THE COMPONENT ARCHITECTURE OF OPEN MPI: ENABLING THIRD-PARTY COLLECTIVE ALGORITHMS*

Jeffrey M. Squyres and Andrew Lumsdaine
Open Systems Laboratory, Indiana University
Bloomington, Indiana, USA
jsquyres@open-mpi.org
lums@open-mpi.org

Abstract   As large-scale clusters become more distributed and heterogeneous, significant research interest has emerged in optimizing MPI collective operations because of the performance gains that can be realized. However, researchers wishing to develop new algorithms for MPI collective operations are typically faced with significant design, implementation, and logistical challenges. To address a number of needs in the MPI research community, Open MPI has been developed, a new MPI-2 implementation centered around a lightweight component architecture that provides a set of component frameworks for realizing collective algorithms, point-to-point communication, and other aspects of MPI implementations. In this paper, we focus on the collective algorithm component framework. The "coll" framework provides tools for researchers to easily design, implement, and experiment with new collective algorithms in the context of a production-quality MPI. Performance results with basic collective operations demonstrate that the component architecture of Open MPI does not introduce any performance penalty.

Keywords: MPI implementation, Parallel computing, Component architecture, Collective algorithms, High performance

1.   Introduction

Although the performance of the MPI collective operations [6, 17] can be a large factor in the overall run-time of a parallel application, their optimization has not necessarily been a focus in some MPI implementations until re-

cently [22]. MPI collectives are only a small portion of a production-quality, compliant implementation of MPI; implementors tend to give a higher priority to reliable basic functionality of all parts of MPI before spending time tuning and optimizing the performance of smaller sub-systems.

As a direct result, the MPI community has undertaken active research and development of optimized collective algorithms. Although design and theoretical verification is the fundamental basis of a new collective algorithm, it must also be implemented and used in both benchmark and real-world applications (potentially in a variety of different run-time / networking environments) before its performance can be fully understood. The full cycle of design, development, and experimental testing allows the refinement of algorithms that is not possible when any of the individual steps are skipped.

1.1 Solution Space

Much research has been conducted in the area of optimized collective operations resulting in a wide variety of different algorithms and technologies. The solution space is vast; determining which collective algorithms to use in a given application may depend on multiple factors, including the communication patterns of the application, the underlying network topology, and the amount of data being transferred. Hence, one set of collective algorithms is typically not sufficient for all possible application / run-time environment combinations. This is evident in the range of literature available on different algorithms for implementing the MPI collective function semantics.

It is therefore useful to allow applications to select at run-time which algorithms are used from a pool of available choices. Because each communicator may represent a different underlying network topology, algorithm selection should be performed on a per-communicator basis. This implies that the MPI implementation both includes multiple algorithms for the MPI collectives and provides a selection mechanism for choosing which routines to use at run-time.

1.2 Implementation Difficulties

There are significant barriers to entry for third-party researchers when implementing new collective algorithms. For example, many practical issues arise when testing new algorithms with a wide variety of MPI applications in a large number of run-time environments. To both ease testing efforts and to make the testing environment as uniform as possible, MPI test applications should be able to utilize the new algorithms with no source code changes. This will even allow real world MPI applications to be used for testing purposes; the output and performance from previous runs (using known correct collective algorithms) can be compared against the output when using the collective algorithms under test.

This means that functions implementing new algorithms must use the standard MPI API function names (e.g., `MPI_Barrier`). Techniques exist for this kind of implementation, but they may involve significant learning curves for the researcher with respect to the underlying MPI implementation: how it builds, where the collective algorithms are located in the source tree, internal restrictions and implementation models for the collective functions, etc.

1.3   A New Approach

To address a number of needs in the MPI research community, Open MPI [5]has been developed; a new MPI-2 implementation based upon the collected research and prior implementations of FT-MPI [3–4] from the University of Tennessee, LA-MPI [1, 7] from Los Alamos National Laboratory, and LAM/MPI [2, 19] from Indiana University. Open MPI is centered around a lightweight component architecture that provides a set of component frameworks for realizing collective algorithms, point-to-point communication, and other aspects of MPI implementations.

In this paper, we focus on the collective algorithm component framework. The "coll" framework provides tools for researchers to easily design, implement, and experiment with new collective algorithms in the context of a production-quality MPI. Collective routines are implemented in standalone components that are recognized by the MPI implementation at run-time. The learning curve required to create new components is deliberately small to allow researchers to focus on their algorithms, not the details of the MPI implementation. The framework also offers other benefits: source and binary distribution of components, seamless integration of all algorithms at compile and/or run-time, and fine-grained run-time selection (on a per-communicator basis).

This paper is therefore not about specific collective algorithms, but rather about providing a comprehensive framework for researchers to easily design, implement, and experiment with new collective algorithms. Components containing new algorithms can be distributed to users for additional testing, verification, and finally, production usage.

Both MPICH and MPICH2 [8–9] use sets of function pointers (to varying degrees) on communicators to effect some degree of modularity, but have no automatic selection or assignment mechanisms, therefore requiring abstraction violations (the user application has to assign function pointers inside an opaque MPI communicator) or manual modification of MPICH itself.

LAM/MPI v7 debuted the first fully-integrated component-based framework that allowed source and binary distribution of several types of components (including collective algorithms) while requiring no abstraction violations or source code changes to the MPI implementation in a production-quality, open-source MPI implementation. Open MPI evolves these abstractions by refining

the concepts introduced in LAM/MPI v7, essentially creating a second generation set of component frameworks for MPI implementations called the MPI Component Architecture (MCA) [5, 23]. This paper presents Open MPI's MCA collective component framework design.

The rest of this paper is organized as follows. §1.2 discusses the current state of the art with regards to implementing third-party collective algorithms within an MPI framework. §1.3 describes Open MPI's component model for collective algorithms, and explores different possibilities for third-party implementations. §1.4 provides overviews of two collective modules that are included in the Open MPI software distribution. Finally, §1.5 and §1.6 discuss run-time performance, final conclusions, and future work directions.

2. Adding Collective Algorithms to an MPI Implementation

Third-parties implementing new collective functions can encounter both technical and logistical difficulties, even in MPI implementations that encapsulate collective function pointers in centralized locations. Not only is it desirable for MPI applications to invoke new collective routines through the standard MPI API, there must be a relatively straightforward mechanism for making the new routines available to other users (download, compile, install, compile / link against user applications, etc.).

2.1 Common Interface Approaches

Common approaches to developing new collective routines include: using the MPI profiling layer, editing an existing MPI implementation, creating a new MPI implementation, and using alternate function names. Each of these scenarios have benefits and drawbacks, but all require the collective algorithm author to implement at least some level of infrastructure to be able to invoke their functions.

Use the MPI Profiling Layer. The MPI profiling layer was designed for exactly this purpose: allowing third-party libraries to insert arbitrary functionality in an MPI implementation. This can be done without access to the source code for either the MPI implementation or the MPI application.

This approach has the obvious advantage that any MPI application will automatically use the new collective routines without modifications. Although the MPI application will need to be relinked against the new library, no source code changes should be necessary. A non-obvious disadvantage is that since the profiling layer uses linker semantics to overload functions, only one version of an overloaded function is possible. For example, MPL_BARRIER cannot be over-

loaded with both a new collective routine *and* a run-time debugging/profiling interface.

Edit an Existing MPI Implementation. This method involves editing an MPI implementation to either include new collective routines in addition to the implementation's existing routines [21–22], or outright replacing the implementation's collective routines with new versions [10]. This can only be used with MPI implementations where the source code is available and the license allows such modifications.

Similar to the profiling approach, this method allows unmodified MPI applications to utilize new functionality. This is perhaps the easiest method for MPI applications because the API is the same and the new routines are in the MPI implementation itself.

However, the learning curve to add or replace functionality in the MPI implementation may be quite large. Additionally, editing the underlying MPI effectively creates a "fork" in the implementation's development path. This may make the code difficult to maintain and upgrade.

Create a New MPI Implementation. Entirely new MPI implementations have been created simply to design, test, and implement new MPI collective algorithms [11–12]. The advantage to this approach is complete control over the entire MPI implementation. This may be desirable for situations where the collective routines are radically different than current MPI implementations allow. For example, PAC-X MPI was created to enable communications in metacomputing environments, requiring alternate collective algorithms for efficiency.

The overhead with this approach is enormous. Writing enough of an MPI implementation such that a simple MPI program that only invokes MPI_INIT, MPI_COMM_RANK, MPI_COMM_SIZE, and MPI_FINALIZE is a monumental task. The time necessary to create an entire MPI framework before actually being able to work on collective algorithms can be prohibitively large.

Use Alternate Function Names. Perhaps the simplest approach from the algorithm implementor's perspective is to use function names other than the ones mandated by the MPI standard. For example, provide an alternate barrier implementation in the function `New_Barrier` instead of `MPI_Barrier`.

Difficulties arise in testing because MPI applications need to be modified to call the alternate functions. This can be as simple as preprocessor macros in a standardized header file, or may entail manually modifying all invocation points in the application. Requiring source code modification necessarily means that precompiled, binary-only MPI applications will not be able to utilize the new functionality.

2.2   A Component-Based Approach

We propose an open, component-based framework for the implementation of collective algorithms that will solve many of the technical and logistical issues faced by third-party collective algorithm researchers. In this framework, a *collective component* is comprised of a set of top-level *collective routines*. A collective routine implements one MPI collective function (such as MPI_BARRIER, MPI_BCAST, etc.). The framework also includes built-in mechanisms for configuration, compilation, installation, and source and binary distributions of components.

The collective component framework was designed and implemented with the following goals:

- Do not require modifying Open MPI source code to import new collective algorithms.

- Allow new collectives to be imported into the MPI implementation at compile- and run-time.

- Provide easy-to-understand interface and implementation models for collective routines that do not require detailed internal knowledge of the MPI implementation.

- Provide minimal overhead before invoking collective routines to maximize run-time performance.

- Allow (but not require) collective routines to be layered upon MPI point-to-point routines.

- Allow collective routines to exploit back-end hardware and network topologies.

- Allow collective components to be layered upon other collective components.

- Facilitate both source and binary distribution of collective components.

- Enable MPI applications to utilize the new collective components without recompiling / relinking.

- Allow multiple collective components to exist within a single MPI process.

- Provide a fine-grained, run-time, user-controlled component selection mechanism.

There are no current plans to allow experimentation with collective algorithms that are not specified by MPI.

3.   Collective Components

Open MPI is based upon a lightweight component architecture, including a component framework for MPI collective algorithms named "coll." The coll component interface was designed to satisfy the goals listed in §1.2.2. coll components can be loaded and selected at compile-time or run-time. For example, multiple coll components are included in the standard Open MPI distribution, but third-party components can also be added at any time.

3.1   Design Overview

The Open MPI component framework manages all coll components that are available at run-time. This management code is typically referred to as the Open MPI coll framework in the discussion below.

Simply put, a coll components is essentially a list of top-level function pointers that the Open MPI infrastructure selectively invokes upon demand. When paired with a communicator, a component becomes a *module* [20]. Top-level MPI collective functions have been reduced to thin wrappers that perform error checking before invoking back-end coll module implementation functions. One coll module is assigned to each communicator; this module is used to implement all MPI collectives that are invoked on that communicator. For example, MPI_BCAST simply checks the passed parameters for errors and then invokes the back-end broadcast function on its assigned coll module.

3.2   Implementation Models

Components are free to implement the standardized MPI semantics in any way that they choose. Most, however, use one or more of the following models: layered over point-to-point, alternate communication channels, or layered over another coll components.

Layered over Point-to-Point.   A simple implementation model is to utilize MPI point-to-point functions to send data between processes. For example, using MPI_SEND and MPI_RECV to exchange data is both natural and easy to understand, freeing the coll component author to concentrate on the components' algorithms and remain independent of how the underlying communication occurs. This model has been used extensively by MPI implementations [8, 19] and third-party collective algorithm researchers [13–14].

Alternate Communication Channels.   Recently, researchers have been exploring the possibility of avoiding MPI point-to-point functionality and instead using alternate communication channels for collective communications. Some network interfaces contain native primitives for collective operations and/or streamlined one-sided operations which can lead to significant performance
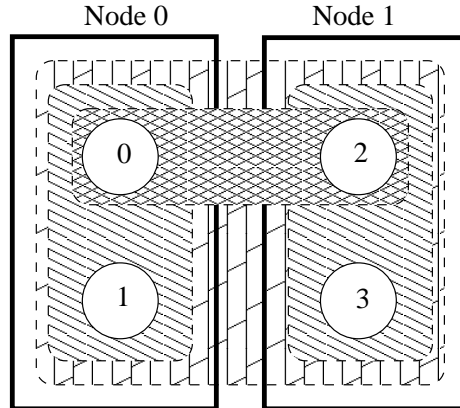
Node 0   Node 1

Figure 1. Four processes in MPI_COMM_WORLD are distributed across two nodes. Three sub-communicators (vertical and horizontal) each contain the two processes local to their respective nodes. One "bridge" communicator (horizontal) contains a representative process from each node.

gains as compared to using traditional point-to-point methods. Examples of alternate communication channels that at least partially support collective operations include (but are not limited to): shared memory [16], UDP multicast [11], Myrinet [24], and Infiniband [15].

Hierarchical coll Components. The coll framework was carefully designed such that coll components can be re-used at run-time in two ways. First, the coll component "basic," as its name implies, is a basic implementation of all of the MPI collectives. It can be used with any communicator and topology. The purpose of this component is to provide a baseline implementation for all MPI collective operations, allowing other components to use its routines as necessary. For example, a component that only provides an optimized scatter algorithm implementation can complete itself by using the methods from the basic component (or other components) for all other collective routines. This allows the optimized scatter component to be used in any MPI program even though it only implements a small number of new/optimized routines.

A second, more complex model involves using a hierarchy of coll modules to implement a single, top-level MPI collective. This is useful when a collective is invoked on a communicator that spans multiple types of networks. For example, Figure 1 shows two SMPs, each running two MPI processes. A single MPI communicator contains all four processes. The top-level communicator's coll module creates three sub-communicators: one for each SMP (containing the two processes on each node), and a third "bridge" communicator connecting one representative process from each node.
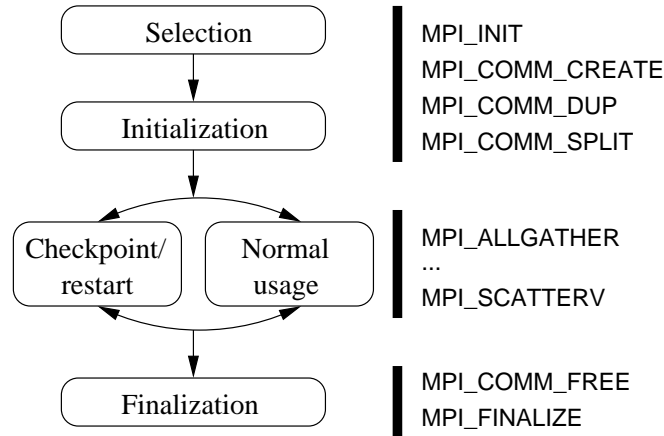
Figure 2. Five phases in the life of a coll component. The component is selected and initialized when a communicator is created. It is used and/or checkpointed during the run, and finalized when the communicator is destroyed.

Note that each sub-communicator will have its own coll module. This hierarchical arrangement of communicators allows each network to utilize its own optimized coll component, resulting in an efficient movement of data across each medium. This model will be explained in more detail in §1.4, where the smp coll component is discussed as an example implementation.

3.3   Component / Module Life Cycle

There are five phases in a coll component's life cycle: selection, initialization, checkpoint / restart, normal operation, and finalization. Figure 2 shows these phases and the corresponding MPI functions that trigger them. Note that a component may be involved in multiple life cycles simultaneously (i.e., several modules of the same component may exist in a single process); coll components have a one-to-many relationship with communicators.

Selection.   As each communicator is created (including MPI_COMM_SELF and MPI_COMM_WORLD), a coll component is *selected* for it from all available components. Specifically, the Open MPI coll framework queries each available coll component to determine if it is available to be used with the newly-created communicator. The queried component analyzes factors such as the current run-time environment and topology of the processes in the communicator. If the component determines that its algorithms are a good match for the target communicator, it returns priority value (from 0 to 100) intended as a relative indicator of the component's expected performance. The priority value

is relative and changable at run-time. Hence, components typically provide default priority values that is a guesstimate (e.g., MagPIe-based algorithms across WANs should return a high priority – it doesn't matter what the priority is, as long as it is higher than the rest). Users can change default priorities to force selection of specific components based on their environment. The component returning the highest priority is *selected*; all MPI collective functions invoked on that communicator will use the selected module.

Initialization. Once a coll module is selected for a given communicator, it is *initialized*. Specifically, the component's initialization function is invoked, passing the target communicator as an argument. The initialization function performs any one-time setup required by the module, and returns a module that contains any local state required to perform collectives on the target communicator. By definition, a communicator's member processes and ordering are static, allowing a module's initialization routine to pre-compute any data structures that will later be used during collective routines. This design emphasizes the potential run-time optimizations that can be obtained by shifting as much overhead calculations and coordination to the one-time initialization function as possible. This can reduce the amount of computational overhead in the run-time of collective routines.

The module is associated with the target communicator by caching its local state (such as the pre-computation results) on the communicator itself. All subsequent phases in the module's life cycle are invoked relative to a communicator for which it was selected; the communicator is passed as an argument to all invocation functions. This allows the module to retrieve its communicator-specific pre-computation data when a collective function is invoked.

Once a component has been initialized, it returns the module – including a list of function pointers for its algorithms – which is then assigned to the communicator. These functions are later invoked by the coll framework during the "normal usage" phase in the module's life cycle whenever a top-level MPI collective function is invoked. The module is then ready to be checkpointed or used for collective operations.

Checkpoint / Restart. Open MPI includes the capability for parallel MPI applications to be transparently checkpointed and restarted. In order for a parallel MPI application to be checkpointed, all its modules must include checkpoint / restart functionality. Much of this work is usually the responsibility of the point-to-point modules: they must ensure that "in flight" messages will not be lost upon restart. This is typically effected either by draining the network or utilizing acknowledgment / retransmission schemes.

coll modules that are layered on top of MPI point-to-point functionality therefore require no additional work to support checkpoint / restart; all the necessary

work is already performed by the point-to-point modules. coll modules that use their own communication channels, however, will typically need to include additional code to support checkpoint / restart functionality. Such modules can provide hook functions that the Open MPI framework will invoke during checkpoints and restarts to perform any required cleanup and re-initialization, respectively.

It is not an error if a module does not include the functionality required for checkpointing and restarting itself; support for checkpoint/restart in a coll module is optional. Currently, the determiniation of whether a process can checkpoint occurs during MPI_INIT: a process is checkpointable only if all the components that may be used in the process support checkpointing (regardless of whether they are selected).

Normal Usage.   After a coll module has been initialized with a communicator, that module's collective routines will be invoked whenever an MPI collective function is invoked on the communicator. Note that since the type of communicator is known at selection and initialization time (i.e., intra- or intercommunicator), it is the module's responsibility to set itself up so that intra- or intercommunicator algorithms are invoked as appropriate.

For example, when the `MPI_Bcast` C function is invoked on MPI_COMM_WORLD, it checks all of the parameters that are passed into it. It then invokes the the module's broadcast function pointer. The module's broadcast function pointer can either be specifically for intracommunicators or dispatch to an intracommunicator algorithm when it detects the type of MPI_COMM_WORLD. This model allows for a natural separation of algorithms and code since the algorithms used for intracommunicators are, by definition, different than the algorithms used for intercommunicators.

Finalization.   The final phase in a coll module's life cycle on a communicator occurs when the communicator is destroyed. The module's finalization method is responsible for cleaning up all resources associated with the communicator that is being destroyed.

3.4   Component and Module Interfaces

The coll component interface is relatively small; it contains data required for all Open MPI MCA modules such as references to the framework that the component belongs to, the name and version number of the component, and once-per-process initialization ("open") and finalization ("close") actions. Finally, two actions are defined specifically for coll components:

- One-time initialization. This method is invoked during MPI_INIT to ask certain threading characteristics about the component, and is mainly used

12

```
coll_component_interface {
 // Metadata identifying what version of the MCA this component
 // adheres to, what framework and version this component belongs to,
 // and this component's name and version.

 version mca_version_number;
 string mca_framework_name;
 version mca_framework_version_number;
 string component_name;
 version component_version_number;

 // Actions defined for all MCA components

 int component_open_function(void);
 int component_close_function(void);

 // Actions defined on coll components.

 int component_init_query(bool &allow_user_threads,
                          bool &have_hidden_threads);
 coll_module component_comm_query(MPI_Comm comm, int &priority);
}
```

Figure 3.   Pseudocode for the coll component interface.

to determine the final threading level that will be used during the process
(MPI_THREAD_SINGLE through MPI_THREAD_MULTIPLE).

- Per-communicator query. The coll framework invokes this method on
  each component, effectively asking the component if it wants to be con-
  sidered for selection. If it does, the component will return a module.

Pseudocode for the component interface is shown in Figure 3.

The module interface is divided into several categories of actions (shown in
Figure 4):

- Initialization and finalization. If a module is selected, its initialization
  method is invoked, allowing the module to complete any setup or pre-
  compute results that are utilized during the module's "normal usage"
  life cycle phase. All modules have their finalize method invoked when
  they are no longer used (which may be immediately if a module is not
  selected).

```
coll_module_interface {
  // Initialization and finalization of a module

  int init(MPI_Comm comm)
  int finalize(MPI_Comm comm)

  // Checkpoint/restart functionality

  int cr_interrupt(void)
  int cr_checkpoint(MPI_Comm comm)
  int cr_continue(MPI_Comm comm)
  int cr_restart(MPI_Comm comm)

  // Collective algorithm methods

  int allgather(buffer sbuf, int scount, MPI_Datatype sdtype,
                buffer rbuf, int rcount, MPI_Datatype rdtype,
                MPI_Comm comm)
  int allgatherv(buffer sbuf, int scount, MPI_Datatype sdtype,
                 buffer rbuf, int rcounts[], int disps[], MPI_Datatype rdtype,
                 MPI_Comm comm);
  // ...and the rest of the MPI collective operations
}
```

Figure 4. Pseudocode for the coll module interface. Module-specific state is cached on the communicator and is therefore passed in to every module method.

- Checkpoint / restart functionality. As described in [18], the checkpoint/restart functionality in LAM/MPI (and carried forward to Open MPI) consists of three distinct phases: checkpoint, continue, and restart. Methods are included to support each of these actions; their functionality is described further in [18].

- MPI collective functions. Modules contain a method for each MPI collective function (e.g., MPI_BCAST, MPI_BARRIER, etc.). Their function signatures are quite similar to their MPI counterparts, but some of the functions and arguments have been streamlined by the coll framework. For example, some components can treat a zero-byte broadcast as a no-op, and the coll framework will not invoke the module in such situations.

4.   Example Components

basic and smp are two of the coll components included in Open MPI. These components serve both as reference algorithms as well as examples of two different implementation models.

4.1   The basic Component

The basic component contains a full set of intra- and intercommunicator collectives. The intracommunicator algorithms are quite mature; they have been in LAM/MPI production code for years. The intercommunicator algorithms are new, but are essentially variations of their intracommunicator counterparts.

Prior generations of LAM/MPI—including the collective algorithms that the basic component is founded on—were based on a monolithic architecture. This made it a natural choice for not only influencing the design of the coll component interface, but also as a first coll component implementation. The successful port of the legacy LAM/MPI collective algorithms to the new framework (originally in LAM/MPI 7.x, and later to Open MPI) served as a validation of the overall coll design.

Although relatively naive, the basic routines can be used on any communicator (regardless of underlying topology), switching between $O(n)$ and $O(log(n))$ algorithms depending on the number of processes in the communicator. All of the basic algorithms essentially use MPI point-to-point functions for moving data between MPI processes. For example, in MPI_BCAST's logarithmic implementation, a traditional binomial tree is used: parent processes send data with MPI_SEND while child processes block in MPI_RECV.

4.2   The smp Component

The smp component was also instrumental in shaping the design of the coll framework. Based on the algorithms from the MagPIe project [13–14], the smp algorithms attempt to maximize bandwidth conservation across multiple levels of network latency. MagPIe focused on uniprocessors communicating across a WAN; the smp component is oriented to SMPs communicating on a LAN. The end effect is the same: two levels of network latency that can be exploited at run-time. Segmenting the communicator into groups of local process peers and electing representatives from each group to communicate with other groups provides a natural segregation of local and global communications.

Similar to the basic component, the smp component uses point-to-point communication to pass messages. Standard MPI functions are used to create sub-communicators and translate rank identifications between them. A direct implication of this model is that the coll framework must be able to handle recursive communicator creation and destruction. During the construction of
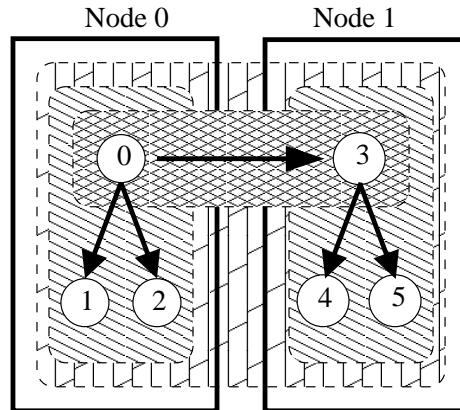
Figure 5. MagPIe algorithm for broadcast from process 0. Process 0 sends to its peer on the remote node (process 3). Each then do a local broadcast to the remaining processes on their nodes (processes 1 and 2, and processes 4 and 5, respectively).

```
int ssi_coll_smp_bcast(buffer, ..., MPI_Comm comm) {
  if (i_am_a_representative) {
    MPI_Bcast(buffer, ..., rep_root, rep_comm);
  }
  MPI_Bcast(buffer, ..., local_root, local_comm);
  return MPI_SUCCESS;
}
```

Figure 6. Pseudocode broadcast implementation using sub-communicators (error handling ignored for this example).

a communicator, the initialization of a coll module may cause the creation of another communicator. This may, in turn, trigger the creation of yet another communicator (and so on).

For example, in the MagPIe broadcast algorithm, the root broadcasts the data to the set of representatives from the other process groups. Each representative (including the root) then broadcasts to the members of its local group (see Figure 5). During the initialization phase of the smp module, the three sub-communicators shown in Figure 5 are created: two containing local-only processes, and one "bridge" communicator between processes 0 and 3. This allows the reducing the MagPIe broadcast algorithm implementation to the pseudocode shown in Figure 6.

Note that there are two calls to MPI_BCAST. These broadcasts use whichever module was selected when the sub-communicators were created. Depending

on the number of processes and topology involved, the broadcasts may be optimized according to however the selected coll component is implemented.

5. Performance

It is critical that the coll framework does not contribute additional overhead to collective algorithm performance. Measuring this is straightforward: compare the performance of Open MPI's collective functions against the prior generation of LAM/MPI (specifically, v6.5.9) that both provided the algorithms used in the basic component and was based on an integrated, monolithic model.

The collective algorithm implementations used in LAM/MPI 6.5, although somewhat naive, had well-understood behavior characteristics. Its main optimization technique is to switch between $O(n)$ and $O(log(n))$ when enough processes are involved in the collective. These collective algorithms were ported to the component architecture in Open MPI (the basic component, as described in §1.4.1). Measuring the performance of the same algorithms in two different architectures allows the comparison of overhead between the two.

A pair of dual-processor 2.0Ghz Intel Xeon nodes connected with Gigabit Ethernet and a dedicated switch was used for testing. Each node was running Red Hat 9 with Linux kernel 2.4.20 SMP and contained 2GB of RAM. The Pallas Benchmarks v2.2.1 were used to measure the wall-clock execution time of several MPI collectives in LAM/MPI and Open MPI.

The performance of MPI_BCAST, and MPI_ALLTOALL is shown in Figures 7 and 8, respectively. These graphs show that the performance of the collective algorithms in the Open MPI are on par with their peers in the LAM/MPI 6.5 series. Similarly, the performance of MPI_BARRIER is nearly identical between the two; wall-clock execution time for two processes was 73.5μs for LAM/MPI, and 80.2μs for Open MPI.

6. Conclusions

Effective, easy-to-use tools for enabling research in high performance computing are critical to meet the ever-growing demands of scientific applications. The component framework of Open MPI allows third-party researchers to develop and test new algorithms within an MPI implementation without the large time investment required to first become an MPI implementor. This allows quicker development of algorithms as well as a robust vehicle to allow users access to cutting-edge research.

Future work includes completing and releasing Open MPI (expected November 2004), writing coll components to exploit high performance in a new environments, tighter integration of MPI topology-based communicators with collective algorithms, and continued development and integration of other com-
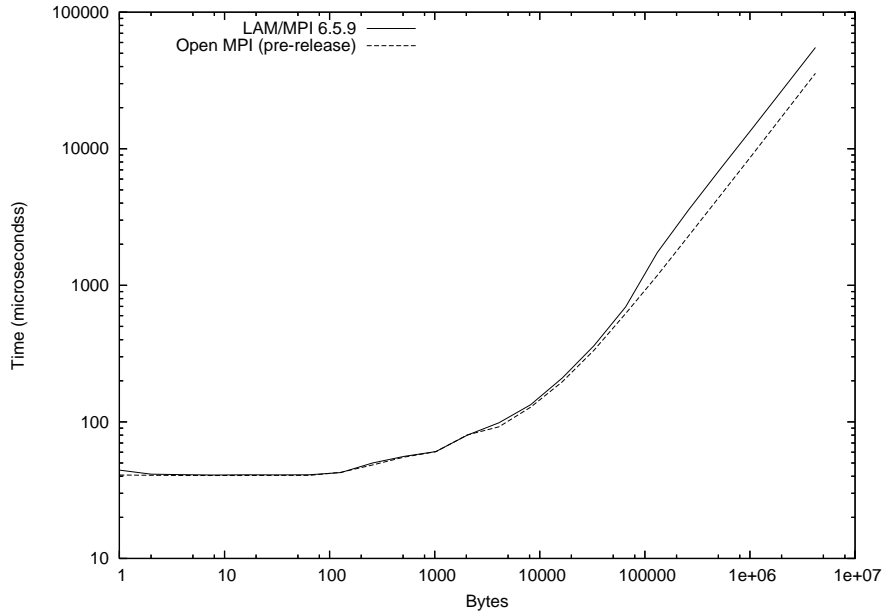
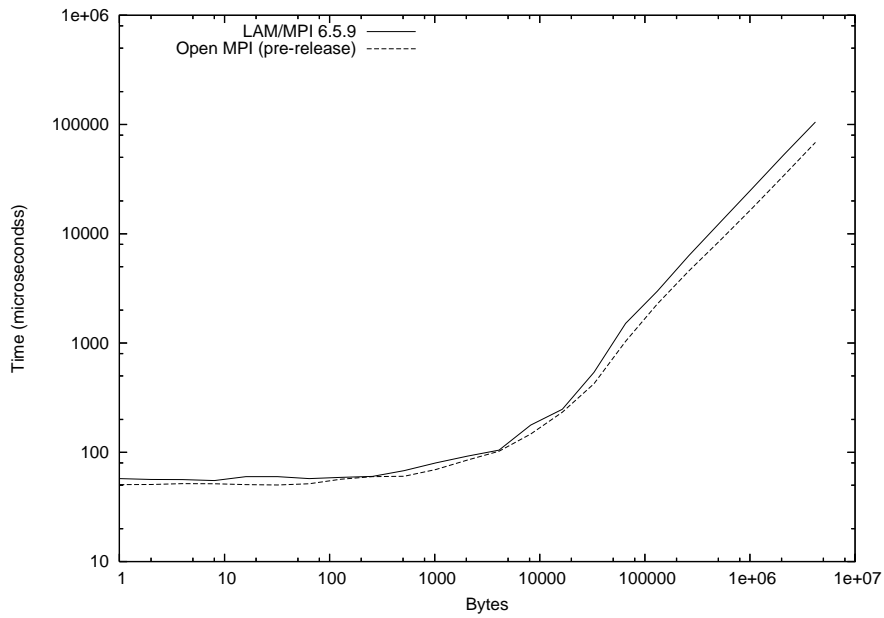*Figure 7.*   Wall-clock execution times for MPI_BCAST.

Figure 8.   Wall-clock execution times for MPI_ALLTOALL.

ponent types within the Open MPI implementation (particularly as they relate to collective algorithms).

References

[1] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mitchel W. Sukalski, Mark A. Taylor, and Timothy S. Woodall. Architecture of LA-MPI, a network-fault-tolerant mpi. In *Los Alamos report LA-UR-03-0939, Proceedings of IPDPS*, 2004.

[2] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[3] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27:1479–1496, 2001.

[4] Graham E. Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Antonin Bukovski, and Jack J. Dongarra. Fault tolerant communication library and applications for high perofrmance. In *Los Alamos Computer Science Institute Symposium*, Santa Fee, NM, October 27-29 2003.

[5] 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, Euro PVM/MPI*, Budapest, Hungary, September 2004. To appear.

[6] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message-Passing Interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing*, number 1123 in Lecture Notes in Computer Science, pages 128–135. Springer Verlag, 1996.

[7] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalksi. A network-failure-tolerant message-passing system for terascale clusters. *Intnl. Journal of Parallel Programming*, 31(4):285–303, August 2003.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[9] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[10] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *14th Intnl. Parallel Distributed Processing Symposium (IPDPS)*, pages 377–84, Cancun, MX, May 2000.

[11] Amit Karwande, Xin Yuan, and David Lowenthal. CCMPI: A compiled communication capable MPI prototype for ethernet switched clusters. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, USA, June 2003.

[12] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Mueller, and Michael M. Resch. Towards efficient execution of MPI applications on the grid: porting and optimization issues. *Journal of Grid Computing*, 1:133–149, 2003.

[13] Thilo Kielmann, Henri E. Bal, and Sergei Gorlatch. Bandwidth-efficient Collective Communication for Clustered Wide Area Systems. In *International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 492–499, Cancun, Mexico, May 2000. IEEE.

[14] 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. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 34(8):131–140, May 1999.

[15] S. P. Kini, J. Liu, J. Wu, P. Wyckoff, and D. K. Panda. Fast and Scalable Barrier Using RDMA and Multicast Mechanisms for InfiniBand-Based Cluster. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.

[16] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[17] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[18] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[19] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.

[20] Clemens Szyperski, Domink Druntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, second edition, 2002.

[21] Rajeev Thakur and William Gropp. Improving the Performance of MPI Collective Communication on Switched Networks. Technical report ANL/MCS-P1007-1102, Mathematics and Computer Science Division, Argonne National Laboratory, November 2002. `ftp://info.mcs.anl.gov/pub/tech_reports/reports/P1007.pdf`.

[22] Rajeev Thakur and William Gropp. Improving the Performance of Collective Operations in MPICH. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.

[23] T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *Proceedings, Euro PVM/MPI*, Budapest, Hungary, September 2004. To appear.

[24] Qianfeng Zhang. MPI collective operations over Myrinet. Master's thesis, The University of British Columbia, Department of Computer Science, June 2002.