CC–MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters*

Amit Karwande, Xin Yuan
Dept. of Computer Science
Florida State University
Tallahassee, FL 32306
{karwande,xyuan}@cs.fsu.edu

David K. Lowenthal
Dept. of Computer Science
The University of Georgia
Athens, GA 30602
dkl@cs.uga.edu

ABSTRACT

Compiled communication has recently been proposed to improve communication performance for clusters of workstations. The idea of compiled communication is to apply more aggressive optimizations to communications whose information is known at compile time. Existing MPI libraries do not support compiled communication. In this paper, we present an MPI prototype, *CC–MPI*, that supports compiled communication on Ethernet switched clusters. The unique feature of *CC–MPI* is that it allows the user to manage network resources such as multicast groups directly and to optimize communications based on the availability of the communication information. *CC–MPI* optimizes one–to–all, one–to–many, all–to–all, and many–to–many collective communication routines using the compiled communication technique. We describe the techniques used in *CC–MPI* and report its performance. The results show that communication performance of Ethernet switched clusters can be significantly improved through compiled communication.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

General Terms

Performance

Keywords

Compiled Communication, MPI, Message Passing Library, Clusters of Workstations

*This work was partially supported by NSF grants, CCR-9904943, CCR-0073482, CCR-0105422, and CCR-0208892

PPoPP'03, June 11–13, 2003, San Diego, California, USA.

1. INTRODUCTION

As microprocessors become more and more powerful, clusters of workstations have become one of the most common high performance computing environments. Many institutions have Ethernet–switched clusters of workstations that can be used to perform high performance computing. One of the key building blocks for such systems is a message passing library. Standard message passing libraries, including MPI [11] and PVM [24], have been implemented for such systems. Current implementations, such as MPICH [12] and LAM/MPI [28], focus on moving data across processors and addressing portability issues. Studies have shown that current implementations of message passing libraries are not tailored to achieve high communication performance over clusters of workstations [6].

Compiled communication has recently been proposed to improve communication performance for clusters of workstations [32, 33]. In compiled communication, the compiler determines the communication requirement of a program. The compiler then uses its knowledge of the application communications, together with its knowledge of the underlying network architecture, to directly manage network resources, schedule the communications, and exploit optimization opportunities. Compiled communication can perform optimizations across communication patterns at the software level, the protocol level, and the hardware level. It is more aggressive than traditional communication optimization techniques, which are either performed in the library [3, 8, 27, 31] or in the compiler [1, 2, 7, 13, 16, 29].

Compiled communication offers many advantages over the traditional communication method. First, by managing network resources at compile time, some runtime communication overheads such as group management can be eliminated. Second, compiled communication can use *long–lived* connections for communications and amortize the startup overhead over a number of messages. Third, compiled communication can improve network resource utilization by using off–line resource management algorithms. Last but not the least, compiled communication can optimize arbitrary communication patterns as long as the communication information can be determined at compile time.

The limitation of compiled communication is that it can only apply to *static communications*, that is, communications whose information can be determined at compile time. It has been found that a large percent of communications in scientific programs and in particular MPI programs are static [10, 19]. Thus, compiled communication can be ef-

fective in improving overall communication performance by optimizing common cases.

To facilitate compiled communication, mechanisms must be incorporated in the communication library to expose network resources to the users. Existing messaging libraries, including MPICH [12] and LAM/MPI [28], hide network details from the user and do not support compiled communication.

In this paper, we introduce an MPI prototype, *CC–MPI*, that serves as a run-time system for compiled communication on Ethernet switched clusters. Since the targeted users of *CC–MPI* are compilers and advanced programmers who know system details, we will use the terms user, compiler, and advanced programmer interchangeably throughout this paper.

CC–MPI optimizes one–to–all, one–to–many, all–to–all, and many–to–many collective communication routines. To support compiled communication, *CC–MPI* extends the MPI standard and uses separate routines for network control and data transmission. Network resources are exposed to the user through the network control routines, and various optimizations can be performed by manipulating (moving, merging, eliminating) the network control routines. Based on the availability of application communication information, *CC–MPI* allows the user to use different combinations of network control and data transmission routines for a communication to achieve high performance.

We describe the techniques used in *CC–MPI* and report our performance study of *CC–MPI*. The results of our study indicates that the communication performance of Ethernet switched clusters can be improved substantially through compiled communication. For example, on 16 nodes, *CC–MPI* speeds up the IS benchmark (class A), a program from the NAS suite [25], by 54% over LAM/MPI and 286% over MPICH.

The contribution of this paper is as follows.

- We develop a compiled-communication capable MPI library prototype that differentiates statically known and unknown communications, allowing optimization and significant performance gains for known communications.
- We develop three group management schemes for implementing one–to–all and one–to–many communications.
- We design a number of phase communication schemes for all–to–all and many–to–many communications.

The rest of the paper is organized as follows. The related work is presented in Section 2. In Section 3, we describe MPI collective communication routines and discuss features in switched Ethernet that affect the method for efficient communication. In Section 4, we present the techniques used in *CC–MPI*. In Section 5, we report the results of the performance study. Finally, Section 6 concludes the paper.

2. RELATED WORK

Extensive research has been conducted to improve communication performance in high performance computing systems. Many projects have focused on reducing the communication overheads in the software messaging layer [3, 8, 27, 31]. *CC–MPI* is different from these systems in that it allows users to select the most effective method based on message sizes and network conditions. Many parallel compiler projects also try to improve communication performance by generating efficient communication code [1, 2, 7, 13, 16, 29]. The communication optimizations performed by these compilers focus on reducing the number and the volume of communications and are architecture independent. While *CC–MPI* is not directly related to compiler optimization techniques, compiler techniques developed, however, can enable effective usage of *CC–MPI*.

The development of *CC–MPI* is motivated by compiled communication [4, 5, 18, 32, 33] and the need to support architecture-dependent communication optimization [14] at the library level. While it has been found that information about most communications in scientific programs and in particular MPI programs can be determined at compile time [10, 19], existing standard libraries, such as MPI [11] and PVM [24], do not support any mechanisms to exploit such information. *CC–MPI* is an attempt to extend the standard MPI library to support the compiled communication model and to allow the user to perform architecture-dependent communication optimization across communication patterns.

The success of the MPI standard can be attributed to the wide availability of two MPI implementations: MPICH[12] and LAM/MPI [28]. Many researchers have been trying to optimize the MPI library [17, 21, 26, 30]. In [17], optimizations are proposed for collective communications over Wide-Area Networks. In [26], a compiler based optimization approach is developed to reduce the software overheads in the library, which focuses on point–to–point communications. In [21], MPI point–to–point communication routines are optimized using a more efficient primitive (Fast Message). Optimizations for a thread-based MPI implementation are proposed in [30]. Our research is different from the existing work in that we develop a MPI library that allows static communication information to be exploited.

3. BACKGROUND

CC–MPI optimizes one–to–all, one–to–many, many–to–many, and all–to–all communication routines for Ethernet switched clusters by exploiting special features in switched Ethernet. In this section, we will briefly introduce MPI collective communication routines and discuss the features in switched Ethernet that affect communication efficiency.

3.1 MPI Collective Communication Routines

MPI is a library specification for message–passing, proposed as a standard by a broad-based committee of vendors, implementors and users. A set of standard collective communication routines are defined in MPI. Each collective communication routine has a parameter called a *communicator*, which identifies the group of participating processes. Next, we will briefly summarize the MPI collective communication routines. Details about these routines can be found in the MPI specification [11].

Function *MPI_Barrier* blocks the callers until all members in the communicator call the routine, and *MPI_Bcast* broadcasts a message from a *root* process to all processes in the communicator. The *MPI_Gather* and *MPI_Gatherv* routines allow each process in the communicator to send data to one process, while *MPI_Scatter* allows one process

to send a different message to each other process. Function *MPI_Scatterv* generalizes *MPI_Scatter* by allowing different-sized messages to each process. it can perform one–to–many communications within the communicator by carefully selecting the input parameters. The *MPI_Allgather* and *MPI_Allgatherv* routines gather fixed- or variable-sized information, respectively, from all processes and puts the results to all processes. Function *MPI_Alltoall* is a generalization of *MPI_Allgather* that allows different messages can be sent to different processes. The general form of all–to–all communication is *MPI_Alltoallv*, which allows many–to–many or one–to–many communications to be performed by carefully selecting the input arguments. Finally, *MPI_Reduce* performs global reduction operations, *MPI_Reduce_scatter* additionally puts the result to all processes, and *MPI_Scan* performs a prefix reduction on data distributed across the members of a communicator.

CC–MPI implements a subset of MPI routines, including all the routines required to execute the NAS parallel benchmarks [25]. *CC–MPI* optimizes the collective communication routines that contain one–to–all, one–to–many, all–to–all and many–to–many communication patterns. More specifically, *MPI_Bcast* and *MPI_Scatter* contain the one–to–all pattern, *MPI_Scatterv* contains the one–to–all or one–to–many pattern, *MPI_Allgather* and *MPI_Alltoall* contain the all–to–all pattern, and *MPI_Alltoallv* contains the all–to–all or many–to–many pattern. The one–to–all, one–to–many, all–to–all and many–to–many communication patterns are also part of the communications in *MPI_Barrier* and *MPI_Reduce_Scatter*.

3.2 Switched Ethernet

CC–MPI is designed for Ethernet switched homogeneous clusters. We assume that TCP/IP protocols are running on the end hosts and IP multicast can be used through the UDP interface. To achieve optimal performance, *CC–MPI* exploits the following features in switched Ethernet.

- Switched Ethernet supports broadcast at the hardware level. This indicates that using multicast primitives to realize broadcast types of routines, including *MPI_Bcast*, *MPI_Scatter*, and *MPI_Scatterv*, will likely result in good communication performance.
- Ethernet switches support unicast traffic effectively when there is no network contention in the system. However, when multiple large messages simultaneously target the same output port of an Ethernet switch, the communication performance may degrade significantly since Ethernet does not have a good mechanism for multiplexing to share the bandwidth. Hence, a communication library should avoid this scenario to achieve high performance.
- Multicast traffic in switched Ethernet negatively affects unicast traffic. Multicast should be used with great caution in Ethernet switched clusters.

4. CC-MPI

CC–MPI supports compiled communication by separating network control from data transmission. For each communication routine, zero, one or more network control routines and one or more data transmission routines are supported in the library. This way, different combinations of network control and data transmission can be used for a communication to achieve optimal performance. This allows the user to directly manage network resources, amortize the network control overhead over a number of communications, and use more efficient methods for static communications when more information about the communication is known. A number of factors allow *CC–MPI* to achieve high communication performance. First, *CC–MPI* uses different methods for each type of communication. Each method does not have to be effective for *all* situations. It only needs to be effective in some cases, because *CC–MPI* relies on its user to select the best method for a communication. This gives *CC–MPI* more flexibility in using customized communication methods. Second, some communication routines in *CC–MPI* make more assumptions about the communications to be performed than the general-case routines. With these assumptions, more effective communication routines are developed. Although such routines are not general, they provide high performance when applicable.

CC–MPI focuses on optimizing one–to–all, one–to–many, all–to–all, and many–to–many communications. To present the techniques used in *CC–MPI*, we will use one representative routine for each communication pattern. More specifically, we will use *MPI_Bcast* to illustrate how we implement one–to–all communication, *MPI_Scatter* for one–to–all personalized communication, *MPI_Scatterv* for one–to–many personalized communication, *MPI_Alltoall* for all–to–all communication, and *MPI_Alltoallv* for many–to–many communication. This section first describes techniques used in one–to–all and one–to–many communications, including issues related to multicast. Then, we discuss all–to–all and many–to–many communications, including our use of *phased* communication [15] to avoid network contention.

4.1 One–to–all and One–to–many Communications

MPI_Bcast, *MPI_Scatter*, and *MPI_Scatterv* realize one–to–all and one–to–many communications. These routines are traditionally implemented using unicast primitives with a logical tree structure [12, 28]. In addition to unicast based implementations, *CC–MPI* also provides implementations using multicast. Multicast based implementations can potentially achieve higher communication performance than a unicast based implementation because multicast reduces both the message traffic over the network and the CPU processing at the end hosts and because Ethernet supports broadcast at the hardware level. However, due to the complexity of reliable multicast protocols and other related issues, a multicast based implementation does not always perform better than a unicast based implementation.

There are two issues to be addressed when using multicast: reliability and group management. The current TCP/IP protocol suite only supports unreliable IP multicast through the UDP interface. MPI, however, requires 100% reliability. *CC–MPI* uses an ACK-based reliable multicast protocol [20] to reliably deliver multicast messages. We adopt this protocol for its simplicity. Group management is another issue to be addressed in a multicast-based implementation. Basically, a multicast group must be created before any multicast message can be sent to that group. A group management scheme determines when to create/destroy a multicast group. Given a set of N processes, the number of potential groups is 2^N. Thus, it is impractical to estab-

lish all potential groups for a program, and group management must be performed as the program executes. In fact, most network interface cards limit the number of multicast groups; as an example, Ethernet cards allow only 20 such groups simultaneously. Because the group management operations require the coordination of all members in the group and are expensive, the ability to manage multicast groups effectively is crucial for a multicast-based implementation. *CC–MPI* supports three group management schemes: the *static* group management scheme, the *dynamic* group management scheme, and the *compiler–assisted* group management scheme.

Static group management scheme In this scheme, a multicast group is associated with each communicator. The group is created/destroyed when the communicator is created/destroyed. Because a communicator is usually used by multiple communications in a program, the static group management scheme amortizes the group management overheads and makes the group management overhead negligible. This scheme is ideal for one–to–all communications, such as *MPI_Bcast*. Using the static group management scheme, *MPI_Bcast* can be implemented by having the root (sender) send a reliable broadcast message to the group.

A multicast based *MPI_Scatter* is a little more complicated. In the scatter operation, different messages are sent to different receivers. To utilize the multicast mechanism, the messages for different receivers must be aggregated to send to all receivers. For example, if messages $m1$, $m2$ and $m3$ are to be sent to processes $p1$, $p2$ and $p3$, the aggregate message containing $m1$, $m2$ and $m3$ will be sent to all three processes as one multicast message. Once a process receives the aggregated multicast message, it can identify its portion of the message (because the message sizes to all receivers are the same and are known at all nodes assuming a correct MPI program) and copy the portion to user space. In comparison to the unicast based *MPI_Scatter*, where the sender loops through the receivers sending a unicast message to each of the receivers, the multicast based implementation increases the CPU processing in each receiver because each receiver must now process a larger aggregated message, but decreases the CPU processing in the root (sender), as fewer system calls are needed. Because the bottleneck of the unicast implementation of *MPI_Scatter* is at the sender side, it is expected that the multicast based implementation offers better performance when the aggregated message size is not very large. When the size of the aggregated message is too large, the multicast based implementation may perform worse than the unicast based implementation because it slows down the receivers.

Realizing *MPI_Scatterv* is similar to realizing *MPI_Scatter*, with some complications. In *MPI_Scatterv*, different receivers can receive different sized messages and each receiver only knows its own message size. While the sender can still aggregate all unicast messages into one large multicast message, the receivers do not have enough information to determine the layout and the size of the aggregated message. *CC–MPI* resolves this problem by using two broadcasts in this function. The first broadcast tells all processes in the communicator the amount of data that each process will receive. Based on this information, each process can compute the memory layout and the size of the aggregated message. The second broadcast sends the aggregate message. Notice that it is difficult (although possible) to perform broadcast with an unknown message size. This is because *MPI_Bcast* requires the message size to be specified. As a result, *MPI_Scatterv* is implemented with two *MPI_Bcast* calls. *MPI_Scatterv* can realize one–to–many communication by having some receivers not receive any data. Using the static group management scheme, the one–to–many communication is converted into an one–to–all communication because all processes in the communicator must receive the aggregated message. This is undesirable because it keeps the processes that are not interested in the communication busy. In addition, this implementation sends a reliable multicast message to a group that is larger than needed, which can affect the performance of the reliable multicast communication. The dynamic group management scheme and the compiler–assisted group management scheme overcome this problem.

Dynamic group management scheme In this scheme, a multicast group is created when needed. This group management scheme is built on top of the static group management scheme in an attempt to improve the performance for one–to–many communications. To effectively realize one–to–many communication, the dynamic group management scheme dynamically creates a multicast group, performs the communication with only the intended participants, and destroys the group. In *MPI_Scatterv*, only the sender (root) has the information about the group of receivers (each receiver only knows whether it is in the group, but not whether other nodes are in the group). To dynamically create the group, a broadcast is performed using the static group associated with the communicator. This informs all members in the communicator of the nodes that should be in the new group. After this broadcast, a new group can be formed and the uninterested processes that are not in the new group can move on. After the communication is performed within the new group, the group is destroyed. With the dynamic group management scheme, *MPI_Scatterv* performs three tasks: new group creation (all nodes must be involved), data transmission (only members in the new group are involved), and group destruction (only members in the new group are involved). Dynamic group management introduces group management overheads for each communication and may not be efficient for sending small messages.

```
(1) DO i = 1, 1000
(2)     MPI_Scatterv(....)
```

(a) An example program

```
(1) MPI_Scatterv_open_group(...)
(2) DO i = 1, 1000
(3)     MPI_Scatterv_data_movement(....)
(4) MPI_Scatterv_close_group(...)
```

(b) The compiler–assisted group management scheme

Figure 1: An example of compiler–assisted group management.

Compiler–assisted group management scheme In this scheme, we extend the MPI interface to allow users to directly manage the multicast groups. For *MPI_Scatterv*, *CC–MPI* provides three functions: *MPI_Scatterv_open_group*, *MPI_Scatterv_data_movement*, and *MPI_Scatterv_close_group*.

MPI_Scatterv_open_group creates a new group for the participating processes in a one–to–many communication and initializes related data structures. *MPI_Scatterv_close_group* destroys the group created. *MPI_Scatterv_data_movement* performs the data movement assuming that the group has been created and that the related information about the communication is known to all participated parties. Notice that *MPI_Scatterv_data_movement* requires less work than *MPI_Scatterv* with the static group management scheme. This is because the message size for each process is known to all processes when *MPI_Scatterv_data_movement* is called, so only one broadcast (as opposed to two) is needed in *MPI_Scatterv_data_movement* for sending the aggregate message.

The *MPI_Bcast*, *MPI_Scatter*, and *MPI_Scatterv* with the static group management scheme are implemented as data transmission routines in *CC–MPI*. *MPI_Scatterv* with dynamic group management and *MPI_Scatterv_data_movement* are also data transmission routines. On the other hand, *MPI_Scatterv_open_group* and *MPI_Scatterv_close_group* are network control routines for *MPI_Scatterv*. Note that when compiled communication is applied, network control routines can sometimes be moved, merged, and eliminated to perform optimizations across communication patterns. The data transmission routines generally have to be invoked to carry out the actual communications. Consider the example in Figure 1, where *MPI_Scatterv* is performed 1000 times within a loop. Let us assume that the *MPI_Scatterv* sends to 5 nodes within a communicator that contains 30 nodes. When static group management is used, all 30 nodes must participate in the communication. When dynamic group management is used, only the 5 nodes will participate in the communication, which may improve reliable multicast performance. However, a multicast group that contains the 5 nodes in the communication must be created/destroyed 1000 times. With compiled communication, if the compiler can determine that the group used by the *MPI_Scatterv* is the same for all its invocations, it can perform group management as shown in Figure 1 (b). In this case, only 5 nodes are involved in the communication, and the multicast group is created/destroyed only once. This example demonstrates that by using separate routines for network control (group management) and data transmission, *CC–MPI* allows the user to directly manage the multicast groups and to amortize network control overheads over multiple communications. In addition, *CC–MPI* also allows more efficient data transmission routines to be used when more information about a communication is known.

4.2 All–to–all and Many–to–many Communications

MPI_Alltoall, *MPI_Alltoallv*, and *MPI_Allgather* realize all–to–all and many–to–many communications. There are many variations in the implementation of these routines. One scheme is to implement these complex all-to–all and many-to-many communication patterns over simpler one-to-all and one-to-many collective communication routines. For example, for N nodes, *MPI_Allgather* can be decomposed into N *MPI_Bcast* operations. While multicast can obviously improve communication performance for one–to–all and one–to–many communications, it may not improve the performance for the more complex many–to–many communications in Ethernet switched clusters. Consider realizing a many-to-many communication where s_1, s_2, and s_3 each sends a message of the same size to d_1, d_2, and d_3. This communication can be realized with three multicast phases:

$$\text{Phase 1: } \{s_1 \rightarrow d_1, d_2, d_3\}$$
$$\text{Phase 2: } \{s_2 \rightarrow d_1, d_2, d_3\}$$
$$\text{Phase 3: } \{s_3 \rightarrow d_1, d_2, d_3\}$$

This communication can also be realized with three unicast phases:

$$\text{Phase 1: } \{s_1 \rightarrow d_1, s_2 \rightarrow d_2, s_3 \rightarrow d_3\}$$
$$\text{Phase 2: } \{s_1 \rightarrow d_2, s_2 \rightarrow d_3, s_3 \rightarrow d_1\}$$
$$\text{Phase 3: } \{s_1 \rightarrow d_3, s_2 \rightarrow d_1, s_3 \rightarrow d_2\}$$

Using an Ethernet switch, the unicast phase and the multicast phase will take roughly the same amount of time and multicast-based implementations may not be more effective than unicast based implementations. Our performance study further confirms this. Thus, while *CC–MPI* provides multicast based implementations for some of the all–to–all and many–to–many communication routines, we will focus on the techniques we use to improve the unicast based implementation.

Traditionally, these complex communications are implemented based on point–to–point communications [12, 28] without any scheduling. Such implementations will yield acceptable performance when the message sizes are small. When the message sizes are large, there will be severe network contention in the Ethernet switch and the performance of these implementations will be poor. *CC–MPI* optimizes the cases when the message sizes are large using *phased* communication [15]. The idea of phased communication is to reduce network contention by decomposing a complex communication pattern into phases such that the contention within each phase is minimal. To prevent communications in different phases from interfering with each other, a barrier is placed between phases. Next, we will discuss how phased communication can be used to realize *MPI_Alltoall* (for all–to–all communications) and *MPI_Alltoallv* (for many–to–many communications).

Phase 0: $\{0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 4, 4 \rightarrow 5, 5 \rightarrow 0\}$
MPI_Barrier
Phase 1: $\{0 \rightarrow 2, 1 \rightarrow 3, 2 \rightarrow 4, 3 \rightarrow 5, 4 \rightarrow 0, 5 \rightarrow 1\}$
MPI_Barrier
Phase 2: $\{0 \rightarrow 3, 1 \rightarrow 4, 2 \rightarrow 5, 3 \rightarrow 0, 4 \rightarrow 1, 5 \rightarrow 2\}$
MPI_Barrier
Phase 3: $\{0 \rightarrow 4, 1 \rightarrow 5, 2 \rightarrow 0, 3 \rightarrow 1, 4 \rightarrow 2, 5 \rightarrow 3\}$
MPI_Barrier
Phase 4: $\{0 \rightarrow 5, 1 \rightarrow 0, 2 \rightarrow 1, 3 \rightarrow 2, 4 \rightarrow 3, 5 \rightarrow 4\}$

Figure 2: All–to–all phases for 6 nodes.

CC–MPI assumes that network contention only occurs in the link between an Ethernet switch and a machine. This assumption is true for a cluster connected with a single Ethernet switch. When multiple switches are involved, this assumption will hold when a higher link speed is supported for the links connecting switches. Under this assumption, the contention that needs to be resolved is in the links between a node and a switch. To avoid network contention within a phase, each node receives at most one message in a phase (receiving two messages potentially results in network contention). All–to–all communication for N nodes can be realized with $N-1$ phases and $N-2$ barriers. The ith phase contains communications

$$\{j \rightarrow (j+i) \ mod \ N \mid j = 0..N-1\}$$

Figure 2 shows the all–to–all phases for 6 nodes. The com-

munication is composed of 5 communication phases and 4 barriers. As can be seen from the figure, within each phase, each node only sends and receives one message, and there is no network contention within each phase. In the following discussion, we will call the phases that can form all–to–all communications *all–to–all phases*. Essentially, scheduling messages in an all–to–all communication according to the all–to–all phases results in no network contention within each phase. Notice that each source-destination pair happens exactly once in the all–to–all phases.

Using $N-2$ barriers potentially can cause a scalability problem. However, all–to–all communication itself is not scalable, and the extra barrier is swamped by data transmission as long as the message sizes are reasonably large. When the message size is large enough, phase communication reduces the network contention and achieves high communication performance. Note also that barriers can be very efficient with special hardware support, such as Purdue's PAPERS [9]. In our evaluation, we do not use any special hardware support, a barrier on 16 nodes takes about 1 millisecond.

Realizing many–to–many communication with phased communication is more difficult. Using *MPI_Alltoallv*, a node can send different sized messages to different nodes. This routine realizes many-to-many communication by specifying the size of some messages to be 0. The first difficulty to realize *MPI_Alltoallv* with phased communication is that the communication pattern information is not known to all nodes involved in the communication. In *MPI_Alltoallv*, each node only has the information about how much data it sends to and receives from other nodes, but not how other nodes communicate. To perform phased communication, however, all nodes involved in the communication must coordinate with each other and agree on what to send and receive within each phase. This requires that all nodes involved obtain the communication pattern information. *CC–MPI* provides two methods to resolve this problem. The first approach uses an *MPI_Allgather* to distribute the communication pattern information before the actual many–to–many communication takes place. The second approach, which can only be used when the user has additional information about the communication, assumes that the global communication pattern is determined statically for each node and stored in a local data structure. It is clearly more efficient than the first method.

Scheme 1: greedy scheduling
  Phase 1: $\{(0 \rightarrow 1, 1MB), (1 \rightarrow 3, 1MB)\}$
  MPI_Barrier
  Phase 2: $\{(0 \rightarrow 2, 10KB), (2 \rightarrow 3, 100B), (1 \rightarrow 5, 100B)\}$
  MPI_Barrier
  Phase 3: $\{(2 \rightarrow 1, 100B)\}$
Scheme 2: all–to–all based scheduling
  Phase 1: $\{(0 \rightarrow 1, 1MB), (2 \rightarrow 3, 100B), (1 \rightarrow 5, 100B)\}$
  MPI_Barrier
  Phase 2: $\{(1 \rightarrow 3, 1MB), (0 \rightarrow 2, 10KB), (2 \rightarrow 1, 100B)\}$

Figure 3: Scheduling messages: $(0 \rightarrow 1, 1MB)$, $(1 \rightarrow 3, 1MB)$, $(0 \rightarrow 2, 10KB)$, $(2 \rightarrow 3, 100B)$, $(1 \rightarrow 5, 100B)$, $(2 \rightarrow 1, 100B)$.

Once the global communication pattern information is known to all nodes, a message scheduling algorithm is used to minimize the total communication time for the many–to–many communication. Figure 3 shows an example that different message scheduling schemes can result in differences in the number of phases (and hence communication performance). In the example, we use notion $(src \rightarrow dst, size)$ to denote a message of *size* bytes from node *src* and node *dst*. Because there is no network contention within each phase, the time a phase takes depends only on the largest message sent in the phase; we will refer to a phase with the largest message of X bytes as an X bytes phase. Figure 3 shows two scheduling schemes. Scheme 1 contains one 1MB phase, one 10KB phase, one 100B phase, and two barriers, and Scheme 2 contains two 1MB phases and one barrier. As can be seen from this example, to minimize the total communication time for a phased communication, we must minimize both the number of phases needed to realize the communication and the amount of time spent within the phases. The latter can be achieved by having a balanced load within each phase.

CC–MPI supports two message scheduling schemes for many to many communications: *greedy* scheduling and *all–to–all* based scheduling. The greedy scheduling algorithm focuses on the load balancing issue. It works in two steps. In the first step, the algorithm sorts the messages in decreasing order in terms of the message size. In the second step, the algorithm creates a phase, considers each unscheduled message (from large size to small size) and puts the message in the phase if possible, that is, if adding the message into the phase does not create contention. Under our assumption, network contention is created when a node sends to two nodes and when a node receives from two nodes. If the sizes of the remaining messages are less than a threshold value, all messages are put in one phase. The greedy algorithm repeats the second step if there exists unscheduled messages. The operation to put all small messages in one phase is a minor optimization to reduce the number of barriers for realizing a communication pattern. This is useful because when the message sizes are small, the network contention is light and a barrier operation can be more costly than the contention. The load in the phases created by the greedy algorithm is likely to be balanced because messages of similar sizes are considered next to each other.

All-to-all based scheduling algorithm:

Input: Communication pattern
Output: Communication phases
(1)Sort messages based on their sizes
(2)**while** (there exist unscheduled messages) **do**
(3) **if** (the largest message size < the threshold) **then**
(4) Put all messages in one phase
(5) **endif**
(6) Let *all–to–all Phase i* (see Figure 2) be the phase that contains the largest unscheduled message
(7) Create a new empty phase P
(8) Schedule all unscheduled messages that appear in *all–to–all Phase i* in P
(9) For each unscheduled message in the sorted list **if** no conflict, put the message in P

Figure 4: All–to–all based scheduling.

The *all–to–all* based scheduling algorithm is shown in Figure 4. The main difference between this algorithm and the greedy algorithm is that messages are scheduled based on

all–to–all phases first before being considered based on their sizes. This algorithm attempts to minimize the number of phases while putting messages of similar sizes in the same phase. It can easily be shown that this algorithm guarantees that the number of phases is no more than $N-1$. The algorithm produces the most effective scheduling for all–to–all communication and will likely yield good results for communication patterns that are close to all–to–all communication.

Consider scheduling the following messages on 6 nodes: $(0 \rightarrow 1, 1MB)$, $(1 \rightarrow 3, 1MB)$, $(0 \rightarrow 2, 10KB)$, $(2 \rightarrow 3, 100B)$, $(1 \rightarrow 5, 100B)$, $(2 \rightarrow 1, 100B)$. To illustrate the idea, let us assume that the threshold value for the small message size is 0 and that the messages are sorted in the order as specified. The greedy scheduling works as follows: messages $(0 \rightarrow 1, 1MB)$, $(1 \rightarrow 3, 1MB)$ will be placed in phase 1 because they do not cause contention. After that, none of the remaining messages can be placed in this phase. For example, message $(2 \rightarrow 3, 100B)$ cannot be placed in this phase because node 3 receives a message from node 1 in message $(1 \rightarrow 3, 1MB)$. The greedy algorithm then creates phase 2 and places messages $(0 \rightarrow 2, 10KB)$, $(2 \rightarrow 3, 100B)$, and $(1 \rightarrow 5, 100B)$ in the phase. Message $(2 \rightarrow 1, 100B)$ cannot be placed in this phase because it conflicts with message $(2 \rightarrow 3, 100B)$, so a third phase is created for message $(2 \rightarrow 1, 100B)$. The all–to–all based scheduling scheme schedules the messages as follows. First, the algorithm searches for the all–to–all phase that contains message $(0 \rightarrow 1)$, which turns out to be *all–to–all phase 0* in Figure 2. The algorithm then creates a phase and puts messages $(0 \rightarrow 1, 1MB)$ and $(2 \rightarrow 3, 100B)$ in the phase because these two messages are in *all–to–all phase 0*. After that, each unscheduled message is considered. In this case, message $(1 \rightarrow 3, 1MB)$ cannot be placed in this phase because it conflicts with message $(2 \rightarrow 3, 100B)$. However, message $(1 \rightarrow 5, 100B)$ will be placed in this phase. After this, a second phase will be created for messages $(1 \rightarrow 3, 1MB)$, $(0 \rightarrow 2, 10KB)$, $(2 \rightarrow 1, 100B)$; none of these messages conflict. The results of greedy scheduling and all–to–all based scheduling are shown in Figure 3.

Depending on the availability of information about the communication, *CC–MPI* provides four different methods for many–to–many communications.

1. Simple communication that realizes many–to–many communication with point–to–point communication routines. This provides good performance when the message size is small and network contention is not severe.

2. Phased communication with the global communication pattern information calculated at runtime. In this case, the global communication information is distributed with an *MPI_Allgather* routine. After that, a message scheduling algorithm is executed at each node to determine how each communication is to be carried out. Finally, the message is transmitted according to the schedule. This routine is efficient when the user determines that large amounts of messages are exchanged with the communication; however, the details about the communication are unknown until runtime. We refer to this scheme as the *Level 1* compiled communication for *MPI_Alltoallv*.

3. Phased communication with the global communication pattern information stored in a data structure local to each node. The difference between this method and (2) above is that the *MPI_Allgather* is unnecessary. This scheme can be used by the compiler when the global communication information can be determined at runtime. We refer to this scheme as the *Level 2* compiled communication for *MPI_Alltoallv*.

4. Phased communication with the message scheduling information (phases) stored in a data structure local to each node. Phased communication is carried out directly using the phase information. This scheme can be used by the compiler when global communication information can be determined statically and scheduling is precomputed. This results in the most efficient phased communication for many–to–many patterns. We refer to this scheme as the *Level 3* compiled communication for *MPI_Alltoallv*.

These different schemes are supported in *CC–MPI* with two network control routines and two data transmission routines. The first data transmission routine supports point–to–point communication based implementation. The second data transmission routine, *MPI_Alltoallv_data_trans2*, performs the phased communication with the assumption that the phases have been computed and the related data structures are established. The first network control routine, *MPI_Alltoallv_control1*, performs the *MPI_Allgather* operation to obtain the communication pattern and invokes the message scheduling routine to compute the phases. The second network control routine, *MPI_Alltoallv_control2*, assumes that the communication pattern information is stored in local variables and only invokes the message scheduling routine to compute the phases. Depending on the availability of the information about the communication, different combinations of the network control and data transmission routines can be used to realize the function with different performance. For example, Level 1 compiled communication can be realized with a combination of *MPI_Alltoallv_control1* and *MPI_Alltoallv_data_trans2*, level 2 communication can be realized with a combination of *MPI_Alltoallv_control2* and *MPI_Alltoallv_data_trans2*, and level 3 communication can be realized with a single *MPI_Alltoallv_data_trans2*.

It must be noted that the system assumptions in *CC–MPI* simplifies the message scheduling algorithms. The method to deal with complex all–to–all and many–to–many communications can be extended for general network topologies, provided that more sophisticated message scheduling algorithms are developed for the general network topologies.

5. PERFORMANCE STUDY

We have implemented *CC–MPI* on the Linux operating system. In this section, we evaluate the routines implemented in *CC–MPI* and compare the performance of *CC–MPI* with that of two MPI implementations in the public domain, LAM/MPI and MPICH. The experimental environment is an Ethernet switched cluster with 29 Pentium III–650MHz based PCs. Each machine has 128MB memory and 100Mbps Ethernet connection via a 3Com 3C905 PCI EtherLink Card. All machines run RedHat Linux version 6.2, with 2.4.7 kernel. The machines are connected by two 3Com SuperStack II baseline 10/100 Ethernet switches as shown in Figure 5. We use LAM version 6.5.4 with direct

client to client mode communication and MPICH version 1.2.4 with device *ch_p4* for the comparison.

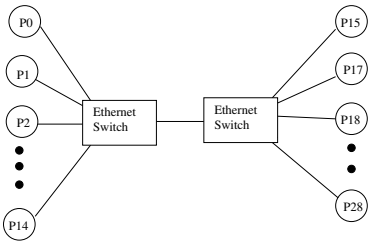

Figure 5: Performance evaluation environment.

This section first measures performance of MPI collective communication routines. Next, we present the results of two programs from the NAS benchmark suite. Finally, we present results from an initial prototype of a software distributed shared memory system that uses *CC-MPI* for efficient communication.

5.1 Individual MPI Routines

```
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
for (count = 0; count < ITER_NUM; count ++) {
  MPI_Bcast(buf, s, MPI_CHAR, 0, MPI_COMM_WORLD);
}
elapsed_time = MPI_Wtime() - start;
```

Figure 6: Code segment for measuring the performance of an individual MPI routine.

We use code similar to that shown in Figure 6 to measure the performance of individual MPI routines. For each experiment, we run the test three times and report the average of all results. For collective communication routines, we use the *average* time among all nodes as the performance metric. Because we are averaging the time over many invocations of an MPI routine, the average time is almost identical to the worst case time.

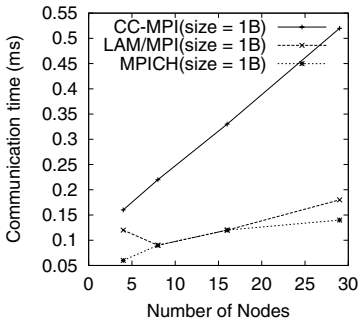

Figure 7: Performance of *MPI_Bcast* (size = 1B).

Figures 7 and 8 show the performance of *MPI_Bcast*. As can be seen from Figure 7, multicasting does not guarantee an improvement in communication performance, even for broadcast communication. The reason that the LAM/MPI and MPICH broadcast implementations are more efficient

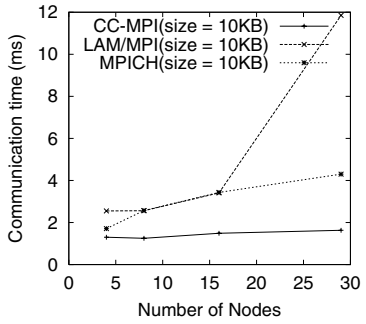

Figure 8: Performance of *MPI_Bcast* (size=10KB).

than our multicast-based implementation when the message size is 1 byte is that LAM/MPI and MPICH use an efficient logical tree-based implementation when the group is larger than 4 processes. This distributes the broadcast workload to multiple nodes in the system. In our implementation, the root sends only one multicast packet, but must process all acknowledgement packets from all receivers. As a result, for small sized messages, our multicast based implementation performs worse. However, when the message size is large (see Figure 8), the acknowledgement processing overhead is insignificant, and sending one multicast data packet instead of multiple unicast packets provides a significant improvement.

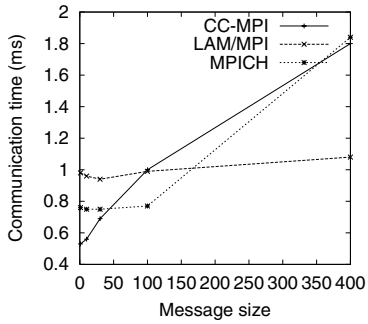

Figure 9: Performance of *MPI_Scatter* on 29 nodes.

Figures 9 shows the performance of *MPI_Scatter* on 29 nodes. In the scatter operation, the root sends different data to different receivers. This has two implications. For unicast implementation, the root must iterate to send a message to each of the receivers, and the sender is the bottleneck. The tree–based implementation used in broadcast cannot be utilized. For multicast implementation, the messages must be aggregated so that each receiver receives more than what it needs, which decreases performance. Thus, the multicast based implementation can offer better performance only when the message size is small ($50B$ in Figure 9).

MPI_Bcast and *MPI_Scatter* only use the static group management scheme. Next, we will evaluate the performance of one–to–many communication for which dynamic group management and compiler–assisted group management were designed. Figure 10 shows the performance of *MPI_Scatterv* with different implementations and group management schemes. In this experiment, the root scatters messages of a given size to 5 receivers among the 29 mem-

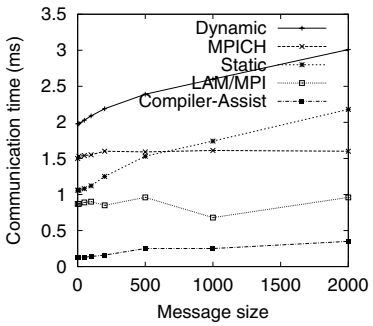

Figure 10: Performance of one–to–five communication using *MPI_Scatterv*.

bers in the communicator. As can be seen in the figure, the compiler–assisted scheme performs the best among all the schemes. The dynamic group management scheme incurs very large overheads and offers the worst performance among all the schemes. The static group management is in between the two. In this case, LAM/MPI outperforms both the dynamic and static schemes. We have conducted several experiments, and all results demonstrate a similar trend.

Data size	CC-MPI			LAM/ MPI	MPICH
	multicast	unicast	phased		
1B	1.8	1.1	17.7	0.7	1.9
1KB	8.7	9.4	18.1	7.6	5.0
8KB	61.4	82.5	28.7	108.4	15.3
64KB	494.2	705.2	112.2	937.7	335.0
256KB	1987.0	2739.6	378.8	3706.8	905.1

Table 1: Communication time of *MPI_Allgather* on 16 nodes (times in milliseconds).

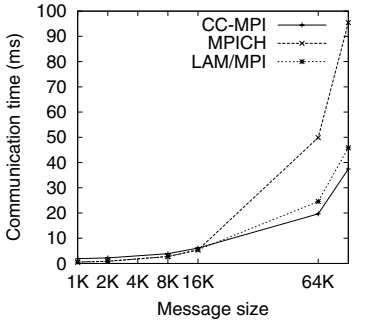

Figure 11: Performance of *MPI_Alltoall* on 4 nodes.

Table 1 shows the performance of *MPI_Allgather* on 16 nodes. *CC–MPI* contains three different implementations for *MPI_Allgather*: a multicast based scheme that reduces *MPI_Allgather* to a series of calls to *MPI_Bcast*, a unicast based scheme that realizes the communication with point-to-point primitives, and phased communication that is the unicast based scheme with message scheduling and synchronization. Columns 2 to 4 shows the performance of the three implementations in *CC–MPI*. As shown in the table, when the message size is very small (1B), the simple unicast based scheme performs best. When the message size is medium (1KB), the multicast based scheme is the best. For large message sizes ($\geq$ 8KB), phased communication is the best. Phased communication performs poorly when the message size is small due to synchronization overheads. However, it improves performance significantly (by up to a factor of 9) for large messages.

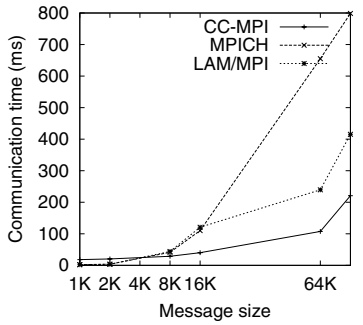

Figure 12: Performance of *MPI_Alltoall* on 16 nodes.

Figures 11 and 12 show the performance of *MPI_Alltoall*. *CC–MPI* uses phased communication to optimize the case when the message size is large. As can be seen from the table, even with 4 nodes, the network contention can still degrade communication performance, and our phased communication outperforms LAM/MPI and MPICH when the message size is larger than 16KB. For all–to–all communication over a larger number of nodes, the network contention problem is more severe, and the advantage of phased communication is more significant. With 16 nodes and a message size of 64KB, our phased communication completes an all–to–all communication about 1.5 times faster than LAM/MPI and 5 times faster than MPICH.

message size	*MPI_Alltoallv* implementation			
	Level 1	Level 2	Level 3	point-to-point
2KB	23.6ms	21.2ms	20.0ms	3.5ms
4KB	26.4ms	24.1ms	22.8ms	8.5ms
8KB	32.5ms	30.1ms	28.8ms	30.2ms
16KB	44.5ms	42.2ms	40.9ms	88.0ms
32KB	67.4ms	65.1ms	63.7ms	142.4ms
64KB	112.0ms	109.7ms	108.4ms	214.1ms

Table 2: Performance of different implementation of *MPI_Alltoallv*.

As described in Section 4, *CC–MPI* provides a variety of schemes for *MPI_Alltoallv*. Two scheduling algorithms, greedy and all–to–all, are supported. For each scheduling scheme, three levels of compiled communication schemes are implemented; Table 2 shows their performance. In this experiment, we use *MPI_Alltoallv* to perform all–to–all communication on 16 nodes. For this particular communication pattern, greedy and all–to–all based scheduling yield similar results, so only the results for all–to–all based scheduling are presented. As can be seen from the table, with more static information about the communication, more efficient communication can be achieved. The Level 3 implementation is about 15.7% more efficient than the Level 1 scheme when the message size is 4KB. As the message size becomes larger, the nearly constant cost of the *MPI_Allgather* and the scheduling operations become less significant. The Level 3

implementation is about 3% more efficient when the message size is 64KB. Notice that the message scheduling does not take a significant amount of time on 16 nodes. For a larger system, the scheduling overhead can be significant.

Libraries		Time
LAM/MPI		165.5ms
MPICH		358.3ms
CC-MPI all-to-all based scheduling	Level 1	108.3ms
	Level 2	106.0ms
	Level 3	105.9ms
CC-MPI greedy scheduling	Level 1	125.5ms
	Level 2	123.3ms
	Level 3	123.2ms

Table 3: Performance of *MPI_Alltoallv* for a random pattern on 16 nodes.

Table 3 shows the performance of different *MPI_Alltoallv* implementations for a randomly generated communication pattern on 16 nodes, which consists of 160 64KB messages, 20 16KB messages and 60 messages of size 100B. For this pattern, all-to-all based scheduling results in better performance than greedy scheduling. The reason is that the greedy algorithm realizes the pattern with 17 phases while the all–to–all based scheduling only needs 15 phases for the communication. It must be noted that both scheduling algorithms are heuristics; each performs better in some cases. When the scheduling can be done statically (Level 3 compiled communication), the better of the two scheduling schemes can be used to achieve the best results.

5.2 Benchmark Programs

In this subsection, we compare the performance of *CC-MPI* with that of LAM/MPI and MPICH using two benchmark programs, IS and FT from the NAS suite [25]. The Integer Sort (IS) benchmark sorts N keys in parallel and the Fast Fourier Transform (FT) benchmark solves a partial differential equation (PDE) using forward and inverse FFTs. These two benchmarks are presented in the NAS benchmarks to evaluate collective communication routines. Communications in other NAS benchmarks are either insignificant or dominated by point–to–point communications. Both IS and FT are communication intensive programs with most communications performed by *MPI_Alltoall* and *MPI_Alltoallv* routines.

Problem Size	MPI Library	Number of Nodes		
		4	8	16
CLASS=S	LAM/MPI	0.11s	0.09s	0.07s
	MPICH	0.10s	0.08s	0.07s
	CC–MPI	0.13s	0.16s	0.28s
CLASS=W	LAM/MPI	1.32s	1.02s	1.60s
	MPICH	2.69s	2.16s	1.57s
	CC–MPI	1.08s	0.69s	0.64s
CLASS=A	LAM/MPI	9.62s	5.88s	4.62s
	MPICH	21.92s	15.40s	11.60s
	CC–MPI	8.45s	4.90s	3.00s

Table 4: Execution time for IS with different MPI libraries, different numbers of nodes and different problem sizes.

Table 4 shows the results for IS, and Table 5 shows the results for FT. We run both benchmarks on 4, 8, and 16 nodes with the three problem sizes supplied with the benchmark—the small problem size ($CLASS = S$), the medium problem size ($CLASS = W$) and the large problem size ($CLASS = A$). LAM/MPI and $MPICH$ do not have any special optimizations for Ethernet switched clusters. As a result, when the communications in an application result in network contention, the performance degrades and is somewhat unpredictable. Different ways to carry out communications may result in (very) different performance under different network situations. As can be seen from the table, LAM/MPI performs much better than MPICH in some cases, e.g. the 'A' class IS on 16 nodes, while it performs worse in other cases (e.g., the 'A' class FT on 4 nodes). With *CC–MPI* we assume that the user determines that the communications will result in severe network contention with the traditional communication scheme and decides to use phased communication to perform *MPI_Alltoall* and *MPI_Alltoallv*. For *MPI_Alltoallv*, we assume that the Level 1 compiled communication scheme is used. As can be seen from the table, *CC–MPI* results in significant improvement (up to 300% speed up) in terms of execution time for all cases except for the small problem sizes. This further demonstrates that compiled communication can significantly improve the communication performance. For IS with a small problem size, *CC–MPI* performs much worse than *LAM/MPI* and *MPICH*. This is because we use phased communication for all cases. In practice, when compiled communication is applied, a communication model selection scheme should be incorporated in the compiler to determine the most effective method for the communications. In this case, for the small problem size, the compiler may decide that the message size is not large enough for the phased communication schemes to be beneficial and resort to point–to–point based dynamic communication scheme to carry out communications. Notice that for IS with a small problem size, the execution time with *CC–MPI* increases as the number of nodes increases. The reason is that both communication and computation take little time in this problem, so the execution time is dominated by the barriers in the phased communications. As the number of nodes increases, the number of barriers for each phased communication increases, and each barrier also takes more time.

Problem Size	MPI Library	Number of Nodes		
		4	8	16
CLASS=S	LAM/MPI	1.00s	0.70s	0.75s
	MPICH	2.04s	1.63s	1.01s
	CC–MPI	1.10s	0.63s	0.42s
CLASS=W	LAM/MPI	2.15s	1.55s	1.42s
	MPICH	4.17s	2.77s	1.46s
	CC–MPI	2.20s	1.28s	0.77s
CLASS=A	LAM/MPI	146.50s	27.37s	12.75s
	MPICH	111.91s	46.71s	28.01s
	CC–MPI	40.19s	21.34s	11.23s

Table 5: Execution time for FT with different MPI libraries, different numbers of nodes and different problem sizes.

5.3 Applying CC-MPI to Software DSMs

A software distributed shared memory (SDSM) provides the illusion of shared memory on a distributed-memory machine [22]. While using a shared-memory programming model simplifies programming in many cases compared to using message passing, one obstacle to widespread acceptance of SDSMs is performance. In particular, the communication traffic at synchronization points to maintain memory consistency is usually complex many–to–many communication with large message sizes. The techniques developed in *CC–MPI* perform such communications effectively. In this section, we report our early work aimed at using *CC–MPI* to improve SDSM performance. Our initial *CC–MPI*-enabled SDSM is built within the Filaments package [23] and uses an eager version of home-based release consistency [34].

We tested the potential of using Level 1 compiled communication for *MPI_Alltoallv* to implement exchange of page information through a synthetic application that first modifies a set number of pages on each node and then invokes a barrier. The barrier causes all pages to be made consistent through collective communication. This process is repeated for 100 iterations.

Table 6 presents the results of this benchmark. We observe that with a small number of nodes, the advantage of Level 1 compiled communication versus dynamic communication (the point–to–point based implementation without message scheduling) is relatively small. In fact, on 4 nodes, Level 1 compiled communication sometimes results in worse performance than dynamic communication. However, as the number of nodes increases, the advantage of Level 1 compiled communication is significant (almost a factor of two when four pages per node are modified). As the message size becomes much larger (starting at eight pages per node), the advantage decreases somewhat, but is still significant.

Pages Modified Per Node	Communication Method	Number of Nodes		
		4	8	16
1	Level 1	0.93s	2.02s	4.57s
	Dynamic	0.90s	4.29s	6.75s
4	Level 1	3.07s	6.76s	15.8s
	Dynamic	4.13s	12.1s	27.5s
8	Level 1	9.46s	15.7s	35.5s
	Dynamic	8.35s	18.6s	48.4s

Table 6: Experiments with our prototype SDSM that uses *CC–MPI* for communication.

6. CONCLUSION

In this paper, we present *CC–MPI*, an experimental MPI prototype that supports compiled communication. *CC–MPI* employs a number of techniques to achieve efficient communication over Ethernet switched clusters, including using multicast for broadcast type communications, supporting the compiler–assisted group management scheme that allows reliable multicast to be performed effectively, separating network control from data transmission, and using phased communication for complex many–to–many and all–to–all communications. We demonstrate that using compiled communication, the communication performance of Ethernet switched clusters can be significantly improved.

Compiled communication will likely be more beneficial for large systems, especially for massively parallel systems. In such systems, traditional dynamic communication is likely to generate significant network contention, which will result in poor communication performance. Compiled communication performs communications in a managed fashion and reduces the burden in the network subsystem. One difficulty with compiled communication is that it requires the user to consider network details, which is beyond of capability of a typical programmer. For the compiled communication model to be successful, it is crucial to develop an automatic restructuring compiler that can perform optimizations with compiled communication automatically. This way, programmers can write typical MPI programs and obtain high communication performance achieved through compiled communication.

7. REFERENCES

[1] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machine. In *the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, June 1993.

[2] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[3] M. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Virtual Memory–mapped Network Interfaces. *IEEE Micro*, pages 21–28, Feburary 1995.

[4] M. Bromley, S. Heller, T. McNerney, and G.L. Steele Jr. Fortran at Ten Gigaflops: the Connection Machine Convolution Compiler. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.

[5] F. Cappello and G. Germain. Toward High Communication Performance Through Compiled Communications on a Circuit Switched Interconnection Network. In *Proceedings of the First Int. Symp. on High–Performance Computer Architecture*, pages 44–53, 1995.

[6] P.H. Carns, W.B. Ligon III, S.P. McMillan, and R.B. Ross. An Evaluation of Message Passing Implementations on Beowulf Workstations. In *Proceedings of the 1999 IEEE Aerospace Conference*, March 1999.

[7] S. Chakrabarti, M. Gupta, and J. Choi. Global Communication Analysis and Optimization. In *ACM SIGPLAN Programming Language Design and Implementation(PLDI)*, pages 68–78, 1996.

[8] David Culler and et. al. *The Generic Active Message Interface Specification*. Available at http://now.cs.berkeley.edu/Papers/Papers/gam_spec.ps.

[9] H. G. Dietz, T. M. Chung, T. I. Mattox, and T. Muhammad, "Purdue's Adapter for Parallel Execution and Rapid Synchronization: The TTL_PAPERS Design", *Technical Report*, Purdue University School of Electrical Engineering, January 1995.

[10] Ahmad Faraj and Xin Yuan. Communication Characteristics in the NAS Parallel Benchmarks. In *Fourteenth IASTED International Conference on*

Parallel and Distributed Computing and Systems (PDCS 2002), November 4-6 2002.

[11] The MPI Forum. *The MPI-2: Extensions to the Message Passing Interface*, July 1997. Available at http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[12] William Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. In *MPI Developers Conference*, 1995.

[13] M. Gupta, E. Schonberg, and H. Srinivasan. A Unified Framework for Optimizing Communication in Data-Parallel Programs. *IEEE trans. on Parallel and Distributed Systems*, 7(7):689–704, July 1996.

[14] S. Hinrichs. *Compiler Directed Architecture–Dependent Communication Optimizations*, *Ph.D. Thesis*, Computer Science Department, Carnegie Mellon University, 1995.

[15] S. Hinrichs, C. Kosak, D.R. O'Hallaron, T. Stricker, and R. Take. An Architecture for Optimal All–to–All Personalized Communication. In *6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 310–319, June 1994.

[16] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed–Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[17] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R.A. F. Bhoedjang. Magpie: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *1999 SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 131–140, May 1999.

[18] M. Kumar. Unique Design Concepts in GF11 and Their Impact on Performance. *IBM Journal of Research and Development*, 36(6), November 1992.

[19] D. Lahaut and C. Germain. Static Communications in Parallel Scientific Programs. In *PARLE'94, Parallel Architecture & Languages*, LNCS 817, pages 262-276, Athen, Greece, July, 1994.

[20] Ryan G. Lane, Daniels Scott, and Xin Yuan. An Empirical Study of Reliable Multicast Protocols Over Ethernet-Connected Networks. In *International Conference on Parallel Processing (ICPP'01)*, pages 553–560, September 3-7 2001.

[21] M. Lauria and A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1), January 1997.

[22] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[23] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using Fine-Grain Threads and Run-time Decision Making in Parallel Computing. *Journal of Parallel and Distributed Computing*, 37:41–54, November 1996.

[24] R. Manchek. Design and Implementation of PVM Version 3.0. Technical report, University of Tennessee, 1994.

[25] NASA. *NAS Parallel Benchmarks.* available at http://www.nas.nasa.gov/NAS/NPB.

[26] H. Ogawa and S. Matsuoka. OMPI: Optimizing MPI Programs Using Partial Evaluation. In *Supercomputing'96*, November 1996.

[27] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2):60–73, April-June 1997.

[28] J.M. Squyres, A. Lumsdaine, W.L. George, J.G. Hagedorn, and J.E. Devaney. The Interoperable Message Passing Interface (impi) Extensions to LAM/MPI. In *MPI Developer's Conference*, 2000.

[29] J.M Stichnoth and T. Gross. A Communication Backend for Parallel Language Compilers. In *8th International Workshop on Languages and Compilers for Parallel Computing*, pages 15–1–15–13, August 1995.

[30] H. Tang, K. Shen, and T. Yang. Program Transformation and Runtime Support for Threaded MPI Execution on Shared-Memory Machines. *ACM Transactions on Programming Languages and Systems*, 22(4):673–700, July 2000.

[31] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U–net: A User–Level Network Interface for Parallel and Distributed Computing. In *the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[32] X. Yuan, R. Melhem, and R. Gupta. Compiled Communication for All–Optical TDM Networks. In *Supercomputing'96*, November 17-22 1996.

[33] X. Yuan, R. Melhem, and R. Gupta. Algorithms for Supporting Compiled Communication. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 2, pages 107-118, Feb. 2003.

[34] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, 1996.

Appendix

The *CC-MPI* package is available at http://www.cs.fsu.edu/∼xyuan/CCMPI. The MPI functions supported by *CC-MPI* include:

`MPI_Abort`	`MPI_Allgather`	`MPI_Allreduce`
`MPI_Alltoall`	`MPI_Alltoallv`	`MPI_Barrier`
`MPI_Bcast`	`MPI_Comm_dup`	`MPI_Comm_rank`
`MPI_Comm_size`	`MPI_Comm_split`	`MPI_Datatype`
`MPI_Finalize`	`MPI_Gather`	`MPI_Init`
`MPI_Irecv`	`MPI_Isend`	`MPI_Request`
`MPI_Recv`	`MPI_Reduce`	`MPI_Scatter`
`MPI_Scatterv`	`MPI_Send`	`MPI_Status`
`MPI_Wait`	`MPI_Waitall`	`MPI_Wtime`