D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

February 4, 1997

This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 6

Extended Collective Operations

6.1 Introduction

Current Status: No votes

Collective communication capabilities are here for MPI-2, covering these areas:

- Extension of MPI collective operations to intercommunicators
- Extension of MPI collective operations to in-place buffers.
- Two-phase collective communication of a limited form and a limited set of operations.
- A generalized all-to-all collective operation

6.2 Two-phase Collective Communication

Current Status: no votes

In some applications, better performance can be achieved by separating the initiation and completion of a collective operation. For example, in some numerical applications, better performance can be achieved by overlapping other work (both computation and communication) with an MPI_Allreduce. At the same time, the full generality of non-blocking collective operations, considered at length by the MPI Forum, is both not needed by some applications, and can be a burden for some MPI implementations. Instead, a very restricted set of collective operations that can be started and completed with separate MPI calls are provided. These are called "two phase" collective communication routines. For each of these operations, there is a separate start and end routine. For example, for the two-phase allreduce operation, the calls are

```
MPI_Allreduce_start( sbuf, rbuf, count, datatype, op, comm );
...
other work and/or MPI calls
...
MPI_Allreduce_end( rbuf, comm );
```

The reason for the `rbuf` in the end call is described below.

These routines allow applications to separate the process of data transmission from synchronization, much as the non-blocking point-to-point operations (such as MPI_Isend and MPI_Irecv) do for point-to-point communication. The two-phase versions can provide important performance advantages even on platforms with fast collective operations by allowing applications to avoid unnecessary synchronizations. For example, an MPI implementation with some shared memory may choose to implement these operations by using a small amount of shared memory attached to each communicator and a special algorithm for each two-phase operation. An MPI implementation without shared memory but with a multicast or simple data-collection capability may use these features. Because the two-phase operations are distinct from the other collective operations, the two-phase version can be optimized, and there is no impact on the implementation or performance of the other collective operations.

6.2.1 General rules for all two phase collective communication

- Only one two-phase collective communication many be active on any particular communicator and MPI process at any time. In other words, on any MPI process, each communicator may have a most one active two-phase operation at any time. The inclusion of process here is intend to make clear that "at any time" applies to each process separately.

  Discussion: could be weakened to only one of each kind.

- Start calls are local; their completion does not depend on the action of any other process.
- End calls are collective over the communicator and follow the ordering rules for collective calls. End calls match the preceeding start call for the same collective operation. When and "end" call is made, exactly one unmatched "start" call for the same operation must preceed it.
- An implementation is free to implement any two-phase collective operation using the corresponding blocking collective routine when the end call (e.g., MPI_Allreduce_end) is issued. The start call is provided to allow the user and MPI implementation to optimize the collective operation.
- Two phase collective operations do not match the corresponding regular collective operation. For example, an MPI_Allreduce and an MPI_Allreduce_start/end do not match.
- Two phase routines that have a destination buffer specify that buffer in both the start and end routines. By specifying the buffer that receives data in the end routine, we can avoid many (though not all) of the problems described in section 2.5.2 (A Fortran Problem with Register Optimization).
- An MPI collective routine that is not a two phase operation may be used at the same time on the same communicator. That is

```
MPI_Allreduce_start( ..., comm );
...
MPI_Barrier( comm );
...
MPI_Allreduce_end( rbuf, comm );
```

- In a multithreaded implementation, any two-phase collective start and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads start a two-phase operation on the same communicator since only one two-phase operation can be active on a communicator at any time.)

  Discussion. There are other alternatives to the start/end options. Jim Cownie provided the following discussion

  Since at least one of the calls must be collective the options are

```
        Start           End

1)      collective      local
2)      local           collective
3)      collective      collective
```

  Option 1 gives you nothing beyond what you already have, so can be ignored.

  Option 2 is what Lloyd appears to be proposing (and the text says).

  Option 3 is what the text said at the last meeting.

  I actually prefer option 3 (both start and end are collective). It has the following benefits :-

  1) A legal implementation of these functions is to replace them with the existing blocking functions at start time, when all the arguments are known.

  2) It provides a way for the implementation only to allocate the resources required for these operations at the time the first such operation is performed in a communicator. (Resource allocation is assumed to require collective communication).

  3) (As a result of 2) It allows an easy extension to the draft which is to permit multiple outstanding such operations in a communicator.

  To expand on the possibility of multiple operations :-

  1) We would allow an arbitrary number of two-phase collective operations in a communicator.

  2) They must be strictly nested.

  3) They must occur in the same order in each member of the communicator.

  Because the implementation can block to allocate the necessary resources in the start operation, it need not allocate any resources to communicators as it creates them, rather it can allocate these (scarce) resources only to the communicators which require them. Since it is now dynamically allocating these resources it is a relatively simple extension to allow multiple such operations in a single communicator.

Note that this proposal does *not* include tags in the collective operations, these would only be required if we wanted to relax the ordering rule (rule 3) above. I see no reason to do that. (For instance Rolf's application example didn't need it). Adding tags would remove the possibility of implementing these operations using the existing, blocking collective operations.

In this version these operations really become hints to the implementation that the user would like things to happen asynchronously, however there is no requirement on the implementation to work that way.

(*End of discussion.*)

Specific routines

MPI_ALLREDUCE_START(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 (integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Allreduce_start(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_ALLREDUCE_START(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM,
             IERROR)¡type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

MPI_ALLREDUCE_END(recvbuf, comm)

OUT	recvbuf	starting address of receive buffer (choice)
IN	comm	communicator (handle)

```
int int MPI_Allreduce_end( void* recvbuf, MPI_Comm comm)
```

```
MPI_ALLREDUCE_END(RECVBUF, COMM, IERROR) ¡type> RECVBUF(*)
INTEGER COMM, IERROR
```

The arguments have the same meaning as for MPI_Allreduce.

MPI_ALLREDUCE_START begins a two phase operation that, when completed with MPI_ALLREDUCE_END, produces the result as defined for MPI_ALLREDUCE.

Advice to users. This is deliberately weaker than "same as MPI_ALLREDUCE", to allow for differing orders of operations which may produce different results. For example, when performing the common operation of MPI_SUM on MPI_DOUBLE data, the exact return value depends on the order of operation, since floating point arithmetic is not associative. (*End of advice to users.*)

MPI_BCAST_START(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

```
int int MPI_Bcast_start(void* buffer, int count, MPI_Datatype datatype, int
              root, MPI_Comm comm )
```

```
MPI_BCAST_START(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)¡type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_BCAST_END(buffer, comm)

INOUT	buffer	starting address of buffer (choice)
IN	comm	communicator (handle)

```
int int MPI_Bcast_end(void* buffer, MPI_Comm comm )
```

```
MPI_BCAST_END(BUFFER, COMM, IERROR)¡type> BUFFER(*)
INTEGER COMM, IERROR
```

The arguments have the same meaning as for MPI_Bcast.

MPI_BCAST_START begins a two phase operation that, when completed with MPI_BCAST_END, produces the result as defined for MPI_BCAST.

Additional functions under consideration:

```
MPI_WIN_BARRIER_START
MPI_WIN_BARRIER_END
MPI_REDUCE_START
MPI_REDUCE_END
MPI_BARRIER_START
MPI_BARRIER_END
```

6.2.2 Examples using MPI_START_ALLREDUCE

The following example shows how a two-phase collective operation is used to avoid unnecessary synchronization in a

```
      call MPI_ALLREDUCE_START(teilresl1, resl1, 1, REALx, MPI_SUM,
     .                          mpi_comm_world_a, info)
      call MPI_ALLREDUCE_START(teilresl0, resl0, 1, REALx, MPI_MAX,
     .                          mpi_comm_world_b, info)

c...solv the equationsystem
       call parallelsolver( rm000, rmf00, rmb00, rm0f0, rm0b0, rm00f,
```

```
     .                     rm00b, dq, rhs, nsubit, nhofhausit )

c...computation of the CFL-Number with resl1
       call MPI_ALLREDUCE_END(mpi_comm_world_a, resl1)
       ...
c...output of all residuals
       call MPI_ALLREDUCE_END(mpi_comm_world_b, resl0)
       ...
```

6.2.3 Proposed modification

Tags are added to two-phase collective operations in order to permit multiple two-phase operations per communicator.

Matching is accomplished in two-phase operations by communicator, type-of-operation, and user tag.

Wildcards are prohibited in the collective tags.

Advice to users. For systems that implement collective on top of point-to-point, this poses no special penalty. For systems that propose to use hardware for collective, I suspect there can be objections, but these same objections apply to overlapping groups posing non-blocking collectives as well, or any case where the special hardware must be multiplexed between activities.

The space of tags may be more proscribed than for point-to-point, in order to allow implementations to encode other needed bits involving the type of operation. (*End of advice to users.*)

Rationale. Tags were dropped from non-blocking collectives a while ago, mostly because Marc argued that this was an asymmetry from blocking collectives. We have agreed to have a totally different non-blocking strategy for collectives now, which is also unanalogous to the request approach of non-blocking point-to-point. Hence, the compelling use of "symmetry" to eliminate tags is no longer compelling at all.

Tags also solve the problem of multithreaded operations using collectives, inasmuch as they permit the threaded program to assert the ownership of on-going non-blocking collectives per thread.

By adding tags to the two-phase collectives, we permit threading, and multiple uses per communicator in single threaded programs. (*End of rationale.*)

Advice to users. Use unique tags in two-phase collective communication operations when multiple operations per communicator are needed, or when writing multi-threaded programs that share a communicator.

The tags need only be unique for each type of operation used. That is, tag=5 for reduce, and tag=5 for bcast will not interfere, when posed in two-phase form. (*End of advice to users.*)

Discussion: Collective tag ranges might be smaller than for point-to-point. An appropriate environmental inquiry function would be needed.

Comments: discussed on mailing list; implementors note that this eliminates some obvious optimizations (like a single area per communicator).

6.3 Intercommunicator Constructors

Current Status: Passed once

The current MPI interface provides only two intercommunicator construction routines ([3]):

- MPI_INTERCOMM_CREATE, creates an intercommunicator from two intracommunicators,
- MPI_COMM_DUP, duplicates an existing intercommunicator (or intracommunicator).

The other communicator constructors, MPI_COMM_CREATE and MPI_COMM_SPLIT, currently apply only to intracommunicators. These operations in fact have well-defined semantics for intercommunicators.

In the following discussions, the two groups in an intercommunicator are called the *left* and *right* groups. A process in an intercommunicator is a member of either the left or the right group. From the point of view of that process, the group that the process is a member of is called the *local* group; the other group (relative to that process) is the *remote* group. The left and right group labels give us a way two describe the two groups in an intercommunicator that is not relative to any particular process (as the local and remote groups are).

In addition, the specification of collective operations (Section 4.1 [1]) requires that all collective routines are called with matching arguments. For the intercommunicator extensions, this is weakened to matching for all members of the same local group.

MPI_COMM_CREATE(comm_in, group, comm_out)

IN	comm_in	original communicator
IN	group	group of process to be in new communicator
OUT	comm_out	new communicator

The language bindings are identical to MPI-1, so are omitted here.
If comm_in is an intercommunicator, then the output communicator is also an intercommunicator where the local group consists only of those processes contained in group (see figure 6.1). The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of comm_out. If the group does not specify at least one process in the local group of the intercommunicator, or if the calling process is not included in the group, MPI_COMM_NULL is returned.

Discussion: In the case where either the left or right group is *not* empty, why not an intercommunicator with MPI_GROUP_EMPTY? Does anyone remember the rationale? Is it just useless?

Example The following example illustrates how the first node in the "left" side of an intercommunicator could be joined with all members on the "right" side of an intercommunicator to form a new intercommunicator.

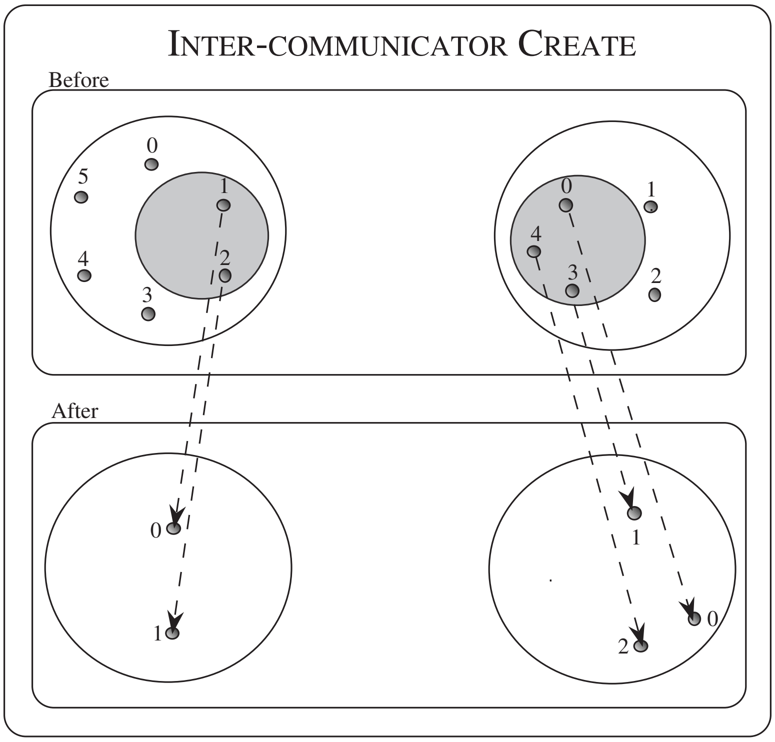

Figure 6.1: Intercommunicator create using MPI_COMM_CREATE() extended to intercommunicators.

```
MPI_Comm  inter_comm, new_inter_comm;
MPI_Group local_group, group;
int       rank = 0; /* rank on left side to include in new inter-comm */

/* Construct the original intercommunicator: "inter_comm" */
...

/* Construct the group of processes to be in new intercommunicator */
if (/* I'm on the left side of the intercommunicator */) {
  MPI_Comm_group ( inter_comm, &local_group );
  MPI_Group_incl ( local_group, 1, &rank, &group );
  MPI_Group_free ( &local_group );
}
else
  MPI_Comm_group ( inter_comm, &group );

MPI_Comm_create ( inter_comm, group, &new_inter_comm );
MPI_Group_free( &group );
```

MPI_COMM_SPLIT(comm_in, color, key, comm_out)

IN	comm_in	original communicator
IN	color	color of subset assignment
IN	key	color of rank assignment
OUT	comm_out	new communicator

The language bindings are identical to MPI-1, so are suppressed here.

The result of split on an intercommunicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see figure 6.2). For those colors that are specified only on one side of the intercommunicator but not the other, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned to those processes that specify MPI_UNDEFINED as the color.

Discussion: In the case where either the left or right group is *not* empty, why not an intercommunicator with MPI_GROUP_EMPTY? Does anyone remember the rationale? Is it just useless?

Example (Parallel client-server model) The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

```
/* Client code */
MPI_Comm  multiple_server_comm;
MPI_Comm  single_server_comm;
int       color, rank, num_servers;

/* Create intercommunicator with clients and servers:
   multiple_server_comm */
...

/* Find out the number of servers available */
MPI_Comm_remote_size ( multiple_server_comm, &num_servers );

/* Determine my color */
MPI_Comm_rank ( multiple_server_comm, &rank );
color = rank % num_servers;

/* Split the intercommunicator */
MPI_Comm_split ( multiple_server_comm, color, rank,
                 &single_server_comm );
```

The following is the corresponding server code:

```
/* Server code */
MPI_Comm  multiple_client_comm;
MPI_Comm  single_server_comm;
```

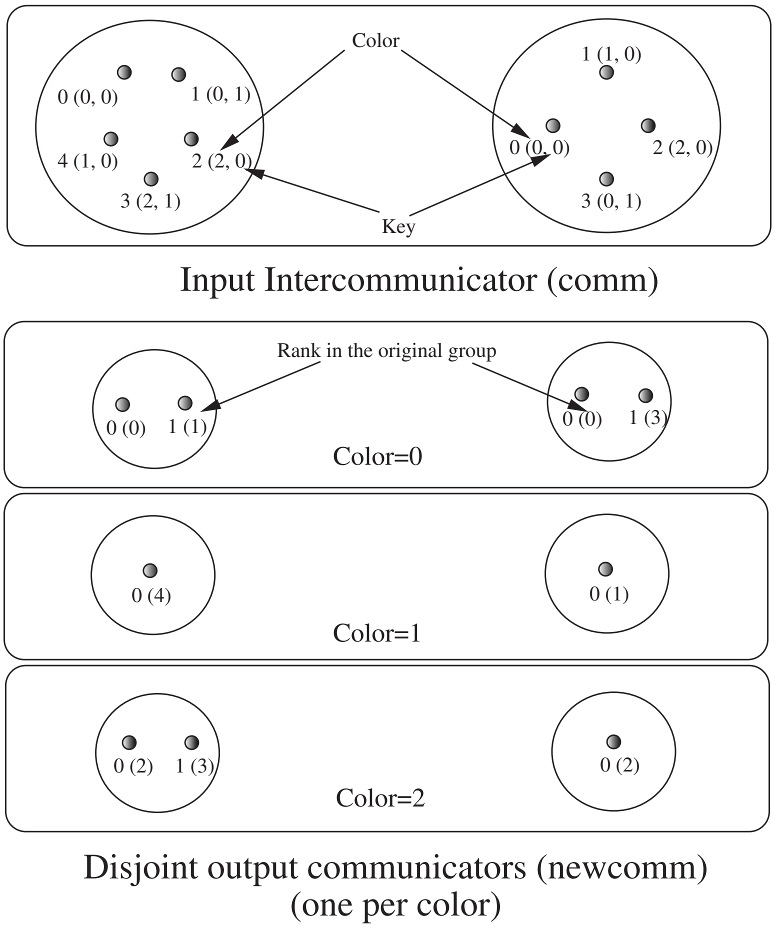

Figure 6.2: Intercommunicator construction achieved by splitting an existing intercommunicator with MPI_COMM_SPLIT() extended to intercommunicators.

```
int       rank;

/* Create intercommunicator with clients and servers:
   multiple_client_comm */
...

/* Split the intercommunicator for a single server per group
   of clients */
MPI_Comm_rank ( multiple_client_comm, &rank );
MPI_Comm_split ( multiple_client_comm, rank, 0,
                 &single_server_comm );
```

6.4 Extended Collective Operations

6.4.1 Intercommunicator Collective Operations

Current Status: Passed once

Discussion: Vote passed only intra-communicator inplace but did not preclude adding back in things that are missing, such as allreduce. The same applies to inter-communicator in-place. Bill Gropp and Marc Snir respectively have promised text.

In the MPI-1 standard, collective operations only apply to intracommunicators [1, Section 4.2]; however, most MPI collective operations can be generalized to intercommunicators. To understand how MPI can be extended, we can view most MPI intracommunicator collective operations into one of the following categories (see, for instance, [3]):

All-To-All All processes contribute to the result. All processes receive the result.

- MPI_Allgather, MPI_Allgatherv
- MPI_Alltoall, MPI_Alltoallv
- MPI_Allreduce, MPI_Reduce_scatter

All-To-One All processes contribute to the result. One process receives the result.

- MPI_Gather, MPI_Gatherv
- MPI_Reduce

One-To-All One process contributes to the result. All processes receive the result.

- MPI_Bcast
- MPI_Scatter, MPI_Scatterv

Other Collective operations that do not fit into one of the above categories.

- MPI_Scan
- MPI_Barrier

The MPI_Barrier operation does not fit into this classification since no data is being moved (other than the implicit fact that a barrier has been called). The data movement pattern of MPI_Scan does not fit this taxonomy.

Intercommunicators contain a "left" and a "right" group. Collective operations can be described in terms of these two groups. For example, an all-to-all MPI_Allgather operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 6.3). As another example, a one-to-all MPI_Bcast operation sends data from one member of one group to all members of the other group For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For symmetry, each such operation is described in two phases, so that it has a full-duplex behavior.

For MPI-2, the following intracommuniator collective operations also apply to intercommunicators:

- MPI_BCAST,
- MPI_GATHER, MPI_GATHERV,
- MPI_SCATTER, MPI_SCATTERV,
- MPI_ALLGATHER, MPI_ALLGATHERV,
- MPI_ALLTOALL, MPI_ALLTOALLV, (MPI_ALLTOALLW)
- MPI_REDUCE, MPI_ALLREDUCE,
- MPI_REDUCE_SCATTER,
- MPI_BARRIER.

These functions use exactly the same argument list as their MPI-1 counterparts and also work on intracommunicators, as expected. No new language bindings are consequently needed.

6.4.2 Operations that Move Data

Two additions are made to many collective communication calls:

- Collective communication can occur "in place" for intracommunicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument.
- Collective communication applies to intercommunicators. If the operation is rooted (e.g., broadcast, gather, scatter), then the transfer is unidirectional. The direction of the transfer is indicated by a special value of the root argument. In this case, only the root process in the group containing the root process calls the collective routine; all processes in the other group (the group that is the remote group relative to the root process) must call the collective routine. If the operation is unrooted (e.g., alltoall) then the transfer is bidirectional.

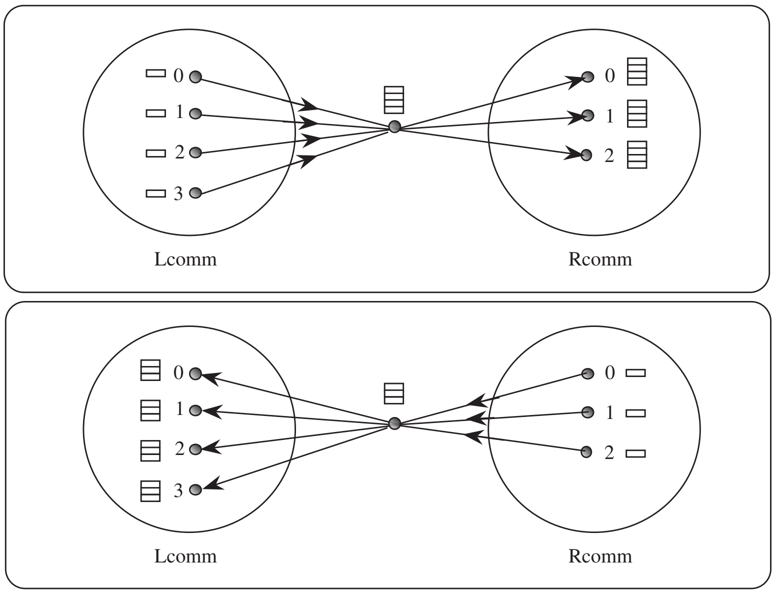

Figure 6.3: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

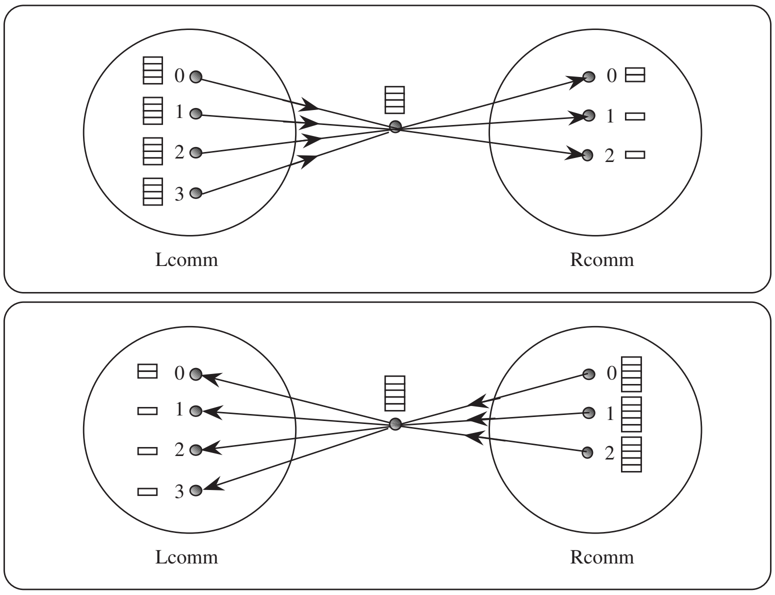

Figure 6.4: Intercommunicator reduce-scatter. The focus of data to one process is logical, not mandated by the semantics.

Discussion: Rather than having the other processes on the root's side not call the collective routine, we could use MPI_NOT_ROOT as the special root value for the other processes in the same local group as the root.

Rationale. The in-place operations are provided to reduce unnecessary memory motion by both the MPI implementation and by the user. Note that while the simple check of testing whether the send and receive buffers have the same address will work for some cases (e.g., MPI_ALLREDUCE), they are inadequate in others (e.g., MPI_GATHER, with root not equal to zero). Further, Fortran explicit prohibits aliasing of arguments; the approach of using a special value to denote in-place operation eliminates that difficulty. (*End of rationale.*)

Rationale. Rooted operations are unidirectional by nature, and there is a clear way of specifying direction. Non-rooted operations, such as all-to-all, will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

In the following, the definitions of the collective routines are provided to enhance the readability and understanding of the associated text. They do not change the definitions of the argument lists from MPI-1. The language bindings for these routines are unchanged from MPI-1, and are not repeated here. The text provided for each routine is appended to the definition of the routine in MPI-1.

Broadcast

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

The "in place" option not meaningful here.

If comm is an intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Data is broadcast from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

Gather

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (user's choice)
IN	sendcount	number of elements in send buffer (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 (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)

"In place" option is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the right place in the receive buffer

If comm is intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcounts	integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (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)

"In place" option is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the right place in the receive buffer

If comm is intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

Scatter

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 (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 (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

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

If comm is intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each processor
IN	displs	integer array (of length group size). Entry `i` specifies the displacement (relative to sendbuf from which to take the outgoing data to process `i`
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

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

If comm is intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

"All" Forms and All-to-all

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 (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 (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

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

```
MPI_GATHER( MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, recvbuf, recvcount,
                                  recvtype, root, comm )
```

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

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

The in-place option is illegal for intercommunicators.

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

(*End of advice to users.*)

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array (of length group size) containing the number of elements that are received from each process
IN	displs	integer array (of length group size). Entry `i` specifies the displacement (relative to recvbuf) at which to place the incoming data from process `i`
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

The “in place” option is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer. Specifically, the outcome of a call to MPI_ALLGATHER in the in place case is as if all processes executed n calls to

```
MPI_GATHERV( MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, recvbuf, recvcounts,
                               displs, recvtype, root, comm )
```

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

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

The in-place option is illegal for intercommunicators.

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 (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 (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

No in-place 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-the receive buffer of process j in group B, and vice versa.

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

(*End of advice to users.*)

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	integer array equal to the group size specifying the number of elements to send to each processor
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process j
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array equal to the group size specifying the number of elements that can be received from each processor
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

No in-place 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-the receive buffer of process j in group B, and vice versa.

6.4.3 Reductions

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 (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)

The “in place” option is specified by passing the value MPI_IN_PLACE to the argument sendbuf at the root. In such case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If comm is intercommunicator, then the call involves a root process in one group (group A) and all processes in the other group (group B). All processes in 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. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

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 (integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

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

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

Discussion: We don't have the option of achieving unidirectional communication with inter-communicators, with the current definition. The same holds for reduce-scatter and scan.

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

The in place option is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the top of the receive buffer. Note that the area occupied by the input data may be either longer or shorter than the data filled 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 scattered among processes in group B, and vice versa. Within each group, all processes provide the same recvcounts argument, and the sum of the recvcounts entries should be the same for the two groups.

Discussion: The last restriction is needed so that the length of the send buffer be determined by the sum of the local recvcounts entries. Otherwise, a communication is needed to figure out how many elements are reduced.

Other operations

MPI_BARRIER(comm)

IN	comm	communicator (handle)

```
int MPI_Barrier(MPI_Comm comm)
```

For MPI-2, comm may be an intercommunicator or an intracommunicator.

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 (integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

The "in place" option 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 illegal for intercommunicators.

6.4.4 Generalized All-to-all Function

Current Status: Passed once

One of the basic data movement operations needed in parallel signal processing is the 2-D matrix transpose. In this section, we show that the existing MPI function that support 2-D matrix transpose is not sufficiently general. To address this deficiency, we propose a new collective function. with "W" appended to its name. We specifically describe the MPI_ALLTOALLW function here.

Consider a 2-D matrix, which is initially distributed by striping rows over the available processors. After a transpose operation, it will be distributed by striping columns over the available processors. This basic data movement can be acheived by using the function MPI_ALLTOALL.

However, in addition to moving data between processors, the transpose operation should also change the layout of the matrix from row major order to column major order (*i.e.*, from elements in a row being contiguous in memory, to elements in a column being contiguous in memory). Thus in addition to moving data between processors, MPI_ALLTOALL should also transpose each "sub-cube."

For matrices whose dimensions are an even multiple of the number of processes, this sub-cube tranpose can easily be performed using the appropriate send and receive datatypes in the call to MPI_ALLTOALL. For example, if the source datatype steps throught the subarray elements row first, the destination datatype should step throught the subarray elements column first.

However, for matrices whose dimensions are not an even multiple of the number of processes a problem occurs. In this case, many of the sub-cubes will have the same dimensions, but one or more sub-cubes will have a different dimension. For example, a 18x18 matrix distributed over 4 processors would contain the following sub-cubes:

```
5x5 5x5 4x5 4x5
5x5 5x5 4x5 4x5
5x4 5x4 4x4 4x4
5x4 5x4 4x4 4x4
```

Even using the additional flexibility of MPI_ALLTOALLV it is impossible correctly to perform this transpose operation in one step. Consider the specification of the source datatype for this operation. The first two sub-cubes of the first row require a vector type, containing five blocks each containing five elements (with a stride between blocks of 18). In contrast, the vector type for the second two sub-cubes should only contain four elements. Unfortunately, MPI_ALLTOALLV does not allow an array of datatypes to be specified for the source buffer (only sizes and displacements). Thus MPI_ALLTOALLV is not sufficiently general to handle matrix tranposition.

Note: The destination datatype for the above operation can be correctly specified to MPI_ALLTOALLV. The first two sub-cubes of the first column require a vector type containing five blocks each containing one element. If the stride of this datatype is set to one (using MPI_TYPE_STRUCT) the first two sub-cubes can be specified by using a count of five. The second two sub-cubes of the first column can use the same datatype, so long as they specify a count of four.

The following function is the most general form of `All-to-all`. Like MPI_TYPE_STRUCT, the most general type constructor, MPI_ALLTOALLW allows seperate specification of count, displacement and datatype. In addition, to allow maximum flexibility, the displacement of blocks within the send and receive buffers is specified in bytes.

> *Rationale.* The MPI_ALLTOALLW function generalizes several MPI functions by carefully selecting the input arguments. For example, by making all but one process have `sendcounts[i] = 0`, this achieves an `MPI_SCATTERW` function. (*End of rationale.*)

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

IN	sendbuf	starting address of send buffer
IN	sendcounts	integer array equal to the group size specifying the number of elements to send to each processor
IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j
IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j
OUT	recvbuf	address of receive buffer
IN	recvcounts	integer array equal to the group size specifying the number of elements that can be received from each processor
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i
IN	comm	communicator (handle)

```
int MPI_Alltoallw(void* sendbuf, int* sendcounts, int* sdispls,
             MPI_Datatype* sendtypes, void* recvbuf, int* recvcounts,
             int* rdispls, MPI_Datatype* recvtypes, MPI_Comm comm)
```

```
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPES, COMM)
    INTEGER SENDBUF, SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVBUF,
    RECVBUFS(*), RECVCOUNTS(*), RDISPLS(*), RECVTYPES(*), COMM, IERR
```

```
int MPI::Alltoallw(void* sendbuf, int sendcounts[], const
             int sdispls[],const MPI::Datatype sendtypes[], void* recvbuf,
             int recvcounts[], const int rdispls[],const
             MPI::Datatype recvtypes[], MPI::Comm comm)
```

The j-th block sent from process `i` is received by process `j` and is placed in the `i`-th block of **recvbuf**. These blocks need not all have the same size.

The type signature associated with **sendcounts[j]**, **sendtypes[j]** at process `i` must be equal to the type signature associated with **recvcounts[i]**, **recvtypes[i]** at process `j`. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{sdispls}[i], \texttt{sendcounts}[i], \texttt{sendtypes}[i], \texttt{i}, ...),$$

and received a message from every other process with a call to

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{rdispls}[i], \texttt{recvcounts}[i], \texttt{recvtypes}[i], \texttt{i}, ...).$$

All arguments on all processes are significant. The argument comm must have describe the same communicator on all processes.

⊤ (Jan) ⊥ (Jan) A clarification is that no in-place support is offered.

6.5 Exclusive Scan

Current Status: Passed once

MPI-1 provides an inclusive scan operation. The exclusive scan is described here.

MPI_EXSCAN(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 (integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int int MPI_Exscan(void* sendbuf, void* recvbuf, int count, MPI_Datatype
              datatype, MPI_Op op, MPI_Comm comm )
```

```
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) ¡type>
              SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

MPI_EXSCAN is used to perform a prefix reduction on data distributed across the group. The value in recvbuf on the process with rank 0 is undefined. The value in recvbuf on the process with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes with rank $i > 1$, 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 $0, \ldots, i-1$ (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE.

Discussion: Do we guarentee that the MPI_Op is not called on the process with rank 1 (or zero)? Note that since the operation does not get any information on the rank of the calling process, it can't get any indication that it is to perform a different operation (copy rather than reduce).

Rationale. The exclusive scan is more general than the inclusive scan provided in MPI-1 as MPI_SCAN. Any inclusive scan operation can be achieved by using the exclusive scan and then locally combining the local contribution. Note that for non-invertable operations such as MPI_MAX, the exclusive scan can not be found with the inclusive scan.

The reason that MPI-1 choose the inclusive scan is that the definition of behavior on processes zero and one was thought to offer too many complexities in definition, particularly for user-defined operations. (*End of rationale.*)

Bibliography

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. To appear in the International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.

[2] Perry Partow and Dennis Cottel. Scalable Programming Environment. Technical Report 1672, Naval Command Control and Ocean Surveillance Center (NRAD), September 1994.

[3] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical report, Mississippi State University — Dept. of Computer Science, April 1994. Draft version.