Optimizing Collective Communication Operations in ARMCI

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Amina Saify, B.E.

* * * * *

The Ohio State University

2002

Master's Examination Committee:

Prof. Dhabaleswar K. Panda, Adviser

Prof. P. Sadayappan

Approved by

____

Adviser
Department of Computer
and Information Science

© Copyright by

Amina Saify

2002

ABSTRACT

Advances in processor, network and protocol technologies have made clusters of workstations an attractive platform for high performance computing. Emerging applications in cluster environments require frequent data transfer between process memories. This tremendous data transfer requires synchronization at various points to maintain the consistency in data. One-sided communication has gained a lot of attention to support efficient data transfer capabilities for irregular applications. Thus, there is a very strong need to synchronize these one-sided operations at various checkpoints, without affecting the performance of one-sided data transfers.

In this thesis we propose ways to optimize the performance of collective communication operations of the one-sided communication library ARMCI by optimizing the current synchronizing primitive *fence* operation and later by introducing a new *barrier* function. The optimization uses the minimum number of messages exchanged between communicating nodes as well as supports efficient buffer management. The factor of improvement in the performance of the fence operation using our implementation has been observed to be up to 1.72 for an 8-node system. The new barrier function gives a factor of improvement of 3.58 over the previous method of performing the barrier operation on an 8-node system. We have also extended our work to implement a new and more efficient remote lock operation to enhance the performance

of the communication library further. We achieve a factor of improvement of 1.7 over the current implementation of locks in ARMCI.

This is dedicated to my parents, my brother, my sister and my beautiful niece

Sakina

ACKNOWLEDGMENTS

I would like to thank my adviser Prof. D.K. Panda for his support and encouragement through the course of my graduate studies. I appreciate the time and effort he invested in guiding me and steering my research. I am grateful to him for all his invaluable suggestions to help me finish my thesis.

I am grateful to Prof. P. Sadayappan for his advise, insightful comments and agreeing to serve on my Master's examination committee.

I am also thankful to Dr. Neiplocha, PNL for his involvement in this work.

Thanks are also due to the student members of the Network-Based Computing Laboratory, particularly Darius Buntinas and Jiesheng Wu for their willingness to help at all times. Also, I would like to thank a former student of NOWlab, Vinod Tipparaju, for always replying to my e-mails and helping me out.

I am indebted to the OSU CIS department for awarding me teaching assistantship and Prof. Panda for supporting me as a research associate.

Finally, I would like to thank all my friends and relatives who made my stay at OSU enjoyable, especially Shreyas who helped me at every stage of my thesis.

VITA

September 11, 1979 .... Born - Indore, INDIA

2000 .... B.E. Computer Science

2000-present .... Graduate Teaching Associate,
Ohio State University.

PUBLICATIONS

Research Publications

Jarek Neiplocha, Vinod Tipparaju, Amina Saify, Dhabaleswar Panda "Protocols and Strategies for Optimizing Remote Memory Operations", 2002. *Proc. of Communication Architecture for Clusters(CAC'02)*, April 2002.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

High Performance Computing : Prof. D.K. Panda

TABLE OF CONTENTS

Page

Abstract ii

Dedication iv

Acknowledgments v

Vita vi

List of Figures ix

Chapters:

1. Introduction 1
   - 1.1 User Level Protocols 2
   - 1.2 Programming models for Clusters 3
     - 1.2.1 Message Passing 4
     - 1.2.2 Shared Memory 4
     - 1.2.3 Get/Put One-Sided Communication 5
   - 1.3 Problem Statement and Approach 6
   - 1.4 Thesis Organization 8

2. Background and Related Work 9
   - 2.1 User Level Communication Protocol - Myrinet/GM 9
   - 2.2 ARMCI - Aggregate Remote Memory Copy Interface 11
     - 2.2.1 ARMCI Operations 12
     - 2.2.2 Client-server Architecture 12

3. Optimization of Fence Operation in ARMCI 15

3.1 Fence Operation 15
3.1.1 Current Implementation 16
3.1.2 Our Implementation 17
3.2 Performance Evaluation 19

4. Implementation of Barrier Operation in ARMCI 23

4.1 Achieving Barrier Operation in ARMCI 23
4.2 Our Implementation 24
4.3 Algorithm and the Pseudo-code 24
4.4 Performance Evaluation 27

5. Optimization of Lock Operation in ARMCI 31

5.1 Current Implementation 31
5.1.1 Server Based Queue Lock 32
5.1.2 Ticket Algorithm Based Lock 32
5.1.3 ARMCI Implementation 34
5.2 Our implementation 35
5.2.1 Trade-offs of Current Implementation 35
5.2.2 Software Queuing 35
5.2.3 Design Challenges 38
5.3 Performance Evaluation 38

6. Conclusions and Future Work 41

Bibliography 43

LIST OF FIGURES

Figure		Page
1.1	(a)Traditional networking architecture and (b) U-Net architecture [1]	3
1.2	Message Passing Programming model	4
1.3	Distributed Shared Memory Programming model	5
1.4	Get/Put One-Sided Communication Programming model	6
2.1	Client-server Architecture of ARMCI	13
3.1	ARMCI_AllFence Block Diagram	17
3.2	Optimized ARMCI_AllFence	18
3.3	Latency of ARMCI_AllFence operation for 4 nodes	20
3.4	Performance improvement ratio of ARMCI_AllFence operation for 4 nodes	20
3.5	Latency of ARMCI_AllFence operation for 8 nodes	21
3.6	Performance improvement ratio of ARMCI_AllFence operation for 8 nodes	21
4.1	Block Diagram for new implementation of the Barrier operation	25
4.2	Pseudo-code for the Barrier function	26
4.3	Latency of ARMCI_Barrier with no delay	28

4.4 Latency of ARMCI_Barrier with different delays for message size 100 bytes 29

4.5 Latency of ARMCI_Barrier with different delays for message size 4000 bytes 29

5.1 Example of a ticket lock 33

5.2 Example of a ticket unlock 33

5.3 Example of a distributed lock 37

5.4 Latency of ARMCI_Lock 39

5.5 Latency of ARMCI_Unlock 39

CHAPTER 1

INTRODUCTION

The importance of parallelism in satisfying the application demand for ever greater performance can be brought into sharper focus by looking more closely at the advancements in the underlying technology and architecture. Parallelism is becoming the basis for solving harder and bigger problems. The quest for performance is so keen that parallelism is being exploited at many different levels and at various points in the computer design space. Such parallelism is being exhibited even on desktop systems with multiple processors.

The transition to parallel programming, including new algorithms, or attention to communication and synchronization requirements in existing algorithms, has largely taken place in the high-performance end of computing. This has increased the use of collection of workstations on a fast network, also known as *clusters* [2], in the world of high-performance computing. Performance of inter-processor communication plays an important role in these clusters. Designing high performance communication subsystems for these clusters is a major challenge. Solutions to this challenge include exploiting the capability of the underlying network and the high speed Network Interface Cards (NICs) [13], while trying to satisfy the communication requirements of

the upper level programming models. In the following sections we provide a brief overview related to these issues before describing our problem statement.

1.1 User Level Protocols

The increased availability of the high-speed local area networks has shifted the bottleneck in local-area communication from the limited bandwidth of network fabrics to the software path traversed by messages at the sending and receiving ends. In particular, in a traditional UNIX networking architecture, the path taken by messages through OS involves several copies and crosses multiple levels of abstraction between the device driver and the user application. The resulting processing overheads limit the peak communication bandwidth and cause high end-to-end message latencies.

This has triggered a lot of research in areas related to utilizing smart NICs and processors on the smart NICs in protocol processing. This has led to the development of the OS bypass protocols or user level protocols [1]. User Level Protocols and their implementations on programmable network interface cards have been alleviating the communication bottleneck for high speed interconnects. User level protocols address these issues by making sure that the parts of the protocol or the entire protocol is moved to the user space from the kernel space. One of the first examples of user level protocols is U-Net.

Figure 1.1 shows the difference between the traditional networking architecture and the user level protocol architecture, like U-Net. In traditional systems, the NIC would simply take the data from the host and put it on the interconnect and upon receiving the data simply forward it to the host node for processing. However, modern high speed interconnects such as Myrinet [3] make use of modern smart NICs

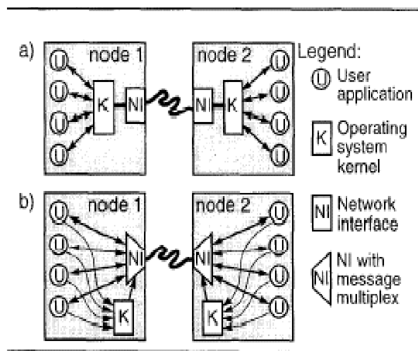


Figure 1.1: (a)Traditional networking architecture and (b) U-Net architecture [1]

which have programmable processors and memory, which make them capable of sharing some of the message processing work with the host. Hence, host can dedicate more cycles to the computation thereby enhancing application speed up. Thus, the use of smart NICs have removed the overhead of kernel processing and reduced the communication latency by reducing the network latency.

1.2 Programming models for Clusters

The basic processing element from PCs to large systems, is rapidly becoming a symmetric multiprocessor system (SMP). As a result, the nodes of a parallel computer will often be an SMP. The resulting mixed hardware models (combining shared-memory and distributed memory) provides a challenge to system software developers to provide users with programming models that are portable, understandable, and efficient.

Generally three programming models are widely used for programming in cluster environment: shared-memory, message passing and get/put one-sided communication.

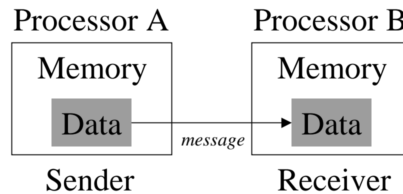


Figure 1.2: Message Passing Programming model

1.2.1 Message Passing

By *message passing* [4] we mean the transfer of data between processes by send/receive operations, in which both processes must participate in the data transfer. The data owner must know when and which process needs the data and data transfer implies a form of synchronization between the sender and receiver. Asynchronous send/receive operations might diffuse the synchronization point, but the cooperation is still required. Figure 1.2 shows the block diagram of message passing programming model.

1.2.2 Shared Memory

A *shared memory* system makes a global physical memory equally accessible to all processors. These systems offer a general and convenient programming model that enables simply sharing through a uniform mechanism of reading and writing shared data structures in the common memory. Since shared-memory system is very flexible, a new concept *Distributed Shared Memory* [5] system has emerged to provide the advantages of shared-memory system on distributed machines. A DSM system logically implements the shared-memory model on a physically distributed system. DSM mechanism allows a process to access shared data which is not physically resident

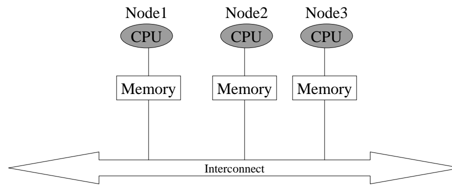


Figure 1.3: Distributed Shared Memory Programming model

on that machine. When a process accesses data in the shared address space, the shared memory address maps to the physical memory address. The mapping is done by a layer of software implemented either in the operating system kernel or as a runtime library. To coordinate access to the shared memory by multiple processes, some form of synchronization, like locks, barriers, semaphores and monitors, must be provided. Figure 1.3 shows the block diagram of the DSM model.

1.2.3 Get/Put One-Sided Communication

In *one-sided communication*[7] a process can be allowed to access dedicated segments of memory of another process for reading, writing or updating, without the explicit participation of the other process. Such remote accesses take effect after an appropriate synchronization operation is performed. A distinguished feature of one-sided communication is that only one process is responsible for initiating the communication between two processes, and must supply all necessary parameters for the communication operation. The *get/put* one-sided communication model uses get and put operations to implement the one-sided communication. A *get* operation is a remote read operation and a *put* operation is a remote write operation. Both get and put operations access remote memory without any intervention from the remote node. One-sided communication certainly has a lot of advantages in many application

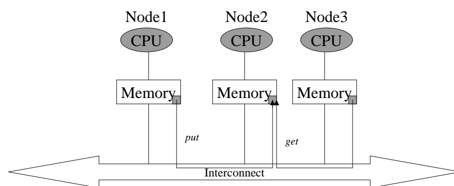


Figure 1.4: Get/Put One-Sided Communication Programming model

categories, specifically applications related to the computational chemistry where a process needs to make unpredictable reference to remote data. Figure 1.4 shows the schematic diagram of get/put one-sided communication model.

1.3 Problem Statement and Approach

One-sided communication has gained a lot of attention in research community because of its simple progress rules and no cooperation required from the receiver side. As a result, one-sided communication is becoming popular and the MPI standard also has defined a set of one-sided calls in its 2.0 release [7]. The SHMEM library developed by Cray is a one-sided library for the CrayT3E and SGI origin systems for contiguous data transfer[6]. The Aggregate Remote Memory Copy Interface (ARMCI) [6] is another library that offers one-sided communication calls for remote memory copy functionality mainly focusing on non-contiguous data transfer.

Ordering of different operations in a one-sided communication library is very important as it simplifies programming and is required in many applications such as computational chemistry. Some systems enforce ordering of the unordered operations by providing synchronization operations. Synchronization operations are essential for ordering operations and thus maintaining data consistency, though they could have a negative impact on the application performance. This motivates the need to develop

synchronization operations on a given platform in the most efficient manner, which can achieve ordering with a lower overhead inside the communication system.

To implement the synchronization operations in the most efficient manner there are three major issues we need to consider:

1. Reduce the number of messages required to communicate among all the participating computing nodes during the synchronization.

2. Reduce the number of buffers required to store control messages.

3. Reduce the waiting time of a computing node to come out of the synchronization operation initiated by it or some other node, participating in the synchronization operation.

In this thesis we address these major issues for ARMCI - A portable aggregate remote memory copy interface. We start with the current synchronization primitives in ARMCI such as fence operation and locks, optimize them and eventually replace them with new and better implementations. The optimization and new implementation is done for ARMCI on the Myrinet/GM communication layer. We address the above mentioned issues as follow:

1. Optimize the fence operation in ARMCI on GM, by reducing the time for a computing node to come out of the fence operation.

2. Implement a new barrier operation in ARMCI on GM which requires less number of messages and less wait time.

3. Implement a new lock algorithm which reduces the contention on one node.

By incorporating our ideas for addressing these issues, we achieve the improvement in the performance of the fence operation up to 70% for an 8-node system. The new barrier function gives up to 138% improvement over the previous method of performing the barrier operation on an 8-node system. The implementation of a new and more efficient remote lock operation, we achieve up to 40% improvement over the current implementation of locks in ARMCI.

1.4 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2 we discuss about user level communication protocol Myrinet/GM and the basic concepts of the ARMCI and general communication paradigm of ARMCI. In Chapter 3 we describe the optimization of fence operation in ARMCI on GM. In Chapter 4 we describe the implementation details of the new barrier function in ARMCI with some experimental results. In Chapter 5 we describe how to implement distributed locks in ARMCI on GM with some experimental results. In Chapter 6 we conclude and discuss future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter we talk about the user level communication protocol GM in general. We also briefly discuss the organization of ARMCI-Aggregate Remote Memory Copy Interface, as a portable one-sided communication library and synchronization operations supported by ARMCI. Later, we provide a brief overview of the architecture and implementation details of ARMCI.

2.1 User Level Communication Protocol - Myrinet/GM

In this section, we discuss the communication protocols for Myrinet, a modern, worm-hole routed, network technology for local and system area networks. GM [8] is a low level message passing system for the Myrinet network. The GM system includes a driver, the Myrinet-interface control program and the GM API, the GM library and header files. GM features include:

1. concurrent, protected, user-level access to the Myrinet interface

2. reliable, ordered delivery of messages

3. automatic mapping and route computation

4. automatic recovery from transient network problems

5. scalability to thousands of nodes

6. low host-cpu utilization

During the execution of a program the driver is used mainly for opening *ports*, pinning and unpinning memory, and to put process to sleep or wake them when blocking functions are used. A *port* is an abstraction through which a process can communicate with the NIC. Once a port is opened, the process can communicate with the NIC, bypassing the operating system and avoiding system call overhead. Each NIC can support up to 8 ports, some of them are reserved.

Flow control is used to avoid the buffer overflows. GM uses the concept of tokens to provide flow control and reliability. When a process opens a port, it has a certain number of *send tokens* and *receive tokens*. Each send event requires a send token. To send a message, a process fills in a send token describing the send event and queues it on the send token queue. The NIC polls this queue for new send tokens. When the NIC gets a new send token it DMAs the data from the specified host buffer, and transmits the packet to the destination. Once the NIC has completed the send, and has freed the resources corresponding to that event, the send token is returned to the process. The host should not modify the data, which is to be sent, until the send token is returned from the NIC.

In order to receive a message, the process must allocate a buffer into which the message will be received and pass a receive token describing the buffer to the NIC. Once the NIC has DMAed the data from a receive message into the buffer, the receive token is returned to the process. Messages may only be sent from and received into buffers which are pinned in memory. Memory is pinned using special functions supplied by GM.

GM is a connectionless model because there is no need for the user process to establish a connection with a remote host. Once the mapping of destination address to routing paths is completed, a user process simply builds a message and sends it to any host in the network. In a large scale Myrinet network, proper mapping of destinations to routing paths is essential to provide deadlock free communication.

GM is a lightweight communication layer, and as such it has certain limitations including inability to send messages from or receive messages into non-DMA-able memory, lack of support for gather and scatter operations and inability to register shared memory under Linux.

2.2 ARMCI - Aggregate Remote Memory Copy Interface

ARMCI, *aggregate remote memory copy interface*, is a one-sided communication library that offers remote memory copy functionality. It aims to be fully portable and compatible with message-passing libraries such as MPI or PVM. ARMCI offers both simpler and low-level model than MPI-2 one-sided communication to streamline the implementation and improve its portable performance. The ARMCI specification does not describe or assume any particular implementation model, for example threads.

In scientific computing, applications require transfer of non-contiguous data transfer. With remote copy APIs supporting only contiguous data transfer, it becomes very inefficient, transferring non-contiguous data into contiguous block of data using multiple communication operation.

The Aggregate Remote Memory Copy Interface [6] is a one-sided communication library targeting non-contiguous data transfer. In particular, ARMCI is meant to be

used primarily by library implementors rather than application developers. Examples of libraries that ARMCI is aimed at include Global Arrays [10], P++/Overture and PCRC Adlib run time system.

It is very important for a communication library such as ARMCI to have straightforward progress rules. Simple progress rules simplify the development and performance analysis built on top of libraries that use ARMCI, and avoid dealing with ambiguities of the platform-specific implementations. Therefore, the ARMCI remote copy operations are truly one sided and complete regardless of the actions taken by the remote process.

Applications that frequently use hybrid programming model require a compatibility with message-passing. Both blocking and non-blocking APIs are needed.

2.2.1 ARMCI Operations

ARMCI supports the following classes of operations:

1. Data transfer operations such as *get* and *put*.
2. Atomic operations such as *accumulate* and *read-modify-write*.
3. Distributed mutex operations such as *lock* and *unlock*.
4. Progress and ordering such as *fence* and *allfence*.
5. Memory allocation such as *malloc*, *cleanup*, *abort* and *error*.

2.2.2 Client-server Architecture

To support the full set of remote memory operations on clusters of workstations with GM protocol, ARMCI uses *client-server* architecture [9]. Figure 2.1 shows the

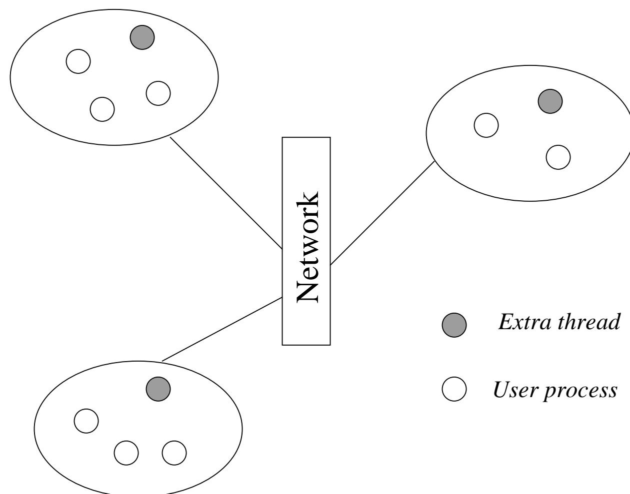


Figure 2.1: Client-server Architecture of ARMCI

client-architecture on SMP. It is implemented by starting on each machine “server” thread(s) dedicated to remote memory operations that are issued by remote clients (user tasks). If the implementation of network protocols is not thread-safe, a heavy-weight process can be used instead. The server thread upon receiving a request executes a handler function corresponding to the appropriate remote memory operation and if needed sends data back to the client. The optimal number of server threads needed depends on several factors such as number of processors and user tasks/processes per node, network throughput and the communication load and patterns in applications. For performance reasons, on the current networks and hardware with low number of processors per SMP node, a single thread is appropriate. In libraries that offer specific interfaces for memory allocation such as MPI-2 and ARMCI, one thread could suffice since their memory allocation operations can allocate the shared memory. Otherwise, one thread per user process would be required. The combination of server threads, network protocols and OS support for mutual

exclusion is sufficient to implement a full set of remote memory operations and also deliver high performance. With this architecture, special care is required to minimize the resource consumption for the benefits of applications.

To prevent server thread/process, in the absence of one-sided communication requests, from consuming CPU resources needed by user processes, blocking wait rather than active polling of the network interfaces is appropriate. GM offers blocking communication calls that effectively block the calling thread until an associated communication event occurs.

With this architecture in mind we explore the implementation of synchronization primitives, *fence*, *barrier* and *locks* of the ARMCI communications library on GM.

CHAPTER 3

OPTIMIZATION OF FENCE OPERATION IN ARMCI

ARMCI supports synchronization operations with data transfer operations. Some of the synchronization operations supported by ARMCI are local and global fence, atomic read-modify-write and mutex operations. In this chapter, we focus on the fence operation.

3.1 Fence Operation

When a blocking put operation completes, the data has been copied out of the calling process memory but has not necessarily arrived at its destination. This is a local completion. A global completion of the outstanding put operations can be achieved by calling ARMCI_Fence or ARMCI_AllFence. ARMCI_Fence blocks the calling process until all the put operations issued by it to the specific remote process complete at the destination. ARMCI_AllFence does the same for all the outstanding put operations issued by the calling process regardless of the destinations.

The *fence* operation assures that all the outstanding remote memory operations issued by the calling process are complete. This is important, for example, in critical sections of the code, to assure that changes to protected data are complete before releasing a mutex. The fence operation applies only for the remote store operations.

Its implementation is closely connected to the underlying network and remote memory operation protocols.

3.1.1 Current Implementation

Since ARMCI has a client-server architecture, there is one server thread and a number of client threads running on a node. The function ARMCI_Fence sends a request for acknowledgment to the remote server on which the client has issued a *put* operation. The ARMCI_Fence operation is specific to one destination only. The function ARMCI_AllFence calls ARMCI_Fence for each server on which computations are going on. The main function of the fence operation is to block the client until it gets an acknowledgment from each server. The client thread starts all the requests for the computation and the server thread does all the data transfer and computation on data. The client thread on each node maintains a list of servers on which it has started computations. In the ARMCI_AllFence operation, the client thread on the client node sends a request for acknowledgment to each of the server in the list. Each server thread on the server node then sends an acknowledgment to the client node once it finishes all the computations issued by that particular client node.

In the original implementation of ARMCI_AllFence, the client node sends a message to the server node and waits for the acknowledgment and then sends a message to the next server and so on, as shown in the figure 3.1. As a result, too much time is spent in waiting for the acknowledgment. The number of messages required in this scheme is 2*(N-1)+lnN, where N is the number of nodes

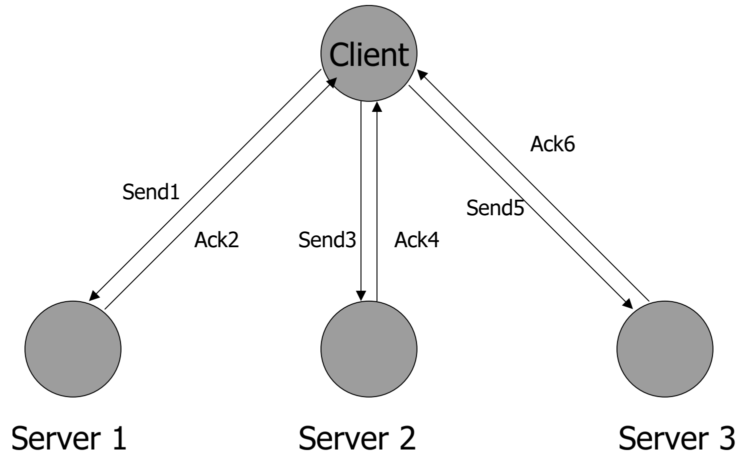


Figure 3.1: ARMCI_AllFence Block Diagram

3.1.2 Our Implementation

In our approach, instead of sending a message to one server, waiting for the acknowledgment from it and then going to next one, we have modified this function in such a manner that it sends a request for acknowledgment to a set of K servers and then receives acknowledgment from each of these servers, as shown in figure 3.2. K can be equal to the number of servers in the list of servers maintained by the client node. But as the number of nodes can be large for scientific applications, the number of buffers required for sending those many messages will require very large amounts of memory. Hence, we have restrained our implementation to K consecutive request messages, where the value of K depends on the number of buffers available for sending the requests.

Suppose a client node has N server nodes in its list of servers, we send a request message to the first K server nodes in the list and wait for the acknowledgment from each of these servers. By doing, so we are overlapping the wait time for the

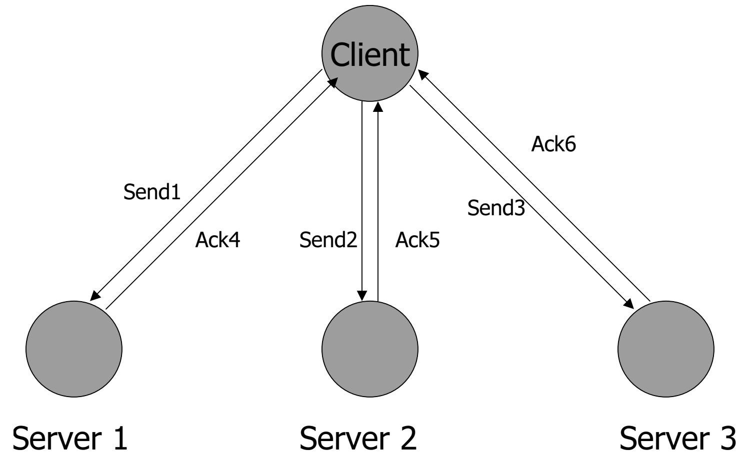


Figure 3.2: Optimized ARMCI_AllFence

acknowledgment with the send time for the next request message. Similarly, we send the request message for the next K servers in the list and wait for acknowledgment from these servers and so on till we have sent request messages to all the server nodes and have received acknowledgments from all of the server nodes. If N is not an exact multiple of K, we repeat the above procedure for $\lceil N/K \rceil$ times plus one iteration for the remaining servers on the list.

In our scheme we also tried to minimize the contention on a particular server. If all the nodes participating in a fence operation send a request for an acknowledgment to a particular server, the server gets overloaded, which increases the delay in response. Hence, we randomize the issuance of requests by a client on different servers so that all the clients do not send a request message to the same server at the same time, which reduces the contention on a particular server.

3.2 Performance Evaluation

The tests were performed on an eight node cluster of quad Pentium III 700MHz machines (Dell 6400's), which have LANai 7.2 cards with 66MHz NIC processors, running Linux with kernel version 2.2.17. The machines are connected by a Myrinet LAN network with LANai 7.2 cards via an 16-port switch.

We tested the performance of our approach and compared it with original implementation. In each iteration, the client node issues an accumulate operation on the server node followed by a fence operation. We timed these tests for 1000 iterations, and then the average was computed for the result. Figure 3.3-3.6 show the results of this test. Fig. 3.3 and 3.5 are the latency graphs for the fence operation. Fig. 3.4 and 3.6 show the performance improvement for 4 nodes and 8 nodes, respectively.

In case of 4 nodes, we performed the test for two cases. In the first case, each client issues a number of *put* operations on two remote servers and in the second case each client issues a number of *put* operations on three remote servers. For the first case, we performed the test by taking K = 1,2, where K is the number of buffers required to send the request messages. For the second case, we performed the test by taking K = 1,2,3. We observed that by providing more number of buffers the performance improves even more. For each case, we took the average of latencies of all the client nodes. The maximum factor of improvement we achieved for 4 nodes is 1.31 over the current implementation.

Similarly, in case of 8 nodes, we performed the test for four cases. In the first case, each client issues a number of *put* operations on two remote servers, in the second case each client issues a number of *put* operations on four remote servers, in the third case each client issues a number of *put* operations on six remote servers and in the fourth

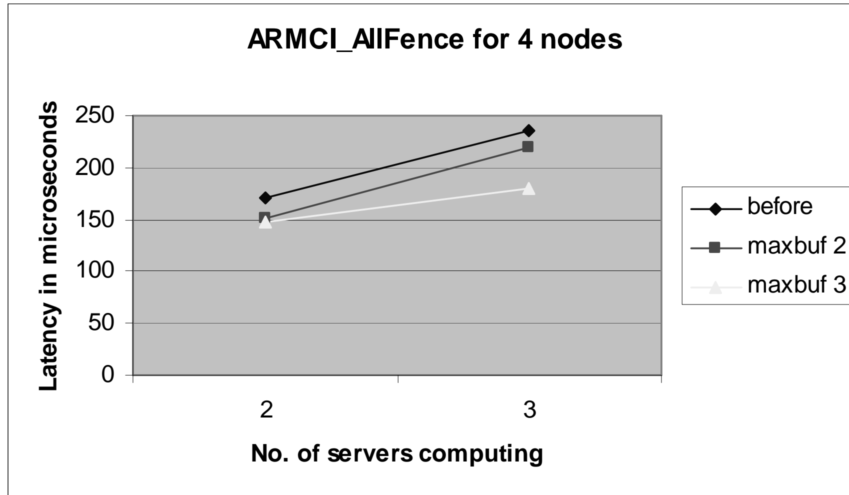


Figure 3.3: Latency of ARMCI_AllFence operation for 4 nodes

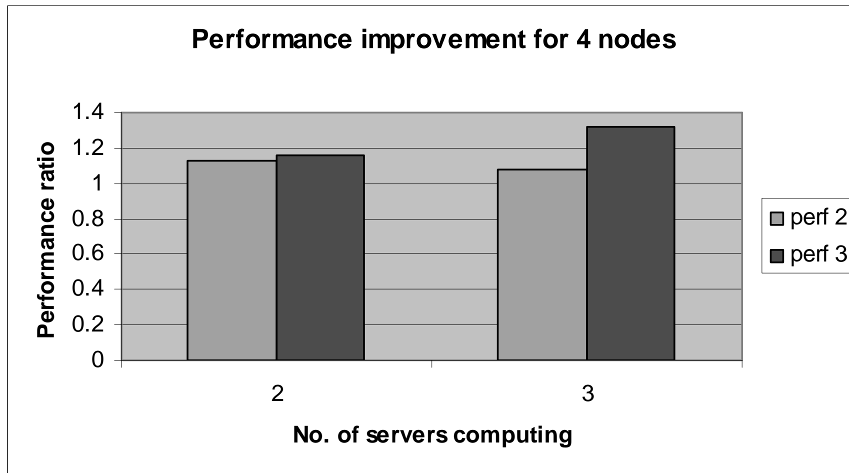


Figure 3.4: Performance improvement ratio of ARMCI_AllFence operation for 4 nodes

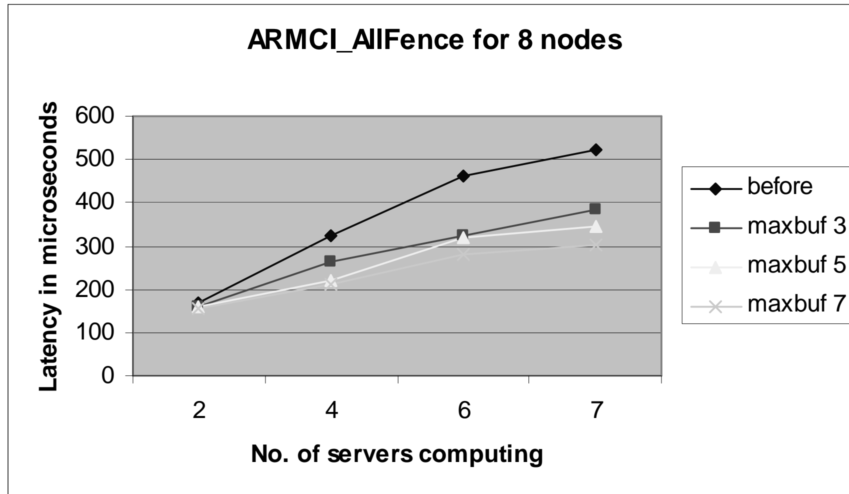


Figure 3.5: Latency of ARMCI_AllFence operation for 8 nodes

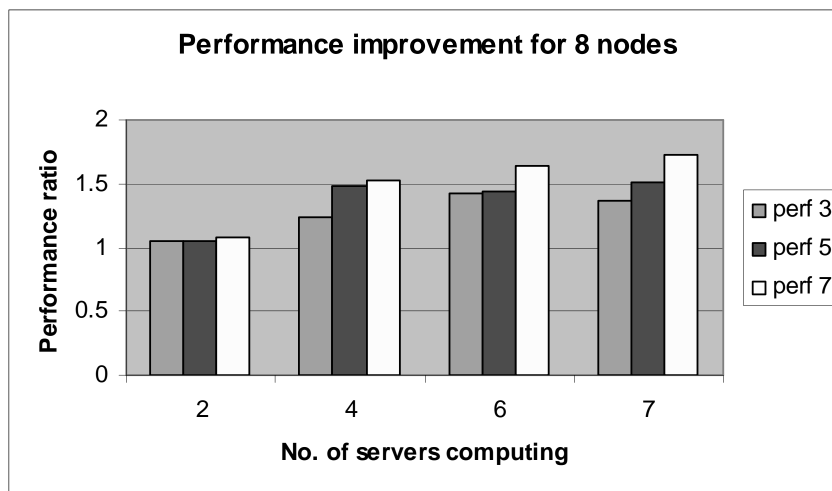


Figure 3.6: Performance improvement ratio of ARMCI_AllFence operation for 8 nodes

case each client issues a number of *put* operations on seven remote servers. For the first case, we performed the test by taking K = 3, for the second case, we performed the test by taking K = 3,5, for the third and the fourth case we performed the test by taking K = 3,5,7. Again, here we observed that by providing more number of buffers the performance improves even more. For each case we took the average of latencies of all the client nodes participating in the fence operation. The maximum factor of improvement in the performance achieved for 8 nodes is 1.72 over the current implementation .

CHAPTER 4

IMPLEMENTATION OF BARRIER OPERATION IN ARMCI

Parallel programs are commonly written using barriers to synchronize parallel processes. Upon reaching a barrier, a process must stall until all the participating processes reach the barrier. A fast implementation of barrier is important because it allows fine grained parallel programs to be more efficient. It is therefore important to minimize the latency of the barrier operation.

4.1 Achieving Barrier Operation in ARMCI

The ARMCI library currently does not have a barrier primitive defined in its implementation. The main function of the fence operation is to block the client until it gets an acknowledgment from each server. In the ARMCI_AllFence operation a client thread on the client node sends a request for acknowledgment to each of the server in the list. Each server thread on the server node then sends an acknowledgment to the client node once it finishes all the computations issued by that particular client node. Currently a barrier can be achieved using ARMCI_AllFence followed by an MPI_Barrier. The number of messages required in this scheme is $2 \times (N-1)$ for the ARMCI_AlFence and $\log_2(N)$ messages for the MPI_Barrier. Hence, the complexity

in terms of number of messages is $2 \times (N-1) + \log_2(N)$, where N is the number of nodes participating in the barrier operation.

4.2 Our Implementation

In our implementation of the barrier operation in ARMCI library, our objective is to achieve a fence as well as a barrier operation. In the new implementation, we keep track of the number of put operations issued by the client thread and the number of put requests completed by the server thread. The server thread increments the counter in the shared memory between the client and the server thread on the same node after the completion of each put operation issued to it. When the client thread reaches the barrier point, it calls a pairwise exchange operation to exchange the counter on each client node to calculate the total number of operations issued on its server. When the number of put operations issued by all the client nodes on a server equal the number of operations completed by that server, the client on that node calls a pairwise exchange one more time to inform the other nodes that it has reached it's barrier point.

4.3 Algorithm and the Pseudo-code

In this algorithm we aim to combine fence and barrier operations. The data structures required for this scheme are:

1. An array on the client side `op_init[N]`.
2. A counter `op_done` in the shared memory between a client and a server on a single node.

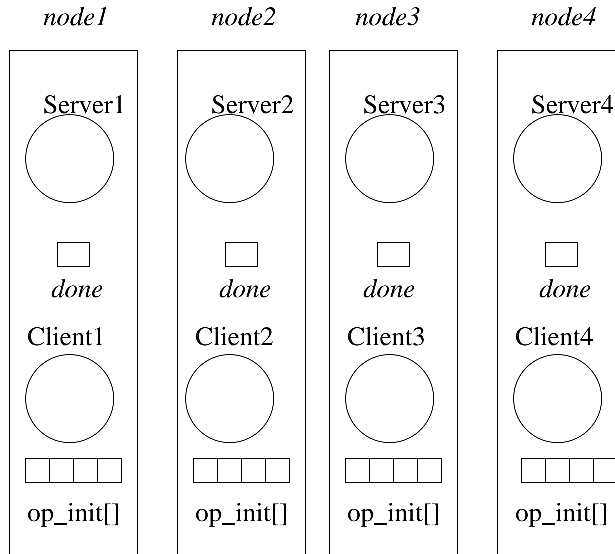


Figure 4.1: Block Diagram for new implementation of the Barrier operation

This is shown in figure 4.1. Size of the `op_init[N]` array is equal to the number of nodes. The index for the array on the client side is the server id `Si`.

When a client initiates a *put* operation on the server of a remote node, it increments the count of the entry in the array `op_init[]` for that server. Similarly, when a server completes a *put* operation issued by a particular client, it increments the count of the counter `op_done` in the shared memory.

When a node reaches the barrier operation, it calls the pair wise exchange algorithm to get the number of *put* operations issued on a particular server. In the pair wise exchange algorithm, a client gets the copy of the array `op_init[]` from other clients. Then we add up the values of the array for the particular server to determine the number of *put* operations issued by all the clients on that server.

Once we get the number of *put* operations issued on a server by all the clients, we wait until the count of the number of *put* operations issued on the server is equal

```
/* N is the number of nodes. */
/* Variables declared in the shared memory of the server and client on one node. */
int op_done;

{/*Client Side*/
int op_init[N];
while (!barrier()){                 /* Until node reaches the barrier point, the client
                                    will keep initiating the put operation and value of Si
                                    will keep changing depending on computation.
                                    Function barrier() will return a true value when the
                                    client will reach the barrier. */
armci_Put(Si);                      /* Operation initiated by client on server Si */
if(Si!=Ci) op_init[Si]++;           /* This will be a part of ARMCI_Put(). */

}

pairwise_exchange();                /* Here Client will call for the pair wise exchange
                                    algorithm to get the array op_init() from the other
                                    nodes. */

add the values in all the arrays (received after pair wise exchange) for all the servers.

while (armci_done!=armci_init[Si]); /* Wait till server completes all the put operations
                                    issued on it. */

pairwise_exchange();                /* Here Client will call for the pair wise exchange
                                    algorithm to inform other nodes that it's server has
                                    finished all the put operations issued on it. */
initialize();                       /* Reinitialize the data structure on the client side as
                                    well as in the shared memory. */
}/*END*/

{/* Server Side */

finished_Put;                       /* Server completes the put
                                    operation. */

if(Si!=Ci)op_done++;

}/*END*/
```

Figure 4.2: Pseudo-code for the Barrier function

to the number of *put* operations completed by the server. When the two counts are equal, client calls the pair wise exchange algorithm to check whether all the other nodes have reached the barrier. All the nodes participate in the pair wise exchange and when all the servers have completed the put operations, all the participating nodes come out of the barrier operation.

The pseudo-code for the algorithm is shown in figure 4.2. Since the new algorithm requires two pairwise exchanges to achieve the barrier operation and the number of messages required for one pairwise exchange is $\log_2(N)$, the complexity of this

algorithm in terms of number of messages required to complete the entire barrier operation is $2 \times \log_2(N)$, where N is the number of nodes.

4.4 Performance Evaluation

The tests were performed on an 8-node cluster of quad Pentium III 700MHz machines (Dell 6400's), which have LANai 7.2 cards with 66MHz NIC processors, running Linux with kernel version 2.2.17. The machines are connected by a Myrinet LAN network with LANai 7.2 cards via a 16-port switch.

A number of different sets of performance readings are taken for the new implementation. One set of readings is for the worst-case scenario in which we issue a number of put operations on remote nodes and immediately call the barrier function. Another set of experiment entails providing different delays between the issuing of put operations and calling the barrier function. We have performed the same set of tests for 2 different message sizes in the put operation, one for a message size of 100 bytes and one for 4000 bytes.

We have obtained the following results for these tests.

The graphs in figure 4.3 show the latency of ARMCI_Barrier for the worst case scenario, in which there is no delay between issuing the put operations and calling the barrier operation for two message sizes: 100 bytes and 4000 bytes. The term 'current' means the current technique for achieving the barrier and 'new' means the new barrier operation. The number following the current/new gives the message size. It is to be noted that we are achieving a factor of improvement of 3.58 over the current implementation of the barrier operation and it scales well as the number of nodes increases.

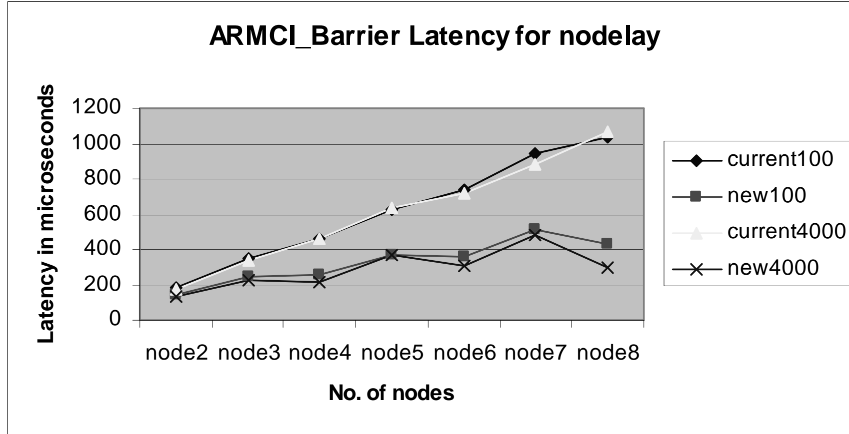


Figure 4.3: Latency of ARMCI_Barrier with no delay

The graphs in figure 4.4 show the latency of the new barrier function for the message size of 100 bytes. We have performed the latency test for different delays of 10us, 50us and 100us. Here again, current and new mean the same as in previous graph, but the number followed by current/new indicates the delay between issuing the put operation and calling the barrier operation.

The graph in figure 4.5 shows the latency of the new barrier function for message size 4000 bytes. We have performed the latency test for different delays ranging from 50us, 100us and 200us.

We wanted to see the impact of delay between the issuance of put operations and the calling of barrier function, to see the robustness of the new algorithm. We also wanted to see the effect of different size of messages in put operation. Hence, we performed the above tests by inserting some delay between the issuance of put operations and calling the barrier function for two message sizes 100 bytes and 4000 bytes. The graphs in figure 4.4 and figure 4.5 show that delay between the computations

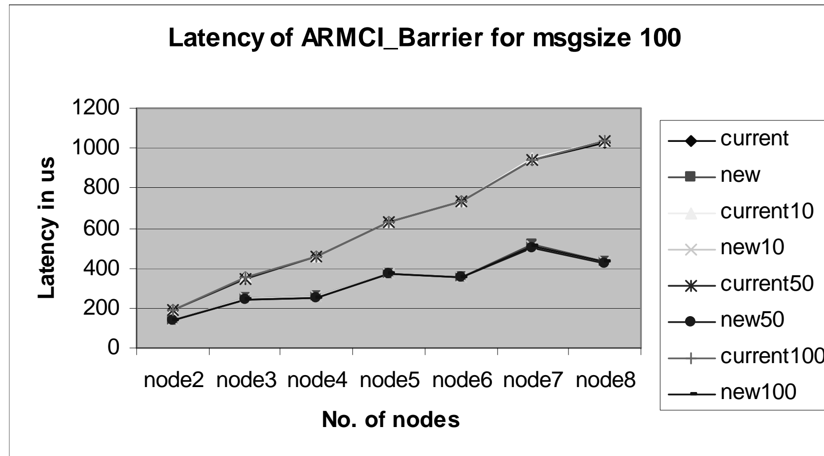


Figure 4.4: Latency of ARMCI_Barrier with different delays for message size 100 bytes

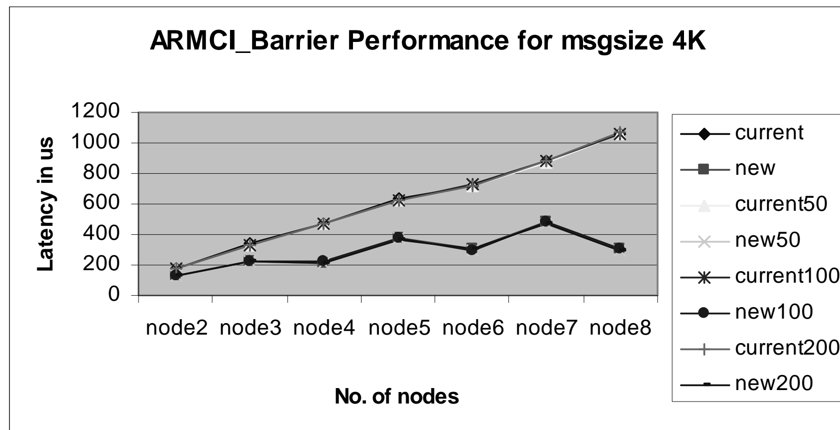


Figure 4.5: Latency of ARMCI_Barrier with different delays for message size 4000 bytes

and calling the barrier function for both the message sizes does not alter the behavior of the barrier operation.

CHAPTER 5

OPTIMIZATION OF LOCK OPERATION IN ARMCI

Efficient implementation of synchronization operations such as locks and semaphores is important in parallel and distributed systems. Mutual exclusion operations are implemented using a wide range of algorithms. The simple algorithms tend to be fast when there is no contention for the lock but inefficient under high contention, whereas sophisticated algorithms are the ones that deal well with contention, but have a higher cost in the low contention case. Some performance goals [11] for implementing locks are *low latency*, *low traffic*, *scalability*, *low storage cost* and *fairness*. In our new implementation of locks in ARMCI, we aim to achieve most of the above mentioned goals.

5.1 Current Implementation

Locks are important for implementing mutual exclusion in remote memory operations. In ARMCI, a user can allocate a set of mutex variables on each process and then use lock and unlock operation to acquire and release a lock. ARMCI has a hybrid implementation for lock mechanism: the *server based queue algorithm* [12] for the remote lock operation and the *ticket based algorithm* [11] for the local lock operation.

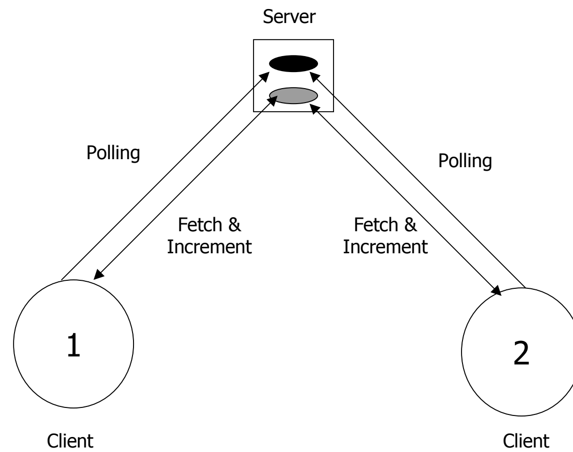
5.1.1 Server Based Queue Lock

In the *server based queue*, the server maintains a queue of processes that request a particular lock on each node. In this scheme, a lock operation involves sending a request to the server. This request is simply a control message that identifies the mutex and the process on which the mutex resides. The server inspects the queue for the specific mutex and if it is free, it responds to the client with the token for that mutex. If the lock is not available, the server adds the process to the queue for that lock and leaves the client blocked waiting for a response. A client releases the mutex by sending a request to the server that includes the token for the mutex. The server inspects the queue of waiting clients and if the queue is not empty, it sends a message to the next process in the queue.

5.1.2 Ticket Algorithm Based Lock

The *ticket based algorithm* for locks works just like the teller line at a bank. Every process wanting to acquire the lock takes a ticket number and then busy-waits on a global variable until the global variable equals the ticket number obtained. To release the lock, a process simply increments the global variable so that the next waiting process can acquire the lock. The atomic primitive needed is *fetch and increment*, which a process uses when it first reaches the lock operation to obtain its ticket number from a shared counter. No atomic operation is needed to actually obtain the lock upon a release since only the unique process that has its ticket number equal to global variable attempts to enter the critical section when it sees a release. Figure 5.1 and 5.2 show the acquire and release of a lock in ticket lock, respectively.

Ticket Based Lock (Acquire)



Ticket Based Lock (Release)

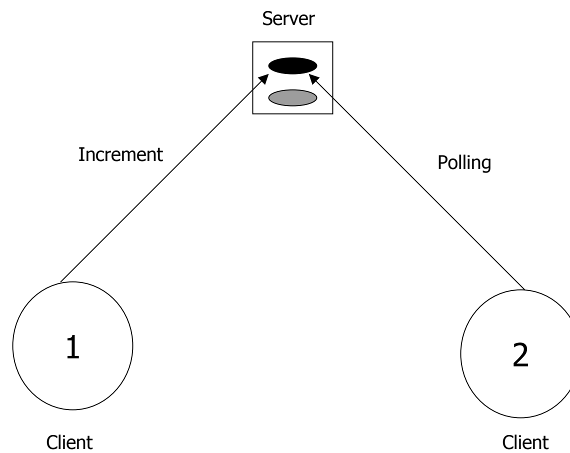


Figure 5.2: Example of a ticket unlock

5.1.3 ARMCI Implementation

In ARMCI, when a client wants to acquire a lock, it first checks whether the lock is residing on its server or a remote server. If it's on the local server, it gets a ticket number from the server. The *fetch and increment* operation is achieved by calling ARMCI *read-modify-write* operation. The client then sends a request to the server, which upon receiving the request sends a ticket number to the client and increments the counter of ticket number. The server which owns the lock has a shared counter. A client waiting for acquiring a lock constantly polls, locally, on the shared counter using the ARMCI *get* operation. When the shared counter equals the ticket number the client has, the client is said to acquire the lock.

If the server which owns the lock is on a remote node then the client requests for a ticket number from the remote server. The remote server upon receiving the request checks whether the ticket number generated for the requesting node is equal to the shared counter. If it is equal, then the server immediately sends the ticket number to the remote client and client acquires the lock. If the shared counter is not equal to the ticket number generated for the requesting client, then, the server puts the client node id in a blocking queue. When a client releases the lock, it sends its token back to the server.The server when receives the token, grants the lock to the next process in the blocking queue. Hence, in case of remote lock, the client does not have to poll constantly on a remote counter using ARMCI *put* operation and instead just gets blocked.

5.2 Our implementation

5.2.1 Trade-offs of Current Implementation

The *ticket lock* generates much less traffic than other algorithms like the *test&set* [11] lock. The *ticket lock* also requires constant and small storage and is fair since the client processes obtain the lock in the order of their *fetch and increment* operation.

The *ticket lock* has a read traffic problem for local locks in a SMP node at the time of lock release. The reason for this is that all the local processes spin-wait on the same shared variable. When that variable is written at release, all processors' cached copies are invalidated, and they incur read misses. The read misses cause unnecessary traffic. There are other algorithms which can reduce the traffic at the time release, but they increase the space requirements, e.g. array-based locks.

In case of remote lock operation, if there are many clients requesting the lock, the remote server is flooded with requests. In case of remote lock operation, the server maintains the queue of the blocked processes. Thus, receiving so many requests simultaneously and the granting of locks would slow it down.

5.2.2 Software Queuing

In our implementation of local lock, we plan to reduce the traffic while not increasing the space requirement and for the remote lock we plan to reduce the workload of server by implementing a distributed queue. We use a *software queuing*[12] lock in our new implementation. A *software queuing* lock both reduces the traffic at the time of lock release and ensures all spinning will be on locally allocated variables.

The basic idea behind the *software queuing* lock is to have a distributed linked list or a queue of the waiters on that particular lock. The head node in the link list

represents the client that holds the lock. Every other node is a client that is waiting on the lock and is allocated in that client's local memory. There is also a tail pointer that points to the last node in the queue, that is, the last node that has tried to acquire the lock.

In this algorithm, each client node has a *next* variable which points to the client which has requested the lock immediately after the current client node, and a boolean variable *locked*, which indicates whether the client is waiting for the lock or not. These two variables are in the shared memory so that remote memory operations can be performed on them. *Lock* a variable which points to the last client in the queue and is created on the node which owns the lock.

When a client requests a lock, it first sets its *next* variable to `NULL`. Next, it performs a *fetch & store* operation on the *lock* variable to determine the address of its predecessor client node that had requested lock before it. If the queue is empty, then this client will acquire the lock. If the queue is not empty, then the requesting client will set its *locked* variable to `true` and performs an ARMCI *put* operation to write its own address to its predecessor client's *next* variable, thereby inserting itself in the queue. It then polls on its local variable *locked* until it becomes false.

To release a lock, the client node which is holding the lock now (current node), checks if its *next* variable is `NULL`. If its *next* variable is not `NULL`, it performs an ARMCI *Put* operation on its successor client node's *locked* variable in the queue, by setting it to `false`, thereby successfully releasing the lock. Otherwise, if it is `NULL` then it performs an ARMCI *compare & swap* operation to ensure that really no one is in the queue waiting to be released and the *lock* variable points to itself. On a successful *compare & swap* operation, *lock* is set to `NULL` and then lock is released.

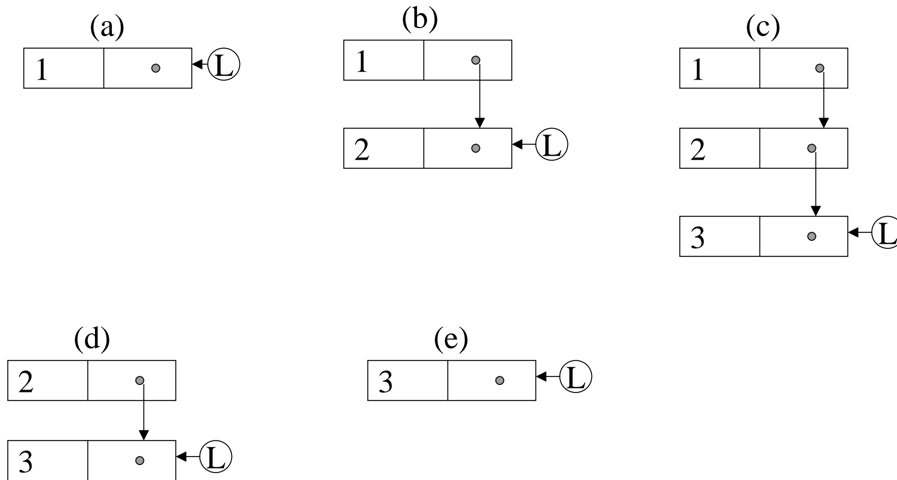


Figure 5.3: Example of a distributed lock

If the *compare & swap* operation has failed, it means that someone just got in the queue and has modified the *lock* variable, but has not modified the *next* variable of the current client node. So the current client node will poll on its *next* variable. Once the new requesting client node sets the *next* variable of the current client node, it sets the *locked* variable of the new requesting client node to `true` by performing an ARMCI *Put* operation.

We will get a better picture of how the software queuing works by looking at the pictorial representation of the lock. Assume that lock in Figure 5.3 is initially free. When process 1 tries to acquire the lock, it acquires it and the queue looks as shown in Figure 5.1(a). In step (b), process 2 tries to acquire the lock, so it is put on the queue and the tail pointer of the queue now points to it. Process 3 is treated similarly when it tries to acquire the lock in step (c). Process 2 and process 3 are spinning on local flags associated with their queue nodes while process 1 holds the lock. In step (d), process 1 releases the lock. It then "wakes up" the next process, 2, in the queue,

by writing the flag associated with process 2 and leaves the queue. Process 2 now holds the lock and it is at the head of the queue. The tail pointer does not change at all in any of these steps. In step (e), process 2 releases the lock similarly, passing it to process 3. There are no other waiting processes, so process 3 is both the head and tail of the queue. If process 3 releases the lock before another process tries to acquire it, then the lock pointer will be `NULL` and the lock will be free again. In this way processes are granted the lock in FIFO order with regard to the order in which they tried to acquire it.

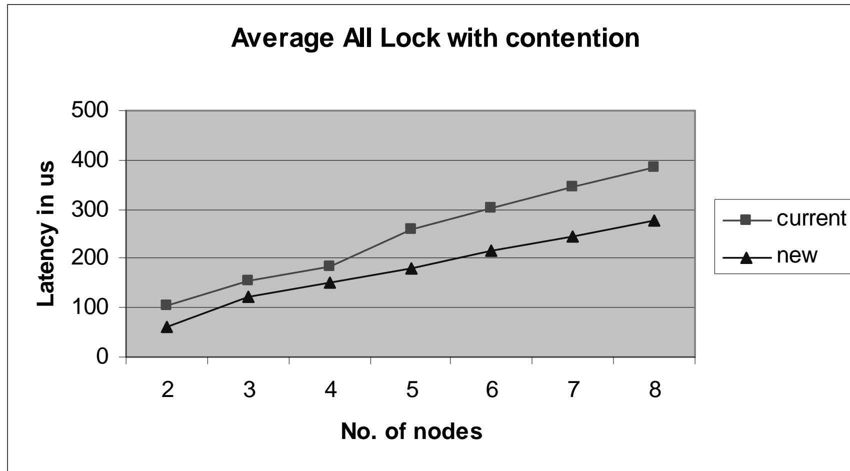
5.2.3 Design Challenges

Since the software queuing algorithm for the lock is implemented for the distributed memory, we cannot simply use memory pointers for the *next* and *lock* variables as used in the original algorithm. A remote memory location is specified as a pair of the node id and memory address.

ARMCI supports *swap* operation for integers and long type variables and the *compare & swap* operation does not exist currently in ARMCI library. Since our implementation requires RMW operation for data structures, we added two functions to the ARMCI API, a *swap* operation and a *compare & swap* for data structures.

5.3 Performance Evaluation

In this section, we evaluate the performance benefits of our implementation. The performance results were obtained by running experiments on a cluster consisting of eight quad 700MHz Pentium III machines each with 1GB of RAM, running Linux kernel version 2.2.18. These machines are connected by another Myrinet LAN network using NICs with 66Hz LANai 9 processors. These are connected to an 16-port switch.



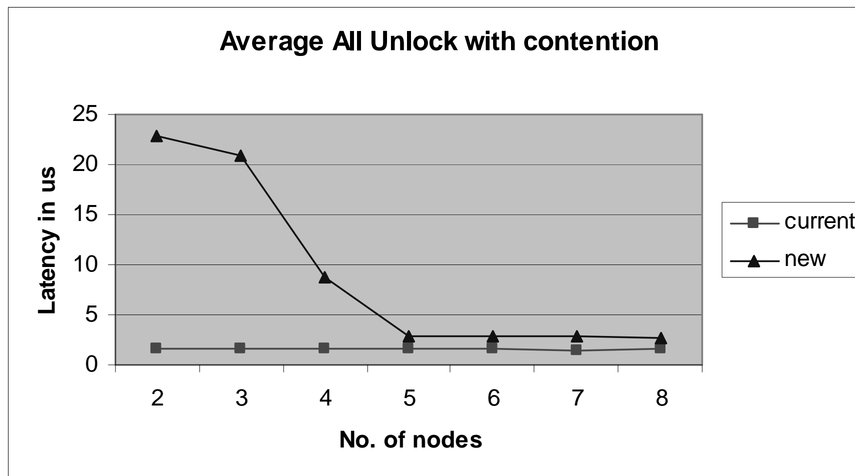


Figure 5.5: Latency of ARMCI_Unlock

In the lock test with contention, we took the average time it takes for all the processes to repeatedly acquire and release a remote lock. To plot the graph we took average of all the averages. To vary the load, we added more nodes which repeatedly locking and unlocking the same lock. The figure 5.4 shows the latency of ARMCI_Lock and figure 5.5 shows the latency of ARMCI_Unlock with contention.

We can see from the graph that the new implementation of ARMCI_Lock performs better than the current implementation of ARMCI_Lock. It scales well with the increase in the number of nodes. We achieve a factor of improvement of 1.7 over the current implementation. In the graph for ARMCI_Unlock, we see that the new implementation performs worse than the current implementation, but as number of nodes increases, it performs on par with the current implementation. The reason for the worse performance of the new implementation is that for very less number of nodes contending for lock the last node releasing performs a read-modify-write operation, which is quite expensive. As the number of nodes increases this time averages out.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis we proposed to optimize the performance of the collective communication of one-sided communication library ARMCI. We know that ARMCI has client-server architecture and the server handles all the data requesting messages. Hence, if the server is flooded with such requests it will degrade the performance of the application using ARMCI. Increase in such requests means that more buffers are required to store all the requests at the same time, which increases the consumption of resources. The increase in contention on server also increases the wait time of the client requesting data from the server. In this thesis we analyzed the factors which increase the contention on the server and tried to eliminate those factors with new and better implementations.

First we analyzed and optimized the *fence* operation in ARMCI, which is the only means of synchronizing all the processes at one point, by reducing the contention. By doing so, we also tried to minimize the number of buffers required for the fence operation. The new implementation results in a significant reduction of the wait time of a node to come out of the fence operation. The factor of improvement in the performance of the fence operation using our implementation has been observed to be up to 1.72 for an 8-node system. Then we implemented a new *barrier* operation to

achieve synchronization in a faster manner. The new implementation of the barrier operation requires less number of messages to be exchanged between the clients and the servers, which eventually reduces the contention on the server. The new barrier function gives a factor of improvement of 3.58 over the previous method of performing the barrier operation on an 8-node system. We also explored the *locks* in ARMCI. We found that the locks can be implemented in a better manner as compared to the current implementation. We implemented a *software queue* based scheme, which reduces the contention on the server. We achieve a factor of improvementof 1.7 over the current implementation of locks in ARMCI.

In future, we would like to do some real life application based evaluation for our new implementations. Though, all the graphs in the previous chapters show that the new implementations are scalable, we would still like to analyze in future the scalability of our implementations on a large cluster.

We would like to implement a non-blocking algorithm for the barrier operation. Also, we would like to explore the one-sided data transfer primitives like get/put. These primitives have already been pipelined by a former student of NOWlab, Vinod Tipparaju, but there is still scope of improvement in the performance of these primitives by making them non-blocking.

BIBLIOGRAPHY

[1] T. von Eicken, A. Basu, V. Buch, W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In Proceedings of the 15th ACM Symposium on Operating System Principles, December 1995.

[2] T. Anderson, D. Culler, D. Patterson. A Case for Networks of Workstations (NOW). IEEE Micro:pages 54-56, Feb 1995.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kualwik, C. L. Seitz, J. N. Seizovic, Wen-King Su. Myrinet-a gigabit-per-second local-area network. IEEE Micro, 15(1):29-36, February 1995.

[4] W. Gropp, E. Lusk, N. Doss, A. Skjellum. A High Performance, Portable Implementation of the MPI message Passing Interface Standard. In Parallel Computing, 22(6), 1999, pages 789-828.

[5] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. IEEE Parallel and Distributed Technology, 4(2):63–79, Summer 1996.

[6] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99, San Juan, Puerto Rico, April 1999,

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, March 1994.

[8] The GM Message Passing System. Documentation available at http://www.myri.com/scs/index.html.

[9] Nieplocha, J. Ju, E. Apra, One-sided Communication on Myrinet-based SMP Clusters using the GM Message-Passing Library, in Proc. CAC01 Workshop of IPDPS'01, San Francisco, 2001.

[10] J. Nieplocha, RJ Harrison, and RJ Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. The Journal of Supercomputing, 10:197-220, 1996.

[11] Singh, Culler. Parallel Computer Architecture: A hardware/software approach. Stanford.

[12] Jarek Nielplocha, V. Tipparaju, A. Saify, and D. K. Panda, Protocols and Strategies for Optimizing Remote Memory Operations on Clusters, Proc. of Communication Architecture for Clusters (CAC '02), April 2002.

[13] Raoul A.F. Bhoedjang, Tim Rühl, Henri E. Bal, User-Level Network Interface Protocols, IEEE Computer, Nov. 1998, pp. 53-60.