Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

© 2008 The authors and IOS Press. All rights reserved.

Process-Oriented Collective Operations

John Markus BJØRNDALEN [a,1] and Adam T. SAMPSON [b]
[a] *Department of Computer Science, University of Tromsø*
[b] *Computing Laboratory, University of Kent*

Abstract. Distributing process-oriented programs across a cluster of machines requires careful attention to the effects of network latency. The MPI standard, widely used for cluster computation, defines a number of *collective operations*: efficient, reusable algorithms for performing operations among a group of machines in the cluster. In this paper, we describe our techniques for implementing MPI communication patterns in process-oriented languages, and how we have used them to implement collective operations in PyCSP and occam-π on top of an asynchronous messaging framework. We show how to make use of collective operations in distributed process-oriented applications. We also show how the process-oriented model can be used to increase concurrency in existing collective operation algorithms.

Keywords. CSP, clusters, concurrency, MPI, occam-π, parallel, PyCSP, Python.

Introduction

Distributing process-oriented programs across a cluster of machines requires careful attention to the effects of network latency. This is especially true for operations that involve a group of processes across the network, such as barrier synchronisation. Centralised client-server implementations are straightforward to implement and often scale acceptably for multicore machines, but have the potential to cause bottlenecks when communication latency is introduced.

The MPI standard, widely used for cluster computation, provides a number of *collective operations*: predefined operations that involve groups of processes in a cluster. Operations such as barriers, broadcasting and reductions are already used in distributed process-oriented programs.

Implementations of MPI are responsible for providing efficient implementations of each operation using algorithms and communication mechanisms suited to the particular cluster technology for which they are designed. We examine one of these implementations, Open-MPI [1,2], to study how these algorithms can be expressed using process-oriented systems such as PyCSP [3,4] and occam-π [5].

We are studying how these algorithms can be reused in distributed process-oriented programs. We show how process-oriented implementations of these algorithms can express a higher level of parallelism than is present in OpenMPI. The Python language makes it easy for us to express these concurrent algorithms in a reusable and concise way.

For CSP-based programs executing on clusters, we believe that effective group operations can be implemented efficiently using a hybrid approach: existing process-oriented communication techniques within each node, and MPI-inspired collective operations between nodes in the cluster. Expressing collective operations using process-oriented techniques allows effective integration of the two approaches.

[1]Corresponding Author: *John Markus Bjørndalen, Department of Computer Science, University of Tromsø, N-9037 Tromsø, Norway*. Tel.: +47 7764 5252; Fax: +47 7764 4580; E-mail: `jmb@cs.uit.no`.

To improve the efficiency of communication between nodes, we have created a lightweight asynchronous messaging framework for occam-π and PyCSP.

This paper is organised as follows: In section 1 we provide a short introduction to PyCSP. In section 2 we describe our implementations of several of the OpenMPI collective operations using Python and PyCSP, and show some of our improvements to them. In section 3, we describe our implementations of lightweight asynchronous messaging. In section 4, we describe how our work can be applied in a CSP-based framework for configurable group operations. This work is a consequence of our experiences with CoMPI [6,7], a runtime configuration system for LAM-MPI [8], and the PATHS [9,10,11] system, used with a tuple-space system to achieve better performance than LAM-MPI for collective operations on clusters of multiprocessor machines. A configuration system like this can help us tune both an application and the collective operations it uses for improved performance on a given cluster. Part of the motivation for this paper is to explore initial possibilities for the configuration project.

1. A Quick Introduction to PyCSP

This section provides an overview of the facilities offered by PyCSP, a library to support process-oriented programming in Python. A more detailed description can be found in "PyCSP – Communicating Sequential Processes for Python" [3], which describes an earlier version of PyCSP; the examples in this paper make use of features introduced in PyCSP 0.3, such as network channels. The source code for PyCSP can be found on the PyCSP homepage [4].

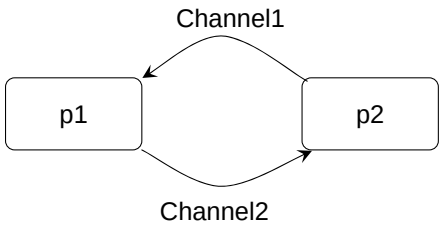

Figure 1 Basic PyCSP process network, with two processes: P1 and P2. The processes are connected and communicate over two channels: Channel1 and Channel2.

As in all process-oriented systems, the central abstractions offered by PyCSP are the *process* and the *channel*. A PyCSP program is typically composed of several processes communicating by sending messages over channels. Figure 1 shows an example with two processes, P1 and P2, communicating using channels Channel1 and Channel2.

Listing 1 shows a complete PyCSP implementation of the process network in Figure 1. A PyCSP process is defined by writing a function prefixed with the `@process` decorator. This ensures that a PyCSP `Process` object is created with the provided parameters when the function is called.

PyCSP processes are executed using the `Sequence` and `Parallel` constructs, as shown in the listing. `Parallel` starts all the provided processes in parallel, then waits for them all to finish; `Sequence` executes them in sequence, waiting for each process to finish before starting the next.

Channels are created by instantiating one of the channel classes. This example uses `One2OneChannel`, the simplest type of channel, which connects exactly two processes without buffering. The methods `write` and `read` are used to send and receive messages across the channel. Other channel classes provide sharing, buffering, and similar features.

PyCSP provides support for network communication using *named channels*. A channel is registered with the networking library by calling `registerNamedChannel(chan, name)`,

```
import time
from pycsp import *

@process
def P1(cin, cout):
    while True:
       v = cin()
       print "P1, read from input channel:", v
       time.sleep(1)
       cout(v)

@process
def P2(cin, cout):
    i = 0
    while True:
        cout(i)
        v = cin()
        print "P2, read from input channel:", v
        i += 1

chan1 = One2OneChannel()
chan2 = One2OneChannel()

Parallel(P1(chan1.read, chan2.write),
         P2(chan2.read, chan1.write))
```

Listing 1: Complete PyCSP program with two processes

which creates an entry in a nameserver. A remote process can then get a proxy to the channel by calling `getNamedChannel(name)`, which may be used in the same ways as a local channel object.

2. Collective Operations

The MPI collective operations are used to execute operations across a group of processes, either to synchronise the processes, exchange data, or execute a common operation on data provided by one or more of the participating processes. Examples of the provided operations are Barrier, Broadcast and Reduce.

OpenMPI uses several algorithms to implement each collective operation. When a program calls any of the operations, OpenMPI chooses which implementation to use based on parameters such as the size of the group (the number of MPI processes involved in that call), and the sizes of the messages involved. Additionally, parameters may be specified when starting MPI programs to influence the choices and select alternative implementations.

Note that for this paper, we are restricting ourselves to discussing the *basic* component of OpenMPI to limit the complexity of the discussions and code examples. The basic component includes well-known algorithms for implementing reasonably efficient group operations in clusters. We are also restricting ourselves to a selection of the group operations, focusing on those where we identify patterns that can be reused and the most common operations.

2.1. MPI Terminology

In MPI, the *size* of a group of processes is the number of processes within that group. Within an MPI program, a group is represented using a *communicator*. The communicator can be thought of as an object that keeps track of information such as the size of the group, which algorithms to use for various operations, and how to reach each of the other processes within the group.

Within a given group, each process has a *rank*, which is a unique ID identifying that process in the group. When a process wants to send a message to another process in the group, the sender invokes the *send* operation on the communicator representing the group, and specifies the rank of the receiving process.

The MPI standard defines a number of operations for sending and receiving messages, including variations such as asynchronous send and receive operations. These low-level operations are used to construct the higher-level collective operations.

Group operations, or collective operations, are called with the same parameters for all members of the group. The ranks of the processes are used to identify which processes is responsible for what part of the algorithm. For example, in the Broadcast operation, one process broadcasts a value to all the other processes. The sender is called the *root* process, and is identified by all participating processes by specifying which rank is the root process in the parameters to the operation.

2.2. Broadcast

In the MPI Broadcast operation, a *root* process distributes the same piece of data to all the other processes. For simplicity, we will assume that the root process is rank 0. OpenMPI handles the cases where root is not 0 by re-mapping the ranks of each participating process (to a virtual rank). Adding this to the implementations would not significantly alter the algorithms.

Listing 18 (in the Appendix) shows the *linear* implementation of Broadcast from OpenMPI 1.2.5. The linear algorithm is usually used for very small groups, where having the other processes communicating directly with the root node will be more efficient than more complex algorithms. In the algorithm, the root process calls `isend_init` (corresponding to `MPI_Isend`) to queue a message for each of the other processes in the group. It then blocks, waiting for all the send operations to complete, using the `wait_all` call.

Starting a number of requests then waiting for them all to complete is a common technique used in MPI programs to perform multiple send or receive operations in parallel. This approach is equivalent to the CSP `Parallel` construct, which starts a set of parallel processes then waits for them to complete; the MPI pattern can be directly translated into a `Parallel`.

Listing 2 shows our PyCSP implementation of the same algorithm. The code uses Python list expressions to create a list of `pass_on` processes which are responsible for passing on the message to processes with rank `1` to `groupsize - 1`. The list of processes are then run in parallel using the `Parallel` construct[1].

```
@process
def pass_on(comm, targ_rank, data):
    comm.send_to(targ_rank, data)

def Broadcast_linear(comm, data):
    ''linear broadcast''
    if comm.rank == 0:
       Parallel(*[pass_on(comm, targ, data) \
                  for targ in range(1, comm.groupsize)])
    else:
       data = comm.recv()
    return data
```

Listing 2: Broadcast with linear algorithm

The code to call `pass_on` in parallel for several destinations will be used in several collective operations; we have separated it out into a helper function, `pass_on_multiple`, as shown in listing 3.

[1]The $*$ prefix in Python allows a list to be used directly as parameters to a function.

```
@process
def pass_on(comm, targ_rank, data):
    comm.send_to(targ_rank, data)

def pass_on_multiple(comm, targ_ranks, data):
    Parallel(*[pass_on(comm, targ, data) for targ in targ_ranks])

def Broadcast_linear(comm, data):
    ''linear broadcast''
    if comm.rank == 0:
       pass_on_multiple(comm, range(1, comm.groupsize), data)
    else:
       data = comm.recv()
    return data
```

Listing 3: Broadcast with linear algorithm. Separating out the Parallel construct.

One of the interesting lessons from this is that we can get some of the same benefits as asynchronous operations (communicating with multiple peers in parallel), without complicating the API by adding separate asynchronous versions of the operations.

2.2.1. Nonlinear Broadcast

For larger groups, the Broadcast operation uses a hierarchical approach with a binomial broadcast algorithm. The general idea is to arrange the ranks into a tree structure, where each internal node is responsible for forwarding a message to its children. The root process is the root of the tree.

```
def Broadcast_binomial(comm, data):
    ''linear broadcast''
    # binomial broadcast
    if comm.rank != 0:
       data = comm.recv() # read from parent in tree
    pass_on_multiple(comm, binom_get_children(comm.rank, comm.groupsize),
                     data)
    return data
```

Listing 4: Broadcast with non-linear algorithm

To perform a binomial broadcast, we use the helper function `binom_get_children` to get the ranks of our children, and then call `pass_on_multiple` to pass on the data to all our children in parallel. The OpenMPI implementation (Listing 19, in the appendix) interleaves the code to compute the tree structure with the communication code; the Python implementation is cleaner and more succinct. In addition, separating the tree-structuring code out into a function makes it possible to reuse it elsewhere, or to replace it with a different function to structure the tree in a different way, specified using the configuration system described in section 4.

2.3. Reduce

The Reduce operation combines the data produced by all the participating processes using a provided function on two operands; in functional terms, it performs a fold across all the results. Depending on how the reduction function is defined, the order in which the reductions occur can make a difference to the result; even a simple operation such as addition can produce different results for different orderings if it is acting on floating-point data. The simplest approach is for the root process to receive the results in order, performing a reduction step after each one; this is shown in listing 5. This approach will always produce a consistent result regardless of the function used.

```
def Reduce_linear(comm, data, func):
    if comm.rank == 0:
        # Start with the highest ranks, reducing towards the root.
        v = comm.recv_from(comm.groupsize - 1)
        for i in range(comm.groupsize - 2, 0, -1):
            t = comm.recv_from(i)
            v = func(t, v)
        v = func(data, v)
        return v
    else:
        # Non-root processes pass the values directly to the root
        comm.send_to(0, data)
        return data
```

Listing 5: Linear reduce

If the reduction function permits it, we can significantly increase the efficiency of the algorithm for larger groups by using a binomial tree. An equivalent of the OpenMPI binomial reduction is shown in listing 6, using the helper function `get_reduce_peers_binom` to compute the tree structure.

Again, the Python implementation is much more concise than OpenMPI's version; the OpenMPI implementation, which manages buffers at this level in the code, is 199 lines of C. The PyCSP collective operations avoid this duplication by managing buffers in the message transport layer.

```
def Reduce_binom1(comm, data, func):
    recv_from, send_to = get_reduce_peers_binom(comm)
    v = data
    for peer in recv_from:
        t = comm.recv_from(peer)
        v = func(v, t)
    for peer in send_to:
        comm.send_to(peer, v)
    return v
```

Listing 6: Logarithmic reduce similar to OpenMPI

The code in listing 6 allows processes executing on different nodes to communicate and reduce data concurrently, but expresses only a limited degree of parallelism.

If a process early in the list is late sending its response message, it will block reception of messages from processes later in the list. To reduce the effects of this, the message transport layer may buffer incoming messages. However, if the messages are large, buffer size limitations may cause messages to be blocked anyway.

A better solution is to express the desired parallelism explicitly, receiving from all processes in the list in parallel. With this approach, shown in listing 7, the message transport layer has enough information to schedule message reception in an appropriate order.

```
def Reduce_binom2(comm, data, func):
    # If communication overhead and timing is the issue, this one might
    # help at the potential cost of delaying the reduction until all
    # messages have arrived.
    recv_from, send_to = get_reduce_peers_binom(comm)
    v = data
    res = recv_from_multiple(comm, recv_from)
    for r in res:
        v = func(v, r)
    for peer in send_to:
        comm.send_to(peer, v)
    return v
```

Listing 7: Reduce using parallel receive

The drawback of that method is that we delay execution of the reduce function until we have all the messages; we cannot use the time that we spend waiting for messages to perform reductions on the messages we have received so far. We must also keep all the received messages in memory until the reduction is performed.

Listing 8 shows an alternative implementation that creates a chain of `_reduce_1` processes using the "recursive parallel" process-oriented pattern. Each process receives a message from its designated peer rank and another from the previous process in the chain, reduces the two, then passes the result on to the next process in the chain. This allows messages from all the peers to be received in parallel, while the reductions happen in as efficient a manner as possible. The code demonstrates another feature of the PyCSP `Parallel` construct: it returns a list of the return values from the PyCSP processes that were executing within that construct.

```
def Reduce_binom3(comm, data, func):
    """Recursive processes, calling 'func' on the current and next in the
    chain. Runs the recv operations in parallel, but 'func' will naturally
    be sequential since it depends on the result from the next stage in
    the chain.
    """
    @process
    def _reduce_1(comm, func, peers):
        if len(peers) > 1:
            res = Parallel(recv_from_other(comm, peers[0]),
                           _reduce_1(comm, func, peers[1:]))
            return func(res[0], res[1])
        else:
            return comm.recv_from(peers[0])

    recv_from, send_to = get_reduce_peers_binom(comm)
    v = data
    if len(recv_from) > 0:
        res = Sequence(_reduce_1(comm, func, recv_from))
        v = func(v, res[0])
    for peer in send_to:
        comm.send_to(peer, v)
    return v
```

Listing 8: Reduce using recursion to create a chain of reduce processes in each node

A further optimisation would be to execute the reduce operation out-of-order, as the messages arrive. We have not implemented this in time for this paper.

2.4. Barrier

```
def Barrier_non_par(comm):
    # Non-parallel version
    if comm.rank > 0:
        # non-root processes send to root and receive ack when all
        # have entered
        comm.send_to(0, None)
        comm.recv_from(0)
    else:
        # Root process collects from all children and broadcasts back
        # response when all have entered
        for i in range(1, comm.groupsize):
            comm.recv_from(i)
        for i in range(1, comm.groupsize):
            comm.send_to(i, None)
```

Listing 9: Linear sequential barrier

The Barrier operation stops processes from leaving the barrier until all processes in the group have entered. When the final process has entered, it releases all the other processes which can then leave the barrier.

Listing 9 shows an implementation of barrier that is equivalent to the OpenMPI linear barrier. Non-root processes send to the root process, and block waiting for a return message from the root process. The root process first receives from all the other processes (the enter phase), and then sends back a reply (the release phase).

An improvement of the above is to do all the send and receive operations in parallel, as in listing 10, but the advantage will probably be limited as the root process is still likely to be a communication bottleneck.

```
def Barrier_linear(comm):
    if comm.rank > 0:
        # non-root processes send and receive a token from the root process
        comm.send_to(0, None)
        comm.recv_from(0)
    else:
        # root process recvs from all other processes, then sends reply
        recv_from_multiple(comm, range(1, comm.groupsize))
        pass_on_multiple(comm, range(1, comm.groupsize), None)
```

Listing 10: Linear parallel barrier

OpenMPI provides a logarithmic barrier for larger groups. A PyCSP implementation based on the OpenMPI implementation is shown in listing 11.

The processes in the group are arranged in a tree. Internal nodes in the tree first receive messages from all of their children telling them that the children have all entered the barrier. The `binom_get_children` function returns the list of children for any node in the tree. For leaf nodes, the list will be empty.

After receiving all the messages, the internal non-root node sends and then receives from its parent. When it receives the "leave" message from the parent, it knows that the algorithm is in the release phase and can pass this on to the children. The root node only needs to receive from all its children in the tree to know that all processes have entered the barrier, and can subsequently send the release messages allowing the child processes to leave the barrier.

Compared to OpenMPI, the main difference is that we are doing parallel receives and sends when communicating with the children in the tree while OpenMPI receives and sends the messages in sequence. Given the message sizes and number of child processes for a typical barrier operation, this may not make as much of a difference as for other operations.

```
def Barrier_binom(comm):
    # binomial/log barrier
    # NB: OpenMPI uses root = rank0 and a similar tree to broadcast
    children = binom_get_children(comm.rank, comm.groupsize)

    recv_from_multiple(comm, children)
    if comm.rank != 0:
        # non-root need to communicate with their parents
        # (pass up, recv down)
        parent = binom_get_parent(comm.rank, comm.groupsize)
        comm.send_to(parent, None)
        comm.recv_from(parent)
    # send to children
    pass_on_multiple(comm, children, None)
```

Listing 11: Logarithmic barrier

2.5. Allreduce

```
def Allreduce(comm, data, func=lambda x,y: x+y):
    data = Reduce(comm, data, func)
    data = Broadcast(comm, data)
    return data
```

Listing 12: Allreduce

The Allreduce operation is a combination of the Reduce and Broadcast operations: it first calls Reduce to collect a value in the root process, and then calls Broadcast to distribute the collected value from the root process back out to all the other processes.

Listing 12 shows a PyCSP implementation that is equivalent to the OpenMPI implementation. The main weakness with this implementation is that it has two distinct phases, and a process that is finished with the Reduce phase is blocked waiting for the broadcasted value.

It is possible to merge the two phases by essentially running a number of concurrent Reduce operations rooted in each participating process or in subgroups within the larger group; this can decrease the time taken for the operation at the cost of increasing the number of messages sent. A potential problem is that for some functions and data types, we might end up with different results in the different processes, as they are performing Reduce operations with the intermediate results in different orders; this does not occur in the existing implementation, because the result is only computed by the root process.

2.6. Gather

The Gather operation gathers values from each process in the group and collects them in the root process. The PyCSP version returns a list of all the received values to the root process, and `None` to the non-root processes. OpenMPI uses an algorithm similar to listing 13, calling `receive` for each of the other processes in sequence.

```
def Gather_seq(comm, data):
    # No log-version of this one, as in OpenMPI
    if comm.rank != 0:
        # Non-root processes send to root
        comm.send_to(0, data)
        return None
    else:
        vals = [data]
        for i in range(1, comm.groupsize):
            vals.append(comm.recv_from(i))
        return vals
```

Listing 13: Sequential gather

For messages that are large enough, an implementation that throttles communication may stall unexpected messages from a sender that is ready while the reader is waiting for data from a sender that is not ready.

As with `pass_on_multiple`, we provide a `recv_from_multiple` helper function to receive data from several other ranks in parallel. This is similar to doing a sequence of `MPI_Irecv` operations followed by an `MPI_Waitall` to wait for them to complete.

The function returns the received messages in a list, and optionally inserts them into a provided dictionary, with the rank of the sending process used as a key. The dictionary representation is convenient for functions that want to reorder the results; see the final line of listing 14 for an example.

```
def Gather_par(comm, data):
    # Using a PAR to receive messages from the other processes
    if comm.rank != 0:
        # Non-root processes send to root
        comm.send_to(0, data)
        return None
    else:
        # Uses a parallel recv. This ensures that we don't stall a send
        # while waiting for a send from a slower process earlier in
        # the list.
        # Using a dict to return the values ensures that we can return a
        # correctly sorted list of the return values (sort on keys/sender
        # ranks even if we use a different order on the ranks to recv from).
        res = { 0 : data}
        recv_from_multiple(comm, range(1, comm.groupsize), res_dict = res)
        return [res[k] for k in range(comm.groupsize)]
```

Listing 14: Parallel gather

2.7. Allgather

```
def Allgather(comm, data):
    vals = Gather(comm, data)      # rank 0 gets all of them
    vals = Broadcast(comm, vals)  # rank 0 will forward all values
```

Listing 15: Allgather

The Allgather operation is a simple combination of the Gather and Broadcast operations. The root process first gathers data from all the other processes, then distributes all the gathered data to all the other processes.

A potential problem with this implementation is that some of the nodes may finish the gather phase early and sit idle waiting for the broadcast phase. To increase parallelism, we may try to use a an implementation that resembles the Alltoall operation below. Whether that will improve the performance of the operation will have to be experimentally validated.

2.8. Scatter

```
def Scatter_seq(comm, data):
    # Sequential implementation, as in OpenMPI basic
    if comm.rank != 0:
        return comm.recv_from(0)
    else:
        for i in range(1, comm.groupsize):
            comm.send_to(i, data[i])
        return data[0]

def Scatter_par(comm, data):
    if comm.rank != 0:
        return comm.recv_from(0)
    else:
        Parallel(*[pass_on(comm, i, data[i])
                   for i in range(1, comm.groupsize)])
        return data[0]
```

Listing 16: Scatter

The scatter operation is similar to the broadcast operation in that the root process sends data to each of the other processes. The difference is that scatter takes a sequence of values, partitions that up into `size` segments and sends one part to each of the processes. Upon completion, each process will have an entry corresponding to its own rank.

The Python implementation of scatter takes a list and sends entry `i` to the process with rank `i`. `Scatter_seq` in listing 16 is equivalent to OpenMPI's scatter implementation, sending a message to each of the other ranks in sequence.

The problem with this approach is that the runtime system has no idea about the next message in the sequence until the current send operation completes. Sending them in parallel allows the message transport layer to consider all of the messages for scheduling and sending when possible. `Scatter_par` is an alternative implementation which explicitly sends all messages in parallel.

2.9. Alltoall

```
def scatter_send(comm, data, targets):
    Parallel(*[pass_on(comm, i, data[i]) for i in targets])

@process
def scatter_send_p(comm, data, targets):
    scatter_send(comm, data, targets)

def Alltoall(comm, data):
    other = [i for i in range(comm.groupsize) if i != comm.rank]

    # OpenMPI version posts all send/recvs and then waits on all of them
    # This can be trivially implemented using one or more PARs
    res = { comm.rank : data[comm.rank]}
    Parallel(scatter_send_p(comm, data, other),
             recv_from_multiple_p(comm, other, res_dict=res))
    return [res[k] for k in sorted(res.keys())]
```

Listing 17: Alltoall

The Alltoall operation is similar to all the processes in the group taking turns being the root process in a Scatter operation. When the Alltoall operation completes, each process in the group has received a value from each of the other processes. The process with rank `i` will have a list of the `i`th entry that each process contributed. One way of viewing this operation is that it transposes the matrix formed by combining the lists contributed by each process.

The implementation in listing 17 works similarly to the OpenMPI implementation. OpenMPI uses non-blocking sends and receives, posting the receives first and then the sends, then waiting for all the operations to complete.

The PyCSP implementation uses a technique similar to the Scatter function for sending entries, but factors that out as a `scatter_send` function and a `scatter_send_p` process. This simplifies sending in parallel with receiving, and allows for reuse: `scatter_send` is also used in the Scatter operation.

PyCSP supports an alternative method of creating processes, using the Process class directly with a regular function. Using this, we could have written:

```
Parallel(Process(scatter_send, comm, data, other),...)
```

This avoids needing to define a separate `scatter_send_p` process. However, explicitly creating a process simplifies reuse, and we believe that the syntax in listing 17 is more readable.

3. Implementations

We have implemented collective operations using two process-oriented systems: PyCSP [3] and occam-π [5]. It is straightforward to implement the algorithms we have described using channels on a single host, but collective operations are most useful across clusters of ma-

chines. Both systems provide transparent networking facilities – PyCSP using the Pyro [12] remote objects framework, and occam-π using pony [13], an implementation of networked channel bundles.

However, with both Pyro and pony, the communication latency over a CSP channel that has been extended over a network is very high compared to local communication. This is because a channel communication must be acknowledged in order to provide the correct CSP blocking semantics, requiring one or more round trips across the network. However, we do not need full CSP channels for the collective operations we have described; asynchronous delivery of messages is sufficient. We can take advantage of this to remove the need for acknowledgements in our implementations of collective operations.

We have implemented an asynchronous network messaging system for both PyCSP and occam-π. The two implementations are broadly similar but have some important differences in their low-level implementation, since PyCSP maps processes to kernel-level threads, whereas occam-π has its own lightweight process scheduler. MPI implementations are typically single-threaded, so we incur some extra latency from context-switching in both implementations, but this is insignificant compared to the latency inherent in network communication; the smarter communication scheduling permitted by process-oriented implementations should more than make up for the difference, similar to the work we describe in section 4.

3.1. Interface

Our messaging system provides facilities similar to Erlang [14] or MPI's low-level messaging system: a system consists of several *nodes* (equivalent to MPI ranks), each of which has zero or more receiving *ports*. Messages are delivered asynchronously from a node to any port in the system. In process-oriented terms, ports can be considered to be infinitely-buffered shared channels. A receiving process may choose to receive from one port or several; this makes it possible to selectively wait for messages from a particular set of other processes.

We aim to provide a reasonably natural interface for the process-oriented programmer, allowing them to mix asynchronous communications with regular local communications and timeouts in alternation. The programmer uses a client-server interface to interact with a *communicator* process on each node, which supports *send* and *receive* operations. In PyCSP, *receive* simply blocks until an appropriate message is received; in occam-π, *receive* immediately returns a mobile channel down which received messages will be delivered.

3.2. Network Communications

Network communications are implemented using TCP sockets, with one socket used between each pair of communicating nodes, carrying messages for multiple ports in both directions. At the network level, messages are just sequences of bytes with a simple header giving their length and addressing information. It is up to the application to serialise messages for network communication; this can be done simply using Python's `pickle` module, or occam-π's `RETYPES` mechanism.

To discover the network addresses of other nodes, we need a name server mechanism. Nameserver lookups are not performance-critical, so this can most easily be implemented on top of an existing network mechanism such as Pyro (as has been done for PyCSP) or pony; alternatively, a second TCP protocol can be used (as in the occam-π implementation). The nameserver is generally a long-running service that handles multiple applications, so it must cope gracefully with nodes connecting and disconnecting at any time.

Since the protocol is simple, it would be straightforward to make our two implementations interoperate, provided they agree on the contents of the messages being sent. The choice of TCP rather than UDP for asynchronous messaging may seem surprising, but we wanted to be able to use our system over unreliable networks without needing to worry about retrans-

mitting lost packets. In a busy system, TCP piggyback acknowledgements will largely negate the need for TCP-level acknowledgement packets anyway.

3.3. IO Scheduling

Each node has several TCP sockets open to other nodes. In a large cluster, this could potentially be a very large number of sockets; it is necessary to manage communications across the collection of sockets in an efficient manner.

At the lowest level, communication is performed on TCP sockets using system calls to read and write data, which may block if the socket's buffer is full. This is a problem for a system with lightweight processes, since blocking one kernel-level thread could result in multiple lightweight processes being blocked. Sockets may be placed into non-blocking mode, which causes the system calls to return immediately with an error code rather than blocking, but system calls are relatively expensive; we cannot afford to repeat a call until it succeeds.

The approach currently used in occam-π is to use a separate kernel-level thread for each operation that may block [15]. This is rather inefficient, in that we must have one thread running for each blocked operation, and each blocking call requires a context switch to a worker thread, followed by another back again when the call is complete. Furthermore, while PyCSP does not yet make use of lightweight processes, it is still inefficient to have very large numbers of threads performing parallel system calls.

Programs that communicate using large numbers of sockets are generally better written in an event-driven style, using the POSIX `poll` or `select` system calls to efficiently wait for any of a set of non-blocking sockets to become ready [16]. When a socket is ready, the next `read` or `write` call on it will not block. (`poll` is poorly-named; it is simply a more efficient version of `select`, and blocks in the kernel rather than actually polling.)

As with the Haskell I/O multiplexing system [17], we provide an *IO scheduler* process. This process maintains a set of open sockets, and repeatedly calls `poll` across the complete set on the user's behalf. When a socket becomes ready, the IO scheduler wakes up an appropriate user process to perform the IO, by communicating down a channel. It then waits for confirmation that the IO has been completed, and returns to the `poll` loop. The user process may opt to keep the socket in the set (if it expects to perform more IO in the future), or to remove it.

The user may add new sockets to the set at any time. This is achieved using a standard trick [18]: in addition to the sockets, the `poll` also waits on the reading end of a pipe. Writing a character to the pipe will interrupt the `poll`, allowing it to handle the user's request.

All processes on a node share the same IO scheduler. The result is that processes using sockets may be written in a conventional process-oriented manner, with a process to handle each individual socket. Processes may block upon socket IO without needing to worry about accidentally blocking other processes or incurring the overheads of using a separate thread. They may even use alternation to wait for any of a set of sockets to become ready. However, the underlying mechanism used to implement communications is the more efficient event-based model supported by poll, which means that very large numbers of sockets can be handled with minimal performance overheads. We have provided these facilities with no modification of the runtime system or operating system kernel.

Again, we have implemented this for both PyCSP and occam-π. The two implementations are very similar; the only difference is that the PyCSP IO scheduler can perform IO itself on behalf of user processes, in order to reduce the number of kernel-level thread context switches necessary.

3.4. Messaging System

Communications for each node are coordinated by a communicator process, which is itself a group of parallel processes.

For each TCP connection to another node, there is an *inbound* process that handles received messages, and an *outbound* process that handles messages to be sent. Both maintain an internal buffer to avoid blocking communications. The inbound process uses the IO scheduler to wait for the socket to become readable, and the outbound process uses the IO scheduler to wait for it to become writable (when it has messages to send).

For the occam-π implementation, the buffers are simple FIFO arrays of bytes, and the inbound messages are delivered to the switch process in strict order of reception. For PyCSP, the inbound buffer is slightly more complex: it is a queue from which messages may be removed at arbitrary points, so that the receive operation may pull out only the messages it is interested in.

A central *switch* process handles user requests, and messages received from inbound processes. It routes incoming and outgoing messages to the correct process. New TCP connections (and their corresponding inbound and outbound processes) are created when the first message is sent, or when an incoming TCP connection is received, the latter being handled by a separate *listener* process.

4. Configuration and Mapping of Operations and Algorithms

Previous experience with the PATHS [9,10,11] and CoMPI [6] systems has shown that changing the algorithms, mapping and communication patterns can significantly improve the performance of operations compared to the standard implementations in LAM-MPI [19]. Similar experiences have led to efforts such as MagPIe [20] and Compiled Communication [21].

The approach taken by PATHS and CoMPI differs in that the configurations are built at runtime, using a specification which can be retained to help analysis of the system afterwards. This simplifies analysis and comparison of several different configurations as both the specifications and the trace data can be compared and studied. Tools and systems for doing this include STEPS [22], EventSpace [23] and the PATHS visualiser and animation tool for display walls [24].

One of the drawbacks of the current implementations of the PATHS system is that the concurrency facilities available are very low-level: threads, mutexes and condition variables. This is one area where CSP-based languages and systems may provide an improvement compared to the current PATHS implementations.

4.1. PATHS

The PATHS system is based around *wrappers*; a wrapper encapsulates an existing object. A thread can call methods in the original object by having a reference to the wrapper. The wrapper usually forwards the call, but can optionally perform extra computations or modify the behaviour of the object's methods. For example, a wrapper could be used to provide tracing or caching of method calls on the object.

Several wrappers can be stacked on top of each other to create a *path* to the object (similar to a call path or stack trace). This means that several different types of wrappers can be combined to provide extra functionality. The call path is built using a *path specification*, which is a list of the wrappers that a call goes through along with the parameters used to instantiate each wrapper.

The system allows several different call paths to be specified to reach the same object, allowing a thread to use multiple paths to the same object, or more commonly, to allow

multiple threads to access the same object through separate paths[2]. Paths can be combined, allowing the system to aggregate calls (especially useful for caching).

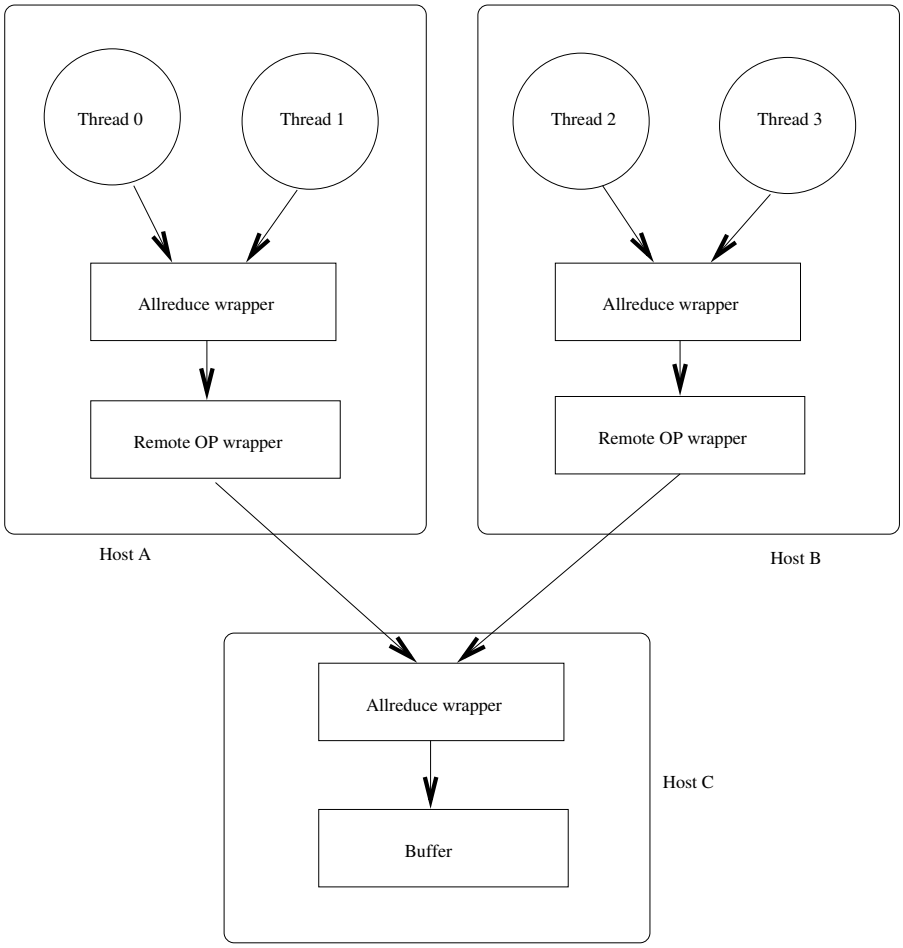

Figure 2 Example configuration of an Allreduce operation using the PATHS system

Figure 2 shows an example configuration of an Allreduce operation, using *Allreduce* wrappers. When several threads share an Allreduce wrapper, the first $n-1$ threads contribute their values and block, waiting for the result. Thread number n contributes the last value at that stage and forwards the partial result to the next wrapper in the path. When all results have been combined in the root object, the result is propagated back up the tree, waking up blocked threads.

Remote operation wrappers are used to implement remote method calls, allowing a path to jump to another computer. The parameters to the remote operation wrapper include information such as the host name and protocol to use when connecting to the other host. This information is also useful when analysing the running system, as it provides the necessary information to determine exactly where each wrapper was located.

The combination of all path specifications is called a *path map*, and describes how a system is mapped onto a cluster as well as how an algorithm is implemented by combining wrappers. *Trace wrappers*, which can be added to the path descriptions, can log timestamps and optionally data going through the wrapper. By combining the traces with the path map, we can analyse the system's performance and identify problems such as bottlenecks.

The CoMPI [6] system is an implementation of the PATHS concepts in LAM-MPI.

[2]Multiple threads can also share the same path

4.2. Towards CSP-Based Wrappers

One of the motivations for this paper is to provide background and tools for a CSP-based version of the PATHS system. As this part of the project is still in the early phases, we will just give a quick overview of a few important points.

A PATHS wrapper is an object that encapsulates some behaviour, can contain some state, and that may forward or initiate communication with other wrappers and objects. This corresponds well with the notion of a process in CSP-based systems.

Setting up a path and binding the wrappers together is similar to activity common in CSP-based programs. In PATHS, a process asks for a path to a given object by calling a `get_path` function. `get_path` retrieves a path specification, either from a library that can be loaded at run-time (usually Python modules or Lisp code), or by querying a central path specification server.

The specification includes all the necessary information to instantiate the wrappers on the hosts in the system (or locate wrappers that have already been instantiated) and bind the wrappers together. This corresponds well with spawning CSP processes and plugging them together using channels.

5. Conclusion

We have described how a number of the MPI collective operations can be implemented using PyCSP, showing how MPI implementation idioms can be translated into process-oriented equivalents. The same techniques can be used to implement collective operations using other process-oriented systems such as occam-π. We find that implementing collective operations using a system with high-level concurrency facilities typically results in more concise, clearer code.

We have shown how process-oriented techniques can be used to express a greater level of concurrency in our implementations of the collective operations than in OpenMPI's implementations. In particular, our implementations generally allow more communications to proceed in parallel, potentially reducing communication bottlenecks and thus making more effective use of the cluster communication fabric.

The basis of process-oriented systems in process calculi makes it possible to formally reason about the behaviour of collective operations. For example, we will be able to ensure that new cluster communication strategies are free from deadlock.

In order to support future experiments with collective operations, we have built an asynchronous network messaging framework for both occam-π and PyCSP. We have demonstrated how an IO scheduler process can be used to efficiently handle large numbers of network connections in a process-oriented program without the latency problems inherent in existing approaches.

This framework will have other uses as well; we have already used it for a distributed complex systems simulation, using a two-tiered strategy with channel communication between process on a single and asynchronous network messaging between hosts. In the future, we would like to make its use more transparent, supporting a direct channel interface and channel end mobility; we would also like to increase the interoperability of the occam-π and PyCSP implementations, and extend this to other process-oriented systems such as JCSP, in order to support distributed multi-language systems.

We have described how systems using process-oriented collective operations may be dynamically reconfigured at runtime using an approach based on the PATHS system. We have shown a straightforward mapping between PATHS' object-oriented wrappers and processes in a process-oriented system.

We have not yet extensively benchmarked our implementations of collective operations. While the expressiveness and correctness of the process-oriented implementations are clear advantages, we still need to measure their performance in comparison with the existing Open-MPI implementations in medium- and large-scale cluster applications. We are optimistic about the potential performance gains from the application of process-oriented techniques to cluster computing based on earlier experiences with the PATHS system.

Acknowledgements

This work was partially funded by EPSRC grant EP/E053505/1 and a sabbatical stipend from the University of Tromsø.

References

[1] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[2] OpenMPI homepage. `http://www.open-mpi.org/`.

[3] John Markus Bjørndalen, Brian Vinter, and Otto Anshus. PyCSP - Communicating Sequential Processes for Python. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, jul 2007.

[4] PyCSP distribution. `http://www.cs.uit.no/~johnm/code/PyCSP/`.

[5] The occam-π programming language. `http://occam-pi.org/`.

[6] Espen Skjelnes Johnsen, John Markus Bjørndalen, and Otto Anshus. CoMPI - Configuration of Collective Operations in LAM/MPI using the Scheme Programming Language. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 189–197. Springer, 2007.

[7] John Markus Bjørndalen, Otto Anshus, Brian Vinter, and Tore Larsen. Configurable Collective Communication in LAM-MPI. *Proceedings of Communicating Process Architectures 2002, Reading, UK*, September 2002.

[8] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. `http://www.lam-mpi.org/`, 1994.

[9] John Markus Bjørndalen, Otto Anshus, Tore Larsen, and Brian Vinter. PATHS - Integrating the Principles of Method-Combination and Remote Procedure Calls for Run-Time Configuration and Tuning of High-Performance Distributed Application. In *Norsk Informatikk Konferanse*, pages 164–175, November 2001.

[10] John Markus Bjørndalen, Otto Anshus, Tore Larsen, Lars Ailo Bongo, and Brian Vinter. Scalable Processing and Communication Performance in a Multi-Media Related Context. *Euromicro 2002, Dortmund, Germany*, September 2002.

[11] John Markus Bjørndalen. *Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters*. PhD thesis, Department of Computer Science, University of Tromsø, 2003.

[12] Irmen de Jong. Pyro: Python remote objects. `http://pyro.sourceforge.net/`.

[13] M. Schweigler. *A Unified Model for Inter- and Intra-processor Concurrency*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, UK, August 2006.

[14] Jonas Barklund and Robert Virding. Erlang 4.7.3 Reference Manual, February 1999. `http://www.erlang.org/download/erl_spec47.ps.gz`.

[15] F.R.M. Barnes. Blocking System Calls in KRoC/Linux. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[16] Dan Kegel. The C10K problem, Sep 2006. `http://www.kegel.com/c10k.html`.

[17] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Extending the Haskell foreign function interface with concurrency. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 22–32, New York, NY, USA, 2004. ACM.

[18] D.J. Bernstein. The self-pipe trick. `http://cr.yp.to/docs/selfpipe.html`.

[19] LAM-MPI homepage. `http://www.lam-mpi.org/`.

[20] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140. ACM Press, 1999.

[21] Amit Karwande, Xin Yuan, and David K. Lowenthal. CC-MPI: a compiled communication capable MPI prototype for Ethernet switched clusters. In *Proc. of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 95–106. ACM Press, 2003.

[22] Lars Ailo Bongo. Steps: A Performance Monitoring and Visualization Tool for Multicluster Parallel Programs, June 2002. Large term project, Department of Computer Science, University of Tromsø.

[23] Lars Ailo Bongo, Otto Anshus, and John Markus Bjørndalen. EventSpace - Exposing and observing communication behavior of parallel cluster applications. In *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 47–56. Springer, 2003.

[24] Tor-Magne Stien Hagen. The PATHS Visualizer. Master's thesis, Department of Computer Science, University of Tromsø, 2006.

Appendix: OpenMPI Source Code

```
/*
 *      bcast_lin_intra
 *
 *      Function:       - broadcast using O(N) algorithm
 *      Accepts:        - same arguments as MPI_Bcast()
 *      Returns:        - MPI_SUCCESS or error code
 */
int
mca_coll_basic_bcast_lin_intra(void *buff, int count,
                               struct ompi_datatype_t *datatype, int root,
                               struct ompi_communicator_t *comm)
{
    int i;
    int size;
    int rank;
    int err;
    ompi_request_t **preq;
    ompi_request_t **reqs = comm->c_coll_basic_data->mccb_reqs;

    size = ompi_comm_size(comm);
    rank = ompi_comm_rank(comm);

    /* Non-root receive the data. */

    if (rank != root) {
        return MCA_PML_CALL(recv(buff, count, datatype, root,
                                 MCA_COLL_BASE_TAG_BCAST, comm,
                                 MPI_STATUS_IGNORE));
    }

    /* Root sends data to all others. */

    for (i = 0, preq = reqs; i < size; ++i) {
        if (i == rank) {
            continue;
        }

        err = MCA_PML_CALL(isend_init(buff, count, datatype, i,
                                      MCA_COLL_BASE_TAG_BCAST,
                                      MCA_PML_BASE_SEND_STANDARD,
                                      comm, preq++));
        if (MPI_SUCCESS != err) {
            return err;
        }
    }
    --i;

    /* Start your engines.  This will never return an error. */
```

```
    MCA_PML_CALL(start(i, reqs));

    /* Wait for them all.  If there's an error, note that we don't
     * care what the error was -- just that there *was* an error.  The
     * PML will finish all requests, even if one or more of them fail.
     * i.e., by the end of this call, all the requests are free-able.
     * So free them anyway -- even if there was an error, and return
     * the error after we free everything. */

    err = ompi_request_wait_all(i, reqs, MPI_STATUSES_IGNORE);

    /* Free the reqs */

    mca_coll_basic_free_reqs(reqs, i);

    /* All done */

    return err;
}
```

Listing 18: Linear Broadcast in OpenMPI 1.2.5

```
int
mca_coll_basic_bcast_log_intra(void *buff, int count,
                               struct ompi_datatype_t *datatype, int root,
                               struct ompi_communicator_t *comm)
{
    int i;
    int size;
    int rank;
    int vrank;
    int peer;
    int dim;
    int hibit;
    int mask;
    int err;
    int nreqs;
    ompi_request_t **preq;
    ompi_request_t **reqs = comm->c_coll_basic_data->mccb_reqs;

    size = ompi_comm_size(comm);
    rank = ompi_comm_rank(comm);
    vrank = (rank + size - root) % size;

    dim = comm->c_cube_dim;
    hibit = opal_hibit(vrank, dim);
    --dim;

    /* Receive data from parent in the tree. */

    if (vrank > 0) {
        peer = ((vrank & ~(1 << hibit)) + root) % size;

        err = MCA_PML_CALL(recv(buff, count, datatype, peer,
                                MCA_COLL_BASE_TAG_BCAST,
                                comm, MPI_STATUS_IGNORE));
        if (MPI_SUCCESS != err) {
            return err;
        }
    }

    /* Send data to the children. */

    err = MPI_SUCCESS;
    preq = reqs;
    nreqs = 0;
    for (i = hibit + 1, mask = 1 << i; i <= dim; ++i, mask <<= 1) {
        peer = vrank | mask;
```

```
            if (peer < size) {
                peer = (peer + root) % size;
                ++nreqs;

                err = MCA_PML_CALL(isend_init(buff, count, datatype, peer,
                                              MCA_COLL_BASE_TAG_BCAST,
                                              MCA_PML_BASE_SEND_STANDARD,
                                              comm, preq++));
                if (MPI_SUCCESS != err) {
                    mca_coll_basic_free_reqs(reqs, preq - reqs);
                    return err;
                }
            }
        }

        /* Start and wait on all requests. */

        if (nreqs > 0) {

            /* Start your engines.  This will never return an error. */

            MCA_PML_CALL(start(nreqs, reqs));

            /* Wait for them all.  If there's an error, note that we don't
             * care what the error was -- just that there *was* an error.
             * The PML will finish all requests, even if one or more of them
             * fail.  i.e., by the end of this call, all the requests are
             * free-able.  So free them anyway -- even if there was an
             * error, and return the error after we free everything. */

            err = ompi_request_wait_all(nreqs, reqs, MPI_STATUSES_IGNORE);

            /* Free the reqs */

            mca_coll_basic_free_reqs(reqs, nreqs);
        }

        /* All done */

        return err;
    }
```

Listing 19: Non-linear Broadcast in OpenMPI 1.2.5.