Lecture about MPI

Angela Italiano

March 13,2010

1 Generic notions

The MPI standard does not mandate how a job is started, so there is considerable variation between different MPI implementation and in different situation. For starting a job interactively with MPI, the most common method launches all the processes involved in the MPI program together on nodes obtained from list in a configuration file. All processes edecute the same program. The comand that accomplishes this is usually called and takes the name of the program as a parameter. mpirun is a shell script that attempts to hide the differences in starting jobs for various devices from the user. Mpirun attempts to determine what kind of machine it is running on and start the required number of jobs on that machine. mpirun typically works like this:
`mpirun -np <number of processes> <program name and arguments>`

The MPI enviroment must be initialized. MPI_INIT initializes MPI in this way:

- opens a local socket and binds it to a port;
- sends that information to POE;
- receives a list of destination addresses and ports;
- opens a socket to send to each one;
- verifies that communication can be established;
- distributes MPI internal state to each task.

Another different thing that you can find in a MPI programs is MPI_FINALIZE. This subroutine is the last MPI call. You must be sure that all pending communications involving a task have completed before the task calls MPI_FINALIZE. You must also be sure that all files opened by MPI_FILE_OPEN have been closed before the task calls MPI_FINALIZE. Although this call terminates MPI processing, it does not terminate the task. It is possible to continue with non MPI processing after calling, but no other MPI calls (including MPI_INIT) can be made. In a threads environment, both MPI_INIT and MPI_FINALIZE must be called on the same thread.

MPI provides a set of send and receive functions that allow the communication of typed data with an associated tag. Typing of the message contents is necessary for heterogeneous support - the type information is needed so that

correct data representation conversions can be performed as data is sent from one architecture to another. The tag allows selectivity of messages at the receiving end. Message selectivity on the source process of the message is also provided.

Processes are represented by a unique "rank" (integer) and ranks are numbered 0, 1, 2, ..., N-1. Typically, a process in a parallel application needs to know who it is (its rank) and how many other processes exist. A process finds out its own rank by calling MPI_Comm_rank().

Another important concept about MPI communication is communicator. A key feature needed to support the creation of robust, parallel libraries is to guarantee that communication within a library routine does not conflict with communication extraneous to the routine. The concepts encapsulated by an MPI communicator provide this support.

A communicator is a data object that specifies the scope of a communication operation, that is, the group of processes involved and the communication context. Contexts partition the communication space. A message sent in one context cannot be received in another context. Process ranks are interpreted with respect to the process group associated with a communicator.

New communicators are created from existing communicators and the creation of a communicator is a collective operation.

Communicators are especially important for the design of parallel software libraries. Suppose we have a parallel, matrix multiplication routine as a member of a library.

We would like to allow distinct subgroups of processes to perform different matrix multiplications concurrently. A communicator provides a convenient mechanism for passing into the library routine the group of processes involved, and within the routine, process ranks will be interpreted relative to this group. The grouping and labeling mechanisms provided by communicators are useful, and communicators will typically be passed into library routines that perform internal communications. Unless the programmer is explicitly creating new communicators, the predefined communicator MPI_COMM_WORLD is usually used. Obviously within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero.

2 MPI_STATUS

Send operation has a status field; An object of type MPI_STATUS is not an MPI opaque object; its structure is declared in mpi.h and mpif.h, and it exists in the user's program.

In many cases, application programs are constructed so that it is unnecessary for them to examine the status fields. In these cases, it is a waste for the user to allocate a status object, and it is particularly wasteful for the MPI implementation to fill in fields in this object. To cope with this problem, there are two predefined constants, MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE , which when passed to a receive, wait, or test function, inform the implementation that the status fields are not to be filled in.

Note that MPI_STATUS_IGNORE is not a special type of MPI_STATUS object; rather, it is a special value for the argument. In C one would expect it to be NULL, not the address of a special MPI_STATUS. In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the structure may contain additional fields. Thus, status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

In Fortran, status is an array of INTEGERs of size MPI_STATUS_SIZE. The constants MPI_SOURCE, MPI_TAG and MPI_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI_SOURCE), status(MPI_TAG) and status(MPI_ERROR) contain, respectively, the source, tag and error code of the receivedmessage. In C++, the status object is handled through the following methods:

```
int MPI::Status::Get_source() const
void MPI::Status::Set_source(int source)
int MPI::Status::Get_tag() const
void MPI::Status::Set_tag(int tag)
int MPI::Status::Get_error() const
void MPI::Status::Set_error(int error)
```

In general, message-passing calls do not modify the value of the error code field of status variables.

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to MPI_GET_COUNT is required to "decode" this information.

```
MPI_GET_COUNT(status, datatype, count)
   IN status return status of receive operation (Status)
   IN datatype datatype of each receive buffer entry (handle)
   OUT count number of received entries (integer)
```

Returns the number of entries received. (we count entries, each of type datatype, not bytes.) The datatype argument should match the argument provided by the receive call that set the status variable.

For example, if you receive matrix row and you have defined the type row and then you ask how many data receive and you pass the flow data type which is the element of the matrix, you should get how many flow you received, so you can ask getCount for a different data type.

If you are not using the same data type that you receive you can have a mismatch o an interesting functionality.

Note that a mismatch also happens when MPI is not communicating the whole buffer received. The point is that `MPI_GetCount` is needed because you don't know if you are receiving the whole data you are expecting for; you know the number that is the length of the buffer but there is no guaranty that you will receive as the much data as the buffer is long. You can receive less and in some cases it is possible that you receive more in the sense that you get the

full buffer but there is still data to receive. MPI program is done to allow a client/server application where the client is written by a programmer, server is written by another programmer and all you have to do is the communication stuff and in this case you don't know the exact size for the data passed. Of course there is a communication protocol, but the matrixes passed don't have always the same size. One possibility is to communicate the size of the data as a parameter but it's not the usual action. It also happens that for some failures in communication you only get partial data and the rest of data are automatically sent by MPI and it's a case when you need the control of the size. If you can assume that the whole data is received you cannot control the size and only check the data received; you don't have to use always `MPI_GetCount` but there are some cases. If the size of the datatype is zero, this routine will return a count of zero. If the amount of data in status is not an exact multiple of the size of datatype (so that count would not be integral), a count of MPI_UNDEFINED is returned instead.

3 Null Process

In many instances, it is convenient to specify a "dummy" source or destination for communication. This simplifies the code that is needed for dealing with boundaries, for example, in the case of a non-circular shift done with calls to send-receive.

The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI_PROC_NULL has no effect. A send to MPI_PROC_NULL succeeds and returns as soon as possible. A receive from MPI_PROC_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with `source = MPI_PROC_NULL` is executed then the status object returns `source = MPI_PROC_NULL`, `tag = MPI_ANY_TAG` and `count = 0`. This pseudo-process quite helpful, especially when you are programming operations on large matrixes, because it's a process that actually doesn't exist but it's is present in each communicator.

You can always try to communicate with the process null and you will always receive 0 bytes and you will always be able to send every amount of data and the send will be a success as there is a real process that receive the data. This particular process is needed for some regions. You can use for testing process .

There are a lot of situations when you have to do operations like this, for example in the border because you don't want to communicate outside and get stuck or you just define your neighbors in a special way.

When you start your program you define the programs with you want to communicate and those process which are the neighbors. There is of course a matching definitions in the standard that says if you are going to receive data from my neighbors is null, the receive will not succeed, so for instance if you receive data you know that they are unsafe then it's up to you to say if I'm on the border than the border must be initialized in some way. One example is solving partial differential equation because what you do is you have a surround initialized outside the program which are the boundary conditions and you want to understand what happens inside, so applying this function, slowly, iteration after iteration, it propagates the results from the border, the boundary, to inside.

So this is a special case when no-touching the buffer is what you want on the border.

If you want to receive always the same value you just put that value in a buffer and then apply the receive with the MPI null process; if your buffer is already filled in the beginning, in this way you'll never change it and of course if you check `MPI_get_Count` you will get 0.

So if you are in the border you may have the same data or empty data while if you are inside you'll have new data each time.

4 Comunication model

We talked about communication models, we explained that there are different ways of implementing send operation and the MPI standard allows the programmer to choose which one use or leave the default by the library implementation. Actually there is not much more beyond the format, infect when you explicitly want a behavior of a send operation, you just change the function and the parameters remain the same.

4.1 Blocking comunication

`MPI_SEND(buf, count, datatype, dest, tag, comm)` is blocking: it does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer. This is the standard communication mode. There are three additional communication modes.

A buffered mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is local, and its completion does not depend on the occurrence of a matching receive.

Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete.

An error will occur if there is insufficient buffer space. The amount of available buffer space is controlled by the user.

A send that uses the synchronous mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive.

If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is non-local.

A send that uses the ready communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined.

On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance.

The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

Three additional send functions are provided for the three additional communication modes. The communication mode is indicated by a one letter prefix: B for buffered, S for synchronous, and R for ready.
For send in buffered mode:

```
MPI_BSEND (buf, count, datatype, dest, tag, comm)
   IN buf initial address of send buffer (choice)
   IN count number of elements in send buffer (non-negative integer)
   IN datatype datatype of each send buffer element (handle)
   IN dest rank of destination (integer)
   IN tag message tag (integer)
   IN comm communicator (handle)
```

For send in synchronous mode:

```
MPI_SSEND (buf, count, datatype, dest, tag, comm)
   IN buf initial address of send buffer (choice)
   IN count number of elements in send buffer (non-negative integer)
   IN datatype datatype of each send buffer element (handle)
   IN dest rank of destination (integer)
   IN tag message tag (integer)
   IN comm communicator (handle)
```

For send in ready mode:

```
MPI_RSEND (buf, count, datatype, dest, tag, comm)
   IN buf initial address of send buffer (choice)
   IN count number of elements in send buffer (non-negative integer)
   IN datatype datatype of each send buffer element (handle)
   IN dest rank of destination (integer)
   IN tag message tag (integer)
   IN comm communicator (handle)
```

There is only one receive operation, but it matches any of the send modes. The receive operation described in the last section is blocking: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed.

4.2 Non blocking comunication

A nonblocking send start call initiates the send operation, but does not complete it. The send start call can return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer.

With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed.

Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it.

The call can return before a message is stored into the receive buffer.

A separate receive complete call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed.

The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer. Nonblocking send start calls can use the same four modes as blocking sends: standard, buffered, synchronous and ready.

In all cases, the send start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code.

Nonblocking communications use opaque request objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it.

These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
   IN buf initial address of send buffer (choice)
   IN count number of elements in send buffer (non-negative integer)
   IN datatype datatype of each send buffer element (handle)
   IN dest rank of destination (integer)
   IN tag message tag (integer)
   IN comm communicator (handle)
   OUT request communication request (handle)
```

If we use the incomplete communication primitive, the result are not safe that means that you cannot reuse it until you have checked the operation has completed. This is actually very useful and corresponds quite to a well-understandable optimization in the case where your communication is not well-performed by the main CPU, but it's performed by communication co-processor. We made the example of a matrix of processor and each one communicates which

the stencils of the neighbors, in this case the communication starts, you wait, another start, you wait and so on. You can start all at the same time and if your hardware allows you to do that you will be faster, on the contrary, you have to sequentially complete. If your hardware is able to perform more communication at the same time, I can prepare a buffer while my transfer the previous buffer on network. Another advantage is that while you are waiting for the communication you can do useful work in local. If the communication in on the processor, this some form of threading actually but does not require that your program define its threads; if the communication is on other stuff, it's not equivalent. These calls allocate a communication request object and associate it with the request handle .The request can be used later to query the status of the communication or wait for its completion.

5 Comunication completation

For complete a nonblocking communication we can use MPI_WAIT. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer. The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed.

```
MPI_WAIT(request, status)
   INOUT request request (handle)
   OUT status status object (Status)
```

A call to MPI_WAIT returns when the operation identified by request is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI_WAIT and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation. The call returns, in status, information on the completed operation. One is allowed to call MPI_WAIT with a null or inactive request argument. In this case the operation returns immediately with empty status. It is convenient to be able to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to MPI_WAITANY or can be used to wait for the completion of one out of several operations. A call to MPI_WAITALL or can be used to wait for all pending operations in a list. A call to MPI_WAITSOME or can be used to complete all enabled operations in a list.

```
MPI_WAITANY (count, array_of_requests, index, status)
   IN count list length (non-negative integer)
   INOUT array_of_requests array of requests (array of handles)
   OUT index index of handle for operation that completed (integer)
   OUT status status object (Status)
```

Blocks until one of the operations associated with the active requests in the array has completed. If more then one operation is enabled and can terminate,

one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing communication. If the request was allocated by a nonblocking communication operation, then it is deallocated and the request handle is set to MPI_REQUEST_NULL.

```
MPI_WAITALL( count, array_of_requests, array_of_statuses)
   IN count lists length (non-negative integer)
   INOUT array_of_requests array of requests (array of handles)
   OUT array_of_statuses array of status objects (array of Status)
```

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations. Both arrays have the same number of valid entries. The i-th entry in array_of_statuses is set to the return status of the i-th operation. Requests that were created by nonblocking communication operations are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. The list may contain null or inactive handles. The call sets to empty the status of each such entry.

6 Collective Comunication

Collective communications transmit data among all processes in a group specified by an intracommunicator object. One function, the barrier, serves to synchronize processes without passing data. The key concept of the collective functions is to have a group or groups of participating processes. The routines do not have group identifiers as explicit arguments. Instead, there is a communicator argument. There are two types of communicators: intra-communicators and inter-communicators. An intracommunicator can be thought of as an indentifier for a single group of processes linked with a context.An intercommunicator identifies two distinct groups of processes linked with a context. All processes in both groups identified by the intercommunicator must call the collective routine. In addition, processes in the same group must call the routine with matching arguments. Note that the "in place" option for intracommunicators does not apply to intercommunicators since in the intercommunicator case there is no communication from a process to itself. Here are the characteristics of MPI collective communication routines:

- Involve coordinated communication within a group of processes identified by an MPI communicator;
- Substitute for a more complex sequence of point-to-point calls ;
- All routines block until they are locally complete
- Communications may, or may not, be synchronized (implementation dependent)
- In some cases, a root process originates or receives all data
- Amount of data sent must exactly match amount of data specified by receiver

- Many variations to basic categories
- No message tags are needed

MPI collective communication can be divided into three subsets: synchronization, data movement, and global computation.

6.1 Barrier Synchronization

In parallel applications in the distributed memory environment, explicit or implicit synchronization is sometimes required. As with other message-passing libraries, MPI provides a function call, MPI_BARRIER, to synchronize all processes within a communicator. A barrier is simply a synchronization primitive. A node calling it will be blocked until all the nodes within the group have called it.

```
MPI_BARRIER( comm )
   IN comm communicator (handle)
```

If comm is an intracommunicator, MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call. If comm is an intercommunicator, the barrier is performed across all processes in the intercommunicator. In this case, all processes in one group (group A) of the intercommunicator may exit the barrier when all of the processes in the other group (group B) have entered the barrier.

6.2 Data Movement Routines

MPI provides three types of collective data movement routines. They are broadcast, gather, and scatter. There are also allgather and alltoall routines. The gather, scatter, allgather, and alltoall routines have vector versions. For their vector versions, each process can send and/or receive a different number of elements. The list of MPI collective some data movement routines are:

- broadcast;
- gather;
- scatter;
- allgather;
- alltoall;

6.2.1 Broadcast

In many cases, one processor needs to send (broadcast) some data (either a scalar or vector) to all the processes in a group. MPI provides the broadcast primitive MPI_BCAST to accomplish this task.

```
MPI_BCAST( buffer, count, datatype, root, comm )
   INOUT buffer starting address of buffer (choice)
```

```
   IN count number of entries in buffer (non-negative integer)
   IN datatype data type of buffer (handle)
   IN root rank of broadcast root (integer)
   IN comm communicator (handle)
```

If comm is an intracommunicator, MPI_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of the group using the same arguments for comm and root. On return, the content of root's buffer is copied to all other processes. General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed. The "in place" option is not meaningful here. If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

6.2.2 Gather and Scatter

If an array is scattered throughout all processors in the group, and one wants to collect each piece of the array into a specified process in the order of process rank, the function to use is GATHER. On the other hand, if one wants to distribute the data into n equal segments, where the ith segment is sent to the ith process in the group which has n processes, use SCATTER. MPI provides two variants of the gather/scatter operations: one in which the numbers of data items collected from/sent to nodes can be different; and a more efficient one in the special case where the number per node is the same.

```
MPI_GATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
root, comm)
   IN sendbuf starting address of send buffer (choice)
   IN sendcount number of elements in send buffer (non-negative integer)
   IN sendtype data type of send buffer elements (handle)
   OUT recvbuf address of receive buffer (choice, significant only
at root)
   IN recvcount number of elements for any single receive (non-negative
integer, significant only at root)
   IN recvtype data type of recv buffer elements (significant only
at root) (handle)
   IN root rank of receiving process (integer)
   IN comm communicator (handle)
```

If comm is an intracommunicator, each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is as if each of the n processes in the group (including the root process) had executed a call to `MPI_Send(sendbuf, sendcount, sendtype, root, ...)`, and the root had executed n calls to `MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount, recvtype, i, ...)`, where `extent(recvtype)` is the type extent obtained from a call to `MPI_Type_extent()`.

An alternative description is that the n messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to `MPI_RECV(recvbuf, recvcount·n, recvtype, ...)`. The receive buffer is ignored for all non-root processes. General, derived datatypes are allowed for both sendtype and recvtype. The type signature of sendcount, sendtype on each process must be equal to the type signature of recvcount, recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed. All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments root and comm must have identical values on all processes. The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

`MPI_SCATTER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`
`IN sendbuf address of send buffer (choice, significant only at root)`
`IN sendcount number of elements sent to each process (non-negative integer, significant only at root)`
`IN sendtype data type of send buffer elements (significant only at root) (handle)`
`OUT recvbuf address of receive buffer (choice)`
`IN recvcount number of elements in receive buffer (non-negative integer)`
`IN recvtype data type of receive buffer elements (handle)`
`IN root rank of sending process (integer)`
`IN comm communicator (handle)`

MPI_SCATTER is the inverse operation to MPI_GATHER. If comm is an intracommunicator, the outcome is as if the root executed n send operations, `MPI_Send(sendbuf + i , sendcount ,extent(sendtype), sendcount, sendtype, i, ...)`, and each process executed a receive, `MPI_Recv(recvbuf, recvcount, recvtype, i, ...)`. An alternative description is that the root sends a message with MPI_Send(sendbuf, sendcount·n, sendtype, ...). This message is split into n equal segments, the i-th segment is sent to the i-th process in the group, and each process receives this message as above. The send buffer is ignored for all non-root processes. The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender

and receiver are still allowed. All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes. If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

6.3 Allgather

MPI_ALLGATHER can be thought of as MPI_GATHER where all processes, not just the root, receive the result. The jth block of the receive buffer is the block of data sent from the jth process.

```
MPI_ALLGATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm)
IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements in send buffer (non-negative
integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcount number of elements received from any process
(nonnegative integer)
IN recvtype data type of receive buffer elements (handle)
IN comm communicator (handle)
```

MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. If comm is an intracommunicator, the outcome of a call to MPI_ALLGATHER is as if all processes executed n calls to `MPI_GATHER(sendbuf,sendcount,sendtype,recvbuf,recvcount, recvtype,root,comm)`, for root = 0 , ..., n-1. The rules for correct usage of MPI_ALLGATHER are easily foundfrom the corresponding rules for MPI_GATHER. If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

6.4 AllToAll

In applications like matrix transpose and FFT, an MPI_ALLTOALL call is very helpful. This is an extension to ALLGATHER where each process sends distinct data to each receiver. The jth block from processor i is received by processor j and stored in ith block.

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
comm)
   IN sendbuf starting address of send buffer (choice)
   IN sendcount number of elements sent to each process (non-negative
integer)
   IN sendtype data type of send buffer elements (handle)
   OUT recvbuf address of receive buffer (choice)
   IN recvcount number of elements received from any process (nonnegative
integer)
   IN recvtype data type of receive buffer elements (handle)
   IN comm communicator (handle)
```

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different. If comm is an intracommunicator, the outcome is as if each process executed a send to each process (itself included) with a call to, `MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount, sendtype, i, ...)`, and a receive from every other process with a call to, `MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount, recvtype, i, ...)`. All arguments on all processes are significant. The argument comm must have identical values on all processes. No "in place" option is supported. If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

6.5 Reduction Operations

The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

```
MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm)
   IN sendbuf address of send buffer (choice)
```

```
   OUT recvbuf address of receive buffer (choice, significant only
at root)
   IN count number of elements in send buffer (non-negative integer)
   IN datatype data type of elements of send buffer (handle)
   IN op reduce operation (handle)
   IN root rank of root process (integer)
   IN comm communicator (handle)
```

If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence.

The operation op is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation.

The following predefined operations are supplied for MPI_REDUCE and related functions MPI_ALLREDUCE, MPI_REDUCE_SCATTER These operations are invoked by placing some of following in op. Name Meaning

- MPI_MAX :maximum
- MPI_MIN :minimum
- MPI_SUM :sum
- MPI_PROD :product
- MPI_LAND :logical and
- MPI_BAND :bit-wise and
- MPI_LOR :logical or
- MPI_BOR :bit-wise or
- MPI_LXOR :logical exclusive or (xor)
- MPI_BXOR :bit-wise exclusive or (xor)
- MPI_MAXLOC :max value and location
- MPI_MINLOC :min value and location

The operator MPI_MINLOC is used to compute a global minimum and also an index attached to the minimum value. MPI_MAXLOC similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

```
MPI_ALLREDUCE( sendbuf, recvbuf, count, datatype, op, comm)
   IN sendbuf starting address of send buffer (choice)
   OUT recvbuf starting address of receive buffer (choice)
   IN count number of elements in send buffer (non-negative integer)
   IN datatype data type of elements of send buffer (handle)
   IN op operation (handle)
   IN comm communicator (handle)
```

The "in place" option for intracommunicators is specified by passing the value

MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data. If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide count and datatype arguments that specify the same type signature.

MPI_OP_CREATE binds a user-defined global operation to an op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. The user-defined operation is assumed to be associative.

```
   MPI_OP_CREATE(function, commute, op)
   IN function user defined function (function)
   IN commute true if commutative; false otherwise.
   OUT op operation (handle)
```

If commute = true, then the operation should be both commutative and associative. If commute = false, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, talking advantage of the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity. function is the user-defined function, which must have the following four arguments: invec, inoutvec, len and datatype.

```
   MPI_SCAN( sendbuf, recvbuf, count, datatype, op, comm )
   IN sendbuf starting address of send buffer (choice)
   OUT recvbuf starting address of receive buffer (choice)
   IN count number of elements in input buffer (non-negative integer)
   IN datatype data type of elements of input buffer (handle)
```

```
IN op operation (handle)
IN comm communicator (handle)
```

If comm is an intracommunicator, MPI_SCAN is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0,...,i (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer, and replaced by the output data. This operation is invalid for intercommunicators.

7 Datatype

More general communication buffers are specified by replacing the basic datatypes that have been used so far with derived datatypes that are constructed from basic datatypes using the constructors described in this section. These methods of constructing derived datatypes can be applied recursively. A general datatype is an opaque object that specifies two things:

- A sequence of basic datatypes;
- A sequence of integer (byte) displacements;

The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. We call such a pair of sequences (or sequence of pairs) a type map. The sequence of basic datatypes (displacements ignored) is the type signature of the datatype.

$$Typemap = (type_0, disp_0), .., (type_{n-1}, disp_{n-1})$$

be such a type map, where $type_i$ are basic types, and $disp_i$ are displacements. Let

$$Typesig = (type_0, ..., type_{n-1})$$

be the associated type signature. This type map, together with a base address buf, specifies a communication buffer: the communication buffer that consists of n entries, where the i-th entry is at address $buf + disp_i$ and has type $type_i$. A message assembled from such a communication buffer will consist of n values, of the types defined by Typesig.

Most datatype constructors have replication count or block length arguments. Allowed values are nonnegative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent. pied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

$$Typemap = (typ_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$

then

$$lb(Typemap) = \min_j disp_j,$$
$$ub(Typemap) = \max_j(disp_j + sizeof(type_j)) + \epsilon, \text{ and}$$
$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

If $type_i$ requires alignment to a byte address that is a multiple of k_i , then is the least nonnegative increment needed to round extent(Typemap) to the next multiple of $max_i k_i$.

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation MPI_SEND(buf, 1, datatype,...) will use the send buffer defined by the base address buf and the general datatype associated with datatype; it will generate a message with the type signature determined by the datatype argument. MPI_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base address buf and the general datatype associated with datatype.

The displacements in a general datatype are relative to some initial buffer address. Absolute addresses can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI_BOTTOM. The address of a location in memory can be found by invoking the function MPI_GET_ADDRESS.

```
MPI_GET_ADDRESS(location, address)
   IN location location in caller memory (choice)
   OUT address address of location (integer)
```

Returns the (byte) address of location.

The following auxiliary function provides useful information on derived datatypes.

```
MPI_TYPE_SIZE(datatype, size)
   IN datatype datatype (handle)
   OUT size datatype size (integer)
```

MPI_TYPE_SIZE returns the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity.

Suppose we implement gather as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified

the extent using the MPI_UB and MPI_LB values. A new function is provided which returns the true extent of the datatype and replaces the three functions MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT because are deprecated.

```
MPI_TYPE_GET_EXTENT(datatype, lb, extent)
   IN datatype datatype to get information on (handle)
   OUT lb lower bound of datatype (integer)
   OUT extent extent of datatype (integer)
```

Returns the lower bound and the extent of datatype. true_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring MPI_LB markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring MPI_LB and MPI_UB markers, and performing no rounding for alignment. If the typemap associated with datatype is

$$Typemap = (type_0, disp_0), ..., (type_{n-1}, disp_{n-1})$$

```
MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)
IN oldtype input datatype (handle)
IN lb new lower bound of datatype (integer)
IN extent new extent of datatype (integer)
OUT newtype output datatype (handle)
```

Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be $lb + extent$. Any previous lb and ub markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and extent arguments. This affects the behavior of the datatype when used in communication operations, with $count > 1$, and when used in the construction of new derived datatypes.