CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers

Vasanth Bala* Jehoshua Bruck† Robert Cypher‡ Pablo Elustondo§
Alex Ho† Ching-Tien Ho† Shlomo Kipnis¶ Marc Snir‡

*Kendall Research Corp.
Waltham, MA 02154

†IBM Research Division
Almaden Research Center
San Jose, CA 95120

‡IBM Research Division
T.J. Watson Research Center
Yorktown Heights, NY 10598

§IBM Argentina
Beunos Aires, Argentina

¶IBM Israel Science and Technology
Haifa, Israel 31905

Abstract

A collective communication library for parallel computers includes frequently used operations such as broadcast, reduce, scatter, gather, concatenate, synchronize, and shift. Such a library provides users with a convenient programming interface, efficient communication operations, and the advantage of portability. A library of this nature, the Collective Communication Library (CCL), intended for the line of scalable parallel computer products by IBM, has been designed. CCL is part of the parallel application programming interface of the recently announced IBM 9076 Scalable POWERparallel System 1 (SP1). In this paper, we examine several issues related to the functionality, correctness, and performance of a portable collective communication library while focusing on three novel aspects in the design and implementation of CCL: (i) the introduction of process groups, (ii) the definition of semantics that ensures correctness, and (iii) the design of new and tunable algorithms based on a realistic point-to-point communication model.

Index terms: collective communication algorithms, collective communication semantics, message-passing parallel systems, portable library, process group, tunable algorithms.

1 Introduction

The need for collective communication arises frequently in parallel computation. Collective communication operations simplify the programming of applications for parallel computers, facilitate the implementation of efficient communication schemes on various machines, promote the portability of applications across different architectures, and reflect conceptual grouping of processes. In particular, collective communication is extensively used in many scientific applications for which the interleaving of stages of local computations with stages of global communication is possible (see [27]).

This paper discusses issues related to the design and implementation of a portable and tunable Collective Communication Library (CCL). This library is intended for distributed-memory parallel computers where explicit communication among processes is achieved via message-passing. In addition to point to point communication operations, such as `send` and `receive`, users can program using CCL routines for collective operations. The set of collective communication operations in CCL consists of the following routines: `bcast` — one-to-all broadcast, `reduce` — all-to-one reduction, `combine` — all-to-all reduction, `scatter` — one-to-all personalized communication, `gather` — all-to-one personalized communication, `concat` — all-to-all broadcast, `index` — all-to-all personalized communication, `prefix` — scanned reduction, `shift` — one-to-one cyclic permutation, and `sync`—barrier synchronization. A complete listing with brief descriptions of the functionality of all the CCL routines is provided in the Appendix.

The CCL was designed for the new IBM line of scalable parallel computers. (The first computer of this line of products, the IBM 9076 Scalable POWERparallel System 1 (SP1), has been announced recently.) Most of CCL is implemented as part of the parallel application programming interface of SP1.

Over the past few years, a large number of programming environments and communication libraries for parallel computers have been developed, including PVM [8], Linda [16], PICL [29], PARMACS [31], Zipcode [35], Express [37], the nCUBE/2 library [34], the CM-5 library [36], and the iPSC/860 library. The design and implementation of the CCL adopts some of the popular communication concepts that already exist in many of these libraries, and, in addition, it provides several novel aspects. In this paper, we mainly focus on three novel aspects in the design and implementation of CCL: (1) the introduction of process groups, (ii) the definition of semantics that ensures correctness, and (iii) the design of new and tunable algorithms based on a realistic point-to-point communication model.

For simplicity and clarity of the discussion in this paper, we only address scenarios in which dynamic process creation or deletion in run-time is not permitted. The CCL routines can either operate over the entire set of processes that are created at the beginning of an application or over user-specified groups of processes. For example, a user can view several processes as forming a two-dimensional array and performing independent broadcast operations in different columns of this array. Moreover, the CCL routines can also operate on groups of processes that are determined dynamically, namely, in run-time. To facilitate

the use of such groups of processes, CCL includes routines that enable the definition and the handling of dynamic process groups. Section 2 examines issues related to process groups in CCL.

The notion of creating and using process groups has been known in the distributed computing community, such as in the V system [17], ISIS [9] and Transis [2]. In the parallel applications environment, the support for creating and using process groups for collective communication routines has been very limited. For instance, most libraries on hypercubes only support broadcasts within subcubes. Although Express also supports collective communication within an arbitrary group of processes, there is no support in creating process groups by partitioning an existing group into subgroups based on a local value supplied by each process. Such notion and its use in parallel programming was first suggested in [3, 4], and our work influenced the introduction of this notion in an earlier MPI proposal [22] and the recent MPIF proposal [21].

A major goal in the design of CCL was the creation of a truly portable library that can run correctly and efficiently on a wide range of parallel and distributed computer architectures, regardless of the topology of the interconnection network and the speeds of the processors and the communication network. Also, the CCL can be easily implemented on top of message-passing libraries such as IBM EUI [28], IBM VIPER Operating System [25, 30], Parasoft Express [37], and PVM [8].

The CCL is a software layer that can sit on top of point-to-point communication primitives. A crucial step in the design of CCL was to define a simple and realistic model for the underlying point-to-point communication. This allows CCL to be both portable and efficient over a wide range of parallel machines. This model is discussed in Section 3. In particular, Section 3 examines several fundamental issues related to the semantics of collective communication operations over Process Groups.

For CCL to be scalable and efficient on a variety of parallel machines, the algorithms designed for collective communication operations must be tunable and exploit the characteristics of the particular parallel computer in a way that is hidden from the user. In Section 4, we address issues related to the design of tunable algorithms for collective communication operations in CCL. Finally, in Section 5 we present some concluding remarks.

2 Process Groups in CCL

Process Grouping is a mechanism for grouping processes into logical sets and for manipulating and communicating among processes in such sets. This section describes the concept of Process Groups in CCL, which is based on a similar concept that was presented in [3]. (In the IBM EUI documentation [28], Process Groups are called Task Groups.)

2.1 Defining Process Groups

A Process Group is an ordered set of processes that has a system-wide unique name. It can be operated upon as a single object. Each Process Group is identified by a unique Process Group Identifier (PGID) in an application program. All the processes that are created at the initialization of an application belong to the predefined Process Group `ALL`. In addition, each process has a unique Process Identifier (PID) named `mypid` which could be identified with the singleton set consisting only of that process.

There are two mechanisms in CCL for defining new Process Groups. One mechanism can be used to define a new Process Group by providing an explicit list of process identifiers (PIDs) that comprise the new Process Group. This mechanism requires each member process of a to-be-formed new Process Group to know the PIDs of all the processes in the group. Using this mechanism, all the processes that comprise a new Process Group must call the `group()` routine:

```
pgid = group(size, list, label);
```

The usage of `label` will become clear later. The call to `group()` must be made by each member process of the to-be-formed Process Group, and each member process must supply the same list of PIDs and in the same order. Processes that belong to the new Process Group obtain a unique PGID for this group. Processes that do not wish to be part of a new Process Group do not call the `group()` routine.

The other mechanism for defining new Process Groups in CCL is by partitioning existing Process Groups according to some criteria. This mechanism does not require the members of a to-be-formed Process Group to know the PIDs of all the processes that will comprise this group. When partitioning an existing Process Group `G`, several new Process Groups are created as follows. Every process in the group `G` supplies a local integer value to `myval`. All the processes that supply the same value to `myval` are made members of the same new Process Group. The partitioning of an existing Process Group is performed by the `partition()` routine:

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

The call to `partition()` must be made by every member of the existing Process Group `G`. No member of the existing group `G` returns from this call until all processes in group `G` have made the call. (In that sense, the call to `partition()` imposes an implicit barrier synchronization on the members of `G`). This call results in partitioning the members of group `G` into as many new groups as the number of distinct integer values supplied by the member processes.

A process can participate in several `group()` or `partition()` calls and, thus, be a member of several different Process Groups. Each call to `group()` or `partition()` may correspond to creating Process Groups based on a new criterion. Each such call, therefore, creates unique PGIDs for the resultant groups.

All the processes that belong to a particular Process Group `G`, regardless of whether the group was created by calling `group()` or `partition()`, are ranked from 0 to $n-1$, where n is the size of the group. The ranking of the processes in a Process Group that was created by a call to `group()` is the same as the order of the PIDs in the list supplied to the `group()` call. The ranking of the processes in a Process Group that was created by a call to `partition()` is determined by the additional parameter, `key`, that is supplied to the `partition()` call as an input argument. Ties in the ranking as a consequence of calling `partition()` are to be broken by processes' ranking in the parent group.

For example, suppose that there are 7 processes, with PID's from 0 to 6 (this is the `ALL` group). For convenience, we refer to a process with an even (resp. odd) PID as an *even* (resp. *odd*) process. Suppose that we want to create the following two types of groups while maintaining the original ordering: G in each even process is defined as the group that consists of all even processes and G in each odd process is defined as the group that consists of all odd processes. There are two ways to do it. The first is to have the following code:

```
EXAMPLE 1:
if (pid_is_even()) {
   label = EVEN;
   G = group(4, [0,2,4,6], label);
}
else {
   label = ODD;
   G = group(3, [1,3,5], label);
}
```

The second approach is to use `partition`. The code in this case is:

```
G = partition(ALL, is_mypid_odd, key);
```

Consider another example where all processes are partitioned into two groups depending on some locally computed value (temperature). The code is:

```
EXAMPLE 2:
if (temperature < BOILING)
   myval = COLD;
else
   myval = HOT;
key = temperature;
G = partition(ALL, myval, key);
```

In this example, each of the two newly formed groups is sorted according to the ascending order of `key`, which was assigned the integer value of temperature. CCL also provides utility routines for processes to determine the size of a Process Group (`getsize(G)`), to obtain the list of PIDs in a given Process Group (`getmembers(G)`), to query the rank of a process in a Process Group (`getrank(G, pid)`), and to query the PID of a process with a given rank in a Process Group (`getpid(G, rank)`). Finally, the user can call `getlabel(G)` to obtain the `label` supplied to `group()` and the `my_val` supplied to `partition()`.

2.2 Using Process Groups

Once established, Process Groups allow entire collections of processes to be identified and manipulated in a single call. Note that when different actions are required for different disjoint groups, the user can use `label` to distinguish groups. For example, the following code may follow the code of Example 1 above.

```
if (getlabel(G) == EVEN) {
   code-segment-1;
}
else  {
   code-segment-2;
}
```

As another example, the following code may follow the code of Example 2 above.

```
if (getlabel(G) == COLD) {
   code-segment-1;
}
else {
   code-segment-2;
}
```

A group-wide transaction of particular interest is collective communication over all the members of a Process Group. For example, a group-wide reduction operation, which takes the PGID of a Process Group as a parameter, can be written as:

```
combine(G, rfunc(), data, result);
```

Such a routine could implement a reduction, using a supplied associative function `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 `result`. All processes in the group make the call with the same actual parameters `G` and `rfunc()`. Similarly, a group-wide broadcast operation can be written as:

```
bcast(G, orig, data);
```

This routine would implement a broadcast communication from a particular member of the Process Group `G`, whose PID is `orig`, to all of the other members of `G`. All processes in the group make the call with the same actual parameters `G` and `orig`, otherwise the user's program is erroneous.

A participating process can proceed past the call to the collective communication immediately after it has finished its participation in it (even though the communication as a whole may still be in progress). For example in a grid-type problem domain, consider the sequence of calls:

```
row_pgid = partition(grid_pgid, row_id, key);
```

```
col_pgid = partition(grid_pgid, col_id, key);
bcast(row_pgid, orig1, data1);
bcast(col_pgid, orig2, data2);
```

where `row_pgid` and `col_pgid` represent two Process Groups for each of the processes invoking the two `partition()` calls. Since `row_pgid` and `col_pgid` are not disjoint, the second broadcast operation may be partially completed in parallel with the first one. The program execution order guarantees that the processes in `row_pgid` $\cap$ `col_pgid` will reach the second broadcast only after they finish their participation in the first broadcast. CCL prevents possible mixing of messages between the two broadcast operations by using the semantics and properties described in Section 3 so that the underlying communication subsystem can distinguish between the two sets of messages.

2.3 Unique Process Groups Identifiers

The actual value of a PGID is determined by CCL, and user code that depends on the actual value of a PGID (such as branching based on the value of a PGID) is not allowed. That is, such code may operate correctly with one implementation of CCL and not with another. On the other hand, the user is allowed to check equality of PGIDs and to include PGIDs in messages sent from one process to another. As a result, CCL must guarantee that all the processes in a single Process Group use the same system-wide unique PGID, and that distinct Process Groups have distinct PGIDs (whether or not they have any processes in common).

We now describe a simple mechanism for generating PGIDs that are system-wide unique. Each process maintains a local counter, which is initialized to 0. Logically, a PGID consists of two fields: `(counter, pid)`. When a new Process Group is created, the process that has rank 0 does the following three things in order: (1) determines the PGID by pairing its local counter and its PID, (2) broadcasts the value of the PGID to all the processes in the Process Group, and then (3) increments its local counter. This scheme guarantees system-wide uniqueness of all the PGIDs in CCL.

The implementation of the above mechanism in CCL is slightly different. Because most existing communication subsystems have relatively large start-up costs compared to the transfer times, CCL appends the value of the local counter of each process to the control information communicated during the setup of a Process Group. This eliminates the extra communication steps needed for broadcasting the PID and the counter of the process with rank 0, since each process has all the information needed to calculate the PGID of the new Process Group to which it belongs.

2.4 Common Group Structure Routines

Recently, a set of Common Group Structure (CGS) routines has been proposed as an extension to CCL [11]. The CGS routines make use of process group routines for defining

process groups that arise in algorithms with grid and hypercube structures. Once these grid-structured and hypercube-structured groups have been defined, the standard CCL routines can be used within the structured groups to perform collective communication operations. The CGS routines also include utilities for converting between 1-dimensional and higher-dimensional addresses.

As an example, the **FORM2DGRID** routine takes an existing group, views it as an $X \times Y$ 2D-grid (where X and Y are specified by the user), and partitions it into a set of nonoverlapping groups corresponding to the columns of a two-dimensional grid and also into a set of nonoverlapping groups corresponding to the rows of a two-dimensional grid. Thus, each process receives two new PGIDs, one for the column in which it is located and one for the row in which it is located in the two-dimensional grid.

3 Semantic Issues in CCL

This section discusses several issues related to the semantics of point-to-point communication and of collective communication operations on Process Groups. These issues are fundamental to the design of CCL and of similar libraries.

3.1 Send and Receive Semantics

In order for CCL to be portable, it is important that it is based on a simple model of point-to-point communication that is available on a wide range of machines.

The basic model of point-to-point communication consists of a single `send` operation and a single `receive` operation. It is assumed that a process can send/receive to/from any other process in the parallel machine. Specifically, for the purpose of defining send and receive semantics here and modeling the communication complexity in the next section, we treat the underlying topology as if it is fully connected. In addition it is assumed that the send and receive operations have the following five properties:

(1) The operations are blocking, namely, a *blocking send* operation and a *blocking receive* operation are used. The blocking send operation is a `send` that does not return to the user until the message has been copied out of the user's buffer. It should be pointed out that *blocking send* is different from *synchronous send* in which the `send` does not return to the user until a matching `receive` operation has been issued.[1] The blocking receive operation is a `receive` that does not return to the user until the requested message has arrived and has been placed in the user's buffer.

(2) The communication subsystem may, but is not required to, buffer any or all of the messages. Thus, a `send` operation may or may not block until a matching `receive` is issued.

[1] Some researchers use the term *blocking send* to indicate what we have defined as *synchronous send*.

(3) The receive operation is a *receive-by-source*, where the receiver specifies the desired source of the message. Only a message from the specified source can be delivered to the receiver. The desired source must always be specified and the use of a wildcard value for the source parameter is not allowed.

(4) Point-to-point communication between any two processes is required to be FIFO such that multiple messages sent from one process to another are received in the same order in which they were sent. However, the `send` and `receive` operations are asynchronous, in the sense that no bounds are assumed on the relative speeds of the processes or the time required to pass messages between processes.

(5) Different `send` operations issued from multiple source processes to a single destination process are *non-interfering*. This means that the existence of messages that the receiver does not wish to select (because these messages do not match the source field specified by the `receive` operation) cannot prevent the reception of a message that the receiver does wish to receive. For example, if processes 1 through $n-1$ each sends a message to process 0 and if process 0 issues a `receive` in order to receive a message from process $n-1$, then this `receive` operation will get the message sent from process $n-1$ even if all of the other messages were sent earlier. While this non-interference property seems very natural, it could be violated by a system that stores messages in buffers on the receiver side, since such buffers could be filled by messages other than the one that the receiver selects to receive.

Although the basic model of point-to-point communication defined above is sufficient for implementing all of the CCL routines, the actual implementation of CCL uses two additional features that are found in many systems. These two features are a `send-receive` operation and the usage of message tags[2]. We next discuss these two additional features and their merits in implementing CCL.

The `send-receive` operation simultaneously sends a message to one process and receives a message from another process without requiring additional buffer space for either of the messages. CCL uses the `send-receive` operation for implementing `shift` operations within Process Groups. The `shift` operation may create a cycle of processes, each of which is sending a message to its successor in the cycle and receiving a message from its predecessor in the cycle. If a `send-receive` operation is not used, then the `send` and the `receive` operations must be paired carefully to avoid deadlocks. (A deadlock can occur if `send` operations are forced to wait for matching `receive` operations that are never issued.) For example, to avoid deadlocks in `shift` operations, one can pair the `send` and `receive` operations, if the cycle is of even length, as follows. Processes in even positions in the cycle would perform a `send` followed by a `receive`, while processes in odd positions in the cycle would perform a `receive` followed by a `send`. (Cycles of odd length can be handled with one more round of communication.) Although such an algorithm is possible in CCL, since each process knows its position in the cycle, it is much simpler and more efficient to use a `send-receive` operation for implementing the `shift`.

[2]Some researchers use the term *type* instead of *tag*.

Some point-to-point communication libraries allow the use of wildcard values in the source parameter of a receive call. Even if wildcard values are not used in the implementation of CCL, there is still the risk that a message sent as part of the implementation of a CCL call will be mistakenly received by a receive call with a wildcard source issued by the user. An additional mechanism is needed to distinguish CCL messages from user messages. Message tags can provide such mechanism. These are values that are appended to the content of point-to-point messages. Tags are typically used to distinguish between different types of messages in an application, as well as between multiple messages sent to a destination from the same source. Each `send` operation specifies a value in the tag field of the message, and each `receive` operation specifies a desired tag value to be matched with the received message.

We assume that a mechanism is available to allocate tag ranges to various libraries, so that user messages and messages generated by CCL have disjoint tag ranges. If wildcard tag values are allowed in receive operations, one also needs to make sure that user receive operations cannot match tag values used by CCL. This can be achieved either with an include/exclude mechanism for wildcard tag ranges, as provided by Express [37] or with an additional context mechanism, as provided by Zipcode [35].

3.2 Implementation Correctness

There are many issues related to a correct implementation of CCL. These issues include, for example, separating user point-to-point messages from CCL point-to-point messages, guaranteeing that the correct messages be delivered for collective communication operations in overlapping Process Groups, and distinguishing between messages that belong to different collective communication operations in the same group.

We will concentrate on the correct implementation of the CCL routines assuming the basic model of point-to-point communication satisfying the five properties listed above. The key to this implementation is the notion of deterministic code. Code will be said to be *deterministic* if (*i*) it uses only the basic `send` and `receive` primitives given above, (*ii*) it does not deadlock even when every `send` operation is forced to wait for a matching `receive` operation, and (*iii*) all local computations are deterministic. It has been proven that a deterministic program will always behave identically, regardless of the system on which it is implemented [19]. The correctness of the CCL implementation follows from the fact that each CCL routine is deterministic. As a result, each CCL routine is guaranteed to operate correctly when it is run in isolation.

Of course the CCL routines must also operate correctly when they are run as part of an application consisting of point-to-point communication and multiple CCL operations. The non-interference property (property 5 above) guarantees that the presence of application-level point-to-point messages and other CCL operations does not prevent the successful completion of a CCL routine.

Furthermore, the semantics of CCL assume that each CCL routine may imply, but need

not imply, a barrier synchronization of the processes calling the CCL routine (this issue is explored in more detail in the following subsection). Therefore, one requirement of a correct program using CCL is that any CCL operation that involves processes p and q should be called in the same order by processes p and q, and no point-to-point communication between processes p and q should overlap a CCL operation involving p and q. As a result, it follows from the FIFO property of the communication (property 4 above) that in a correct application, messages from CCL operations on overlapping groups, repeated CCL operations on the same group, and application-level point-to-point communication cannot be erroneously received by CCL, and vice-versa.

3.3 Barrier Semantics

As discussed above, for program correctness, each CCL operation should be viewed as possibly imposing a barrier synchronization on its participants. However, from the performance point of view, imposing a barrier synchronization on all the participants in CCL operations may be undesirable. To address this problem, CCL allows two modes of operation: *barrier mode* and *non-barrier mode*. Selection between these two modes is done globally at the beginning of an application. The default mode is the barrier mode. When a CCL operation is executed in barrier mode, no process can complete its call to the routine until all the processes in the relevant Process Group have executed their (corresponding) calls to the same routine. Therefore, when an operation is executed in barrier mode, there must exist some point in time at which all the processes in the group are *simultaneously* executing the same operation. In contrast, when an operation is executed in non-barrier mode, each process blocks only until it has completed its participation in the operation. Notice that the role of each process may depend not only on the semantics of the operation but also on the specific algorithm used to implement it.

The differences between the two modes may imply differences in the behavior of an application program. In a non-barrier mode, processes may return more quickly from calls to CCL routines, resulting in a better performance. However, there is also some danger in using non-barrier mode routines as in some of CCL routines the role of each process depends also on the particular algorithm used to implement the routine. As long as the user's program is correct and deterministic, then non-barrier mode routines will operate correctly (as argued above) and in an implementation-independent manner. However, if wildcard point-to-point communication is used, then the behavior of a non-barrier mode routine may be implementation-dependent. As an example, consider the following piece of code, in which `G` is the group identifier of a Process Group consisting of processes `a`, `b`, and `c`. In this example, each `brecv` is a blocking receive operation with a wildcard source parameter.

```
process a       process b       process c
---------       ---------       ---------
bsend(c)                        brecv(*)
bcast(G)        bcast(G)        bcast(G)
```

```
bsend(c)        brecv(*)
```

Assume that the source of the `bcast` operation is process `b`. If barrier semantics are used for the broadcast operation, then the first receive of process `c` will receive the message sent by process `a`, and the second receive of process `c` will receive the message send by process `b`. On the other hand, if process `b` can exit the broadcast call before process `c` starts executing the broadcast, then it is possible for the messages to be received in the reverse order: process `c` receives before the broadcast the message that `b` sent after the broadcast, and receives after the broadcast the message that `a` sent before the broadcast!

In summary, non-barrier mode CCL routines may exhibit implementation dependent blocking behavior when nondeterministic point-to-point communication operations are allowed. Furthermore, even if no wildcard point-to-point communication is performed, a program that would deadlock with barrier mode CCL routines may complete successfully with non-barrier model CCL routines. Therefore, barrier mode should be used when debugging code (that is, in identifying bugs that may not be caught in non-barrier mode). Once a program is known to operate correctly in barrier mode, it is possible to switch to non-barrier mode for better performance (provided that the barrier mode is not required for the correctness of the program). Finally, notice that it is always possible to use an explicit synchronization (i.e., to call the routine `sync`) when a barrier is needed.

In our CCL, some collective communication operations have semantics that imply a barrier, while others do not. Operations that imply a barrier are `combine`, `concat`, `index`, and `sync`. Thus, the implementation of these operations is the same for barrier and for non-barrier modes. Other operations that do not imply a barrier are `bcast`, `reduce`, `scatter`, `gather`, `prefix`, and `shift`. In non-barrier mode, the latter operations (except `shift`) are implemented with a fan-in tree to a particular destination or a fan-out tree from a particular source. For implementing these operations in barrier mode, we use two internal partial synchronization routines: `sync-in(dest,G)` which builds a fan-in tree in the group `G` that converges at process `dest`, and `sync-out(src,G)` which builds a fan-out tree in the group `G` that diverges from process `src`. A barrier mode for the latter operations is implemented by performing a `sync-in()` before a fan-out type operation (like `bcast` and `scatter`), or by performing a `sync-out()` after a fan-in type operation (like `reduce`, `gather`, and `prefix`). Notice that for the `prefix` operation, the fan-out tree is rooted at the highest ranked process in the group. Finally, regarding the Process Group creation operations, `partition()` implies a barrier mode within the parent group while `group()` does not.

3.4 Develop and Run Modes

In addition to the choice between barrier and non-barrier modes, the user can select between *develop* and *run* modes. In *develop* mode, CCL performs more detailed consistency checkings of the parameters that the user specifies for CCL calls. This increases the overhead of the CCL operations and may require more communication between processes. In *run* mode, many of the consistency checkings are suppressed, and only checkings that are

required by the semantics of the operations are performed. The default in CCL is *run* mode and the mode can be changed by the user. Examples of consistency checkings that are done only in *develop* mode are checking the correctness of the source, the destination and the message length supplied to a CCL routine, as well as checking the membership list of a Process Group to be valid.

4 Performance Issues in CCL

This section discusses several performance issues and describes some of the algorithms used in CCL. The design of communication algorithms for CCL attempts to address both the efficiency and the portability of the communication operations. In the discussion of the communication algorithms in this section, we assume one process per processor.

4.1 Communication Model

In designing algorithms for the CCL operations, we assume a model of a fully-connected message-passing system in which each process can communicate directly with any other process and every pair of processes are equally distant. We also assume that each process can send one message and, simultaneously, receive another message in the same communication step. (This is usually done using a `send-receive` operation found in many parallel systems.) In most existing message-passing parallel systems, the time for sending an m-byte message from process p to process q, without congestion, can be modeled as $T = t_s + mt_c$, where t_s is the overhead (start-up time) associated with each send and/or receive operation, and t_c is the communication time for sending each additional byte (or any appropriate data unit).

Such a fully-connected model addresses emerging trends in many modern distributed-memory parallel computers and message-passing communication environments. These trends are evident in systems such as Thinking Machines' CM-5 [36], Intel's Paragon [38], NCUBE's nCUBE/2 [34], MIT's J-Machine [20], IBM's Vulcan [10, 39], and the recently announced IBM's Scalable POWERparallel System 1 (SP1), and in environments such as Express [37], PARMACS [31], PICL [29], Zipcode [35] and Venus [4]. These systems and environments generally ignore the specific structure and topology of the communication network and assume a fully-connected collection of processes, in which each process can communicate directly with any other process by sending and receiving messages. The fact that the model does not assume any single topology makes it more general and flexible. This model, for instance, allows the creation of algorithms that are portable between different machines, that can operate within arbitrary and dynamic subsets of processes, and that can operate in the presence of faults (assuming connectivity is maintained). In addition, algorithms developed for this model can also be helpful in designing algorithms for specific topologies.

To examine the performance of communication algorithms, we define the following three

communication complexity measures:

- C_1: the number of communication steps required by an algorithm. C_1 is an important measure when the communication start-up time is high relative to the transfer time of one unit of data and the message size per send/receive operation is relatively small.
- C_2: the amount of data (in the appropriate unit of communication: bytes, flits, or packets) transferred in sequence by any process. C_2 is an important measure when the start-up time is small compared to the message size.
- C_3: the total amount of data communicated over the network (in the appropriate unit of communication: bytes, flits, or packets). Measures C_1 and C_2 do not address the issue of load on the network. C_3 also considers the fact that communicating more data over the network causes the network to become more congested.

Thus, under the fully-connected model, an algorithm has an estimated communication time complexity of $T = C_1 t_s + C_2 t_c$. The term C_3 does not affect the complexity here, but may be helpful in estimating the congestion behavior of a real parallel machine. For instance, one can model $T = C_1 t_s + C_2 t_c + f(C_3)$, where the function f can be derived for a given machine either explicitly or by curve-fitting experimental results.

It should be noted that there are more detailed communication models, such as the Postal model [5] and the LogP model [18], which further take into account that a receiving process generally completes its `receive` operation later than the corresponding sending process finishes its `send` operation. However, designing efficient algorithms for these models seems to be more complicated. Another important issue is the uniformity of the implementation. For example, in the LogP model, the collective communication algorithms are designed based on P (the number of processors). The optimal algorithms for two distinct values of P are sometimes very different. This presents a challenge if the goal is to support collective communication algorithms for various sizes process groups using the same collective communication library.

4.2 Tunable Algorithms

One goal in the design of the algorithms for CCL was that they be tunable, that is, that they exhibit a trade-off between the different communication complexity measures. This goal is important for such a library to be both portable and efficient. In the following discussion, we use n to denote the number of processes (processors) involved in a CCL operation, and we use m to denote the size of data each process has initially. All CCL routines can be implemented with $C_1 = \lceil \log_2 n \rceil$ communication steps, which is optimal for this measure. (The `shift` routine can be implemented optimally in one communication step, by using the `send-receive` operation.) However, when designing an algorithm to minimize C_1, some of the corresponding terms for C_2 or C_3 may not be optimal. In general, there are tradeoffs between C_1 and C_2, or between C_1 and C_3.

As an example, consider the `index` operation. A straightforward implementation of the `index` operation can be achieved with $C_1 = n-1$ communication steps and $C_2 = m(n-1)/n$ units of data. This is simply by sending the data directly from source to destination. Namely, each process sends m/n units of data to $n-1$ other processes. Another approach is by using a different radix for representing the PIDs of the processes (the straightforward approach is using a radix n representation). In the case of a radix 2 representation we get $C_1 = \lceil \log n \rceil$ communication steps and $C_2 \approx (m \log n)/2$ units of data. In general, by choosing an appropriate radix r, the `index` operation can be implemented with $C_1 \approx (r-1)\log_r n$ communication steps and with $C_2 \approx (m(r-1)\log_r n)/r$ units of data. Hence, it is possible to implement a parameterized algorithm which can be tuned according to the start-up time t_s, per-byte transfer time t_c, the message size m, and possibly the number of "parallel ports" that can support concurrent sends and receives effectively (see [15]).

As a second example, consider the broadcasting problem. The algorithm for the `bcast` routine is straightforward when the size of the data is $m = 1$, that is, when the source of the broadcast has one item to broadcast. In this case, a divide-and-conquer algorithm provides an optimal solution, also known as *recursive doubling*. However, the broadcasting problem becomes much more complicated when there are $m > 1$ units of data to broadcast. In this case, a lower bound on the time is $\lceil \log n \rceil\, t_s + (m + \lceil \log n \rceil - 1)t_c$. The common divide-and-conquer algorithm based on a binomial tree takes $\lceil \log n \rceil\, (t_s + mt_c)$ time, which may be far from optimal. When n is a power of two, an algorithm based on $\log n$ edge-disjoint spanning trees on a $(\log n)$-cube is given in [33, 32]. This algorithm requires $C_2 = m + \log n - 1$, which is optimal, and takes $T = (\sqrt{mt_c} + \sqrt{\log n t_s})^2$ time. For arbitrary values of n, an algorithm based on the generalized Fibonacci trees was given in [13] with $C_2 \leq m + \log n + 3\log\log n + 15$. More recently, an algorithm based on edge-disjoint spanning trees of cascaded decreasing-size hypercubes was given [6] with nearly optimal $C_2 = m + \lceil \log n \rceil$. The `reduce` operation is usually implemented in a similar manner to the `bcast` operation by reversing the flow of the messages.

As another example, consider the `combine` and `prefix` operations. These operations exhibit an interesting trade-off between measures C_1 and C_3. On one hand, these operations can be implemented with $C_1 = 2\lceil \log n \rceil$ and $C_3 = O(mn)$ using a reduction tree followed by a broadcast tree. On the other hand, these operations can be implemented with $C_1 = \lceil \log n \rceil$ and $C_3 = O(mn \log n)$ using a butterfly-type or circulant-graph-type communication pattern. In fact, hybrid algorithms for these operations exist. For instance, a hybrid algorithm for the `combine` operation requires $C_1 = \lceil \log n \rceil + k$ communication steps and communicates $C_3 = m\frac{n}{2^k}(\log n + 2^{k+1} - k - 2)$ units of data (see [1]).

The `scatter` and `gather` operations resemble the `bcast` and `reduce` operations in their functionality. However, in terms of their performance, these operations can be implemented with $C_1 = \lceil \log n \rceil$ communication steps and $C_3 \approx (m \log n)/2$ units of data, or, alternatively, they can be implemented with $C_1 = n - 1$ communication steps and $C_3 = m(n-1)/n$ units of data.

4.3 An Optimal Algorithm for Concatenation

In this subsection, we outline an algorithm for the `concat` operation. (In [14] we showed that this algorithm is optimal with respect to measures C_1, C_2 and C_3.) In the `concat` (all-to-all broadcast) operation among n processes, each process has a fixed-size message (also called a *data block)* that it needs to broadcast to the other $n-1$ processes. Thus, at the end, each process has all n data blocks. Because n need not necessarily be a power of two, the simple optimal algorithm for concatenation, which is based on a butterfly-type communication pattern, cannot be directly used.

The optimal algorithm for the `concat` operation is based on the structure of a circulant graph. A *circulant graph* $G(n, S)$ is characterized by two parameters: the number of nodes n, and a set of integer offsets S. The nodes of the graph are labeled from 0 through $n-1$. Each node i is connected to nodes of the form $((i \pm s) \bmod n)$ for all $s \in S$ (see [24]). Circulant graphs are an important class of networks which can be used as fault-tolerant networks for many other networks [12, 23]. We use a circulant graph with power-of-two offsets $S = \{1, 2, 4, \cdots, 2^{k-1}\}$, where $k = \lceil \log n \rceil$, as a structure for the `concat` algorithm [14] (see Figure 1).

The algorithm consists of k steps. In step 0, each process i sends its data block to process $(i-1) \bmod n$, receives a data block from process $(i+1) \bmod n$, and appends the received data block to its current data. In general, in step j, for $0 \leq j \leq k-2$, each process i sends all its current cumulated 2^j data blocks to process $(i-2^j) \bmod n$, receives 2^j data blocks from process $(i+2^j) \bmod n$, and appends the received data to its cumulated data blocks. In the last step, step $k-1$, each process i sends only the last $n-2^{k-1}$ blocks of data that it cumulated to process $(i-2^{k-1}) \bmod n$, and it receives the same number of data blocks from process $(i+2^{k-1}) \bmod n$. Figure 2 presents an example of this algorithm.

In general, all the homogeneous operations (operations for which there is no notion of a distinct source and/or destination, [26]) in CCL, can be implemented with the minimal number of start-ups ($\lceil \log n \rceil$ communication steps) using the same circulant graph structure. Examples of other such operations are `combine`, `index`, and `sync`. An algorithm for the `combine` operation with $T = \lceil \log n \rceil (t_s + mt_c)$, which is optimal with respect to C_1 (and also to C_2 when $m = 1$), appears in [7].

4.4 Specialized Algorithms

For certain CCL operations, the best-known algorithms for the case when n is a power of 2 may perform better than algorithms for the general case. Since, in many situations, n is a power of 2, it is worthwhile to implement this case separately. For instance, the `concat` algorithm for n that is a power of two can use a well-known hypercube recursive exchange algorithm which eliminates shifting local arrays at the end of the operation. As another example, when n is a power of two, the `bcast` algorithm described in [32] is substantially simpler, in terms of local data structures and control, than the algorithm for arbitrary

values of n described in [6].

5 Conclusions

We have described the main issues that we have encountered in designing and implementing a Collective Communication Library (CCL) for the recently announced IBM Scalable POWERparallel System 1, (SP1). We have focused on three novel aspects in the design and implementation of CCL: the introduction of process groups, definition of semantics that ensures correctness and the design of novel algorithms based on a realistic point-to-point communication model. Each of these novel aspects suggests interesting avenues for further learning and research.

Acknowledgements

We thank the following people: Dan Frye for his valuable comments and help in revising the CCL semantics as part of the spec in [28], Magda Konstantinidou for her contributions to the initial stage of the CCL design, Eric Leu for his comments on a draft of this paper, many internal and external reviewers of [28] for their valuable comments to the CCL semantics, and Dragutin Petkovic for his constant support and help.

Appendix: List of CCL routines

CCL consists of two parts, the collective communication routines (`CC` part) and the process group routines (`PG` part which is called `TG` for task group in the EUI document.) The `CC` routines provide the functionality of operations involving collective communication, and the `PG` routines provide the functionality of specifying and manipulating groups.

The available `CC` routines are:

`bcast()`: Single source broadcast, i.e., broadcast a message from one process to all tasks in a group.

`reduce()`: Apply an associative (but not necessarily commutative) reduction operation on all the processes in a group, and place the reduction result in one user-specified process.

`combine()`: Apply an associative (but not necessarily commutative) reduction operation on all the processes in a group, and place the reduction result in each of the processes in the group.

`scatter()`: Distribute distinct messages from a single source to each process in a group.

`gather()`: Gather distinct messages from each process in a group to a single destination process.

`concat()`: Concatenate to all processes in a group. Each process in a group performs a `bcast` within the group.

`index()`: Each task in a group performs a `scatter` operation.

`prefix()`: Parallel prefix operation. It is sometimes called `scan`.

`shift()`: Shift data up or down some number of steps in a group.

`sync()`: Barrier synchronization in a group.

The `PG` routines are summarized below.

`group()`: Create a process group by explicitly specifying the processes in the group.

`partition()`: Define a process group by partitioning an existing group based on a locally supplied integer label.

`getsize()`: Get the size (the number of processes) of an existing process group.

`getmembers()`: Get the ordered array of process ids of an existing process group.

`getrank()`: Get the rank of a process in a process group.

`getpid()`: Get the process id of the process with a certain rank in a process group.

`getlabel()`: Get the label (which was supplied by a user when the group was formed) of an existing process group.

References

[1] A. Aggarwal and S. Kipnis, "Message-time tradeoff for combining data in message-passing systems", *IBM Research Report*, RC-18349, September 1992.

[2] Y. Amir, D. Dolev, S. Kramer, D. Malki, "Transis: a communication sub-system for high availability", *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, July 1992, pp. 76–84.

[3] V. Bala and S. Kipnis, "Efficient Collective Communication over Dynamically Determined Sets of Processors in Massively Parallel Computers", *IBM Research Report*, RC-17771, March 1992. Presented at the First CRPC Workshop on Standards for Message Passing in a Distributed Memory Environment, April 29–30, 1992, in Williamsburg, Virginia.

[4] V. Bala and S. Kipnis, "Process Groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library", *Proceedings of the 7th International Parallel Processing Symposium*, IEEE, April 1993.

[5] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems", to appear in *Mathematical Systems Theory*. Also appeared in *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1992, pp. 11–22.

[6] A. Bar-Noy and S. Kipnis, "Broadcasting multiple messages in simultaneous send/receive systems", to appear in *Discrete Applied Mathematics*. Also appeared in *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, December 1993.

[7] A. Bar-Noy, S. Kipnis, and B. Schieber, "An optimal algorithm for computing census functions in message-passing systems", *Parallel Processing Letters*, Vol. 3, No. 1, March 1993.

[8] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "A user's guide to PVM Parallel Virtual Machine", *ORNL Technical Report*, ORNL/TM-11826, May 1992.

[9] K. Birman, R. Cooper, T.A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck and M. Wood, "The ISIS system manual", Department of Computer Science, Cornell University, Spet. 1990.

[10] J. Bruck, R. Cypher, L. Gravano, A. Ho, C. T. Ho, S. Kipnis, S. Konstantinidou, M. Snir and E. Upfal, "Survey of routing issues for the Vulcan parallel computer", *IBM Research Report*, RJ-8839, June 1992.

[11] J. Bruck, R. Cypher, P. Elustondo, A. Ho, and C.T. Ho, "A proposal for common group structures in a collective communication library", *IBM Research Report*, RJ-9421, March 1993.

[12] J. Bruck, R. Cypher, and C.T. Ho, "Efficient fault-tolerant mesh and hypercubes architectures", *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, pp. 162–169.

[13] J. Bruck, R. Cypher, and C.T. Ho, "Multiple message broadcasting with generalized Fibonacci trees", *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, December 1992, pp. 424–431.

[14] J. Bruck, C.T. Ho, and S. Kipnis, "Concatenating data optimally in message-passing systems", *IBM Research Report*, RJ-9191, January 1993.

[15] J. Bruck, R. Cypher, C.T. Ho, and S. Kipnis, "Efficient algorithms for the index operation in message-passing systems", *IBM Research Report*, RJ-9300, April 1993.

[16] N. Carriero and D. Gelernter, "Linda in context", *Communication of the ACM*, Vol. 32, No.4, April 1989, pp. 444–458.

[17] D.R. Cheriton and W. Zwaenpoel, "Distributed Process Groups in the V Kernel", *ACM Trans. Comput. Syst.*, 2(3):77–107, May 1985.

[18] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: towards a realistic model of parallel computation", *Proceedings of the 4th SIGPLAN Symposium on Principles and Practices of Parallel Programming*, ACM, May 1993.

[19] R. Cypher and E. Leu, "Message-Passing Semantics and Portable Parallel Programs", *IBM Research Report*, RJ-9654, January 1994.

[20] W. J. Dally, A. Chien, S. Fiske, W. Horwat, J. Keen, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick, and G. Fyler "The J-Machine: a fine-grain concurrent computer", *Information Processing 89*, Elsevier Science Publishers, IFIP, 1989, pp. 1147–1153.

[21] *Document for a Standard Message-Passing Interface*, Message Passing Interface Forum, University of Tennessee, Technical Report No. CS-93-214, November, 1993.

[22] J. Dongarra, R. Hempel, A. Hey, and D. Walker, "A proposal for a user-level, message-passing interface in a distributed memory environment", *ORNL Technical Report*, ORNL/TM-12231, October 1992.

[23] S. Dutt and J. P. Hayes, "Designing fault-tolerant systems using automorphisms", *Journal of Parallel and Distributed Computing*, Vol. 12, 1991, pp. 249–268.

[24] B. Elspas and J. Turner, "Graphs with circulant adjacency matrices", *Journal of Combinatorial Theory*, No. 9, 1970, pp. 297–307.

[25] B. G. Fitch and M. E. Giampapa, "The Vulcan operation environment: a brief overview and status report", *Proceedings of the 5th Workshop on Use of Parallel Processors in Meteorology*, ECMWF, November 1992.

[26] G. Fox and W. Furmanski, "Optimal communication algorithms for regular decompositions on the hypercube", *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, ACM, 1988, pp. 648–713.

[27] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[28] D. Frye, R. Bryant, C.T. Ho, P. de Jong, R. Lawrence, and M. Snir, "An external user interface for scalable parallel systems: FORTRAN interface", *Technical Report*, IBM Highly Parallel Supercomputing Systems Laboratory, November 1992.

[29] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "A user's guide to PICL: a Portable Instrumented Communication Library", *ORNL Technical Report*, ORNL/TM-11616, October 1990.

[30] M. E. Giampapa, B. G. Fitch, G. R. Irwin, and D. G. Shea, "Vulcan operating environment: programmer's reference", *Internal Memorandum*, IBM T.J. Watson Research Center, March 1991.

[31] R. Hempel, "The ANL/GMD macros (PARMACS) in FORTRAN for portable parallel programming using the message passing programming model, user's guide and reference manual", *Technical Memorandum*, Gesellschaft für Mathematik und Datenverabeitung mbH, West Germany.

[32] C.T. Ho, "Optimal broadcasting on SIMD hypercubes without indirect addressing capability", *Journal of Parallel and Distributed Computing*, Vol. 13, No. 2, October 1991, pp. 246–255.

[33] S. L. Johnsson and C.T. Ho, "Spanning graphs for optimum broadcasting and personalized communication in hypercubes", *IEEE Transactions on Computers*, Vol. C-38, No. 9, September 1989, pp 1249–1268.

[34] J. F. Palmer "The NCUBE family of parallel supercomputers", *Proceedings of the International Conference on Computer Design*, IEEE, 1986.

[35] A. Skjellum and A. P. Leung, "Zipcode: a portable multicomputer communication library atop the Reactive Kernel", *Proceedings of the 5th Distributed Memory Computing Conference*, IEEE, April 1990, pp. 328–337.

[36] *Connection Machine CM-5 Technical Summary*, Thinking Machines Corporation, 1991.

[37] *Express 3.0 Introductory Guide*, Parasoft Corporation, 1990.

[38] *Paragon XP/S Overview*, Intel Corporation, 1991.

[39] "Vulcan system summary", *Internal Memorandum*, IBM T.J. Watson Research Center.

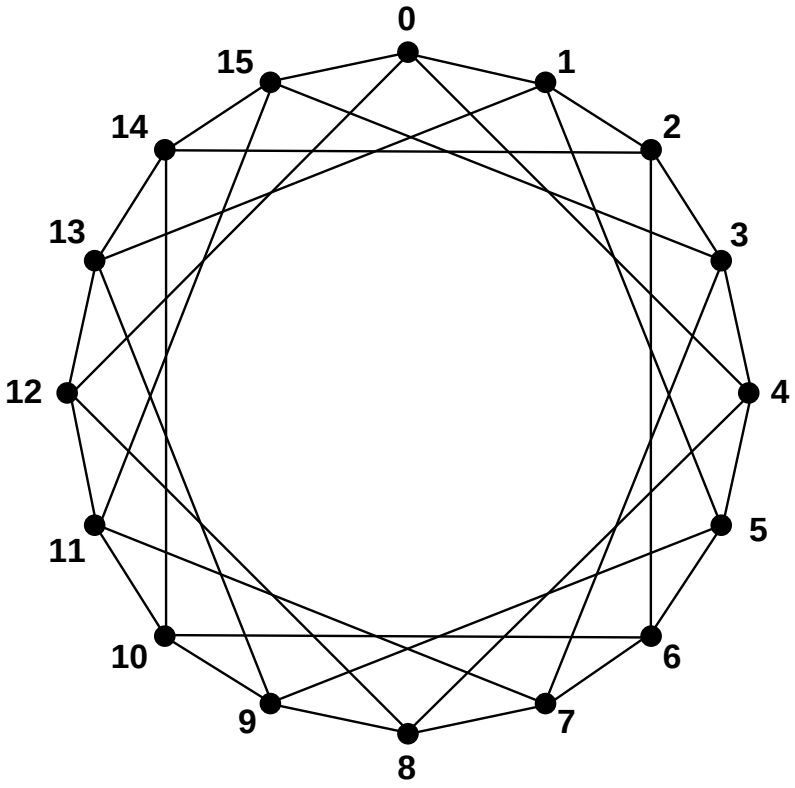

Figure 1: A circulant graph with 16 nodes and offsets 1 and 4.

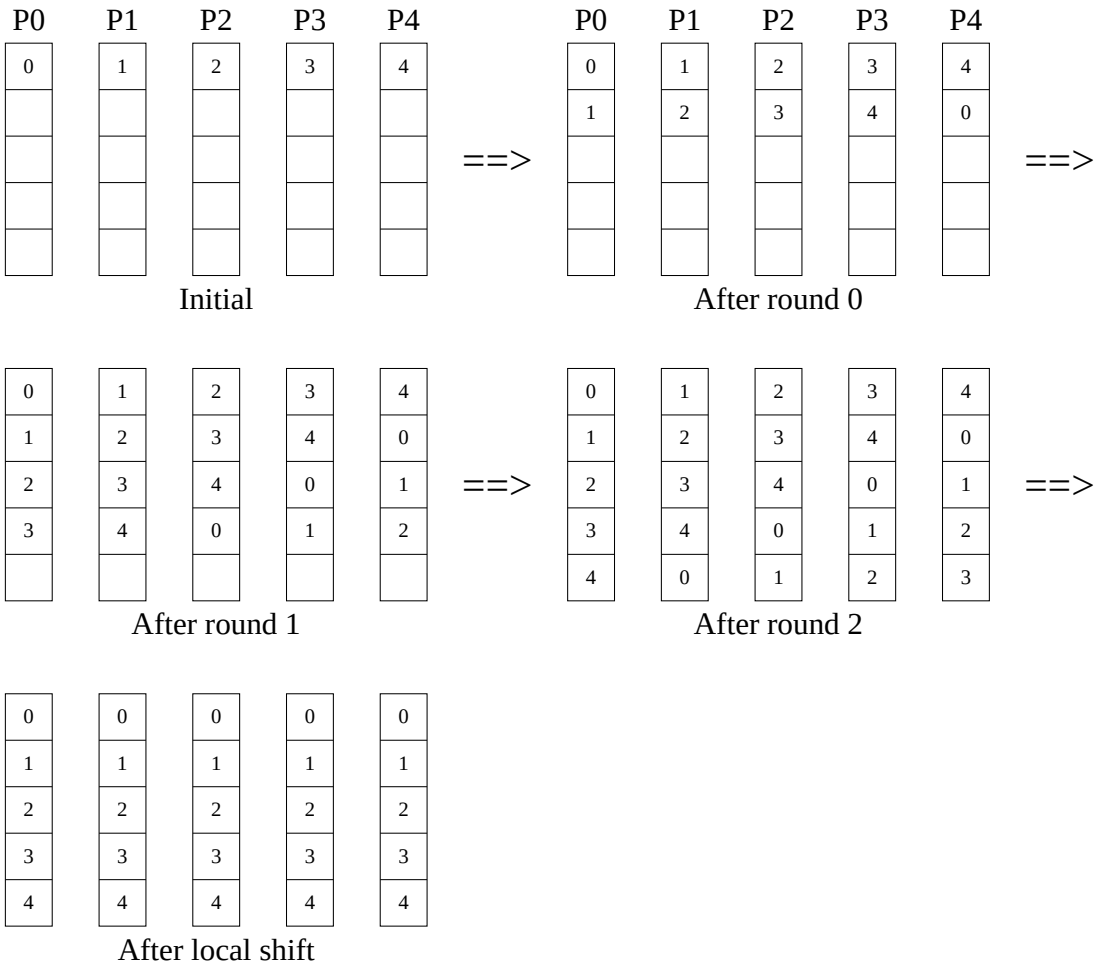

Figure 2: An example of the concatenation algorithm with 5 processors.