View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by PDXScholar

Portland State University

PDXScholar

Dissertations and Theses | Dissertations and Theses

10-27-1995

Performance Evaluation of Specialized Hardware for Fast Global Operations on Distributed Memory Multicomputers

Rajesh Madukkarumukumana Sankaran
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Electrical and Computer Engineering Commons

Let us know how access to this document benefits you.

Recommended Citation

Sankaran, Rajesh Madukkarumukumana, "Performance Evaluation of Specialized Hardware for Fast Global Operations on Distributed Memory Multicomputers" (1995). *Dissertations and Theses.* Paper 4919.

10.15760/etd.6795

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

THESIS APPROVAL

The abstract and thesis of Rajesh Madukkarumukumana Sankaran for the Master of Science in Electrical and Computer Engineering were presented October 27, 1995, and accepted by the thesis committee and the Master's program.

COMMITTEE APPROVALS: ______________________
Douglas V. Hall, Chair

____
Michael A. Driscoll

____
Jingke Li
Representative of the Office of Graduate Studies

MASTERS PROGRAM APPROVAL: ______________________
Rolf Schaumann, Chair
Department of Electrical Engineering

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by ______________________ on 14 December 1995

ABSTRACT

An abstract of the thesis of Rajesh Madukkarumukumana Sankaran for the Master of Science in Electrical and Computer Engineering presented October 27, 1995.

Title: Performance Evaluation of Specialized Hardware for Fast Global Operations on Distributed Memory Multicomputers

Workstation cluster multicomputers are increasingly being applied for solving scientific problems that require massive computing power. Parallel Virtual Machine (PVM) is a popular message-passing model used to program these clusters. One of the major performance limiting factors for cluster multicomputers is their inefficiency in performing parallel program operations involving collective communications. These operations include synchronization, global reduction, broadcast/multicast operations and orderly access to shared global variables. Hall has demonstrated that a secondary network with wide tree topology and centralized coordination processors (COP) could improve the performance of global operations on a variety of distributed architectures [Hall94a].

My hypothesis was that the efficiency of many PVM applications on workstation clusters could be significantly improved by utilizing a COP system for collective communication operations. To test my hypothesis, I interfaced COP system with PVM. The interface software includes a virtual memory-mapped secondary network interface driver, and a function library which allows to use COP system in place of PVM function calls in application programs. My implementation makes it possible to easily port any

existing PVM applications to perform fast global operations using the COP system. To evaluate the performance improvements of using a COP system, I measured cost of various PVM global functions, derived the cost of equivalent COP library global functions, and compared the results. To analyze the cost of global operations on overall execution time of applications, I instrumented a complex molecular dynamics PVM application and performed measurements. The measurements were performed for a sample cluster size of 5 and for message sizes up to 16 kilobytes.

The comparison of PVM and COP system global operation performance clearly demonstrates that the COP system can speed up a variety of global operations involving small-to-medium sized messages by factors of 5-25. Analysis of the example application for a sample cluster size of 5 show that speedup provided by my global function libraries and the COP system reduces overall execution time for this and similar applications by above 1.5 times. Additionally, the performance improvement seen by applications increases as the cluster size increases, thus providing a scalable solution for performing global operations.

PERFORMANCE EVALUATION OF SPECIALIZED HARDWARE FOR FAST GLOBAL OPERATIONS ON DISTRIBUTED MEMORY MULTICOMPUTERS

by

RAJESH MADUKKARUMUKUMANA SANKARAN

The thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

In

ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1995

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisors, Dr. Douglas V. Hall and Dr. Michael A. Driscoll who guided me all through this project, and taught me much about computer architecture. I thank Dr. Jingke Li and all the above people collectively for serving on my thesis committee and for carefully reviewing my work. Thanks to Mr. Jim Binkley for his very helpful suggestions in the areas of networking protocols and tips for advanced network programming.

On a personal note, I wish to thank my family back home, for providing me the opportunity to pursue my graduate studies at Portland State University. Also thanks to Pyramid Technology Corporation for allowing me to use their resources for my academic purposes.

TABLE OF CONTENTS

PAGE

ACKNOWLEDGEMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

I INTRODUCTION 1

The Distributed Computing Scene 1

II PROGRAMMING MODELS FOR CLUSTER MULTICOMPUTERS 5

Introduction to Cluster Multicomputers 5

A Brief Survey of Multicomputer Programming Models 7

III CHOOSING A PROGRAMMING MODEL FOR THE COP SYSTEM 10

Why Choose PVM? 10

PVM Message Passing System 13

PVM Communication Facilities 20

PVM Collective Communication Primitives 22

Performance Limiting Factors in PVM 25

IV COP SYSTEM ARCHITECTURE AND OPERATIONS 27

Goals 27

Top Level View 28

The Compute Node to COP Network Interface 32

COP Architecture and Operations 35

V DESIGN AND IMPLEMENTATION OF THE COP SOFTWARE 43

COP Message Format 43

Layered Architecture of the COP-PVM System 49

COP Controller Address Spaces 52

Additional COP Hardware Features 57

The COP Software Implementation 60

Libcop Library 63

VI COP PERFORMANCE ANALYSIS AND COMPARISONS 79

Performance Analysis of Libcop Functions 81

Performance Evaluation of Libpvm Functions 94

PVM Collective Communication Performance 98

Effect of Global Operation Speedup on Overall Execution Time 105

Performance Comparisons 108

VII RELATED WORK 112

Cluster Computing Over High Speed Networks 112

VIII CONCLUSIONS 119

LIST OF TABLES

TABLE PAGE

I COP COMMANDS AND OPCODES 42

II SUMMARY OF LIBCOP & LIBPVM LIBRARY FEATURES 78

III LIBCOP FUNCTION TIMES AND EXPRESSIONS 93

IV LIBCOP AND LIBPVM PERFORMANCE COMPARISONS 104

V TIMING ANALYSIS OF MOLECULAR DYNAMICS APPLICATION..........110

LIST OF FIGURES

FIGURE PAGE

1 PVM Generic Task Identifier 15

2 PVM Multiprocessor Task Identifier 16

3 PVM TID Sub-Space Allocations 17

4 Single Level COP System Topology 28

5 Two Level COP System Topology 29

6 COP Channel Interface 33

a) Compute Node to COP network interface circuitry 33

b) Format for COP instruction word showing sub-fields 33

7 Block Diagram of a Co-ordination Processor 36

8 Protocol Stacks for PVM and COP 41

a) Protocol Stacks for PVM (Normal Route) 41

b) Protocol Stacks for COP system 41

9 COP Message Format 44

a) Instruction Word 44

b) Address Word 44

c) Data1 Word 44

d) Data2 Word 44

10 Combined COP-PVM Model Communication Resources 49

11 Control and Status Register 51

a) Control Bits 51

a) Status Bits 51

12 Address Space Partitions for Data SRAM on COP controller 53

13 Task Address Space Partitions within a Channel Address Space 54

14 COP Message Format for SETTC operation 60

15 PVM Latency Measurements for small sized messages 97

16 PVM Barrier timing measurements 99

17 PVM Reduction timing measurements 100

a) PVM Reduction timings for varying number of participating tasks 100

b) PVM Reduction timings as a function of data array size 100

18 PVM Broadcast/Multicast timings 102

a) PVM Broadcast timings as a function of message size 102

b) PVM Multicast timings as a function of message size 102

CHAPTER I

INTRODUCTION

The Distributed Computing Scene

One of the main motivations for developing powerful parallel computers has been to solve very large scientific problems. The objectives and focus of these development efforts is summarized in the report on "High Performance Computing and Communications: Foundation for America's Information Future" [Ostc95a] by the U.S Office of Science and Technology's Committee on Physical, Mathematical, and Engineering Sciences. The HPCC report includes a "wish list" of important computational problems that scientists and engineers would like to be able to solve in the 90's.

The computational needs of these "Grand Challenge" problems are so great that a large massively parallel array of the fastest currently available processors will be required to meet them. In a distributed system, each compute node works in parallel with other nodes in the system and communicates with other nodes primarily by passing messages. The performance of many distributed applications depends on the efficiency of the underlying communication system. Workstation cluster multicomputers are increasingly being applied for solving scientific problems that require massive computing power [Begu93a]. Workstation clusters offers several advantages over other types of machines. These advantages include significant computing power, general availability, large cumulative resources and low cost. Many programming models have been developed for programming workstation clusters efficiently. Parallel Virtual

Machine (PVM) [Sund90a] is a popular message-passing model used to program these clusters. Many state-of-the art applications have been ported to this environment with encouraging results [Begu93a].

The major factors that limit the performance of these types of systems are the low interconnection network performance and the software overhead imposed by the "protocol stacks" which must be traversed in order to send a message to or receive a message from another machine in the cluster [Matts93a]. These factors are more visible in global program operations because these operations make very heavy use of network resources. The operations most likely to benefit from hardware acceleration include global operations such as barriers, broadcast/multicast, global reduction, scatter/gather, parallel prefix etc., because they are found to be very expensive on these systems. Careful analysis of previous attempts at hardware assist for global operations showed that all of them had serious limitations. However, Hall [Hall94a] has demonstrated that a secondary network with wide area topology and one or more centralized coordination processors (COPs) optimized for these global operations can improve the performance of global operations on a variety of multicomputer architectures. My hypothesis was that the efficiency of many PVM applications on workstation clusters can be significantly improved by utilizing the secondary network resources in a COP system for collective communication operations. To test my hypothesis, I used the following standard methodology.

First, I developed software to interface PVM with the COP system. The software includes a virtual memory-mapped secondary network interface driver and an easy to use function library. The function library makes it possible to use the COP system in place of PVM function calls for various parallel program operations. Second, as the COP hardware prototype does not exist, I deduced the timings for various global

operations in the COP function library based on the analysis of software overhead of the COP interface pseudo device driver and the hardware timings reported originally by Hall for various COP operations [Hall94a]. I measured timings for the same global operations with PVM, and compared them with the timings deduced for the COP system. Third, I obtained and instrumented a complex, real-life molecular dynamics simulation PVM application (100K lines) and made measurements to evaluate the cost of global operations on overall execution time.

To show the various programming models available for cluster multicomputers, Chapter II provides a brief overview of several popular programming models. Chapter III provides justification for choosing the COP system to work with the PVM message passing model, and provides an overview of PVM. The collective communication primitives in pvm are discussed followed by a preliminary analysis of some of their performance limiting factors. Chapter IV is a detailed description of the COP system architecture and operations. Included are discussions on the cop network architecture, compute node to cop interface, and the cop itself. A discussion of how the target global operations are performed on the COP system is also provided in this chapter.

In Chapter V, I first describe the cop software design followed by a detailed section on implementation of global operation primitives. This section also describes how the system works with pvm and how easily existing pvm applications can be ported to use the cop hardware features. In Chapter VI, I derive expressions for global operation timings with the COP system, and provide detailed analysis of the measurements done on pvm global operations. These measurements are then compared with the performance data derived for a COP-PVM combined system. The analysis of a sample molecular dynamics simulation application is provided to show the common communication patterns that can benefit from COP system, and their overall predicted improvement in

performance by using a COP system.

Having thoroughly demonstrated the benefits of the COP hardware and software design, I then in Chapter VII compare and contrast other attempts to improve the efficiency of global operations on workstation clusters. Finally in Chapter VIII, I provide some suggestions to further improve the performance of the COP software system based on experience gained while working on this project and give my conclusions.

CHAPTER II

PROGRAMMING MODELS FOR CLUSTER MULTICOMPUTERS

Introduction to Cluster Multicomputers

The purpose of this chapter is to describe the advantages of workstation clusters and to describe various programming models used to program them efficiently.

Considerable research effort in the recent past has been directed towards using clusters of workstations, loosely coupled by high speed networks, for distributed computing. Many state-of-the-art applications have been ported to this environment with encouraging results. For example, execution speeds for some molecular dynamics simulations, an application with a high volume of communication, using IBM RS/6000 workstations averaged only 30 percent slower than an iPSC/860 hypercube with a comparable number of processors [Begu93a]. Applications like propagation of seismic waves [Ewing93a], molecular dynamics simulations [Heller91a], logic simulations, etc. have been successfully run on workstation clusters.

The advantages of using clusters of networked workstations as multicomputers are:

1. The computing power offered by present day workstation class machines has improved significantly such that they can easily deliver required performance for high speed computing applications.

2. The workstation class machines are mostly general purpose machines which can perform many different functions and because of that are advantageous from a cost

standpoint.

3. The cumulative resources like memory, disk space, compute power, I/O capability of the workstation cluster can be very large.

4. Above all, clustering these machines allow the harvesting of otherwise unused clock cycles for productive use.

The major performance limiting factors for this class of machines are the low performance of the interconnection network and the software overhead of traversing the communication protocol stacks in order to send or receive messages [Matts93a]. The Local area networks (LANs) used for interconnecting these workstation clusters commonly use Ethernet networks that have a maximum bandwidth of only 10 Mbits/sec. In addition to this relatively low hardware bandwidth, a further limitation of ethernet is its common bus topology which requires workstations to compete for network access. As the number of nodes on the network and/or the amount of message traffic increases, the effective bandwidth available to the application decreases. With newer generation networking technologies like switched Ethernet and ATM, the performance of the interconnection network is significantly improved, but they still impose the software overhead of traversing the protocol stacks in order to send or receive messages.

The popularity of cluster based distributed computing can be easily understood from the large number of research groups and prototype systems currently in this field. Many programming models have been developed for programming workstation clusters efficiently. As my work is focused on these cluster based machines, a brief survey of the various programming models commonly used to program workstation cluster MIMD multicomputers is given next.

A Brief Survey of Multicomputer Programming Models

Message Passing

Message Passing is a programming model, generated by distributed-memory machines but applicable to others, for example multiprogrammed single-processors. Almost all other programming models have message passing in their lowest level. A program is divided into components or subprograms, which may run on different nodes of the machine. Nodes may all run copies of the same program (SPMD) or they may run completely different programs (MPMD). The nodes communicate with one another by explicitly sending and receiving messages, which are arrays of data copied from one node to another. Some of the popular models of this type includes p4 [Butl92a] from Argonne National Laboratory, Express [Flow91a] developed by ParaSoft, Inc., TCGMSG [Harr91a] maintained at Pacific Northwest Laboratory, Parallel Virtual Machine (PVM) [Manch94a] from Oak Ridge National Laboratory etc. There are hundreds of models in this class. The Message Passing Interface (MPI) [For93a] forum is an effort to collect the knowledge gained in the last ten years of building message-passing systems into a single, standard programming interface.

Virtual Shared Memory

Virtual Shared Memory is a model that can be used to program either tightly-coupled or distributed-memory multiprocessors. Messages can be implemented in terms of shared memory or vice-versa, and which of these is the lower layer is a performance issue. The Shrimp [Blum94a] system uses a virtual memory-mapped network interface to give applications protected but direct access to network hardware, eliminating the overhead of system calls. This allows high-bandwidth, low-latency data exchange and can be used to support either message-passing or shared memory.

Distributed Objects

Distributed Objects is a programming system in which data structures or objects are shared across a set of processors, without specifying exactly how the sharing is done. These systems give the programmer the abstraction of distributed shared memory (DSM). DoPVM [Sund93a] is an object oriented distributed environment implemented on top of PVM (Version 3). It defines shared object classes, uses operator overloading to move data transparently between processes, and also provides process scheduling tools. There are a set of other systems currently under development that use shared objects for better performance in synchronization, load balancing and fault tolerance.

Parallel Languages

Parallel Languages for distributed systems try to automate the tasks of communication and synchronization in parallel applications. The challenge for parallel languages is to provide the programmer the necessary expressiveness to get his work done without specifying the inner details, and to build a compiler that can translate the language into something that runs efficiently. Clark [Clark92a] provides a detailed evaluation of some of the parallel languages in the context of molecular dynamics computations.

Other models for parallel programming include systems like Linda (Scientific Computing Associates) that uses tuple spaces for sharing data between processes, process control etc. These models can be used easily on shared and distributed memory machines and networks. Distributed operating systems provide another level of primitives for parallel programming. They provide a more complete environment than other models, like file system and memory management, control of peripheral devices etc. But as they are custom made for the hardware of the target machine, they are less

portable than other systems. The Amoeba [Mull90a] distributed operating system treats nodes on a network as a central pool of processors, and uses remote procedure calls (RPC) both at the kernel and application level. Mattson [Matts93a] provides a thorough evaluation of these various programming models and discuss the pit-falls and strengths of each of them.

Message passing was used as the underlying model in this work. Message passing works at a very low level compared to other programming models, and hence allows more flexibility in experimenting with innovative ideas. Also the COP architecture is easily adaptable to the message passing model.

The task of next chapter is to provide an overview of the PVM message passing system. This chapter provides the justification for using the COP system with the Parallel Virtual Machine (PVM) and provide a brief overview of PVM. Also, a preliminary analysis of the collective communication primitives of PVM is provided in the next chapter.

CHAPTER III

CHOOSING A PROGRAMMING MODEL FOR THE COP SYSTEM

The purpose of this chapter is to introduce the programming model followed for the COP system to work with workstation clusters. As described earlier in chapter II, there are a variety of programming models that have been developed for cluster based distributed memory MIMD machines. In this chapter I provide the justification for using the COP system with the Parallel Virtual Machine (PVM) programming model followed by a brief overview of PVM. I also discuss in detail the various parallel programming primitives offered by PVM and the reasons for low performance of some of the group functions.

Why Choose PVM?

The need to follow a popular programming model for the COP system was obvious even in the early stages of the cop software design. The primary motivation behind this is that a vast number of applications already have been developed successfully using this model. Also, as one of the design objectives of the COP system is to coexist with any existing distributed system, applications already developed for the original model would be easily portable to use the cop resources.

The Parallel Virtual Machine (PVM) message passing system fulfills many of the requirements sought from a programming model. These include:

1. PVM has more users than any other portable parallel programming environment

and has become the de facto standard for message passing environments.

2. PVM further distinguishes itself from other message passing systems by being specifically designed to handle heterogeneous networks of computers. Given the architecture independent design of the COP system, this will help to apply the cop model to a variety of architectures.

3. Many state-of-the-art applications which include molecular dynamics simulations, seismic wave studies, logic simulations, etc. already run on top of PVM.

4. PVM ranks high in qualitative comparisons in areas like support groups, ease of debugging etc [Matts93a].

5. PVM provides some superior features like efficient task management, dynamic task groups, flexibility in using connection oriented communication protocols etc.

6. Availability of the software package and the programmer's prior experience with using this system.

7. Support based on the experiences of a large and accessible user base. When a problem is encountered - either with PVM or with expression of some algorithm in PVM, the chances are good that you will be able to find someone who has already encountered and solved a similar problem.

8. Simplicity in cop software design to interface with PVM message passing system when compared to interfacing with any other model.

The overhead of PVM function calls affect the performance of the COP software system if it is layered on top of PVM. As the cop software module has to extract information on task management, virtual machine configuration, dynamic groups etc. from the underlying PVM layer, the overhead involved in this can affect the performance of cop functions. Due to these above mentioned advantages and disadvantages the following design decisions were made for the COP system software

design:

1. The cop software system should coexist with PVM, so as to take full advantage of the functionality provided by pvm which includes simple task management, central console to control the whole virtual machine, asynchronous notification of events, dynamic task groups etc.

2. The cop software utilities should depend on the information supplied by PVM functions as little as possible to reduce the overhead involved. It use effective "Software Caching" mechanisms to reuse the information provided by pvm functions in multiple contexts, thereby reducing the effective number of pvm function calls.

3. The application programmer should be given enough freedom to choose between a cop function, or an equivalent pvm function, or a hybrid of these two, to express any given algorithm. The application developer can make his choice by estimating the I/O requirements and the degree of communication/computation overlapping possible in the application algorithms.

For these reasons the cop programming interface and libraries are designed in a machine independent style. The compute node interface device driver can be easily tuned to any target architecture and is designed as a loadable driver. i.e, the cop driver can be added to or deleted from the running kernel dynamically, without rebooting the system.

These features of the COP system software provide enough portability for any existing or future pvm applications. Even though the task of choosing appropriate cop or pvm functions for a given problem decreases the ease of programming for application developers, this provides enough flexibility to investigate faster and efficient solutions on an experimental architecture such as the COP system. With the requirements for a message passing model for programming the COP system to work with workstation

clusters fresh in mind, I will provide a brief overview of the PVM message passing system.

PVM Message Passing System

The PVM (Parallel Virtual Machine) software package provides the software infrastructure for programming heterogeneous networks [Begu93a]. PVM provides mechanisms for configuring a virtual machine on a network, initializing processes on this network and communicating among these processes. It is a lightweight package intended for user installation. Nearly any UNIX or UNIX like machine can be used as a processor in a virtual machine as long as the user has an account on the machine and it is accessible over a network. The most important goals for version 3 release of the PVM package are fault tolerance, scalability, heterogeneity and portability.

PVM is able to withstand host and network failures. It doesn't automatically recover an application after a crash, but it provides polling and notification primitives to allow fault-tolerant applications to be built. The virtual machine is dynamically reconfigurable. This goes hand-in-hand with fault tolerance, because an application may need to acquire more resources in order to continue running, once a host has failed or crashed. In pvm task management is made as decentralized and localized as possible, so virtual machines should be able to scale to hundreds of tasks. To allow pvm to be highly portable, the use of operating system and language features such as multi-threaded processes and asynchronous I/O etc., that would be hard to retrofit if unavailable are not used. To easily understand the theory of operation of the pvm system, I will provide the overview of important concepts used in pvm, followed by a detailed section on parallel programming primitives offered by pvm with emphasis on collective communication operations.

Architecture Classes

PVM assigns an architecture name to each kind of machine / OS combination on which it runs. The reason behind this is to distinguish between machines that run different executables due to hardware or operating system differences. Many standard names are defined and others can be easily added. Some machines with incompatible executables use the same binary data representation. PVM takes advantage of this to avoid data conversion. Architecture names are mapped to data encoding numbers, and the encoding numbers are used to determine when it is necessary to do data conversion. A complete description of the various architecture classes supported by pvm is documented in "PVM 3 Users Guide and Reference Manual" [Geist94a].

PVM Daemon

The pvm daemon (pvmd) is an important entity which runs on each host of a virtual machine. Pvmd serves as a message router and controller. It provides a point of contact and fault detection. An idle pvmd occasionally checks that its peers are still running. Pvmds continue to run even if application programs crash, to aid in debugging. Pvmds owned by (running as) one user do not interact with those owned by others, in order to reduce security risk, and minimize the impact of one PVM user on another.

The first pvmd (started manually) is designated as the master, while the others that are started by the master pvmd are designated as slaves. During normal operation, all the pvmds are considered equal, but only the master can start new slaves and add them to the virtual machine configuration. Re-configuration requests originating on a slave host are forwarded to the master pvmd, and only the master pvmd can forcibly delete hosts from the virtual machine.

Programming Library

The programming library "libpvm" allows a task to interface with the pvmd and other peer tasks in the virtual system. It contains functions for packing/composing and unpacking messages, and functions to perform pvm "syscalls" to send service requests to the pvmd. It is made as small and simple as possible. Since it shares an address space with unknown, possibly buggy, code, it can be broken or subverted. Minimal sanity checking of parameters is performed, leaving further authentication to the pvmd. The top-level of the libpvm library, including most of the programming interface functions, is written in a machine-independent style. The bottom level is kept separate and can be modified or replaced with a new machine-specific file while porting pvm to a new environment.

Task Identifiers

PVM uses a "Task Identifier" (tid) to address pvmds, tasks, and groups of tasks within a virtual machine. The tid contains four fields as shown in Figure 1. Since the tid is used heavily, it is made to fit into the largest integer data type (32 bits) available on a wide range of machines. Later I will show how the COP programming library redefines the pvm tids to manage the cop channel resources.

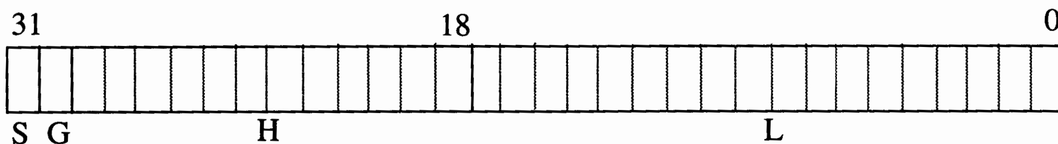


Figure 1. PVM Generic Task Identifier (tid)

The fields S, G and H have global meaning. Each pvmd of a virtual machine interprets them in the same way. The H field contains a host number relative to the

virtual machine. As it starts up, each pvmd is configured with a unique host number and therefore owns a part of the tid address space. The maximum number of hosts in a virtual machine is limited to 2^H -1 (4095). This is the same number of hosts that can be supported by a two-level COP system as described in later sections.

The mapping between host numbers and hosts is known to each pvmd, synchronized by a global host table. Host number zero is used to refer to the local pvmd. The S bit is used to address pvmds, with the H field set to the host number and the L field cleared. The G bit is set to form multicast addresses (GIDs), which refer to groups of tasks.

Each pvmd is allowed to assign its own format to the L field (with the H field set to its own host number), except that all bits cleared is reserved to mean the pvmd itself. The L field is 18 bits wide, there by allowing up to 2^{18} - 1 tasks to coexist on each host. In the generic UNIX port, L values are assigned by a counter, and the pvmd maintains a map between L values and UNIX process ids. In multiprocessor ports the L field is subdivided as shown in Figure 2. The P field specifies a machine partition, (physical group of processors), sometimes called a process type or job. The node number (N) determines a processor in a partition, and the W bit indicates whether a task runs on a compute node or host processor (service node).

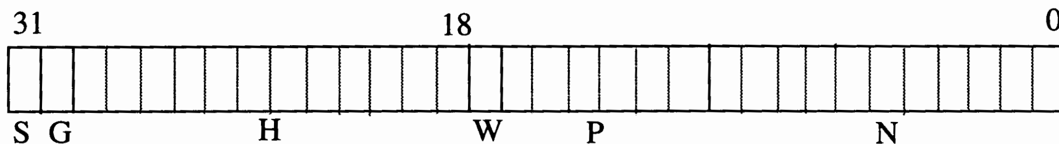


Figure 2. PVM Multiprocessor Task ID

The design of the tid enables the implementation to meet the design goals. Tasks can be assigned tids by their local pvmds without off-host communication. Messages can

be routed from anywhere in a virtual machine to anywhere else, due to hierarchical naming. Portability is enhanced because the L field can be redefined. When sending a message, a task on a multiprocessor node can compare its own tid with destination tid to determine whether to use native communication or send to the pvmd for routing. Finally, space is reserved for error codes. When a function can return a vector of tids mixed with error codes, it is useful if the error codes don't correspond to legal tids. The tid space is divided as shown in Figure 3.

Tids are intended to be opaque to the application and the programmer should not predict their values or modify them without using functions supplied in the programming library. More symbolic naming can be obtained by using a name server library layered on top of the raw PVM calls, if the convenience is deemed worth the cost of name lookup.

Use	S	G	H	L
Task identifier	0	0	$1...H_{max}$	$1...L_{max}$
Pvmd identifier	1	0	$1...H_{max}$	0
Local pvmd (from task)	1	0	0	0
Multicast address	0	1	$1...H_{max}$	$0...L_{max}$
Error code	1	1	small neg. number	

Figure 3. PVM tid Sub-Space Allocations

Message Model

PVM daemon and tasks can compose and send messages of arbitrary lengths containing data. The data can be converted using Sun Microsystem's "External Data

Representation Standard" (XDR) when passing between hosts with incompatible data formats. Messages are tagged at send time with a user defined integer code called message-tags, and can be selected while receiving by source address or message-tag.

The sender of a message does not wait for an acknowledgment from the receiver, but continues as soon as the message has been handed to the network and the message buffer can be safely deleted or reused. Messages are buffered at the receiving end until received. PVM reliably delivers messages, provided the destination exists. Message order from each sender to each receiver in the system is preserved. If one entity sends several messages to another, they will be received in the same order. Both blocking and non-blocking receive primitives are provided, so task can wait for a message without consuming processor time by polling for it. A receive with time-out is also provided, which returns after a specified time if no message has arrived.

Asynchronous Notification

PVM provides notification messages as a means to implement fault recovery in an application. A task can request that the system send a message on events like a task exiting or crashing, a host getting deleted or crashing, or new hosts being added to the virtual machine. Notify requests are stored in the pvmds attached to the objects they monitor. Requests for remote events are kept on both hosts. The remote pvmd sends the message if the event occurs, while the local daemon sends the message if the remote host goes down.

Protocols

PVM communication is based on TCP and UDP. While other, more appropriate protocols exist, they are not as generally available as TCP and UDP. One drawback of TCP is that it cannot take full advantage of the performance of modern high speed

networks due to a window size limit of 64kB. Experimental ports of pvm using other networking technologies and protocols like the ATM adaptation layer (AAL5) [Chang95a] are being developed to solve these problems.

PVM daemons communicate with one another through UDP sockets. UDP is an unreliable delivery service which can lose, duplicate or reorder packets. So an acknowledgment and retry mechanism is used. UDP also limits packet length, so pvm fragments long messages. TCP is not used for pvmd-to-pvmd communications due to several reasons. First is scalability. In a virtual machine of N hosts, each pvmd must have connections to the other (N - 1) pvmds. If TCP is used each connection consumes a file descriptor in the pvmd, and some operating systems limit the number of open files to as few as 32. But a single UDP socket can communicate with any number of remote UDP sockets. The second factor is overhead. N pvmds require a total of N(N-1)/2 TCP connections, which would be expensive to set up. The PVM/UDP protocol can be initialized with no communication as it is a connectionless protocol. Third factor to choose UDP over TCP for pvmd-to-pvmd communications is fault tolerance. The communication system detects when foreign pvmds have crashed or exited, or the network has gone down. For these features time-outs need to be set in the protocol layer. The TCP keep-alive option offer similar features, but not all operating systems provide adequate control over the TCP parameters.

A task talks to its pvmd or to other tasks through TCP sockets. TCP is used because it delivers data reliably. UDP can loose packets even within a host. Unreliable delivery requires retry (using timers) at both ends, but tasks should not be interrupted while computing to perform I/O. This provides the reason for selecting TCP for task communications.

PVM Communication Facilities

PVM provides two types of communication modes, Normal mode and Direct mode. In the Normal mode, for a source task to communicate with a remote task, it must first communicate through a UNIX domain socket to its local pvmd daemon. The local pvmd daemon of the source task then communicates through a UDP socket to the remote pvmd. The remote pvmd then communicates locally to the destination task again through a UNIX domain socket. Thus two TCP connections and two UDP connections are required for bidirectional communications between any two communicating application processes.

In the direct mode, PVM sets up a direct TCP connection between the two communicating processes or tasks via the indirect mode mechanism. The detailed transmission facilities of the direct and normal modes are hidden from the end-users. The advantage of the direct mode is that it provides a more efficient communication path than the normal mode. The major reason for providing the normal mode, despite its lower performance, is because of the limited number of file descriptors some UNIX systems provide. Thus the drawback of the direct mode is its limited scalability and the latency in setting up the connections.

Sending a message is composed of three steps in pvm. First, a send buffer must be initialized by a call to `pvm_initsend()` or `pvm_mkbuf()`. Second, the message must be packed into this buffer using any number and combination of `pvm_pk*()` routines. Third, the completed message is sent to another process by calling the `pvm_send()` routine or multicast with `pvm_mcast()` routine. (`pvm_bcast()` can be used for broadcasting). PVM also supplies the routine, `pvm_psend()` which

combines the three steps into a single call. This allows for the possibility of faster internal implementations, particularly by MPP vendors. `pvm_psend` only packs and sends a contagious array of a single data type. A message is received by calling either a blocking or non-blocking receive routine and then unpacking each of the packed items from the receive buffer. The `pvm_upk*()` routines provide the unpacking features. The receive routines can be set to accept any message, or any message from a specified source, or any message with a specified message tag, or only messages with a given message tag from a given source. PVM also supplies the routine, `pvm_precv()`, which combines a blocking receive and unpack call. Like pvm_psend, pvm_precv is restricted to a contiguous array of a single data type.

The encoding options provided by PVM include PvmDataDefault, PvmDataRaw and PvmDataInPlace. For PvmDataDefault, XDR encoding is used because pvm cannot know if the user is going to add a heterogeneous machine before this message is sent. If the user knows that the next message will only be sent to a machine that understands the native format, then he can use PvmDataRaw encoding and save on encoding costs. If the PvmDataRaw encoding option is used, no encoding is done. Messages are sent in their original format. If the receiving process cannot read this format, it will return an error during unpacking. For the PvmDataInPlace scheme, as the name specifies the data is left in place. Buffers only contain sizes and pointers to the items to be sent. When pvm_send() is called the items are copied directly out of the user's memory. This option decreases the number of times the message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent. Another use of this option would be to call pack once and modify and send certain items (arrays) multiple times during an application. An example would be passing of boundary regions in a discretized PDE implementation.

PVM Collective Communication Primitives

PVM provides facilities to do collective communications that operate over a entire group of tasks. This includes primitives for common parallel program operations like barriers, global reductions, multicast/broadcast, scatter/gather etc. Before describing the actual pvm functions or routines to perform these operations, I will discuss the dynamic process group features in pvm.

Dynamic process groups are implemented on top of PVM3. In this implementation a process can belong to multiple groups, and groups can change dynamically at any time during a computation. Pvmd does not perform the group operations. This is handled by a special group server that is automatically started when the first group function is invoked. There are trade-offs between using static and dynamic groups, and future versions of pvm may include efficient collective communications between static group members. (Currently `pvm_staticgroup()` can be used to declare a group as static).

PVM group functions are designed to be very general and transparent to the user at some cost in efficiency. Any pvm task can join or leave any dynamic group at any time without having to inform any other task in the affected groups. Tasks can broadcast messages to groups of which they are not a member, and in general any pvm task may call any group functions at any time. The exceptions are pvm_lvgroup called by a task to leave a group, pvm_barrier called by a task to participate in a barrier construct and pvm_reduce called by a task to perform a global reduction. These functions by their inherent nature require the calling task to be member of the specified group.

The functions `pvm_joingroup()` allows a task to join a user named group. The first call to `pvm_joingroup` creates a group with the specified name, and puts the calling task in this group. It returns the instance number (inum) of the process in this

group. Instance numbers run from 0 to number of group members minus 1. The routine `pvm_lvgroup()` allows a task to leave a specified group. The task calling `pvm_lvgroup` to leave a specified group will still be member of all other groups it had joined. The routine `pvm_gettid()` returns the tid of the process with a given group name and instance number. pvm_gettid allows two tasks with no knowledge of each other to get each other's tid by simply joining a common group. `pvm_getinst()` returns the instance number of a specified tid in the specified group. `pvm_gsize()` returns the number of members in a specified group. As I show later, these pvm group management functions are used appropriately by the cop software library to manage the cop communication channels.

The barrier is a common parallel programming construct. In pvm a barrier is implemented as a function `pvm_barrier()` that blocks the process until the specified number of group members have called the barrier function. In general count should be the total number of members of the group, but it is required as a function argument because with dynamic process groups pvm cannot know how many group members are in a group at a given instant. It is an error for processes to call pvm_barrier with a group it is not a member of. It is also an error if the count argument across a given barrier call does not match. `pvm_bcast()` labels the current message with an integer identifier (message tag), and broadcasts the message to all tasks in the specified group except itself (if it is a member of the group). `pvm_reduce()` performs arithmetic operation across the group, for example, global sum or global max. The result of the global operations is returned only to the task that is specified as the root. If other members of the group require the reduced result the root task should ship it to them using `pvm_mcast()` or `pvm_bcast()` functions. The reduction operation is done element-wise on the input

data. PVM also supports using any user specified reduction function. Even though this feature makes global reductions very flexible, most of the real applications seem to use only the common predefined functions like global sum, product, max, min, logical operations etc.

Multicast

Libpvm function pvm_mcast(), sends a message to multiple destinations simultaneously. The current implementation only routes multicast messages through the pvmds. It uses a 1:N fanout to ensure that failure of a host doesn't cause the loss of any message (other than ones to that host). The packet routing layer of the pvmd cooperates with the libpvm to multicast a message.

To form a multicast address tid (GID), the G bit in the tid is set (Figure 2). The L field is assigned by a counter that is incremented for each multicast, so a new multicast address is used for each message, then recycled. To initiate a multicast, the task sends a message containing a list of recipient tids to its pvmd. The pvmd creates a multicast descriptor and gid. It sorts the addresses, removes bogus ones and duplicates and caches them. It sends a message to each destination pvmd (ones with destination tasks), with gid and destinations on that host. The gid is sent back to the task. Later the task sends the multicast message to the pvmd, addressed to the gid. As each packet arrives, the routing layer copies it to each local task and foreign pvmd. When a multicast packet arrives at a destination pvmd, it is copied to each destination task. Packet order is preserved, so the multicast address and data packets arrive in order at each destination. Due to this type of implementation the pvm multicast is dependent on the UDP based pvmd-to-pvmd communication schemes.

Finally, to use collective constructs in application programs, there is a separate

library libgpvm3.a that must be linked with user programs. Later I show that in a much similar way, the COP library (libcop.a) linked to application programs can use the cop resources for efficiently performing global operations.

Performance Limiting Factors in PVM

PVM has several performance limiting factors. Some of these exist due to the trade-offs made to provide enough generality to the application user, and due to the inherent heterogeneous nature in the design. The networks used during the development of the package are the ethernet based networks that has a maximum bandwidth of only 10 Mbits/sec. The primary communication protocol followed (TCP) cannot take full advantage of the performance of modern high speed networks due to its window size limits. Other major performance limiting factor is the overhead of traversing the protocol stacks to send/receive messages to/from other tasks. A typical example is the number of times an application data buffer is copied before it makes it to the application layer of the peer task. Apparently, most of the on-host latency results from copying of large numbers of data buffers [Sten94a]. The low performance of the networking medium and communication protocols accounts for the low inter-host performance.

These performance limitations have more effect on the collective communication primitives because of their heavy use of the network resources. For example, the generic port of pvm assumes the underlying network cannot support multicast. As a result it cannot directly make use of the inherent multicast capabilities of networking technologies like ATM. Also the overhead due to additional communications needed with the group-server, to perform broadcasts limits the usability of the broadcast primitives in pvm. The scalability of the multicast feature is limited due to the 1:N direct fanout, because of the heavy contention caused by the acknowledgment messages. A

more detailed explanation of the performance limiting factors in pvm is given later when I provide the performance measurements and analysis of stand alone PVM and combined COP-PVM system for workstation clusters.

CHAPTER IV

COP SYSTEM ARCHITECTURE AND OPERATIONS

This Chapter describes the original COP system architecture, associated hardware modules, and their theory of operation. For accuracy, this chapter follows the work of Hall [Hall94a] closely.

Goals

The goals of the COP system are:

1. Be applicable to "Big Iron" multicomputers, workstation cluster multicomputers, and distributed shared memory systems.

2. Improve the efficiency of a wide variety of common parallel programming operations so as to better justify the cost of implementation.

3. Retrofit easily to the hardware of current generation machines so that it would not be necessary to wait for the next generation of machines to gain the benefits.

4. Require minimum modification of existing programming paradigms so as to not waste the massive efforts that have been invested in them.

5. Be compatible with MPI, PVM, and other current efforts to insulate programmers from low level system details.

6. Be compatible with advances such as thread scheduling and object oriented parallel programming that are likely to be included in future machines.

7. Have a high benefit-to-cost ratio.

As I discuss the COP architecture and the software in the following sections, I will describe how the COP hardware and software was designed to meet these goals.

Top Level View

Figure 4 shows the network topology for a single level COP system. As shown, each compute node in a group of 64 is connected to a coordination processor (COP) by an independent high-speed, half-duplex serial data link.

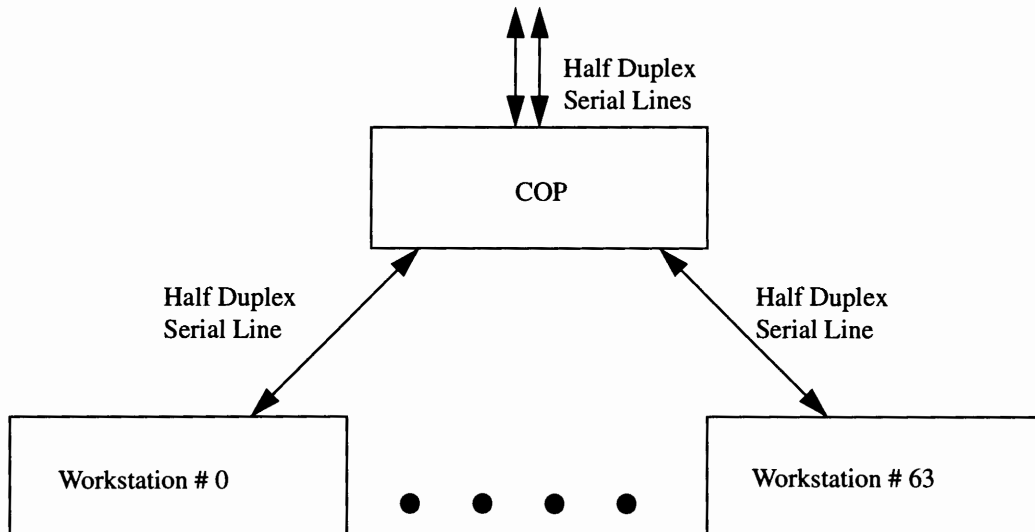


Figure 4. Single Level COP System Topology

Since the communication links between compute nodes and a cop controller are independent, all the compute nodes in a group can send data words or synchronization signals to their controller simultaneously. With these dedicated direct links, the source and destination are hardwired, so no complex message formatting is required. To send a word to its controller, a compute node simply does a write to its cop channel interface port. The dedicated signal lines also mean that no time is required to establish a connection with the controller, and there is no network contention. The result of these

capabilities is that each compute node can transmit a synchronization signal or data to its controller in a very short time. The independent communication links also mean that a cop controller can broadcast a synchronization signal or data to all the compute nodes in its group simultaneously.

Bit-serial data transmission was chosen to minimize the number of conductors in each link and for compatibility with relative inexpensive, non-multiplexed fiber-optic data transmission.

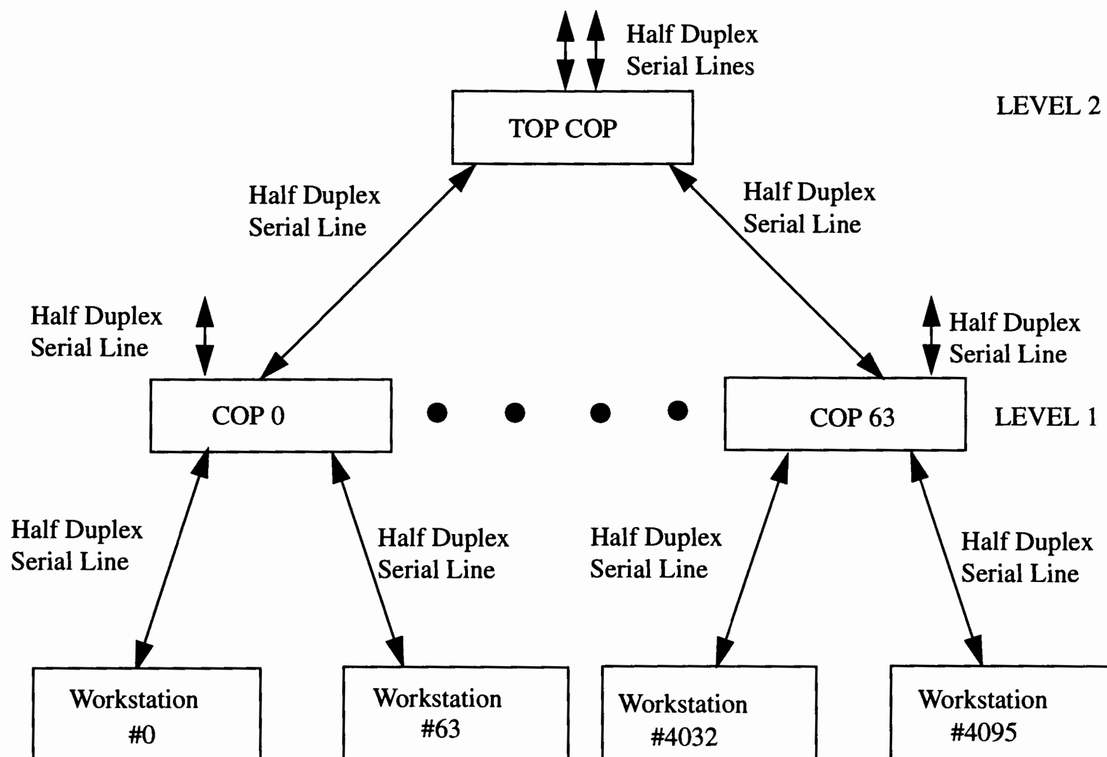


Figure 5. Two Level COP system Topology

The decision to assign 64 compute nodes to each controller was made partially so masks, bit-vectors, etc. are compatible with the data path widths of the latest compute node processors [Hall94a]. Assigning 64 compute nodes to each controller means that a

two level hierarchy of cop controllers can service up to 4096 compute nodes as shown in Figure 5. Keeping the number of levels low reduces the number of controllers and the number of connecting links for a given size machine. Keeping the number of levels low also reduces the overhead involved in traversing the tree for global operations in which a large number of compute nodes participate. To broadcast a value to all 4095 other nodes, for example compute node 0 send to controller 0, controller 0 sends the value to the Level 2 controller, the level 2 controller broadcasts the value to all the Level 1 controllers, and each of the level1 controllers broadcasts the value to its 64 compute nodes. The whole process requires only three passes through a cop level.

A very important point here is that the topology of the COP system is independent of the topology of the underlying machine. This means that the COP system is equally applicable to "Big Iron" multicomputers, cluster multicomputers, and distributed shared memory multiprocessors. Note that the COP system will be most efficient if a particular physical partition or "virtual machine" is created with all its compute nodes connected to one cop, but this is not required. In this case only the level 1 cop is used for all operations within the partition. For a two level system it is somewhat more efficient to assign the compute nodes of a partition to level 1 cops that are connected to adjacent input channels on the level 2 cop, but again, this is not required.

The software receive latency for a cop broadcast is usually very low because the receiving node is waiting for the control data word and immediately reads it from the cop network interface as soon as it arrives. In the case of a global sum operation, for example, a compute node would most likely write its data value and the appropriate opcode to its cop interface port, poll the cop interface port Data Ready strobe until the global sum arrives, and then immediately read the sum from the port.

Each cop controller has one extra serial channel in addition to those used to connect to 64 compute nodes or to lower level cops. One of the 64 channels is used to connect to a higher level COP if present. The extra serial channel can optionally be used to export performance or debugging data to external recording equipment.

A 64-bit integer ALU in each cop is used for performing global integer SUM, MIN, MAX, bitwise AND, bitwise OR, and bitwise EXOR operations. A floating point unit in cop controller is used for performing global floating point SUM, MIN and MAX. Each cop controller also contains a bank of very fast RAM which is used to hold intermediate results and shared write-able variables, to function as a global name space, and to accumulate performance data. A second, smaller bank of RAM holds broadcast/ multicast masks. A third, small bank of RAM holds bit vectors which identify the compute nodes participating in a barrier or other global operation. As a brief, introductory example of how a COP system works, we will use a global sum operation.

To start, each compute node writes a command consisting of a data value and the appropriate opcode to its cop network interface port. The cop network interface controller then automatically transmits the command to the cop controller. Arrival of a command at the network port of a controller sets a DATA_RDY flag for that input channel. The controller cycles through the input channel service requests on a round-robin basis. When the controller services a channel, it adds that channel's contribution to the intermediate result and resets the appropriate bit in the bit vector which identifies the compute nodes participating in the operation. When all the compute nodes have contributed, the controller broadcasts the sum simultaneously to all the participating compute nodes.

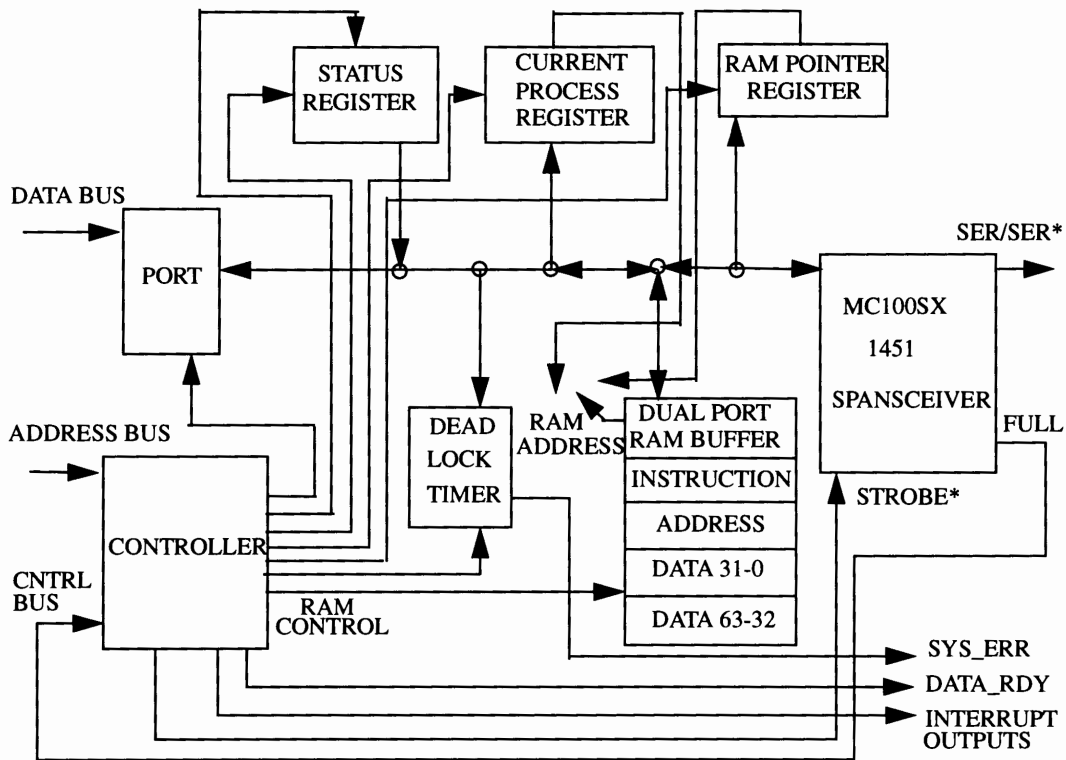
Operations that a COP system can directly perform include: synchronized read-modify-write access to global variables, barriers, integer and floating point global sum,

MIN, MAX, bitwise AND, OR, EXOR, one-to-all broadcast or multicast, and all-to-all broadcast or multicast.

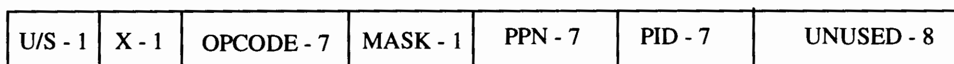
The Compute Node to COP Network Interface

Figure 6a shows a block diagram of the compute node to cop network interface. This interface is basically just a serial port with receive buffering, deadlock detection, and virtual machine protection capabilities. The compute node can interact with the interface on either a polled or an interrupt basis. To the compute node the interface appears simply as a 64-bit read/write port.

Each communication between a compute node and a cop consists of one to four 32-bit words. The first word in a cop command is always an instruction word with the format shown in figure 4b. Depending on the particular command, this instruction word is inserted by a user instruction, an operating system command, or hardware. The U/S bit in the instruction word indicates whether the command is a user level command or a supervisor level command which can only be invoked within an operating system call. The PPN in the instruction word is a number which identifies the physical partition to which the processor has been assigned. The X bit is used to indicate whether the physical partition extends beyond the local, level 1 cop. In other words, the X bit specifies whether a COP command should be passed on to a level 2 cop and applied to more than one level 1 cop. The PID in the command word includes context, group, and rank numbers which identify the process sending or receiving the command. This process identification mechanism was chosen for compatibility with the Message Passing Interface Standard [tennea]. The cop software interface section described in later chapters describes how this basic message format of the COP system was adapted to be used on workstation clusters.



(a)



(b)

Figure 6a,b. COP Channel Interface
(a) Compute Node to COP network interface circuitry
(b) Format for COP instruction word showing sub-fields.

The opcode bits in the instruction word specify the operation to be performed. The MASK bit in the instruction word is used to select one of two programmable masks in the cop mask RAM. Additional bits in the instruction word are reserved for future use.

If required, the second word in a command contains an address which is used to

access a global shared variable or partial result in the cop Data RAM. The third and fourth words in a command are used for 32-bit data values, 64-bit data values, or 64-bit mask values. The basic message format offered by the cop hardware is carefully broken down into software defined fields and is described in later sections on cop programming libraries.

If a command consists of more than one word, the additional words are transferred to the dual-port RAM buffer as they are written to the interface by the compute node. However, as soon as an instruction word is written to the buffer, the interface controller transfers it to the UART and the UART automatically sends the word on to the controller. Additional words of a command are transferred to the UART and sent to the cop controller in sequence. In the current design, the actual UART sections of both the compute node to cop network interface and the network to cop interface use Motorola MC100SX1451 Autobahn Spanceivers rather than custom modeled devices. These devices not only fill a need in the COP system, but also demonstrate that 200-400 Mbyte/sec serial transmission is possible with currently available commercial technology.

The spanceiver serializes each 32-bit word and transmits it over a positive emitter coupled logic (PECL) differential transmission line. High quality triaxial cable can be used to connect spanceivers that are within 10 feet of each other. For longer distance connections between spanceivers, the PECL signals can be converted to non-multiplexed optical signals and transmitted over relatively inexpensive multi-mode fiber optic cables. The deadlock timer on the interface can be used to trap to the operating system if a compute node sends a command to its controller and does not receive a reply within some pre-programmed time interval.

In summary, the compute node to cop network interface provides fast data transfer

with the protection features required for virtual machine operation. The relative simplicity and standard bus connections of this interface allow it to be implemented as a small daughter board which can be added to an existing compute node for performance enhancement or can be easily included in a new system design.

COP Architecture and Operations

The COP to COP Network Interface

Figure 7 shows block diagram of a cop controller. Each of the cop to network interfaces has a Motorola MC100SX1451 spanceiver, four 32 bit registers for buffering words received from a compute node, four 32 bit registers for buffering words to be sent to a compute node, and a mini-controller. The first register in each set is an instruction register. The second register in each set is used for data RAM addresses which identify global shared variables or partial results. The third and fourth registers in each set are used for 32-bit or 64-bit words and 64-bit masks.

When the first word arrives at the cop, the interface controller transfers the word to the first buffer register, and extracts two bits which specify the number of words in the command. As additional words are received, they are transferred to the appropriate buffer register.If the spanceiver detects an error while receiving a word, it will assert its error signal. If this signal is asserted, the interface controller aborts the receive operation and, as soon as the spanceiver is available, writes a "resend" command to the spanceiver for transmission back to the compute node. In response to a resend command, the compute node interface controller resends the entire command which is still in the RAM buffer on the compute node interface. After some number of unsuccessful attempts to receive a message from a compute node, the buffer controller sends an error word which causes a trap to the operating system on the compute node. A major advantage of this

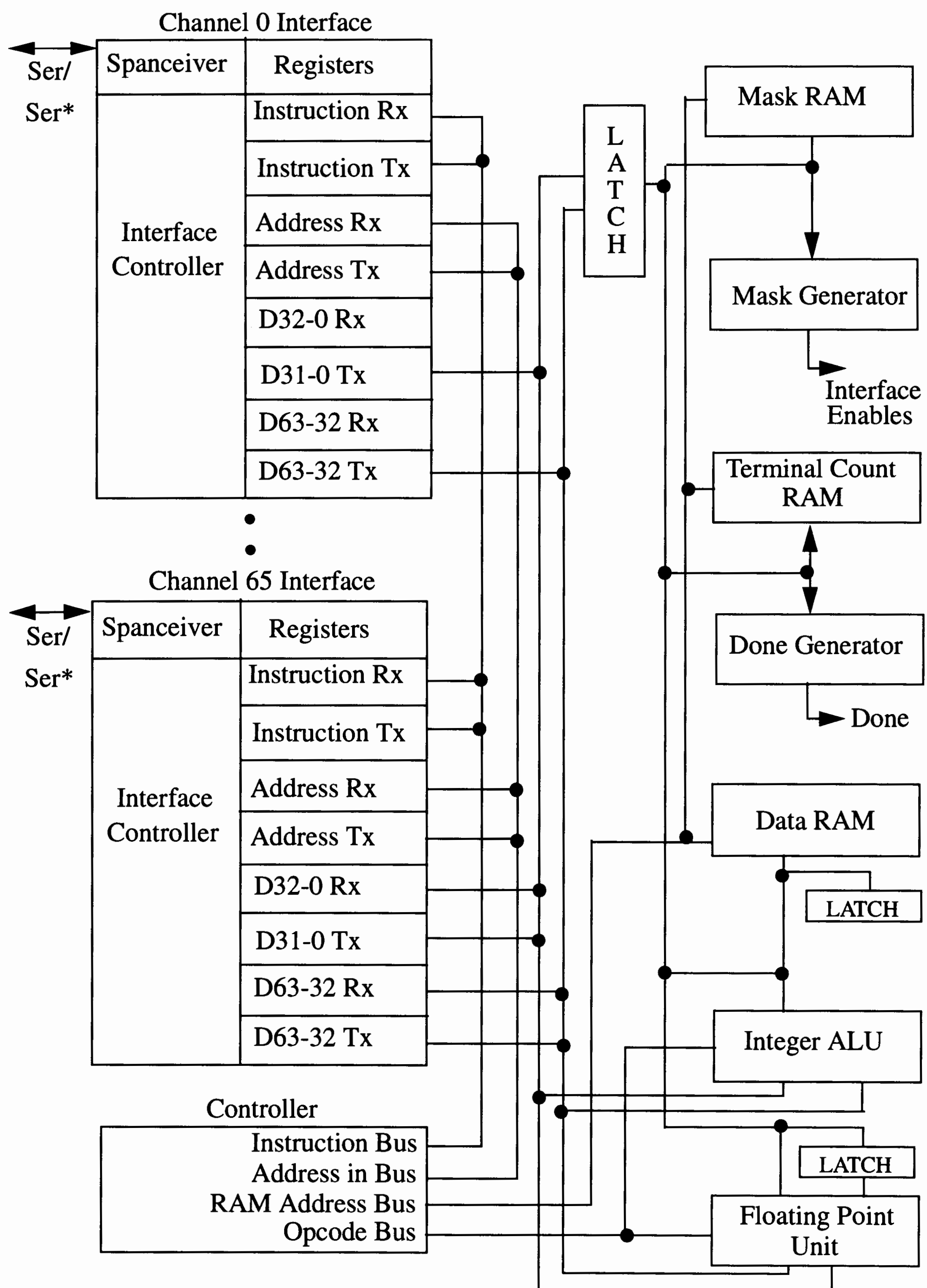


Figure 7. Block Diagram of a Coordination Processor

approach is that the compute node to controller link can cycle through multiple attempts to deliver a message without involving the controller. This reduces the cop controller overhead.

When all of the words of a command have been received without errors, the cop interface controller asserts a DATA_RDY signal. The cop controller polls the DATA_RDY signals of the 64 network interfaces on a round-robin basis and services ready interfaces in sequence. As soon as the cop controller reads a command from a ready interface, the interface controller writes an acknowledge word to the spanceiver for transmission back to the compute node. Arrival of this acknowledge word at the compute node indicates that the receive registers on the cop end of the link are available. Requiring that the compute node interface waits for this acknowledge prevents overwriting the receive buffers on the cop and assures that a command is still available in the compute node interface RAM buffer for resending in case of an error.

To send a command to a compute node, the cop controller transfers the command, address and data components of the command in parallel to the four transmit buffer registers in the interface. The interface controller then transfers the buffered words to the spanceiver in sequence for transmission. If the compute node interface detects an error in received word, it will direct the cop interface controller to resend the command for a pre-programmed number of times.

COP network communication links are asynchronous. This means that no global clock is required and that cables do not have to be cut to specific lengths in order to synchronize transmitters and receivers. This makes it easier to use the COP system with cluster multicomputers.

Overview of COP Operations

As mentioned previously, a cop controller contains a 64-bit integer ALU, a double precision floating point unit, three banks of 64-bit wide, very fast RAM, and a hardwired controller. The data RAM can be used to hold shared write-able variables, hold intermediate computational results, function as a global name space, and accumulate performance data. The mask RAM holds programmable masks which are used to enable the desired output channels during broadcast and multicast operations. The terminal count RAM holds the bit vectors that are used to keep track of which compute nodes have participated in a global operation. In this section I will give a brief overview of the original cop hardware design. Detailed theory of operation of each major hardware block is provided by Hall [Hall94a]. Later in the software design sections I will describe how these cop hardware features are used in a workstation cluster environment.

As a first example of how a cop controller operates, suppose that one compute node needs to broadcast a data value to all or a subset of the other nodes in its partition. To do this the compute node sends the data word and the appropriate instruction word to its cop controller. When the cop controller reads the command, it will use the PPN, PID, and a couple of other bits in the instruction word as a pointer to the Mask RAM. The mask read from this RAM will enable the transmit buffers of the channels that are to receive the broadcast data word. After the transmit buffers are enabled, the controller writes the data word to all of them simultaneously. The interface controllers then transmit the data word to all the destination compute nodes at the same time.

If the partition is larger than can be serviced by a single cop, then the local, Level 1 cop forwards the command on to the Level 2 cop. The level 2 cop uses a mask in its Mask RAM to broadcast the command to the appropriate level 1 cops and each of these then uses a mask from its Mask RAM to broadcast the data value to the desired nodes.

To enable the compute node to efficiently broadcast vectors larger than the eight-byte maximum for a single broadcast command, the cop controller has a channel lock capability. When a cop controller receives a lock command, the round-robin servicing of input channels is disabled so the controller continues servicing the locked channel until it receives an unlock command. This feature allows the node associated with the locked channel to pipeline back-to-back sequences of words through the cop. Using the PPN and PID to access a mask assures that the mask belongs to the currently executing process.

The COP system can also be used to implement a barrier very efficiently. As mentioned earlier, the Terminal Count (TC) RAM in the COP is used to hold bit vectors which identify the compute nodes participating in global operations such as barriers, reductions etc. Each bit in one of these vectors corresponds to one of the attached compute nodes. When each participating compute node reaches the barrier, it sends a single 32-bit command word to the cop. In response to this word the cop controller resets the corresponding bit in the barrier bit vector and determines if all the bits are reset. If all the bits in the barrier vector are reset, the 'done' signal is asserted. In response to the done signal, the cop controller writes a barrier exit command word to all the network interfaces which are enabled by the corresponding mask from the Mask RAM. The barrier exit command is thus broadcast to all the participating compute nodes simultaneously rather than sequentially.

Global sum and other similar global operations can also be performed very efficiently by a COP system. For this operation each compute nodes sends a contribution to its cop. The cop adds each contribution to a partial result stored in a Data RAM location. When all the values have been added, the controller uses a mask from the Mask RAM to broadcast the result to the participating compute nodes. For protection, the PPN

and PID in the instruction word are used as part of the address for the temporary result in the Data RAM and for the mask in the Mask RAM. Note that each intermediate result could be broadcast to all or to a subset of the participating nodes at the same time as it is written back to the Data RAM, if this were required by the particular algorithm.

Still another type of operation that a COP system can easily perform is global shared variable access. For simple read access, a compute node sends the appropriate command and a variable identifier (address) to its cop. The cop controller uses the variable identifier, the PPN, and the PID received from the compute node to address the desired location in its data RAM and sends the addressed data value back to the compute node. In a case where it is important that a compute node should very quickly read a series of values from the Data RAM or write a series of values, the channel lock feature can be invoked.

Read-Modify-Write access to the COP Data RAM is essentially the same, except that as the value read from the Data RAM is being copied to an interface transmit buffer, it is passed through the ALU, modified as specified in the command, and then written back to controller memory.

Table I shows the list of commands that the original version of the cop controller is programmed to implement. Note that with the COP design there is no conflict if, for example, one node sends a Data RAM read (RSRAM) command, while a global reduction is in progress. Assuming the controller services the interface with the global sum command first, the cop controller will simply add that interface's contribution to the intermediate sum in the data memory. If the global sum is now complete, the controller will broadcast it to all the nodes waiting for the sum. As mentioned earlier, an instruction is broadcast along with the sum to identify the sum for the receiving nodes. (The cop software design has more features like message tags to identify specific messages). If the

global sum was not complete, the cop controller will just go on servicing other channels in a round robin basis and eventually the channel that issued RSRAM message will get serviced.

The COP system provides reliable communication between the compute nodes and the COP controller over the COP channels. As a result, the key issue is that, unlike the PVM model, application tasks are not required to traverse the protocol stacks to access the COP channels. Figure 8 below shows the protocol stacks in PVM (normal route) and the COP. A final point here is that the COP system does not prevent use of the standard features on a given system. It simply provides a more efficient mechanism for global operations.

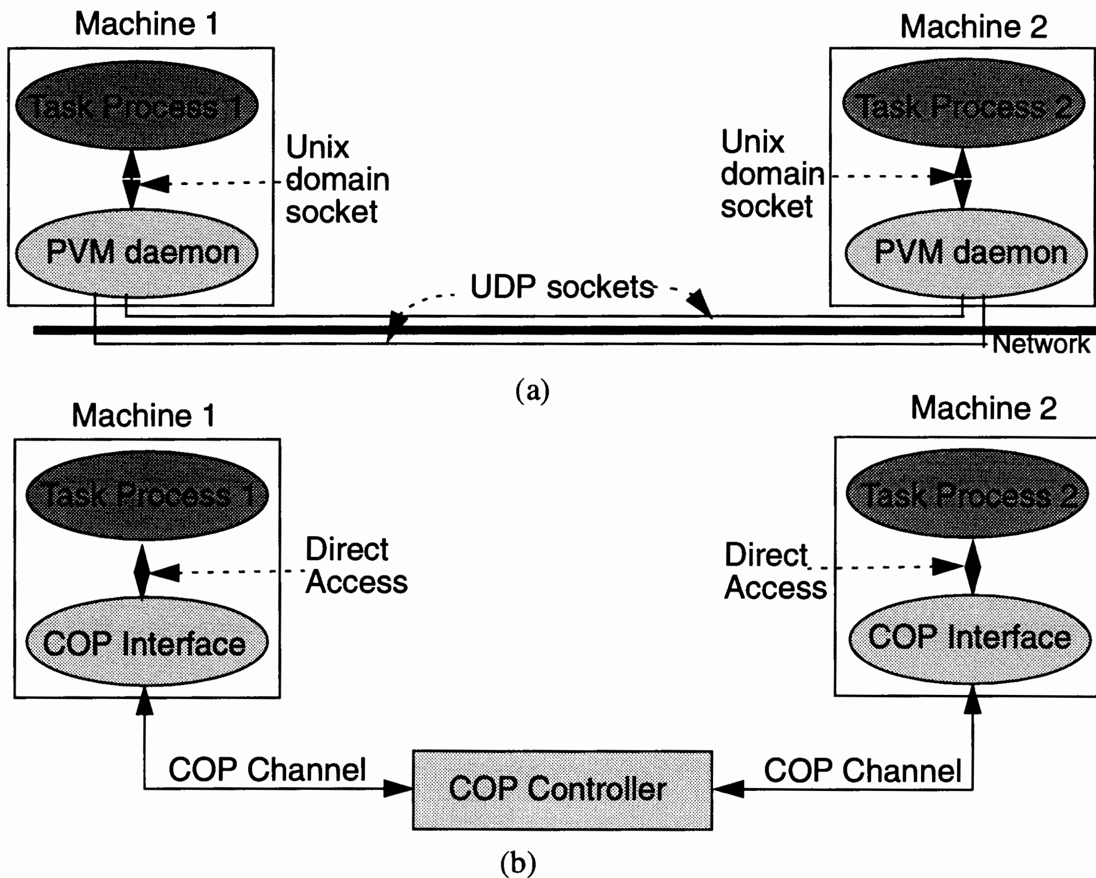


Figure 8. (a) Protocol stacks for PVM (normal route)
(b) Protocol stacks for COP system

TABLE I

COP COMMANDS AND OPCODES

Command	Opcode	Description
RSRAM32	1011100	Read 32-bit data from Data RAM
WSRAM32	1011101	Write 32-bit data to Data RAM
RMW32	1011110	Read-Modify-Write 32-bit data
SUMI32	1010000	32-bit integer global sum
MAXI32	1010001	32-bit integer global Maximum
MINI32	1010010	32-bit integer global Minimum
OR32	1011000	32-bit global logical OR
AND32	1011001	32-bit global logical AND
EXOR32	1011010	32-bit global logical EXOR
RSRAM64	1111100	Read 64-bit data from Data RAM
WSRAM64	1111101	Write 64-bit data to Data RAM
RMW64	1111110	Read-Modify-Write 64-bit data
SUMI64	1110000	64-bit integer global sum
SUMF	1110100	64-bit floating point global sum
MAXI64	1110001	64-bit integer global Maximum
MAXF	1110101	64-bit floating point global Maximum
MINI64	1110010	64-bit integer global Minimum
MINF	1110110	64-bit floating point global Minimum
OR64	1111000	64-bit global logical OR
AND64	1111001	64-bit global logical AND
EXOR64	1111010	64-bit global logical EXOR
BENTRY	0000001	Barrier entry and broadcast
LOCK	0000010	Lock a channel
UNLOCK	0000011	Unlock a locked channel
XMIT	0000100	Retransmit a message
ACK	0001000	Acknowledgment for a message
BCAST32	0011100	Broadcast 32-bit word
SETBM	0111110	Set Broadcast Mask
SETTC	0111111	Set TC Bit vector
BCAST64	0111100	Broadcast 64-bit word

CHAPTER V

DESIGN AND IMPLEMENTATION OF THE COP SOFTWARE

In Chapter IV, we discussed the basic programming model followed for the COP system, and a brief overview of the cop software design goals. As mentioned originally, the cop software subsystem is designed to work with the PVM message passing system. Chapter III showed that pvm provides the necessary software utilities to program heterogeneous networks. With the cop hardware features and pvm model fresh in mind, in this chapter I will provide the cop software subsystem design and implementation. Since the size of most of the present day workstation cluster multiprocessor systems is well below the capacity of a two level COP system (4096 compute nodes), the cop software design discussed here concentrates on a single level COP system. A single level COP system can support a maximum cluster size of 64. Features are provided whereever possible to easily extend the software design for a multilevel COP system. Also, even though the underlying pvm model allows multiple concurrent users, only one user can access the cop resources on a compute node at a time. This restriction was made to make the design simple, but it can be removed later and the software can be modified to add concurrent simultaneous users.

COP Message Format

As mentioned originally, the basic message format between a compute node and a cop controller consists of one to four 32-bit words. The first word is the instruction word and specifies the basic operation requested by the compute node. The second word is

specified as the address word and is used to address variables in controller memory owned by any task using the cop resources. The third and the fourth words are data words, and are used to carry either application data or mask bit vectors. Figure 6b in Chapter IV showed a basic partitioning of the instruction word as originally proposed. For the work here, the entire message structure is re-partitioned to better use the cop resources along with the pvm model. The cop hardware design can be easily modified to support the current specifications. Figure 9 shows the message format supported by the COP software. The number of words belonging to a single logical message varies from one to four depending on the type of operation specified in the instruction word.

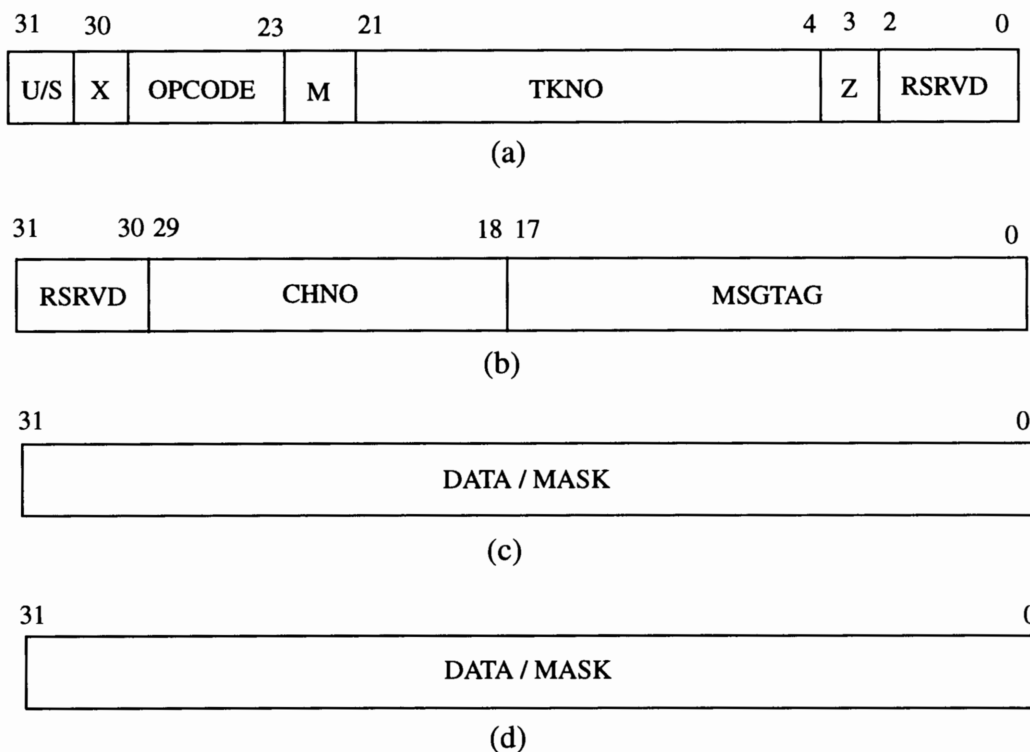


Figure 9. COP Message Format (a) Instruction Word (b) Address Word (c) Data1 Word (d) Data2 Word

The U/S bit in the instruction word differentiates an application message from an

operating system generated message. The X bit specifies whether the message should be restricted to the local controller or whether it should be forwarded to a higher level cop controller. The X bit needs to be used only for a multi level COP system, which has more than 64 compute nodes. The OPCODE specifies the operation requested by the compute node sending this message. The opcodes supported by the COP system is illustrated in Table I included in chapter IV. The M bit is used to select one of the two programmable masks on the cop controller. The Z bit is set if the compute node sending this message wishes to access the address space of other channels either on the same controller or on a different controller. If the other channel belongs to a different controller the X bit discussed before will be set along with the Z bit. If the Z bit is off in a instruction word, the cop controller automatically use the current value of the channel poll counter as the channel number (CHNO). This is appropriate as the poll counter always indicates the channel currently serviced. The cop controller address spaces are described later in this chapter.

The task number (TKNO) field represents a specific task using the controller resources. The channel number (CHNO) field in the address field represents a specific cop channel. As each channel is attached to a compute node, the 'CHNO' field indirectly points to the host connected to the respective channel. As a result of this partitioning of message space, a (CHNO, TKNO) tupule can easily identify any task that use the cop controller resources in a cluster. In the next section I will show how the (CHNO,TKNO) tupule replaces the task identifiers (tids) used in the pvm model to identify tasks. The MSGTAG field in the address word serves the same function as message-tags for pvm messages. They help to de-multiplex one received message from another. Additionally, message-tags in a COP system provide the offsets from the base address of variables in COP Data RAM. The data1 and data2 words in the cop message are used to carry either

32/64 bit data or the 64 bit masks. The 'RSRVD' fields are reserved for future use.

Channel Identifiers

PVM uses 32 bit Task Identifiers (tids) to address pvmds, tasks, and groups of tasks within a virtual machine. Figure 1 in Chapter III showed a generic pvm task identifier. Out of the 32 bits, 12 bits (H field) are used as host identifiers. The H field identifies the location of a task in the virtual machine; i.e., the host on which a given task resides. The L field in the tid has eighteen bits and is used to identify a given process local to a host. The H and L fields together can identify any process in the workstation cluster. This hierarchical naming allows pvm to assign tids to local tasks by their local pvmds without costly off-host communication. This prevents a bottleneck at an ID server.

The cop software uses Channel Identifiers (cids) to identify any task using the cop controller resources. The cid consists of three fields: a controller number (CNTNO) field, a channel number (CHNO) field and a task number (TKNO) field. The controller number field addresses a particular cop controller. This is always zero for a single level cop, but ranges from 0 to 63 in a 2 level COP system. The CNTNO and CHNO fields together is analogous to the H field in pvm task identifiers. From the perspective of the cop controller, CHNO is a way of addressing the cop channels. As each host in the cluster has a single channel, channel number (CHNO) indirectly identifies a specific host in the cluster. The TKNO in cid is analogous to the L field in pvm task identifiers. Task numbers (TKNOs) are assigned local to a host and identifies a task within the host.

The cop software system creates a channel identifier (cid) for any given pvm task identifier. Two approaches are considered for performing the H field in tid to CHNO in cid mapping. First is a table lookup method to convert any given tid to a cid. The table

requires an initialization phase, and is updated whenever a task is spawned or completed. Each compute node keeps a copy of this table. The cop function library is designed in such a way that, if a new local task is spawned or completed it gets notified by the pvm notify function, updates its local table, and broadcasts the changes to all other compute nodes to provide consistency. As the cop channels cannot be used before the initialization of this table, pvm collective communication primitives are used to initialize the lookup table on all the compute nodes. But once the initialization is over, the cop channels are used to broadcast the changes due to tasks entering or leaving the system. For example, if there are N compute nodes, and one task per compute node, the table contains N entries for all the N tasks. The initialization is done via all-to-all broadcasting mechanisms, each compute node sending its contribution to all other N-1 compute nodes.

As the table-lookup procedure discussed above proved to be inefficient due to the use of slow pvm group functions for table initialization, a second method was devised to overcome this overhead. This method is very simple and makes use of the mechanism used to assign tids in pvm. PVM assigns host numbers using a simple counter policy. The host numbers are provided in the same order as hosts are added. Even though pvm provides features to add or delete a host dynamically, most applications use a static host cluster. Each channel in a cop controller is linked to a host in the cluster. As described in the previous chapters, the cop controller polls each channels using a poll counter on a round robin basis. In the current cop hardware design, the channel numbers cannot be configured automatically. Each channel has a channel number between 1 and 64, and is the value of the poll counter used to poll them (Value of zero is not used as the H field in pvm tids is used by pvm to identify the local pvmd). If hosts are added to the pvm system in the order of their channel numbers, there will be a one-to-one correspondence

between the channel numbers and the host numbers assigned by pvm. For a single level cop, as the maximum cluster size is only 64, the H field can be mapped directly to the CHNO field in the cid. The controller number CNTNO is always zero for a single level COP system. For a cluster size of 4096, we require 64 level 1 cop controllers and a level two cop controller. As the H field in tid is 12 bits long, the maximum cluster size supported by pvm is 4095 (A zero in H field identifies a local pvmd). In a 2 level COP system, a cop controller with CNTNO of zero is attached to hosts with host numbers ranging from 1 to 64, a controller with CNTNO of one will be attached to hosts with their H fields ranging from 65 to 128, and so on.

The only restriction on the direct H-to-CHNO mapping mechanism is that, pvm should add hosts to the system in the order of channel numbers. This is fair for almost all applications as most of them do not care about the order in which hosts are added to the cluster. As the L field in a tid is assigned locally to a host, it is used as the TKNO field in cid. As a result of this mapping, any task in the cluster that owns part of the tid space also owns a part of the cid space in the COP system. Similarly the L field in pvm tid is assigned locally to a host using a simple counter when tasks are spawned. Due to this simple local assignment of process numbers by pvm, the L field of the tid is used directly as the task number (TKNO) field in cid.

Either of the mapping schemes can be selected, depending on requirements. The first scheme is more general and does not impose any restrictions on the applications, but is inefficient. The second method is simple and requires no overhead of initialization, but restricts the order in which hosts can be added or deleted from the cluster. I will follow the direct host number to channel number mapping in my further discussions for sake of simplicity.

Layered Architecture of the COP-PVM System

PVM runs as a user process and uses the protocols built into the operating system of the compute nodes. A pvm daemon runs on each compute node and functions as a message router and controller for pvm messages. They also assist in additional functions like task management, host failure notification etc. Application programs that use pvm functions are linked to the 'libpvm.a' and 'libgpvm3.a' libraries.

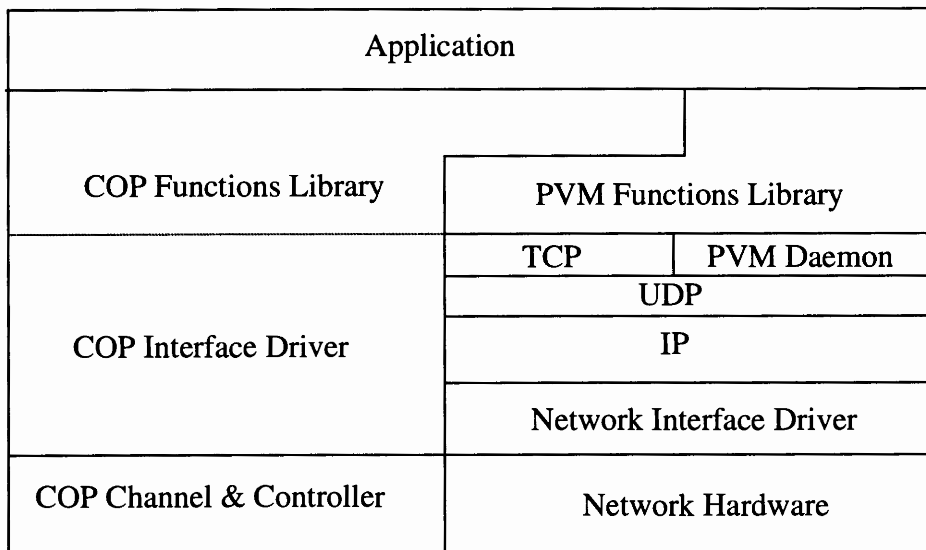


Figure 10. Combined COP-PVM Model Communication Resources

Figure 10 shows the layered architecture of the combined COP-PVM model's communication resources. The application programs can access the pvm resources directly using the original libpvm functions. They can access the cop resources using the functions in 'libcop.a' library. A detailed explanation of the libcop library is provided later in this chapter. As the cop functions make use of the task management and dynamic group features of pvm, part of the COP library is layered on top of pvm. This allows cop functions to extract information about tasks, dynamic groups etc. from pvm as shown in

Figure 10. As this dependency on pvm adds additional overhead, the number of pvm function calls is made as small as possible. Effective 'software caching' schemes allow the task information to be used in multiple contexts.

The pvm asynchronous event reporting functions are used by the cop library functions to determine events like host failure, task existence etc. As the latency of the cop channels is very low for small messages, the cop channels are considered for features for providing "heartbeat" messages between the compute nodes in future software versions. This will provide reliable and fast notification of events like host failures.

Compute Node to COP Channel Interface Registers

The compute node channel interface hardware provides a clean interface for accessing the cop channels. There are registers available on the compute node channel interface for control, status, data transmission and data reception functions. The control and status features are provided by a Control & Status Register (CSR) on the compute node interface. The control functions are valid on a write to this register and status functions are valid while reading the register. Figure 11 shows the structure of the CSR register.

The RST control bit is used to reset the cop channel interface by the compute node. When the interface is powered up or a reset is applied using the RST control bit, the interface hardware performs a self test and reports the status using the status bits in the CSR register. If the self test has passed, the RDY status bit will be set by the hardware. The cop device driver use this RDY signal to check whether the interface is ready for use or not. The enable-interrupt (EIN) control bit is used to enable or disable receiver interrupts. The interface issues an interrupt to the compute node whenever a

message arrives at the interface, if it is configured in the interrupt model.

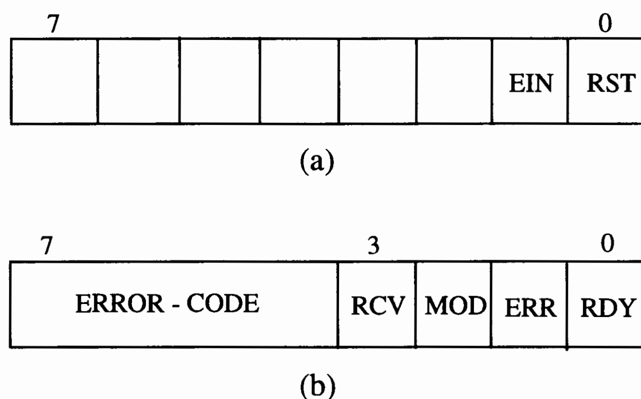


Figure 11. Control and Status Register (a) Control Bits (b) Status Bits

The RDY status bit is used to verify the status of a message transmitted over the channel. When the interface receives an acknowledgment from the cop controller for the current message, the RDY bit is automatically set. The cop driver software can check this bit to verify the status of the last transmission. If the acknowledgment (ACK) for the message transmitted does not show up before the transmit timer goes off, the interface posts the error through the ERR status bit. The ERR bit is set to notify time-outs during transmission, checksum and parity errors while transmission/reception, or if the cop controller responded with a negative acknowledgment (NACK) for the current message transmitted. The ERR_CODE bits specify a predefined error code to distinguish between the various error conditions. The driver software checks the ERR and ERR_CODE bits to check the status of a transmission.

The MOD status bit is used to read back the current mode of the compute node interface to check whether interrupts are enabled or not. The RCV status bit is set automatically by the interface hardware whenever there is an unread message in the compute node interface. This bit is automatically reset by the interface controller when

all the messages are read in by the compute node. This feature allows the driver to read in multiple messages upon single interrupt notification by the interface.

Other registers on the compute node interface include a DataTransmit register, DataReceive register and a Pid register. The Pid register allows the compute node interface to provide enough protection for messages arriving from the COP channel to various local tasks. Due to this mechanism, a message arriving at the compute node interface from the channel can be read only by the destination task. Provisions are provided in the interface hardware to tune the channel parameters like transmit time-out period, number of retransmissions etc.

COP Controller Address Spaces

As described in the cop hardware design in chapter IV, the COP controller has three independent memory banks, namely Data RAM, Terminal Count Mask RAM and Broadcast Mask RAM. As the name specifies the Data RAM is used to store user data, the broadcast Mask RAMs are used to store the multicast and broadcast masks which specify the channels that should receive the partial/final result of a collective parallel operation. The terminal count mask specifies the channels that participate in a collective operation.

Data RAM (DRAM)

The Data RAM of a cop controller is used to store the data portion of application messages. The data space is divided primarily into three different address spaces based on the type of cop operations possible on these address spaces. They are: (a) channel address spaces(CAS), (b) a global address space (GAS) and (c) a system address space (SAS). Figure 12 shows the sub-divisions of the Data RAM.

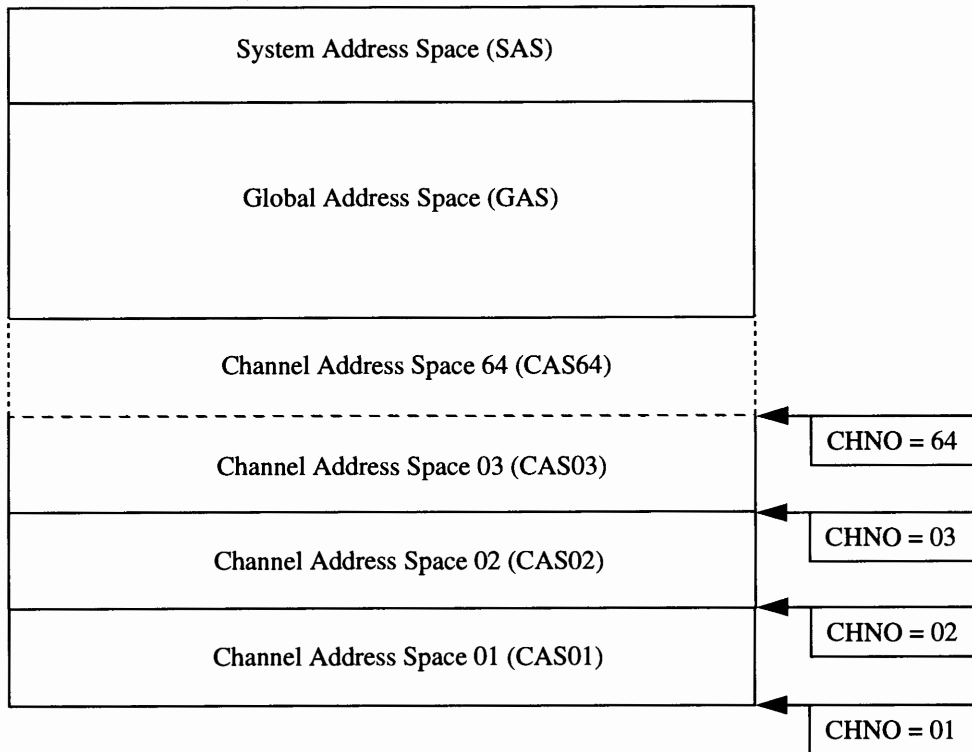


Figure 12. Address Space Partitions for Data RAM on COP Controller

Channel Address Space (CAS)

The Channel Address Space on the cop controller Data RAM is sub-divided primarily into 64 channel spaces, so that each channel owns a part of the total channel address space. They are identified as CAS1 to CAS64 for the 64 channels attached to a cop controller. Each CAS space is further divided into different task address spaces (TAS). As a result each task spawned on a host attached to the cop controller gets a chunk of memory on the controller. Figure 13 shows the task address space partitions within a channel address space. Presently the CAS and TAS address space locations and limits are statically allocated, with the maximum number of tasks per host using the cop resources restricted by available cop memory. As the task numbers are always locally

assigned within a host by pvm, the base of each task address space within a channel space can be easily derived from the TKNO in the instruction word. Better allocations of CAS and TAS address spaces, such as dynamically allocating them whenever a task is spawned and registered to use the cop resources can be designed, but is left out in the current design to make it simpler. In the current design each CAS base is derived by the controller from the corresponding channel number (CHNO) (either specified in message or the value of channel poll counter) and each TAS address space base is derived from the corresponding task number (TKNO) used in the cop message instruction word.

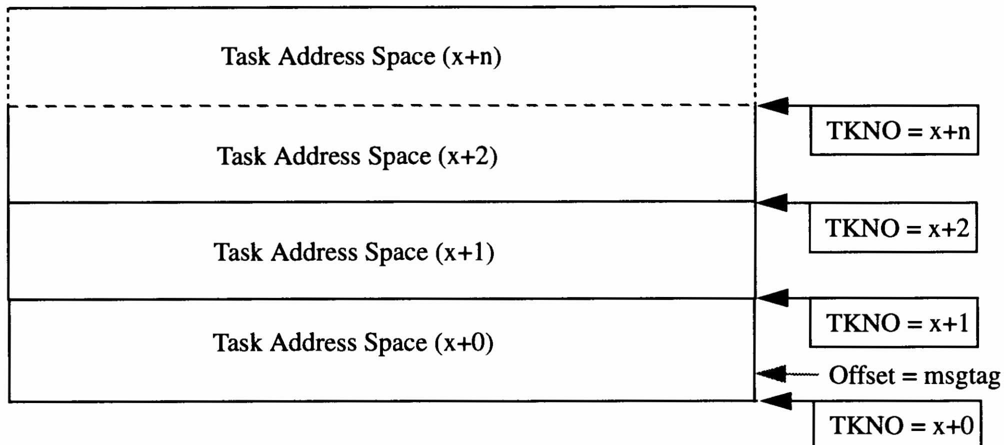


Figure 13. Task Address Space Partitions within a Channel Address Space

Each memory location in a task's address space is derived by adding an offset to the TKNO base. The message tags (MSGTAG) used by the application while calling the cop functions is directly used as offset into the TAS space. This memory model of the task address spaces is very much similar to the classical segmentation model. The limit checking mechanisms of the cop controller provides graceful error reporting if the tasks try to access unauthorized address spaces. It is the responsibility of the applications to use sensible message tags for their messages depending on controller memory

configuration.

Task Address Space as Protected Shared Memory

The task address spaces are configured in such a way that only tasks that own a TAS have write access to them. But all tasks that use the COP controller resources have read access to all task address spaces. As a result, the task address spaces work as a one directional way of passing information between tasks. This is much similar to the classical PIPES used in UNIX systems for inter process communication.

The task address spaces allows participating tasks to use the COP controller memory as some kind of protected shared memory mechanism. This is specially useful, if a variable is occasionally modified by the owner task, but is very frequently read by other participating tasks. Storing these shared variables in the COP controller allows efficient and faster access to them by other tasks. Also this sharing does not use any processing time of the task that actually owns the address space. Later I will show how this feature is implemented by the cop software system using the read (RSRAM) and write (WSRAM) RAM operations.

Global Address Space (GAS)

The global address space is primarily an address space to perform parallel operations between participating tasks that involve collective communications. PVM allows tasks to join groups, so as to easily perform group operations. The original idea behind the global address space, was that it can be dynamically partitioned into group address spaces. Only tasks belonging to the specific group can access its group address space. To reduce the complexity of the software and hardware design, this was not attempted in the current version of the COP system. If multiple global operations are done simultaneously by one or more groups of tasks, the application should take care not

to use the same sub-sections in global address space. This can be easily achieved by making sure that the message tags used along with the group operations do not overlap.

All the global operation data manipulations are performed in the global address space. This implementation allows tasks participating in a group operation to read the partial results of the operation, simply by reading the appropriate locations in the global address space.

The system address space contains initialized variables like the controller number (CNTNO) of the cop controller if it is multilevel COP system. Also this space was allocated so as to implement any messaging protocols between level-1 and level-2 cop controllers in a multi-level COP system in future versions.

Terminal Count Mask RAM (TCMASK RAM)

As described in chapter IV, this is a 64 bit wide memory and holds the terminal-count mask. Each bit in the TC mask corresponds to a channel attached to the cop controller. The terminal-count mask specifies which all channels will participate in global operation. For each location in the global address space in the Data RAM, there is a corresponding address in the TC RAM. For example, a TC entry with all bits set at offset N in the TC RAM indicates that, all the 64 channels of the cop controller have to participate in the global operation performed on the variable at address offset N in the global address space of the Data RAM. As message tags are directly used as offsets in the global address space, a global operation performed by the participating tasks with message tag N requires the contribution from all the 64 cop channels. Later I will show how tasks local to a channel/host performs a partial global operation and then one of these tasks sends the partial result to the cop controller as the contribution from the channel they represent. As mentioned in Table I in Chapter IV, the TC mask can be set

only using the SETTC operation. A transmit mask entry can be read from the TC Mask RAM using a RSRAM64 message with the mask bit (M) set to indicate the mask type. This allows any participating task to query the cop controller to find out which all channels already contributed for a current global operation.

Broadcast Mask RAM (BCAST RAM)

The broadcast Mask RAM is 64 bit wide memory and holds the broadcast mask if required for global operations. The broadcast mask specifies which all channels should receive the result of a global operation. Similar to the TC Mask RAM, for each location in global address space in Data RAM, there is a corresponding address in the BCAST Mask RAM. For example a BCAST mask entry with all bits set at offset N in the BCAST Mask RAM indicates that, all the 64 channels of the cop controller will receive the result of the global operation performed on the variable at address offset N in the global address space of Data RAM. In other words, this implies that all the 64 channels will receive the result of the global operation with message tag N. The broadcast mask can only be setup using the SETBM operation of the cop controller. Similar to the case of TC masks, a broadcast mask entry in the BCAST Mask RAM can be read out by issuing an appropriate RSRAM64 message to the cop controller.

Additional COP Hardware Features

During the cop software development, a set of additional hardware features were devised for the COP system. All these features increase the programmability of the cop controller and allow the cop software system to reduce the overhead when used along with the pvm message passing system. These features are described in detail in the following paragraphs.

1. A new command to probe the channel number (CHNO) of an attached channel by the compute node interface is introduced. Any task can probe the cop controller to get the channel number assigned to that host. The cop controller will simply send the current value of the channel poll counter, upon receiving the probe command. This allows hosts to automatically configure the virtual machine for effectively using the cop resources.

2. The Mask RAMs were modified so that each mask entry in the terminal count and broadcast Mask RAMs has an 'accessed' bit associated with it. When a mask entry is initialized by a task, its 'accessed' bit is automatically set by hardware. For the terminal-count mask, this bit will be automatically reset when all the channels specified by the corresponding mask entry have contributed for the global operation. For the broadcast mask, the hardware resets the accessed bit when the global operation result is broadcast to all the channels specified in the corresponding broadcast mask entry. Therefore a mask entry with the 'accessed' bit 'on', implies that the global operation corresponding to that mask entry is in process. During the SETTC and SETBM operations to initialize a mask, the cop controller first verifies that the target mask entry is not already initialized or in use by other tasks, by checking the 'accessed' bit. If the 'accessed' bit is already set, the cop controller reports the error condition to the channel that initiated the SETBM/SETTC operation. The task that initiated the mask initialization operation can read the error condition reported by the cop controller from the 'ERROR-CODE' field in compute node interface status register.

Another reason for introducing this hardware feature is to support the 'spmd' (single program multiple data) programming model followed by some applications. The group operations in a classical master-slave model are initiated by the master task, and the slave programs simply contribute to the group operation. In a COP system, the

master task can be responsible for all mask initialization, and the slave tasks simply contribute to the global operation performed on the cop controller. But in 'spmd' programming models, all the tasks run the same program. The tasks arbitrate between themselves based on a well defined rule (for example, the task who joined the group first becomes the master) to perform any initialization. Simple software techniques include using spinlocks, mutex locks etc. This mechanism requires application programmers to provide very fine load balancing between participating tasks, so that no tasks wait for the master to perform the initialization. In the case of a COP system all the tasks probe the cop controller to check whether the respective masks are already initialized. The task that first access the cop controller succeeds in initializing the TC/BCAST masks, thereby setting the 'accessed' bit on. All the other tasks participating in the global operation see that the masks are already initialized and simply continues by making their contribution to the global operation. In summary, the 'accessed' bit for mask entries in a cop controller provides necessary protection from application bugs, and also allows efficient 'spmd' programming model used by many applications.

3. Another hardware feature considered for the cop controller was to provide ways to initialize an array of consecutive mask entries using a single SETTC/SETBM operation. Each entry in the terminal-count/broadcast mask corresponds to an address location in the Data RAM global address space. Therefore, in-order to perform global operations on application data arrays, the corresponding mask arrays need to be initialized. It is grossly inefficient to initialize a TC/BCAST mask array of size N by issuing N SETTC/SETBM operations. The revised hardware feature provides a count parameter to be used along with SETTC and SETBM operations to initialize a mask array. Figure 14 shows the cop message format for a SETTC operation.

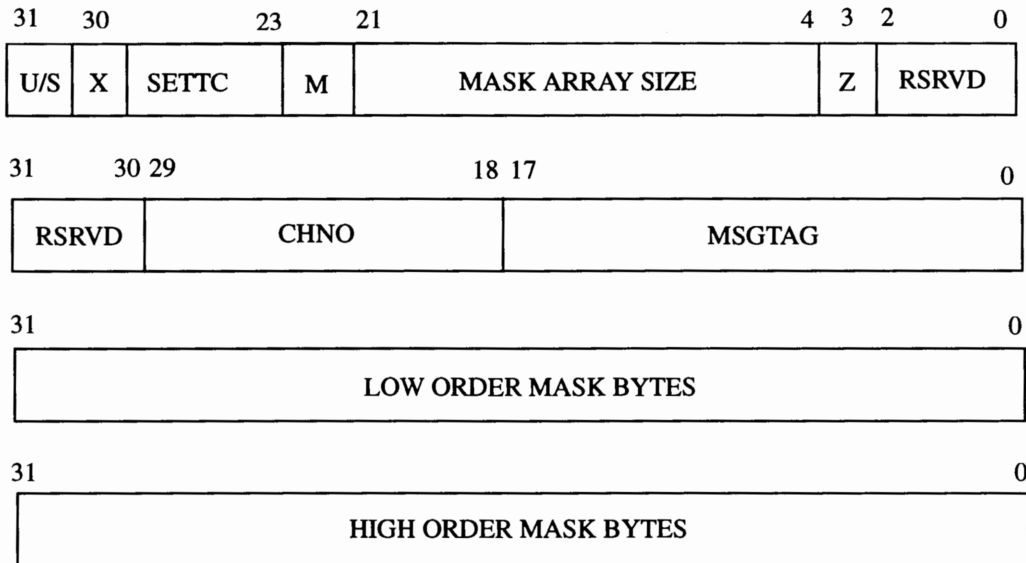


Figure 14. COP Message Format for SETTC operation

The COP Software Implementation

A brief introduction to the layered architecture of the COP system was shown in Figure 10 earlier in this chapter. One of the philosophical decisions made in the early stages of the design was to provide the user enough flexibility to select a pvm function or an equivalent cop function for any given operation. In the following sections of this chapter, I will discuss the implementation of the COP software system with function prototypes for various parallel operations.

The COP Interface Driver

The interface driver is the lowest layer in the cop software system. The interface driver is implemented as a generic UNIX pseudo-device driver, as the hardware prototype was not available. This can be easily ported to support the cop interface

hardware when it becomes available. The cop interface driver runs in kernel space and needs to be integrated into the operating system by re-configuring the kernel running on each compute node. In order to avoid rebuilding the kernel to attach the driver, the cop driver is implemented as a 'loadable' driver. Loadable drivers can be easily attached to or detached from a running kernel using the standard UNIX [Sunm90] 'modload'/ modunload' features. The prototype driver was designed, coded and tested for SUNOS on SUN4 architectures.

One of the primary overheads incurred by pvm functions is time to traverse the communication protocol stacks. The most important reason behind this overhead is the number of times the application data buffer is copied before it reaches the destination. Profiling results of many applications has shown function 'bcopy()' accounting for a major percentage of the processing power and time [Manch94a]. The cop driver reduces these overhead problems by providing direct, but protected network interface access to the application tasks [Blum94a]. As a result of this direct network access, the application data buffers are directly transferred to the network. The COP driver entry point to application tasks is provided through the '/dev/cop' device file. The entry points provided include `open()`, `close()`, `read()`, `write()`, `ioctl()` and `mmap()` system calls. The device can be set to a normal or direct mapped mode using a standard ioctl function call. When in direct map mode, the mmap system call is used by the cop library layer to provide memory mapped direct access to the network interface. The cop driver maps the hardware registers into a given virtual memory location in user address space in response to the `mmap` system call. Due to protection reasons, as there is no buffering of service requests from application tasks in the COP interface driver, the current implementation does not allow multiple concurrent users of the cop resources when the hardware is mapped for direct access by any task. A simple semaphore lock

mechanism is used to share the cop channel access between multiple tasks.

The COP interface driver makes use of the Pid register in the compute node interface to provide necessary protection for application tasks. The application task provides its TKNO (deduced from PVM tid) when it requests the COP driver for direct access to the COP channel interface. The COP driver updates an internal table with the requesting task's process-id (pid) and the TKNO provided in the system call. The driver initializes the Pid register on the channel interface with the TKNO provided, and maps the DataRead/DataWrite registers to the calling task's address space. From now on, the channel interface will only allow those messages to be read out that has the TKNO field identical to the value in its Pid register. All messages with different TKNO values (out of context messages) will be buffered in the channel interface. Whenever the internal buffers cross a pre-specified watermark the COP driver will read those messages out to a shared memory block attached to the driver. This protection mechanism makes sure that even in direct mapped mode, an application task cannot read in messages arriving for other local tasks.

If a message which is not expected by the local task currently accessing the interface arrives at the interface, the message will be saved in an attached shared memory block. Later, when the task expecting this message accesses the interface, it can read out the message from the shared memory provided its TKNO matches the value in the TKNO field in the message. This simple shared buffer allows buffering of out of context messages that may arrive at a compute node. A simple example for the need to buffer out of context messages will be a broadcast over the cop channels. If the out of context message buffering were not available at the channel interfaces, a task performing broadcast would have to make sure that all the recipient tasks in various hosts are currently accessing the channel before actually starting the broadcast. Access

to the shared memory block is restricted to only the process that owns the cop channel interface at any given time. These features allows the cop driver to perform data transfers over the cop channels with the least overhead. Also as the channels are dedicated links, there is no overhead incurred due to packet assembly/reassembly as in general purpose networking protocols used by pvm. Also the driver makes use of the dead-lock detection timer and automatic re-try mechanism of the cop channel interface.

Libcop Library

The libcop library functions provide the entry points for application tasks to use the cop resources. Part of the libcop library is layered on top of the pvm library, so as to make use of the various pvm features like task management, dynamic groups, asynchronous notification of events etc. The libcop functions masks all the inner details of the COP channel access and messaging protocols from the application developer. Also effort is made to keep the cop function prototypes similar to the corresponding pvm functions whenever possible, so as to allow easy porting of existing pvm applications. Libcop is written in C and directly supports C and C++ applications. A Fortran library can be easily written to wrap the current libcop library so that it conforms to the fortran calling conventions.

The access to the compute node interface is protected by means of a semaphore lock mechanism. Tasks belonging to a group can be distributed in different channels/ hosts. For global operations like barrier/reductions, a partial reduction is performed between tasks local to a host/channel using shared memory and semaphore mechanisms. The partial reductions helps to reduce the number of access to the cop channels otherwise needed. The partial reduction results are sent to the cop controller by one of the participating local tasks from each channel. The cop controller performs the

reduction on contributions from all channels, and broadcast the result to the appropriate channels. All the local tasks on a host waiting for the global reduction result share the result received from the channel, again using a shared memory/semaphore mechanism. In summary, tasks local to a host perform partial synchronization/reduction using shared memory constructs, and channels perform the global synchronization/reduction on the cop controller. This approach allows to reduce the contention for the channels. Also tasks who only contribute, but do not require the global operation result can continue as soon as their contribution is passed for local reduction, instead of spinning for the operation to be over.

The functions currently provided by libcop library includes initialization functions like `cop_mytid()`, management function like `cop_setopt()`, `cop_getopt()`, `cop_notify()`, point-to-point communication functions like `cop_write()`, `cop_read()` and global operation function like `cop_barrier()`, `cop_bcast()`, `cop_mcast()`, `cop_reduce()`. Some of the other possible constructs include `cop_rmw()` for read modify write operations, `cop_scatter()` for scatter operations and `cop_gather()` for gather operations. With the basic software architecture of the COP system fresh in mind, next I will discuss in detail the design, implementation of each cop function and their resemblance to the parent pvm functions.

Task Initialization Functions

```
int tid = pvm_mytid();
```

This pvm routine enrolls a process into pvm on its first call and generates a unique tid if this process was not spawned already. pvm_mytid returns the tid of the calling process and can be called multiple times in an application.

```
int tid = cop_mytid();
```

The equivalent function provided by the libcop library is `cop_mytid()`. The function prototype is identical to `pvm_mytid` and is essentially a wrapper around the `pvm_mytid()` function. Similar to the original function, the new function returns the pvm tid assigned by the pvm daemon. Additionally, the function verifies the presence of a healthy channel to the cop controller, probes the attached controller for the channel number (CHNO) assigned to the channel, and retrieves the controller number of the attached controller (CNTNO) if it is a multi-level COP system. The channel/controller numbers are stored in static variables internal to the libcop library. The function call reports error if it encountered with a problem in channel initialization. Other management functions in the libcop library are `cop_setopt()` and `cop_getopt()`. Using cop_setopt we can specify a pvm group of tasks as static, which indirectly calls the pvm function pvm_staticgroup. All the tids of member tasks in the group will be stored in internal variables to libcop. Later, if a libcop function needs to access the tids of member tasks of this group, the internally stored values will be used instead of retrieving them from the pvm group server. cop_getopt simply allows a program to probe the options set using cop_setopt.

Memory Write/Read Functions

As described before, the COP system Data RAM is divided into channel address spaces, and later into task address spaces. A task address space can be accessed for writing only by the owner task, but any task can read any task address spaces. Only read-modify-write operations from a task can write to the task address spaces of other tasks. Therefore the task address spaces are used for unidirectional communication between tasks using the COP system. This remote shared memory operation is a by-product of the COP system architecture and as I show later, it is a effective mechanism for low

latency communications. Even though these operations extends beyond a message passing model, they are included in the COP functions library. The cop functions cop_write and cop_read allows an application program to write/read variables in task address spaces on cop controller RAMs. The function prototype for write operation is:

```
int ret = cop_write (void *data, int count, int datatype,
                     int tid, int msgtag, int mode)
```

The function cop_write allows an application task to write to its own task address space on the cop controller. The first argument is a pointer to the data array that needs to be written to the task address space. The 'count' argument specifies the number of elements in the data array. The 'datatype' argument specifies the type of data present in the data buffer. The supported types are integer, float and double. The 'tid' argument specifies the tid of the task that owns the target task address space. Presently, the only valid value of this argument is the calling task's tid, as the calling task can write only to its own address space. This argument is present in the function prototype to support any future changes in access restriction policies. The 'msgtag' argument specifies a tag for the cop operation. It is used as an offset into the task address space to start storing the data buffer. Two types of modes can be specified using the 'mode' argument: Urgent and Normal. In the urgent mode, the software locks the cop channel, sends all the elements of the data buffer to the cop controller one by one, and then unlocks the channel. This mode allows programs to atomically initialize data buffers in task address space, but will stall operations on other channels until the lock is released. In the normal mode, the lock/unlock features are not used. As a result the controller may service requests from other channels between sending two consecutive elements in the data array to the controller. The data elements are sent to the cop controller as WSRAM messages.

The read operation on task address space is performed by application tasks using

the cop_read function. The function prototype is similar to the write operation.

```
int ret = cop_read (void *data, int count, int datatype,
                    int tid, int msgtag, int mode)
```

The 'data' argument points to the buffer to read in the data array from task address space. The 'tid' argument specifies the task identifier of the task which owns the task address space to be read. The CHNO and TKNO values that point to the base of the task address space are derived from the H and L fields in the 'tid' argument (direct mapping). 'count' number of data elements are read out one after the other, starting from offset 'msgtag' in the target task address space. The data elements are read out by issuing RSRAM messages to the COP controller. The datatype, msgtag and mode arguments work the same as for the cop_write function.

The read modify write operation works exactly the same as cop_write and cop_read. The function is called with the data to be used for modifying the variable in the 'data' array. The cop controller modifies the specified variable by adding the new data to the old variable value. The old value of the variable before modification is returned to the compute node. On return the 'data' array will contain the variables before it was modified. The RMW32/64 opcode is used in these cop messages. The function prototype is similar to cop_write as

```
int ret = cop_rmw (void *data, int count, int datatype,
                   int tid, int msgtag, int mode)
```

Collective Communication Functions

The richness of cop architecture is designed mostly for collective communication functions. Some of the global operations implemented on the COP system include broadcast, multicast, global reductions and barriers. In this section, I will explain in

detail how these operations are implemented.

Broadcast/ Multicast Implementation

PVM uses the pvm_bcast function to broadcast the data in the active message buffer. The function prototype is:

```
int info = pvm_bcast (char *group, int msgtag)
```

where the argument 'group' is an existing group name and 'msgtag' is an integer tag supplied by the user to distinguish between different kinds of messages. A return value of less than zero indicates an error.

In a COP system, the equivalent function for broadcasting is cop_bcast. The function prototype is

```
int info = cop_bcast (void *data, int count, int datatype,
                      char *group, int msgtag, int mode, int srctid)
```

where argument 'data' points to the data array which needs to broadcast, 'count' specifies the number of elements in data array, 'datatype' specifies the type of data in the array, 'group' specifies an existing pvm group name and 'msgtag' is a user specified tag which gets used as an offset in the global address space of cop controller. The 'mode' argument specifies whether the function is called to send a broadcast or to receive a broadcast from some other tasks. When in receive mode, the 'srctid' argument specifies the process tid which originally performed the broadcast. A value of -1 for srctid will receive broadcasts from any peer tasks.

As the cop software functions do not explicitly pack or unpack data buffers, there are no active send/receive buffers as in pvm. The COP hardware follows a specific data format and byte order and compute nodes are required to do the format translations for any application data before sending to the COP controller. A separate function library

can be easily developed similar to the PVM pack/unpack functions to perform this translation if needed.

From the group name provided, the cop_bcast function retrieves the tids of all tasks who are members of the specified group from pvm's group database. The broadcast to all local tasks (tasks in the same hosts) in this group is performed using a shared memory mechanism. The data elements are broadcast to tasks in other compute nodes using the cop channels.

The sender allocates a local shared memory and a lock with message tag as their key, if there are any local recipient tasks for the broadcast. The sender copies the broadcast data to the shared memory and leaves it unlocked. The broadcast to foreign tasks in other compute nodes continues from the sending tasks's local buffer. The receiving tasks call the cop_bcast function in the receive mode. The tid argument is used to identify whether the sender is a local task or a foreign task. If the sender is a local task, the broadcast data is read out of the local shared memory, co-operating with other local recipient tasks. If the sender is a foreign task, the cop channel is accessed to read out the broadcast data. If there are multiple recipient tasks at a remote compute node, a shared memory mechanism is used to share the broadcast data received from the channel.

If there are foreign recipient tasks, the sender will use the cop channel to perform the broadcast. This is done in the following way: The channel is first locked to assure memory consistency in the cop controller global address space currently used. The tids of all recipient tasks in the group are analyzed to generate a broadcast mask. i.e if there is at least one recipient task on a compute node connected to cop channel 'X', the Xth bit is set in the broadcast mask. The SETBM message is issued to the cop controller, which in turn will initialize the mask entries at offsets 'msgtag' to 'msgatg+count' in the BCAST RAM. A mask entry per data element in the array has the potential for

combining scatter operations after reduction operation on a data array. If any of these mask entries are already in use, the controller will return an error using the negative acknowledgment (NACK) message. When a data element is received for broadcast by the cop channel, it is broadcast to all channels specified by the corresponding broadcast mask. As a result broadcast data elements will start showing up in the receive buffers of all receiving channels even before all the elements of data array is sent by the sending channel. After all the data elements are sent, the function unlocks the channel using an UNLOCK operation, and returns the status of the broadcast to the sending task.

A cop multicast is very similar to a cop broadcast, but instead of the group name, the application provides the task identifiers of all recipient tasks. COP multicast is more efficient than cop_bcast as there is no overhead due to pvm function calls to get the tids of tasks belonging to the given group (The tids of member tasks in the group is assumed to be retrieved in the initialization phase of applications to be used later for multicasts). The function prototype for COP multicast is

```
int info = cop_mcast (void *data, int count, int datatype,
           int *tids, int numtasks, int msgtag, int mode,
           int srctid)
```

where argument ‘tids’ point to an array of recipient task identifiers, and ‘numtasks’ specify the size of tids array. All the other arguments work exactly same as in cop_bcast function.

COP broadcast does not send the message back to the sender in the current implementation. A task can broadcast to a group, even if it is not a member of the group. cop_bcast is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving tasks. Due to the simultaneous broadcast on all specified cop channels by the cop controller, cop_bcast is highly

efficient.

Barrier Implementation

The PVM function `pvm_barrier` blocks all the calling processes until all processes in a group have called the function. The function prototype is:

```
int info = pvm_barrier (char *group, int count)
```

where the argument 'group' is an existing group name and 'count' is an integer specifying the number of group members that must call pvm_barrier before they are all released.

Though not required, count is expected to be the total number of members of the specified group. A count parameter is required, because with dynamic process groups pvm cannot know how many members are in a group at a given instant. During any given barrier call all participating group members must call barrier with the same count value. Once a given barrier has been successfully passed, pvm_barrier can be called again by the same group using the same group name. A return value of less than zero indicates an error condition encountered.

In a COP system, the equivalent function for barrier synchronization is cop_barrier. The function prototype is

```
int info = cop_barrier (char *group, int count, int msgtag)
```

where argument 'group' and 'count' arguments function exactly same as in pvm_barrier. 'msgtag' argument is used as an offset in the global address space where the barrier operation is performed.

Similar to the cop_bcast function, the cop_barrier function retrieves the tids of all member tasks of the specified group from the pvm group database. The function waits until 'count' number of members have joined the group. If there are multiple local tasks

(tasks in same host) participating in the barrier, a partial barrier is performed between them at the compute node. This is done using shared memory and semaphores. The first local task entering the barrier, grabs the 'barrier_start' lock and initializes the barrier variable (counter) in shared memory. Later, the lock is released and the task waits for the 'barrier_end' lock. All the other participating tasks wait for the 'barrier_start' lock, update the barrier variable in shared memory when the lock is granted, and wait for the 'barrier_end' lock. The last local task entering the barrier will access the cop channel to perform the barrier operation between compute nodes, if there are participating tasks in other compute nodes. The channel access for barrier is done as follows:

First the terminal-count and broadcast mask bit vectors are generated from the tids of all tasks in the group. As mentioned originally, the terminal-count mask indicates the channels participating in a global operation, and the broadcast mask indicates the channels to which the result of a global operation will be broadcast. For a barrier, as tasks on each participating channel will be blocked until the barrier is completed, the broadcast mask is same as the terminal count mask. So if there is at least one task in the specified group residing in the host connected to channel 'X', the Xth bit will be set in the terminal count and broadcast mask vectors. The channel is first locked by issuing a LOCK message to the cop controller. Then a SETTC message is issued to the cop controller to initialize the terminal count mask. The controller computes the address of the mask entry in the Mask RAM from the message tag supplied in the SETTC message, and initializes it with the mask bit vector specified in the data fields of the message. If the mask is already initialized by a participating task from another channel, the cop controller sends a negative acknowledgment indicating the mask already exists. The 'accessed' bit associated with each mask entry will be used by the cop controller to determine whether it was already initialized or not. If the response for the SETTC

operation indicates that masks are already initialized, the SETBM operation to initialize broadcast mask is skipped. This saves the overhead involved in trying to set the broadcast mask, when we know that a peer channel has already initialized them. The broadcast mask and terminal-count mask will be in sync for barriers, because we do them atomically by locking the channel. If the SETTC operation was successful, the broadcast mask is initialized using the SETBM operation. Next, the contribution from the channel for the barrier operation is registered by issuing a BENTRY message. Upon receiving this message, the cop controller resets the bit corresponding to the channel in the terminal-count mask entry. The channel is unlocked for other channels to participate in the barrier. Upon receiving the contributions from all channels, all the bits in tc-mask vector will be reset and the controller broadcasts the barrier completion message to all the channels specified in the corresponding broadcast mask entry.

The sending tasks at each channel block for the barrier completion response broadcast from the controller. When the barrier completion response finally arrives, the sending task in each channel updates (decrements) the barrier variable (counter) in shared memory, and unlocks the 'barrier_end' lock. Local tasks waiting for this lock, grab the lock one after another and update the barrier variable. The last local task leaving the barrier performs additional book-keeping responsibilities like flushing the shared memory buffers and semaphores.

The cop_barrier function blocks the calling process until count members of the group have called cop_barrier. Therefore, the logical function of cop_barrier is to provide an efficient low-latency group synchronization. The overhead involved in retrieving the tids of all member tasks in a group can be avoided, if the groups are made static using pvm_staticgroup. Any unsolicited messages (like broadcasts) received from the channel while the task accessing the cop interface is blocked on the barrier is

buffered in the shared memory block attached to the cop interface.

Global Reduction Implementation

PVM function pvm_reduce performs a reduce operation over members of the specified group. The function prototype is

```
int info = pvm_reduce (void (*func)(), void *data,
                       int count, int datatype, int msgtag,
                       char *group, int root)
```

where the argument 'func' defines the operation on the global data. Predefined operations include Max, Min, Sum and Product. Users do have the flexibility to define their own reduction functions. The argument 'data' points to the starting address of an array of local values. On return, the data array on the root task will be overwritten with the result of the reduce operation over the group. The 'count' argument specifies the number of elements in the data array, 'datatype' specifies the type of entries in the data array, 'msgtag' indicates the message tag supplied by the user, 'group' is the name of an existing group and 'root' is the integer instance number of the group member that gets the result. A return value of less than zero indicates error.

All group members call pvm_reduce() with their local data, and the result of the reduction operation appears on the user specified root task identified by its instance number in the group. If more than one member of the group requires the reduced result, the root task has to broadcast/multicast it to the group.

In a COP system, the equivalent function for all global operations is cop_reduce. The function prototype is

```
int info = cop_reduce (int func, void *data, int datatype,
```

```
int count, int msgtag, char *group, int *roots,
int rootcount)
```

where argument 'func' specifies one of the predefined global operations and 'data' points to the starting address of an array of local values. On return, the data array on all the root tasks will be overwritten with the result of the reduce operation over the group. The 'count' argument specifies the number of elements in the data array, 'datatype' specifies the type of entries in the data array, 'msgtag' indicates the message tag supplied by the user, 'group' is the name of an existing group, 'roots' point to an array of integer instance numbers of group members that receives the reduced result and 'rootcount' specify the size of the 'roots' array. A return value of less than zero indicates error. cop_reduce makes use of the low-overhead broadcasting mechanism of the COP system to accommodate more than one root task for a global reduction. This takes away the overhead involved in broadcasting or multicasting the reduced result if more than one member of the group requires the reduced result. As the cop channels are not shared between compute nodes, the reduced data will be broadcast simultaneously over all the participating channels.

Similar to other group functions, cop_reduce retrieves the tids of all member tasks of the specified group from the pvm group database. It also retrieves the instance numbers assigned to these tasks. The instance numbers specified in the 'roots' array are mapped to their corresponding tids to form an array of tids of all root tasks. If there are multiple local tasks participating in the reduction operation on a compute node, a partial reduction of their contributions is done before accessing the cop channel. The last local task to call the reduce function, accesses the cop channel to submit the partial reduced data from the channel. This is done as follows. The tids of all member tasks in the group

are analyzed to determine to which channels they belong. The terminal count mask vector is generated using this information. Similarly the tids of all root tasks are analyzed to generate the broadcast mask vector. The cop channel is first locked using the LOCK operation, and a SETTC message is issued to initialize the tc-mask entry at offset specified by 'msgtag' in the TC Mask RAM. The 'count' field is specified in the message, so that the cop controller can initialize mask entries at offsets ranging from 'msgtag' to 'msgatg+count' in the Mask RAMs. If the cop controller reports that the mask entry is already initialized, the SETBM operation to initialize the broadcast mask is skipped as we know a peer channel has already performed the mask initialization

If the SETBM operation is done by a channel (which is true for only the first channel that contributes for the reduction), it should also initialize the data buffer to be used in the global address space with its own partially reduced contribution. This is needed to make sure that stale data in these data buffers will not be used for global reductions. The data initialization is done using WSRAM operations. The channel is unlocked after data initialization. All the other participating channels will contribute one element at a time whenever the attached channel is serviced by the controller. The contribution is issued using reduction messages, with opcodes depending on the function and datatype arguments specified (listed in Table I in Chapter IV), but the message tag field in the message is incremented each time. This allows overlapping the data fetch time and buffer management time in one compute node with the communication times of other channels. The controller updates the corresponding TC mask entry upon performing a reduction of each data element with the contribution from a channel. When all bits in a TC mask entry are reset, the controller broadcasts the reduced data element to all channels specified in the respective broadcast mask entry. Between each data

element contribution sent to the controller, a check for a possible reduced element in the receive buffer is done in all root compute nodes. This is done because, each element in the data buffer in the global address space is broadcast whenever the contributions for that element are accumulated from all participating channels, irrespective of other elements in data array. The sending task in each root node waits until it receives all the reduced elements of the data array. If there are multiple local root tasks, the reduced information is distributed using shared memory. All the non-root tasks do not wait for the reduction result, but will continue after they have made their contribution for the reduction operation. If all the tasks participating in the global reduction are local tasks, the cop channel is never accessed. The partial reduction result in this case is same as the global reduction result.

Some of the limiting factors of the libcop function include the size of the data array that can be used depending on the memory available on the controller, and possibility of using only predefined reduction functions as reduction is performed by the controller hardware. But these limitations are inconspicuous as we can recursively do fast reductions on small data arrays efficiently, and as most applications use only common logical/arithmetic reduction functions like SUM, MAX, MIN etc.

Some of the other group functions that can be efficiently performed using the cop channels include scatter and gather functions. In the scatter operation an application data buffer can be scattered across different task address spaces in the cop controller. Similarly data buffer fragments across multiple task address spaces can be gathered and presented to any participating task(s) in a gather operation. The COP architecture is best suited for parallel programming constructs that involve collective communications. The independent channels, custom designed controllers and the efficient software interface

account for the low latency implementation of these global operations. Table II summarizes various libcop library features/functions currently implemented and their equivalent libpvm library functions.

TABLE II

SUMMARY OF LIBCOP & LIBPVM LIBRARY FEATURES

Feature	Libcop Function	Libpvm Function
Group Operations		
Broadcast	cop_bcast	pvm_bcast
Multicast	cop_mcast	pvm_mcast
Barrier	cop_barrier	pvm_barrier
Global Reduce	cop_reduce	pvm_reduce
Gather	cop_gather	pvm_gather
Scatter	cop_scatter	pvm_scatter
Point-to-Point		
Send Message		pvm_send
Receive Message		pvm_recv
Shared Memory		
Write to shared memory	cop_write	
Read from shared memory	cop_read	
Read Modify Write	cop_rmw	
Task Initialization		
Get Options	cop_getopt	pvm_getopt
Set Options	cop_setopt	pvm_setopt
Enroll in COP/PVM system	cop_mytid	pvm_mytid

CHAPTER VI

COP PERFORMANCE ANALYSIS AND COMPARISONS

In the last chapter I showed how the cop software libraries are designed to efficiently provide various common parallel group operations and protected shared variable access. In this chapter, I extend the analysis of the COP system, derive equations for their global operation timings, compare them with the performance of equivalent pvm constructs, and project the actual improvement in execution times for real applications.

Reducing the COP Software Overhead

One of the primary performance limitations of the popular message passing systems is the software overhead involved in traversing the protocol stacks [Sten94a]. A major portion of this overhead is incurred due to copying of application data buffers between various software layers in the stack. A primary objective in the cop software design was to provide the application program the entire hardware bandwidth of the channels with adequate protection schemes. Also as the COP system channels are not general purpose, and as they are not time shared, no complex message formatting is needed to communicate between compute nodes and the cop controller.

The cop interface driver provides an efficient way to reduce the software overhead that otherwise would result while accessing the cop channels. As described originally in the previous chapter, in the normal mode of operation an application data buffer is presented to the device driver with a write() call. The application buffer is first copied into the system address space of the driver and later sent out by accessing the cop

channel. Similarly when a message arrives at the compute node interface it is first stored in the receive buffers of the driver and later presented to the application on a read() call. This normal mode of operation involves two copies of the application data on both transmit and receive operations before the data is available at the destination. Since the number of bytes copied increases with the size of the data buffer, this overhead increases rapidly with the data buffer size and the number of read and write calls issued.

The protected direct access mode provided by the cop driver allows application tasks to directly access the cop interface ports. The application task can make an ioctl call to put the driver in this mode, before it maps the cop transmit and receive registers to the address space of application task. Also as the access to control/status registers are privileged, the channel interface is protected from applications that behave abnormally. As mentioned in the section on the COP interface driver in the previous chapter, the Pid register in the channel interface is used to provide adequate protection, so that application tasks cannot read in messages arriving for other local tasks. The direct mapping of interface ports, the protection schemes, and the underlying communication protocol are masked from application programmers in the libcop library functions. As a result, the application programs can simply call the proper libcop function to perform high speed I/O over the cop channels. When the cop driver is put in direct mapped mode, the libcop functions can talk directly to the cop channels without unnecessary copying of application data. Whenever out of context data appears at the interface from the cop controller, the interface/driver buffers it, so that the destination tasks for those messages can access them after proper authentication whenever cop channel access is obtained.

As described in the previous chapters, partial reductions and synchronization of local tasks are performed at each compute node using shared memory mechanisms before the total contribution from the compute node is presented to the cop controller.

Even though the local synchronization is done in local memory, its efficiency depends on the efficiency of inter-process communication (IPC) implementations of the operating system, memory to memory copy speed of the processor, and the number of tasks awaiting execution in the scheduling mechanism of the kernel. For these reasons, a global operation using a cop controller between N tasks will work most efficiently, if the N participating tasks reside on N different compute nodes (channels) connected to the COP controller. Even though this seems to be counter-intuitive, there are similar examples on several existing systems. For example, on an Intel Paragon, the local memory to memory copying is surprisingly slow. The provided bcopy() routine in the C library peaks at about 65 to 70 MB/sec. For comparison, a transfer of a large message from one node to a different node can attain speeds of 130 Mb/sec for the most recent network interface component (NIC) [Para94a]. PvmDataDefault encoding performs extremely slowly compared to PvmDataRaw encoding for any pvm function on these MPP systems as packing and unpacking of data buffers involves bcopy on both sender and receiver. For these reasons the PVM psend() functions perform noticeably faster than their equivalent functions that involve data packing/unpacking.

Performance Analysis of Libcop Functions

In all the timing calculations presented here, only one task per channel is considered, so that the COP system performance results are not affected by external variables like the efficiency of the underlying operating system implementation, or the speed at which compute nodes can perform local memory copying. Therefore, the effective performance seen by any application depends on how the tasks are distributed across the virtual machine and the native speed of the original system. I will analyze the cost of various libcop functions in the following sections. In all the cases the libcop

functions use a direct mapped compute node interface to access the channels.

Assuming only one task per host in the virtual machine, the overall time involved in a libcop library global function can be subdivided into the following.

1. Given the name of a task group, the time involved in retrieving the tids of all member tasks in that group.

2. The time involved in interpreting the PVM tids to identify their respective CHNO and TKNO values.

3. The time involved in accessing the compute node channel interface.

4. The total hardware time involved in performing the specified parallel operation. This includes time for actually sending messages to the controller, the time involved in setting up masks/variables in controller RAM, time to perform the global operation by the controller, and the time to send the result to all receiving tasks.

The first three timings involve the software overhead in the libcop library functions. The PVM group server stores all the information about dynamic task groups, and can be queried to get the tids of all member tasks of a group at a given instant. This communication involves some overhead, but is only needed if the specified group is a dynamic group. As the majority of the 10 applications I surveyed did not make use of the dynamic nature of task groups in PVM, I consider groups declared as static. If a group is declared static, the libcop library will contact the PVM group server only in the first call. The tids of member tasks received from the group server, are stored in internal variables in the libcop library. For all the subsequent calls involving global operations on the same task group, these internally stored tids are used. Hence the overhead of retrieving tids from group names is assumed to be a negligible value over multiple function calls.

As mentioned in the previous chapter, if the hosts are added to the virtual machine in the order of their channel numbers, the H field in the pvm tid of a task in any host is

equal to the channel number (CHNO) of the host where it resides. Also as the L field in a pvm tid is assigned locally to the host, it can be used directly as the TKNO field in the COP system. Thus, no additional time is involved in deducing TKNOs and CHNOs from a pvm tid, thereby zeroing out the time for interpreting the pvm tids.

The time involved in actually accessing the secondary network interface and related buffer management can be significantly reduced by using the protected direct access mode of the COP device driver where the network interface registers and memory blocks are mapped directly mapped to the virtual memory space of the application task.

The basic hardware timing values of the COP system involve the time spent to send a message to the cop controller, the time spent on servicing the request by the controller, and the time involved in sending the result back to the compute nodes from the controller. As described in last chapter, each libcop library global function uses a variety of COP system messages. Assuming the software overhead involved in accessing the secondary network to be a negligible minimum, the performance of libcop functions largely depends on the number of messages sent to the COP controller and the hardware time involved in processing them.

For the hardware timings of various COP system operations, I refer to the simulation results reported by Hall [Hall94a]. The minor hardware modifications referred in the previous chapter to the controller for increasing the programmability of the system are not expected to make any major difference from the original timings reported. In the original simulation, a 10 ns clock period was used. The number of clock cycles required for a compute node to output a word and for the interface to transmit it to a cop controller will be represented as $t_{CN\text{-}COP}$. Assuming the compute node can write a 64-bit word to its interface port in two clock cycles, $t_{CN\text{-}COP}$ is equal to 11 clocks for a command word, instruction word, and a 64-bit data word. The number of cycles

consumed by the COP controller to service a single COP message from a channel is referred as t_{OP}. The value reported for t_{OP} is 3 for integer and Data RAM operations, but increases to 7 for floating point operations. To implement broadcast functions, the controller uses a mask entry from the broadcast mask RAM to enable the transmit buffers at the same time. The interfaces transmit the data word to all or a desired subset of the compute nodes simultaneously. Again assuming a compute node can read a 64-bit value from its interface port in two cycles, 17 clock cycles are required for the network interfaces to transfer a broadcast data value and the receiving node to read the word. This time will be referred to as $t_{COP\text{-}CN}$.

The total number of clocks to cycle through all the requesting channels and get to the target channel requesting service is $\overline{C} \times \overline{t_{OP}}$. In this expression $\overline{C}$ represents the average number of channels requesting service at a time in a cop controller and $\overline{t_{OP}}$ represents the average number of clock cycles that a cop controller requires to service each input channel. The value of $\overline{C}$ for a single level COP is in the range of 0 (best case condition) to 63 (worst case condition, if all the channels are requesting service).

When a channel is locked using the LOCK message, it will block the cop controller from servicing any other channels until the current channel releases the lock by sending an UNLOCK message. Thus the average time taken by the cop controller to service a channel ($\overline{t_{OP}}$) will be drastically increased if tasks use the lock feature excessively. One of the easiest ways to restrict the time a channel is locked is to restrict the size of data buffers that can be used with the libcop functions. If the data buffer sizes are small, the time a channel is in the locked state is much less, and therefore every channel gets an almost equal share of the controller resources. The current implementation limits the size of data arrays used with a single libcop function call to 1024 bytes (128 cop messages). Due to this implementation, in the worst case condition

(for broadcasts and shared memory read operations) the number of controller cycles to service a channel will be the time to send 128 messages, $t_{MAX\text{-}OP}$, where $t_{MAX-OP} = 128 \times (t_{OP} + t_{COP-CN})$.

For large data arrays, application tasks can make multiple calls to the libcop functions, each time with a different portion of the data array, so that between each function call the controller is released for use by other channels. As the channel interface registers are directly mapped to the task's address space, multiple calls to libcop functions do not impose significant extra overhead (only the function call overhead) compared to performing the same operation in a single function call with a large data array. Therefore the value of $\overline{t_{OP}}$ for a single level COP is in the range of t_{OP} (best case with no locking) to $t_{MAX\text{-}OP}$ (worst case if broadcast/read performed after locking). This implies that excessive channel locking, can significantly degrade the performance of the libcop functions. The worst case scenario will be all channels requesting broadcast or shared memory read operations of data arrays repeatedly.

Broadcast Using `cop_bcast()`

The `cop_bcast` libcop function allows an application task to perform a broadcast to all tasks in a specified group. Tasks can dynamically join and leave groups using the `pvm_joingroup()` and `pvm_lvgroup()` functions. As described in the previous chapter, the `cop_bcast` function involves locking the channel using LOCK operation, setting up the broadcast masks using SETBM operation, broadcasting each data element using BCAST operation, and finally unlocking the channel using UNLOCK operation. In the worst case, the controller may cycle through servicing all the other channels requesting service before it gets to the channel desiring broadcast to service the first lock operation. The maximum number of clock cycles to service the lock

operation is therefore $(t_{CN-COP} + \left(\overline{C} \times \overline{t_{OP}}\right) + t_{OP})$. But once the channel is locked, successive broadcasts can proceed in a 'pipeline' fashion. The key here is that as soon as the cop controller reads a command, a one word ACK is returned to the sending node to indicate that it can send the next command. As the broadcast mask setup inside cop_bcast does not allow the controller to send the broadcast words back to the sending node, a new command can be coming into the controller from the sending node at the same time the broadcast word is being transmitted to the receiving nodes. Due to this pipelined servicing of channel commands a data element can be broadcast every $(t_{OP} + t_{COP-CN})$ cycles. The maximum size of data element that can be attached to a cop message is 8 bytes. As the mask initialization operation is performed only once in a function, I approximate them to take only t_{OP} cycles to simplify the expressions. Therefore the total time to broadcast an application a data array of M bytes containing N-byte data elements is about

$$t_{bcast} = \left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + t_{OP} + \left(\frac{M}{N}\right)(t_{OP} + t_{COP-CN}) + t_{OP}\right] P_{CLK}$$

For a case where all the channels are requesting service ($\overline{C} = 63$), and none of the channels use channel locking ($\overline{t_{OP}} = t_{OP}$) the expression reduces to $t_{bcast} = 4.8 + (M \times 0.03)$ microseconds/byte for 64 node broadcasts after substituting the previously stated values for 64-bit (8 byte) floating point data elements. Broadcasting a 1 kbyte vector in this way requires about 35 microseconds. Performing a 16 kbyte one-to-all broadcast for a cluster size of 64 (broadcast over all the 64 channels) requires around 560 microseconds. The broadcast function performance seen by application programs depend on the number of channels requesting service at a time ($\overline{C}$) and the rate of channel locking ($\overline{t_{OP}}$) performed by these channels. As described in the last chapter, an additional start-up cost will be incurred to retrieve the task identification

numbers of member tasks in the specified group from the pvm group server, but this can be nullified over multiple broadcasts if the group is declared static using `pvm_staticgroup`. For comparison, later I show that the standard `pvm_bcast` function takes thousands of microseconds to perform a 16 kbyte broadcast between comparable number of compute nodes. Also I show that, the pvm broadcast timings increase drastically as the number of receiving nodes is increased.

Multicast Using `cop_mcast`

The libcop multicast function implementation is very similar to the broadcast implementation. As the tids of all recipient tasks are specified in function argument, unlike `cop_bcast`, `cop_mcast` does not involve any communication with the pvm group server for retrieving tids. As a result the above timing derivation for broadcast function can be directly applied for cop multicast operations. Again for comparison, later I show that even though pvm multicast performs much efficiently than pvm broadcasts, its performance is far less than the libcop multicast function performance.

Global Reduction Using `cop_reduce`

The `cop_reduce` libcop function allows a group of application tasks to perform a global reduction of their individual contributions. As described in the previous chapter, the first channel serviced by the cop controller for the global reduction performs data initialization involving operations like locking the channel using the LOCK operation, setting up the broadcast masks using SETBM operation, setting up the terminal-count masks using SETTC operation, initializing the controller Data RAM buffer using WSRAM operations and unlocking the channel using the UNLOCK operation. For an M byte data array containing N-byte data elements, the initialization time can be derived as

$$t_{init} = \left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + t_{OP} + t_{OP} + \left(\frac{M}{N} \times t_{OP}\right) + t_{OP}\right] P_{CLK}$$

All the other participating channels sends out their contribution, one data element at a time whenever the cop controller services that channel. Assuming that all the participating compute nodes send a contribution to the cop controller at the same time, the approximate time needed to broadcast the reduction results to all the root tasks after the above initialization is

$$t_{contrib} = \left[\left((\overline{C} - 1) \times \overline{t_{OP}} + t_{OP} + t_{COP-CN}\right) \times \frac{M}{N}\right] P_{CLK}$$

Each channel has to wait for all the other participating channels to contribute the appropriate data element, before the reduced data element can be broadcast to all root channels. Therefore, the total time involved in a global reduction of a M byte data array can be derived as $t_{init} + t_{contrib}$ and for 64-bit data elements the equation reduces to $t_{reduce} = 4.8 + (0.58 \times M)$ microseconds/byte for 64 node reductions. As all channels participate in global reduction, the average time to service request each channel $\overline{t_{OP}}$ is only t_{OP}. Using this expression, to produce a sum vector for 1024 element double precision floating point vectors over 64 compute nodes (all channels participating in reduction) requires about 4800 microseconds. Using the equivalent expression for integer vectors, to produce a sum vector for 1024 integer vectors over 64 compute nodes requires about 2200 microseconds. These performance numbers will vary if a subset of the cop channels is performing a broadcast (using channel locks) at the time of global reduction. Note that this time also includes the time required to broadcast or multicast the reduction result to all task members in the group. I will show later in this chapter that by comparison, the equivalent `pvm_reduce` function is very poor in performance, and the additional broadcast/multicast function overhead is incurred at the root task while

sending the reduction result to other group members.

Barrier Using `cop_barrier`

Even though barrier operation is a simple case of global reduction, it is considered separately due to the inherent difference in their implementation. The cop controller uses the BENTRY operation for implementing barriers. As described in the previous chapter, the first channel entering the barrier issues operations like locking the channel using LOCK, setting up the terminal-count mask using SETTC, setting up the broadcast mask using SETBM, registering its own barrier contribution using BENTRY, and finally unlocking the channel using UNLOCK. Therefore the total initialization time for barrier operation can be expressed mathematically as

$$t_{init} = \left[\left(t_{CN-COP} + \bar{C} \times \overline{t_{OP}} + t_{OP}\right) + t_{OP} + t_{OP} + t_{OP} + t_{OP}\right] P_{CLK}$$

As all channels participate in the barrier, assuming all the other cop channels entered the barrier at the same time, the $\overline{t_{OP}}$ in above expression is equal to t_{OP}. Each of the other participating channels issues operations like locking the channel using LOCK, checking whether the transmit mask entry is already initialized using SETTC (for spmd programming models), participating in barrier using BENTRY, and finally unlocking the channel using UNLOCK. Therefore the number of cycles required for each participating channel to contribute for the barrier operation is $(4 \times t_{OP})$. After all the channels have participated in the barrier, the controller takes t_{COP_CN} cycles to broadcast the barrier exit status to the participating compute nodes. Therefore, the total time involved for a barrier operation can be expressed as

$$t_{barrier} = t_{init} + [\bar{C} \times 4t_{OP} + t_{COP-CN}] P_{CLK}$$

Using the hardware values reported for integer data elements (a mask entry is a 64

bit integer), a barrier operation between 64 compute nodes in a single level COP system takes about 9.8 microseconds. If a master-slave programming model is followed, the slave tasks do not have to check for mask initialization, as it is guaranteed that the master task has already done the mask initialization. In this case, the lock, settc and unlock operations is not needed for `cop_barrier` called from slave tasks. As a result the number of cycles required for each participating channel to contribute for the barrier operation reduces to t_{OP}, reducing the barrier time for 64 nodes to about 4.21 microseconds. As I show later, the timings measured for the equivalent pvm barrier function takes thousands of microseconds, and increases enormously as the virtual machine size increases. Note that the mask mechanism in the COP system allows any subset of the compute nodes to participate in a barrier. Also, it permits multiple barriers to be in force with different task groups at the same time and thus provides for fine grained synchronization which can greatly improve the performance of certain class of problems like pre-conditioned conjugate gradients [Gupta94a].

Shared Variable Access Using `cop_write()`, `cop_read()` and `cop_rmw()`

As described originally, shared variables in a COP system can be stored in the various task address spaces in the cop controller's Data RAM. The task generating the variables can write them to its task address space in controller Data RAM by calling `cop_write`, and any task that needs to read the shared variables can access them by calling `cop_read`. Only the read-modify-write operation issued through the `cop_rmw` function can update shared variables in task address spaces owned by other tasks. A `cop_write` function call invoked in urgent mode for a data array of M bytes containing N-byte data elements involves a lock operation, $\left(\frac{M}{N}\right)$ wsram operations, and an unlock operation.

Equivalent `cop_read` and `cop_rmw` functions involves additional $\left(\frac{M}{N}\right) \times t_{COP-CN}$ cycles as the value read from Data RAM needs to be sent out to the receiving compute node. Assuming simultaneous requests from all channels of a single level COP system for same or different shared variables, the maximum time for all of the requesting nodes to receive the individual values is

$$t_{read} = \left[\left(t_{CN-COP} + \bar{C} \times \overline{t_{OP}} + t_{OP}\right) + \left(\frac{M}{N} \times (t_{OP} + t_{COP-CN})\right) + t_{OP}\right] P_{CLK}$$

Assuming no other channels is in locked state and substituting the previously stated timing values in the above expression, a single level cop controller can supply different or the same variables (of type double) to 64 compute nodes in the rate of $4.7 + (0.03 \times M)$ microseconds/byte. So for accessing a shared variable array of size 64K, the maximum time incurred by a channel will be about 2000 microseconds.

Surprisingly, this is the same time involved in broadcasting an equivalent array to all the 64 compute nodes. The efficiency of the cop architecture for global operations is apparent from this comparison itself. As the time incurred in read-modify-write operation is same as the read operation, the maximum time needed to update a shared integer variable (such as a lock implemented using cop memory) by a cop channel is about 2.2 microseconds. Note that the cop shared memory provides sequential consistency in that the value read by a compute node will be the last value written. Sequential consistency over large data arrays is provided using the lock and unlock operations when the libcop functions are called in urgent mode.

COP Performance Summary

Table III summarizes the performance data calculated for various libcop functions using the expressions shown and hardware timing values reported for 64 bit integer and floating point data elements for a 10ns clock period. The performance numbers shown in

Table III (for a single data element of type integer or double) assume only one participating task per channel, and a static group of tasks. The best case timing values in the table consider the case where all the channels perform integer operations without channel locking and use the average time to service a channel, $\overline{t_{OP}}$ as ($\overline{t_{OP}} = t_{OP}$). The upper bound (max) timing values in the table consider the case where all the other channels perform floating point operations with channel locking and use the average time to service a channel, $\overline{t_{OP}}$ as ($\overline{t_{OP}} = t_{MAX-OP}$). The wide variation in the best case and maximum values shows the performance degradation that can happen due to excessive channel locking. Exact timings seen by applications will vary as the average time a channel is locked depends on the communication patterns in the application (repeated broadcasts and/or shared memory reads of data arrays being the worst).

TABLE III

LIBCOP FUNCTION TIMES AND EXPRESSIONS

Function	μS *Best*	μS *Max*	Timing Expressions
cop_write for integer	2.09	1936	$\left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + \left(\frac{M}{N} \times t_{OP}\right) + t_{OP}\right] P_{CLK}$
cop_write for double	4.73	1936	$\left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + \left(\frac{M}{N} \times t_{OP}\right) + t_{OP}\right] P_{CLK}$
cop_read for integer	2.26	1936	$\left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + \left(\frac{M}{N} \times (t_{OP} + t_{COP-CN})\right) + t_{OP}\right] P_{CLK}$
cop_read for double	4.90	1936	$\left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + \left(\frac{M}{N} \times (t_{OP} + t_{COP-CN})\right) + t_{OP}\right] P_{CLK}$
cop_rmw for integer	2.26	1936	$\left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + \left(\frac{M}{N} \times (t_{OP} + t_{COP-CN})\right) + t_{OP}\right] P_{CLK}$
cop_rmw for double	4.90	1936	$\left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + \left(\frac{M}{N} \times (t_{OP} + t_{COP-CN})\right) + t_{OP}\right] P_{CLK}$
cop_bcast cop_mcast for integer	2.29	1936	$\left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + t_{OP} + \left(\frac{M}{N} \times (t_{OP} + t_{COP-CN})\right) + t_{OP}\right] P_{CLK}$
cop_bcast cop_mcast for double	4.97	1936	$\left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + t_{OP} + \left(\frac{M}{N} \times (t_{OP} + t_{COP-CN})\right) + t_{OP}\right] P_{CLK}$
cop_reduce for integer	4.21	3840	$t_{reduce} = t_{init} + t_{contrib}$ $t_{init} = \left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + t_{OP} + t_{OP} + \left(\frac{M}{N} \times t_{OP}\right) + t_{OP}\right] P_{CLK}$ $t_{contrib} = \left[\left((\overline{C} - 1) \times \overline{t_{OP}} + t_{OP} + t_{COP-CN}\right) \times \frac{M}{N}\right] P_{CLK}$
cop_reduce for double	9.45	3840	*"same as cop_reduce() for integer"*
cop_barrier (master slave model)	4.21	1943	$t_{init} = \left[\left(t_{CN-COP} + \overline{C} \times \overline{t_{OP}} + t_{OP}\right) + t_{OP} + t_{OP} + t_{OP} + t_{OP}\right] P_{CLK}$ $t_{barrier} = t_{init} + [\overline{C} \times 4t_{OP} + t_{COP-CN}] P_{CLK}$

Performance Evaluation of Libpvm Functions

In this section, I will describe the basic procedures followed to measure the performance of the PVM message passing functions, the environment in which the measurements were made, and the detailed analysis of the measurements. The performance of PVM point-to-point communication functions, collective communication functions (broadcast and multicast) and aggregate functions (barrier, global sum) are described in this section.

Measurement Methodology

Message passing performance is usually measured in units of time or bandwidth (bytes per second). In this report, I choose time as the measure of performance for sending a small message. The time for a small message is usually bounded by the speed of the signal through the media and any software overhead in sending/receiving the message. Small message times are important in synchronization and determining optimal granularity of parallelism. For large messages, bandwidth is the bounded metric, asymptotically approaching the maximum bandwidth of the media. Choosing two numbers to represent the performance of a network can be misleading, so analyzing communication time as a function of message length is the approach used to compare the performance.

Message passing time is usually a linear function of message size for two processors that are directly connected. For more complicated networks, a per-hop delay may increase the message passing time. Message passing time, t_N can be modeled as $t_N = \alpha + \beta N + (h-1)\gamma$ where α is the start-up time, β is the per-byte cost, γ is the per-hop delay, h is the number of hops the message should travel and N is the number of bytes per message. The setup for my measurements includes a collection of SUN

workstations running SunOS 4.0, interconnected using a ethernet based local area network. Measurements for cluster sizes up to 8 involves hosts only on the same subnet. All the machines are of comparable speed (Sparc 10) and can directly communicate with each other (no store and forward nodes or routers). As a result the value of h in the previous equation is 1, and so there is no per-hop delay involved, reducing the equation to $t_N = \alpha + \beta N$. A linear least-squares fit can be used to calculate α and β from experimental data of message passing times versus message length.

Point-to-Point Communication

As the COP system is primarily targeted for small sized message passing operations, to provide acceptable comparisons, latency was considered as the primary metric. A simple echo test between adjacent nodes is used to measure the latency. A receiving node simply, echoes back whatever it is sent, and the sending node measures the round trip time. Times are collected for some number of repetitions over various message sizes due to the inadequate clock resolution for small message sizes. Also, individual time values for each send and receive may show clock jitter from time-sharing interrupts in the underlying OS. The send-receive (round trip) time divided by two is what I report for latency. Figure 15 shows the latencies measured for varying messages sizes. The measurements were made with both sending and receiving tasks on same host, and on different hosts to analyze the effect of on-host latencies compared to network latency. Also measurements were done for both normal and direct routing policies. In all the cases PvmDataRaw encoding scheme is used to avoid any data packing/unpacking costs. Latencies were measured for message lengths up to 16 Kbytes as the effect of bandwidth overtakes the latency effects for large sized messages.

On-host latency is highest when normal route is used, because the message is

routed through the pvm daemon (pvmd) instead of directly from task to task, thereby resulting in an extra copy of the data buffer and an extra context switch to wake up the pvmd. `pvm_psend` which combines packing and sending is not used for measurements, as the encoding mechanism used is PvmDataRaw. Enabling direct route improves the performance almost by 50%. The default pvm fragment size is 4K bytes, so on close analysis we see a bump in latency values at multiples of 4 kilobyte message lengths when an additional fragment becomes necessary. The buffer management takes more time than one might expect. Manchek [Manch94a] has reported a pvm buffer management overhead as approximately 900 microseconds.

Inter-host latency for normal route is much worse than that of direct route because of two reasons. First, another pvmd is in the message path. Second, even though the pvmd-to-pvmd protocol allows multiple outstanding (unacknowledged) packets to be present, it only sends one packet at a time. So the bump at multiples of 4K message sizes is more noticeable with normal routes.

Point to point messages are used to pass pre-processed variables or data arrays from one task to other. From the above figure, pvm takes about 7000 microseconds to send a 8K data buffer between two hosts and for the receiving side to receive it even with direct route. The same operation can be performed using a cop_write() and a cop_read() for this 8K data buffer using the task address spaces in cop controller memory. Applying the equations from the previous section, the libcop implementation takes a total time of only about 350 microseconds (80 for cop_write and 270 for cop_read) for the same operation. Additionally any number of tasks can simultaneously read out the data buffer from the cop controller memory, once the cop_write is completed.

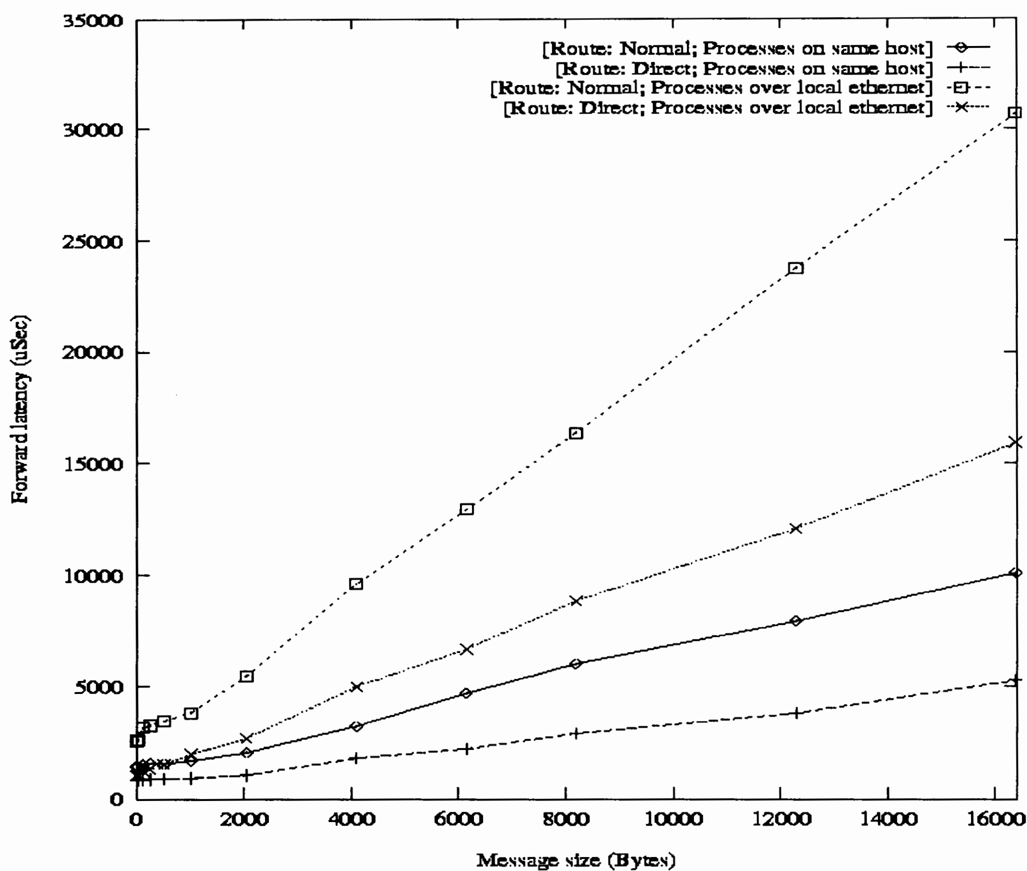


Figure 15. PVM Latency Measurements for small sized messages

In summary, UNIX machines perform poorly for small message sizes due to the latency in the kernel. Performance drops off sharply for very large messages, because the kernel is probably paging fragments of the message buffers. Performance is limited by the memory-to-memory copy speed of the processor, efficiency of the networking protocol implementation and the network hardware and media. The effects of process scheduling on communication times is not seen as dramatically when sending messages between hosts as with on-host communication. This is because the network moves data in the background and is somewhat slower than the processor, so the tasks have time to

catch up when they are scheduled back in.

PVM Collective Communication Performance

Some of the collective/aggregate communication primitives in the libpvm library considered for analysis are pvm_barrier for barrier operations, pvm_reduce for reduction operations, pvm_bcast for broadcast operations, and pvm_mcast for multicast operations. Even though barrier is a special case of reduction, it is considered separately to evaluate any implementation specific performance differences in their respective pvm functions. Similarly multicast is analyzed along with broadcast to understand any performance improvements. A set of tests were developed to exercise these collective operation functions for small-to-medium sized messages. Also the start-up latencies were considered dominant over bandwidth costs for the performance of these small sized messages. For non-blocking pvm functions (i.e. those functions that return as soon as the data is on its way to the destination), explicit short-sized acknowledgment messages were used to measure the effective time involved in these functions between the sender sending the data and the receiver receiving the same. The latency of these short (1 byte) acknowledgment message has been deducted from all data points presented. Also to make sure that all member tasks in a group call the global functions almost at the same time, a barrier was performed between the member tasks immediately before calling the global function to be measured.

Barrier Using `pvm_barrier`

Figure. 16 shows the pvm_barrier timing measurements as a function of the number of participating tasks. Measurements were made with all tasks on one host, with one task per host, and also, measurements were made for both normal and direct routing

policies.

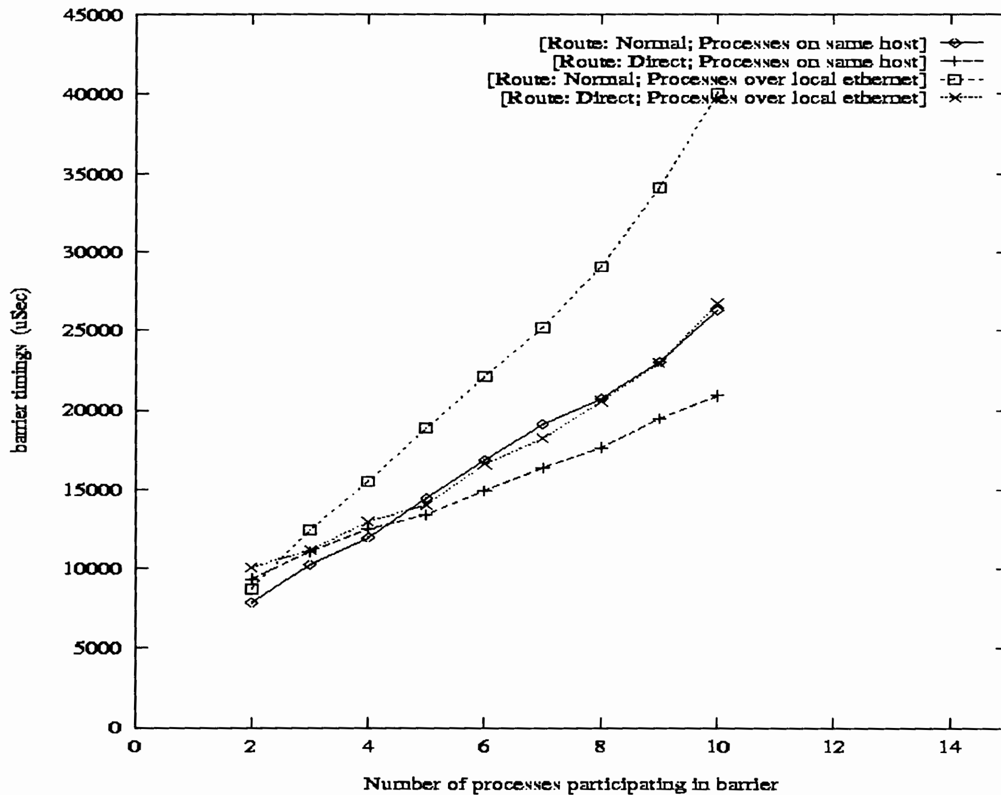


Figure 16. PVM Barrier timing measurements

Direct routing performs better than normal routing, and the performance difference is more visible for inter-host barriers when the number of processes (nodes) participating in the barrier is increased. The cop_barrier function analysis in previous section showed that we require less than 10 microseconds to perform a barrier between 64 compute nodes. As we can see, the equivalent pvm barrier is very slow consuming thousands of microseconds for performing a barrier even with 10 compute nodes.

Global Reduction Using `pvm_reduce`

Figure 17a. shows the timing measurements for a global summation using

pvm_reduce function. The data array size has only one element and is of type double, so as to compare the performance with the cop_reduce function in Table III. From the figure it is obvious that the reduction timings rise drastically almost by an order of 2, as the number of participating tasks is increased. Surprisingly, the on-host reduction for both normal and direct route took more time than inter-host reduction times with the same number of tasks. This is probably because of some synchronization mechanism used for managing the contributions from multiple tasks on the same host, and the overhead due to process scheduling and wakeups.

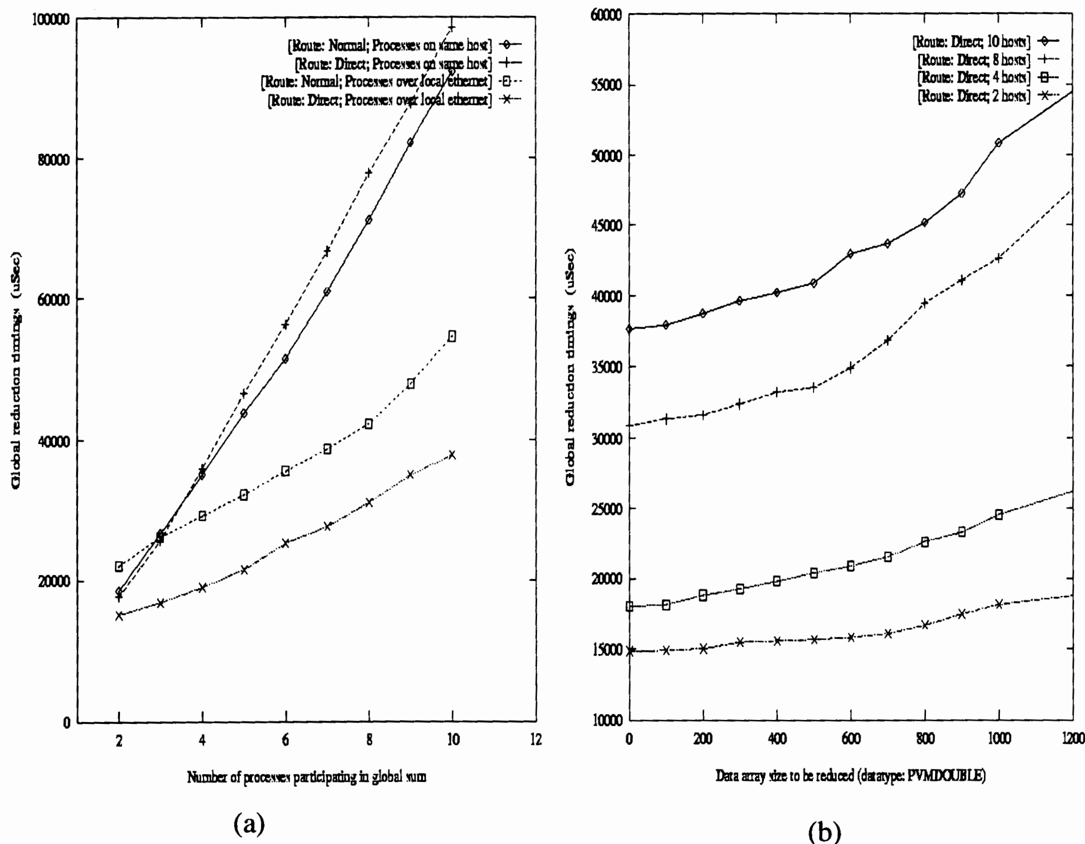


Figure 17 a, b. (a) PVM Reduction timings for varying number of participating tasks (b) PVM Reduction timings as a function of data array size

The average time for a single data element reduction between tasks on different

hosts (one task per host) is in the millisecond range for both direct and normal route when the number of participating processes were increased from 2 to 10. The corresponding time for the cop_reduce function for a reduction operation between 2 to 64 nodes is only around 10 microseconds. Figure 17b. shows the reduction timings as a function of data array size for different task group sizes. For larger number of participating processes, the reduction times increased faster for measurements using direct route compared to the same with normal route. This is probably because, with normal routing pvm daemons act as buffers. This might allow a sending task to finish sending before the receiver becomes ready, and go on to send another message.

Broadcast/Multicast Using `pvm_bcast/pvm_mcast`

Figure 18a shows the timing measurements for pvm broadcast primitive and Figure 18b shows the pvm multicast timings. PVM multicast first determines the location of all pvm daemons (pvmds) that contain the specified recipient tasks, then passes the message to these pvmds which in turn distribute the message to their local receiving tasks. So the original multicast message from the sending process is replicated for each pvmd containing one or more local destination tasks, and later each receiving pvmd replicates the message for each local destination task. The pvm_bcast function broadcasts to all member tasks in a group. The broadcast function first gets the tids of all task members in the specified group from the pvm group server, and then calls pvm_mcast() to multicast the message to all destination tasks. From the performance difference between broadcast and multicast functions, it is obvious that the communication to the pvm group server is costly. As mentioned before, the libcop library functions allow a user to declare a group as static to overcome this overhead of communicating with the group server. The scalability of the pvm multicast feature is limited due to the 1:N direct fanout which results in heavy contention of

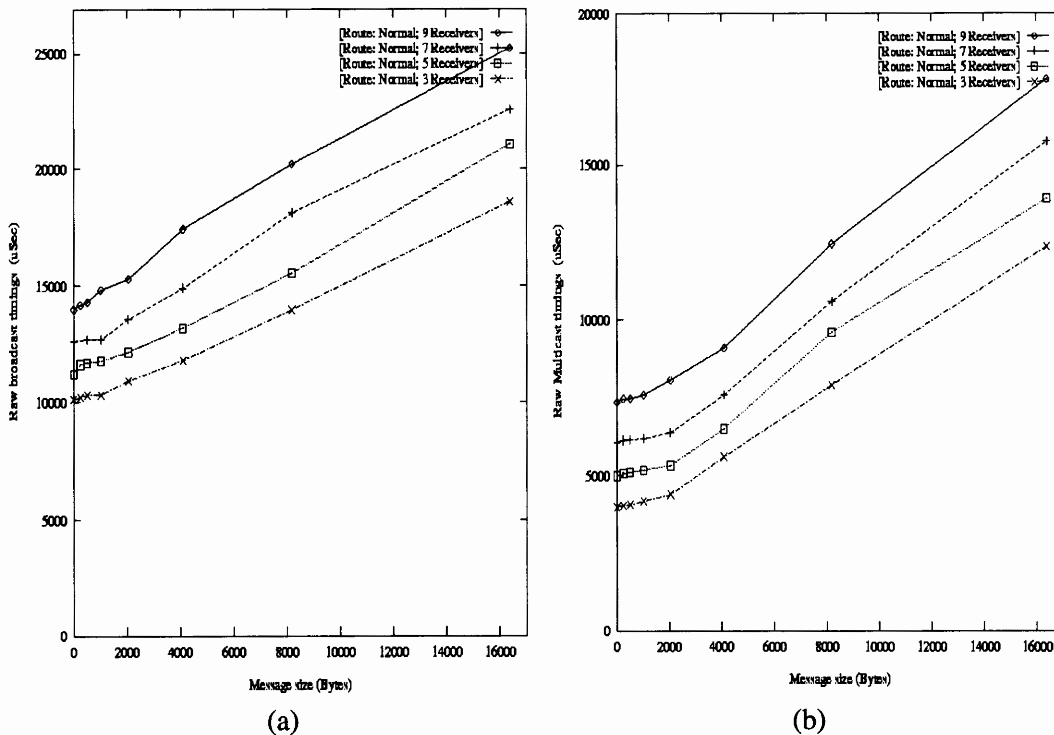


Figure 18 a, b. (a) PVM Broadcast timings as a function of message size
(b) PVM Multicast timings as a function of message size

acknowledgment messages. The performance measurements provided above for multicasts and broadcasts matches the same reported by other research groups [Chang95a]. From the expressions derived in the previous section for cop broadcast and multicast shows that the latency for them are less than 10 microseconds compared to the millisecond range performance of pvm functions. Chang [Chang95a] has reported that using their PVM-ATM (AAL5) re-implementation, when increasing the number of remote hosts of the receiving pool from 1 to 4 the largest multicast time difference observed is approximately 20 milliseconds. Even though the multicast performance over ATM (AAL5) is better than the original pvm multicast, it is still far below from the performance predicted for the cop multicast primitives as it still needs to traverse the protocol stacks.

PVM Performance Summary

The communication performance of pvm leaves a little to be desired. The major reason for the performance limitation is the heterogeneous nature of the software itself. The performance is affected by three major factors. First, the pvm application protocol drivers run in user space (in the pvm daemons and tasks). The pvmd-to-pvmd protocol suffers most, because it manages timers and resend queues. It is expensive to read timers from user space because it must be done using system calls. Manchek [Manch94a] has reported that a sample profiling of pvmd showed 10% of its time spent in the function gettimeofday(). Performance might improve if UDP could be replaced with a protocol with reliable delivery, eliminating the need to resend packets. The time-of-day clock could be mapped to user memory (same way the cop channel interface registers are mapped) to eliminate system calls. The pvmd-to-task (also task-to-task) protocol is based on TCP and so does not require any timers. Also as both pvmd and tasks maintain a number of connections, a large fraction of time is spent on select() calls, multiplexing different inputs. The COP system gains significant performance in this regard, as the cop channels are independent and not general purpose.

Secondly, even though the pvmd-to-pvmd protocol allows multiple outstanding packets, the pvmd sends them only one-by-one. On a high-bandwidth network, a single packet is not enough to keep the communication pipe (media) full. Therefore, the pvmd-to-pvmd communication speed is limited by network latency and bandwidth, instead of just bandwidth.

Thirdly, the message data is copied a number of times before it reaches the destination [Sten94a]. Normal message routing (through pvmds) incurs five copies: The data must be packed, routed through four processes (three copies), and finally unpacked. Direct routing improves over that (three copies total) since the message is sent directly

between tasks, but the cost to establish a route is high because the request and acknowledgment message travels via default route. DataInPlace encoding eliminates one more copy by leaving data in place until send time. Also, some scatter and gather operations such as spawning tasks and multicasting do not scale well because the communication uses a 1:N fanout. Acknowledgments tend to come back all at once, swamping the central host and causing it to drop packets which then have to be retransmitted.

Table IV provides an easy comparison of the libcop and libpvm library function costs. The libpvm times provided is for a sample group size of 5, with no packing or unpacking costs and using direct route when ever possible. The libcop timings provided represent a maximum cluster size of 64 without any channel locking. Performance improvements by using the libcop functions is limited if the channel locking is used excessively. Having demonstrated the performance improvement in implementing global operations using the COP system, I will now describe the projected effect of this performance difference on overall execution time of applications.

TABLE IV

LIBCOP AND LIBPVM PERFORMANCE COMPARISONS

Operation	libcop μS	libpvm μS
16K Broadcast using `cop_bcast/pvm_bcast`	500	17000
16K Multicast using `cop_mcast/pvm_mcast`	500	12000
Barrier using `cop_barrier/pvm_barrier`	10	10000
Reduction of 1024 element array of type double using `cop_reduce/pvm_reduce`	4800	19000
16K point-to-point message using `cop_write & cop_read/pvm_send & pvm_recv`	500	15000

Effect of Global Operation Speedup on Overall Execution Time

In the preceding sections, I have shown that the COP system provides substantial speedup for common global operations on workstation cluster multicomputers. As discussed before, the COP system is most efficient for parallel operations that involve collective communications between a group of tasks. In this section, I will describe the effect of this performance improvement of global operations on an example real life application. I chose a complex molecular dynamics simulation PVM application as an example to analyze the performance difference brought by using a COP system. In the next section I will describe the sample application program and analyze the improvement in execution time by using a COP system.

EGO - A Molecular Dynamics Simulation Program

'EGO' is a parallel molecular dynamics simulation program that was originally developed at Beckman Institute, University of Illinois, Urbana. The program was originally written in the OCCAM language to run on transputers. For my work, I have used a port of the above program [Heller91a] that runs on top of the PVM message passing system on a network of workstations.

The fact that most of the pvm global functions are inefficient was very clear in that they were little used in most of the complex applications that I considered. Some of the applications like the above mentioned molecular dynamics application, replaced the pvm global reductions with point-to-point messages in a logical ring topology formed between the tasks in the virtual machine. Using this method, each task has to send only one message to the task above it in the logical ring, and has to receive only one message from the task below in the ring, causing less contention. The global reduction advances slowly as the message leaves each task in the ring. But if any type of data encoding

needs to be used, the encoding/decoding cost at each task will take away some or all of the performance. Further, due to the broadcast nature of ethernet, each message in the logical ring is serialized on the network. Some applications may be able to overlap computation and communication times using this strategy, but it is very cumbersome to implement a global reduction in this way.

The EGO source was instrumented to collect performance data for various parallel program operations. A debug library was developed to wrap the libpvm and libgpvm functions so as to collect the time spent in these pvm functions. Also the logical ring method for performing global reductions was replaced by the pvm reduction function `pvm_reduce` with the control task as the root, as most of the applications I surveyed used the standard pvm reduction function and not the logical ring method for global reductions. Due to the massive resource requirements of the program, I was able to run the program on only a limited range of data sets. Also as the COP timing expressions are derived assuming one task per host, to make comparisons fair only one task per host was spawned to run the program.

The program uses a modified verlet algorithm [Heller90a] to evaluate the various Newtonian forces. The program repeatedly computes the total force on each atom and then use Newton's laws of motion to determine the new position and velocity for each atom. The different major parts of the program are non-bonded force calculation, pairlist generation, shake, bonded force calculation, global reduction (sum) of force contributions and finally load balancing. The program runs in a master-slave model, with one controlling task and N slave tasks. The controlling node is responsible for spawning the slave tasks, data initialization/distribution, analysis of globally reduced data, and controlling computational cycles of slave tasks. Next, I will provide a brief description

of the major operations performed by both the control and slave tasks.

Control Task Operation

The control task in the EGO application performs the following parallel program operations:

1. After all the slave tasks are spawned, the control task multicasts the task management and initialization information to all the slave nodes using `pvm_mcast`. This include the number of slave tasks spawned, slave task tids, data encoding mechanism to be followed and the routing policy.

2. Next the control task sends the loading information to each slave task. As the atoms loaded to each slave task is different, the control task performs N `pvm_send` calls to communicate with the N slave tasks.

3. Later, the control task multicasts the global data needed to perform the computational cycle to all the slave tasks again using `pvm_mcast`.

4. After the initialization phase the control task sits in a big loop, receiving travelling co-ordinates moving in the logical ring, and travelling forces multicast from each slave task. In each cycle the control task performs a global reduction of the energy contributions between all the slave tasks, and multicasts the result to all the slave tasks.

5. The control node analyzes the updated atom positions and velocities for convergence to determine whether to proceed with another computation cycle or not. If the computation cycle needs to be continued the control task multicasts a 'running' flag to signal the slave tasks to continue.

Slave Task Operation

The slave tasks in the EGO program performs the following parallel program operations:

1. First, it receives the task management information, initial atom co-ordinates and other initialization variables multicast and/or unicast from the control node.

2. After the initialization phase, each slave task starts the computation cycle. First it computes the interaction between its 'own' atoms and sends the new atom co-ordinates computed to the node below in the logical ring, and receives the new set of co-ordinates from the node above in the logical ring.

3. Each slave communicates the force information between all slave tasks using barriers and all-to-all multicasts, so that the interactions between local and foreign atoms can be computed.

4. All the slave tasks take part in the global reduction of energy contributions with the control task as the root. Later, it receives the reduced energy information multicast from the root (control) task.

5. Finally, all the slave tasks wait for the 'running' flag from the control node to determine whether to continue with another computation cycle or not.

Performance Comparisons

The sample runs for the EGO program were performed for an 'alanin' molecule with 66 atoms, for 256 integration steps, with almost equal loads on all the virtual machine nodes. Virtual machine sizes ranging only from 2 to 5 was experimented due to the heavy use of system and network resources by the program. The average message-size for these global operations was less than 16 kilobytes, which makes them applicable to be performed efficiently over the COP channels. For a virtual machine size of 5, on the average, 40% of the total execution time on both the slave and control tasks was spent on the pvm communication functions. Analysis of the timing information collected showed that multicasts accounted for 16% of total execution time, global reduction

accounted for 14% of the execution time, barrier operations accounted for 4% of execution time and point to point communication functions accounted for 6% of the total execution time, aggregating to the 40% total communication time. Note that, as the virtual machine size is increased the collective communication function costs rise drastically as shown in the early sections of this chapter.

The PVM functions in this application can easily be replaced by equivalent libcop functions so as to use the secondary network for performing global operations. The multicast and shared memory read operations were serialized in the master and slave tasks by barrier constructs in this application. As a result, only one channel performed broadcast/multicast operation at a time. Due to this reason, the average time to service a channel request ($\overline{t_{OP}}$) is assumed to be almost equal to t_{OP}.

As shown in Table IV, for a virtual machine size of 5 the libcop multicast performs about 24 times more efficiently than the pvm multicast. Similarly, from Table IV one can see that the libcop reduction performs about 4 times faster than pvm reduction, libcop barrier performs about 1000 times more efficiently than pvm barrier and point-to-point communication implemented using COP controller memory shows a improvement of 30 times. Applying these improvement factors to the above global operation execution times, the total time spent on collective communication operations can be reduced by a factor of about 9. This results in an improvement in the overall execution time of the application by more than 1.5 times. As the libcop global reduction function can send the reduced result to any number of participating tasks at no extra cost (using the broadcast masks), the number of multicast operations required while using the COP system will be reduced, further adding to performance improvement. Table V summarizes the timing analysis performed on the EGO application.

TABLE V

TIMING ANALYSIS OF MOLECULAR DYNAMICS APPLICATION

Operation	libpvm normalized timings	speedup	libcop normalized timings
Multicast	0.16	24	0.00700
Global Reduction	0.14	4	0.03500
Barrier	0.04	1000	0.00004
Point-to-Point	0.06	30	0.00200
Total Communication Time	0.40		0.04404
Total Computation Time	0.60		0.60000
Total Execution Time	1.00		0.64404

As shown previously, as the virtual machine size increases the cost of the pvm collective communication functions rises sharply. But the overhead of libcop functions is almost the same for cluster sizes of up to 64 compute nodes if excessive channel locking is avoided. Thus the COP system provides better scalability for application programs.

Summary

A final note here is that, the COP system is intended for message passing operations with small-to-medium sized messages. The latency for various operations over the cop channels is very low due to several reasons including no overhead for packet assembly/re-assembly, direct protected access to the network interface, ability to broadcast simultaneously through all channels etc. But the performance decreases gradually for long messages. Some of the major reasons for the performance degradation at large message sizes are the half-duplex nature of the serial link, low utilization of the channels (out of the four words in a cop message, only two words contain data resulting

in 50% overhead), and the low payload in cop messages (need a cop message for each data element). Also, excessive use of channel locking by the compute nodes will disturb the fair sharing of COP controller resources by all the channels. Therefore, the benefit seen by applications will vary depending on the size and pattern of message-passing performed by the application. Application with small-to-medium message sizes and high percentages of collective operations will most likely see the highest performance improvement. The high performance difference and the low cost nature of the system makes it very attractive and promising for application to workstation cluster multicomputers.

CHAPTER VII

RELATED WORK

The expressions in Table III shows that the COP system can significantly reduce the execution time for several common global operations on small-to-medium sized messages. In the last chapter I discussed and showed an example application that can benefit from using the COP system. Also I showed that message-passing systems can use the cop controller's task address spaces to minimize the latency on small synchronization messages. In this chapter I describe and compare other proposed or practiced methods for decreasing the global operation and/or latency times. As my work was primarily to investigate the performance aspects of using a COP system on workstation cluster multicomputers, I will concentrate on the related work for these class of machines.

Cluster Computing Over High Speed Networks

One of the factors which caused much skepticism on the feasibility of network-based parallel computing was the limitation imposed by using traditional local area networks, such as an Ethernet, as the system interconnect. For many typical network applications which require only infrequent small amounts of data to be transmitted between workstations, an ethernet based cluster is adequate. However, for network based applications, such as communication intensive, course grain parallel applications it has been proved that the bus based networking technology cannot provide acceptable performance [Chang95a]. Some of the motivational factors for considering the

implementation of a parallel computing platform over a high speed local area network are:

1. High speed switch-based network architectures, such as the High Performance Parallel Interface (HIPPI) and Asynchronous Transfer Mode (ATM) feature aggregate throughputs of several gigabits/sec. Moreover each host usually has a dedicated high-speed connection to the switch, unlike shared medium architectures where the network capacity is shared among all the interconnected processors.

2. High-speed switch-based networks may easily be scaled up, in terms of processing power by adding additional switches/links.

3. Inherent features, such as dedicated connections, of switch-based high-speed networks allow them to support low latency data transfers.

4. Switch based networks inherently support efficient multicasting, and thus may be attractive for supporting paradigms like distributed shared memory, where multicast operations are frequently used to update, lock and unlock multiple data copies.

All these advantages of high-speed networks led researchers to develop and implement switch-based cluster interconnects. Objectives for most of the works discussed below are to reduce message latency or to support efficient collective communications, even though they may follow different programming models.

The Princeton Shrimp

One cluster multicomputer system that uses a unique network interface to reduce message latency is the Princeton Shrimp system described by Li [Blum94a]. Shrimp is actually an example of distributed shared memory system. The Shrimp system uses an Intel Paragon router backplane to implement its interconnection network. In Li's algorithm, known as Shared Virtual Memory (SVM), the shared address space is

partitioned into pages, and copies of these pages are distributed among the hosts, following a Multiple-Reader/Single- Writer (MRSW) protocol. Pages that are marked read-only can be replicated and may reside in the memory of several hosts, but a page being written to can reside only in the memory of one host.

One advantage of Li's algorithm is that it can easily be integrated into the virtual memory of the host operating system. If a shared memory page is held locally to a host, it can be mapped into the application's virtual address space on that host and therefore be accessed using the normal machine instructions for accessing memory. An access to a page not held locally triggers a page fault, passing control to a fault handler. The fault handler then communicates with the remote hosts in order to obtain a valid copy of the page before mapping it into the application's address space. When a page fault occurs due to a write access, the fault handler has to invalidate all the other copies in the system before marking the local copy as writable and allowing the faulted process to continue. As a result the model achieves functional transparency, in the sense that a program written for a shared memory multiprocessor system can run without change. On the other hand performance transparency can only be achieved to a certain degree, as the physical locations of the data being accessed affect application performance which is not entirely true in a shared memory multiprocessor machine.

But as reported by Li, the virtual shared memory model depends largely on the performance of the collective communication primitives like multicast operations as they are frequently used to update, lock and unlock multiple data copies. Since the backplane has a 2-D mesh topology, spanning trees are needed to implement broadcast and other global operations. Not only do message latencies cascade along the branches of these trees, but also multiple operating system calls are required to set up the mapping for the trees before the global operation and to restore the previous communication link

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *"PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing,"* MIT Press, 1994.

Gupta94a.
Anshul Gupta, George Karypis, Vipin Kumar and Ananth Grama, *Introduction to Parallel Computing, Design and Analysis of Algorithms,* The Benjamin/ Cummings Publishing Company, Inc., Redwood City, CA, 1994.

Hall94a.
Douglas. V. Hall, "Hardware for Fast Global Operations on Distributed Memory Multicomputers and Multiprocessors," *Dissertation for the degree of Doctor Of Philosophy,* Portland State University, December 1994.

Hall95a.
Douglas V. Hall and Michael A. Driscoll, "Hardware for Fast Global Operations on Multicomputers," *Proceedings of the 9th International Parallel Processing Symposium,* pg. 673-680, IEE Computer Society Press, Los Alamitos, CA, April 25-28, 1995.

Harr91a.
R. J. Harisson, "Portable tools and applications for parallel computers," *International Journal on Quantum Chemistry,* 40:847-863, 1991.

Heller91a.
H. Heller, A. Windemuth, H. Grubmuller, and K. Schulten, "Generalized Verlet Algorithm for Efficient Molecular Dynamics Simulations with Long-Range Interactions," *Molecular Simulation,* 1991, Vol. 6, pg. 121-142.

Heller90a.
H. Heller, H. Grumbmuller, and K. Schulten, "Molecular Dynamics Simulation on a Parallel Computer," *Molecular Simulation,* 1990, Vol. 5, pg. 133-165.

Huang95a.
C. Huang, Yih Huang, and Philip K. Mckinley, "A Thread-Based Interface for Collective Communication on ATM Networks," *15th International COnference on Distributed Computing Systems,* Vancouver, Canada, May 30 - June 2, 1995.

Kung91a.
H.T. Kung et al., "Network-Based Multicomputers: An Emerging Parallel Architecture," *Proceedings Supercomputing 91 Conference*, ACM/IEEE, Alberquerque, NM, 1991.

Manch94a.

Robert J. Manchek, "Design and Implementation of PVM Version 3," *MS Thesis*, University of Tennessee, Knoxville, TN, May 1994.

Manch95a.

Robert J. Manchek, 1995. Personal communication.

Matts93a.

T.G. Mattson, C.C. Douglas, and M.H. Schultz, "Parallel Programming Systems for Workstations Clusters," Yale University Computer Science Department, *Technical Report*, YALEU/DCS/TR-975, 1993.

Mlin94a.

Menjou Lin, Jenwei Hseih, et al., "Distributed Network Computing over Local ATM Networks," *Proceedings Supercomputing '94,* pg. 154-163, IEEE Computer Society Press, Washington, D.C, November 14-18, 1994.

Mull90a.

S. Mullender, Guido van Rossum, Andrew Tanenbaum, et al., "Amoeba: A distributed operating system for the 1990s," *IEEE Computer,* 23(5):44-53, May 1990.

Ostc95a.

U.S. Office of Science and Technology Committee on Physical, Mathematical, and Engineering Sciences, "High Performance Computing and Communications: Foundation for America's Information future," *(FY 1996 Blue Book),* 1995.

Para94a.

Paragon(TM) System C System Calls Reference Manual.

Saaved93a.

H. Saveedra, R. S. Gaines, and Micheal J. Carleton, "Micro Benchmark Analysis of the KSR1," *Proceedings Supercomputing '93 Conference, pg. 202-213,* ACM/IEEE, Portland, OR, November 1993.

Sgupt94a.

Satya Gupta, M. Barnett, D. Payne, L.Shuler, R. van de Geijn, and J. Watts, "Interprocessor Collective Communication Library (InterCom)," *Proceedings of Scalable High Performance Computing Conference,* pg. 357-364, IEEE Computer Society Press, Knoxville, TN, May 23-24, 1994.

after the operation. The idea of providing protected direct access to the network hardware in Shrimp is similar to the direct mapped access to COP system channels. Both systems try to reduce the extra software overhead by offering direct access to the network, but as the COP channels are independent there is no overhead of traversing the spanning trees to do global operations.

Similar distributed shared memory (DSM) models atop ATM networks also exist. Thekkath et al. [Theka93a] have reported very promising results for emulating distributed shared-memory across over ATM LAN, using a remote procedure call paradigm. Dwarakadas et al. [Dwark93a] have studied software implementations of distributed shared memory on ATM network which points out the need for fast synchronization and multicast in order to support release consistent software DSM protocols.

Improving PVM performance using ATOMIC system

Enhancing the performance of existing popular programming models by using emerging networking technologies is also a very common approach. This allows all the applications using the original programming model to increase performance without any significant re-design. The COP system is designed to work along with the PVM message passing model to take advantage of the above aspects. One similar work is reported by Fisher [Fishr95a] for improving the performance of PVM using the ATOMIC user-level protocol. The ATOMIC LAN is a high-speed network that offers 640-Mbps bandwidth at an inexpensive per-host cost. It is a switch-based local area network composed of host interface boards and network switches and uses a source-routed cut-through packet switching technique. The performance improvement is achieved by separating the pvm data-transmission path from the pvm control-message path, and transmitting pvm data

messages over a user-level application programming interface (API) provided by the Myrinet ATOMIC interface. The system uses the idea of interleaving memory copies with DMA operations on the sender to reduce memory copy overhead. Even though, the implementation has reported reducing the PVM latencies times by half, the performance for small-to-medium sized messages is still far below that offered by the COP system. Also, the Myrinet-API, although significantly faster than the TCP/IP kernel stack, does not provide reliable communication. Therefore, the application programmer has to provide a user-level protocol to offer reliable sequenced packet delivery. The channels in a COP system provide reliable data transfer, and hence application programmer do not have to deal with providing a reliable user level protocol.

Faster Message Passing in PVM using ATM

Several attempts have been made by researchers to enhance the communication facilities of PVM by using the Asynchronous Transfer Mode (ATM) technology. ATM networks are characterized by their switch-based architecture instead of the bus based architecture of the first and second generation networks. A switched network is capable of supporting multiple connections simultaneously and multiple messages can be transmitted across the network concurrently. One of the effort to enhance pvm communications reported by Geist [Geist95a] is by using ATM's lower layer API to implement a faster message passing route in pvm. The Fore Systems ATM API based library functions provides a connection-oriented client-server model. After a connection is set up, the network makes a 'best effort' to deliver ATM cells to the destination. But cells may be dropped during the transmission depending on the availability of the resources. So unlike the connection-oriented TCP socket, flow control and retransmission facilities have to be provided by the applications. Geist's work represents

only the performance of point to point communications. He has reported a bandwidth improvement of AAL5-based routes (PvmRouteAtm) by about 57% compared with the TCP/ATM based PvmRouteDirect route. But average latency times of about 3200 microseconds was incurred by this route due to the inefficiency of the acknowledgment procedure required for reliable sequenced packet delivery. This makes them less applicable for short synchronization messages where latency is the performance limiting metric.

Collective Communication on ATM Networks

Collective operations are usually defined in terms of a process group, and include broadcast, scatter, gather, global operations across distributed data, and synchronization. Huang et al. [Huang95a] reports a software framework for implementing communication operations on ATM networks. The approach is based on reliable one-to-many connections which are implemented atop unreliable ATM multicast virtual channels. The system uses a thread-based interface for collective communications. Their experimental setup includes a cluster of Sun SPARC stations interconnected by Fore Systems ASX100 ATM switches. The switch fabric of these first generation ATM switches is a 1.2 Gbps time-division-multiplexed bus. The bus-based switch fabric provides cell-by-cell replication of messages to multiple output ports, thereby implementing multicast. The Fore system software includes an Application Program Interface (API) that allow user-level processes direct access to AAL5 software. The Fore API also provides functionality that cannot be efficiently implemented with TCP/IP messages, such as the above mentioned hardware multicast. Multiple threads manage ATM virtual channels for data and acknowledgments on behalf of the application process. Given a connection-oriented unreliable multicast service, the reliable collective

operations are realized with a "combining-tree" among the destinations, which performs a reduction operation on the acknowledgments as they proceed towards the source node. Data transmitted on the multicast channel arrives at the destinations nearly simultaneously, due to the pipelining of ATM cells. The acknowledgment channels are arranged as a spanning binomial tree. An interior node in the tree waits for acknowledgments from its children before forwarding an acknowledgment to its parent. In essence, a reduction operation is performed on the acknowledgments, with the reduction operation being minimum. Since this strategy introduces log (N + 1) - 1 additional delays for N destinations, a sliding window protocol is used to improve the throughput. The average completion time for a 8-node multicast in the original PVM implementation executed over ATM is about 160,000 microseconds. Even though this implementation reported a multicast completion time 15 times less than the above mentioned time for PVM, it is still far behind the time for an equivalent COP multicast. Also in the COP system, the cost remains constant for multicasting to any number of channels (maximum of 64) on a single controller. Another point is that, in combining-tree acknowledgment patterns, a failure at one node will cause the multicast reported as failed for all the nodes sending acknowledgments to the failed node. But in a COP system as the communication to each node is independent, the multicast fails only for the node that crashed, providing better fault tolerance. Also, unlike the COP system ATM based schemes do not support fast aggregate operations.

Many similar research projects are currently investigating the use of high-speed switch based networking technologies to reduce latency and to provide efficient and reliable collective communications. Also similar interesting works are going on to come up with better performance metrics and benchmarks for collective communication operations.

CHAPTER VIII

CONCLUSIONS

The analyses in Chapter VI conclusively demonstrate that the COP system can improve the performance of global operations on workstation clusters by factors of 5-25 over PVM. Furthermore, the analysis of the sample application program in Chapter VI shows that speeding up these operations decreases overall execution time of this type of PVM applications by more than 1.5 times.

The COP system software libraries are implemented to work along with the PVM message passing system. This was done to make use of some of the superior features of the PVM model like task management, asynchronous event notification, process groups, etc. and to make the COP system resources readily available for numerous existing PVM applications without significant rework. Also following an existing popular programming model allows future applications to be easily written to make use of the COP system. Porting application programs to use the COP resources is as simple as replacing any PVM functions with equivalent COP library functions and linking the program with the libcop library. The modular design of the COP system software allows easy porting to other message passing systems by replacing the libcop library functions.

From a software perspective, the low overhead for accessing the channel is provided by allowing direct protected access to the COP network interfaces. The network interface is mapped to the application task's address space, allowing it to access the COP controller memory locations by regular read and write machine instructions. This implementation can also be used to provide a virtual shared memory model similar

to the Shrimp system, but using the shared memory on the COP controller. The timing expressions derived in Chapter VI shows that the COP system performance is far ahead as far as latency and collective operations are considered, compared to the native PVM functions. Also unlike the ATM based schemes described before, the transmission timers and retry mechanisms are implemented in compute node interface hardware, so application programs need not worry about implementing any reliable user-level protocol over this. Also as the communication channels are not shared, there is no time spent in framing the packets before transmission and extracting the application data from packets upon reception. This saves a considerable amount of time that otherwise would be spent on buffer management.

Suggestions for Further Performance Improvements

The COP system provides many benefits, but it also has some limitations. The primary limitation is that it is applicable to only small-to-medium sized messages. This is because the effective payload in each COP message is very low. In a COP message containing 4 words, only 2 words contain application data. Even though this is negligible in small messages, for large message sizes, the overhead involved takes away any performance improvements. Comparing the COP library timing values for various global operations with the equivalent functions in Interprocessor Collective Communication (InterCom) library and the NxLib library [Geijn95a], I found that the COP system is best suited for messages below 16 kilobytes and performs moderately for message sizes between 16 kilobytes and 64 kilobytes.

The COP system libraries use shared memory and semaphore mechanisms to perform local reductions, synchronization and broadcast/multicast between the tasks residing on the same compute node, before sending the reduced contribution from the

compute node to the cop controller. Even though, this reduces the contention for the COP channel by different local tasks, the local reduction performance depends on the efficiency of the implementation of IPC mechanisms on the host's operating system. The shared memory approach for inter-process communication turned out to be very inefficient compared to other IPC mechanisms. The major reason for this is the overhead in semaphore operations and the extra context switching with these operations. Geist [Geist95a] reported similar problems when he tried a shared memory route to replace the TCP based PvmRouteDirect for local task to pvmd communications. He suggests the use of Unix domain sockets instead of internet domain TCP connection between local task-to-task and task to pvm daemon communications. In PVM version 3.3, the internet domain TCP socket connection has been replaced by the Unix domain socket connection as default connection. One of the possible improvements to the COP software design is to replace the shared memory/semaphore mechanism with other IPC mechanisms. I expect significant performance improvement using a thread based management for the cop channels, but this is not possible in the current implementation as the generic port of PVM does not support threads. Excessive use of channel locking features to perform atomic operations on data arrays in the COP controller RAM disrupts the fair sharing of cop controller resources by the channels, limiting the performance improvements seen by applications. Also the current COP system software implementation does not support multiple concurrent users to access the COP controller resources. To implement this facility (which PVM supports) with less overhead, the COP controller hardware may need to be modified to increase its programmability.

The COP system described here presents many potential research topics. Also as the system does not change the basic programming paradigm, it can support a variety of programming models and machine architectures. I feel the work described here for

applying the COP system to workstation clusters will provide the motivation and starting point to actually design and implement a prototype system, measure and compare the performance of real applications and open an active topic for interested researchers.

REFERENCES

Begu93a.
A. Beguelin, J. Dongarra, A. Geist, R. Mancheck, S. Otto and J. Walpole, "PVM: Experiences, Current Status and Future Direction," *Proceedings Supercomputing '93,* IEEE Computer Society Press, Portland, OR, November 1993.

Blum94a.
M. Blumrich, K.Li, R. Alpert, C. Dubnicki, et al., "A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer," *Proceedings of the 21st Annual Intl. Symposium on Computer Architecture*, April 1994.

Butl92a.
R. Butler and E. Lusk, "User's guide to the p4 programming system," *Technical Report ANL-92/17*, Argonne National Laboratory, Argonne, IL, 1992

Case93a.
David A. Case, "Computer Simulations of Protein Dynamics and Thermodynamics," *Computer,* pg. 47-57, October, 1993.

Chang95a.
Sheue-Ling Chang, David H. C. Du, et al., "Enhanced PVM Communications Over a High-Speed LAN," *IEEE Parallel & Distributed Technology,* pg. 20-32, IEEE Computer Society Press, vol. 3, Number 3, Fall 1995.

Clark92a.
Terry W. Clark, et al., "Evaluating Parallel Languages for Molecular Dynamics Computations," *Proceedings of Scalable High Performance Computing Conference,* Williamsburg, Virginia, April 26-29, 1992.

Cohen94a.
D. Cohen et al., "ATOMIC: A High-Speed Local Communication Architecture," *Technical Report*, USC/Information Sciences Institute, Marina del Rey, CA, January 94.

Dwark93a.
S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel, "Evaluation of release consistent software distributed shared memory on emerging network technology," *Proceedings of the 20th International Symposium on Computer Architecture,* pg.

144-155, May 1993.

Ewing93a.
R. E. Ewing, et al., "Distributed Computation of Wave Propagation Models Using PVM," *Proceedings Supercomputing '93,* pg. 22-31, IEEE Computer Society Press, Portland, OR, November 1993.

Fatoo94a.
Rod Fatoohi, and Sisira Weeratunge, "Performance Evaluation of Three Distributed Computing Environments for Scientific Applications," *Proceedings Supercomputing '94*, pg. 400-409, IEEE Computer Society Press, Washington, D.C, November 14-18, 1994.

Fishr95a.
Hong Xu and Tom W. Fisher, "Improving PVM Performance using ATOMIC User-level Protocol," Proceedings of the first international workshop on high-speed network computing, pg.108-117, IEEE Computer Society Press, Santa Barbara, CA, April 25, 1995.

Flow91a.
J. Flower, A. Kolawa, and S. Bharadwaj, "The Express way to distributed Processing," *Supercomputing Review*, pg. 54-55, May 1991.

For93a.
MPI Forum, "MPI: A message passing interface," *Proceedings of Supercomputing '93,* pg. 878-885, Los Alamitos, CA, 1993, IEEE Computer Society Press.

Geijn91a.
R. A. van de Geijn, "Efficient Global Combine Operations," *Sixth Distributed Memory Computing Conference Proceedings,* pg. 291-294, IEEE Computer Society Press, 1991.

Geijn95a.
R. A. van de Geijn et al., "InterCom: Lean Mean Collective Communication Machine," *Intel On-Line* March 1995.

Geist95a.
Al Geist, and Honbo Zhou, "Faster Message Passing in PVM," *Proceedings of the First International Conference on High-Speed Network Computing,* April 25, 1995 Santa Barbara, CA, pg. 67-73, IEEE Computer Society Press.

Geist94a.

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *"PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing,"* MIT Press, 1994.

Gupta94a.

Anshul Gupta, George Karypis, Vipin Kumar and Ananth Grama, *Introduction to Parallel Computing, Design and Analysis of Algorithms,* The Benjamin/ Cummings Publishing Company, Inc., Redwood City, CA, 1994.

Hall94a.

Douglas. V. Hall, "Hardware for Fast Global Operations on Distributed Memory Multicomputers and Multiprocessors," *Dissertation for the degree of Doctor Of Philosophy,* Portland State University, December 1994.

Hall95a.

Douglas V. Hall and Michael A. Driscoll, "Hardware for Fast Global Operations on Multicomputers," *Proceedings of the 9th International Parallel Processing Symposium,* pg. 673-680, IEE Computer Society Press, Los Alamitos, CA, April 25-28, 1995.

Harr91a.

R. J. Harisson, "Portable tools and applications for parallel computers," *International Journal on Quantum Chemistry,* 40:847-863, 1991.

Heller91a.

H. Heller, A. Windemuth, H. Grubmuller, and K. Schulten, "Generalized Verlet Algorithm for Efficient Molecular Dynamics Simulations with Long-Range Interactions," *Molecular Simulation,* 1991, Vol. 6, pg. 121-142.

Heller90a.

H. Heller, H. Grumbmuller, and K. Schulten, "Molecular Dynamics Simulation on a Parallel Computer," *Molecular Simulation,* 1990, Vol. 5, pg. 133-165.

Huang95a.

C. Huang, Yih Huang, and Philip K. Mckinley, "A Thread-Based Interface for Collective Communication on ATM Networks," *15th International COnference on Distributed Computing Systems,* Vancouver, Canada, May 30 - June 2, 1995.

Kung91a.

H.T. Kung et al., "Network-Based Multicomputers: An Emerging Parallel Architecture," *Proceedings Supercomputing 91 Conference*, ACM/IEEE, Alberquerque, NM, 1991.

Manch94a.

Robert J. Manchek, "Design and Implementation of PVM Version 3," *MS Thesis*, University of Tennessee, Knoxville, TN, May 1994.

Manch95a.

Robert J. Manchek, 1995. Personal communication.

Matts93a.

T.G. Mattson, C.C. Douglas, and M.H. Schultz, "Parallel Programming Systems for Workstations Clusters," Yale University Computer Science Department, *Technical Report*, YALEU/DCS/TR-975, 1993.

Mlin94a.

Menjou Lin, Jenwei Hseih, et al., "Distributed Network Computing over Local ATM Networks," *Proceedings Supercomputing '94,* pg. 154-163, IEEE Computer Society Press, Washington, D.C, November 14-18, 1994.

Mull90a.

S. Mullender, Guido van Rossum, Andrew Tanenbaum, et al., "Amoeba: A distributed operating system for the 1990s," *IEEE Computer,* 23(5):44-53, May 1990.

Ostc95a.

U.S. Office of Science and Technology Committee on Physical, Mathematical, and Engineering Sciences, "High Performance Computing and Communications: Foundation for America's Information future," *(FY 1996 Blue Book),* 1995.

Para94a.

Paragon(TM) System C System Calls Reference Manual.

Saaved93a.

H. Saveedra, R. S. Gaines, and Micheal J. Carleton, "Micro Benchmark Analysis of the KSR1," *Proceedings Supercomputing '93 Conference, pg. 202-213,* ACM/IEEE, Portland, OR, November 1993.

Sgupt94a.

Satya Gupta, M. Barnett, D. Payne, L.Shuler, R. van de Geijn, and J. Watts, "Interprocessor Collective Communication Library (InterCom)," *Proceedings of Scalable High Performance Computing Conference,* pg. 357-364, IEEE Computer Society Press, Knoxville, TN, May 23-24, 1994.

Sinha92a.

Amitab B. Sinha, Helmet Heller, and Klaus Schulten, "Performance Analysis of a Parallel Molecular Dynamics Program," *Technical Report* UIUC-BI-TB-92-13, The Beckman Institute, University of Illinois at Urbana-Champaign, IL., July 1992.

Sten94a.

Peter A. Steenkiste, "A Systematic Approach to Host Interface Design for High-Speed Networks," *Computer,* pg. 47-57, March 1994.

Sund90a.

V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience,* 2(4):315-339, December 1990.

Sund93a.

V. S. Sunderam, and C. Hartley, "Concurrent programming with shared objects in networked environments," *Proceedings of 7th Intl. Parallel Processing Symposium,* pg. 471-478, Los Angels, April 1993.

Sunm90

Joan Stigliani, "Writing SBus Device Drivers", Sun Microsystems, Inc, 1990.

Tennea.

University of Tennessee and NSF, *Draft for a Standard Message-Passing Interface*

Theka93a.

C. A. Thekkath, H. M. Levy, and E. D. Lazowska, "Efficient Support for multicomputing on ATM networks," *Tech. Rep* 93-04-03, Department of Computer Science and Engineering, University of Washington, Apr. 1993.