COMMUNICATION LIBRARY FOR PARALLEL ARCHITECTURES

BY

KRISHNAN VARADARAJAN

B. Tech., University of Kerala, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

Abstract

Communication optimizations tend to be specific to a particular architecture or an application. To improve portability and to reduce the cost of developing parallel applications a mechanism to integrate these different optimizations should exist. Moreover, it should be possible to automatically adapt the strategy to the situation at hand. In this thesis we present a communication library that integrates the different strategies to perform each-to-many multicast. The framework provided is flexible enough to absorb new strategies and communication patterns. It also provides the capability to do dynamic switching of strategies. This helps the library to adapt itself to the existing environment. The library is written on top of Converse, which supports interoperability among different parallel programming paradigms. Thus the library can be used by a wide range of applications. The thesis also discusses some of the issues related to the design of re-usable parallel libraries. A performance comparison of the different strategies for each-to-many multicast on various architectures is also presented.

Acknowledgements

I would like to thank my advisor, Professor L. V. Kale, for his guidance and support. He was a constant source of assistance and the discussions I had with him were always stimulating.

I would also like to thank the members of the Parallel Programming Laboratory and the Theoretical Biophysics Group for providing a wonderful working environment. In particular, I would like to thank Robert Brunner and Milind Bhandarkar for their valuable suggestions and help.

I wish to thank Robert Brunner for running my benchmark programs on the ASCI Red and the CRAY T3E. Milind Bhandarkar ran my benchmark programs on the Origin2000. I acknowledge the U.S. Department of Energy (DOE) for the computer time on the ASCI Red, the Pittsburgh Super-computing Center for the computer time on the CRAY T3E and National Center for Supercomputing Applications (NCSA) for the time on the SGI Origin2000.

Finally, I am grateful to my parents for their constant support and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

High performance computing emerged as a result of the insatiable need for more computing power. Parallel computation attempts to offset the speed limitation imposed by the hardware, by expanding the computational capability in another dimension. In the initial stages, the parallel architectures used to be one of a kind and required specialized software to develop parallel applications. This made the cost for parallel computation exhorbitantly high. This gave impetus to the research in high performance computing on clusters [4, 16, 18, 2]. The clusters can take advantage of many of the commodity hardware. Recent research has improved the communication latency and bandwidth in a network of workstations [19, 20]. The use of workstation clusters for high performance computing has been well established [15, 22, 26, 23]. The strategies for optimizing the communication operations are mostly specific to an architecture.

To take advantage of all the available architectures, the applications should be able to run on all of them in a nearly optimal fashion, without requiring any major modifications. The communication library presented in this thesis attempts to integrate the various communication optimizations in the form of re-usable libraries. The library runs on top of Converse [10], a run-time system that has been ported to most of the commercial parallel machines and network of workstations. The framework presented here can be used to plug-in new strategies, test them on various platforms and use them in different applications. The library provides run-time switch-ability which could be exploited to perform "adaptive communication load balancing".

The thesis also addresses the issue of developing parallel libraries that are flexible, easy to use and re-usable. The objective is to provide the user with all the functionalities that he needs, delivering the best possible performance, but at the same time keeping the user interface simple and uncluttered.

1.1 Motivation

The plethora of factors that contribute to the communication cost and the dynamic nature of many of them make it very difficult to optimize the communication operations. The background communication load, the communication pattern of the application etc. are examples of the parameters that are dynamic in nature. For a portable application, the underlying topology is another dynamic factor. What is needed is an algorithm that can adapt to the situation at hand. When the background communication load is high, it should be possible to switch from one communication strategy to another. Similarly when the application is run on a different architecture, the optimal scheme for the underlying topology should be used. The user should be oblivious to this adaptive switching of strategies. This is analogous to dynamic load balancing in the case of computation. However, the impact would not be as profound because the communication optimizations are at discrete levels. The granularity is much coarser than in the case of computation. These optimizations can be performed at various levels. For example, at the lowest level the traffic over a particular link can be used to trigger a different routing scheme to be used. At a higher level, the short messages from an application could be combined to reduce the overall per message overhead. An optimization at a higher level has more flexibility and hence probably, is more effective.

In this thesis we present a framework that would address these issues to a very large extent. The communication library presented here has the two essential features required for dynamic adaptablilty - flexibility and run-time switch-ability. These features are discussed in more detail in the next section.

1.2 Thesis Objectives

In this thesis we describe the design and implementation of the communication library. The main objectives of the communication library were :

- To capture widely used communication patterns and provide alternate implementations for them in the form of reusable libraries.

- To provide a flexible framework, where the various strategies can be plugged in easily and tested.
- Mechanism to switch between the different strategies at run-time.
- To attain all the above at no significant performance cost over the hardwired code optimized for individual situations.

The library provides the adequate mechanisms to develop adaptive communication strategies. This would eliminate most of the architecture specific optimizations and encourage the development of portable applications.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 describes the communication operations included in the library and the related work in the field of collective communication. An overview of the Converse run time system which was used to develop the communication library, is also presented towards the end of the chapter. The architecture of the communication library is described in chapter 3. It also includes a brief discussion of the major design issues. In addition to the overall architecture, the alternate designs considered and the rationale behind choosing one over the other along with the various optimizations used in implementing the communication library are described. A brief description of the various benchmarks used to evaluate the performance of the library, the performance results and its analysis is presented in Chapter 4. The applicability of the library is illustrated by incorporating it in a real application called namd2. A brief description of the application and the performance results are included. The conclusions drawn and the future work is discussed in chapter 5.

Chapter 2

Communication Operations

Parallel applications tend to spend a lot of time in performing communication between the different nodes in a system. When the communication operation span most or all the nodes in a parallel machine, it is termed collective communication. Broadcast, multicast, barrier synchronization, scatter, gather, total exchange etc. are examples of such operations. Parallel applications make extensive use of these operations. A lot of research has gone into optimizing collective communication operations. The next section describes the communication operations included in the library. This is followed by a discussion of the related work in this field.

Communication Operations

The communication library includes several variants of each-to-many multicast. Each-to-many multicasts involves every processor in the system sending a message to a subset of the processors. If the set of destination processors consist of all the processors in the system, then it is referred to as each-to-all multicast. In another variant, the processors are allowed to send individualized messages. This is termed as each-to-many personalized sends. The library is aimed at applications that generate a large number of short messages. One example of such an application is the molecular dynamics program described in section 4.4.1. The other applications include permutation, matrix transpose, fast Fourier transform etc.

For each of the variants mentioned above, the communication library implements five different strategies. The first strategy consists of sending the message to the destination processor in a loop. This is the naive method of performing multicast. However, this method benefits from the fact that the sends can be pipelined and part of the overhead is masked by the overlap of computation and communication. All the other strategies are compared against this scheme. The broadcast based multicast scheme takes advantage of the underlying implementation of the broadcast operation. The tree, grid [25] and hypercube [25] based implementations try to reduce the overall per message overhead, by sending fewer messages. The scheme essentially consists of combining the messages on each processor and propagating it along any one of the topologies mentioned above. A detailed discussion of these schemes are given in section 4.1.

2.1 Related Work

The network topology is one of the factors that contribute to the communication cost. Different topologies like ring, fat trees, hypercube,Omega network, Benes network etc. have been used to interconnect the processing elements (PE) in a system. Most of the communication optimizations are specific to a particular topology. So it is worth while to look at the various topologies and the parameters that are used to evaluate them. The interconnection networks can be broadly classified into regular and irregular networks. The regular networks can be further classified into static and dynamic networks [7]. The point-to-point connections in the former case do not change during program execution. Hypercube, mesh, torus, ring etc. are examples. The dynamic networks are characterized by switched channels, which can be dynamically configured to match the communication demands of the user program. A crossbar switched interconnection network is an example of dynamic networks. Most of the commercial parallel machines employ static networks. The important parameters that characterize the different topologies are - node degree, network diameter, bisection bandwidth and wire length. The number of channels incident on a node is termed the node degree. The network diameter is the maximum shortest distance between any two nodes. The minimum number of channels that need to be cut to separate the network into two equal halves is termed the bisection bandwidth. An analysis of k-ary n-cube interconnection networks with respect to these parameters is presented in [5].

Gossip in trees under a line-communication mode is described in [14]. Line communication mode refers to the model in which every node can communicate with every other node in the system. This relaxes the usual restriction where only the neighbors can communicate. The algorithm proceed in phases. In each phase the edge-disjoint paths are determined. The paper also presents the upper and lower bound for the gossip time. This method can be applied to any connected graph. A total exchange algorithm on Cayley networks using a single port assumption is described in [6]. It exploits the symmetries inherent in Cayley graphs. Gossiping on meshes of arbitrary dimensions is presented in [3]. It assumes a worm-hole routed network. The algorithms concentrate data on certain processing units in the initial phase. In the subsequent phases, increase the number of such concentration points or bridgeheads by a constant factor.

The 2D mesh and hypercube based strategies for each-to-many multicast implemented in the communication library were borrowed from [25]. The authors describe the methods to perform one-to-all and all-to-all broadcast using message combining along a ring, mesh and hypercube interconnection networks. The store and forward and cut-through routing based schemes are discussed separately. The case where the messages are individualized is also presented. Other interesting communication operations like permutation, prefix-sum and reduction operation are also discussed. An extension of the mesh algorithm to torus is presented in [8].

The next section provides a brief overview of the Converse run-time framework that was used to develop the communication library.

Converse

The communication library was developed on top of Converse [9], an inter-operable parallel run-time framework that abstracts the underlying architecture from the user and extends to him a uniform and consistent machine model. Converse supports interoperablilty among multiple languages and parallel programming paradigms. This allows the communication library to be used by several parallel languages. The library is portable because it will run on all the machines that Converse supports. Converse has been ported to most of the commercial parallel machines like the Cray T3E, the SGI Origin2000, the ASCI Red etc. and network of workstations. Converse provides a wide spectrum of communication primitives that can be tailored to suit the performance needs of the user's application.

Chapter 3

The Architecture

The communication library was designed to be flexible, re-usable and extensible. But performance was the most important criteria in the design of this library. Some of the design issues involved in developing the library are discussed below.

The library was designed to be flexible enough to cater to the varied needs of the users. An implicit restriction of the *each-to-many* multicast operation was that all the processors should participate in the multicast. One way to relax this restriction was to let the user establish multicast groups. Now, only the processors in the multicast group need to participate in the operation. To improve the flexibility further, the library allows the user to make dummy multicast calls. Thus a processor with no data to send in a particular iteration can participate in the operation by making a dummy multicast call. This feature was essential to incorporate the library into namd2.

One harmful side effect of improving flexibility was the increased number of function calls that the user needs to make to perform an operation. For example, in the most common case, the user would prefer to make just one function call- *each-to-all Multicast*, rather than 3 different calls to establish the group, to indicate the number of deposits and the actual multicast function. To address this problem the common case was separated from the special cases. All the complex interfaces were moved to the special cases and the interface for the common case was kept simple and clean. The common case was defined to be the default mode of operation. Thus to perform a simple *each-to-all Multicast* operation, the user need not worry about establishing groups and so on.

Re-using a well known pattern decreases the learning curve for the user. For example, most of the libraries in Converse use the same sequence of operations for initialization :-

1. *Call Init() on all the processors*

2. *Create an instance on one of the processors and obtain the id*

3. *Broadcast the id to all the processors*

This was adopted in our implementation too.

Another important feature of any library is extensibility. In addition to being easy to modify and maintain, the library should be able to absorb new strategies and algorithms,transparent to the user. The component based architecture allows new strategies to be plugged in relatively easily. An object oriented approach was used to enhance the extensibility of the library. The different types of objects used in the library are self-reliant with well defined interfaces. Any object that exports the same interface can replace them. The API to plug-in new strategies are described in appendix A.

Achieving all this at no extra performance cost was the most daunting task. The next few sections describe the architecture of the communication library and the various optimizations employed to improve its overall performance.

3.1 The Framework

In order to achieve the objectives mentioned above, the important functional units of the communication library were first identified. The router objects were designed to abstract the algorithms used to route the messages. Thus a *Tree-Router* routes the messages along a spanning tree, while a *Grid-Router* routes it along a 2D mesh. Since the user program could consist of multiple modules worked on by different groups, a mechanism to create multiple instances of the library was needed. These instances were identified by a communication id (called *comID*) and can be created on any processor. The user can also specify the routing scheme to be used for any particular instance. The *comID* encodes all this information in a struct as shown in Figure 3.1. The first two fields of comID were used to ensure the uniqueness of the id across all the processors. The next field determines the communication strategy associated with that instance. The other fields store the switching and the multicast group information.

```
struct {
int srcpe;
int tblindex;
int impltype;
int switchrefno;
int numParticipants;
CmiGroup grp;
} comID;
```

Figure 3.1: Structure of comID

A table of router objects (Figure 3.2) was maintained on each processor to keep track of the different instances of the communication library. The first two fields of the *comID* were used to index into this table. Every time a user request or a message was received, the *comID* was used to index into the table and retrieve the router object for that particular instance. If the router object is not yet created, the information in the *comID* was used to create and install the router object. The update is lazy as the objects are created only when the *comID* is used on that processor. It also takes care of the race condition where a processor receives a multicast message before the *comID* is broadcast by the user.

Dynamic switching required deleniation of boundaries between the different iterations of the multicast operation within a particular instance. This involved the additional overhead of generating iteration numbers, buffering messages across iterations etc. The overlap manager was introduced to take over these responsibilities.

3.2 The Overlap Manager

The overlap manager is the layer between the user and the router objects that hides the complexity of the underlying communication strategy and extends to the user a simple, consistent interface. This enables the router objects to be switched transparent to the user. The overlap manager generates an iteration number for each multicast and keeps track of the iteration boundaries. It also performs the necessary buffering of messages to ensure their in-order delivery. The overlap manager has the additional responsibility of switch management which is discussed in the subsection 3.6.

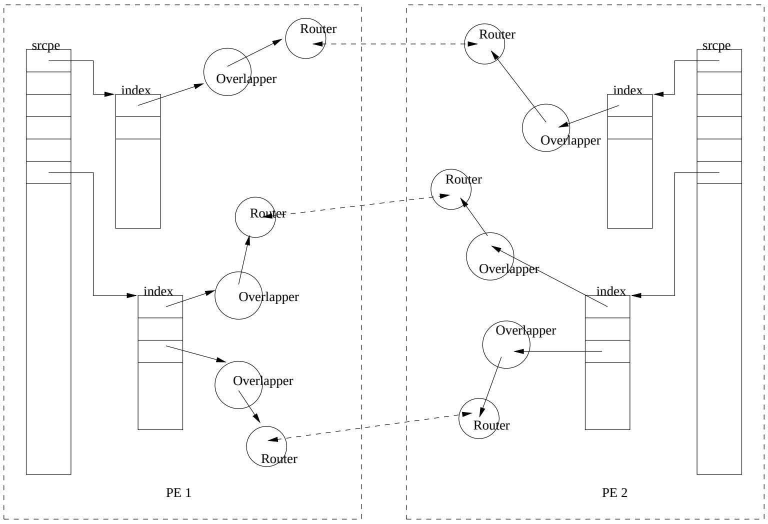

Figure 3.2: Overall Design of the Communication Library

All the entry methods (methods that take only a message as its parameter) in the router objects have a representative in the overlap manger. Moreover, the overlap manager exports the user interface of the communication library. Each of these methods in the overlap manager was associated with an overlap buffer as show in Figure 3.3. The messages transported by the communication library have the iteration number to which they belong to, encoded in their header. Any message or user request that arrive too early was buffered in the corresponding overlap buffer. These buffers are recycled to reduce the memory allocation overhead. When all the messages for a particular iteration of the multicast are received, the router object informs the overlap manager, which in turn prepares for the next iteration. It first drains all the buffers of all the messages for the next iteration number and delivers them to the corresponding entry methods. The user requests are retrieved in the first-in first-out order, while only the messages for that iteration number are processed. Thus the multicasts using the same id are performed in a strict sequential non-pipelined manner. To overlap multicasts, the user should use different instances. The advantage of non-overlapped multicasts are that it is easier to switch from one strategy to another at run-time. Again, it guarantees in order delivery of messages originating from the same processor, thus eliminating the need for reference counting by the user. The internal iteration number that is generated is used to specify the time to switch. Since the iteration number is just an integer, it is easy to publish the time to switch. (It is actually encoded in the *comID*).

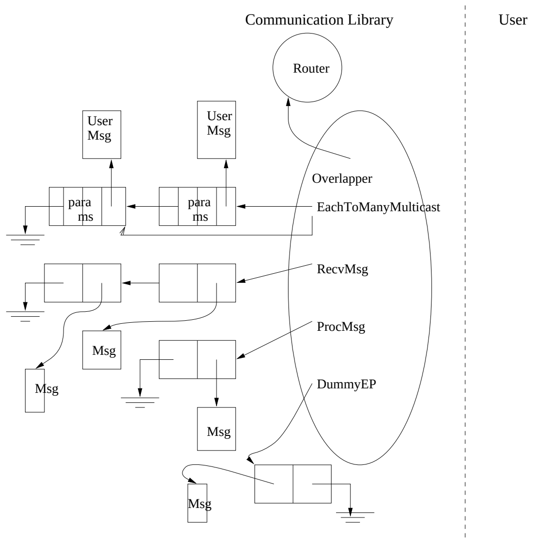

Figure 3.3: Overlap Manager

The overlap manager is also responsible for the mapping of processors in a multicast group. When a multicast group is established, the overlap manager generates the processor mapping and informs the router objects about it. The router objects consult the map before sending any messages. For performance benefits, the overlap manager provides two kinds of multicast groups. The first kind of group let the users specify an upper bound on the processor numbers. The domain then consists of all the processors from 0 to n-1. In this case, no processor mapping need to be generated. The specified upper bound is stored in the comID, thus eliminating the need for any further communication. The other kind of multicast group provides the user with the flexibility to choose any set of processors to belong to the group. This uses the underlying Converse multicast group establishment mechanism, which utilizes some network bandwidth and a small amount of memory on each processor. Once the group is established, no additional overhead is involved in sending messages.

3.3 Communication Strategies

The router objects abstract the underlying communication strategy. They are designed in such a way that no messages are buffered across different iterations. The communication library currently consists of three different communication strategies. Each of these strategies are described in detail in the next few subsections.

3.3.1 Tree

In the tree based scheme, the multicast is performed along a spanning tree of all the processors in the group. Experiments have shown that the optimal degree of the spanning tree for such an operation is 4. Each node in the spanning tree waits for a message from all its children. Once all the messages are received, they are combined together and sent up to its parent in the spanning tree. At the root, the individual messages are sorted out, recombined and sent down the tree.

One disadvantage with the above scheme is that the root node has more work to do than any other node in the spanning tree. To reduce this bottleneck, a slightly different scheme for combining messages was used in our implementation. At each node, the messages whose destination processors belong to the current subtree are retained. All the other messages are combined and sent up the spanning tree. Thus each node retains all the messages that need to be transported from one of its child subtree to another. Consider an each to all personalized send operation as shown in Figure 3.4. Each node has 15 messages to send of which one of the messages is destined for itself. The other 14 messages are combined and sent to its parent. Thus node 3 receives 2 such messages, one each from node 7 and 8. Out of the 14 constituent messages that it received from node 7, it retains the message for node 8 and consumes the message for node 3. Thus it combines the 36 messages - 12 from node 7, 12 from node 8 and 12 from node 3, and sends it to its parent node 1. The root node now processes 112 messages as compared to 196 messages in the original scheme.

To detect the end of an iteration, the local messages are delivered only at the final stage. In the case of a spanning tree, a node enters the final stage, when it receives a message from its parent. Another reason to postpone the consumption of local messages is that the user program in that node could exit once it receives all the local messages. However, the router object on that node might still be needed to route messages. By delivering the local messages only at the final stage, this problem is taken care of.

In certain occasions the nodes send dummy messages to signal the completion of a stage and to trigger the next stage in the multicast operations. The dummy messages are sent only if no user message is destined for that node.

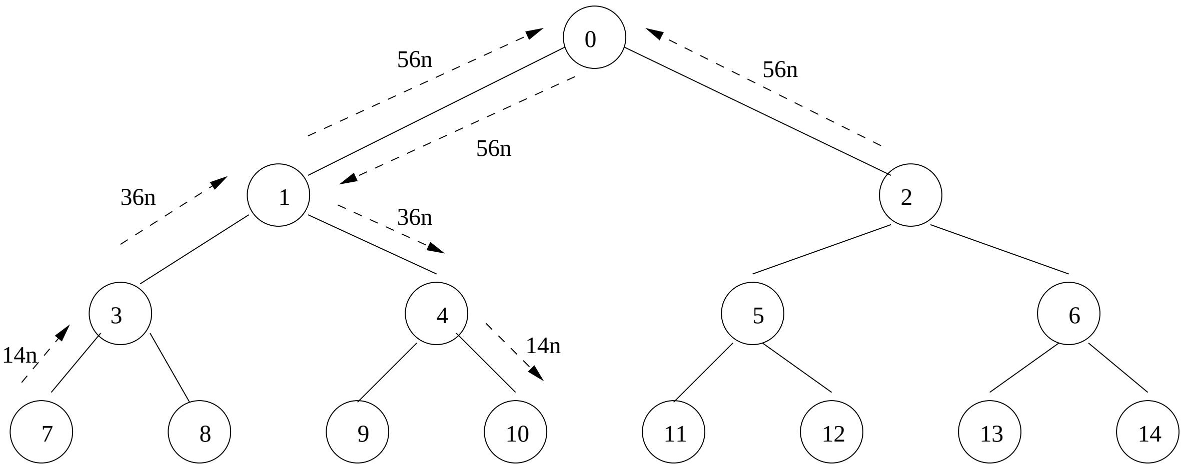

Figure 3.4: Tree based Each-To-All Multicast. Each processor sends a personalized message to every other processor. All the messages are assumed to be of size n.

3.3.2 Mesh

In this scheme, the messages are routed along a 2D mesh. In the first phase of the multicast, each node sends the messages to all the nodes in his column. In the second phase, the nodes sort these messages and send it to the nodes in their row. Thus all the messages travel at most two hops before reaching the destination.

In our scheme the mesh is assumed to be a square mesh. When the number of nodes is not a perfect square, the mesh is constructed using the next higher perfect square. This gives rise to holes in the mesh. Figure 3.5 illustrates the mapping of nodes to holes. The holes in a row are evenly mapped to the nodes in that row. Thus nodes 14 and 15 are mapped into nodes 12 and 13 respectively. No messages are sent to a row with no nodes in it.

As in the case of the tree-based scheme the local messages are consumed only at the final stage. The final phase is entered when a node receives messages from all the nodes in its row. Dummy messages are used to communicate vital information on certain occasions.

3.3.3 Hypercube

The hypercube based scheme (Dimensional Exchange) consists of $log_2(p)$ stages, where p is the number of processors. In each stage, the neighboring nodes in one dimension exchange messages. In the next stage, these messages are combined and exchanged between the nodes in the next

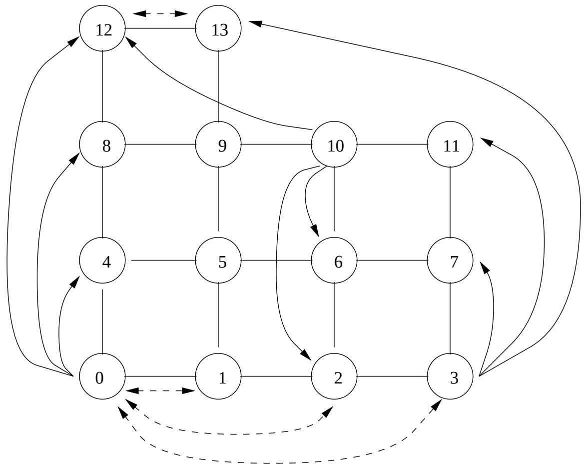

Figure 3.5: Mesh based Each-To-All Multicast. The dotted arrows show the second stage. The holes in the mesh are mapped to the remaining processors in that row.

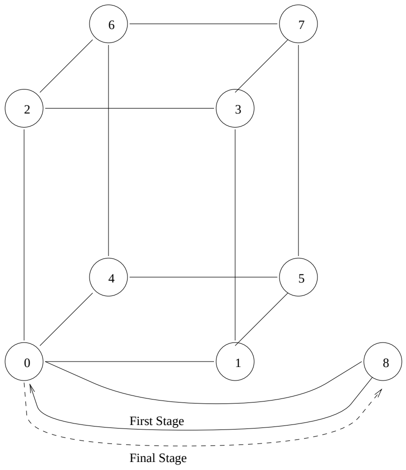

Figure 3.6: Hypercube based (Dimensional Exchange) Each-To-All Multicast. In the case of an imperfect hypercube, a lower dimensional hypercube is constructed.

dimension. This continues until all the dimensions are exhausted.

In the case of an imperfect hypercube (when the number of nodes is not a power of 2), the next lower hypercube is formed. In the first step, the nodes that are outside this hypercube send all their messages to their corresponding neighbor in this hypercube. For example, in Figure 3.6, node 8 sends it messages to node 0 in the first stage. As in the usual scheme, dimensional exchange of messages happens in this hypercube. All the messages for node 8 are sent to node 0. In the final stage, node 0 combines all the messages for node 8 and sends it to node 8, which then delivers it to the user program.

As in the previous schemes, no messages are buffered across iterations and dummy messages are used sparingly.

In all the above schemes, if the total size of the messages for a particular processor exceeds a threshold, they are dispatched directly to the processor. The threshold is calculated using the formula :

$$Thresholdsize = \frac{\alpha}{((No.of hops - 1) \times \beta)} + margin \tag{3.1}$$

The number of hops is determined by the algorithm that is used to route the message. The formula essentially says that if the time to transmit size bytes (hop-1) times is greater than the overhead for sending one message, then send the message directly. The second term in the RHS is to account for the extra copying overhead involved in sending the message directly. The message is not dispatched directly to the user handler. This is because, the user program may exit the scheduler on receiving that message, jeopardizing the ongoing multicast operation. To avoid this the message is buffered on the destination node and is delivered to the user program in the final stage of the multicast operation.

Multicast based on Broadcast

In many shared memory architectures, broadcast can be implemented efficiently by copying the message to the shared memory and en-queuing the messge pointer in all the destination processors receive queue. To take advantage of this scheme, a broadcast based multicast strategy was included in the library. The messages on a particular processor are combined and broadcast to all the processors. The destination processors for the individual messages are encoded in the header of the combined message. On receiving the message, the processors check the header to see if the message belongs to them or not. The messages destined for them are consumed, while the extraneous ones are dropped.

Multicast based on Repeated Sends

The basic method of sending the messages to the different processors in a loop to perform multicast was included in the library as the fifth strategy. All the other methods were compared against this scheme.

3.4 The Message Buffer

The user messages that are not delivered instantly need to be buffered. Since accessing the buffer is a very common operation, major effort was expended to optimize the message buffer. To avoid unnecessary copying, the message pointer was stored, rather than copying the whole message. The current design evolved after an extensive study of the buffer access patterns. The common operations are :

- Insert a message for a list of processors
- Retrieve the messages for a list of processors

Messages intended for a list of processors are enqueued in the message queue for each of those processors as shown in Figure 3.7. While retrieving the messages for a list of processors, the common messages should not be duplicated. A comparison of the message pointers would not suffice, since this solution is not scalable. In order to capture this feature, a magic field is stored along with each message. The magic field is essentially a monotonically increasing counter, which can be used to detect common messages. When a message is retrieved, the magic field is set to the current value of the counter. Every other access to the message will find the magic field to be equal to the current counter value, and avoids duplicating it. Notice that a running counter is needed to differentiate between the different stages. Another solution would be to store a processor list bit vector along with each message. However, this would take up more storage and involves a search in the bit vector to find out the destination processors. Message deletion was the other major issue that had to be tackled. Each message was augmented with a reference count, which keeps track of

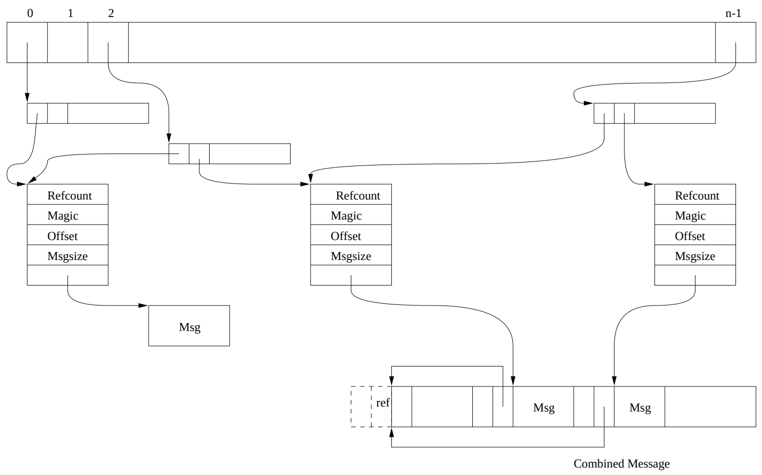

Figure 3.7: Design of the Message Buffer

the number of references to it. The routers were designed in such a way that all the local messages get delivered at the last stage. Thus instead of deleting the message, the pointer is given to the user,who can later delete it. This avoids any copying for local messages.

For each message that is to be buffered, a structure needs to be filled up with the information pertinent to the message (as shown in Figure 3.7). As the number of messages increases the overhead involved in allocating space for the structs becomes significant. To avoid this cost, the structs that are freed are added to a free list. The next time a struct is to be allocated, it is procured from the free list. As shown in Figure 3.7, the messages for a particular processor are stored as an array of pointers. The array is resized to twice its original size when it overflows. This scheme has less overhead as compared to maintaining a linked list.

In order to reduce the cost to retrieve messages from the buffer, two different message extraction mechanisms are supported. Notice that the messages are retrieved either to be delivered directly to the local user program or to be combined with the other messages and packed in to a buffer to be sent to the remote node. These two cases are handled differently. In the case of local messages, the messages are immediately dispatched. That is the message queue for the local processor is traversed just once. For remote messages, the message queue for the list of processors are traversed twice, the first time to determine the total size and the second time to actually copy them into the combined buffer. The messages can be obtained either as a packed buffer or as a message vector. The latter can be used by *CmiVectorSend* which avoids the additional layer of copying.

3.5 Combining Messages

The combined message format is shown in Figure 3.8. In this format, each PE is associated with a list of messages pointed to by the offsets. Another alternative would be to store a PE bit vector with each message. The former case is better if the messages are distinct, while the latter is better if there are lot of common messages. To illustrate this point, consider the case where m messages, each of size s are destined for P processor. Let there be h common messages and k messages per processor. Therefore m can be expressed as :

$$m = P \times k - h \tag{3.2}$$

The size of the combined message for the two cases are shown in equations 3.3 and 3.4 respectively.

$$Size = (k+1) \times P + m \times s \tag{3.3}$$

$$Size = \frac{m \times log_2 P}{8} + m \times s \tag{3.4}$$

Substituting equation 3.2 in 3.4, we can see that as h increases the second format is better and vice versa. The optimal solution would be to switch between the two formats depending on the number of common messages.

One of the major design considerations was to capture common messages in the combined message. This is essential to avoid message duplication and for scalability. While combining messages, the offset of the common message needs to be known. A search of the combined message buffer is very inefficient. So an offset field is stored along with each message in the buffer. Whenever, the message is copied to the combined message, the offset field in the buffer is updated. Thus the magic and the offset field can be used to avoid message duplication completely with no extra overhead. In addition to the message size, an additional field precedes each message in the combined message. This field serves two purposes. At the receiving end, there should exist a mechanism to keep track of the pointers to the common messages, so that they can be enqueued in all the processor message queues. This additional field acts as a place holder for the same. In order to avoid copying,

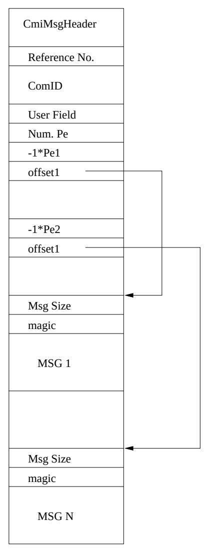

Figure 3.8: Combined Message Format

pointers into the combined message are stored in the buffer. The combined message can be deleted only after all the constituent individual messages are used or deleted. The extra field is used to store the offset to the beginning of the block. Every time an individual message is deleted, it decrements the reference count associated with the block. Once the reference count becomes zero, the block is freed.

3.6 The Switch Manager

The per message cost (α) was the major factor considered in the communication strategies mentioned above. However, there are various other factors like the underlying network topology, the background communication load, the communication pattern of the application etc that contribute to the overall communication cost. Many of these factors like the background communication load, the applications communication pattern etc are dynamic and varies in time. They can be determined only at run time. Moreover the user of the communication library may not be aware of the optimal algorithm that is to be used for his communication needs. The communication library should adapt itself to these situations. Run-time switching capability was introduced for this purpose. Now, depending on the current conditions, the library can switch to the appropriate routing scheme. The communication library will adapt itself to the various communication patterns exhibited by the application.

To switch the router objects transparent to the user, there should exist a mechanism to delineate the boundary between the different iterations of the multicast operation. This functionality came free with the overlap manager. Since all the interactions with the router object go through the overlap manager, it can keep track of the various communication patterns. As a result, the overlap manager was awarded the additional duty of switch management. Initially, the overlap manager empirically estimates the values for the various communication parameters such as the per message overhead (α), per byte overhead (β) etc. The user requests are monitored for any known communication pattern. Based on these parameters, the overlap manager chooses the optimal communication strategy.

In order to avoid any contention between the overlap managers on the different processors, corresponding to a particular instance, a procedure to elect the leader on a per-instance basis, had to be formulated. A simple strategy would be to elect the representative on the processor where the instance was created to be the leader. This is the strategy that is currently employed.

The switching information essentially consists of the time to switch and the new strategy that is to be deployed. The iteration number was used to specify the time to switch. Thus the switching information required just two integers and this was incorporated into the *comID*.

The next hurdle in the path was to publish the switching information to all the processors with minimal performance penalty. Broadcasting the information to all the processors would consume valuable network bandwidth. Moreover, it leads to the race condition, where a processor receives the switching information very late and fails to switch. A clever strategy to circumvent these difficulties could be devised by observing the fact that an *each-to-all Multicast* would deliver messages to all the processors. Since each of these messages contain the *comID*, all the processors can obtain the switching information. However, the library cannot internally generate an *each-to-all Multicast* method invocation, since all the processors need to participate in the operation. Instead, the user request for the next iteration was used to piggyback the switching information. When the decision to switch is taken, the *comID* is encoded with all the pertinent information such as the time to switch, the new strategy etc. The processors that do not belong to the destination set of the user request for the next iteration, is determined. For these set of processors dummy messages are inserted and the user request is transformed into an *each-to-all Multicast* method invocation. Everytime a processor receives a message, it checks the *comID* for any switching information. If present, the overlap manager is updated with this information. At the specified iteration number, the overlap manager deletes the old router object and installs the router object that corresponds to the new strategy. Since this will happen on all the processor, no additional buffering overhead is incurred.

Chapter 4

Performance Evaluation

The performance evaluation of the communication library was conducted on four different architectures-the Linux cluster, the Cray T3E, the SGI Origin2000 and the ASCI Red. The four communication strategies were compared for performance on a range of processors and for various message sizes. An analytical model of the communication strategies based on the various factors that affect the performance is presented in the next section. This is followed by a description of the benchmarks that were used to evaluate these algorithms. The experimental results obtained are analyzed in the subsequent section. To illustrate its applicability, the library was used in a real application namely Namd2. A brief description of this molecular dynamics program and the impact of the communication library on it are presented in the last section.

4.1 Analytical Model

The algorithms involved imposing a virtual topology on the network and performing communication along that topology. The topologies used were:

1. Tree
2. 2D Grid
3. Hypercube

These strategies were compared against the base implementation which consisted of sending the messages repeatedly to each of the processors.

Run time was the main metric that was used to compare the different strategies. There are various factors that could affect the performance of these communication schemes. The important ones are the per message cost (α), the per byte cost (β) and the copying cost (γ). The parameter α measures the overhead involved in delivering the message to the network. This includes the time to setup the connection and the send buffers, the time to fragment the message into packets etc. The time required to transmit a message to the remote node is indicated by β. This depends on the number of byte transferred and the latency of the network. The copying overhead involved in performing a multicast is captured by the parameter γ. A message might need to be duplicated for it to be multicast. This involves message copying which is measured by γ. The cost equation for the different schemes based on these factors are given below.

Repeated Sends

This is the most basic method used to multicast messages. Consider an each-to-all personalized sends in a system with P processors. Let the message size be m bytes. Each processor sends P-1 messages (excluding the local message) of size m bytes. Assuming that the buffer used to hold the message cannot be reused, this operation involves copying the message P-1 times. (Converse provides Asynchronous sends which allows the user to reuse the message buffer. However, the sends can then proceed only in a non-overlapped manner. Thus it is a trade off between the non-overlapped mode of sending messages and the cost to copy the messages). The cost equation for repeated sends is:

$$Cost = (P-1) \times \alpha + (P-1) \times m \times (\beta + \gamma) \tag{4.1}$$

The cost for each-to-all multicast is the same as shown in equation 4.1. This is because multiple copies of the message that is multicast have to be made, which then resembles each-to-all personalized sends.

Tree

Figure 3.4 shows the message transmission pattern along a spanning tree of degree two. All the leaf nodes send one message, the internal nodes send one message up the tree and *degree* messages down the tree except the root which does not send any message up the tree. Thus the worst case α

contribution in a *degree*-ary tree is $(1 + degree) \times \alpha$. As we go higher up the tree, the messages get more and more fat. The maximum size messages are those going to and coming back from the root node of the spanning tree. Consider an each-to-all personalized send operation along a complete tree that spans P processors. The root has *degree* subtrees each with $\frac{(P-1)}{degree}$ nodes. The size of the combined message from any one child to the root is $(P - \frac{(P-1)}{degree}) \times (\frac{(P-1)}{degree}) \times m$, where m is the size of each individual message. The root gets *degree* such messages. It sends the same amount of data down the tree. Thus the worst case cost is given by the equation:

$$Worst - caseCost = (degree + 1) \times \alpha + ((degree - 1) \times P + 1) \times \frac{(P-1)}{degree} \times m \times (\beta + \gamma) \tag{4.2}$$

Even though the α factor is a constant independent of the number of processors, the β and γ cost is of order O(P^2). So tree is not very efficient for large number of processors.

In the case of each-to-all multicast, each processor has just one message that is to be send to all the other processors in the system. This saves the $\frac{P-1}{d}$ factor in the last term of equation 4.2. The worst case cost for each-to-all multicast is given by the equation 4.3.

$$Worst - caseCost = (degree + 1) \times \alpha + ((degree - 1) \times P + 1) \times m \times (\beta + \gamma) \tag{4.3}$$

Grid

In this scheme, the messages are routed along a 2D mesh. In the first phase each processor sends to all the other processors in his column, the data destined for that row. That is, it sends $\sqrt{P} - 1$ messages each of which contains $\sqrt{P} \times m$ bytes. In the second phase, the messages are distributed along the rows.The cost equation for each-to-all personalized sends is shown in equation 4.4.

$$Cost = 2 \times (\sqrt{P} - 1) \times \alpha + 2 \times (P - \sqrt{P}) \times m \times (\beta + \gamma) \tag{4.4}$$

The size of the combined message is just "m" in the case of each-to-all multicast as compared to $\sqrt{P} \times m$ in each-to-all personalized sends. However, the combined message size for the second phase is the same in both the cases. The cost equation for each-to-all multicast is:

$$Cost = 2 \times (\sqrt{P} - 1) \times \alpha + P \times m \times (\beta + \gamma) \tag{4.5}$$

Hypercube

This method, also termed dimensional exchange, consists of log_2P phases. During each phase, the processors along one dimension exchange data. Thus in phase 1, each processor combines the messages for P/2 processors and sends it to its neighbor in that dimension. In the second phase, the messages destined for P/4 processors are combined. But now, each processor has the data it received in phase 1 in addition to its own data. Thus it combines $2 \times (P/4) \times m$ bytes and sends it to its neighbor. The overall cost is represented by the equation 4.6.

$$Cost = log_2P \times \alpha + \frac{P}{2} \times log_2P \times m \times (\beta + \gamma) \tag{4.6}$$

In the case of each-to-all multicast, the message size in each stage follows the geometric progression $1m, 2m, 4m, ...$ The number of terms is log_2P. Thus the cost equation for each-to-all multicast is

$$Cost = log_2P \times \alpha + P \times m \times (\beta + \gamma) \tag{4.7}$$

From the above equations, it can be seen that the tree, grid and hypercube based schemes are better than repeated sends if α is considerably greater than $\beta + \gamma$. In addition to this, reducing the number of messages reduces the network congestion. Again, there will be fewer collisions. However, the above cost model does not take the underlying network topology into consideration. Thus the neighboring processors in the virtual topology might actually be a few hopes apart in the underlying interconnection network. This could result in channel contentions and more collisions. The underlying communication protocol is another factor that is not weighed in the above equations. For example, superior multicast schemes are possible in a worm-hole routed network [17, 21].

The communication library was designed to be portable across different architectures. It provided a framework to test the different strategies on different architectures. The library was augmented with run-time switching capability, thus enabling the choice to be made at run time, when the system knows about both the underlying topology and the user's communication patterns.

4.2 Description of Benchmarks

This section describes the benchmarks that were used to evaluate the strategies mentioned above.

4.2.1 Each-to-all Multicast

This benchmark measures the performance of *each-to-many Multicast*, where Many=All. The communication instance is created on processor 0 and broadcast to all the processors. Each processor performs a fixed number (say 1000) multicast and waits for the messages from the other processors. Once all the messages are received, it sends a message to processor 0. Processor 0 waits for a response from all the processors. From the total time, the time for one multicast operation is calculated.

4.2.2 Each-to-all Personalized Sends

This benchmark is similar to the previous one, except for the fact that the messages are individualized. To achieve this, the number of deposits on each processor was specified to be the total number of PEs in the system. A different message was passed to each multicast invocation. The total time for a fixed number of iterations was measured.

4.2.3 Each-to-many Multicast

In this benchmark, each processor multicast messages to a subset of processors. The subset was determined using the formula $PE <= MyPe$. That is the message was sent to me and all the processors below me. Thus the amount of communication involved depends on the total number of processors. As in the previous cases, the total time for 1000 iterations was measured and the time per multicast was calculated from it.

4.2.4 Each-to-many Personalized Sends

This is similar to the each-to-all personalized sends benchmark. Here the messages are sent to a subset of processors determined by the formula described above.

4.2.5 Run-time Switching

A fixed number of multicast are performed on each processor. The time for each multicast was measured. The time to switch was hard oded in the library. The communication library currently provides the mechanism to switch, but not the strategy. A decision making algorithm should be incorporated into the library to perform automatic switching.

4.3 Experimental Results

The above benchmarks were run on different platforms to evaluate the impact of the various factors like the underlying topology, communication protocol etc. Representative results from each architecture are chosen and analyzed in the following paragraphs.

Linux

The benchmarks were run on a 16 node cluster of Linux PCs interconnected using 100Mbps switched ethernet. Each node consisted of dual 400 MHz Intel Pentium II CPUs. The underlying protocol used by Converse for communication is the connection-less UDP protocol. This eliminates the limitation on the number of PCs in the cluster imposed by the number of open sockets in the case of TCP protocol. To make it reliable, the TCP protocol was spoof-ed by implementing a window-based flow control mechanism. The messages that are delivered using Converse are first fragmented into UDP packets at the sender side and later re-assembled into the original message at the receiver side.

The performance comparisons for each-to-all multicast is shown in figure 4.1. The copying cost on each node is the same for the grid, hypercube and repeated send schemes as shown by equations 4.5, 4.7 and 4.1 respectively. The plot for the hypercube has dips at powers of 2 as expected. In the case of imperfect hypercubes, some of the processors are idle. For small no. of processors the cost for the three schemes are almost the same. As we increase the number of

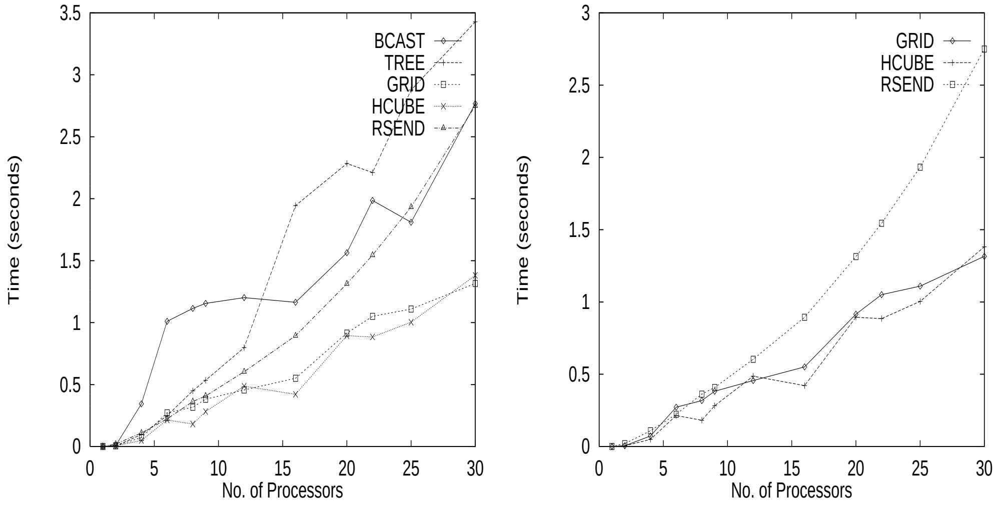

Figure 4.1: each-to-many Multicast Performance on the Linux cluster, where Many=All. Right: Comparing the grid, hypercube (Dimensional Exchange) and repeated sends.The message size was 10,000 integers.

processor, the α factor in equation 4.1 for repeated sends dominates.

The effect of message size is shown in Figure 4.2. For small message sizes, repeated sends work better. This is because the sends are fast and overlapped. The grid and hypercube schemes proceed in stages resulting in increased latency. As the message size increases, more number of UDP packets are generated and the α factor dominates.

Figure 4.3 clearly indicates the trade-offs involved. In this case the equations 4.4 and 4.6 are valid. That is the grid and the hypercube based schemes have extra copying overhead. No extra overhead is incurred for repeated sends as shown by equation 4.1. Moreover, the number of packets increases as the combined message size increases. However, this remains the same for repeated sends. To annul this overhead a large number of processors are required.

T3E

The benchmarks were run on the Cray T3E. The Cray T3E has 512 Digital Alpha 64-bit microprocessors running at 450MHz. The underlying interconnection network is a 3D Torus.

Figure 4.4 presents the performance of each-to-all benchmark for different number of processors. The grid and the hypercube are better than the repeated sends for more than 10 processors , as expected. In the case of each-to-many personalized sends, the grid and the hypercube based schemes have extra copying overhead as compared to the repeated sends. So it requires larger

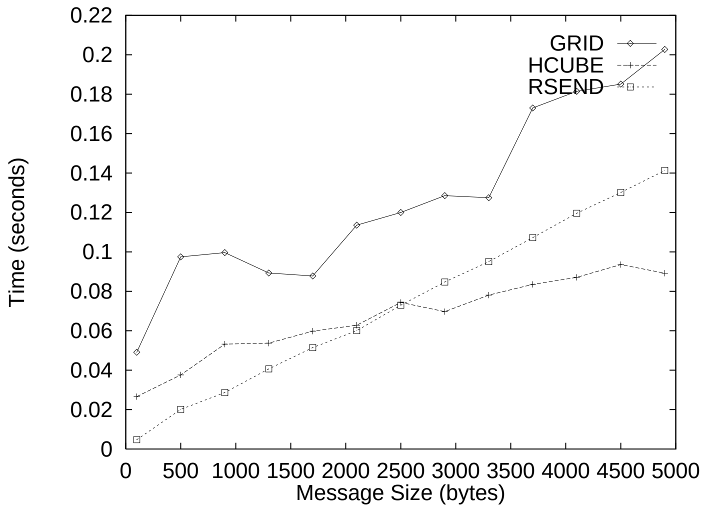

Figure 4.2: Effect of message sizes on each-to-many Multicast (Many=All) on the Linux cluster. No. of processors=8.

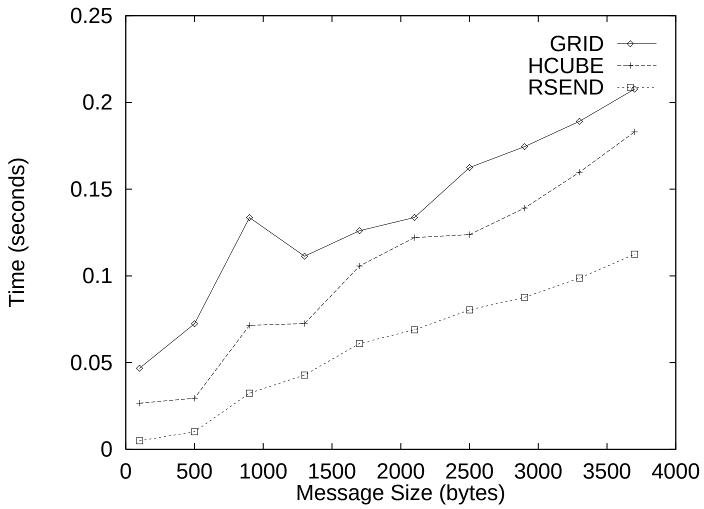

Figure 4.3: Effect of message sizes on each-to-many Personalized sends (Many=All) on the Linux cluster. No. of processors=8.

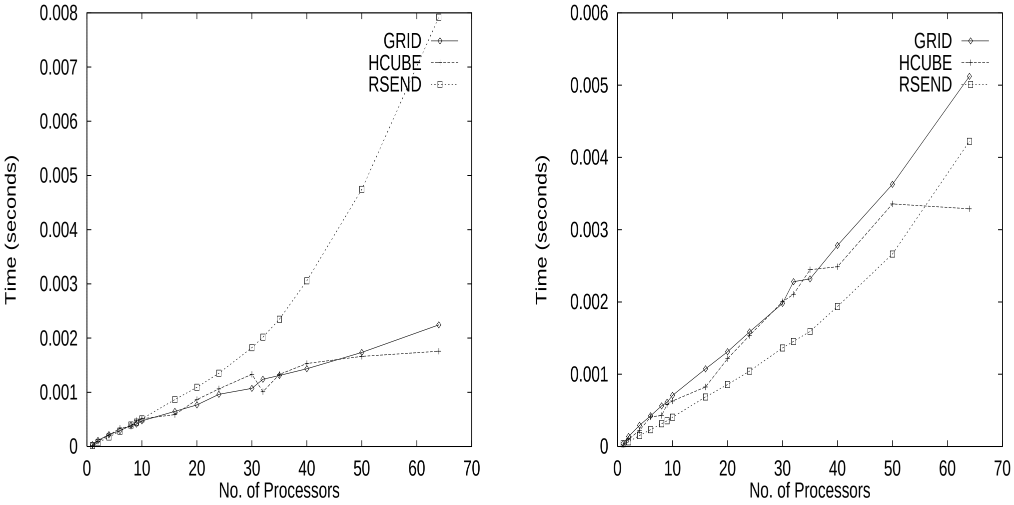

Figure 4.4: Left: each-to-many Multicast Performance on the T3E, where many=all. Right: Each-to-many personalized sends, where many=me+all the pes below me. The message size was 100 integers.

number of processors for the hypercube to catch up with the repeated sends. The horizontal lines for hypercube occur when the number of processors is a power of two.

Origin2000

The Origin2000 is a scalable shared memory multiprocessor powered by the 250MHz R10000 chip. The underlying topology is based on a hypercube. The Converse run-time system ported on top of the MPI [1] implementation was used to run the benchmarks.

Figure 4.5 shows the run time as a function of processors. Expectedly, the grid and hypercube out perform repeated sends as the number of processors is increased.

Figure 4.6 shows the impact of message size on the run time. The benchmark was run on 8 processors. The time for repeated sends, grid and hypercube are comparable for message sizes up to 32KBytes. This is because the copying costs dominate. The sudden jump for larger messages in the case of repeated sends is probably due to some message buffer overflow in the underlying communication layers, which reduced the overlap of copying with sends.

Run-time switching is illustrated in Figure 4.7. The switch happens at the fourth iteration as indicated by the jump in the graph. This is because of the overhead involved in removing the existing router objects, creating the new one and installing them. However, there is no explicit synchronization operation. The plot shows the adaptability of the communication library. The

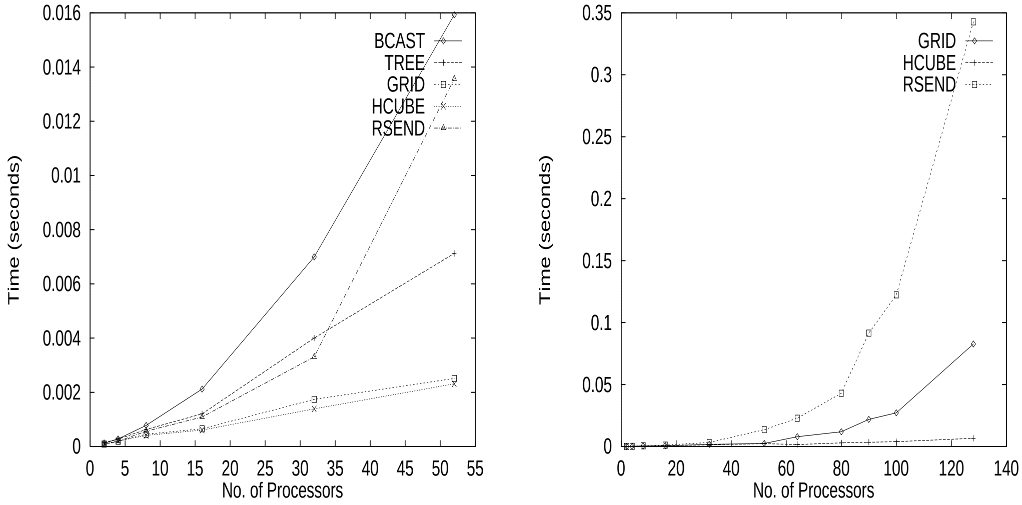

Figure 4.5: each-to-many Multicast Performance on the Origin2000, where Many=all. The message size was 448 bytes Right: Comparing the grid, hypercube (dimensional exchange) and repeated sends.

time to switch in Figure 4.7 was hard coded. However, it is trivial to calculate this based on the timing measurements for the past few iterations.

ASCI Red

ASCI (Accelerated Strategic Computing Initiative) Red [24] is a massively parallel, distributed memory, MIMD, message passing supercomputer powered by 200MHz Pentium Pro processors. It uses an $N \times M \times 2$ split mesh topology. The outing is done along the Z direction first, then X, followed by Y and finally the Z direction again. The benchmarks were run without using the communication coprocessor.

Figure 4.8 compares the different strategies for each-to-all multicast on varying number of processors. The message contained 100 integers. The tree and the broadcast algorithms are worse due to the extra copying overhead associated with it. For small number of processors (up to 32) repeated sends is better because of the overlap of message copying and the send operation. The first term in the equation 4.1 becomes significant as P increases. However, in the case of imperfect hypercubes half the processors are idle while the other half perform double the amount of computation. Since the processors are slow this becomes a significant overhead. As P increases this is offset by the gain obtained in the α term in the equation 4.7. Each-to-all personalized sends involve extra copying overhead for grid and hypercube based schemes as shown by equation 4.4 and 4.6. Since the message size is small, the impact is not as pronounced as indicated by Figure 4.10.

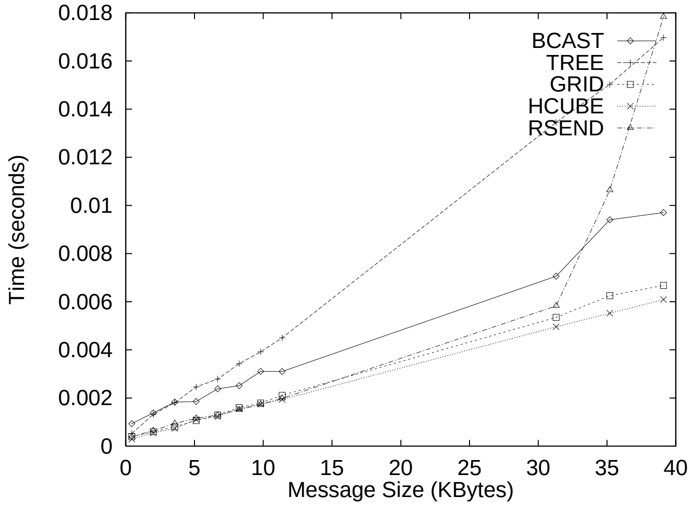

Figure 4.6: Effect of message sizes on each-to-many Multicast (Many=All) on the Origin2000. No. of processors=8.

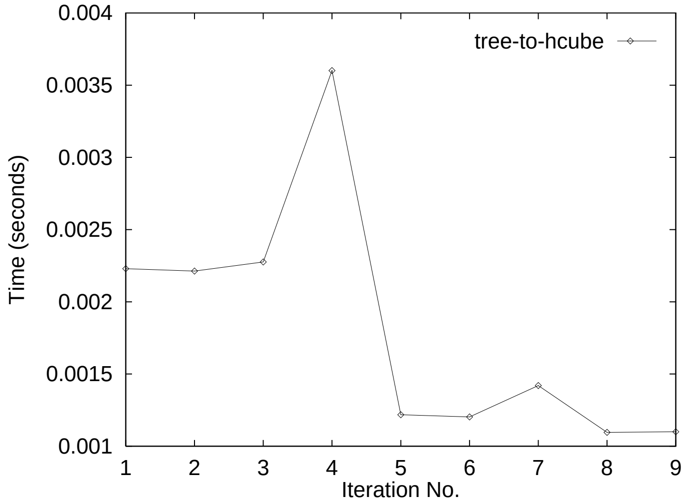

Figure 4.7: Run-time Switching. The library starts off using the tree based scheme and on the 4th iteration switches to dimensional exchange. The experiment was carried out on Origin2000 on 8 nodes with message size=4048 bytes.

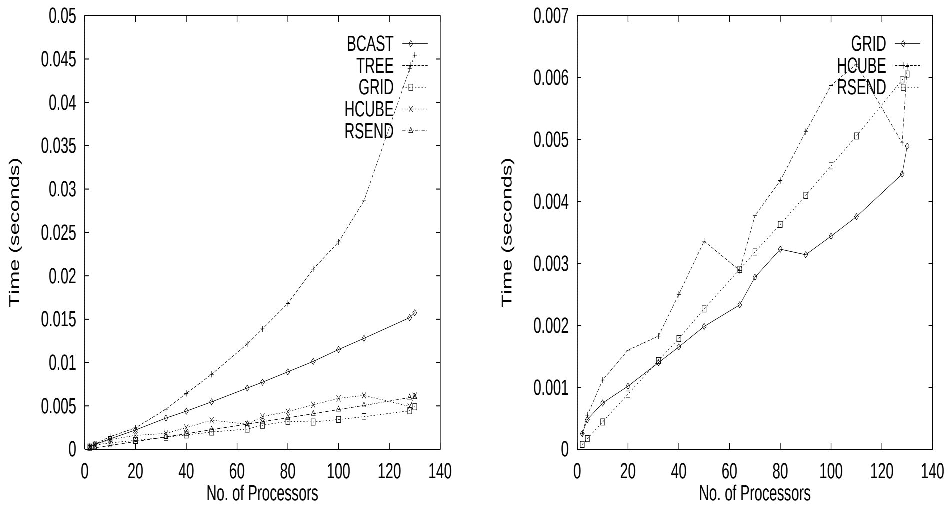

Figure 4.8: each-to-many Multicast Performance on the ASCI Red, where Many=All. The message size=400 bytes. The figure on the right compares the grid, the hypercube (dimensional exchange) and the repeated sends

The overlap of communication with copying is more pronounced in the case of each-to-many personalized sends. Moreover, the grid and hypercube schemes use dummy messages to communicate stage information if there is no user message for a particular stage. As a result of all this overhead, in addition to the extra copying cost in the case of personalized send, the grid and hypercube schemes perform worse than the repeated sends as shown in Figure 4.10.

The impact of the message size on the communication time can be seen from Figure 4.9. From equations 4.4, 4.6 and 4.1, it can be seen than the copying cost on each processor for all the three schemes is of the order O(mP). As m increases more time is spent copying the messages. However, in the case of repeated sends, this cost is masked by the overlap of sends. Even though the copying operation is split into stages in the case of grid and hypercube based schemes, the largest number of bytes copied in a stage is $m \times \sqrt{P}$ and mP/2 respectively. With a slow processor, this overhead is considerable.

4.4 Description of Applications

4.4.1 NAMD2

Namd2 [11] is a production quality parallel molecular dynamics program that is being developed in the Parallel Programming Laboratory group in collaboration with the Theoretical Biophysics

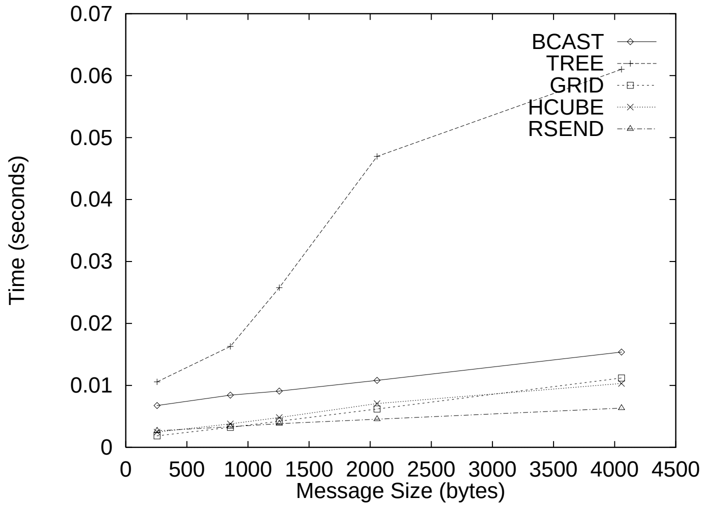

Figure 4.9: Impact of message size on each-to-many Multicast Performance on the ASCI Red (Many=All). The benchmark was run on 64 processors.

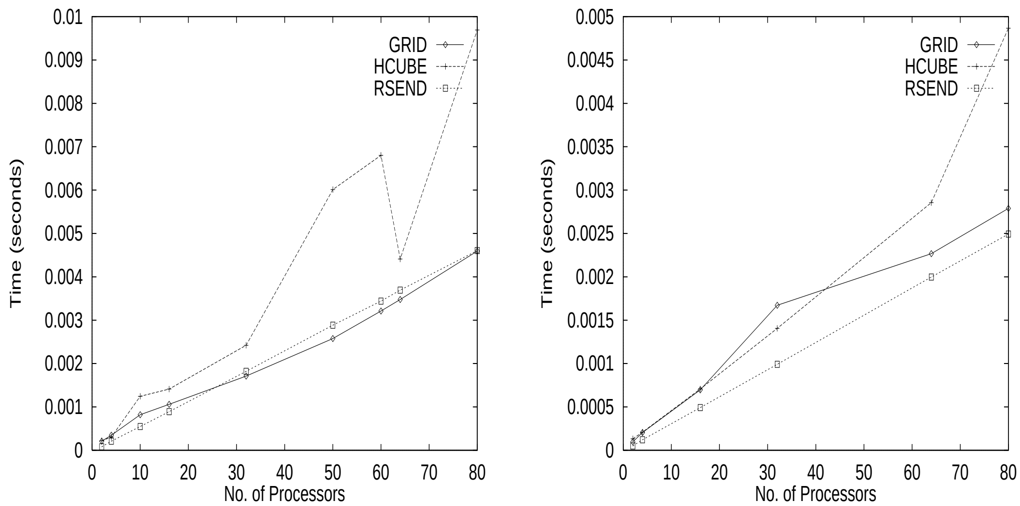

Figure 4.10: Left: each-to-many Personalized Sends Performance on the ASCI Red, where Many=All. Right: Many= me + all the PEs below me. Message size=400bytes.

PE	RSEND	DE
2	7.05	7.05
4	3.62	3.60
8	1.86	1.87
16	0.97	1.04

Table 4.1: Execution time (sec./timestep) for ER-ERE (36,573 atoms, 12Å ccutoff).

group in the Beckman Institute. It is written using charm++ [13, 12], an object based parallel programming language developed on top of Converse.

Namd2 uses a unique hybrid of spatial and force decomposition schemes [11]. The simulation space is divided into regular cubic regions called patches. The patches are responsible for the atoms that are within its boundary. The size of the patch is slightly bigger than the cutoff radius. Thus it need to interact with only its immediate neighbors. The compute objects are responsible for calculating the interactions between the atoms. The compute object depends on the patch to obtain the information regarding the atom coordinates etc. to calculate a particular kind of force. These objects represent units of work and are migratable.Since the number of compute objects is far more than the number of patches, better load balancing can be achieved. A compute object could depend on a patch that is resident on a remote node.In-order to reduce the communication cost, a representative of the patch called a proxy patch is created on the node. The proxies behave just like the home patches. At the end of every time-step it sends its results to the home patch. Similarly the home patches need to update the proxies with the atom positions. This is referred to as proxy population.

The pattern of communication in proxy population is similar to each-to-many multicast. Figure 4.11 shows the communication involved in proxy population. Every time step the patches update their proxies with the current atom positions. A patch can have up to 26 proxies. The message containing the atom position data needs to be multicast to these proxies. Since every patch performs the same operation, a large number of messages are generated. The communication library was used to optimize this part of Namd2.

To use the library in namd2, a charm++ wrapper for it was first developed. The wrapper routine accepts a charm++ message and converts it to a Converse message. This is sent using

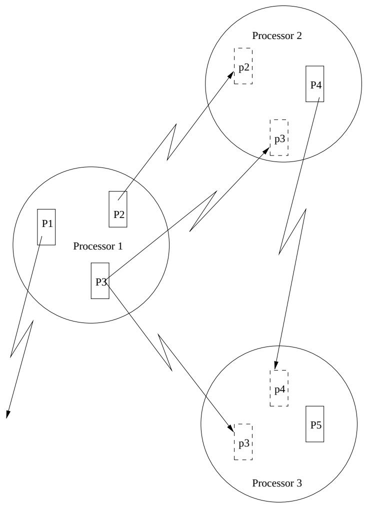

Figure 4.11: Communication involved in proxy population. Px represents the patch and px its proxy. The figure also illustrates the concept of multiple clients on a processor

the communication library. In namd2, multiple patches can reside on a processor as shown in Figure 4.11. The library was then extended to take care of the generic case of multiple clients on a processor. The extension was relatively easy to develop and plug in because of the component based architecture of the communication library. A new object called depositor was developed to deal with the issue of multiple clients on a processor. The depositor talks to the clients and translates them to the interface exported by the communication library. The depositor also keeps track of the number of clients on a processor, accepts their messages and buffers out of order messages. The clients can now specify a sequence number for their messages. The depositor makes sure that the messages are processed in the correct sequence. This feature was needed in namd2 because a patch could go ahead of the other patches in a processor, confusing the communication library. Sequence numbers were used to overcome this problem. At the other end of the spectrum a processor could have no clients. In namd2 this could happen for small molecules. Note that all the processors need to participate in an each-to-many multicast operation. Multicast groups were introduced to relax this restriction. In some situations, a patch might not have any proxies. To keep the number of deposits constant, the dummy deposit call was added to the library.

Atom migration was another area that was optimized using this library. The migration messages are small and numerous. These messages could be combined and sent to their respective processors.

PE	RSEND	DE
2	6.55	6.56
4	3.29	3.30
8	1.65	1.65
16	0.84	0.85

Table 4.2: Execution time (sec./time-step) for ER-ERE (36,573 atoms, 12Å cutoff).

It was a straight forward application of the communication library.

The execution time (sec/time-step) of namd2 with and without the communication library is shown in Tables 4.1 and 4.2. There is no perceptible improvement in performance, because of the few number of proxies that were created. It is worthwhile to note that the communication library did not add any overhead of its own. Performance gains are anticipated for large number of processors, which involves more proxies and possibly more atom migrations.

Chapter 5

Conclusion

Most of the research in the area of communication optimization are specific to an architecture. However, to reduce the cost of developing parallel applications, a mechanism to run applications on the different architectures in an optimal manner needs to be devised. This would require the integration of the different architecture specific optimizations. The communication library presented here attempts to unite the different strategies under a single framework. The library is extensible in that new strategies could be plugged in easily. It also exports to the user a simple and consistent interface. It captures some of the common communication patterns and implement alternate strategies for them in the form of re-usable libraries. The run-time switching capability allows the library to adapt itself to the situation at hand. The communication library was also used to evaluate the performance of each-to-many multicast on various platforms.

5.1 Future Work

The area of adaptive communication optimization needs further exploration. Currently, the library just provides the mechanism to do this. The library should be enriched further by incorporating a decision making algorithm that can impart enough intelligence to determine the time to switch and the strategy to be used. A very simple technique would be to use a measurement based switching algorithm, which essentially tries all the different strategies that it is aware of and chooses the one that minimizes the overall communication time. Certain heuristics can be used to optimize this scheme. Mapping of the user's communication pattern to one of the communication strategies is a more tricky issue. Determining the communication cost for a strategy in terms of α and β is another area of research.

The library currently provides routines to perform each-to-many multicast and each-to-many personalized sends. It could be enhanced with other well known communication patterns.

The each-to-all multicast is currently implemented in terms of each-to-many multicast. This can be optimized further. Some of the message format optimizations mentioned in the section 3.5 could be used to improve the performance further.

Appendix A

Interface

The user interface for the communication library is divided into three sections:

1. Initialization

2. Multicast

3. Strategy Interface

The initialization section describes the routines that are required to create an instance of the communication library. The multicast routines are explained in the second section. A programmer who wish to plug in a new strategy needs to adhere to the interface specified in the last section.

The routines in ***bold italics*** are optional. They provide additional features. The routines in **bold** face are the basic ones.

Initialization

void ComLibInit()

This function initializes the various data structures used by the communication library. It should be called on all the processors.

eg: `ComlibInit(); //in the init routine which is called on all pes`

comID CreateInstance(int ImplType, int NumPes, int NoSwitch)

To use the communication library, the user first needs to create an instance of it. This routine can be called on any processor. It returns a communication id which uniquely represents the particular instance. It is the responsibility of the user to broadcast the *comID* to all the processors that need to use it. The first parameter specifies the communication strategy that is to be used. The second parameter is used to specify the domain of the multicast library. All the processors in this domain should participate in the multicasts. The domain consists of processors 0 to NumPe-1. To specify a random set of processors, the user should use multicast groups, as explained below. The user can disable switching by setting the third parameter to 1.

eg: `comID id=CreateInstance(2, CmiNumPes(), 0); //any processor`

`//Broadcast this id to all the processors`

comID ComlibEstablishGroup(comID id, int npe, int *pelist)

This helps the user to choose a random set of processors as the domain for the multicast operations. The group information is encoded into the id and returned to the user. The group has to be established before any multicast operation. Notice that the used needs to broadcast this id to all the processors in the group. The default group consists of all the processors in the user specified domain (refer: CreateInstance).

eg: `id=ComlibEstablishGroup(id, 2, pelist);`

`// The group consists of pelist[0] and pelist[1]`

ComlibObj * GetComlibObject(comID id)

The user can convert the id into an object reference using this function call. The interface exported by the object is shown in Figure A.1. Notice that the user still needs to use the id for most of the operations.

void DeleteInstance(comID)

This deletes the instance corresponding to the *comID*. The user should not use the id, once it is deleted.

Multicast

void NumDeposits(id, n)

```
class Overlapper {
  public :
        Overlapper(comID);
        ~Overlapper();
        void NumDeposits(comID, int);
        void EachToAllMulticast(comID id, int size, void *msg);
        void EachToManyMulticast(comID, int , void *, int, int *);
        void DeleteInstance();
};

typedef Overlapper ComlibObj;
```

Figure A.1: The Interface exported by ComlibObj

This specifies the number of expected deposits on that processor. The n deposits are treated as one multicast operation. That is, the iteration number that is generated internally, is incremented by just 1. The default value for the number of deposits is 1.

eg: `NumDeposits(id, 2); //The next two multicasts are to be combined`

void EachToAllMulticast(id, size, msg)

All the processors in the domain(group) should participate in this operation. The message is broadcast to all the processors in the group/domain. Thus if the multicast group consists of just 2 processors, then both those processors should participate in this operation and the message is delivered to both the processors. The msg can be any Converse message and is freed internally. The user cannot reuse the message. If there is no data to deposit, the user is allowed to make dummy deposits by setting the size field to 0 and the msg field to NULL. A null deposit usually does not consume any network bandwidth. However, if in a particular iteration a dummy deposit is made on all the nodes, dummy messages are exchanged to communicate information regarding the different stages in the operation.

eg:`EachToAllMulticast(id, sizeof(MYMSG), msg);`

void EachToManyMulticast(id, size, msg, numpe, pelist)

All the processors in the domain should participate in this operation. If a multicast group was established, then the pelist should not contain processor numbers that do not belong to this group. The msg can be any Converse message. The system will internally free the message. So the user cannot reuse the message. The user can specify the list of processors to which the message is to be multicast. If a processor is repeated in the pelist, the message is delivered to the destination handler in that processor twice. Notice that the handlers are NOT delivered combined messages. If the pelist contains just the local processor, the message is delivered to the handler without any message copying overhead. However, the message may not be delivered immediately. As in the previous case, dummy deposits are allowed in this operation also.

eg: `EachToManyMulticast(id, sizeof(MYMSG), (void *)msg, 2, pelist);`

EachToManyPersonalizedSend can be simulated using NumDeposits and EachToManyMulticast calls as in the example shown below.

eg:
```
NumDeposits(id, 2);
int pe=1;
EachToManyMulticast(id, sizeof(MYMSG), msg1, 1,pe);
pe=3;
EachToManyMulticast(id, sizeof(MYMSG), msg1, 1, pe);
```

Figure A.2 shows a very simple example program. It illustrates the ease of using this library.

Strategy Interface

int ComLibRegisterStrategy(NEWFUN newfn)

The user can plug in a new communication strategy using this function. Figure A.3 shows a typical newfn. The type definition of NEWFUN is in comlib.h.

The strategy should be written as follows. It should inherit from the Router class (refer Figure A.4). RecvMsg, ProcMsg and DummyEP act as the *Entry Points* or *handlers*. The handles to the RecvMsg and ProcMsg entry points are stored in the global variable,RecvHandle and ProcHandle. The message format should be as shown in Figure 3.8. The refno is the internal reference number generated by the system. This can be obtained by calling **KActiveRefno(comID id)**. Messages to the DummyEP entry point can be sent using the function **KSendDummyMsg(comID**

```
#include ``converse.h''   //To use Converse
#include ``comlib.h''     //To use the Communication Library
#define GRID 2

CmiHandler DoneHandler(MYMSG *msg)
{
  static int count=0;
  count++;
  CmiPrintf(``%d received data=%d\n'', CmiMyPe(), msg->data);
  if (count==CmiNumPes()) CsdExitScheduler() //I am done, exit
}

CmiHandler StartMcast(MYMSG *m)
{
  MYMSG *msg=(MYMSG *)CmiAlloc(sizeof(MYMSG));
  msg->data=CmiMyPe(); //Fill in some data
  CmiSetHandler(msg, CpvAccess(DoneHandle));
  EachToAllMulticast(m->id, sizeof(MYMSG), msg);
}

//Called on all the processors
void MyInit(int argc,char **argv)
{
  ComlibInit();

  //Create an instance on processor 0
  if (CmiMyPe()==0) {
    comID id=ComlibInstance(GRID, CmiNumPes(), 1);
     MYMSG *msg=(MYMSG *)CmiAlloc(sizeof(MYMSG));
    msg->id=id;
    CmiSetHandler(msg, CpvAccess(StartHandle));
    CmiSyncBroadcastAllAndFree(sizeof(MYMSG), msg);//Broadcast the id
  }
}

main(int argc, char **argv)
{
  ConverseInit(argc, argv, MyInit, 0, 0);
}
```

Figure A.2: A simple Converse program using the communication library. The program illustrates the use of EachToAllMulticast

```
Router * newmystrategy(int n, int me)
{
  return(new MyRouter(n, me));
}
```

Figure A.3: Example of a NEWFUN.

```
//Base class for routers
class Router
{
  private:
  public:
   Router() {};
   virtual ~Router() {};
   virtual void NumDeposits(comID, int);
   virtual void EachToAllMulticast(comID, int , void *, int);
   virtual void EachToManyMulticast(comID, int , void *, int, int *, int);
   virtual void RecvManyMsg(comID, char *);
   virtual void ProcManyMsg(comID, char *);
   virtual void DummyEP(comID, int );
   virtual void SetID(comID);
   virtual void SetMap(int *)l
};
```

Figure A.4: Router base class

id, int pe, int magic) The overlap manager should be notified at the end of each iteration by invoking **KDone(comID id)**. The **SetMap** method is invoked by the overlap manager to set the processor map. The router object should store the pointer passed to it by the overlap manager to map the processors in a multicast group. **SetID** is used to set the comID used by the router.

Multiple clients and reference counting

A new object called the depositor was introduced to handle reference counting with multiple clients on a processor. Consider a processor with two clients on it. The communication library expects two deposits every time-step (or stage). However, if one of the client goes ahead and deposits the data for the next stage, the communication library will consider it as data from the two clients and proceeds to the next step. To handle this situation, the library provides the user with the opportunity to associate a reference number with each deposit. Thus, in the above example, the

```
class Depositor
{
  public:
   Depositor(comID, numDeposits);
   ~Depositor();
   void Deposit(comID, entryPoint, void *msg, chareID, npe, pelist, ref);
};
```

Figure A.5: Depositor Interface

time-step can be used as the reference number. The communication library will buffer the data that arrives too early. The interface exported by the depositor object is shown in Figure A.5.

The constructor argument specifies the number of deposits to expect on a particular processor. The Deposit call can currently be used only from Charm++. It can easily be extended to Converse. Deposit takes the communication id, the entry point, the message, the chareid, the processor list and the reference number as it arguments. It buffers out of order deposits and performs the number of deposits as specified by the constructor argument.

References

[1] MPI: A message passing interface standard. In *M. P. I. Forum*, 1994.

[2] Ron Brightwell andLee Ann Fisk, David. S. Greenberg, Tramm Hudson, Mike Levenhagen, Arthur B. Maccabe, and Rolf Riesen. Massively parallel computing using commodity components. In *Special Issue of Parallel Computing, Parallel Computing on Clusters of Workstations*, oct 1998.

[3] P.S. Rao Ben H. H. Juurlink and Jop F. Sibeyn. Worm-hole gossiping on meshes. In *Euro-Par'96, Parallel Processing*, aug 1996.

[4] Andrew Chien, Pakin, Lauria, and et. al. High performance virtual machines (hpvm): Clusters with supercomputing apis and performance. In *Conference on Parallel Processing for Scientific Computing (PP97)*, 1997.

[5] Wlliam J. Dally. Performance analysis of k-ary n-cube interconnection networks. In *IEEE transactions on Computer*, jun 1990.

[6] Vassilos V. Dimakopoulos and Nikitas J. Dimopoulos. Total exchange in cayley networks. In *Euro-Par'96, Parallel Processing*, aug 1996.

[7] T. Y. Feng. A survey of interconnection networks. In *IEEE Computer*, dec 1981.

[8] David. S. Greenberg and Kevin S. McCurley. Bringing theory to practice: The reality of interprocessor communication. In *Presented at the Workshop on Scalability at the International Parallel Processing Symposium*, 1993.

[9] L. V. Kale, Milind Bhandarkar, Robert Brunner, and Joshua Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on*

Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA, Lecture Notes in Computer Science, March 1998.

[10] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.

[11] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *J. Comp. Phys.*, 1998. In press.

[12] L.V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[13] L.V. Kalé and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[14] C. Laforest. Gossip in tree under line-communication mode. In *Euro-Par'96, Parallel Processing*, aug 1996.

[15] James Phillips Laxmikant V. Kale, Robert Brunner and Krishnan Varadarajan. Application performance of a linux cluster using converse. In *To appear in the Proceeding of the 3rd Workshop of Runtime Systems for Parallel Processing*, April 1999.

[16] Anindya Basu Matt Welsh and Thorsten von Eicken. Low-latency communication over fast ethernet. In *Euro-Par'96 Parallel Processing*, August 1996.

[17] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. In *IEEE Computer*, Feb 1993.

[18] Lauria Pakin and Andrew Chien. High performance messaging on workstations: Illinois fast messages(fm) for myrinet. In *Supercomputing '95*, 1995.

[19] Andrew Geweke Philip Buonadonna and David E. Culler. An implementation and analysis of the virtual interface architecture. In *Proc. of SuperComputing'98*, nov 1998.

[20] L. Prylli and B. Tourancheau. Bip: A new protocol designed for high performance networking on myrinet. In *Lecture Notes in Computer Science*, 1998.

[21] Ranjith Rajagopalan Ran Libeskind-Hadas, Dominic Mazzoni. Tree-based multicasting in wormhole routed irregular topologies. In *International Parallel Processing Symposium and Symposium on Parallel and D istributed Processing*, apr 1998.

[22] David E. Culler Richard P. Martin, Amin M. Vahdat and Thomas E. Anderson. Effect of communication latency, overhead and bandwidth in a cluster architecture. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, june 1997.

[23] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF : A parallel workstation for scientific computation. In *International Conference on Parallel Processing, Vol.1: Architecture*, pages 11–14, Boca Raton, USA, August 1995. CRC Press.

[24] D. Scott T. G. Mattson and S. T. Wheat. A teraflop supercomputer in 1996: the asci tflops system. In *Proceedings of the International Parallel Processing Symposium*, 1996.

[25] Anshul Gupta Vipin Kumar, Ananth Grama and George Karypis. *Introduction to Parallel Computing, Design and Analysis of Algorithms*, pages 66–101. The Benjamin/Cummings Publishing Company Inc., 1994.

[26] Michael S. Warren, Timothy C. Germann, Peter S. Lomdahl, David M. Beazley, and John K. Salmon. Avalon: An alpha/linux cluster achieves 10 gflops for $150k. Technical report, Los Alamos National Laboratories, http://loki-www.lanl.gov/papers/sc98, SC98 Gordon Bell Award Entry, 1998.