PROCEEDINGS OF THE SIXTH SIAM CONFERENCE ON
PARALLEL
PROCESSING FOR
Scientific
Computing
Volume II

Edited by Richard F. Sincovec
David E. Keyes
Michael R. Leuze
Linda R. Petzold
Daniel A. Reed
Designing Efficient, Scalable, and Portable Collective Communication Libraries

Vasanth Bala* Shlomo Kipnis* Larry Rudolph† Marc Snir*

Abstract

We describe methods for designing efficient, scalable, and portable collective communication primitives for multiprocessor systems. To achieve efficiency, we introduce the technique of split-phase communication that allows the programmer to recover much of the communication overhead costs and allows the compiler to optimize the communication operations. To support scalability, we use Process Groups and Process Channels that enable specifying group and neighborhood communication patterns among collections of processes in a simple and scalable manner. Finally, to facilitate portability, we provide only general mechanisms for communication operations, without constraining our methodology to specific implementations.

1 Introduction

Collective communication, as an alternative to point-to-point send/receive operations, is gaining increased acceptance for specifying communication between processes in multiprocessor systems. Examples of collective communication primitives include broadcast, reduce, combine, shift, scatter, gather, transpose, prefix, barrier-sync, and other operations that require simultaneous information interchange between a collection of processes. Although collective communication operations are usually implemented using point-to-point send/receive operations, they offer the programmer a more intuitive "application-level" interface and ease the design and debugging of programs. Recent studies [5] have also indicated that there are numerous scientific applications in which the need for collective communication arises quite naturally.

Existing communication libraries for multiprocessor systems emphasize one of three important goals: efficiency, scalability, and portability. Vendor-specific libraries, like the NCUBE library [11], the CM-5 library [14], and the iPSC/860 library [16], focus on efficiency as the primary goal. The corresponding multiprocessors use communication controllers that can offload part of the communication processing from the main processor. Some machines are also designed with hardware support to enhance the efficiency of specific kinds of collective communication. Examples include the special networks on the Fujitsu AP1000 [9] and the CM-5 [10] machines for performing fast barrier synchronization and reduction operations. Of vendor-specific libraries originated from users Express [15] emphasizes the other goals acceptance of parallel performance, that supports between different architectures for designing efficient.

To achieve efficiency allows the programmer compiler to optimize Process Groups and Process Channels communication pattern without making explicit PIDs. To facilitate port definition of the difference allowing actual implementation.

The remainder of this technique for increasing section discusses the use of Process applications. Section 4 of our proposed design.

2 Efficiency Through Split-Phase Communication

A collective communication that include parameters who to communicate with, completion, and error return routine, which the program operation. For example in Figure 1.

This implementation of communication: all and each process return communication is finished (even [5]. The broadcast app returns from it until all.

However, such a is wish to participate in the (their buf is set to N) blocking within the box "setup overhead" for protection checks, and is.

The first drawback communication, can be interesting processes act
Designing Collective Communication Libraries

reduction operations. Efficient use of the communication hardware is an important goal
of vendor-specific libraries. On the other hand, collective communication libraries that
originated from users of parallel computers, like PICL [6], PARMACS [7], Zipcode [13], and
Express [15] emphasize portability across several machines as the primary goal.

When designing a library for one of these goals — efficiency, scalability, and portability
— the other goals are usually sacrificed to some extent. However, with the increased
acceptance of parallel computers, grows the need for environments that provide good
performance, that support the scalability of applications, and that facilitate code portability
between different architectures. In this paper, we address these issues and describe methods
for designing efficient, scalable, and portable collective communication primitives.

To achieve efficiency, we introduce the technique of split-phase communication that
allows the programmer to recover much of the communication overhead costs and allows
the compiler to optimize the communication operations. To support scalability we use
Process Groups and Process Channels that enable specifying group and neighborhood
communication patterns among subsets of processes in a simple and scalable manner,
without making explicit references to the number of processes involved or their respective
PIDs. To facilitate portability of the primitives, our methodology only lays out the semantic
definition of the different components of a collective communication operation, thereby
allowing actual implementations of these components to be tailored to specific machines.

The remainder of this paper is organized as follows. Section 2 describes the split-phase
technique for increasing the efficiency of collective communication operations. Section 3
discusses the use of Process Groups and Process Channels to support the scalability of
applications. Section 4 concludes with an overview and some comments on the portability
of our proposed design.

2 Efficiency Through Split-Phase Communication

A collective communication operation typically consists of a complex sequence of actions
that include parameter checking, possible buffer allocation, computation for determining
who to communicate with, initiation of the communication, waiting for the communication
completion, and error reporting. All of these actions are usually encapsulated within a single
routine, which the programmer invokes in order to perform the collective communication
operation. For example, a typical implementation of the broadcast operation is outlined
in Figure 1.

This implementation example illustrates the "loosely synchronous" paradigm of collective
communication: all the processes invoke the same broadcast routine asynchronously,
and each process returns from the call as soon as its participation in the collective commu-
nication is finished (even though the entire broadcast operation may still be in progress)
[5]. The broadcast appears to be an atomic operation for each process, since none of them
returns from it until all processes have started the operation.

However, such an implementation has three drawbacks. First, not all processes may
wish to participate in the communication; the call could be redundant for some processes
(their buf is set to NULL). Second, processes may waste a significant amount of time
blocking within the broadcast routine, waiting for messages to arrive. And third, the
"setup overhead" for making such a call is typically very high, due in part to the error and
protection checks, and in part to the creation of the communication structures and buffers.

The first drawback, the redundant involvement of noninterested processes in the
communication, can be handled by designing the communication operation so that only
interested processes actually participate in it while other processes skip it. We defer
int broadcast (orig, buf, nbytes)
PID orig; /* originator pid */
long buf; /* ptr to data buffer */
int nbytes; /* how many bytes to send/recv */
{
    /* Setup Phase (setup the communication structures) */
    Check parameter list (legality of "orig", "buf", etc...);
    Set up broadcast tree structure (determine my position in tree);

    /* Initiation Phase (initiate my participation) */
    /* Part 1: synchronize with everyone else */
    if (leaf(mypid)) /* I am a leaf in the broadcast tree*/
        Send message to my parent in the broadcast tree;
    else /* I have children in the broadcast tree */
        Block waiting for messages; /* unpredictable wait time! */
        Receive messages from all children;
        if (mypid != orig) /* I have a parent in the broadcast tree */
            Send message to parent;

    /* Part 2: perform the communication */
    if (mypid = orig) /* I am the originator */
        Send data in buf, to my children in the broadcast tree;
    else /* I am a recipient */
        Block waiting for a broadcast msg; /* unpredictable wait time! */

    /* Wait Phase (wait for my participation to complete) */
    if (mypid != orig) /* I am a recipient */
        Receive msg from my parent in the broadcast tree;
        Forward msg to my children in the broadcast tree;
        Truncate msg data to nbytes and put in buf, and report errors;
}

FIG. 1. Example of a loosely synchronous broadcast algorithm.

discussion of this issue to Section 3.

The second performance hit is caused by communication latencies, which are governed
by several factors. An initial delay at each process is caused by the time the process
reaches the broadcast call. A "load-balanced" program, where all processes reach a loosely
synchronous call around the same time, can reduce this penalty. However, even if all
processes reach the broadcast call at the same time, some processes will wait additional time
for messages to reach them over the broadcast tree. This propagation time is unpredictable;
it depends on software and hardware delays in the communication subsystem and on the
type of communication algorithm used (ring-broadcast, tree-broadcast, degree of broadcast
tree, etc...).

The third drawback, the high setup overhead cost, presents another problem. The
setup overhead tends to dominate the communication cost when message sizes are small
(1Kbyte or less on current machines). For this reason, users seldom place communication
calls within compute-intensive loops. Such calls are usually done outside of loops by
combining successive data items into a single collective communication call ("vectorizing" in
the compiler jargon), so as to result in longer messages. However, this is not an ideal solution
for two reasons. (i) A sending process has to generate all the data before communication
starts, and a receiving process has to receive all the data before it can start its computation.

This reduces the opportunity to overlap computation and communication. For example,
if data were pipelined, communication would occur if data were pipelined. (ii) Pipelining allows
improving communication performance.

The second and third drawbacks suggest that split-phase design strategies can be used to
efficiently implement broadcast. The communication operation in the second program (automatically
split-phase) can be invoked as a subroutine: (1) The setup broadcast routine includes error and
progress checking. The information about the broadcast call in the parameter list is
automatically checked by the broadcast routine does not do a physics check as
in the buffer is ready for the do broadcast routine.

The advantage of using a split-phase design is that the entire broadcast needs to be
involved in a computation loop. The setup broadcast routine is amortized over multiple
iterations, and even in the presence of communication costs.

do i = 1, 
   ...
   rc = broadcast(ori g, 
enddo

For the split-phase design, the loop overhead of a single loop iteration is smaller than
the loop overhead of a single loop iteration, (2) The broadcast operation is cheaper than the
communication between the processes involved in conventional broadcasting. For example,
communication is moved outside a loop.
This reduces the opportunity for overlapping communication and communication that could occur if data were pipelined in small incremental amounts as it is produced within the loop.

(ii) Pipelining allows the use of smaller communication buffers than vectorization, thus improving communication efficiency.

The second and the third drawbacks above can be alleviated to some extent by using a split-phase design strategy, similar to the one suggested by Chase, Bala, and Reeves [3] for efficient implementation of point-to-point communication. The basic idea is to decouple each communication operation into well-defined, distinct components. Each of these components can be invoked as a separate routine and can be moved around to different points in the program (automatically by the compiler or manually by an expert programmer), provided certain rules are followed to ensure the integrity of the underlying communication.

We demonstrate the split-phase strategy on the broadcast operation. For example, the routine:

```
rc = broadcast(orig, buf, nbytes);
```
can be split into the two components:

```
id = setup_broadcast(orig, buf, nbytes);
rc = do_broadcast(id);
```
The setup_broadcast routine does all the work of the “Setup Phase” (see Figure 1), which includes error and protection checks, buffer allocation, and setting up the broadcast tree. The information about processes’ position in the tree, as well as the information supplied in the parameter list is saved and a handle id to this information is returned. The setup routine does not do any communication. This allows it to be invoked before the data in the buffer is ready to be transmitted or to be overwritten with incoming data. The do_broadcast routine does the work involved in the “Communicate Phase” (see Figure 1).

The advantage of this decoupling can be appreciated by an example. Suppose that the same broadcast needs to be repeated several times in a loop. Such a situation arises often in stencil-oriented computations like Finite Element problems, where each process needs to communicate the same boundary data to a set of neighboring processes, several times within a loop. The setup_broadcast can be moved out of the loop, so that the setup overhead cost is amortized over multiple invocations of the same communication. This optimization works even in the presence of data dependences in the computation that may prevent moving the entire broadcast call outside the loop:

```
do i = 1, n
...  
rc = broadcast(...)
enddo
```
```
do i = 1, n
...  
rc = do_broadcast(id)
enddo
```
For the split-phase design to be profitable, the cost of invoking do_broadcast should be significantly lower than the cost of invoking the original broadcast call. If a given loop, the overhead of a do_broadcast call is small enough compared to the computation time per iteration, then the call can be left inside the loop. (This overhead refers to processor cycles consumed in invoking the communication, not to time spent in performing the actual message passing which is done off-processor by the communication subsystem hardware.) This approach may be profitable in situations where it is desirable to pipeline communication between processes. As mentioned earlier, due to the high setup overheads involved in conventional library designs, compilers will always attempt to “vectorize” data communicated inside a loop and move the entire communication outside the loop. The split-phase design offers an alternative option in which only the setup component of a communication is moved out of the loop.
Although this separation helps amortize the setup overhead over multiple invocations of the same collective communication call, it still does not address the communication latency problem. In order to achieve good performance, users should be allowed to cover this communication latency. To provide this capability, we further split the `do_broadcast` into the two components:

```
init_broadcast(id);
rc = wait_broadcast(id);
```

The routine `init_broadcast` starts a process's participation in the broadcast operation, and the routine `wait_broadcast` blocks the process until its participation in the broadcast is completed. The `init_broadcast` can be implemented using asynchronous communication calls, such as the Intel `isend` or `irecv` [16], which enables the invoking process to become available for further computation. The `wait_broadcast` can be implemented by a busy-wait on the communication semaphore.

Using this further splitting, some of the communication latency is coverable by performing computation between an `init_` and the corresponding `wait_`. The only restrictions are that the process should not modify the data buffers while the communication may be in progress, and that is should not initiate other communication which would violate the broadcast atomicity. It may now be possible to restructure our earlier example as follows:

```
id = setup_broadcast(...)
do i = 1, n
...       --->
rc = do_broadcast(id)         ... /* computation */
enddo
```

A routine `done_comm(id)` may also be needed to free storage allocated by a corresponding `setup_comm`. Upon return from a `done_comm(id)`, the handle `id` can be reused by the communication subsystem.

Thus, for any given collective communication ‘comm’, the split-phase design strategy is to allow the communication to be split into four suboperations:

(a) `id = setup_comm(...parameters...);` Does all the setup for the operation `comm` and returns a handle `id` that can be used to execute `comm`. No communication is involved.

(b) `init_comm(id):` Starts the execution of operation `comm`.

(c) `wait_comm(id):` Waits until operation `comm` has completed.

(d) `done_comm(id);` Free all resources associated with the handle `id`.

These four suboperations are invoked in a sequence of the form “setup [init wait] ... done”. These suboperations are provided to the programmer in three levels (see Figure 2). The highest level (Level 1) provides the most intuitive programming interface and is relatively easy to debug, but it may not offer very good performance. The lowest level (Level 3), on the other hand, offers the opportunity for getting good performance, but its programming interface is not very intuitive and debugging with it is much harder. The intermediate level (Level 2) offers the best compromise to the programmer, in terms of ease of use, ability to debug the communication, and performance. Level 3 is best suited for use by an optimizing compiler, which can split Level 2 calls in the code down to Level 3, and then move the components around to get better performance. Expert programmers and designers of optimized scientific subroutine packages may also want to program in Level 3. In fact, the Level 3 routines `init_comm` and `wait_comm` can be compiled inline, especially if they are implemented using simple and compact code (see, for example, [3]). This will further increase efficiency by increasing interface with existing systems.

**LEVEL 1 (Compiler):**
```
rc = comm(...);
```

**LEVEL 2 (Programmer):**
```
rc = do_comm(id);
```

**LEVEL 3 (Compiler):**
```
rc = wait_comm(id);
```

**Fig. 2.**

Another source of communication latency is the communication between different processes. By splitting the broadcast algorithm (as shown in Figure 1, one process completes its participation in the tree and has sent data to one more, but the `wait_comm` is not yet complete.

The same choice as with the `do_broadcast` algorithm (as shown in Figure 1, the broadcast tree is completed its participation in the tree and has sent data to one more, but the `wait_comm` is not yet complete). We thus have six different broadcast algorithms:

- A communication operation where `wait_comm` can be used to synchronize the sender and a receiver.
- A communication operation where `init_comm` and `wait_comm` can be used to synchronize the sender and a receiver.
- A communication operation where `init_comm` and `wait_comm` can be used to synchronize the sender and a receiver.

We thus have six different broadcast algorithms:

- A communication operation where `wait_comm` can be used to synchronize the sender and a receiver.
- A communication operation where `init_comm` and `wait_comm` can be used to synchronize the sender and a receiver.
- A communication operation where `init_comm` and `wait_comm` can be used to synchronize the sender and a receiver.

**3 Scalability**

An important factor in the design of scalable communication packages is the increased number of processors available. Although the latter is not the primary focus of this paper, scalability with increased communication performance and a wide range of available processors is important. Groups and Process Channels are treated as single objects. A Process Channel is a...
Another source of optimization is the decoupling of communication operations on different processes. For point-to-point communication, one can replace synchronous communication, where a sender blocks until a matching receive occurs, by asynchronous communication, where a sender completes its call as soon as its participation in the communication ended (i.e., as soon as data has been copied out of its memory). This avoids busy waiting by the sender, but the communication does not appear atomic anymore.

The same choice applies to collective communications. Rather than using a two-step broadcast algorithm (ascending to synchronize, then descending to communicate data), as shown in Figure 1, one can use a one-step broadcast scheme (descend only). Each process completes its participation as soon as it has received data from its parent in the broadcast tree and has sent data to its children in the tree. The broadcast operation is not atomic anymore, but the waiting time is reduced.

A communication operation `comm` comes in two flavors: "atomic" or "synchronous", where `wait_comm` can return at a process only after all participating processes have entered `init_comm`; and "nonatomic" or "asynchronous", where `wait_comm` returns as soon as the local process has ended its participation in the communication. Atomic broadcast is an intuitive, safer programming interface. It can be replaced by a nonatomic broadcast by a compiler or an expert programmer, in situations where no races can occur.

We thus have six different types of broadcast (three levels and two flavors). Notice that a broadcast in a set of two processes is analogous to a point-to-point communication between a sender and a receiver. Most of the six corresponding point-to-point communication operations appear in existing communication libraries.

### 3 Scalability Through Process Groups And Process Channels

An important factor in designing a parallel program is ensuring *scalable performance* with increased number of processors. In this paper, we focus only on the scalability of the *communication* performance and not on the scalability of the computational performance. Although the latter is an equally important issue, it is not relevant to the topic of this paper.

Scalability with increased number of processors can be achieved if the specification of communication can be done in a manner that is fairly independent of the actual number of processors available to the application. At the same time, the implementation of the communication routines must be such that their performance is roughly uniform over a wide range of number of processors. We propose two mechanisms to do this: Process Groups and Process Channels. A Process Group is a collection of processes that may be treated as a single object for purposes of specifying collective communication among them. A Process Channel is a collection of virtual connections among processes, which is treated as

---

**Figure 2.** The split-phase design of a collective communication routine `comm`.
a single object for the purpose of specifying neighborhood communication among collections of processes.

3.1 Process Groups

A Process Group is a logical set of processes that has a system-wide unique name and that can be dynamically defined at run-time. Each Process Group is identified in an application program by a unique Process Group Identifier (PGID). At the initialization of an application, each process belongs to the predefined Process Groups ALL, representing the set of all processes assigned to the application. New logical Process Groups can be created by partitioning an existing group, with PGID = G, as follows. Every process in the group G supplies an integer value myval, and all the processes that supply the same value are made members of the same new logical Process subGroup. Processes that are not interested in being included in any of the newly formed subgroups can supply the special value DON'T CARE.

The partitioning of an existing Process Group can be performed by the partition routine:

```c
pgid = partition(G, myval);
```

The call to partition must be made by every member of the existing Process Group G. The collective call to partition is atomic: no member of the existing group G returns from this call until all processes in group G have also made the call. This call results in the partitioning of the members of group G into as many subgroups as the number of distinct integer values supplied by the member processes. Upon completion, the invoking process is returned the PGID of the new subgroup to which it belongs. Any non-members of the group G that make this call will return immediately and will continue further execution.

For each new subgroup created by the partition call, a new system-wide unique identifier (PGID) must be created. In addition, a Process Group data structure must be maintained on each process that is a member of that group. This data structure can contain any relevant information about the group that may be useful in implementing efficient collective communication. An efficient technique for generating a system-wide unique PGID without performing any communication, as well as data structure choices and utility routines for Process Groups, are discussed by Bala and Kipnis in [2].

Process Groups created by one partition call are disjoint by definition. A process can, however, participate in several partition calls, and thus be a member of several different Process Groups. Each call to partition may correspond to partitioning according to a different criterion. Once created, a Process Group may be used several times for specifying various kinds of collective communication among the processes belonging to that group. Process Groups thus allow entire collections of processes to be identified and manipulated in a single call. Here we give two examples of using Process Groups, one for specifying parallelism between disjoint sets of processes, and the other for performing collective communication over the members of a Process Group.

The Process Group Identifier (PGID) can be used to selectively specify group-wide transactions over the members of that Process Group. For example, the following code:

```c
if (ismember(G, mypid))
    code-segment-1;
else
    code-segment-2;
```

causes all members of Process Group G to execute the first code segment, while all other processes skip to the second code segment, and execute it in parallel. Since Process Groups can be recursively subdivided, this style of programming can be used, say, for implementing a divide and conquer algorithm.

A group-wide transaction is specified by a call to the reduce function:

```c
reduce(G, rfunc, di)
```

This line of code specifies a reduction operation over all members of a Process Group G. For example, an application program might use this routine for performing collective communication. The function rfunc to specify the operation to be performed for each process. The final result of the reduction is then broadcast back to all members of the group G, using the broadcast function:

```c
broadcast(G, orig, r)
```

This routine implements a group-wide broadcast operation. For example, the value of an integer variable available or involved in a Process Group G can now be written as:

```c
if (cond)
    orig = broadcast(G, myval);
else(orig = ...);
```

In this scenario, only processes that make the callOrig = broadcast(G, myval) will perform the broadcast operation. Efficient implementations of the above group-wide broadcast and reduce functions are discussed by Bala and Kipnis in [2].

An advantage of Process Groups is that they can be used to specify parallelism between disjoint sets of processes. For example, if G1 and G2 are disjoint Process Groups, then this sequence of calls:

```c
broadcast(G1, orig, r);
broadcast(G2, orig, r);
```

where G1 and G2 are disjoint Process Groups, then this sequence can be simultaneously over all processes in G1 and G2.

The use of Process Groups allows many repeated uses of a single Process Group to be traced and monitored for debugging.

3.2 Process Clusters

Often, it is convenient to group processes into virtual topologies, such as clusters of processes, to implement a virtual topology, etc.
A group-wide transaction of particular interest is collective communication over all the members of a Process Group. For example, a group-wide reduce operation, which takes the PGID of a Process Group as a parameter, can be written as:

\[
\text{reduce}(G, \text{rfunc}, \text{data}, \text{result});
\]

This line of code could appear in the node program of every process involved in the application program, assuming that a Single Program Multiple Data (SPMD) programming model is used, but only the members of the Process Group \( G \) would participate in this communication. Such a routine implements a reduction, using a supplied associative function \( \text{rfunc} \) to combine data from the members of Process Group \( G \), and return the final result of the reduction to each member in the buffer \( \text{result} \). Similarly, a group-wide broadcast operation can be written as:

\[
\text{broadcast}(G, \text{orig}, \text{data});
\]

This routine implements a broadcast communication from a particular member of the Process Group \( G \), whose PID is \( \text{orig} \), to all other members of \( G \). An important point to note here is that since \( G \) can be dynamically defined, it may not be possible for each process to specify the PID of the originating process \( \text{orig} \). In a library such as Express, for example, the value \( \text{orig} \) must be known to every process in the group \( G \) before the group-wide broadcast can be initiated [15]. Our proposed design, however, allows collective communication algorithms to be used even when such global information is either not available or involves additional communication overhead to create. Thus, the code can now be written as:

\[
\begin{align*}
\text{if (condition)} & \quad \text{orig = mypid;} \\
\text{else} & \quad \text{orig = DONTCARE;} \\
& \quad \text{broadcast}(G, \text{orig}, \text{data});
\end{align*}
\]

In this scenario, only the originator process knows that it is the broadcast originator, while all other members of \( G \) know only that they are not the originators of this broadcast! Efficient implementation of collective communication, such as broadcast, for dynamically determined Process Groups are given in [1].

An advantage of using Process Groups is that processes that are not members of Process Group \( G \) do not participate in the group-specific communication and continue to the instruction following the call to broadcast. Moreover, a participating process can proceed past the call immediately after it has finished its part of the collective communication (even though the communication as a whole may still be ongoing). For example, consider the sequence of calls:

\[
\text{broadcast}(G1, \text{orig1}, \text{data1});
\]

\[
\text{broadcast}(G2, \text{orig2}, \text{data2});
\]

where \( G1 \) and \( G2 \) represent two Process Groups. If \( G1 \) and \( G2 \) are disjoint Process Groups, then this sequence of calls results in two parallel broadcast operations occurring simultaneously over the two disjoint collections of processes in groups \( G1 \) and \( G2 \).

The use of Process Groups allows to amortize the fairly expensive setup overhead over many repeated uses of the group structure. The use of system-wide unique PGID's also provides an elegant and scalable debugger interface, in which selective Process Groups can be traced and monitored.

### 3.2 Process Channels

Often, it is convenient to treat the processes in a Process Group as connected by some virtual topology, such as a 2-dimensional grid. Each process in this virtual topology
has a set of neighbors, to which it is connected by virtual communication channels. A global communication pattern, such as a left shift on a grid, may use only a subset of the available communication channels, by having each process mask out unused channels. Boundary processes, in particular, can have certain channels permanently masked out. At the application level, when a process communicates with its neighbors on a virtual topology (called a Process Channel), it does so without regard to which processes are its neighbors. The communication subsystem can figure out, based on the definition of the topology and the channel masks, which processes need to be communicated with.

A virtual topology can be imposed on the processes in a Process Group \( G \) by calling the routine:

\[
\text{chid} = \text{topology}(G, \text{outkeys}, \text{inkeys});
\]

As with other group related operations, the call to \text{topology} must be made by every member of \( G \). Each process supplies two lists: one list of distinct integer keys in array \text{outkeys}, and another list of distinct integer keys in array \text{inkeys}. The \text{outkeys} list of each process is matched with the \text{inkeys} lists of other processes in group \( G \), and a virtual communication channel is established between processes for which a match occurs. Lists of different processes need not be of the same length, but there should be a one-to-one correspondence between \text{outkeys} and \text{inkeys}. When every member of \( G \) has made the call, the lists of \text{outkeys} and \text{inkeys} supplied by the processes are matched up, and a virtual channel connection is made for each match. The list of virtual channel connections for each process is entered into a table associated with that process. At this point, storage for the communication buffers associated with each virtual channel are not yet allocated. Upon successful return from the \text{topology} call, the invoking process is returned an identifier that uniquely identifies the Process Channel network that has been established over the group \( G \). This table has additional fields for buffer pointers associated with each virtual connection, flow-control information, and lists of process addresses. These fields are initialized and used by the \text{gshift} communication operation described below.

Process Channels can be used for specifying communication patterns that are local to the defined topology, such as shifts on meshes, swaps on hypercubes, ascend/descend on trees, or flow through a directed graph. A process may belong to several Process Channels. Process Channels allow processes in a Process Group \( G \) to send and receive messages on a virtual topology \( C \). This is done using the \text{gshift} (“generalized shift”) operation:

\[
\text{gshift}(C, \text{outmask}, \text{outbuf}, \text{inmask}, \text{inbuf});
\]

This is a loosely synchronous call over the Process Group \( G \) that utilizes the associated Process Channel \( C \) to exchange data between every process and its neighbors in the Process Channel topology. The parameters \text{outmask} and \text{inmask} are bit-masks, the length of which is equal to the number of input and output channels attached to the invoking process. These masks specify which channels are active during the data exchange. The parameters \text{outbuf} and \text{inbuf} are arrays of pointers to input and output data buffers.

As with all other collective communication operations, the \text{gshift} operation may also be implemented as a split-phase communication. The different phases of the \text{gshift} operation include: setting up the communication resources, sending the data on the appropriate channels, receiving the data from the appropriate channels, and reclaiming the channel communication resources. Also, the \text{gshift} operation can have an atomic or a nonatomic implementation.

Using Process Channels simplifies the coordination of neighbor-type communication among processes. Moreover, since communication on a particular Process Channel is an operation that involves only the members of the associated Process Group, it can execute in parallel with operations in other Process Groups that have no common members.

4 Conclusion

The methods described in this paper improve the portability of collective communication operations by recognizing the advantages of specifying optimizations of communication patterns in terms of Process Channels, and provide an extensible method for collections of processes to communicate based on specifications of Process Groups.

In essence, each Process Channel forms a mini-application level virtual network over which both the efficiency and scalability of communication can be improved. To achieve scalability, the Process Channels must provide unified communication capabilities across the design enables isolation of the individual communication patterns. This can be done by isolating the communication patterns that take advantage of the structure of the network and the channel masks which are part of the virtual topology it is defined on.

In addition, this level of abstraction provides high-level ad-hoc and low-latency communication without the overhead due to calling the general library routines. Moreover, the setup cost of the channels and the additional burden of creating channels can be amortized over the lifetime of the virtual communication.

Throughout this paper, we have emphasized the ability to tolerate the high overhead of applying the communication patterns in parallel with other collective communication operations. The method described here can be used to specify communication patterns at a high-level model, while leaving the implementation details to the library routines.

In conclusion, Process Channels is a flexible and powerful abstraction for specifying communication patterns. In this paper, we have described how Process Channels can be used to improve the efficiency and scalability of communication operations, while providing a high-level interface for specifying communication patterns.

References


Designing Collective Communication Libraries

4 Conclusion

The methods described in this paper are aimed at increasing the efficiency, scalability, and portability of collective communication libraries for multiprocessor systems. We introduced the split-phase design technique, which can increase the efficiency of communication operations by recovering much of the communication overhead costs and enabling compiler optimizations of communication operation. We discussed the ideas of Process Groups and Process Channels, which provide abstractions for coordinating and communicating among collections of processes in a simple and scalable manner, without involving the explicit specifications of PID values or architectural details of the machine.

In essence, each of the methods and techniques discussed here has implications on both the efficiency, scalability, and portability of collective communication primitives. For instance, the split-phase design strategy also addresses some issues of program scalability. To achieve scalability with increasing problem sizes, collective communication operations must provide uniform performance over a wide range of message sizes. The split-phase design enables isolating the code responsible for the setup overhead from the rest of the communication. Such an optimization takes advantage of the temporal locality of communication patterns found in many scientific applications (see [8]). This lessens the impact of the setup overhead on communication performance when message sizes are small. Similarly, by splitting the actual communication into init_comm and wait_comm, the impact of the network latency on the communication performance is reduced for large message sizes. This is done by allowing part of the network latency to overlap with computation.

In addition, it may be possible to build communication networks with high-bandwidth and low-latency links between any pair of processors, in which the configuration time is expensive [12]. That is, in such networks, the time to setup the communication link has high overhead. Many proposed optical communication schemes have this feature. In such cases, the setup_comm routine of our design may also be used to configure the network before communicating data.

Throughout this paper, we have attempted to specify only the semantic interface of each of the communication operation, without describing low-level implementation details. We believe that each of these operations can be implemented in several ways, depending on particular architectural details and the communication subsystem capabilities. Our design emphasizes tolerating communication latencies rather than low latency communication, and the ability to amortize setup overhead rather than low setup overhead. This enhances the portability of application programs between different architectures, at the cost of some additional burden on the user. To offset this aspect, we offer three different levels of using the library routines: users can continue programming in the familiar loosely synchronous model, while leaving it to the compilers to perform the actual optimizations.

In conclusion, we believe that the ideas described in this paper are fundamental to communication libraries that are designed for both efficiency, scalability, and portability. Some of the ideas in this paper are incorporated into a proposal on a message-passing standard for distributed-memory systems (see [4]).

References


Using PVM and PVM-Express for Heterogeneous Systems

Jack Dongarra and Anthony Skjellum

Abstract. This paper describes the use of the PVM (Parallel Virtual Machine) and PVM-Express. One of the attractions of parallel processing is the ability to run PVM on a variety of multiprocessors as well as on large scale distributed multiprocessor interconnection networks using some of the same tools.

Several computer centers at the Oak Ridge National Laboratory (ORNL), National Energy Research Supercomputer Center (NERSC) from first principles, have been used to evaluate and tune the performance of these programs.

1. Introduction

The architecture of networks and distributed systems, which is getting more aggregate power and is thus the most cost-effective way of handling the trend is being driven by the need for more processing power, robustness, ease of use, and a low cost. By utilizing these tools and software, it can be generated with minimal effort.

One of the initial approaches to this problem, which is getting more attention, is the use of heterogeneous computing environments. PVM has been used on a variety of parallel computer systems, such as the Connection Machine (CM-5) and iPSC/860 at Emory University, as well as on a variety of distributed systems, such as the University of Tennessee's MPP system.

* This research was performed under the auspices of the U.S. Department of Energy's Office of Science and Energy Research, U.S. Department of Energy's Office of Science and Energy Research.

1Oak Ridge National Laboratory
2Oak Ridge National Laboratory
3University of Tennessee