-

OPTIMIZING ARMCI GET AND PUT OPERATIONS ON MYRINET/GM

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

Vinod Tipparaju, B.E.

* * * * *

The Ohio State University

2001

Master's Examination Committee:

Prof.Dhabaleswar K. Panda, Adviser

Prof.Ponnuswamy Sadayappan

Approved by

Adviser

Department of Computer
and Information Science

Abstract

Advances in processor, network and protocol technologies have made cluster of workstations an attractive platform for high performance computing. Emerging applications in cluster environments require very frequent data transfer between process memories. The main focus thus has been on utilizing the advances in protocol and network technologies and the low latency communication layers to develop efficient data transfer between process memories in clusters. One-sided communication has gained a lot of attention to support efficient data transfer capabilities for irregular applications. Thus there is a need to design these one-sided communication operations in such a way that they utilize the underlying network to the maximum extent and deliver maximum possible throughput.

In this thesis we propose ways to optimize performance of "get" and "put" One-Sided operations in ARMCI one-sided communication protocol by developing a pipelining architecture for these operations and methods for tuning the parameters that affect the pipeline performance. This design relies mostly on efficient buffer management and methods to tune the parameters that affect the performance of one sided operations. Our implementation optimizes ARMCI library on the Myrinet/GM user level communication system by applying the proposed pipelining design to it. Our implementations achieve up to 24% improvement in the bandwidth for the Put operation and a 8.9% improvement in the bandwidth for the Get operation.

Dedicated to my parents, sister, Rao and Vasu Unnava.

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 for all his invaluable suggestions to help me finish this thesis.

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

I am thankful to Dr.Jarek 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 Mohammad Banikazemi for their willingness to help at all times. Special thanks to Rinku Gupta and Amina Saify for all their help in finishing this thesis.

This thesis would not have been possible without the financial support I received from various sources. I am indebted to Prof.Sandy Mamrak for awarding me research associate position in the first year of my M.S., OSU CIS department for awarding me teaching assistantship, and to Prof.Panda for supporting me as a research associate in the following year.

Finally, I would like to thank all my friends and relatives who made my stay at OSU enjoyable, especially Rao Unnava, Vasu Unnava, Partha and Vaasavi.

VITA

September 1, 1976	Born - Hyderabadbad, India.
1999	B.E., Computer Engineering, Nagarjuna University, Guntur, India.
Spring 2001 - Summer 2001	Graduate Research Associate, The Ohio State University.

FIELDS OF STUDY

Major Field: Computer and Information Science

Contents

List of Figures

Chapter 1

INTRODUCTION

Scalable systems research is motivated by the belief that two processors are better than one. This belief forms a "divide and conquer" approach to computation. Parallelism thus becomes the basis for solving ever harder and bigger problems. In the last few years, microprocessors have gained a lot of demand and have been used successfully in PCs and the PC market in turn has been fueling the improvements in the microprocessor performance. Building blocks of PCs have become very fast, inexpensive and readily available. This has lead to development of PC clusters. Performance of inter-processor communication plays a major 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 underlying network and smartness in the Network Interface Cards (NICs) 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 Smart NICs and User-Level Protocols

In recent years there has been a lot of research in areas related to utilizing smart NICs and processors on the smart NICs in protocol processing. Modern interconnects like Myrinet[6] and Gigabit Ethernet[12] offer Gb/s speeds which has put the burden of reducing the communication latency on messaging software. This has led to the development of OS bypass protocols or user level protocols [11]. These protocols remove the kernel from the critical path and hence achieve reduced end-to-end latency. Programmable NICs have made it possible to move some of the protocol processing from user space to the NIC leaving the host processor to dedicate more cycles to the application. Communication latency on any cluster is primarily composed of two components[5]: time spent in processing the message and the network latency. Modern high speed interconnects such as Myrinet have reduced the communication latency by significantly reducing the network latency. In older systems, the processing of the messages by the kernel caused multiple copies and many context switches which increased the overall end-to-end latency. User level network protocols address this issue by making sure that the parts of the protocol or the entire protocol moved to the user space from the kernel space.One of the first examples in this case is U-Net[10].

In the traditional systems, the NIC would simply take the data from the host and put it on the interconnect. However, modern NICs have programmable processors and memory which makes them capable of sharing some of the message processing work with the host. Thus, the host can give more of its cycles to the application, enhancing application speed up. Under these developments, modern messaging systems are implemented outside the kernel and try to make use of available NIC processing power.

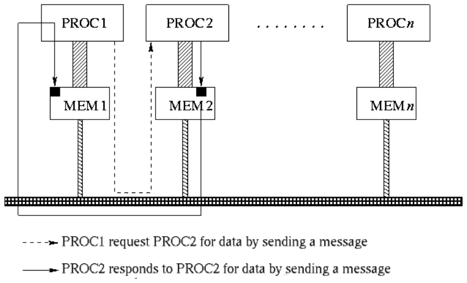

PROC1
PROC2
....
PROC*n*
MEM1
MEM2
MEM*n*
PROC1 request PROC2 for data by sending a message
PROC2 responds to PROC2 for data by sending a message

Figure 1.1: Message Passing Programming Model

1.2 Programming models for clusters

Generally three paradigms exist to facilitate programming in cluster environments: shared memory, message passing and get/put one-sided communication. These paradigms have been used for inter-process communication and also for synchronization in multi-process computation.

In the traditional message passing model, each node has its own memory that is directly accessible only by processors in this node. Communication in the message passing model requires cooperation between sender and receiver. The data owner must know when and which process needs the data, and data transfer implies a form of synchronization between the process that owns the data and the process that needs the data. Asynchronous (non blocking) send/receive operations might diffuse the synchronization point, but the cooperation between sender and receiver is still required. Message-passing is popular, not because it is particularly easy, but because it is so general. Figure 1.1 shows the structure of the message passing model. This simple figure also shows how generic this model can be.

Shared memory on the other hand has been an appealing model from the point of view of programming ease. Flexibility of the shared memory model is one of the many reasons to why researchers have been thinking about architectures implementing the shared memory model on physically distributed machines. Several architectures have been proposed for abstraction of shared memory in a physically non-shared architecture like a cluster of workstations. This abstraction is referred to as the Distributed Shared Memory(DSM) model. In this system, a set of nodes connected via an interconnect network do not physically share memory. DSM mechanisms allow an application to access shared data which is not physically resident on that node. These mechanisms are usually provided as a software layer either integrated with or on top of an operating system. Figure 1.2 shows how a DSM system looks like. DSM systems rely on replicating shared data items and allowing concurrent access at many nodes. However, if the concurrent accesses are not carefully controlled, memory accesses may be executed in an order different from that which the programmer expected. Informally, a memory is coherent if the value returned by a read operation is always the value that the programmer expected. For example, it is quite natural for a programmer to expect a read operation to return a value stored by the most recent write operation. Thus, to maintain the coherence of shared data items, a mechanism that controls or synchronizes the accesses is necessary. The coherency protocols that DSM systems typically use are: Write Invalidate and Write Update [13]

The get/put one-sided communication model uses get and put operations in implementing a one-sided communication. Get is a remote memory read operation which gets data from remote memory without the intervention of the computing process on the remote machine. Put operation is like a write operation which writes into a

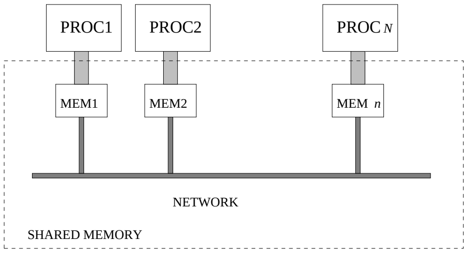


Figure 1.2: Distributed Shared Memory Programming Model

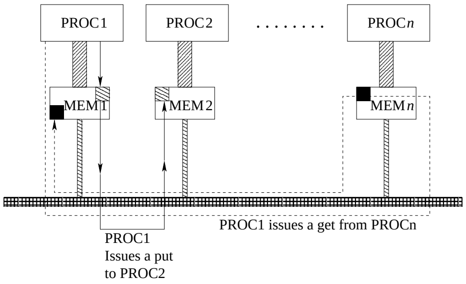


Figure 1.3: Get and Put one-sided programming model

remote memory without the intervention of the computing process on that machine. Thus get and put operations are one-sided. They do not require cooperation from the receiver node. Its one-sidedness certainly has a lot of advantage in many application categories, specially applications related to computational chemistry where need to make unpredictable reference to remote data [3]. Figure 1.3 shows the working of a get/put one-sided communication model.

This approach certainly sounds superior to using two-sided communications, where it is necessary to coordinate each data transfer on both sides. And it is definitely useful at times. In general, it is not as easy to use as it would seem since it often requires that the application programmer manually perform hand-shaking that is done automatically with two-sided communications.

The MPI[1] standard has defined a set of one-sided calls in the newest 2.0 release. At this time, not all of the MPI implementations support these calls, so MPI programs that use them may not be portable for a while. The SHMEM library developed by Cray is a one-sided library for the CrayT3E and SGI Origin systems [15]. It has become so popular that it is growing into a standard and is being implemented on a variety of platforms. The GPSHMEM library is a general purpose SHMEM library that provides the same one-sided interface but is implemented on top of lower level libraries.

One-sided communication assumes that a process can access data on a remote node

- asynchronously,
- without explicit cooperation of the process on the remote node, and
- with latency and overhead costs that are comparable to standard send and receive operations.

Most of the times there is a need for one-sided communication to coexist with conventional cooperative message passing. Many applications need to use both strategies, typically in different phases of the application. One-sided communication is useful whenever parallel programs need to make unpredictable references to remote data. It is particularly required for applications that also use dynamic load balancing and have wide variation in task size.

In comparison, traditional message passing is designed in such a way that the data owner must know when and which process needs the data, and data transfer implies a form of synchronization between the process that owns the data and the process that needs the data. Implementation with asynchronous (non blocking) send/receive operations might diffuse the synchronization point in a message passing system, but the cooperation between sender and receiver is still required.

1.3 Problem Statement and approach

One-sided communication protocols, designed to run on one or more underlying user-level communication protocols, do not sufficiently utilize or configure the communication.This motivates the need to develop a set of guidelines as to how one-sided operations should be designed and what are the so called 'parameters' that require tuning for improving the performance of a one-sided operation. These guidelines in particular should

- Manage flow, frequency, and number of buffer to use and other aspects of data transfer so as to utilize the network to the maximum possible extent.
- Pipeline the flow of buffers so as to completely utilize the transmission latency for other sub-operations
- Define a relationship between the size of buffer used inside the implementation of a one-sided operation and the number of such buffers.

In this thesis we propose and implement these guidelines for ARMCI[3]- A portable aggregate remote memory copy interface. We focus on ARMCI's Get and Put operations for contiguous and non-contiguous data transfers and optimize them for

Myrinet/GM[8]communication layer. We attempt to address the above mentioned issues as follows

- We describe the ways to manage flow, frequency and number of buffers to use for ARMCI get and ARMCI put operations.

- we describe how the pipelined Put implementation utilizes the buffer transmission latency and in also how we overlap and pipeline the stages in an ARMCI get operation

- we describe the behavior of the ARMCI put operation for different number of transmission buffers and discuss in detail the effects of using different buffer sizes for transmission.

1.4 Thesis Organization

In chapter 2, we provide a brief discussion about user-level communication protocols and ARMCI, is detailed description of the get and put operations and general communication paradigm in ARMCI. Chapter 3 talks about pipelining the Put operation in ARMCI. Pipelining of the put operation reveals a few generic parameters that require to be tuned for efficient implementation of any one-sided operation. In chapter 4 we use these parameters in describing the get pipeline and show how it can been tuned. We also discuss an implementation of the new pipelined get operation. Finally in chapter 5 we discuss the effect of using buffers of different sizes in pipelining the one-sided operations in ARMCI. Conclusions and future work are presented in chapter 6.

Chapter 2

BACKGROUND AND RELATED WORK

In this chapter we discuss about the background material and related work. After a discussion about the user level communication protocols in general, we discuss about one-sided communication protocols and their implementation on user level communication protocols. We also briefly discuss about ARMCI as a portable one-sided communication library and give a description of the two important one-sided operations supported by ARMCI.

2.1 User level communication protocols and Myrinet/GM

Since clusters have become a popular platform for high performance computing, it has become necessary to make the cluster communication systems very efficient. In recent years, communication systems with user level protocols have been proposed to address the efficiency issues in cluster communication. These communication systems use much simpler protocols in comparison to legacy protocols and the role of the operating system in these protocols has been significantly reduced. That way, applications have direct access to the network while bypassing the operating system.

The Myrinet/GM communication system is a very popular communication system in cluster environments and Myrinet is the most popular high-speed interconnect for commodity clusters [6]. In this thesis we focus on the Myrinet network and the GM communication system as our underlying user-level communication protocol.

Myrinet network is constructed out of switching elements and host interfaces. The core of the switching ship is a pipelined crossbar that supports non-blocking cut-through routing of packets. The building block of a Myrinet network is a 16-port switching chip. It can be used to build a 16-port switch, or can be interconnected to build various topologies of varying sizes. The routing of Myrinet packets is based on the source-routing approach. Each Myrinet packet has a variable length header with complete routing information. When a packet enters a switch, the leading byte of the header determines the outgoing port before being stripped off the packet header. At the host interface, a control program called MCP (Myrinet Control Program) is executed to perform source-route translation. Myrinet provides reliable, connectionless message delivery between communication end points, called ports. This is achieved by maintaining reliable connections between each pair of hosts in the network and multiplexing the traffic between end points over these reliable paths.

Myrinet's GM communication system is a connectionless model because there is no need for a user process to establish a connection with a remote host. Once the mapping of destination addresses 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.

In general, for most of the user level communication protocols OS has an initial involvement for memory registration, setting up connection channels and later for memory deregistration of DMAable memory. These user level communication protocols usually allow data transfers to and between the registered, DMAable memory. Different user level communication protocols have different terminology for representing these channels: Ports in GM, Virtual interfaces in VIA and contexts in FM. These channels provide a way for user applications to be able to send/receive data through the NIC. Communication to NIC is usually done by posting "descriptors" describing the operation the user application is expecting the NIC to perform.

NIC firmware has its own ways of obtaining the descriptor to act upon it. In GM it is done by polling a descriptor queue, which has been memory mapped onto the the NIC address space and in VIA the descriptors are obtained through a doorbell mechanism. Since only the data in registered memory can be transmitted, user level protocols can do without worrying about the data consistency or making a copy of the data for transmission. Figure 2.1 shows the NIC DMA engines processing the send and receive descriptors posted by the application. The number of outstanding sends/receives allowed is usually limited by the length of the send/receive descriptor queue.

2.2 Mechanisms for implementing One-sided communication protocols

Implementing a one-sided communication protocol has many issues and hence there are several mechanism for implementing one-sided communication[4]. These mechanisms can be broadly classified as follows

- interrupt-receive
- active-messages

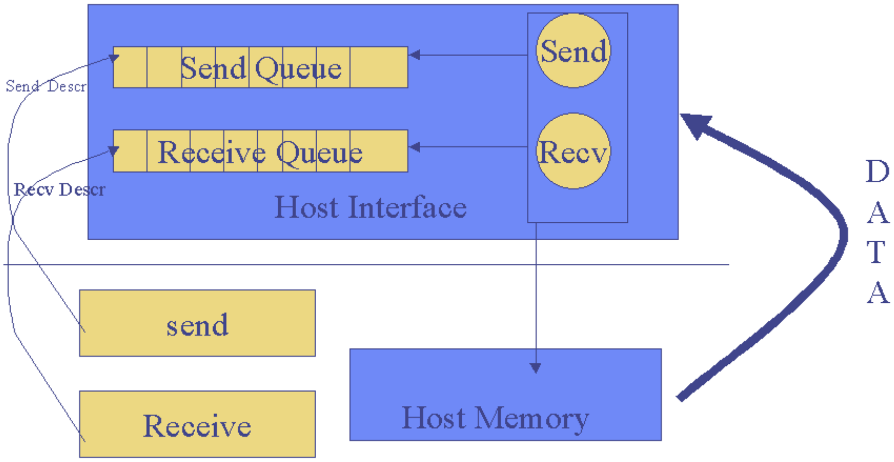


Figure 2.1: The NIC DMA engines and send/recv queues

- data server threads
- globally-addressable or shared memory

Interrupt receive is a form of non-blocking receive operation. It has a message buffer for receiving the data and allows the establishment of a user-specified interrupt handler routine, which is called when a matching message arrives. Interrupt receive lets the programmer treat incoming messages as interrupts.

The idea of Active Messages (AM)[7] is in fact very similar to the interrupt receive. A message includes a control information header that contains an address of an application routine that has to be executed upon its delivery to the destination process, hence the term“Active”. The message initiates an application once it reaches the destination. Under the Active Message model, the sender injects a message to the network and continues computing; the destination process is notified or interrupted when the message arrives and then executes a handler routine. Interrupt receive requires explicit posting and specification of the buffer address and handler routine by the message receiver. In short, interrupt receive pre-specifies the handler routine for an incoming message. Active Message does not require the receiver to specify any such routine as the handler.

One-sided communication can be implemented on networks of workstations and other parallel machines using multiple processes or threads. One process or thread is used for handling communication and one is used for computation. This is called a data server model. ARMCI [3], the one-sided communication library uses this model for implementing one-sided communication operations. The data server model implements one sided communication using a send-receive model. The communication thread continuously receives request-messages from the application threads on other processors/nodes. Efficiency of this implementation is dependent on how the communication thread is executed. The communication thread should block on any incoming messages and should execute with the highest priority. This ensures that processor spends less time on the communication thread and thus incurs a low message latency.

In a globally-addressable memory system, any location in memory can be referenced using the same address format by any processor in the system. To achieve this, the address format might contain information about the process that owns the data. Access to memory might be possible either through library functions or special language constructs. A special case of globally-addressable memory is shared-memory, which doesn't require special API or language constructs to access the shared memory locations.

The MPI-2 also has one-sided communication primitives implemented in it. The one-sided communication specifications include remote memory copy operations such as put and get [1]. Non contiguous data transfers are fully supported through the

MPI derived data types. There are two models of one-sided communication in MPI-2: “active-target”and “passive-target”. The MPI-2 one-sided communication and in particular its “active-target”version have been derived from message-passing, and its semantics include rather restrictive (for a remote memory copy) progress rules closer to MPI-1 than to existing remote memory copy interfaces like Cray SHMEM, IBM LAPI or Fujitsu MPlib. A version of MPI-2 one-sided communication called “passive-target” offers more relaxed progress rules and a simpler to use model than “active-target”.

ARMCI [3] is one such one-sided communication library. ARMCI offers both simpler and lower-level model than the MPI-2 [1] one-sided communication to streamline the implementation and improve its portable performance on cluster environments. It has very straight forward and simple progress rules.

2.3 ARMCI remote memory copy library

The Aggregate Remote Memory Copy Interface (ARMCI) is a new library that offers remote memory copy functionality. It aims to be fully portable and compatible with message-passing libraries such as MPI or PVM. It focuses on the non-contiguous data transfers. ARMCI in particular, unlike other one-sided communication software is meant to be used by the library rather than application developers. Example libraries that ARMCI is targeting include Global Arrays, P++/Overture,and Adlib PCRC run-time system. ARMCI offers both simpler and lower-level model than the MPI-2 one-sided communication to streamline the implementation and improve its portable performance. ARMCI has very straightforward progress rules. It makes it really easy to develop or measure performance of applications built on ARMCI. Its simple progress rules also make it easy to deal with ambiguities of platform specific implementations. Therefore, the ARMCI remote copy operations are truly one-sided and complete regardless of the actions taken by the remote process.

The ARMCI operations are completed in the order they are issued, when referencing the same remote process. Operations issued to different processes can complete in an arbitrary order. This is in fact in synchronization with how ordering should be handled in an one-sided communication protocol. Ordering simplifies programming model and is required in many applications such as computational chemistry. For applications that enforce ordering of the otherwise unordered operations by providing a fence operation that blocks the calling process until the outstanding operation completes so that the next operation issued would not overtake it. This approach accomplishes more than the applications desire, and could have negative impact on the application performance on platforms where the copy operations are otherwise ordered. Usually, ordering can be accomplished with a lower overhead inside the communication library by using platform-specific means rather than at the application level with a fence operation. A compatibility with message-passing libraries (primarily MPI) is necessary for applications that frequently use hybrid programming models. Both blocking and a non-blocking APIs are needed. The non-blocking API can be used by some applications to overlap computations and communications. ARMCI requires a message-passing library for the process start up and user environment initialization.

The library provides three classes of operations:

- data transfer operations including put, get, and accumulate

- synchronization operations including local and global fence and atomic read-modify-write, mutex operations

- utility operations for allocation and de-allocation of memory and error handling.

It offers two formats to describe non-contiguous layouts of data in memory.

1. Generalized I/O vector: It is the most general format intended for multiple sets of equally-sized data segments moved between arbitrary local and remote memory locations. It extends the format used in the UNIX readv/writev operations. It uses two arrays of pointers: one for source and one for destination addresses. The length of each array is equal to the number of segments. Put operations in ARMCI include scatter and gather. The format would also allow to transfer a triangular section of a 2-D array in one operation.

```
typedef struct {
void *src_ptr_ar;
void *dst_ptr_ar;
int bytes;
int ptr_ar_len;
} armci_giov_t;
```

For example, with the generalized I/O vector format a put operation that copies data to the process proc memory has the following interface:

```
int ARMCI_PutV(armci_giov_t dscr_arr[], int arr_len, int proc)
```

The first argument is an array of size arr_len. Each array element specifies a set of equally-sized segments of data copied from the local memory to the memory at the remote process proc.

2. Strided: This format is an optimization of the generalized I/O vector format. It is intended to minimize storage required to describe sections of dense multidimensional arrays. Instead of including addresses for all the segments, it specifies only an address of the first segment in the set for source and destination. The addresses of the other segments can be computed using the stride information.

For example, with the strided format, a put operation that copies data to the process proc memory has the following interface:

int ARMCI_PutS(src_ptr, src_stride_ar, dst_ptr, dst_stride_ar,

count, stride_levels, proc)

The first argument is an array of size arr_len. Each array element specifies a set of equally-sized segments of data to be copied from the local memory to the memory at the remote process proc. The ARMCI counterpart of the contiguous put operation available in other libraries can be easily expressed in terms of either vector or strided ARMCI put operation.

For example: ARMCI_PutS(src_ptr, NULL, dst_ptr, NULL, &nbytes, 0, proc)

2.3.1 ARMCI Implementation Details

ARMCI's implementation for clusters of workstations with generic TCP/IP networking support exploits the socket interface. An extra "data server" process or thread is forked on each cluster node to service one-sided communication requests from its clients. To prevent server thread/process in the absence of one-sided communication requests from consuming resources needed by the user processes, special care is needed to reduce CPU utilization by using blocking wait rather than active polling of the network interfaces. In the case of the TCP/IP protocol, "data server" exploits select with a descriptor set that includes sockets dedicated to communication with every task on remote nodes. This system call blocks until data or a request is received on at least one of the sockets. The performance is consistent with the hardware and IP protocol limitations.

The TCP/IP implementation of ARMCI can be used on Myrinet as well, as the Myrinet network, in addition to the GM interface, can also support the standard IP protocols. The Myricom implementation of IP, if enabled, consumes one of the eight GM ports. There have been several implementations of the TCP/IP protocol on Myrinet in the last several years, most notably by the Trapeze project [14] for FreeBSD.

2.3.2 Architecture of ARMCI over GM

The baseline GM communication end point for ARMCI was derived from its TCP/IP implementation. An extra pthread is created on every SMP node to play a role of data server. This thread opens two GM ports: one for receiving requests and one for sending a response. The flow control issues in GM make one port insufficient for that purpose the server would not be able to respond to the current client request unless all other pending requests from other clients are received and stored in memory for further processing. The number of GM ports used by ARMCI is independent of the number of MPI tasks on that node. The send port used in ARMCI is same as the one allocated by MPICH [16] to send. ARMCI requests to a port opened by the remote data server thread. Since ARMCI does not use GM receive operations on that port, it does not interfere with the MPI receive messages send to that port. Any response from the server is delivered by directed send (put) that does not require

GM receive operations (i.e., the data is directly placed in the client memory). In addition, the server sets a flag indicating to the client that the data transfer is complete.In order to avoid deadlock, the GM tokens associated with send operations on the client side must be recovered by GM. In GM send tokens are relinquished when application calls gm_unknown operation while polling for arriving messages using the GM receive operations. Since ARMCI shares the MPICH send port, it cannot relinquish the send tokens by issuing a gm_unknown call. Instead, it relinquishes its send tokens indirectly by calling MPI_Iprobe while waiting for a response from the server. MPI_Iprobe checks the network interface for MPI messages received on the same port and, by doing so, unknowingly relinquishes tokens associated not only with MPI but also with ARMCI messages.

The client communicates with the server by sending a request messages. The request message contains one required component, a header describing the request, and depending on the request type, it might also include a descriptor for the client buffer, a descriptor for remote memory that the request references, and data depending on the message type. The descriptor for the client buffer has been introduced in the GM port to provide the server with the ability to respond directly to the client, thus avoiding any intermediate buffering. As mentioned before, GM only supports data transfers from/to DMAable memory. Such memory can be obtained with a GM memory allocation routine. Alternatively, on some systems GM allows registration of already-allocated memory. There has been a lot of discussion in the research community lately regarding these memory registration issues in GM on Linux platforms. To maximize transfer rate, the zero-copy user level protocols in application level libraries require registration of the user memory. Since ARMCI applications can potentially use the entire physical memory in the system in the context of one-sided communication, it does not pin user data statically in ARMCI_Malloc. Instead, pinning is done dynamically as a part of data transfer operations, and when these operations are complete, the memory must of course be unpinned. This requires the server to use an extra copy when sending shared memory data via a DMAable buffer. On the client side, the user buffer is located most often in local memory and therefore, it can be pinned. ARMCI's baseline implementation for GM does not require memory registration and uses two DMA-able buffers allocated with GM by the server and client at the start up. The data must be transferred to and from these buffers, thus the corresponding two extra memory copies. This implementation scheme is in particular used on systems such as Solaris where the registration is not possible. ARMCI uses different, user level protocol friendly schemes for short messages or messages involving multiple short contiguous data segments. GM unlike VIA does not support scatter/gather operations.

With this architecture in mind and with an attention to the constraints GM imposes, we explore the implementations of the ARMCI Put and ARMCI Get strided operations in the next two chapters and recognize and implement optimization to them.

Chapter 3

PIPELINING THE PUT OPERATION IN ARMCI

ARMCI de-couples the communication and other aspects of any one-sided operation that it implements. This make it easier to expand the scope of the communication protocols on which ARMCI runs currently. Implementation of ARMCI Put or for that matter any one-sided operation in ARMCI has a general structure as shown in Figure 3.1. In this figure, the dotted box next to the first arrow indicates the data still being processed and the solid box adjacent to the lower arrow indicates completely processed data being transmitted out of the system.

We described about the different kinds of Put operations that ARMCI provides in section 2.3. Out of the two versions (vector and strided) of put operations available in ARMCI, we target the strided version for the wider range of data it covers and the challenge it poses in terms of non-contiguous data. The data processing layer breaks up a very large request into meaningful small requests and passes the handling of the requests sequentially to the communication layer. It also processes the requests to transmit non-contiguous data by copying them into contiguous transmittable locations. Among the data processing and the communication layers in ARMCI, the data processing layer is already very efficient. Since it is completely de-coupled from the communications part, it gives us an option to have our own buffer management

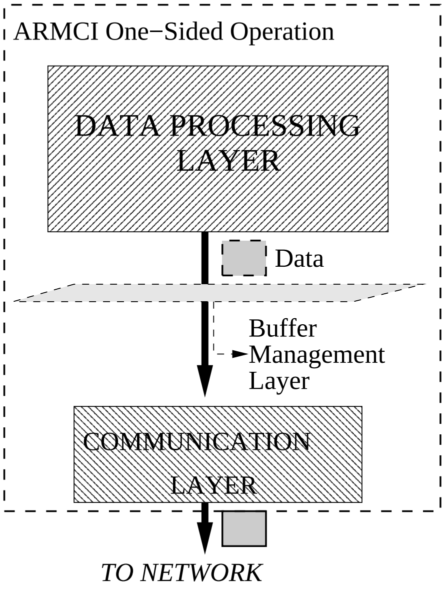


Figure 3.1: General Structure of an ARMCI one-sided operation.

scheme in between the data-processing and communication layers. It also gives us an opportunity to modify the communication layer without affecting anything in the data processing layer. The current ARMCI put operation follows a sequential transmission, both between simultaneous transmission and when using multiple buffers in a single transmission. With all these issues in mind, in this thesis, we attempt to analyze the data copy and communication aspects of the ARMCI put operation on the Myrinet GM network to improve its performance.

After describing the phases involved in a Put operation, we define and generalize the optimization possible for the current one-sided operation structure. As mentioned in section 1.1, most of these optimizations can be generalized as guidelines to how the communication layer in an one-sided communication library has to be defined to get the most out of the underlying user-level communication protocol. This chapter also discusses design alternatives that were evaluated to optimize the put operation in ARMCI.

3.1 Overall architecture

The architecture of ARMCI put can be viewed as a set of different implementations, each for one format of the transmission data. When the data is non-contiguous and large, as in a part of a multi-dimensional array, the request to transmit is broken down into multiple transmission requests of smaller-dimensional parts of this data. The division is based on the amount of data that fits in the transmission buffer. As a part of transmitting non-contiguous data, whether broken down or not, data has to be copied into a contiguous transmission buffer. After copying a buffer, the next buffer is processed/copied only after the first buffer has been transmitted by the communication layer (in Figure 3.1) and an acknowledgment for it has been received.

The current implementation of ARMCI Put can broadly be understood as comprising of three phases. Note that 'client' is a node that issues the put operation and 'server' is the destination node for this operation. Figure 3.2 shows these phases.

1. A data copy from the source data to the message send buffer (a pre-pinned, pre-allocated buffer, allocated when ARMCI is initialized), which we represent as COPYS phase. This data copy is done in the chunks of the sizes of the message send buffer.

2. Actual data transmission phase, where the the gm_send_with_callback function is used to actually transmit data in the message send buffer to the destination. In this phase the data from the message send buffer at the source node is sent to a receive buffer at the destination node. This phase is represented

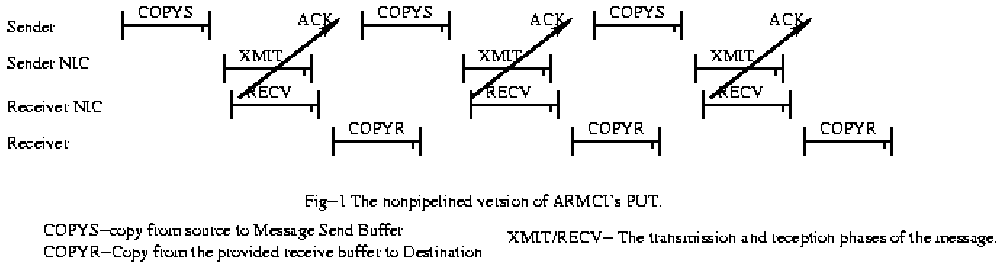

Sender
Sender NIC
Receiver NIC
Receiver
COPYS
ACK
XMIT
RECV
COPYR
Fig–1 The nonpipelined version of ARMCI's PUT.
COPYS–copy from source to Message Send Buffer
COPYR–Copy from the provided receive buffer to Destination
XMIT/RECV– The transmission and reception phases of the message.

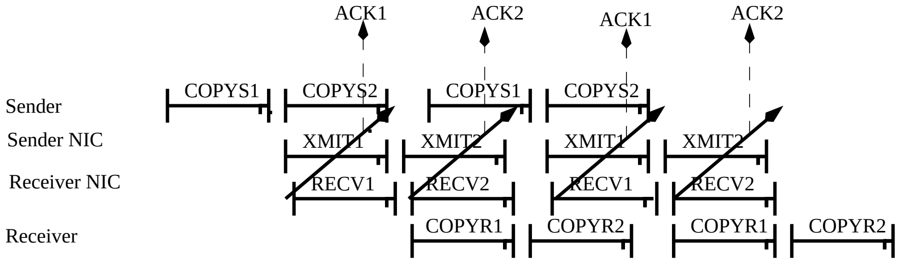
Figure 3.2: Three Phases of ARMCI Put operation in its current implementation.

as XMIT and RECV phase, where XMIT stands for an operation performed at the sender NIC to DMA the data and RECV is used to represent an operation performed at the receiver to receive the data into a Receive buffer.

3. Copying data to destination memory location from the receive buffer at the destination side, which is represented as the COPYR phase.

The Figure 3.2 shows the current implementation of ARMCI Put operation in terms of above defined phases. In phase 2, which comprises of both XMIT and RECV , the receiver NIC sends an acknowledgment for the received buffer to the source. This is represented by an arrow labeled ACK. Phase 1 occurs at the source node and Phase 3 occurs at the destination node. Phase 2 can be assumed to occur at both sender and receiver NIC. COPYS stands for copy from source data to send buffer. COPYR stands for copy from the receive buffer to the Receiver memory.

Figure 3.2 clearly indicates the possibility to utilize the time it takes to transmit a buffer in performing some other tasks. This task could be one or more of the Put phases. The following section explores and implements any improvements possible to this operation.



Fig−2 Pipelined implementation of PUT

There are two buffers on the sender side, COPYS1 indicates copy into one such buffer and COPYS2 into the second one. Similarly, One the receiver side, COPYR1 indicates copy into one of the two Available buffers, COPYR2 indicates a copy into the second one

Figure 3.3: Pipelined ARMCI Put operation

3.2 Pipelining the put operation in ARMCI

Below is a description of the design and implementation of a pipelined version of ARMCI Put operation essentially pipelining both the receiver and the sender side. In the pipeline, we overlap the XMIT and the COPYS phase as well as the RECV and the COPYR phase described above. Figure 3.3 shows how the pipelined Put operation has been implemented in terms of the phases defined above.

The pipelined implementation has multiple send and receive buffers. Hence the COPYS phase here is overlapped with the XMIT and the COPYR phase overlapped with RECV. Instead of copying one chunk of data, transmitting it, and then waiting for an acknowledgment, we maintain a set of send buffers. A heuristic to identify the number of required send buffers is discussed in this thesis in Section 3.2. After copying data into a buffer i and transmitting it, we check and wait for the previous chunk of data transmitted using buffer i+1 to be acknowledged. Once it has been acknowledged, we copy next chunk of data into the buffer i +1 and transmit it. This way, we end up overlapping all the three phases. This implementation of pipeline doesn't interfere with any other part of the existing ARMCI code. We continue using MPI_Iprobe instead of issuing a gm_unknown to receive any acknowledgments. COPYS1 indicates a copy into the first send buffer and COPYS2 indicates a copy into the second send buffer. Figure 3.3 shows how this can be overlapped with the transmission and reception of the previous buffer. Similarly on the receiver side, we have multiple receive buffers. COPYR1 shows copy from one such buffer to the local destination location. This again overlaps with phase-2 of the last buffer transmitted. This is achieved by having two receive buffers and receiving from either of them alternatively.

Of the options we had to design a 'pipeline' for efficiently executing three phases, we chose the above described method for the amount of overlap it provides between the phases in the put operation. We briefly discuss the design alternative we had to justify our choice in design.

3.2.1 Design choices

Our main objective was to utilize the time spent in waiting for an acknowledgment. Most of the design choices we had were based on ordering the phases in the put operation and to decide which of these phases can be executed concurrently and how. The following alternatives were explored:

- Registering the source and destination buffer and attempting a direct DMA transfer from the source to the destination (the no-copy alternative) and pipelining the registration and transmission phases.

- Using multiple send and receive buffers and alternating between them attempting to overlap phases involved in the operation.

In the first alternative, source buffer and destination buffer can be registered using a GM register memory call and the data can be DMAed directly between them. This scheme of course would need data copy if data is non-contiguous. Even if the data is contiguous, OS might have a constraint to the maximum size in the memory that can be registered. Distribution of the data determines if it is feasible to register the source and destination memories and DMA data directly between them.

The second alternative involves a single data copy, but the optimization can be designed in such a way to overlap the transmission with this data copy so that the effect of this copy latency is not perceived in the operation. Thus we select this alternative.

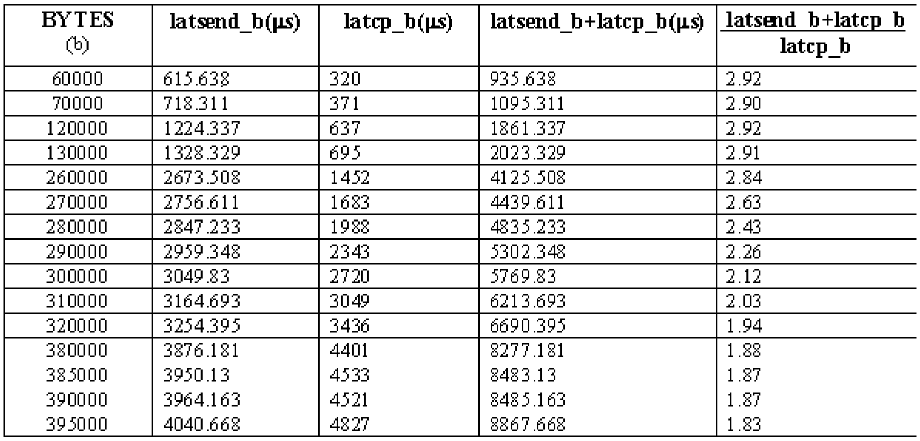
3.2.2 Depth and Efficiency of the Put Pipeline

If we compute the latencies of the three phases described in section 3.1 and also compute the time it takes to receive an acknowledgment, we can arrive at a theoretical verification of the enhancements our pipelined implementation can deliver. Currently, the send buffer size in ARMCI is 400000 bytes. For the above purpose, let the latency to copy a message of size B bytes using the ARMCI_copy function call, be latcp_b , and let the latency to transmit B bytes of data using a gm_send_with_callback be latsend_b. We can conclude that, if latsend_b is not significantly small in comparison to latcp_b, then there will be a definite improvement in the performance, if we can pipeline these two operations. We can also derive a number to the factor of improvement expected and compare it to the actual improvement achieved. This information can be used to dynamically determine a very close approximation to the ideal number of buffers that are required to be used for a given system. On repeating this for various buffer sizes, it is possible to find an ideal combination for the size of the send buffer and the number of send buffers required. The ratio of (latsend_b + latcp_b) to latcp_b for most of the larger buffer sizes (>300000 bytes) rounded to the nearest integer was found to be 2, and hence we use only two message send buffers. This analysis is reiterated by Figure 3.5 . Table-1 in Figure 3.4 shows the ratio of (latsend_b + latcp_b) to latcp_b for different message sizes on Pentium II 300MHz dual processor machines with LANai 7.2 and 66 MHz NIC processors. From Table-1 (3.4), for a buffer size of approximately 395000 bytes, if we can pipeline the copy and sending phases with two send buffers instead of one, we should be able to gain around 4040 microseconds. The latency of ARMCI put operation for a message size of 154880 bytes (approximately three times 395000) is around 35100 microseconds(from Table-1 in Figure 3.4). That means for this message size the improvement in Bandwidth should be around 13%. From Figure 3.7 and Figure 3.6 we can see that we actually get around 13.4% improvement on average with two send buffers for a message of size(1548800 bytes).

ARMCI Put operation has another design constraint that restricts the transmission pipeline from being deeper than 2, the number of GM send tokens available for it to transmit. In the following sub-section we briefly discuss this constraint.

3.2.3 Constraints imposed by MPICH on design

Since ARMCI uses MPICH's [16] send port to transmit data, the conflicts that could arise would mainly be because of the limit on the number of send tokens. Send tokens limit the number of outstanding sends from a port. MPICH people have agreed to designate one send token exclusively for the use by ARMCI. Thus our multi buffer



BYTES (b)	latsend_b(μs)	latcp_b(μs)	latsend_b+latcp_b(μs)	$\frac{\text{latsend\_b+latcp\_b}}{\text{latcp\_b}}$
60000	615.638	320	935.638	2.92
70000	718.311	371	1095.311	2.90
120000	1224.337	637	1861.337	2.92
130000	1328.329	695	2023.329	2.91
260000	2673.508	1452	4125.508	2.84
270000	2756.611	1683	4439.611	2.63
280000	2847.233	1988	4835.233	2.43
290000	2959.348	2343	5302.348	2.26
300000	3049.83	2720	5769.83	2.12
310000	3164.693	3049	6213.693	2.03
320000	3254.395	3436	6690.395	1.94
380000	3876.181	4401	8277.181	1.88
385000	3950.13	4533	8483.13	1.87
390000	3964.163	4521	8485.163	1.87
395000	4040.668	4827	8867.668	1.83

Figure 3.4: Table showing the copy and transmission latencies on Dual Pentium II 300 MHz machines

pipeline implementation has to make sure that there is only one outstanding send token at any given time. This in fact imposes another constraint on the design. We accommodate this constraint in our design and made sure by our way of receiving acknowledgment that there is always no more that one outstanding send from ARMCI through its GM send port.

Also, since the port ARMCI uses to send is an MPICH send port, it cannot listen on that port for acknowledgments. Though this saves ARMCI one active thread, receiving acknowledgments becomes a problem. Section 2.3 discussed how ARMCI uses the MPI_Iprobe call to indirectly retrieve the send token it uses. The following section discusses our pipelined implementation in detail and shows how this constraint has been incorporated into the implementation.

3.2.4 Implementation Details

The ARMCI Put operation currently uses only one pre-allocated and pre-pinned message send buffer for all the data transfers. It copies the data into the send buffer and transmits it using a gm_send_with_callback call. Since there is only one GM context, once the message is transmitted, it waits for the callback function to execute. This callback function updates the send flag in the context record. Our modifications to ARMCI Put use multiple buffers, a different callback function and a context record array, instead of a single context record variable. The context array is of the same size as the number of send buffers we have. When the acknowledgment for buffer i in our buffer array is received, it goes and updates the send flag in the context at the ith index in the context array. In short, an ith element in the context array corresponds to the ith buffer in the array of send buffers we have. Every time we need to transmit data, we check if the status of the previous send from the next buffer in order is complete. If it is not, then we wait on the corresponding send flag for that buffer to be updated. Otherwise, we copy the data into that buffer and transmit. We also order the different phases on the sender/client to take maximum advantage of the delay that always occurs between the transmission of a buffer and the receipt of an acknowledgment for it.

In addition to the above mentioned enhancements, the server side has also been modified to use multiple receive buffers, which actually enhances the performance of other remote memory operations as well. When the server receive function receives a buffer of a certain size it immediately provides another buffer of the same size before copying the current received buffer into the local destination location. This enhances the receiver (server) side of the ARMCI Put operation. It should be noted

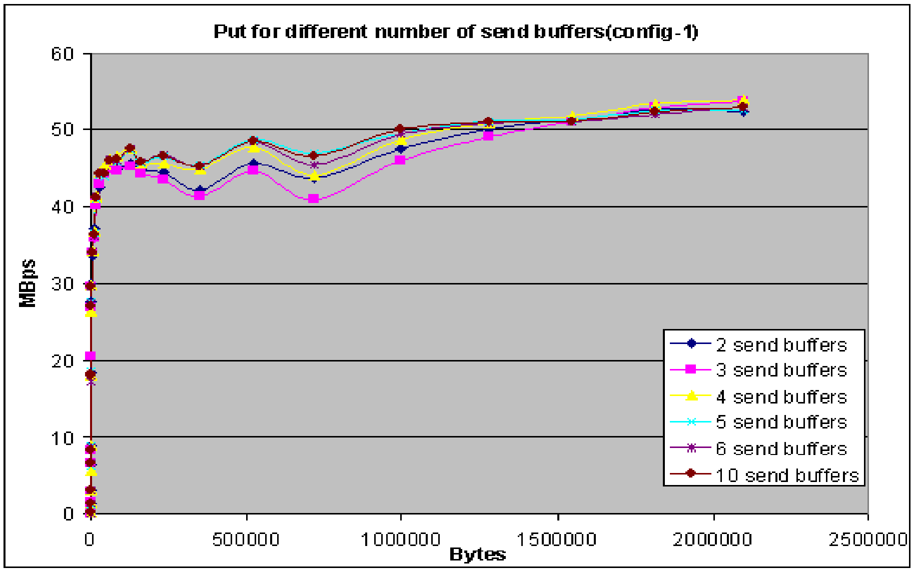


Figure 3.5: Put with multiple send/receive buffers on Dual Pentium 300MHz machines.

that on the receiver side, it helps to have more that two receive buffers even though it does not have any significant effect on the performance of the ARMCI put operation. Performance of ARMCI for multiple send buffers is shown in Figure 3.5. Notice that as the number of buffers increases beyond two, there isn't a significant difference in the performance.

3.3 Generalizing pipelining parameters

From the above discussions, we can come up with a set of parameters that actually affect the performance of ARMCI operations and then generalize them as parameters that need to be tuned, either dynamically or in a static sense for any one-sided communication library. A few of these parameters would be

- Number of send buffers

- Size of the send buffers

- Number of receive buffers

- Transmission Latency (dependent of the size of the buffer)

- Copy latency

- Memory registration latency

Number of send buffers and the size of each of these send buffers have a very significant impact on the performance of one-sided operation. Later in Chapter 5 we propose a dynamic scheme for determining the same. In any generic one-sided library for cluster environments the performance of the underlying communication protocol for different transmission buffer sizes acts as a preliminary estimate to what should the size of the transmission buffer be. Note that, in cases where source memory is registered before transmission, the chunk of the source memory being transmitted at one go is the transmission buffer size. A very good estimate of the size and the number of send buffers that are to be used for any one-sided communication library can be determined by examining the transmission latency and comparing it with the memory registration and memory copy latencies for multiple message sizes. The number of receive buffers to be used need not necessarily depend on the number of send buffers. The criterion to determine the number of receive buffers for any operation in a one-sided communication should be to make sure that no sent packet has to be re-transmitted for the lack of ready receive buffers the GM documentation[8]mentions that two receive buffers for a message size should be enough to take care of this flow control issue in most of the cases. A discussion to how all these parameters have been determined for pipelined put in ARMCI for its communication on GM can be seen in sections 3.1 and 3.2.

3.4 Performance Evaluation

We evaluated our results on two different configurations

1) A cluster of dual Pentium II 300 MHz machines which have LANai 7.2 cards with 66 MHz NIC processors connected to a Myrinet LAN via a 8 port Myrinet switch (config-1, for future references).

2) A cluster of quad Pentium III 700MHz machines (Dell 6400's) which have LANai 7.2 cards with 66MHz NIC processors connected to a Myrinet LAN via a 4 port Myrinet switch (config-2 for future references).

We evaluated the pipelined put scheme on both the system configurations mentioned above. Though the put pipeline also enhances the performance of the ARMCI accumulate operation, we concentrate only on ARMCI Put results since the behavior of Accumulate operation is very similar to the Put operation. We initially show the put performance on both system configurations for a different number of send buffers and then compare the pipelined implementation to the non-pipelined implementation.

Figure 3.6 the pipelined ARMCI put performance improvements on config-1 (defined above). It shows an average improvement of around 9% for all the messages greater that 400000 bytes and a best case improvement of around 13%. The upper two lines represent contiguous and non-contiguous data transfers for our pipelined implementation and the lower two lines represent contiguous and non-contiguous data transfers for the current ARMCI implementation.

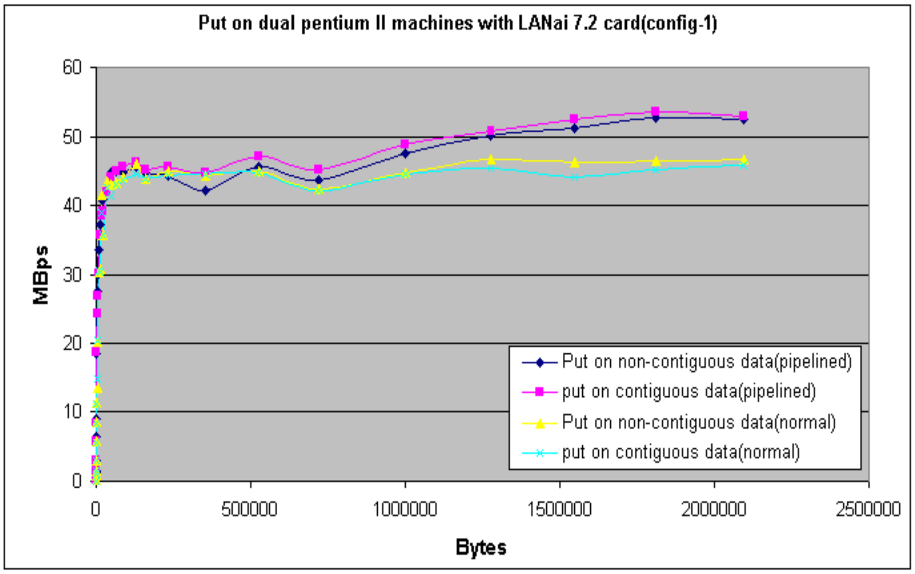


Figure 3.6: Comparison of current and optimized put on Config-1

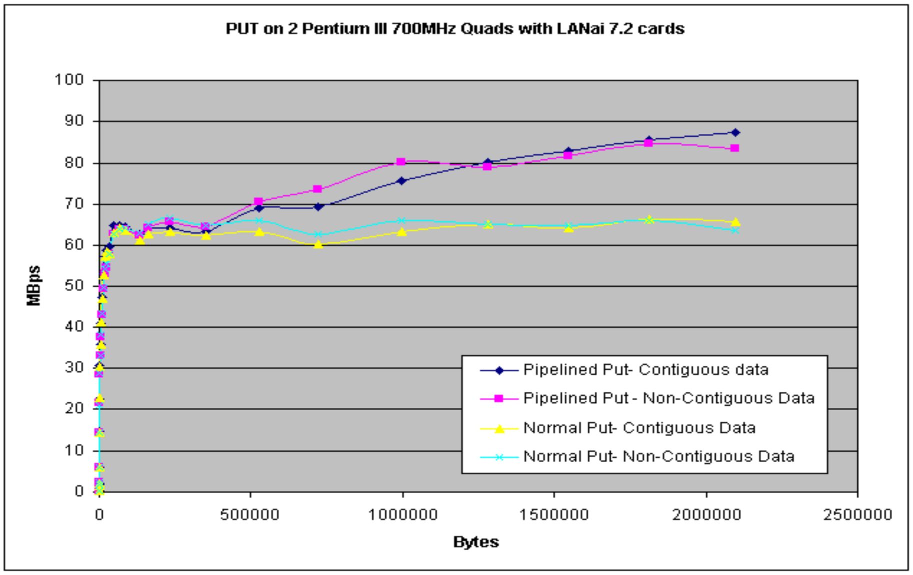


Figure 3.7: Comparison of current and optimized put on Config-2

Figure 3.7 shows the same performance on config-2 (defined above). In this case we get an average case improvement of 19% and a best case improvement around 24%. Even in this figure, the upper two lines represent contiguous and non-contiguous data transfers for our pipelined implementation and lower two represent contiguous and non-contiguous data transfers for current ARMCI put implementation on config-2.

Chapter 4

PIPELINING THE ARMCI GET OPERATION

ARMCI Get operation is also implemented based on the generic one-sided operation structure shown in Figure 3.1. A node that issues the Get call is the client node or client and the node that has the data and to which this request is directed to is referred to as the server node or server. This should not be confused with the data server, which is a process/thread running on all the nodes. In Get, the actual data communication is performed on the server node. ARMCI get operation's implementation on GM performs the packing of the data to be transmitted on the client side. Based on the size of the data requested, client either directly requests for the entire data or requests for data in chunks of the client buffer size. For requests that are larger than one client buffer the client can do one of the following:

1)Register the memory into which data has to be received and request a direct transfer into this location or

2) send requests for multiple chunks of data each of the size of the client message buffer till all the requested data is received.

The second case is crucial because due to Operating System constraints and high memory registration latencies, it is not always possible to register memory. Since the entire process of sending a request, waiting for data to arrive and copying the data is done sequentially, it consumes a lot of time and paves a way for some improvement.

4.1 Overall architecture

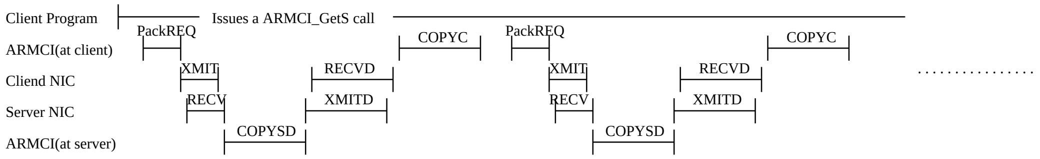
In ARMCI get operation client requests the server for data and the server responds to the clients request by sending the data to the client node. This client node initially copies its request for one buffer into a send buffer (pre-registered buffer) and transmits it to server. Server receives the request and copies the data client requested into a send buffer and transmits it to client. Based on whether the client is able to register memory (if the request is for contiguous data) or not, the server sends the data either directly into the client buffer or into the message receive buffer on the client node which in turn copies the data into the client buffer. This entire sequence of events can be understood clearly by dividing the get operation into phases. The phases in the get operation include:

1. Break-up of the get request into multiple requests on data in the size of the pre-pinned receive buffer, represented as the PackREQ phase.

2. XMIT/RECV indicate the transmission and reception of the request from client to server

3. A phase representing the copy of data from the process memory on the server node into a pre-pinned message send buffer on the server node, for transmission, is shown as the COPYSD phase.

4. XMITD/RECVD phases which represent the transmission of the message by the server NIC and receipt of the message by the client as a response to a get request.



The Current ARMCI get − Order of events.

1. PackREQ − Client in the get call, breaks up the entire get request into multiple requests for data of the size of the pinned buffer
2. XMIT/RECV − The tranmission and the reception phases of each of the multiple requests
3. COPYSD − The copy of the data at the servernode into a pre−pinned buffer for transmission
4. XMITD/RECVD − The transmission and reception phases of each buffer transmitted by the server in responce to the client request
5. COPYC − The copy of the message into the destination location fron the pinned receive buffer

Figure 4.1: Phases of the ARMCI Get Operation

5. COPYC phase representing copy of the message/data at the client from the message receive buffer into the client location.

. These phases are executed sequentially leaving a lot of place for improvement. For the cases where the client memory can be registered, current ARMCI Get implementation pipelines and overlaps the request for data and registration of memory, otherwise, it follows the above described sequence of events. There is no overlap in the sequence of execution of these phases currently. Phase 1 starts again only after completion of phase 5. Another concern is the number of sends that are associated with each buffer transfer. From Figure 4.1 it can be seen that for each buffer that is received as a part of the Get operation, two transmissions are required, one in the form of a request from client to server node and the other in the form of the data from server to the client node. Another transmission hidden in Figure 4.1 is the acknowledgment that the server sends for each buffer informing the client that it has finished transmitting the buffer. Also, for each of the RECV's seen in the Figure 4.1 the data server process is interrupted once.

The following section shows an efficient way to implement the Get operation so as to overlap the phases in it and also to reduce the total number of sends and interrupts associated with each Get call.

4.2 Optimization to the ARMCI get operation

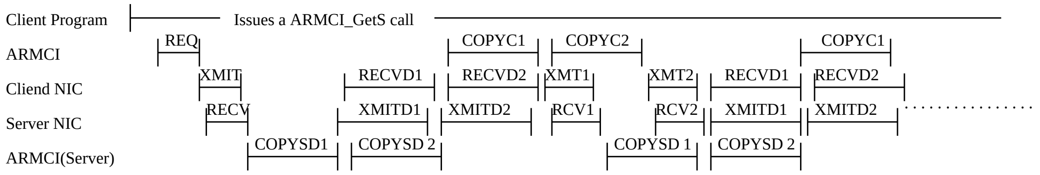
For the cases when the buffer cannot be registered, either because the data is strided or because of the failure of the gm_register call, we still pipeline the Get operation. The current Get operation for strided data and for data that cannot be registered, we were using a sequential approach to transmit data. We improve this part of the get operation to support pipelining. To achieve this, we use two buffers each on the client and server side. The client informs the server of the availability of a buffer by directly writing to a ready flag at the server. Server checks for a ready flag before doing a direct send into a pinned buffer on the client side. Instead of breaking up Get request into smaller requests of the size of a pinned buffer at the client, we just send one request to the server and let the server break up the request, thereby reducing the total number of interrupts involved in a get.

The phases in the improved get operation when the registration of the client memory is not possible are

1. The Phase XMIT/RECV represent a single request for the entire get operation.

2. The COPYSD phase represents the copying of the data from the source location at the server to the pinned send buffer at the server.

3. The XMITD/RECVD phases represent the transmission of data from the server pinned buffer to the client pinned buffer.



The Pipelined ARMCI get call, Order of events.

XMIT/RECV − The tranmission and the reception phases of each of the multiple requests

COPYSD − The copy of the data at the servernode into a pre−pinned buffer for transmission

XMITD/RECVD − The transmission and reception phases of each buffer transmitted by the server in responce to the client request

COPYC − The copy of the message to the destination location

XMT/RCV − Transmission and reception phases of an ack informing the availability of a buffer to the server.

Figure 4.2: Pipelined Get Operation

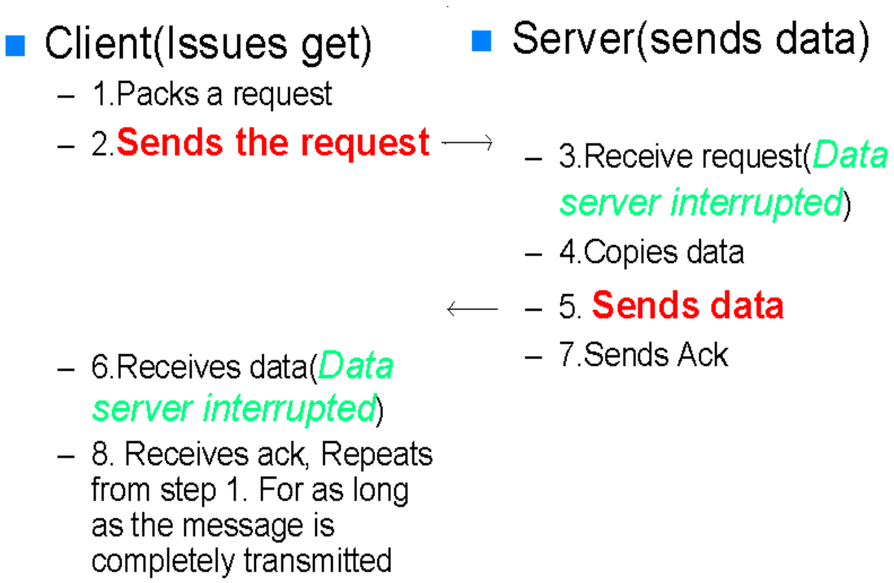
4. COPYC1 and COPYC2 describe the copying from the client pinned buffer into the process memory on the client.

5. XMT/RCV represent the transmission and receipt of the ready flags indicating the availability of receive buffer at the client.

An acknowledgment RCV1 represents the readiness of the first buffer. Figure 4.2 shows the overlap of these phases. On the server node, the transmission of a buffer to client and the copying of a buffer from process memory to the pinned buffer on server are overlapped. Similarly on the client side, the receiving of a buffer is overlapped with copying of the buffer into the requesting process memory on client. Other than the visible advantages of overlapping the phases, we also reduce total number of sends and interrupts associated with a get operation.

4.3 Design choices

The sequence of events in the Get operations give us two options to implement the pipeline.



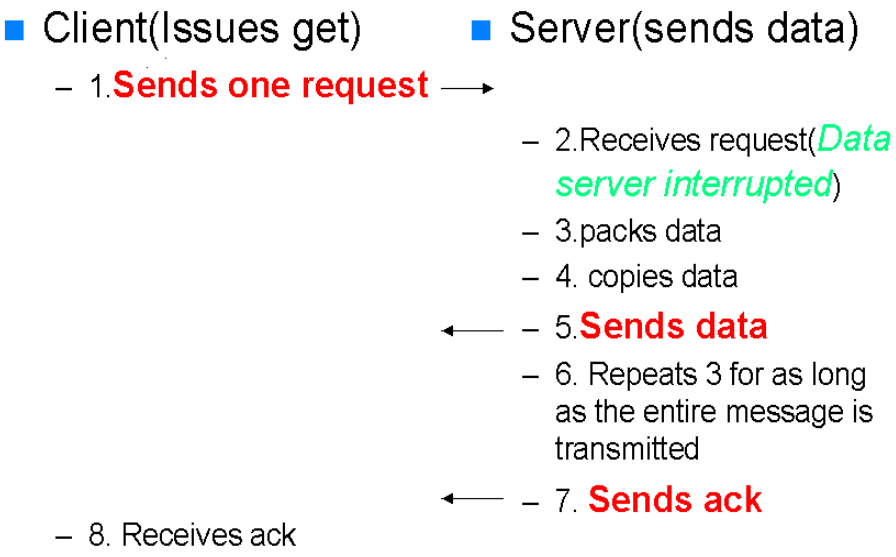
- Client(Issues get)
  - 1.Packs a request
  - 2.**Sends the request** ⟶
  - 6.Receives data(*Data server interrupted*)
  - 8. Receives ack, Repeats from step 1. For as long as the message is completely transmitted
- Server(sends data)
  - 3.Receive request(*Data server interrupted*)
  - 4.Copies data
  - ⟵ 5. **Sends data**
  - 7.Sends Ack

Figure 4.3: Sequence of events in Get

- Pipelining with packing on the client side
- Pipelining with packing on the server side

In order to explore both the options we have, and justify the choice made, the sequence of events in a Get operation are shown in figure 4.3 . For each 'buffer' transmitted, there are two communications, one in the form of client sending the request to server and other in the form of server sending the data to the client. The data server server process in the client and the server is interrupted once each.

In this figure, The steps 2 and 5 indicate transmission and steps 3 and 6 indicate an interrupt. The objective of any design that modifies the sequence of events in the current Get operation should be to reduce the number of interrupts and transmissions required to complete a get operation. The design choices we have vary by where we want to pack the data in case it is non-contiguous or larger than a single buffer. The



- Client(Issues get)
  - 1.**Sends one request** →
  - 8. Receives ack
- Server(sends data)
  - 2.Receives request(*Data server interrupted*)
  - 3.packs data
  - 4. copies data
  - ← 5.**Sends data**
  - 6. Repeats 3 for as long as the entire message is transmitted
  - ← 7. **Sends ack**

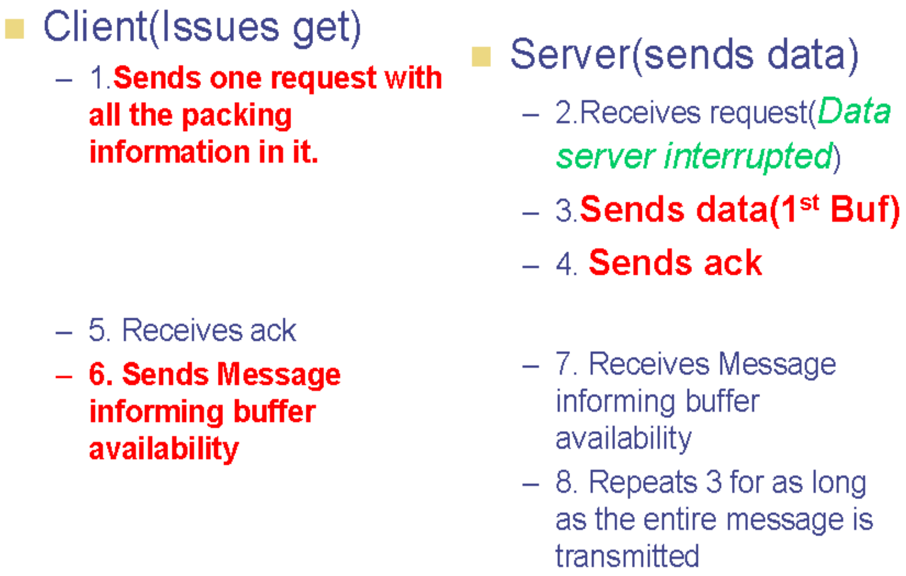
Figure 4.4: Sequence of events in get design-1

sequence of events with packing on client side and packing on the server side are shown in Figures 4.4 and 4.5.

Figure 4.4 shows that when we pack on the server side, we just send one request and the data server is interrupted only once. With the packing on the client side as seen in 4.5, we have client packing and sending the entire packing information in one message. In this design we see that the number of sends in not significantly reduced. Clearly we have lesser number of transmission and interrupts with the first design, and hence our implementation.

4.3.1 Implementation Details

We designed and implemented a more efficient version of the Get operation. Instead of breaking up the request at the client side and sending multiple requests, we break up the request a the server side and do direct sends into the client pre-pinned



- Client(Issues get)
  - 1.**Sends one request with all the packing information in it.**
  - 5. Receives ack
  - **6. Sends Message informing buffer availability**

- Server(sends data)
  - 2.Receives request(*Data server interrupted*)
  - 3.**Sends data(1st Buf)**
  - 4. **Sends ack**
  - 7. Receives Message informing buffer availability
  - 8. Repeats 3 for as long as the entire message is transmitted

Figure 4.5: Sequence of events in get design-2

receive buffers after confirming their availability. Once the client detects data in one of these receive buffers it decodes the descriptor and writes the data to the destination. The client process in not interrupted for every buffer transmitted. Instead it continuous polls for the buffer to be completely written. With this kind of polling, CPU cycles are not wasted because the client starts polling only when and if it requests some data server for data. The client informs the server about buffers availability by a direct write into a server buffer flag. Server polls on this flag to do a direct write into a client buffer. Since we use multiple buffers, client alternates between the multiple buffers it has and thereby utilizing the delay between consecutive server writes. Similarly, since the server uses multiple buffers for transmission, it alternates between the buffers and overlaps the transmission into a buffer and copy into the next available buffer.

4.4 Performance evaluation

We evaluated our results on two different system configurations

1) A cluster of dual Pentium II 300 MHz machines which have LANai 7.2 cards with 66 MHz NIC processors connected to a Myrinet LAN via a 8 port Myrinet switch.

2) A cluster of quad Pentium III 700MHz machines(Dell 6400 s) which have LANai 7.2 cards with 66MHz NIC processors connected to a Myrinet LAN via a 4 port Myrinet switch.

We evaluated the improved get scheme on both the system configurations mentioned above.

Figure 4.6 shows the performance of the optimized get operation on dual Pentium III 300MHz systems. The darker line is the current ARMCI put implementation for non-contiguous data transfers and the upper line shows our optimized implementation. We achieve an 8.9% improvement over the current get operation.

Figure 4.7 shows the performance of the optimized get on Quad Pentium III 700 MHz systems. On these systems, we get an improvement of 5% over the current implementation.

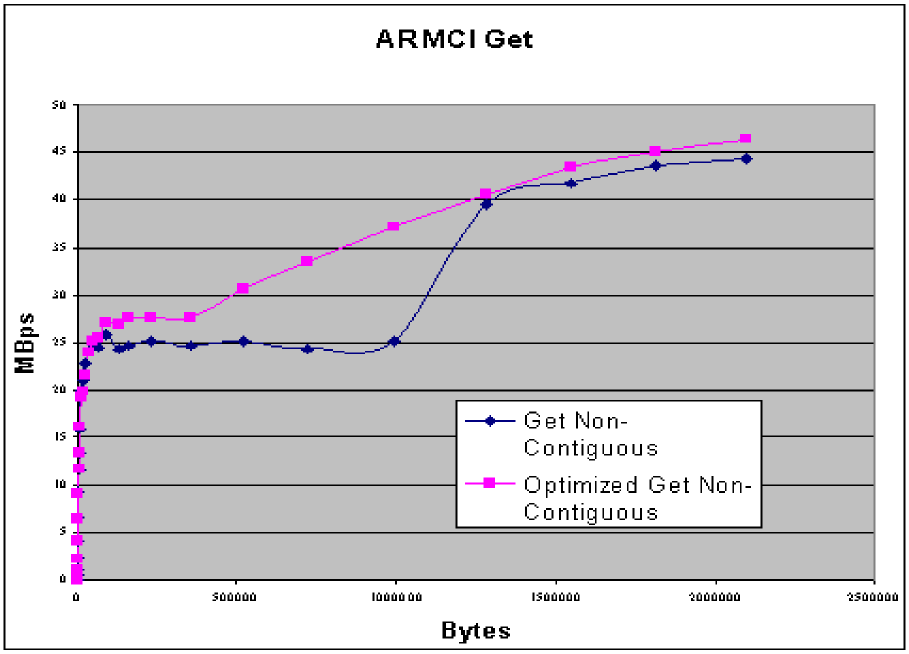


Figure 4.6: Get Performance on dual 300MHz Machines (for Non-Contiguous data)

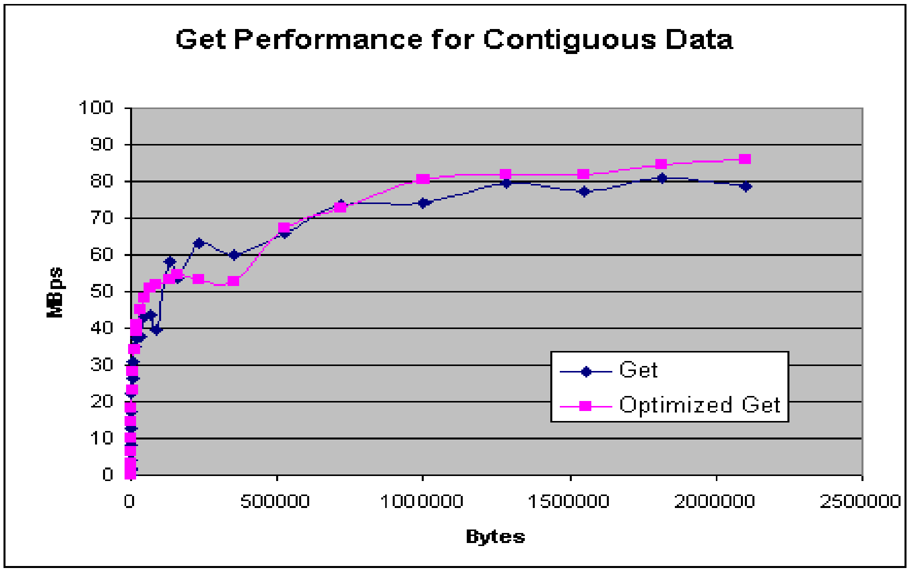


Figure 4.7: Get Performance on Quad 700MHz machines for contiguous data

Chapter 5

PIPELINING WITH BUFFER SIZE AS A DYNAMIC PARAMETER

Parameters that influence the pipelined implementation of Put and optimized Get implementation can also be tuned dynamically. One parameter that has a lot of influence on the performance of all the one-sided operations is the buffer size. Currently ARMCI has a fixed buffer size of 400000 bytes. For modified implementations of Get and Put, we still use multiple buffers each of size 400000 bytes. Hence in multi-buffer cases, OS ends up allocating a lot of non-swappable memory. With so much memory allocated, it becomes necessary to try and utilize the requested memory efficiently. By fixing the buffer size in transmission, we end up under-utilizing most of this pre-allocated memory due to internal fragmentation in buffers. Since allocating pinned memory is very costly for an operating system, this is a terrible waste of available physical memory and hence the resources on the system.

For a Put operation with two buffers available, consider two simultaneous put requests, one for 390000 bytes of data and the other for 10 bytes of data. As these requests arrive at the buffer management layer (Figure 3.1) requesting for a buffer, first buffer is given to the first request for transmitting 390000 bytes. Second buffer which is assigned to the second request for transmitting 10 bytes. Thus we end up not utilizing 399990 bytes in the second buffer till we receive an acknowledgment for the data in it. Though having a simpler buffer management scheme is an advantage, it is very important to make sure this scheme efficiently utilizes the memory. The question thus becomes: Can the buffer management scheme be made more memory efficient without any compromise in performance? The solution is to make buffer size as a dynamic parameter. We treat the whole available pinned/registered memory as a single chunk of memory. As a request of certain size arrives we check if a memory of that size is available, and assign it to the request. This technique definitely utilizes the space available in the form of a transmission buffer more efficiently. The sections below describe the implementation of this scheme with buffer size as a dynamic parameter and discuss the issues with this dynamic buffer scheme.

5.1 Implementation details

In the buffer management layer as seen in Figure 3.1 we pass the size of the buffer required as a parameter. Based on this, the dynamic buffer assignment routine assigns a part of the available memory as the buffer and returns a pointer to this assigned memory. The buffer assignment routine also keeps track of the available buffer space and acknowledgments to the transmitted buffers. On request for a buffer, if enough buffer space is not available, it waits for all the previous pending requests to complete and starts re-assigning memory. As buffers are assigned, memory is divided into slots. Thus, on assignment of the buffer space for the first request, the entire buffer space will have two slots, first slot representing the assigned memory and second representing the free memory. With a new request for buffer space, the buffer assignment routine starts searching from the first assigned slot and goes through all the slots for available/acknowledged memory. If no slot is available, it waits and frees all the slots and restarts the assignment process. The entire implementation revolves around breaking up the buffer and managing these slots. Though waiting for all the pending requests is not the most feasible solution, it simplifies this buffer management scheme by a lot. Also, any intelligence applied in freeing the slots can result in more external fragmentation. In comparison to the static buffer allocation scheme, which does not consider any memory utilization, this scheme has a lot of advantages and is also efficient in eliminating any internal fragmentation due to under-utilization of buffers.

5.2 Issues with dynamic buffer size

There are many issues associated with implementing the dynamic buffer scheme. Below is a description of these issues and a discussion to how these issues have been addressed or need to be addressed in our implementation of the dynamic buffer scheme. The trade off between the advantages and complications of using dynamic buffers is particular to the implementation of a one-sided communication library.

The general issues with a dynamic buffer scheme are

- External Fragmentation of memory
- Memory registration
- Acknowledgments for Direct Write
- Complicated Buffer Management

Any two (or more) slots with available memory totaling to the requested buffer space cannot still be utilized as they could be physically separated. Since our scheme uses 'memory slots' to assign memory, and it continues to assign memory from the buffer as long as it is available, there is an obvious chance of fragmentation. This is partly addressed by searching from slot 0 every time a slot has to be assigned.

For user level communication protocols that require data to be in registered memory locations, dynamic buffers can save a lot of resources. Any pinned memory cannot be swapped out, which means that the OS doesn't have that memory for other applications. Minimizing this memory usage is a definite advantage. Since the user level communication protocols allow direct write into the pinned memory locations, the challenge is to support the dynamic buffer scheme at the receiver side. This is accomplished by using a tag field in the message header informing the buffer size with other details. Also, since the receiver does not generate specific acknowledgments for direct write, it becomes necessary to incorporate intelligence in the algorithm to understand the acknowledgments generated for a direct write.

This buffer management technique is definitely more complicated than the traditional static buffer scheme because of its dynamic nature but it is very memory efficient. We are working towards a robust implementation of this dynamic buffer management scheme.

Chapter 6

CONCLUSIONS AND FUTURE WORK

In this thesis we proposed and implemented a pipelined implementation for ARMCI Put operations by pipelining and overlapping both client and server side of the Put operation. We also optimize ARMCI get operation by not just pipelining the buffer flow but also by reducing the total number of sends and interrupts associated with every buffer transfer. We also introduced a dynamic buffer management scheme to manage buffer allocation and flow more efficiently. Our work paves a path for defining a substrate between user level communication protocols and one-sided operations.

As future work, we would like to try and make the buffer sequencing more intelligent in both get and put operations. We also would like to incorporate the dynamic buffer concept in optimizing the Get and Put operations. Based on our work in this thesis, we are trying to design a common abstract layer that can be used as a layer between one-sided operations and multiple user level communication protocols with similar architecture. We would be attempting to address the issue by tuning the parameters that we realized in this thesis as a part of the definition of this abstract layer.

Bibliography

[1] MPI Forum.MPI-2: Extension to message passing interface. Technical report, University of Tennessee, July 1997

[2] J. Nieplocha, J. Ju, E. Apra. One-sided Communication on Myrinet-based SMP Clusters using the GMMessage-Passing Library

[3] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries

[4] J. Nieplocha, R. Littlefield, and M. Rosing Beyond Message Passing: A Case for One-Sided Communication in MPI

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

[6] 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.

[7] T. von Eicken, D. Culler, S. C. Goldstein, K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In Proceedings of the 19th International Symposium on Computer Architecture:256–266, May 1992.

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

[9] S. Pakin, M. Lauria A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM)for Myrinet. In Proceedings of Super-computing '95, December 1995.

[10] 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.

[11] R. A. F. Bhoedjang, T. Ruhl, H.E. Bal. User-Level Network Interface Protocols

[12] 10 Gigabit Ethernet Technology Overview White Paper available at http://www.10gea.org/Tech-whitepapers.htm .

[13] Daniel Scales, Kourosh Gharachorloo, Anush Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters (1997) Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)

[14] Kenneth G. Yocum, Jeffrey S. Chase, Andrew J. Gallatin, and Alvin R. Lebeck. Cut-Through Delivery in Trapeze: An Exercise in Low Latency Messaging, IEEE Symposium on High-Performance Distributed Computing (HPDC), Portland OR, August 1997.

[15] Hongzhang Shan and Jaswinder Pal Singh. A Comparison of MPI, SHMEM and Cache-coherent Shared Address Space Programming Models on the SGI Origin2000.

[16] William Gropp, Ewing Lusk. User's Guide for mpich, a Portable Implementation of MPI (1996)