI_COMM_PARENT is of size 0. See *Creating Communicators*.

The maxprocs parameter is the number of copies of the single program that will be created. Each process will be passed command line arguments consisting of the program name followed by the arguments specified in the argv parameter. (The argv parameter should not contain the program name.) The program name, maxprocs and argv are only significant in the parent process whose rank is given by the root parameter. The result of each individual process spawn is returned through the errs parameter, an array of MPI error codes.

Portable Resource Specification

New processes require resources, beginning with a processor. The specification of resources is a natural area where the MPI abstraction succumbs to the underlying operating system and all its domestic customs and conventions. It is thus difficult if not impossible for an MPI application to make a detailed resource specification and remain portable. The info parameter to MPI_Spawn is an opportunity for the programmer to choose control over portability. MPI implementations are not required to interpret this argument. Thus the only portable value for the info parameter is MPI_INFO_NULL.

Consult each MPI implementation's documentation for (non-portable) features within the info parameter and for the default behaviour with MPI_INFO_NULL.

A common and fairly abstract resource requirement is simply to fill the available processors with processes. MPI makes an attempt, with no guarantees of accuracy, to supply that information through a pre-defined attribute called MPI_UNIVERSE_SIZE, which is cached on MPI_COMM_WORLD. In typical usage, the application would subtract the value associated with MPI_UNIVERSE_SIZE from the current number of processes, often the size of MPI_COMM_WORLD. The difference is the recommended value for the maxprocs parameter of MPI_Spawn(). See *Miscellaneous MPI Features* on how to retrieve the value for MPI_UNIVERSE_SIZE.

`MPI_Errhandler_create`	Create custom error handler.
`MPI_Errhandler_set`	Set error handler for communicator.
`MPI_Error_string`	Get description of error code.
`MPI_Error_class`	Get class of error code.
`MPI_Abort`	Abnormally terminate application.
`MPI_Attr_get`	Get cached attribute value.
`MPI_Wtime`	Get wall clock time.
`MPI_Errhandler_get`	Get error handler from communicator.
`MPI_Errhandler_free`	Release custom error handler.
`MPI_Get_processor_name`	Get the caller's processor name.
`MPI_Wtick`	Get wall clock timer resolution.
`MPI_Get_version`	Get the MPI version numbers.
`MPI_Keyval_create`	Create a new attribute key.
`MPI_Keyval_free`	Release an attribute key.
`MPI_Attr_put`	Cache an attribute in a communicator.
`MPI_Attr_delete`	Remove cached attribute.

Miscellaneous MPI Features

Error Handling

An error handler is a software routine which is called when a error occurs during some MPI operation. One handler is associated with each communicator and is inherited by created communicators which derive from it. When an error occurs in an MPI routine that uses a communicator, that communicator's error handler is called. An application's initial communicator, MPI_COMM_WORLD, gets a default built-in handler, MPI_ERRORS_ARE_FATAL, which aborts all tasks in the communicator.

An application may supply an error handler by first creating an MPI error handler object from a user routine.

```
MPI_Errhandler_create (void (*function)(),
     MPI_Errhandler *errhandler);
```

Error handler routines have two pre-defined parameters followed by implementation dependent parameters using the ANSI C <stdargs.h> mechanism. The first parameter is the handler's communicator and the second is the error code describing the problem.

```
void function (MPI_Comm *comm, int *code, ...);
```

The error handler object is then associated with a communicator by MPI_Errhandler_set().

```
MPI_Errhandler_set (MPI_Comm comm,
     MPI_Errhandler errhandler);
```

A second built-in error handler is MPI_ERRORS_RETURN, which does nothing and allows the error code to be returned by the offending MPI routine where it can be tested and acted upon. In C the error code is the return value of the MPI function. In Fortran the error code is returned through an error parameter to the MPI subroutine.

```
MPI_Error_string (int code, char *errstring,
     int *resultlen);
```

Error codes are converted into descriptive strings by MPI_Error_string(). The user provides space for the string that is a minimum of MPI_MAX_ERROR_STRING characters in length. The actual length of the returned string is returned through the resultlen argument.

MPI defines a list of standard error codes (also called error classes) that can be examined and acted upon by portable applications. All additional error codes, specific to the implementation, can be mapped to one of the standard error codes. The idea is that additional error codes are variations on one of the standard codes, or members of the same error class. Two standard error codes catch any additional error code that does not fit this intent: MPI_ERR_OTHER (doesn't fit but convert to string and learn something) and MPI_ERR_UNKNOWN (no clue). Again, the goal of this design is portable, intelligent applications.

The mapping of error code to standard error code (class) is done by MPI_Error_class().

```
MPI_Error_class (int code, int class);
```

Attribute Caching

MPI provides a mechanism for storing arbitrary information with a communicator. A registered key is associated with each piece of information and is used, like a database record, for storage and retrieval. Several keys and associated values are pre-defined by MPI and stored in MPI_COMM_WORLD.

`MPI_TAG_UB`	maximum message tag value
`MPI_HOST`	process rank on user's local processor
`MPI_IO`	process rank that can fully accomplish I/O
`MPI_WTIME_IS_GLOBAL`	Are clocks synchronized?
`MPI_UNIVERSE_SIZE`	#processes to fill machine

All cached information is retrieved by calling MPI_Attr_get() and specifying the desired key.

```
MPI_Attr_get (MPI_Comm comm, int keyval,
     void *attr_val, int *flag);
```

The flag parameter is set to true by MPI_Attr_get() if a value has been stored the specified key, as will be the case for all the pre-defined keys.

Timing

Performance measurement is assisted by MPI_Wtime() which returns an elapsed wall clock time from some fixed point in the past.

```
double MPI_Wtime (void);
```