Network Performance Studies in High Performance Computing Environments

by

Ben Huang

Supervised by Dr. Michael Bauer and Dr. Michael Katchabaw

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario, Canada

ABSTRACT

With the advances of modern computers and network technologies, High Performance Computing (HPC) is becoming increasingly accessible in numerous institutions and organizations. The commodity cluster consisting of general purpose computers is a popular model and appears to be a trend for future HPC. Good network performance is crucial for high performance computing in this kind of environment. In this thesis, we explore network performance characteristics in Linux-based HPC clusters. The behavior of Gigabit Ethernet, Myrinet and Quadrics' QsNet are investigated. To do this, we developed a network benchmark tool – Hpcbench, to measure the UDP, TCP and MPI communication throughput and latency for these high performance networks, and to trace system kernel interactions with the network subsystems. We then proceed to show that system architecture, configuration, workload, drivers of network interface cards, and other factors can significantly affect the network performance of these cluster environments.

Keywords: High Performance Computing, cluster computing, network performance analysis, Linux, UDP, TCP, MPI, Gigabit Ethernet, Myrinet, Quadrics.

ACKNOWLEDGMENTS

This thesis would not have been finished without the companionship and support of many people. It is a pleasure that I can express my deepest gratitude and sincerest appreciation to them.

I would like to thank my supervisors, Dr. Michael Bauer and Dr. Michael Katchabaw. I have greatly benefited from their advice, generosity and support. I would also like to thank Baolai Ge, John Morton and Gary Molenkamp for giving me kind assistance with my work on SHARCNET.

A lot of friends at UWO have made my life much easier and happier during my graduate studies. Although I can not enumerate their name here, I would like to thank all of them.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ACK	Acknowledgement flag; TCP header
ADI	Abstract Device Interface
API	Application Program Interface
ATM	Asynchronous Transfer Mode
BDP	Bandwidth Delay Product
bps	Bit per Second
BSD	Berkeley Software Distribution
CIFS	Common Internet File System
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access/Collision Detect
DAS	Direct-Attached Storage
DF	Do not Fragment flag; IP header
DLM	Distributed Lock Manager
DMA	Direct Memory Access
FC	Fibre Channel
FDDI	Fiber Distributed Data Interface
FIN	Finish flag; TCP header
FLOP	Floating Point Operation per Second
GUI	Graphical User Interface
HBA	Host Bus Adapter
HPC	High Performance Computing
HPCC	High Performance Cluster Computing
IBA	InfiniBand Architecture
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
KB	Kilobyte
Kbps	Kilobit per Second
LAN	Local Area Network

MAC	Medium Access Control
Mbps	Megabit per Second
MB	Megabyte
MPI	Message Passing Interface
MPMP	Multi Program Multiple Data
MPP	Massively Parallel Processor
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
NAS	Network-Attached Storage
NFS	Network File System
NIC	Network Interface Card
OS	Operating System
OSI	Open Systems Interconnection
PCI	Peripheral Component Interconnect
PVM	Parallel Virtual Machine
QoS	Quality of Service
RFC	Request for Comment
RTT	Round Trip Time
SAN	Storage Area Network
SCI	Scalable Coherent Interface
SCSI	Small Computers System Interface
SDH	Synchronous Digital Hierarchy
SDRAM	Synchronous Dynamic Random Access Memory
SMP	Symmetric Multi-Processing
SPMD	Single Program, Multiple Data
SRAM	Static Random Access Memory
SYN	Synchronize Sequence Numbers flag; TCP header
TCP	Transmission Control Protocol
TTL	Time-to-Live
UDP	User Datagram Protocol
WAN	Wide Area Network

Chapter 1

Introduction

1.1 Thesis Statement

To solve larger and more complex problems in shorter periods of time is one of the main purposes of building a high performance computing (HPC) system. Today, HPC systems act in a crucial role in many fields of research, such as life sciences, aerospace, atmospheric studies. To achieve higher computing power is one of the main tasks of the information technology industry.

Back in the 1980s, supercomputers that had much more powerful computing ability than high-end desktops and workstations were mainly Massively Parallel Processor (MPP) systems or vector machines. Those extremely expensive machines were only available in some large organizations or companies.

Things have changed considerably since the 1980s. Modern computers, even personal computers, are becoming more and more powerful. As well, high-speed, low-latency network products are becoming increasingly available and less expensive. It is now possible to build powerful platforms for high performance computation from off-the-shelf computers and network devices. These "supercomputers" are commonly referred to as commodity clusters.

One key concept behind most "supercomputers" and high performance computing is the parallel mechanism: combining the power of many standalone processes that are working together to solve a complex problem. A lot of parallel architectures have been developed in the past forty years. However, there is a trend towards using a generic parallel architecture for HPC systems: a cluster of standalone computers interconnected by a communication network. This is called High Performance Cluster Computing (HPCC).

This trend is mainly due to the advancement of networking technologies. Consider a standard 32 bit, 33MHz PCI bus. Its maximum throughput is less than 1 Gigabit/s, which is the bandwidth of Gigabit Ethernet. If network communication is as fast as system

buses, we clearly can take advantage of this for parallel computing: let a collection of computers work together through the network, as if they are working off of the same "motherboard".

Beowulf systems are the typical example. Beowulf clusters are made up of inexpensive off the shelf computers connected together with high-speed networking, running a Linux operating system and open source software. This commodity architecture greatly reduces the cost of building high performance computing systems, and makes HPC environments more and more accessible for researchers.

Performance analysis in HPC systems is an incredibly complex problem, however. Many factors can affect the overall performance of these computing environments. In the past, supercomputers were systematically tested by vendors before they were delivered to customers. In the commodity HPC world, most equipment is hooked up at the site, and the devices of the system may come from different vendors. This raises a problem: are all these components cooperating well enough? How does the system perform as a whole? A detailed performance analysis is very important for system designers, administrators, and application developers to discover potential bottlenecks and other performance problems and tune both the system and its applications accordingly.

In this thesis, we focus on investigating how well the network system in an HPC environment functions and identifying the main performance factors in a real HPC system. To do so, we developed our network benchmark – Hpcbench to evaluate the UDP, TCP and MPI communication, and analyze the performance of three high performance interconnects: Gigabit Ethernet, Quadrics' QsNet and Myrinet in our testbed SHARCNET [22], a leading distributed high performance computing network across southern Ontario in Canada. A new tool was required as existing benchmarking tools available are not suitable for this task, and lack features required for experiments in today's HPC environments.

1.2 Thesis Organization

The thesis is organized as follows: Chapter 2 introduces background material on high performance cluster computing environments. Chapter 3 discusses the design and

implementation of Hpcbench, a comprehensive network benchmark for HPCC systems. Chapter 4 investigates the performance of Gigabit Ethernet and Chapter 5 briefly examines other two high performance interconnects – Myrinet and QsNet. Chapter 6 concludes and gives suggestions for future work.

Chapter 2

Background of High Performance Cluster Computing

In this chapter, we provide some background on High Performance Cluster Computing environments. The systems in our test-bed will be also discussed.

2.1 HPC History and Its Evolution to Cluster Computing

A key reason we use computers is to speed up our work. Researchers always desire more powerful computers to solve larger and more complex problems. To achieve more computational power, parallel computing that distributes workload among processors has been well studied since the 1960s. From the 1960s to the 1990s, most parallel machines, often called supercomputers, were mainly vector computers and Massively Parallel Processing (MPP) systems. They were extremely expensive and only available in a very few organizations.

Information technology changes rapidly. The term "supercomputing" is used less frequently, and the more general term "High Performance Computing (HPC)" is used more widely since the 1990s. The term "HPC" refers to a variety of machine types whose computational ability is more powerful (at least two orders of magnitude in computing power) than current high-end personal computers and workstations. Today, "HPC" connotes computing ability in the Teraflops computational range [23].

To improve a computer's performance, an obvious way is to increase the operating speed of the processors, system buses, memory, and other components. As we have seen, the maximum CPU speed could approximately double every 18 months. This exponential improvement is one of the main forces of high performance computing. Although the phenomena might continue for a relatively long time, future improvements are constrained by the speed of light.

Another way to improve performance is to make multiple processors work together and combine their computational power. This approach is known as parallel computing or parallel processing. In the past forty years, numerous parallel architectures have been

proposed, most of which systems incompatible with one another. These parallel machines required careful construction, and ran specialized operating systems and software tools. Consequently, these machines tended to be costly to construct and maintain.

From the mid-1990s, with the advancement of networking technologies, cluster computing has become more and more prevalent. A cluster computing system is built with a number of stand-alone computers interconnected by high performance networks. The computers in a cluster can be general purpose, off-the-shelf personal computers, workstations, or small symmetric multi-processor (SMP) systems. The network system to connect these computers needs to provide high-speed, low-latency communication. This communication could be Fast Ethernet, FDDI, ATM, SCI (Scalable Coherent Interface), Gigabit Ethernet, or some other proprietary technology, such as Quadics Network [19] and Myrinet [21].

As cluster computing and related technologies become mature, more and more High Performance Computing systems have been built using this paradigm. In looking at the trend of TOP 500 supercomputers we can see this evolution, as shown in Figure 2-1.

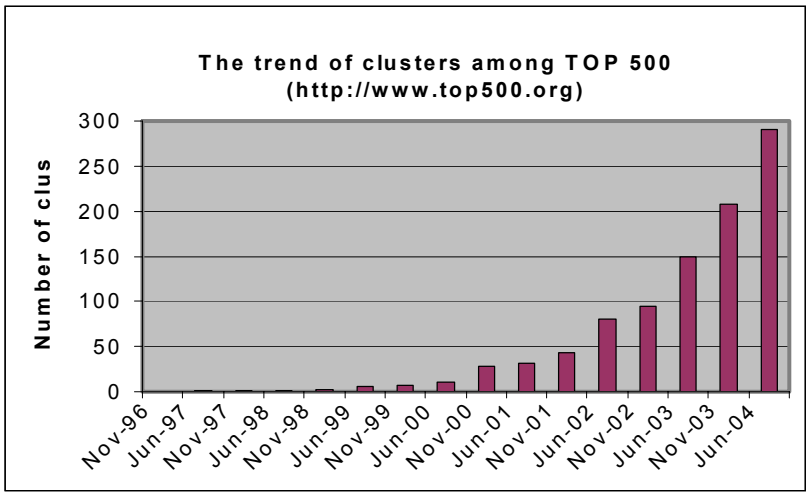

Figure 2-1 Trend of Clusters among TOP500 Supercomputers

Figure 2-1 lists the architecture statistics of the top 500 supercomputers in the world surveyed by top500.org [14]. Impressively, we noticed there were 6 clusters on the top 10 list in the latest survey in June 2004. This statistics show the change in the field of HPC: toward clusters.

The advantages of high performance cluster computing can be summarized as follows:

- Cost effective: Clusters can be built with commodity devices and run commodity software.
- Flexible: Just about any PC, workstation, or SMP can be clustered into the system.
- Extensible: More nodes can be added into the system when more computational power is needed without great difficulty.
- Easy to be managed: It is easier to monitor and control the system by using common, off-the-shelf monitoring tools rather than specialized, proprietary monitoring tools. Moreover, a single node's failure will not affect the whole system and it is possible to repair the failed node without interrupting the rest of system.
- Easy to be integrated: It is easier to merge a cluster into a global distributed Grid computing environment by using the standardized OGSI interface [24].

Figure 2-2 shows the basic structure of a High Performance Computing Cluster. In cluster systems, basically all nodes are standalone computers with an operating system installed on the local file system. Additional libraries, such as the implementation of MPI for example, may be necessary to support parallel computing in the cluster.

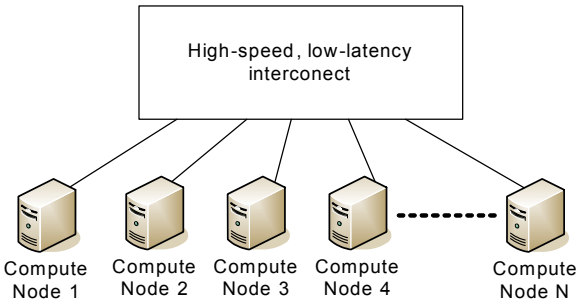

Figure 2-2 Structure of High Performance Cluster Computing

Currently the operating systems in the HPC world are dominated by Unix-based systems. (TOP500 list does not specifically identify the operating systems). According to the 2004 International Supercomputer Conference [24], the operating systems of the TOP500 systems (2004 June version) break down to around 55 percent Linux, 40 percent of other Unix systems, and less than 5 percent with a Windows platform. Meanwhile, more than

half of the systems (287) in the TOP500 use Intel processors, as compared to only 119 a year ago. This phenomenon is matching the trend of commodity cluster computing. Like the success of WinTel (Windows+Intel) systems in PC market, LinTel (Linux+Intel) computers are playing an important role in the HPC world. People appear to prefer a solution that is less costly and more compatible, while having the same performance.

Linux-based clusters with commodity software and hardware are often referred to as Beowulf systems. Since Linux is open source and is becoming more and more stable and powerful, it is expected to be the dominating operating system in HPC clusters in the future. Even industry giants are zealously providing complete Linux solutions for HPC systems, putting their own operating systems aside. For example, IBM has 226 supercomputer systems on the TOP 500 list (June 2004), 149 of which were Linux clusters [14]. All systems in our experimental test-bed environment use Linux, most of the discussions in this thesis are also based on Linux.

2.2 HPC Networking

2.2.1 High Performance Network Technologies

The interconnection network is a key component for High Performance Computing Clusters. Traditional FDDI, token ring, Ethernet, and Fast Ethernet may not satisfy HPC requirements for high-speed and low-latency communication between compute nodes.

In the past ten years, many advanced network technologies have been developed and used in HPC systems. Three high performance networks that are used in SHARCNET are typical examples: Gigabit Ethernet, QsNet, and Myrinet.

- Gigabit Ethernet: Ethernet is the world's most pervasive networking technology based on CSMA/CD protocols. Gigabit Ethernet (1000Mbit/s) has evolved to achieve higher bandwidth and lower latency while keeping the original protocol semantics as traditional Ethernets. The standards of Gigabit Ethernet (802.3z for UTP, and 802.3ab for Optic Fibre connection) were developed by the Gigabit Ethernet Alliance [15] in 1996 and the first commercial products came to market in 1997. Now, Gigabit Ethernets have become the most popular and cost-effective solution for users with desires for high performance networks. The latest 10 Gigabit Ethernet was

standardized in 2002 (802.3ae) [15], with commercial products appearing in 2003. Gigabit Ethernet's familiarity, easy installation, and maintenance features make this technology very competitive in cluster computing now and in the future. However, Ethernet's overhead can result in a relatively long latency during communication, which may degrade the performance of some parallel applications. The maximum throughput of Gigabit Ethernet in practice can exceed 900 Mbps, while the Round Trip Times of UDP and TCP protocols in Gigabit Ethernets are usually greater than 50 μsec (refer to Chapter 4 for more detail). There were 163 clusters using Gigabit Ethernet for message passing on the list of TOP 500 supercomputers (June 2004).

- Quadrics Network (QsNet) [19]: QsNet is a high bandwidth, low latency interconnect for high performance clustering systems. QsNet integrates individual nodes' address spaces into a single, global, virtual-address space. QsNet can detect faults and automatically re-transmit packets during communication. QsNet includes two main parts: Quadircs's network interface adaptor (Elan), and a QsNet switch (Elite) that is constructed with a fat-tree topology. Most clusters in SHARCNET were built with a Quadrics interconnect, in addition to Gigabit Ethernet. The one-way achievable throughput in QsNets can reach 1500Mbps while the network latency in QsNets can be lower than 10 μsec (refer to Chapter 5 for more information).

- Myrinet [21]: Myricom's Myrinet is one of the most widely used high-performance interconnects for scalable systems. In June-2004 TOP500 supercomputers list, there were 186 (37.2%) systems using Myrinet technology. The point to point links by Myrinet switches can achieve around a 4 Gbps (bidirectional) data rate and as low as 7 μs one-way latency (refer to Chapter 5 for more detail). A low-latency protocol called GM is the preferred software for Myrinet, and MPICH-GM is the MPICH binding of GM. An efficient version of TCP/IP over GM is also available. This network technology is deployed in some parts of SHARCNET as well.

The proprietary QsNet and Myrinet technologies have a better performance than Gigabit Ethernet in many respects. One key technology of Quadrics and Myrinet to improve performance is the Zero-copy technique [20]. Using this approach, an application's network subsystem directly accesses network packets by going through the network

interface's driver using a virtual-to-virtual memory mapping. The goal is to eliminate expensive system calls and other overhead; consequently this method is also referred to as OS-bypass communication. This is illustrated in Figure 2-3.

In our experimental test-bed environment, the network interface cards (NICs) of these three network systems were PCI-based. One cluster in our test-bed (mako) deployed PCI-X NICs. The theoretical maximum data rate of the PCI-X bus (64bit/133MHz) is about 8.5 Gbps (PCI-X version 1) [17]. To meet higher I/O needs for network systems, a new InfiniBand Architecture (IBA) [16] was developed and released by a group of leading IT corporations in 2000. This approach defines a new standard for computer I/O and inter-computer communication whose bandwidth is not restricted to system's shared buses. InfiniBand is being adopted by many IT vendors as the next generation networking connectivity infrastructure.

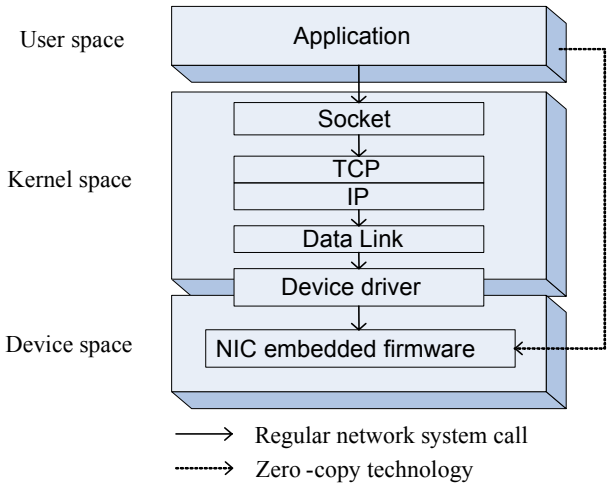

Figure 2-3 Zero-copy Communication

2.2.2 Networking of HPC Clusters

Generally, all compute nodes inside a cluster are controlled by a "Master node" that is responsible for dispatching jobs and managing the whole cluster. The master node usually does not participate in the actual computation work, but assumes more of a coordination role. There are three main kinds of data flow in a cluster: message passing between compute nodes for parallel applications, data traffic between compute nodes and shared storage, and management traffic. All traffic can be conducted through one network system, or can be separated into different network systems. This is shown in Figure 2-4.

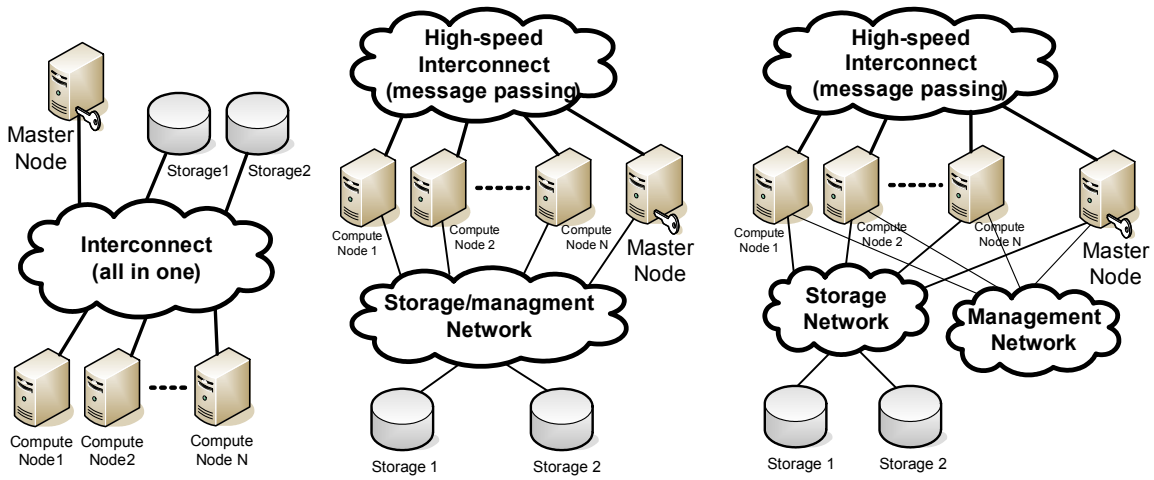

Figure 2-4 Three Types of Networking for HPC Clusters

Using two interconnects to separate message passing from other traffic (the centre diagram in Figure 2-4) is the most common model for HPC clusters (including SHARCNET). Gigabit or Fast Ethernets are widely used for data access and system management because of its compatibility with traditional TCP/IP protocols and tool supports. In such a configuration, private IP addresses are used inside the clusters to avoid IP conflicts with the global Internet. As well, class B private address spaces (172.16-31.x.x and 192.168.x.x in SHARCNET's case) are often used in cluster systems, as the class A private address space is often already used in the organization's Intranet.

A cluster usually has some kind of link to the Internet, where users are able to login and submit their jobs. As showed in Figure 2-5, a submission server is accessible from the Internet and is responsible for sending the jobs to the master node.

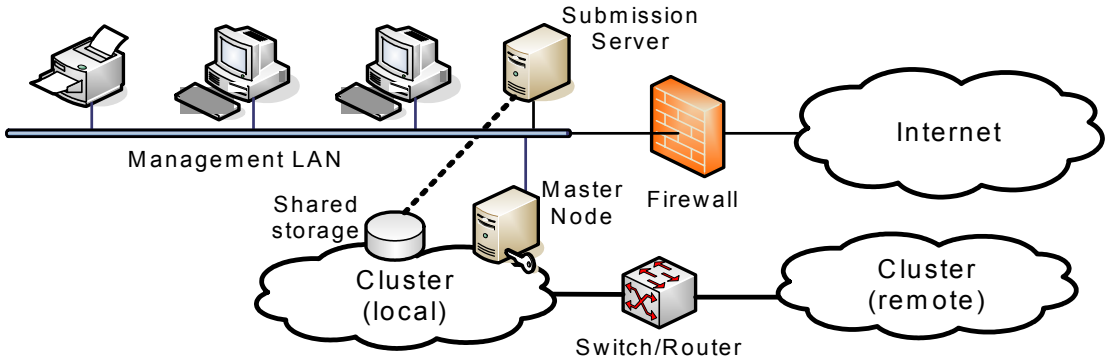

Figure 2-5 Cluster Internet Connection

In our test-bed, the master nodes in all clusters also act as the gateway to the Internet. For example, in the greatwhite cluster, the master node (greatwhite.sharcnet.ca) has three network interface cards: Elan for a Quadrics interconnect, eth0 (192.168.3.1) for a Gigabit Ethernet interconnect, and eth1 (129.100.171.40) connected to the campus Internet. Because the master node is connected to the broader Internet, the master node is vulnerable to attack or misuse by users. All jobs running in compute nodes will be stopped if the master node crashes.

2.3 Message Passing Interface (MPI)

2.3.1 MPI Introduction

Parallel computation typically divides a job into many subtasks, with each subtask handled by separate processes. A communication protocol between cooperating processes is necessary to support this parallel computing. Message Passing Interface (MPI) is a vendor-independent library specification defined by MPI forum [13], and is frequently used as the means for separate processes to communicate during a parallel computation.

The first version of MPI (MPI-1) was developed during 1993-1994 by a group of researchers from industry, government, and academia. Their two main goals were to provide source code portability and allow efficient implementation for parallel programming. The second MPI standard (MPI-2) that was completed in 1998 includes a set of extensions of MPI-1. When we discuss MPI in this thesis, we will, generally, not distinguish between the two versions.

The MPI specification defines a model of message passing in a parallel computing environment and defines a library interface for programmers. It does not specify a particular implementation or require a particular programming language. Figure 2-6 shows the MPI role in a system. MPI is now the de facto standard for parallel computation based on message passing model, and implementations of MPI exist for most parallel computing environments. One reason for MPI's success is its portability. Parallel programs containing MPI subroutine and function calls can work on any machine on which the MPI library is installed with little or no modification necessary.

Most implementations of MPI are based on MPI-1 with C and Fortran bindings. Currently, the most two widely used MPI implementations are MPICH and LAM/MPI. Although MPI-1 defines 125 functions, most of them are rarely used by common users. A simple parallel program can operate using 6 MPI function calls: MPI_Init(), MPI_Com_rank(), MPI_Com_size(), MPI_Send(), MPI_Recv(), and MPI_Finalize().

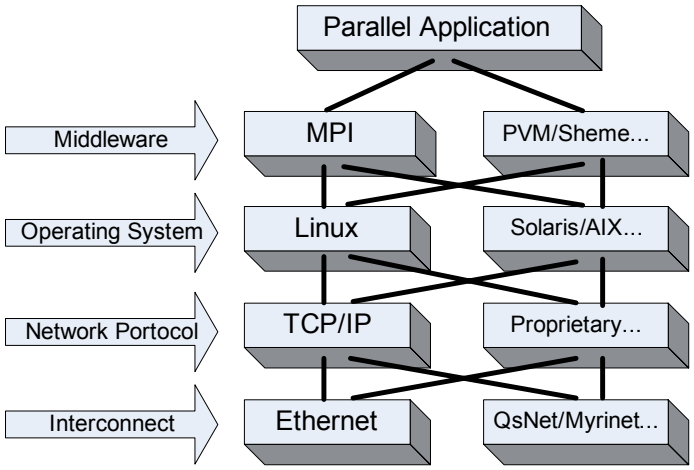

Figure 2-6 MPI Role in the System

2.3.2 MPICH

MPICH (stemmed from MPI-Chameleon) is a freely available MPI implementation developed by Argonne National Laboratory and Mississippi State University [12]. MPICH is the default MPI implementation in all clusters of SHARCNET. MPICH has played an important role in the development of MPI. It is also the "parent" of many other commercial or vendor implementations of MPI, such as Myrinet's MPICH-GM. MPICH was developed concurrently with the development of MPI, and during the standardization process since the developers of MPICH were participating in the MPI definition at the time. The first version of MPICH was released at approximately the same time as the original MPI 1.0 specification was released.

MPICH can run on a wide variety of systems and its portability of MPICH comes from its efficient two-layer design. Most MPICH code is device independent and is implemented on top of an Abstract Device Interface (ADI) which hides most hardware-dependent details. This enables it to be easily ported to new architectures. The ADI layer

defines a number of "devices"; each MPICH device defines a new implementation, while most MPI syntax and semantics have been implemented on the top layer (see Figure 2-7).

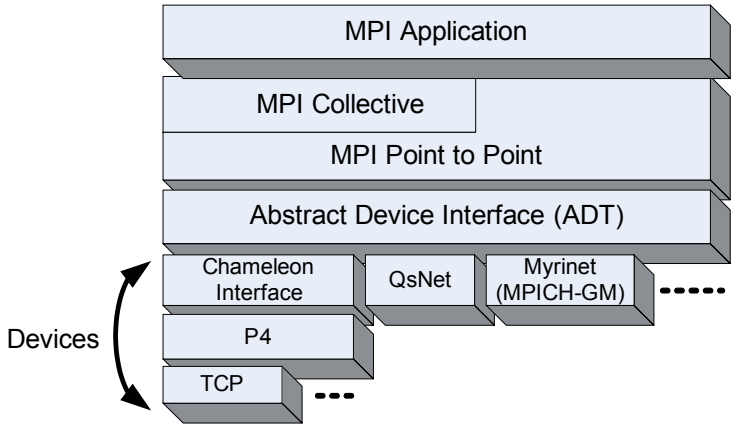

Figure 2-7 The Layering Design of MPICH

An important note about MPICH is that it can be installed on top of TCP/IP communication stacks by default and run without root privilege in a cluster of computers. The following 10-line code is the MPI version of HelloWorld.

```
$ cat HelloWorld.c
/********** HelloWorld.c **********/

#include <stdio.h>
#include "mpi.h"

int main(int argc, char * argv[])
{
    int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello world! Process ID: %d Total process: %d \n", rank, size);
    MPI_Finalize();
    return 0;
}
$ mpicc HelloWorld.c -o HelloWorld
$ mpirun -np 4 HelloWorld
Hello world! Process ID: 0 Total process: 4
Hello world! Process ID: 2 Total process: 4
Hello world! Process ID: 1 Total process: 4
Hello world! Process ID: 3 Total process: 4
$
```

The script *mpicc* is used to compile (using *gcc* by default) and link the MPI program. The executable is started by another script *mpirun* (MPI-1). In our example, we ask for 4 processes to run the program. The startup script *mpirun* will first setup the execution environment and check whether the defined resources are sufficient to run the program. For MPICH, the machine file (-machinefile option) and the process group file (-p4pg

option) define what machines and processes will be used to launch a parallel job. If both of them are not defined, *mpirun* will examine *$MPIHOME/util/machines/mahince.xxxx* to determine the available machines and processes, where *xxxx* is the architecture of the system such as LINUX. If condition matches, *mpirun* will try to communicate with those machines by *rsh* or *ssh* utilities, depending on the configuration of MPICH during the installation. If communication to all nodes is successful, remote execution of the program (*HelloWorld*) with tailored environment and argument settings occurs at each node. At this point, the same executable program (*HelloWorld*) will be running on all participating processes (nodes), and each process can communicate with each other using MPI libraries on each machine. In our example, the four processes simply printed out their process information.

The above example illustrates the Single Program, Multiple Data (SPMD) model: the same program runs on all participated nodes (processes). In SPMD model, each process typically accesses different data. In most cases, SPMP is the default MPI working type, although we could convert some SPMP to a Multi Program, Multiple Data (MPMD) model of computation using MPI.

2.4 Job Management System

2.4.1 Goals of JMS

There are often hundreds or thousands jobs submitted by many users in HPC systems. Controlling these jobs can be quite a challenge. The startup script *mpirun* that comes with all MPI implementations is unable to actually schedule the jobs, and is unable to monitor the workload in the system for optimized job dispatching. It can only send user jobs to the nodes from the predefined machine files. This may result in unreasonable usage of system resources; some nodes in the cluster may be extremely busy, while the others may be relatively idle. In practice, using *mpirun* to submit a complex job involving many processes is uncommon and even forbidden in many HPC clusters.

A job management system is also referred to as a workload management system. Its goal is to optimize the computing resources to let submitted jobs run and ultimately complete.

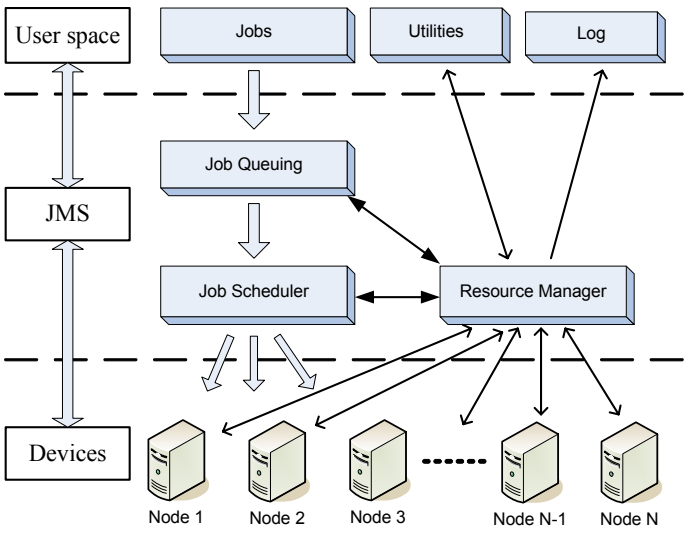

Figure 2-8 Roles of Job Management System in HPC Clusters

There are several elements involved in job management systems, including job queuing, job scheduling, job dispatching, resource and workload monitoring, resource and workload management, accounting, and so on. We abstract these elements into three parts, according their functionalities:

- Job queuing: When a job is submitted, a description of the task to be executed, along with arguments and a set of resource requirements for the execution are put into a container of the Job Management System (JMS). The JMS will return a job ID to the user immediately, and queue the job to await scheduling.

- Job scheduler (workload manager): The job scheduler attempts to choose the best-fit job to run from the job queue based on defined policies and available resources. A scheduler tells the resource manager what to do, as well as when and where to run jobs. "Best-fit" means that the scheduling is not necessarily first-comes first-served. The job scheduler is a key component of the JMS. Many factors have to be taken into account to make a decision about which job to schedule and where, such as application type, user privileges, usage policies, availability of cluster resources, and so on. The JMS should balance policy enforcement with resource optimization.

- Resource manager (monitoring and management): It manages the job queue and manages the compute nodes. The resource manager collects system resource information from each node in the cluster for the job scheduler. It also handles the

startup and cleanup of jobs in each node, and logs the necessary messages.

A number of job management systems have been developed in the past decade. Popular job management systems include Condor [25], LSF (Load Sharing Facility) [26], Moab [27] and PBS (Portable Batch System) [28].

2.4.2 LSF (Load Sharing Facility)

LSF is a sophisticated job management system with more than a ten-year history. It is the default JMS system in SHARCNET. Some of LSF's key features include support for job migration, system/user level check pointing, inter-cluster launching, and so on. LSF also provides a graphical user interface (GUI) for both the user and the administrator to use. Figure 2-9 depicts a snapshot of the LSF GUI.

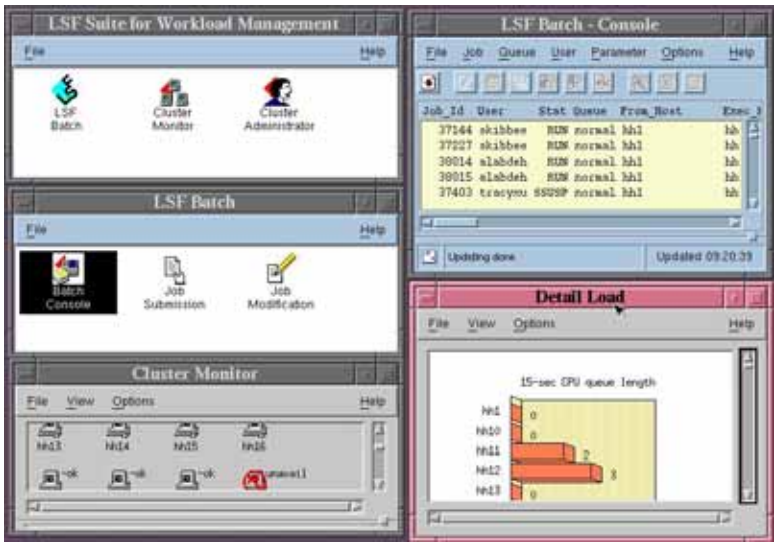

Figure 2-9 LSF GUI

In most cases, users submit their jobs by command line on a login terminal. LSF includes utilities for job submission, job modification, and job querying. Figure 2-10 shows the job statistics in the cluster of deeppurple. We can see there were more than 100 jobs submitted to the cluster, and most of them were pending in the job queue. Notice that all compute nodes (dp2-dp12) were extremely busy with 100% CPU usage, showing the resource intensive jobs running in the cluster. Figure 2-11 shows the interaction of LSF daemons and how a job is handled by LSF.

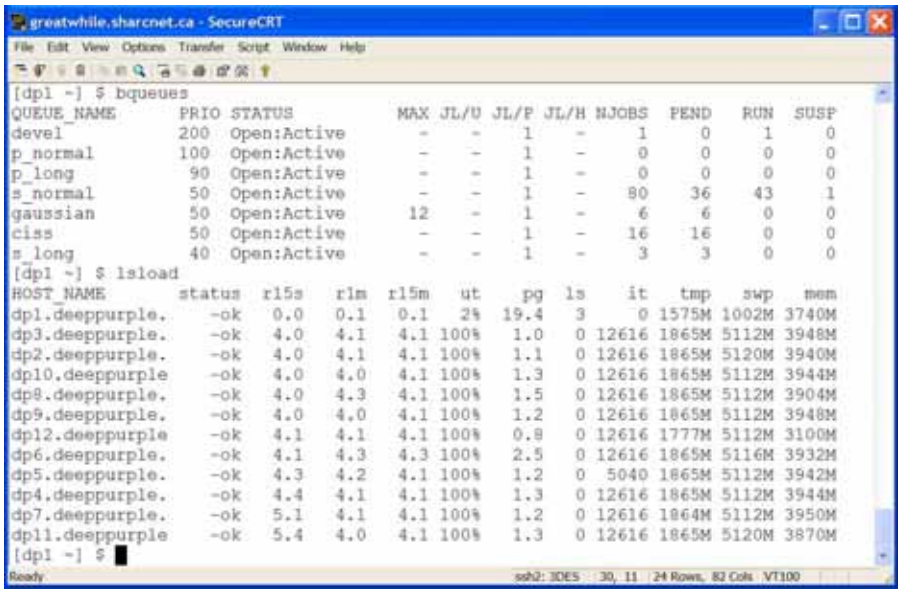

Figure 2-10 Viewing Managed Jobs by LSF

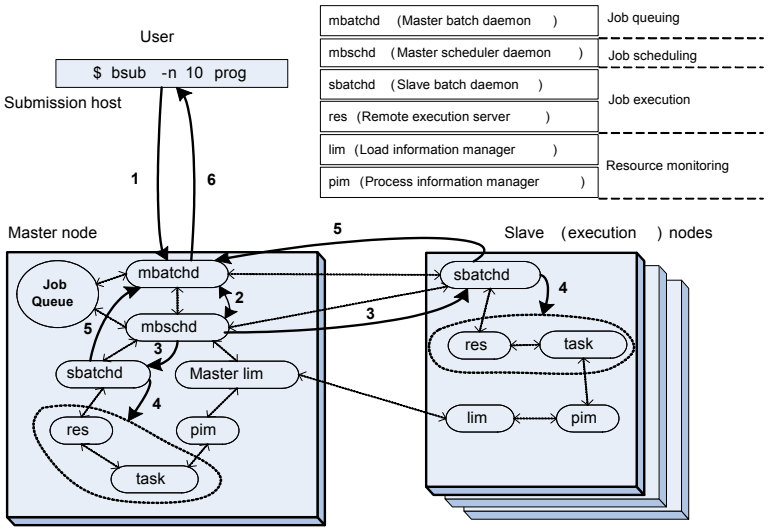

Figure 2-11 Daemon Interactions in LSF

2.5 File Systems in HPC Clusters

2.5.1 Storage Networking

There are three categories of storage architectures used in HPC cluster systems, depending on how the storage is connected: direct-attached storage (DAS), network-attached storage (NAS), and storage area networks (SAN) (see Figure 2-12).

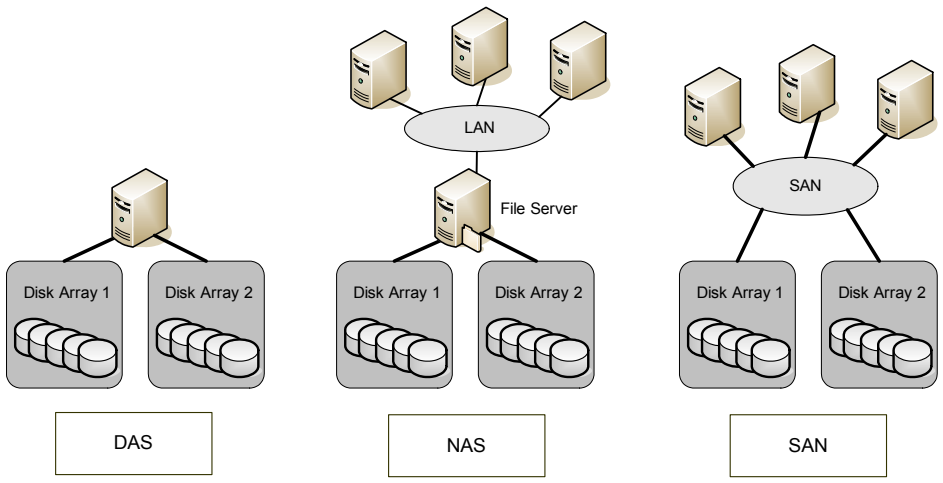

Figure 2-12 Local and Shared Storage Architectures

A DAS storage device is directly attached to a host system, and can only be used by the connected host. It is, in essence, a local file system. Both NAS and SAN architectures provide some form of shared storage. They are called shared file systems or distributed file systems. In HPC clusters, a distributed file system can be necessary since each inner (compute) node has to run the same executables and work with the data assigned by the master node.

A NAS device is a file server that typically uses NFS (Network File System) or CIFS (Common Internet File System) protocols to transport files over the local area network (LAN) to clients. In the NAS model, the internal physical disk drives, connected by IDE, ATA or SCSI interfaces [41], are transparent to the user.

A SAN is a special type of network that connects computers and storage. It changes the relationship between hosts and storage resources. Hosts and storage are joined together through a peer-to-peer network. Each host in the SAN system essentially has a Host Bus Adapter (HBA) that links to the SAN storage. All I/O requests are done solely between the hosts and the storage. The communication protocols for SAN can be Fibre Channel (FC), iSCSI, FC over TCP/IP (FCIP) and Internet Fibre Channel Protocol (iFCP). Storage resources can be arbitrarily assigned to any hosts; for example, we can subnet (group) several SAN file systems over one physical Fibre Channel connection.

In essence, a SAN is a communication architecture between hosts and storage in the network. The real systems are not restricted to SCSI over Fibre Channel or SCSI over

Ethernet. For example, a peer-to-peer access model using native block-level communication instead of file-level can be considered as a SAN system. We may have ATA-SAN or IDE-SAN in the future. SAN storage is well supported in the InfiniBand architecture mentioned earlier.

2.5.2 Cluster File Systems

There are many implementations of distributed file systems that are based on the NAS or SAN models, or some hybrid of the two of them. Some are designed for LANs, such as NFS and CIFS; some are dedicated to the Wide Area Network (WAN), such as Coda [42] and AFS [43]; and some focus on parallel I/O like PVFS (Parallel Virtual File System) [32], where the data are striped (usually 64KB) across multiple file servers to balance the data load in the network. Many of these distributed file systems are able to efficiently serve hundreds or even thousands of nodes in a cluster, and are typically referred to as cluster file systems.

Data inconsistency is one of the most challenging tasks in the implementation of cluster file systems. In a cluster, each node will load the file allocation table of the mounted (shared) disks into its own memory. The allocation table can be modified by each node while the other nodes are not aware of the changes made. It will be very costly if all nodes are simply informed to update the table whenever there is a modification. In addition, each node will cache some data received from networked storage. If applications want to use this data later on, the system will provide the data from its cache to applications without retrieving this data again from the network, even though this data might have be modified by other nodes.

To avoid these problems, a synchronization mechanism should be implemented in cluster file systems, such as the distributed lock manager (DLM) [29]. All computers in a cluster will notify each other of their activities regarding modifications to shared files, using a fairly sophisticated means. The interconnection can be Ethernet, Fibre Channel or proprietary technologies such as Myrinet and Quadrics Network. Some solutions have a metadata layout. A metadata server manages the all file and directory information while the real data are stored in separate networking storage.

Due to many advantages of the SAN approach, most cluster file systems can support or are based on a SAN architecture. Among these are Redhat's GFS (Global Files System) [31], IBM's GPFS (General Parallel File System) [32], Cluster File System's Lustre [33], Polyserve's Matrix Server [34], and Dataplow's SFS (SAN File System) [35].

2.5.3 Network Storage in SHARCNET

Clustered file systems offer significant performance advantages. Native SAN systems are fast and efficient, but also costly. Currently, most shared file systems in SHARCNET are based on NFS systems over Gigabit Ethernet, as showed in Figure 2-13.

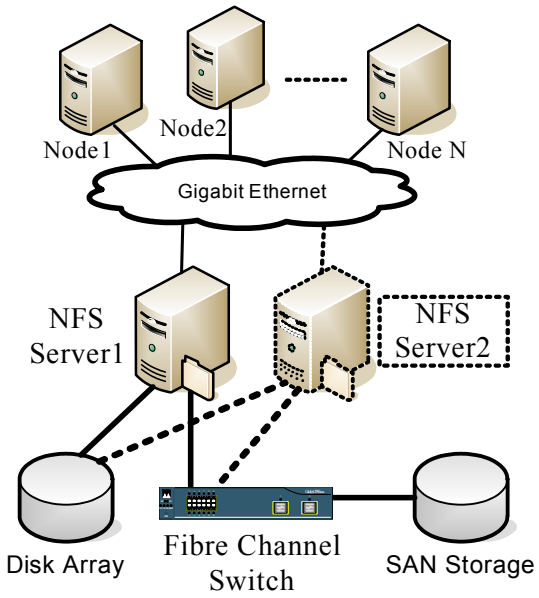

Figure 2-13 Shared File System in SHARCNET

The dedicated file server shares storage using the NFS protocol. This storage can be a disk array or SAN storage. To balance the data load, there may be multiple NFS servers used in some clusters. A SAN can be connected to the NFS servers through a variety of communication networks; Figure 2-13 illustrates a Fibre Channel connection. With this configuration, the management of file systems is fairly easy, and the SAN environment is transparent to all client nodes. This type of configuration, however, implies that it may not be possible to take advantages of all the features of a SAN.

NFS is based on TCP/IP communication and uses a stateless client/server architecture. This raises several performance problems. Firstly, all data has to be fetched from the

storage via the server for every I/O request. NFS does have a client cache mechanism based on access patterns, which can provide a significant improvement for some I/O operations. However, it may fail in most cases in HPC environments, since many scientific parallel applications run in HPC systems do not access data of files in a sequential order (recall the SPMD parallel model for cluster computing). Secondly, transferring data over NFS with a TCP/IP protocol incurs extra protocol and CPU overheads. Thirdly, locking files may be a serious problem when a cluster has many nodes. To ensure consistent data, NFS applies this locking mechanism. However, when many clients are trying to read/write the same file, a race condition may occur and overall application performance may dramatically decrease. In HPC systems, all nodes are working in parallel, and some applications that need to work closely with storage may encounter a serious problem due to locking. Figure 2-14 shows the differences between file access in NFS and a SAN.

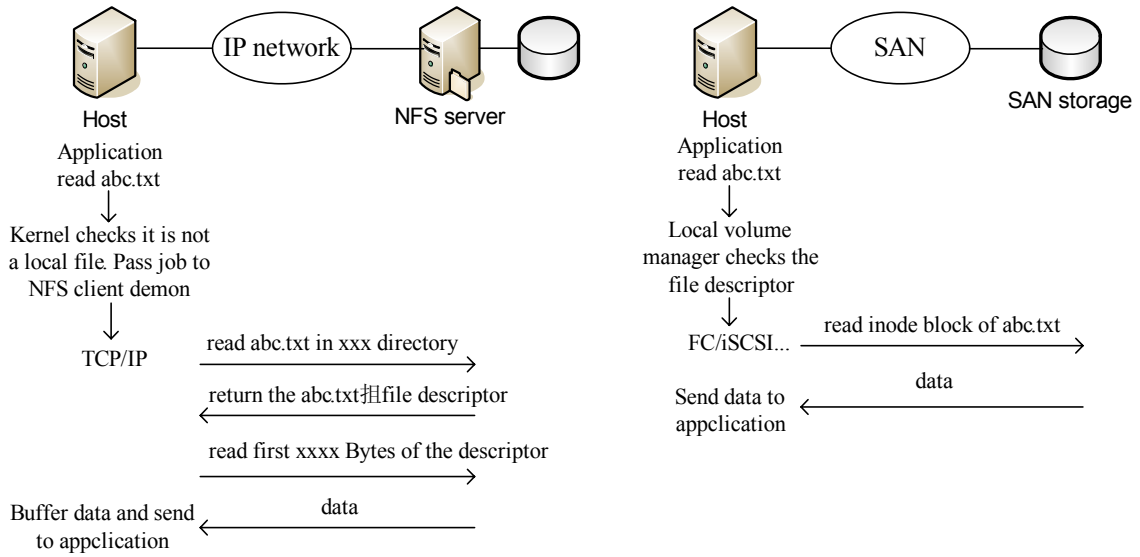

Figure 2-14 NFS vs. SAN

2.6 Test-bed Specifications

SHARCNET, the Shared Hierarchical Academic Research Computing Network, is a multi-institutional HPC distributed network across 9 universities in southern Ontario. We will examine four clusters in SHARCNET: greatwhite (UWO), deeppurple (UWO),

hammerhead (Guelph) and mako (Guelph). Figure 2-15 shows the network structure of our test-bed and Table 2-1 lists the detailed system information of these four clusters.

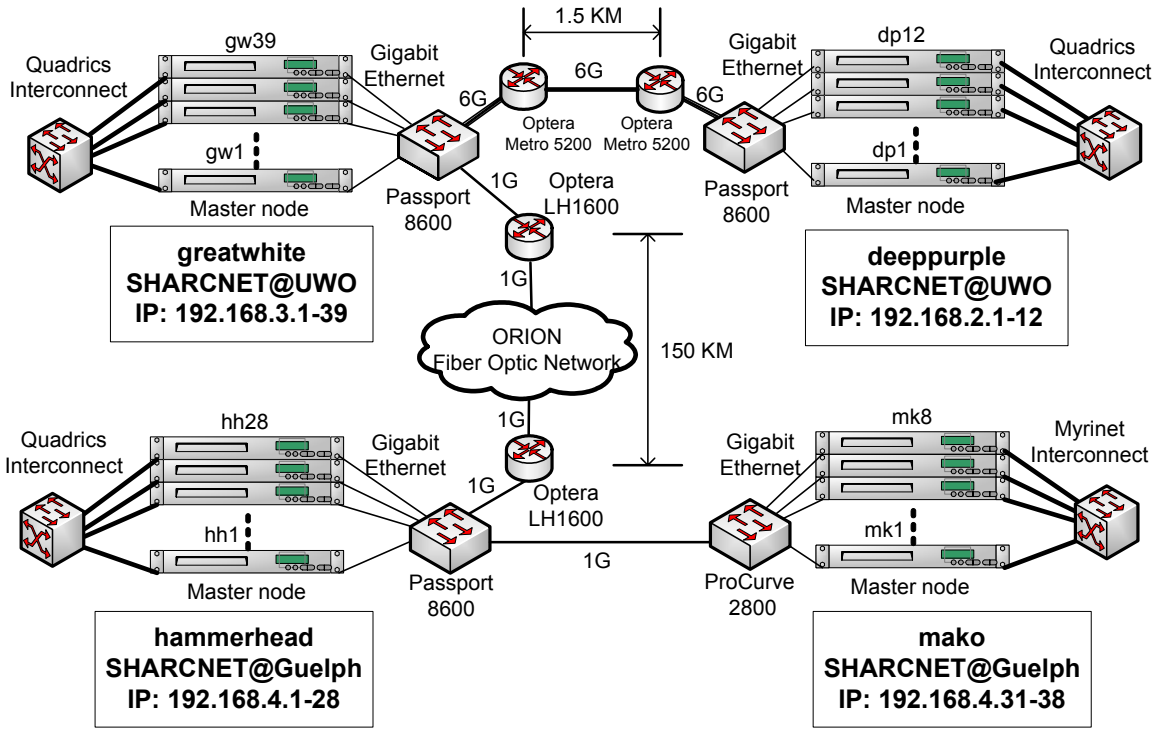

Figure 2-15 The SHARCNET Test-bed Network Structure

Greatwhite	
Master-node: gw1 (greatwhite.sharcnet.ca), Compute-note: gw2-gw39	
Architecture	Compaq ES40 (gw1-gw37) Alpha and ES45(gw38-gw39) Alpha
OS	Linux 2.4.21-3.7qsnet #9 SMP
CPU	gw1:4x500MHz gw2-gw37:4x833MHz gw38-gw39:4x1GHz
Memory	gw1-gw37:4G RAM gw38-gw39:32G RAM
Interconnect	Eth0: Alteon AceNIC Gigabit Ethernet 64bits/33MHz PCI Elan0: Quadrics QSW Elan3 PCI Network Adaptor
Deeppurple	
Master-node: dp1 (deeppurple.sharcnet.ca), Compute-note: dp2-dp12	
Architecture	Compaq ES40 Alpha
OS	Linux 2.4.21-3.7qsnet #9 SMP
CPU	dp1:4x500MHz dp2-dp12:4x666MHz
Memory	4G RAM
Interconnect	Eth0: Alteon AceNIC Gigabit Ethernet 64bits/33MHz PCI Elan0: Quadrics QSW Elan3 PCI Network Adaptor
Hammerhead	
Master-node: hh1(hammerhead.sharcnet.ca), Compute-node: hh2-hh28	
Architecture	Compaq ES40 Alpha
OS	Linux 2.4.21-3.7qsnet #9 SMP
CPU	hh1:4x500MHz hh2-hh28:4x833MHz
Memory	4G RAM
Interconnect	Eth0: Alteon AceNIC Gigabit Ethernet 64bits/33MHz PCI Elan0: Quadrics QSW Elan3 PCI Network Adaptor
Mako	
Master-node:mk1(make.sharcnet.ca), Compute-node: mk2-mk8	
Architecture	HP DL360 Intel Xeon
OS	Linux 2.4.20-8smp #1 SMP
CPU	4x3GHz Hyperthreading
Memory	2G RAM
Interconnect	Eth0: Broadcom Tigon 3 Gigabit Ethernet PCI-X 64bits/100MHz Myri0: Myricom PCI Network Adaptor

Table 2-1 System Information for the SHARCNET Test-bed

Chapter 3

Implementation of Hpcbench

In this chapter, we first survey a number of different network benchmark techniques and tools, and then introduce our benchmarking tool: Hpcbench.

3.1 A Survey of Network Measurement Tools

To measure network attributes such as throughput and latency on an end-to-end basis, two types of measurement, active and passive, are commonly used.

Active measurement is typically based on a client/server model. The client sends probe packets to a server that replies back to the client, where both client and server might perform a timing measurement. For instance, in network throughput testing, one approach is to send as much data as possible from the client to the server, and compute the throughput by the size of transferred data (sent or received) and the time of transmission. This is not the only way to measure throughput, however. Different methodologies have been proposed and used in order to evaluate network throughput without trying to saturate the path in such an intrusive fashion. The typical examples are one packet, packet pair and Multi-packet models [44]. These mathematical models are based on the fact that characteristics of each packet traveling in a link have some relation to the throughput and network delay. Consequently, these methods rely on certain assumptions, and may not be as accurate as direct injection measurement. For instance, the *packet pair technique* may send as few as two packets, and the server then judges network throughput by analyzing the timestamps of these two consecutive packets. These types of measurements are considered non-intrusive, since they do not significantly increase network traffic during testing. Examples of tools adopting these techniques include *Pathchar* [45] and *Pathrate* [46]. Active non-intrusive measurement tools are usually used in public networks to avoid affecting other users.

Passive measurement does not depend on deploying test applications to specific hosts in the network. Instead, it probes network traffic to compute network performance attributes.

For example, consider TCP's three-way handshaking sequence—the time interval between a SYN packet and the corresponding SYN/ACK packet is a measure of the round trip time (RTT) along the network link between the source and the destination. Passive measurement does not create extra network traffic and does not have to run client and server processes in the network. Researchers often use tools such as *tcpdump* [54] to capture the raw network traffic for analysis, which is a kind of passive analysis. The *ntop* utility [47] is another passive measurement tool showing network statistics with an output format similar to what the Unix *top* utility presents. The *nettimer* [48] utility is another end-to-end network bandwidth measurement tool that can operate in passive mode. Passive measurement is usually used in network system monitoring and often works with SNMP. Generally, people do not use passive measurement to conduct network benchmarking since it is inflexible and quite difficult to control.

Numerous network measurement tools have been developed; the following are some of the most commonly used:

- *Ping*: *Ping* checks network connectivity and reports the round trip time of a remote machine. It sends a small ICMP echo request to a destination and then waits for the ICMP echo response from the host. Ping is a handy tool to check both whether a networked computer is operating, and the network latency of the path.

- *Traceroute*: *Traceroute* shows the network routing and latency information. It is based on the ICMP error message of the IP protocol that is generated when the Time-to-Live (TTL) of a packet has been exceeded. When a packet's TTL reaches zero, an intermediate router or host returns an ICMP error message to the source host. *Traceroute* starts by sending a UDP datagram (not ICMP packet like *ping*, as routers may not generate ICMP errors for ICMP messages) to the destination with TTL of 1. The first-hop router automatically decreases the TTL by 1 in the UDP datagram and finds the packet has already reached the hop limit (zero). So the router drops the UDP datagram and sends an ICMP error message to the source. The source host processes the error message and prints out the round trip time of the first hop, then increases TTL number by 1 and sends another UDP datagram to the network, waiting for another ICMP error message from the second-hop router. The same steps continue

until the destination host is reached.

- *Ttcp*: One of the first TCP throughput testing tools ever written was *Ttcp*. Many variations have since been created with some improvements and new options, such as *nuttcp* [49] which supports UDP measurement as well and has various configurable parameters.

- *Udpmon* [50]: A UDP latency and throughput measurement tool capable of providing detailed logs for its tests. It uses assembly language to access an Intel CPU cycle counter for high-precision timing, thus it can only work on IA32/IA64 platforms.

- *Netperf* [51]: A sophistical network benchmark that can be used to measure the performance of many different types of networking technologies, including TCP/UDP, DLPI, Unix Domain Sockets, Fore ATM API, and HP HiPPI Link Level Access. *Netperf* is capable of testing both end-to-end unidirectional throughput and latency.

- *Iperf* [52]: *Iperf* is a relatively new network benchmark written in C++, while all other tools discussed above were written in C. It can measure TCP and UDP throughputs with various tunable parameters and with multi-threaded support. *Iperf* does not measure network latency, however.

- *NetPIPE* [53]: *NetPIPE* is a protocol independent network performance measurement tool. It performs simple ping-pong or stream tests. To be protocol independent, *NetPIPE* encapsulates network and timing system calls into different modules. *NetPIPE* only works with connection-oriented protocols such as TCP and MPI. Consequently, it does not support UDP communication.

All of these utilities were designed as general purpose network measurement tools. In HPC environments, however, these tools may not work well and provide a complete picture of network performance. For example, if we use the utility *ping* to measure the network round trip time between two clusters in our test-bed (greatewhite and deeppurple) we get:

```
[gw25 ~] $ping dp10
PING dp10.deeppurple.sharcnet (192.168.2.10) from 192.168.3.25 : 56(84) bytes of data.
64 bytes from dp10.deeppurple.sharcnet (192.168.2.10): icmp_seq=0 ttl=62 time=0 usec
64 bytes from dp10.deeppurple.sharcnet (192.168.2.10): icmp_seq=1 ttl=62 time=0 usec
64 bytes from dp10.deeppurple.sharcnet (192.168.2.10): icmp_seq=2 ttl=62 time=0 usec
64 bytes from dp10.deeppurple.sharcnet (192.168.2.10): icmp_seq=3 ttl=62 time=0 usec
```

```
--- dp10.deeppurple.sharcnet ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.000/0.000/0.000/0.000 ms
[gw25 ~] $
```

The zero RTT time shows that the precision of *ping* is not high enough for low-latency networks. For throughput experiments, we used three popular network benchmarks, *nuttcp*, *netperf* and *iperf* (all latest versions), to test the local throughput of an idle machine in greatwhite:

```
[gw20 ~]$ uptime
11:05am  up 40 days, 22:05,  1 user,  load average: 0.00, 0.00, 0.00
[gw20 ~]$ cd nuttcp-5.1.3/
[gw20 ~/nuttcp-5.1.3]$ ./nuttcp -t localhost
 1189.1250 MB /  10.00 sec =  997.4572 Mbps 41 %TX
[gw20 ~/nuttcp-5.1.3]$ cd ../netperf-2.2pl5/
[gw20 ~/netperf-2.2pl5]$ ./netperf -H localhost
TCP STREAM TEST to localhost
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
87380  65536   65536    10.00    955.92
[gw20 ~/netperf-2.2pl5]$ cd ../iperf-1.7.0/
[gw20 ~/iperf-1.7.0]$ ./iperf -c localhost
------------------------------------------------------------
Client connecting to localhost, TCP port 5001
TCP window size: 64.0 KByte (default)
------------------------------------------------------------
[  5] local 127.0.0.1 port 51984 connected with 127.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec  1.12 GBytes   983 Mbits/sec
[gw20 ~/iperf-1.7.0]$
```

The results varied with different implementations: the *nuttcp* reports the highest throughput (997Mbps) and *netperf* reports lowest throughput (956Mbps). It is difficult to say which measurement is more accurate and better reflects reality. The difference may be due to different default settings, and the implementation details of the tools.

This creates problems for network performance analysis. If tests produce abnormal results, we must trace the cause from the network subsystem, and the source code of the implementation as well.

All of the tools listed above have their own design methodologies and different sets of tunable options. However, they are still limited in some functionality and do not include tunable options for a number of interesting parameters. For instance, none of the tools were able to test non-blocking communication. The cross-platform design of these tools makes the code very complex and inefficient. For example, the *netperf* utility consists of

more than 40,000 lines of C code. In such a case, it is quite hard to tune and inset new source code for additional functionality.

As a result, it was best to implement our own network benchmark tool – Hpcbench, focusing specifically on the needs and requirements of HPC systems. Hpcbench is designed to measure high-speed, low-latency communication networks. The objectives of Hpcbench include:

- High accuracy and efficiency;
- Support for UDP, TCP and MPI communications;
- Tunable communication parameters of interest in an HPC environment;
- Detailed recording of test results, communication settings, and system information.

The implementation of Hpcbench is based on active, intrusive measurement since we are primarily interested in measuring or benchmarking network behavior and performance.

The MPI benchmarks that test MPI communications actually evaluate the efficiency of the MPI implementations. With the same benchmark and same system, the results may vary significantly for different MPI implementations, such as MPICH and LAM/MPI. We do not consider specific MPI benchmarks in our survey, since they are typically designed to examine all MPI functions that involve process communication, instead of network communication, and most of them are not of interest in our current study. We only test the blocking and non-blocking point-to-point MPI communication, which is typically associated with TCP end-to-end communication in the network. All other MPI communications, such as collective operations, are not included in the initial version of our benchmark.

3.2 Network Performance Metrics

When we discuss network performance, we mainly refer to its four attributes: throughput, latency, jitter, and loss rate. From the user's perspective, they are related to speed, delay, stability, and reliability.

Jitter is the variation of transmission delay in the network, mainly due to network congestion. Jitter may cause some problem for real-time applications, and jitter is usually associated with discussions of quality of service (QoS). Jitter is difficult to accurately

measure in HPC networks, since the interval between consecutive packets is very short. Benchmark overhead may affect the results of experimentation if we attempt to do timings for each packet. As a result, we have elected not to implement jitter measurements in the initial version our benchmark.

The packet loss rate is easy to measure in the application layer when an unreliable transport is used. In UDP communication, suppose the datagram size is less than the MTU size for the network. In this case, the receiving side retrieves the total number of sent packets from the last packet's sequence number (added in application layer), and the number of lost packets can be counted by subtracting the number of received packets. However, if the communication protocol used is reliable, such as TCP and MPI, the measurement of packet loss rate is not trivial, as the transport hides loss by retransmitting missing or corrupted packets. For example, consider TCP communications. The best way to compute packet loss in this case is to analyze all input and output packet headers for a target TCP connection. In order to do this in a typical Unix system, administrative privileges are needed to install the packet capture library *libpcap* [54], configure the network interface cards into a promiscuous mode, and run the appropriate program. Because of these restrictions, we will not implement the measurement of packet loss for TCP and MPI in this initial benchmark implementation.

The terms network bandwidth and throughput are often used interchangeably. But this is, in fact, incorrect. Bandwidth is the data rate supported by the networks, and throughput is what we actually get in the real world, mostly at the application level. The bandwidth of Gigabit Ethernet is 10^9 bit/sec, but we can never reach this number in practice. Network benchmarks report the effective bandwidth, or achievable throughput, or simply throughput. Benchmarks typically have tunable options for testing different set of parameters which could lead to different throughputs. There is a maximum achievable throughput in a system for each benchmark. Our goal to be able to characterize throughput results with different parameters for a given HPC system.

To accurately measure network bandwidth, we should let the tested object (the network) be the bottleneck in the whole system. If the bottleneck is somewhere else, the measurement will not likely be correct. For example, if a slow machine with a poor

network interface card is used to test a high-speed network, the end results will most likely depend on that machine instead of the network. Second, the effect of the operating system should be minimized. Measurements could be affected when system resources are taxed and scarce. Benchmarks are applications and so can be affected by system resources, or a lack thereof. We should always attempt testing with two idle machines (no user programmers running) to measure the maximum throughput possible. Third, the benchmark itself should be well designed to minimize overhead. A good benchmark should not consume too many system resources, as this may degrade the system performance and thereby affect the benchmark itself. The minimization of benchmark overhead is crucial for testing a relatively slow system. Some expensive system calls may provide a higher precision calibration, but we should be mindful of the tradeoffs as well. Protocol overhead is another consideration. For example, the relative protocol overheads in increasing order are UDP, TCP, and MPI; thus, the measured throughput of UDP should be higher than TCP's, which is higher than MPI's, if the benchmark is well designed and fine tuned properly.

Hpcbench is designed to measure UDP/TCP/MPI unidirectional and bidirectional throughput. For TCP and MPI communication, both blocking and non-blocking communication can be examined. To avoid benchmark overhead, Hpcbench attempts to minimize system calls for throughput tests.

Generally, network latency refers to the delay introduced by the network, specifically the time it takes for a small packet to traverse from one end to the other in the network. This definition, however, is still unclear. The small amount of data could be one byte or ten bytes, and the end could be the physical connection point of a network interface card or at the application layer.

In most cases, we are only able to measure the network latency using an application. With application level latency measurement, the time synchronization of the two end systems involved in the communication can be a problem. Even using NTP (Network Time Protocol), there is still some difference and drift over the synchronized time. Consequently, researchers often use a request/response model—an application sends a request to destination and waits for the response from the other end, and then divides the

measured time by 2 to approximate one-way network latency. This approach only uses one clock in the source host without needing to consider the time synchronization issue. There are two assumptions in this approach, however: the network has symmetric delays and the processing time for responding to the request in the destination host is trivial.

In this thesis, we use the term round trip time (RTT) for our network latency tests. We also need to specify the application or protocol for the tests, such as UDP round trip time. Since the ICMP-based *ping* utility is widely used, we also use UDP *ping*, TCP *ping*, or MPI *ping* in our discussion, corresponding to the UDP, TCP or MPI RTT test respectively.

Hpcbench is designed to be able to measure the UDP/TCP/MPI round trip times. Since one RTT in HPC systems can be very short (at the microseconds level), we iterate the request/response many times to minimize the effects of a timer's precision. So, in essence, our RTT tests are actually blocking ping-pong tests. Hpcbench automatically calculates the iteration for all RTT tests (refer to Section 3.5 for more details).

3.3 Communication Model

Hpcbench was written in C and uses the MPI API. It is comprised of three independent sets of benchmarks measuring UDP, TCP, and MPI communications. The UDP and TCP communications are via BSD sockets.

In TCP/IP communication tests, our client/server model includes two channels during the testing: a data channel and a test channel. The first is a reliable TCP connection for critical data communication and the real test channel (UDP or TCP) is over the other channel, as shown in Figure 3-1.

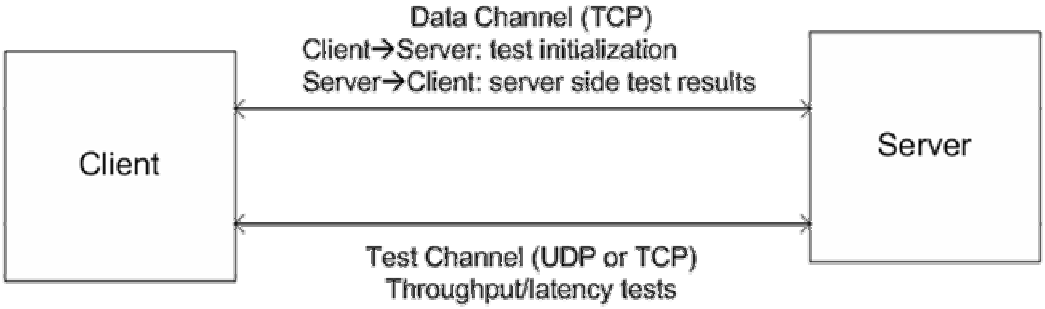

Figure 3-1 Two-channel Communication Model

This two-channel design makes it easier to control the tests (we might repeat tests many times), especially for UDP tests. The data channel is used by Hpcbench for control of the tests and only involves data transfer before each test starts and after each test ends. Thus, it does not introduce significant extra traffic or overhead during actual testing.

Another reason we use two connections for the tests is due to BSD socket settings. Some socket options, such as socket buffer size settings (*setsockopt()* system call), should be set before the *listen()* system call in the server and *connect()* system call in the client. If there is only one connection between the client and server, we must start the server process with a long argument set according to the client's test setting, and we have to restart the server process with different parameters every time the test mode changes. With two connections, the client is able to send all the test parameters to the server by first establishing the data channel and then creating the test channel with the desired options. Thus, the server process does not have to be concerned with test parameter settings during startup.

MPI communication is on top of the network subsystem of the operating system. It is much easier to control the test procedure involving MPI, since most detailed implementations of communication is transparent to the application layer. For point-to-point communication tests, we only need to specify the MPI send and receive functions, data type, and data size for master and slave processes.

3.4 Timers and Timing

All operating systems need a way to measure and keep track of time on the system. A high-precision timer is vital for all kinds of benchmarks.

A typical Linux system, for example, includes several types of clocks. A real-time clock (RTC) is a battery-powered hardware clock, which is independent of CPUs and other devices. The RTC is used to keep time and date information even when the computer is turned off. It is possible to access and change this time through the kernel (such as the usage of NTP), but it is very expensive and rarely used by regular applications.

The second is an architecture-dependent CPU Cycle Counter. The CPU Cycle Counter has a very high resolution (1 nanosecond for a 1GHz CPU). For example, Intel processors

(Pentium or later) include a 64-bit Time Stamp Counter (TSC register) that is updated by hardware at each CPU clock signal, which can be read by *rdtsc* assembly instruction. In the network subsystem, each packet timestamp is created by this counter. The platform-dependent CPU cycle counter is the highest resolution timer available for the systems. Unfortunately it cannot be accessed in a general, platform-independent way very easily.

The third clock is the Programmable Interval Timer (PIT), also called kernel timer or kernel clock. Linux uses the PIT to produce the "heartbeat" for the kernel to keep track of the time. The heartbeats are issued by means of hardware interrupts. The time interval of each heartbeat is called a tick with coarse granularity (e.g., 10ms for Intel x86 systems and around 1ms for Alpha systems). The term of *jiffies* refer to the number of ticks since system boot time. Shorter ticks result in higher resolution timer, which can speed up the response time of I/O multiplexing (the *select()* system call, for instance). The trade-off is that the CPU spends more of its time in kernel mode and less time in user mode, which will slow down user programs. In Linux boxes, the resolution of the PIT (tick) is defined by the *HZ* constant (frequency of *HZ* per second) in */usr/include/asm/param.h,* and it is not allowed to be changed after kernel compilation.

For some SMP systems, each CPU may have a local timer called the APIC (Advanced Programmable Interrupt Controller) timer that produces interrupts similar to the PIT's. The APIC timer is mainly used for kernel scheduling. The APIC is seldom used by applications.

From the user space, there are a number of system calls that are related to these timers:

- *clock()*: returns an approximation of process time in clock ticks (*clock_t* type) used by the program. The ticks are counted from an arbitrary point in the past such as system or process startup time. To convert to the number of seconds, the value should be divided by the constant *CLOCKS_PER_SEC*.
- *times()*: returns the process time of process in clock ticks with data structure *tms*. The structure includes the user/system times of the process and its children.
- *getrusage()*: returns the current resource usage in a data structure of type *rusage*, including user/system time used and some other process information.

- *time()*: returns the number of seconds (*time_t* structure) since the Epoch defined as 00:00 (midnight) of Jan. 1, 1970 (UTC).
- *gettimeofday()*: returns the value of elapsed time since the Epoch with resolution of microseconds, with the *timeval* data structure that contains the number of seconds and microseconds.
- *clock_gettime()*: a POSIX system call that returns a *timespec* structure that contains the number of seconds and nanoseconds. Typically *clock_gettime()* has a higher resolution than *gettimeofday()*. By default, *clock_gettime()* is not available on most Linux distributions with kernels before 2.5.x, including RedHat, but is available through kernel patches [55].

The *clock(), times()* and *getrusage()* system calls are associated with process (virtual) time, while *time(), gettimeofday()* and *clock_gettime()* system calls are considered as the real elapsed time, or wall-clock time. Hpcbench uses *getrusage* to trace the process usage and use *gettimeofday()* to record the time spent for the tests.

Figure 3-2 shows the resolutions of these system calls. Although the process times (*ru_utime* and *ru_stime*) in the data structure *rusage* returned by *getrusage* include microseconds, its resolution is limited to the HZ value (tick), since its implementation is based on the kernel timer.

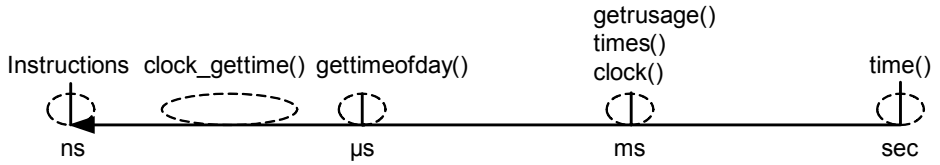

Figure 3-2 Time Scale of Computers with 1GHz CPU (HZ=1000)

The following code tests the precision of *gettimeofday()* system call by invoking the call 100000 times.

```
#include <stdio.h>
#include <sys/time.h>

#define NUM 100000

int main (int argc, char *argv[]) {
    int i, count=0, loop=0;
    int elapsed_time[NUM]={0}, resolution[NUM]={0}, cost[NUM]={0};
    struct timeval start, end;
```

```
    gettimeofday(&start, NULL);
    for ( i = 0; i < NUM; i++ ) { // Repeat calling gettimeofday() many times
        gettimeofday(&end, NULL);
        elapsed_time[i]=(end.tv_sec-start.tv_sec)*1000000+(end.tv_usec-start.tv_usec);
        start = end;
    }
    for ( i = 1; i < NUM; i++ ) { // Calculate the resolution and its idle loops
        if ( (resolution[count]=elapsed_time[i]-elapsed_time[i-1])>0 ) {
            cost[count++] = loop + 1;
            loop = 0;
        } else
            loop++;
    }
    printf("Output: Resolution[usec] Overhead[usec]\n");
    for ( i = 0; i < count; i++ ) // Print out the results
        printf("%d  %f\n", resolution[i], resolution[i]*1.0/cost[i]);

    return 0;
}
```

For comparison, we tested the resolution of *gettimeofday()* in several platforms, including Alpha, Intel Xeon and Pentium, SPARC and SPARC SMP systems. The results are summarized in Table 3-1. The cost of the system call is also included. We can see the SMP systems spent more time on *gettimeofday()* invocation. This is probably due to some form of kernel synchronization.

Architecture	OS	Resolution (μsec)			System call cost (μsec)		
		Min	Avg	Max	Min	Avg	Max
HP Alpha SMP[1]	Linux 2.4.21	976	976	977	0.57	0.61	0.86
Intel Xeon SMP[2]	Linux 2.4.20	1	1	1	0.33	0.47	0.50
Intel Pentium[3]	Linux 2.4.20	1	1	1	0.14	0.17	0.20
Sun Sparc SMP[4]	SunOs 5.8	1	1	67	0.17	0.49	67
Sun Sparc[5]	SunOs 5.8	1	1	213	0.06	0.15	106.5

Table 3-1 The Resolution and Overhead of *gettimeofday()* in Different Architectures

Results show that the resolution of *gettimeofday()* is around 1μsec in most cases, which is as we expected. But in the Alpha SMP systems, it drops to lower than 1ms. Why? Let us take a closer look of how *gettimeofday()* works in a Linux system. *gettimeofday()* leads to kernel function *sys_gettimofday()* call that is defined in */Linux-src/kernel/time.c*, which invokes *do_gettimeofday()* to retrieve the current time. *do_gettimeofday()* is implemented in */Linux-src/arch/i386/kernel/time.c* for Intel architectures and in */Linux-src/arch/alpha/kernel/time.c* for Alpha systems. It actually updates the kernel's global variable *xtime* (kernel clock). In the Intel architecture, the *do_fast_gettimeoffset()*

[1] Tested on an idle machine in the greatwhite [2] Tested on an idle node in the mako [3] Tested on an idle desktop with Pentium 500MHz CPU [4] Tested on algernon.csd.uwo.ca [5] Tested on durendal.syslab.csd.uwo.ca

function in *time.c* is invoked to compute the precise time in microseconds after the last tick. This is done by reading the high-precision CPU cycle counter (TSC register) with assembly language. On the other hand, the Alpha system does not do this for a finer granularity, and all computation is based on *jiffies*. The reason not to read the CPU cycle counter for the Alpha system is that in the Alpha system, only the low 32-bits of the CPU cycle counter are readable, and this value will wrap around in a very short time (about 4 seconds for a 1GHz CPU), making this high-precision timer unusable for most timing tools. Thus, *gettimeofday()* only has the resolution of a kernel clock tick, which is equal to 1000000/1024 = 976.5 μsec (*HZ*=1024 in Alpha systems), agreeable with our measured results.

Even though *gettimeofday()* in the Alpha system has a relatively low resolution, it is still the best system call we can use for timing currently. To improve accuracy during testing, we can increase the test workload and test time. For example, if the test time is more than 5 seconds, the error caused by the timing will be less than 1/5000=0.02%, which is acceptable in most experiments.

For latency (RTT) tests, we only measure time using the client's timer. For throughput tests, we must consider timing synchronization between the client and the server. One approach is to put a timestamp in each packet. The other end (client or server) compares the local time to the time in the packet to get a time offset for the sender. To take the network latency into account, a round trip time test can be conducted.

This approach is not perfect. First, as we mentioned, network latency may not be equal to ½ RTT. Second, getting a high-precision local time is expensive (refer to Table 3-1). If the *gettimeofday()* system call is invoked for each packet sent or received, the total cost is considerable, especially in high-speed networks which support tens of thousands packets per second. Third, as we have seen, the Alpha systems only support 1ms timer resolution, making any other effort for time synchronization not applicable.

Our solution to this issue is a high level synchronization. We do not synchronize the timer of the two end machines. Instead, the client and server notify each other before each test starts. After this initial communication, both the client and the server start timing, and then compute throughput by their local elapsed time separately. With this

approach, the *gettimeofday()* is only called at the start and the end point of each test rather than one invocation for each packet sent or received. We will discuss this communication synchronization in the next section.

All MPI implementations include a *MPI_Wtime()* function call that is able to select the best timer for MPI communication timing. We will use this function call in our MPI benchmark. In fact, MPICH also invokes *gettimeofday()* if it is available.

3.5 Iteration Estimation and Communication Synchronization

In HPC environments, the round-trip time is too short to be measured by a relatively low-precision timer, as the *ping* test example showed in Section 3.1. Timer resolution may also affect the results of throughput tests. For example, in a Gigabit Ethernet, a message of 1 MB can be sent out in less than 0.01 seconds. To reduce the effects of relatively poor timer resolutions, test durations should be long enough to minimize this effect. Consequently, we repeat transmission of the same message many times to measure network throughput and network latency. In Hpcbench, the number of times to repeat transmission (iteration) is computed by an estimation test conducted before the real test starts. Estimation test starts from a small number of repetitions (iteration), and compares the elapsed time with the defined test time. The repetition number will exponentially increase until the elapsed time is greater than the defined test time, then the iteration number for desired test time can be calculated. With this approach, Hpcbench is able to measure the network latency and throughput with nearly any message size and desired test time in high performance networks.

Most network measurement tools such as *netperf* and *iperf* conduct throughput tests for a time period with a default sending size, and their message size is not configurable. This can be inconvenient and unnecessarily limit the test options that are available during experimentation. This limitation may have been introduced as TCP/IP was treated as a stream protocol instead of a message protocol, with the rationale that there is no difference between sending 100MB and 100KB, because all data are sent by system calls that divides this data into smaller packets. This is a common misconception, but is slightly inaccurate for detailed performance analyses. Consider the case of sending 100MB of data through the network. First, consider an implementation in which all data

to be sent can be buffered in memory. The function *write(socket, buffer, data-size)* is then repeatedly called until all of the data has been sent. Second, consider the case where there is only a small memory buffer, say with a relatively small default size of 4KB or 8 KB. Now the function *write(socket, buffer, default-size)* is again called many times until all data has been sent out. The second method is used by all of the benchmarks we have examined and mentioned earlier in this thesis. Are these two methods the same? In the end, it depends on the underlying socket implementation. One system call *write(socket, buffer, 100MB)* could (and likely would) send more than 8KB. In such a case, the number of *write()* system calls in the first case will be less than that of second case for the same total amount of data transferred, which may lead to different results. This is especially true for a network that supports large frames (refer to Section 4.1.2). Furthermore, even the resulting memory access with different patterns may result in different performance. As a benchmark, it is important to consider these different situations. Hpcbench supports tunable options for message size and the data size of each transmission as well. If the latter is not defined, then 8KB is used for TCP tests and 1460-bytes for UDP tests. To avoid the effects of potential data compression functionality in networks, Hpcbench also carries out a randomization process for the data being transmitted.

Another pre-test is the synchronization procedure that ensures that the client and the server have the same baseline for timing. This test should be conducted in the data channel. The synchronization before the start of timing is pre-synchronization and starts by sending a tiny SYNC1 packet (2 bytes) from the client. When the server receives this packet, it bounces back a small ACKSYNC packet (2 bytes) to the client. The client receives this acknowledgement, and sends another SYNC2 packet (2 bytes) to the server. At this point, the client starts the timing for the test. The server will start timing after receiving the SYNC2 packet. The purpose of SYNC2 is to let the server be aware of the RTT value. Now both client and server can start actual testing, and have some measure of the round trip time of the link when the test began.

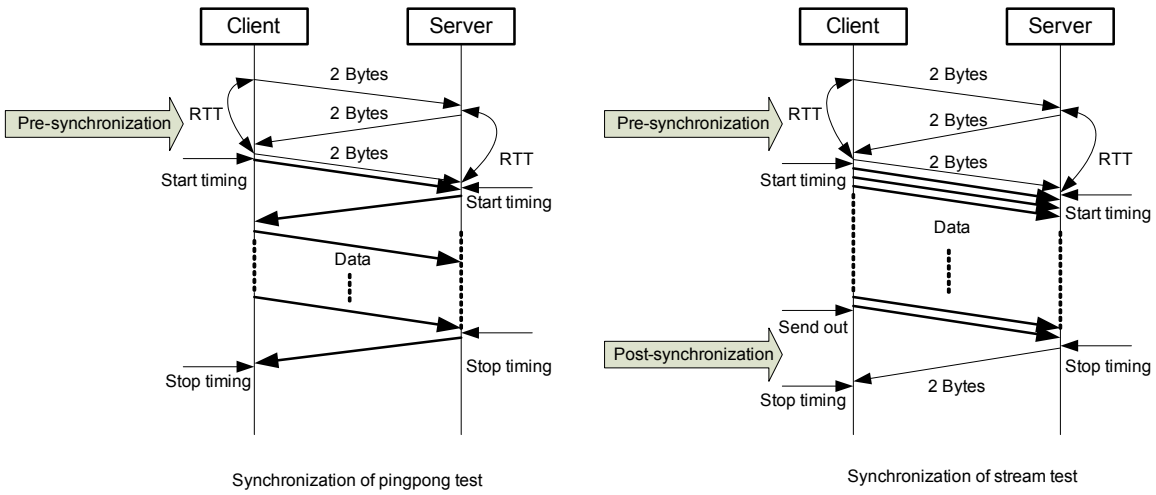

Figure 3-3 Communication Synchronization

For unidirectional stream tests, we have a post-synchronization step to help measure the computation of the client's throughput, as illustrated in the right side of Figure 3-3. This is because the client may complete before the server receives the data. When the send function returns, the data will first be buffered in the kernel space. The amount of buffered data could be as large as 1MB, or possibly higher, if the socket buffer is large enough. The time that it takes to send this data out to the network depends on the system workload and the congestion of the local network (LAN). The time for the server to receive all of this data depends on network bandwidth and network congestion. We take both of these possible delays into account when we calculate the throughput on the client side. This post-synchronization does introduce additional overhead (about ½ RTT). To produce more accurate results, we can subtract this ½ RTT to compute the elapsed time on the client side for the post-synchronization process, and for server we can add ½ RTT for the pre-synchronization to compute throughput on the server side.

The communication synchronization model shown in Figure 3-3 has been applied to Hpcbench for TCP and MPI communication tests. In UDP tests, the post-synchronization step is unnecessary since UDP communication is connectionless. The UDP local throughput is computed when the last packet is sent out (e.g. *write()* returns). To inform the server of the termination of a test, the client sends a number of UDP FIN packets (the number is defined by MAXFIN in *udplib.h*) to the server, assuming that the server is able to receive at least one of them. The server stops timing when this signal is received,

computes the server side throughput and packet loss rate, and then sends the server side statistics to the client on the TCP data channel. If all UDP FIN packets are lost, the result of this round of experimentation is ignored, as an extremely abnormal event must have occurred in the network to produce this effect.

While we have strived to achieve as high an accuracy as possible in our measurements, in high performance networks, some additional functionality of the NICs (e.g. interrupts coalescence) could cause some unforeseen delays, and slightly reduce the accuracy we would ordinarily expect. We will discuss this issue further in the next chapter.

3.6 System Resource Tracing

System resource limitation can greatly affect network performance. For instance, the performance of an entire system (including the network subsystems) would dramatically degrade when the system's physical memory is completely used and extensive page faulting occurs. As well, system load can also affect network performance. We test the network throughput between two nodes using *iperf*. We select three nodes in hammerhead to test their throughput, where the idle node hh5 ran as client, the idle node hh20 ran as server, and the busy node hh10 (100% usage) ran as another server:

```
[hh5 ~/iperf-1.7.0] $ uptime
  8:12pm  up 16 days,  7:02,  1 user,  load average: 0.00, 0.00, 0.00
[hh5 ~/iperf-1.7.0] $ ./iperf -c hh20
------------------------------------------------------------
Client connecting to hh20, TCP port 5001
TCP window size: 64.0 KByte (default)
------------------------------------------------------------
[  5] local 192.168.4.5 port 52213 connected with 192.168.4.20 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec   624 MBytes   518 Mbits/sec
[hh5 ~/iperf-1.7.0] $./iperf -c hh10
------------------------------------------------------------
Client connecting to hh16, TCP port 5001
TCP window size: 64.0 KByte (default)
------------------------------------------------------------
[  5] local 192.168.4.5 port 52214 connected with 192.168.4.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.1 sec  9.84 MBytes  8.19 Mbits/sec
[hh5 ~/iperf-1.7.0] $uptime
  8:13pm  up 16 days,  7:03,  1 user,  load average: 0.04, 0.01, 0.00
[hh5 ~/iperf-1.7.0] $
```

The *iperf* reports that the achievable TCP throughput between hh5 and hh20 was about 518 Mbps; while throughput between hh5 and hh10 was less than 10 Mbps. The difference of server's system load led to different achievable throughputs. At the same time, network communication tests also introduce workload to the system. Consequently,

besides the throughput itself, the system resources, such as the CPU utilization during the testing, is important for us to understand the system behavior. We would like to implement such functionality in Hpcbench.

Process resource usage statistics can be retrieved using the *getrusage()* system call, as mentioned in Section 3.4. The measurement of CPU resource consumption is not trivial, however. In most Unix systems, the CPU usage is measured by sampling the CPU status. This approach is highly platform-dependent. For example, the popular system monitor utility *top* includes more than 30 source files in its implementation, and supporting CPU measurement for different architectures accounts for a large portion of its code. Because of this, most network measurement tools do not trace system resources.

In a Linux system, we can measure the CPU usage by parsing files in the */proc* virtual file system (VFS). Linux's VFS is a wrapper implementation to handle all kinds of file system calls. It provides an interface for different file systems such as disk-based local file systems (Ext3, NTFS, XFS, etc.) and network file systems (NFS, AFS, Coda, etc.). The files in the */proc* directory are part of a special virtual file system that is not associated with a block device, but rather exists only in kernel memory. These special pseudo-files refer to different kernel memory areas that contain many kernel configuration parameters. This allows programs in the user space to access certain kernel information with ease. There are several subdirectories in the */proc* directory, with the numerical subdirectories corresponding to running processes, and others storing system kernel parameters and operating statistics. To trace system kernel resources related to networking, we are interested in four files in */proc*:

- */proc/interrupts*: records the number of interrupts for each IRQ.
- */proc/meminfo*: records physical and swap memory and their usage.
- */proc/stat*: records kernel statistics including CPU jiffies, pages in/out, swaps in/out, context switches, total interrupts, and so on.
- */proc/net/dev*: records network device status information such as the number of packets and bytes received and sent, the total collisions observed, and other statistics.

By parsing these virtual files, we are able to monitor a system's CPU usage and workload distribution, memory usage, the number of interrupts the kernel received, the number of interrupts each network interface card (NIC) raised, and other statistics for each NIC. The pseudo code for a client to monitor the system during throughput tests (fixed size, ping-pong mode) is as following:

```
start trace-system
sleep 1 second
stop trace-system
record the pre-test syslog
do iteration-estimation
for ( i = 1 to repeat ) do{
        inform server the iteration
        start trace-system
        start timing
        for ( j = 1 to iteration ) do {
                send data
                receive data
        }
        stop timing
        stop trace-system
        compute i^th throughput
        record i^th syslog
}
start trace-system
sleep 1 second
stop trace-system
record the post-test syslog
write all to log files
```

Figure 3-4 The Pseudo Code of System Resource Tracing for Throughput Tests

The pre-test and post-test system resources are logged for comparison purposes. The following illustrates the logged information:

```
[hh25 ~/hpcbench/tcp]$ ./tcptest -h hh26 –c -o out -r 3
 (1) : 521.247565 Mbps
 (2) : 522.272930 Mbps
 (3) : 521.964908 Mbps
Test done!
Test-result: "out"  Local-syslog: "out.c_log"  server-syslog: "out.s_log"
[hh25 ~/hpcbench/tcp]$ cat out.c_log
# hh25 syslog -- Mon Aug 3 18:25:30 2004
# Watch times: 5
# Network devices (interface): 1 ( eth0 )
# CPU number: 4

##### System info, statistics of network interface <eth0> and its interrupts to each CPU #####
#        CPU(%)   Mem(%) Interrupt  Page  Swap  Context          <eth0> information
#  Load User  Sys  Usage  Overall  In/out In/out  Swtich   RecvPkg   RecvByte  SentPkg   SentByte  Int-CPU0 Int-CPU1 Int-CPU2 Int-CPU3
0     0    0    0    17     4106       0     0       12         0          0         1         66         0        2         0         0
1    21    0   21    17    67269      32     0    12894     47529    3137248    223862  338915925         0    46857         0         0
2    22    0   21    17    67557       0     0    12746     48299    3190803    223866  338921537         0    47188         0         0
3    21    0   21    17    67508       0     0    12726     48328    3189688    223862  338916005         0    47132         0         0
4     0    0    0    17     4144     224     0        8         4        414         0          0         0        3         0         0
```

```
## CPU workload distribution:
##        CPU0 workload (%)          Overall CPU workload (%)
#  < load  user  system  idle >  < load  user  system  idle >
0     0.0   0.0    0.0   100.0      0.0   0.0    0.0   100.0
1     0.0   0.0    0.0   100.0     22.0   0.0   21.9    78.0
2     0.0   0.0    0.0   100.0     22.0   0.0   22.0    78.0
3     0.0   0.0    0.0   100.0     21.5   0.0   21.5    78.5
4     0.0   0.0    0.0   100.0      0.0   0.0    0.0   100.0

##        CPU1 workload (%)          Overall CPU workload (%)
#  < load  user  system  idle >  < load  user  system  idle >
0     0.0   0.0    0.0   100.0      0.0   0.0    0.0   100.0
1    60.5   0.0   60.5    39.5     22.0   0.0   21.9    78.0
2    60.5   0.0   60.5    39.5     22.0   0.0   22.0    78.0
3    61.0   0.0   61.0    39.0     21.5   0.0   21.5    78.5
4     0.0   0.0    0.0   100.0      0.0   0.0    0.0   100.0

##        CPU2 workload (%)          Overall CPU workload (%)
#  < load  user  system  idle >  < load  user  system  idle >
0     0.1   0.0    0.1    99.9      0.0   0.0    0.0   100.0
1    27.3   0.1   27.2    72.7     22.0   0.0   21.9    78.0
2    27.6   0.2   27.4    72.4     22.0   0.0   22.0    78.0
3    25.1   0.1   25.0    74.9     21.5   0.0   21.5    78.5
4     0.2   0.1    0.1    99.8      0.0   0.0    0.0   100.0

##        CPU3 workload (%)          Overall CPU workload (%)
#  < load  user  system  idle >  < load  user  system  idle >
0     0.0   0.0    0.0   100.0      0.0   0.0    0.0   100.0
1     0.0   0.0    0.0   100.0     22.0   0.0   21.9    78.0
2     0.0   0.0    0.0   100.0     22.0   0.0   22.0    78.0
3     0.0   0.0    0.0   100.0     21.5   0.0   21.5    78.5
4     0.0   0.0    0.0   100.0      0.0   0.0    0.0   100.0
[hh25 ~/hpcbench/tcp] $
```

These detailed reports of system resources are important for us to understand the interaction of the kernel and the network subsystem. In the above example, we know that the client system was originally idle; the system load was about 20% for TCP stream communication, and its workload was distributed to CPU1 and CPU2. We will look at this in more detail in the next chapter.

It is important to note that the system information traced by */proc* files is not accurately synchronized to each test because of the parsing delay and the deviation of updates of those virtual files. These statistics can consequently only show us a rough view of kernel information during tests.

3.7 UDP Communication Measurement Considerations

UDP communication is connectionless and so there are some subtle considerations in implementing UDP communication tests. In UDP latency (RTT) tests, the server is responsible for bouncing back the datagram it received. Generally, the server application may use the following code to measure the UDP latency:

```
  {  // Start example block
    timeOut.tv_sec  = 5;   // Maximum time 5 seconds
    timeOut.tv_usec = 0;

    for ( i = 0; i < iteration; i++ ) {
        FD_SET( serverUdpSocket, &readSet );
        rval = select(serverUdpSocket+1, &readSet, NULL, NULL, &timeOut );
        if ( (rval > 0) && (FD_ISSET(serverUdpSocket, &readSet))) { // Socket readable
            msg = recvfrom(serverUdpSocket, buffer, packetSize, 0,
                           (struct sockaddr*)&clientAddress, &addrSize);
            if ( msg < 0 && errno != EINTR ) { // Read error
                perror("UDP server receive");
                return -1;
            }
            if( sendto (serverUdpSocket, buffer, msg, 0,
                        (struct sockaddr*)&clientAddress, addrSize) < 0 ) {
                perror ("UDP server send");
                return -1;
            }
        } else if ( rval == 0 ) {  // Timeout
            perror("UDP server time out");
            return -1;
        }
    } // End of for loop
    return 0;
  } // End of example block
```

There is nothing wrong with above code; but, it is not optimized as the *select()* system call can be quite expensive. In Gigabit Ethernet, there can be tens of thousands of iterations for a one second test. This many system calls can introduce additional unwanted overhead for a RTT test. We can reduce the number of system calls by using a signal-driven approach, which is illustrated below:

```
int timeOutFlag;  // global variable
static void timeOutHandler() // Signal handler
{
    timeOutFlag = 1;
    return;
}
…

 { // start example block

    if ( signal(SIGALRM, timeOutHandler) == SIG_ERR ) {
        perror("Set alarm signal");
        return -1;
    }
    timeOutFlag = 0;
    alarm (5);   // Maximum test time 5 seconds

    for ( i = 0; i < iteration; i++ ) {
        if ( (msg=recvfrom(serverUdpSocket, buff, packetSize, 0,
                       (struct sockaddr*)&clientAddress, &addrSize)) < 0 ) {
            alarm(0); // Cancel timer
            perror("UDP server receive error");
            return -1;
        }
        if ( timeOutFlag ) {
            fprintf(stderr, "UDP server time out\n");
            return -1;
        }
        if ( sendto (serverUdpSocket, buff, rval, 0,
                     (struct sockaddr*)&clientAddress, addrSize) < 0 ) {
            perror("UDP server send error");
```

```
            alarm(0); // Cancel timer
            return -1;
        }
    } // End of for loop
    alarm(0); // Cancel timer
    return 0;
  } // End of example block
…
```

In this code, there is no extra *select()* system call inside the for loop. The system load was significantly reduced, and thus the accuracy of measurement can be improved in some relatively slow systems. Experiments showed that measured UDP RTTs were about 5 to 10 percent less than those measured using the previous approach, depending on the packet size (as tested in greatwhite).

However, there are two issues in the above implementation. First, there are two possible race conditions for the *alarm(5)* call in the above code. It is possible that the alarm signal occurs before the first *recvfrom()* call. It is also possible the alarm signal occurs after testing the *timeOutFlag* condition and before next *recvfrom()* call. In both cases, the server will be pending in *recvfrom()* forever if the awaited UDP datagram is lost. Second, in Linux (kernel 2.4.x), many system calls are automatically restarted when a SIGALRM signal handler returns, including *recvfrom()*. Consequently the time out signal handler does not function properly in this code.

To solve the problem, we use *sigsetjmp()* and *siglongjmp()* system calls. One might use the *setjmp()* and *longjmp()* system calls, but they may not work properly in some Linux SMP systems since the SIGALRM may be ignored by the kernel.

```
static sigjmp_buf alarm_env;    // Time out environment
static void timeOutHandler()    // Time out handler
{
    siglongjmp(alarm_env, 1);
    return;
}
…
  {  // Example block

    if ( signal(SIGALRM, timeOutHandler) == SIG_ERR ) { // Set signal handler
        perror("Set alarm signal");
        return -1;
    }
    if ( sigsetjmp(alarm_env, 1) != 0 ) { // Set the jump point
        error = TIMEOUT;
        return -1;
    }
    alarm (5);   // Maximum test time 5 seconds
    for ( i = 0; i < iteration; i++ ) {
        if ( (msg=recvfrom(serverUdpSocket, buff, packetSize, 0,
                    (struct sockaddr*)&clientAddress, &addrSize)) < 0 ) {
            alarm(0); // Cancel timer
```

```
            perror("UDP server receive error");
            return -1;
        }
        if ( sendto (serverUdpSocket, buff, rval, 0,
                    (struct sockaddr*)&clientAddress, addrSize) < 0 ) {
            perror("UDP server send error");
            alarm(0); // Cancel timer
            return -1;
        }
    } // End of for loop
    alarm(0); // Cancel timer
    return 0;
  } // End of example block
```

Exception and time out handling is very important in UDP communication because UDP datagrams may be lost during tests. In practice, we can use *setitimer()* instead of *alarm()* to get a finer alarm timer, and use *sigaction()* to more safely control the signals. Notice there is only one timer available to generate a SIGALRM signal for each process, and a new alarm timer will override the previous timer setting. Also note that the alarm timer should be canceled before the function normally returns.

If several time-out controls are needed at the same time, we can combine calls to *alarm()* and *select()*, or use nested *select()s*. There is one significant difference between the implementation of *select()* in Linux and many other variations of Unix, such as Solaris: Linux modifies the fields of *timeval* data structure passed into the function depending how much time is left when *select* returns, while most Unix systems keep it intact. The implementation should take care of this for different platforms when the time out control is placed inside a loop.

Since UDP communication is connectionless, it has no congestion and flow control mechanisms as TCP does. It is good to have a UDP traffic generator with throughput-constraint functionality so than we can simulate network traffic for some analysis, such as the study of QoS in congested environment. Consider a machine capable of sending UDP datagrams of size 1000 Bytes with 800Mbps throughput. The average length of each transmission is:

time_interval = total-sent-bits / throughput = 1000 x 8 / 800 = 10 μsec

If we have the sender pause 10 μsec between each send function call, the throughput will be around 400Mbps. Similarly, a 5 μsec delay interval may result in 600Mbps throughput:

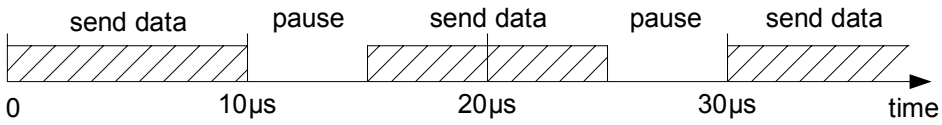

Figure 3-5 Ideal Pause for UDP Sender

This approach looks reasonable; however, there are no system calls with this kind of high-precision delay. The *usleep()* and *nanosleep()* system calls imply that they have microsecond and nanosecond resolution respectively, but this may not actually be the case in reality. We conducted resolution tests for these two system calls in different platforms with similar testing to what was used earlier with *gettimeofday()*. Table 3-2 lists the results, showing that both *usleep()* and *nanosleep()* are coarse-grain in resolution. The reason is that these two system calls are based on the kernel timer, which is low-precision as we discussed earlier in Section 3.4.

Architecture	OS	Elapsed time in µsec for *usleep()* with different parameters of 0, 1, 10, 10000, and 100000. (average of 100 tests)				
		0	1	10	10000	100000
HP Alpha SMP	Linux 2.4.21	976	1953	1953	11719	101570
Intel Xeon SMP	Linux 2.4.20	9916	19999	19999	19999	110000
Intel Pentium	Linux 2.4.20	9944	19999	19998	19996	110127
Sun Sparc SMP	SunOs 5.8	1	19979	19990	19993	109995
Sun Sparc	SunOs 5.8	1	19989	19991	19996	109997
Architecture	OS	Elapsed time in µsec for *nanosleep()* with different parameters of 0, 1, 10, 10000, and 100000. (average of 100 tests)				
		0	1	10	10000	100000
HP Alpha SMP	Linux 2.4.21	976	1953	1953	2011	101853
Intel Xeon SMP	Linux 2.4.20	9998	19999	19999	19999	110000
Intel Pentium	Linux 2.4.20	10012	19992	19998	19998	109969
Sun Sparc SMP	SunOs 5.8	3	4	4	9994	107292
Sun Sparc	SunOs 5.8	2	4	4	9998	105799

Table 3-2 The Elapsed Times for *usleep()* and *nanosleep()* System Calls

The same problem occurs to the *setitimer()* system call. We could not rely on the *setitimer* timer function for high-precision delay because of the usage of low-resolution kernel timer, although it includes fields with microsecond resolution. We use a high level self-judgment approach to ensure the sending rate converges to the specified throughput.

If the real-time sending rate at a given moment is greater than the specified transfer rate, a longer time for delay, corresponding to the previous delay time, will occur before the next transmission; otherwise the delay between each transmission will decrease, as illustrated as follows:

```
void delay_usec(int usec) // Pause for microseconds
{
    struct timeval startTime, endTime;
    int delay = 0;

    if ( usec <= 0 ) return;
    gettimeofday(&startTime, NULL);
    while ( delay < usec) {
        gettimeofday(&endTime, NULL);
        delay = (endTime.tv_sec - startTime.tv_sec) * 1000000
            + endTime.tv_usec - startTime.tv_usec;
    }
    return;
}
…
  { // example block
    if ( targetThroughput > 0 ) { // Init the delay if throughput constraint defined
        delayTime = (8*1000000LL*dataSize/targetThroughput);
        if ( delayTime < 1 ) delayTime = 1;
    }
    gettimeofday(&startTime, NULL); // Start timing
    do {
        if ((rval=sendto(udpSocket, buff, dataSize, 0,
                  (struct sockaddr *)&serverAddress, sizeof(serverAddress))) < 0 ){
            perror("UDP send error");
            return -1;
        }
        sentBytes += rval;
        sentPackets ++;
        gettimeofday(&endTime, NULL);
        elapsedTime = (endTime.tv_sec - startTime.tv_sec) * 1000000LL
                    + endTime.tv_usec - startTime.tv_usec;
        if ( targetThroughput > 0 ) { // Adjust the delay between each sending
            if ( elapsedTime > 0 )    // Compute the current throughput
                throughput = sentBytes*8*1000000/elapsedTime; // bit/sec
            if ( throughput > targetThroughput ) // Increase delay
                delayTime++;
            else if ( delayTime > 0 ) // Reduce the delay
                delayTime--;
            delay_usec(delayTime);
        } // End of delay adjustment
    } while ( elapsedTime < targetTime );
  } // End of example block
…
```

In Hpcbench, when a throughput constraint is defined, the UDP sender self adjusts its sending pace to achieve the target throughput. When the –g option is specified, Hpcbench works as a UDP traffic generator with desired throughput and packet size. In such a case, the target host can be any networked machine, regardless of whether or not it is running a UDP server process, since UDP is a connectionless protocol. In such a case, it is important to remember, however, that the injected UDP traffic could affect the performance of the network and the target machine.

3.8 Summary

In this chapter, we presented a survey of network benchmarks, and discussed key issues for benchmarking in High Performance Computing environments. We introduced Hpcbench, a Linux-based network measurement tool, and discussed its design methodology, including its communication model, test timing, synchronization, benchmark overhead.

Hpcbench was carefully designed to work with high performance networks to provide accuracy, efficiency, and functionality. It is capable of logging test result details and detailed system information of both a client host and server during the tests, including CPU utilization, memory usage, interrupts, and so on. We will use Hpcbench to examine the network performance of Gigabit Ethernet, Myrinet and QsNet in the next two chapters of this thesis.

Chapter 4

Investigation of Gigabit Ethernet in HPC Systems

In this chapter, we present results from a study of the performance of Gigabit Ethernet in HPC systems using Hpcbench. We begin with a discussion of underlying Gigabit Ethernet technologies, and then report on several studies of its performance. Specifically, we look at interrupts coalescence, jumbo frame size, and zero-copy techniques that are associated with modern Gigabit Ethernet technologies. We then explore the interactions between network communications and the system kernel in Alpha and Intel Xeon architectures, and study the network performance associated with various configuration parameters and settings.

4.1 A Closer Look at Gigabit Ethernet

In this section, we first review a number of technologies that have been used to implement Gigabit Ethernet and improve its overall performance. We examine properties of the protocol itself, interrupt handling and the use of jumbo frames, and finally zero copy techniques for optimizing packet processing.

4.1.1 Protocol Properties

Like traditional Ethernet, Gigabit Ethernet was designed for a connectionless delivery service based on the IEEE 802.3 protocol. From application point of view, the difference between fast Ethernet and Gigabit Ethernet is only a matter of speed. The communication scheme and protocol stack remain the same for different Ethernet technologies. On top of the Ethernet scheme, TCP/IP protocols tend to be deployed to support data communication. Within the TCP/IP stack, the UDP protocol is connectionless and unreliable; on the other hand, the connection-oriented TCP communication provides reliable data delivery.

To support fair and reliable communication, TCP introduces flow control and congestion control mechanisms. The goal of flow control is to balance the sender's transfer rate

against the receiver's data accepting ability. In this case, a sliding window is used to control the transmission rate. The window size is determined by the "advertised window" field in the receiver's acknowledgement (ACK) packets, where the receiver tells the sender how much data it can still accept. If the receiver's buffer is full, or its system is too busy to deal with more data at the moment, it acknowledges the sender with a window size of zero (close window).

The original TCP specification (RFC793) only supported a 16-bit window size, or 64 KB limit. This value, however, is not large enough for some circumstances. RFC 1323 (TCP Extensions for High Performance) defines a window scaling extension to solve this problem, where the Window Scale Option is used to calculate a 32-bit value in the 16-bit window field of the TCP header. This improvement supports up to a 4 GB TCP window size.

Congestion control tries to balance the sender against the network by trying to limit the sender's transmission speed to match the network's capacity. It introduces a second *cwnd* window (congestion window) for the sender. The idea is to observe data loss during transmissions. If there is no loss (all packets are acknowledged in a specific time interval), the congestion window increases, allowing the sending speed to increase as well; otherwise the congestion window is reduced. Generally, four types of mechanisms are used for TCP congestion control: slow start (*cwnd* starts from 1 and then increases exponentially), congestion avoidance (*cwnd* increases slowly by 1 after a threshold), and fast retransmit and fast recovery (resend loss packets depending on the retransmission timeout value (RTO) and duplicate ACKs, and halves the value of threshold). Notice that all kinds of data loss and unexpected delays of packets could result in the reduction of the congestion window, although the real reason may not be network congestion.

In TCP communications, the sender will not allow a stream of in-flight data greater than the minimum of the sliding window and the congestion window at any time. This can be considered a bottleneck of the network outside the sending machine. People often refer to this minimum window as the TCP control window, or simply the TCP window. This TCP window size determines how many bytes a sender is allowed to send out while waiting

for acknowledgments from the receiver. Therefore, the TCP sending rate is roughly equal to Window-size/RTT.

To prevent tiny packet congestion, the Nagle algorithm (RFC896) is also implemented in most TCP/IP implementations. The Nagle algorithm will delay sending a small block of data (waiting for more data to send together) when the outstanding window (unacknowledged packets) is relatively large. The Nagle algorithm can reduce a lot of overhead when the application produces many small packets; such as in X window systems, but it can have a negative effect on benchmarking, especially for network latency tests. In many TCP/IP implementations, the TCP_NODELAY option can be used to turn off the Nagle algorithm.

In Linux systems, there is usually an extra TCP_CORK option (the opposite of TCP_NODELAY) to avoid delays by small packets in bulk data transfers. The idea behind TCP_CORK is to have small packets padded with bulk data and send them immediately. The partial frame will be pending to be "corked" (padded) for a while. An example of using TCP_CORK is for Web servers, sending the header and the body of HTTP response together.

TCP transmission control can have a negative impact on performance; however, it is necessary to prevent a congestive collapse in a shared network. In order to improve TCP performance, many suggestions and extensions have been proposed over the past decade. Some have been accepted as standards and implemented in many operating systems, such as TCP SACK [75] and TCP Reno [73][74]. Other proposals are still under investigation. Two new TCP proposals are of interest to those who working with high performance networks: HighSpeed TCP (RFC 3649) [59][86] and Fast TCP [58].

In Gigabit Ethernet, the possibility of data loss is typically low, and TCP congestion control mechanisms can result in lower throughput and higher RTT. UDP does not include any transmission control as TCP does; thus, it tends to have less overhead for communication. So, in theory, UDP communication statistics should be closer to the actual network characteristics than other protocols. On the other hand, MPI communications can have more protocol overhead than TCP, since MPI has its own synchronization, timing, and delay detecting mechanisms on top of TCP.

4.1.2 Interrupts Coalescence and Jumbo Frame Size

Gigabit Ethernet switches only need to retrieve the Ethernet header of each packet, and then forward it out the appropriate port. This can be relatively easy to implement by hardware (packets are only stored in SRAM, for instance). In the end hosts, however, network processing is not that trivial. To achieve Gbit/sec level throughput, the system has to guarantee enough bandwidth and system resources for all subsystems involved in the handling of network traffic. Most Gigabit NICs are compatible with 4 variants of shared PCI buses (32/64bit and 33/66MHz), but obviously a 32bit/33MHz bus incurs a serious bottleneck for Gigabit NICs. If several Gigabit NICs are installed in one computer, it is usually better to use a PCI-X bus that increases the clock rate to 100 or 133MHz (PCI-X version 1) [17], with a maximum throughput of up to 8.5Gbps.

Besides hardware aspects, system software may also reduce network performance. A Gigabit NIC is only part of the system, and a lot of work has to be done by the operating system kernel. If the operating system cannot keep up with the pace set by the NIC, network performance will degrade.

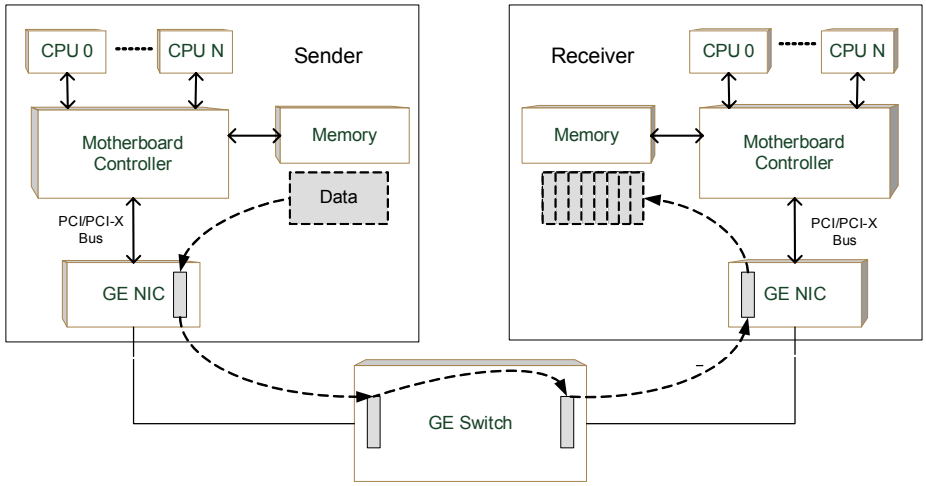

Figure 4-1 Data Flow in Gigabit Ethernet

One problem in Gigabit Ethernets is interrupt flooding. Consider the data flow in a Gigabit Ethernet system, as shown in Figure 4-1. Suppose that both the sender and the receiver have only one Gigabit Ethernet adaptor (NIC) installed, and that there is no routing procedure in the IP layer. On the sender side, the system will copy data from the

user space to the kernel space, segment the data into small packets with protocol headers, and then transfer them to the network interface. The NIC will raise an interrupt for each successful frame departure, notifying the kernel to enable further processing, such as freeing the corresponding memory. On the receiver side, when the NIC receives a frame belonging to its MAC address, it strips the frame's Ethernet header, places the IP packet into a local buffer, and then generates an interrupt telling the kernel of the packet's arrival. The kernel copies the packet into kernel space, removes all protocol headers, reassembles the data with proper ordering, and finally copies the data into user space.

Both incoming and outgoing frames at a NIC will incur hardware interrupts to the system kernel. The rate of interrupts is very large for Gigabit Ethernet communication, e.g. with a 1500-bytes packet and a 600Mbps transfer rate, that results in $600*10^6/(1500*8)$ or 50,000 packets per second. Costly context switches occur when a CPU processes the interrupts, and then executes network routines (system and library calls), which may possibly lead to more context switches. If the processing time for each interrupt is longer than the interval of interrupts (20μsec in our example), the kernel will consume all CPU resources just to service interrupts and the system will cease being able to support productive work in user applications.

The interrupt coalescence technique, also called interrupt mitigation, is used to solve this problem. Nowadays, most Gigabit NICs support this functionality to reduce system load. When interrupt coalescence is enabled, a NIC will wait for a short period of time to generate one interrupt when there is a frame arrival or departure. The idea is based on the assumption that there may be more packets (frames) coming to a NIC, so the NIC treats a collection of sending or receiving operations as a single event (interrupt) to the kernel, thereby reducing the kernel's workload. Interrupt coalescence is usually based on the packets per interrupt rate combined with a threshold time of delay, which can be fixed or dynamic, as shown in Figure 4-2.

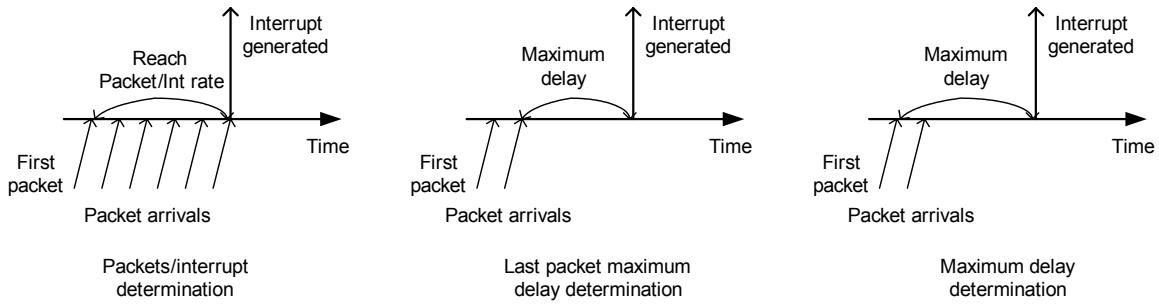

Figure 4-2 Interrupt Coalescence Technique

Interrupt coalescence lowers system CPU usage. It does not introduce many hardships for the sender; however, it can result in extra delay for the receiver, as interrupts are not raised and processed immediately. Consequently, network latency measured by roundtrip time may be inaccurate because interrupt coalescence delays the response time.

Another approach to reduce system load for packet processing is to use jumbo frames. The standard 1526-bytes frame size (1500-bytes MTU, Preamble and Delimiter included) was designed almost three decades ago for low speed Ethernet, with a relatively high error rate from the physical layer connection. Although it still works well in current networks, it is not optimized for high-speed, low-latency, and low-loss networks. Extending Ethernet's frame size to reduce end system work load has become an attractive option. From a study by Alteon Networks [62], jumbo frames could provide 50% more throughput with 50% less CPU usage than standard 1500-bytes frames (Figure 4-3).

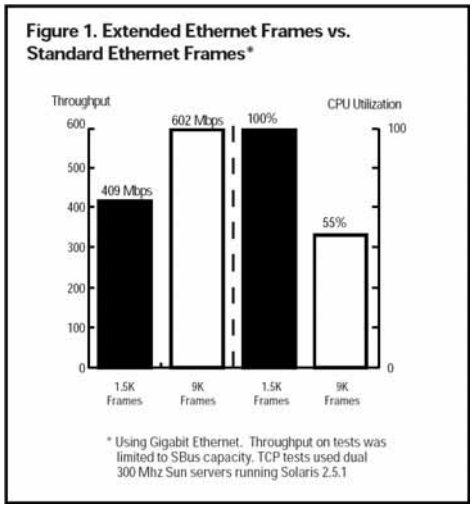

Figure 4-3 A Study of Jumbo Ethernet Frames by Alteon [62]

This result is not surprising. Smaller frames means higher frame rates. As we discussed earlier, higher frame rates would result in more interrupts, and more system processing overhead for a given data transfer size.

The jumbo frames approach to reduce system overhead for high performance networks appears to be better than interrupt coalescence. First, it does not change any network stacks for processing packets. Second, interrupt coalescence consumes more resources than jumbo frames. Although a NIC could buffer several small packets for a single interrupt, the kernel still needs to handle them one by one; on the other hand, for jumbo frames, the kernel copes with fewer packets with larger payload which does not need to be touched. Third, jumbo frames have less protocol overhead. One TCP packet has at least 86-bytes in TCP/IP and Ethernet headers. This results in about a 5.64% protocol overhead for 1526-bytes frames and 0.95% overhead for 9026-bytes frames. Finally, we no longer need to wait for a pending interrupt that is delayed just to be sure no more are arriving. At the same time, though, the packets are larger, so there is a slightly longer delay in transmitting and receiving them.

Today, jumbo frames around 9KB are supported by many high-speed network devices. Careful considerations went into selecting this number. First, research showed that Ethernet's CRC mechanism is able to effectively detect bit errors with frame size only less than 11445-bytes [64]; second, 9KB can support most NFS systems that transfer data in 8KB blocks.

Currently all switches in our SHARCNET test-bed use the standard 1500-bytes MTU. From vendor documentation, we found that they all support 9KB jumbo frames functionality, however. So, it is possible to use jumbo frames to improve system performance in SHARCNET. It would not cause any problem for applications using TCP/IP since it includes an MTU discovery mechanism.

Enabling jumbo frames in a Linux system can be done simply by using the command (with root privileges):

```
/sbin/ifconfig eth0 mtu 9000
```

At the application level, for TCP applications, the user must also set the TCP_MAXSEG option to tell the kernel the desired Maximum Segment Size (MSS), which is the size of the MTU minus the TCP/IP header size (40-byte):

```
int mss = 8960;
setsockopt(socket, IPPROTO_TCP, TCP_MAXSEG, &mss, sizeof(mss));
```

This setting is to override the default MSS value. It is not necessary to do a similar setting for UDP applications; because a large UDP datagram is just filled into a jumbo frame (fragmentation in kernel still occurs if the datagram is too big to put in one frame). Hpcbench is able to test communications using different MTU sizes. Note that the minimum MTU in a link is always used for data communications, no matter how large the MTU is set in application layer.

4.1.3 Data Buffers and Zero-Copy Technique

Data buffering is necessary for network communication. For example, consider a message that is sent from one host to another host. When a system call (e.g. *write()*) is invoked, the data will be first copied into the kernel space from the user space. The kernel fragments the data into small pieces, adds protocol headers to them, and sends these packets to the network card, which adds Ethernet headers to each packet and sends them out over the wire or fibre. The receiver carries out similar, but inverse operations to make the data available to the user's application. Figure 4-4 briefly depicts how the Linux system processes TCP/IP communication in this fashion.

Consider the buffers (queues) involved in Linux TCP/IP communication. First, the user data is copied from process memory into a kernel buffer (socket buffer). The data in the kernel buffer goes through TCP/IP processing and the resulting packets are put into a queue (qdisc). At this point, these packets (in kernel space) are ready to be sent through the network card. The NIC driver implements a ring buffer (*tx_ring*) to transfer the packets into the NIC's local buffer, and then the packets are packed into Ethernet frames and sent to the physical media. On the other end, the NIC buffers the arriving frames, checks their Ethernet header, and discards the frames not belonging to the local host. The desired packets are put into the buffer ring (*rx_ring*). The interrupt handler takes the packets from the buffer ring to a queue (*backlog* queue), and triggers the TCP/IP engine

to process the packets. Finally, the payload is copied from the kernel into an application buffer provided by the application process.

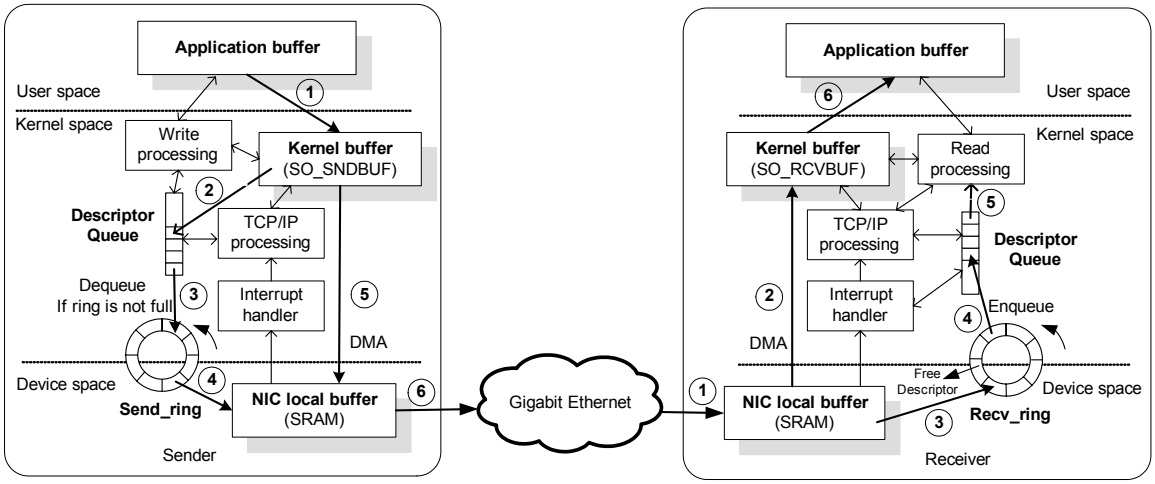

Figure 4-4 Illustration of Linux TCP/IP Implementation

Behind the BSD socket system calls, Linux implements INET sockets with the *sock* data structure holding connection information. Among the many data types used in the Linux kernel, the *sk_buff* (*/linux-src/include/linux/skbuff.h*) data structure plays a key role in networking, as the *mbuf* data type does in traditional BSD socket implementations. Each *sk_buff* holds one packet and several other fields. In the kernel space, all packets are constructed into a linked list of *sk_buff*s. On the sender side, when data are copied into the kernel, a number of *sk_buffs* are created to hold the data, and each *sk_buff* will exist until the corresponding packet is acknowledged by the receiver (TCP). At the receiver, the network driver asks the kernel to allocate an *sk_buff* for each incoming packet, and this *sk_buff* will be freed once the payload is copied into an application buffer (TCP). More specifically, the life cycle of an *sk_buff* starts from the socket INET layer (*sock->prot->sendmsg()* and then *sock_alloc_send_pskb()*) for the sender, and the network device driver for the receiver. All protocol handling in the kernel is based on this crucial data structure. To prevent extra data copying, the kernel only passes the descriptor of the *sk_buff*; the payload is only copied twice during the entire data transfer: between the application and the kernel, and between the kernel and the network device. The latter is usually done by DMA (Direct Memory Access).

Among these buffers, the NIC's local buffer is not configurable. If it is full, the NIC just discards arriving frames. The NIC's ring buffers are simply arrays of *sk_buff* structures. The configuration of ring buffers is highly device-dependent. Some Gigabit Ethernet cards provide drivers where the ring buffer size is settable. Most Gigabit NICs, however, have a fixed ring buffer size.

The kernel descriptor queue is tunable. For example, in the Linux kernel 2.4.x, the send descriptor queue (*tx_queue_len*) is set in */kernel-src/drivers/net/net_init.c*, with a default value of 100. A user with root privilege can set a larger queue:

```
/sbin/ifconfig eth0 txqueuelen 1000
```

This setting can be verified by */sbin/ifconfig –a* command. In a high performance network, when this queue is too small, TCP transmissions might be pending on awaiting the ACKs if the RTT is large. A large send queue, on the other hand, could improve network throughput for high-speed WAN communications.

The receive backlog queue (*netdev_max_backlog*) is a constant value of 300 defined in */kernel-src/net/core/dev.c*. Root users can also adjust this value:

```
/sbin/sysctl -w sys.net.core.netdev_max_backlog=3000
```

Or equivalently:

```
echo "3000" > /proc/sys/net/core/netdev_max_backlog
```

This setting can be verified from the virtual file */proc/sys/net/core/netdev_max_backlog*. If this queue is too small, packets may be dropped in the kernel, resulting in UDP data loss or TCP retransmissions. A large receive queue could improve network throughput in high-speed LAN communications.

The kernel buffer is usually referred to as the socket buffer (IP's *SO_SNDBUF* and *SO_RCVBUF* options). Among all of the buffers discussed, the socket buffer is the only one that an application is able to set. Each computer has a maximum and default socket buffer size for applications. If an application asks for a larger socket buffer than the maximum, Linux will give it the maximum instead. The following commands set the maximum and default send/receive socket buffers with sizes of 8MB and 64KB respectively:

```
/sbin/sysctl -w net.core.wmem_max=8388608
/sbin/sysctl -w net.core.rmem_max=8388608
/sbin/sysctl -w net.core.wmem_default=65536
/sbin/sysctl -w net.core.rmem_default=65536
```

These system settings can be verified in */proc/sys/net/core* directory. In our SHARCNET test-bed, the clusters greatwhite, deeppurple, and hammerhead use the above settings, while the cluster mako has a 128KB maximum socket buffer size and a 64KB default socket buffer size. The actual maximum settable socket buffer size is double these values, as we will discuss below.

It is interesting to note that, in Linux, the actual socket buffer setting is double the value passed to *setsockopt()* system call. For example, with 8MB maximum system setting, we tested the following code:

```
int value=8388608, size=sizeof(value);
setsockopt(socket, SOL_SOCKET, SO_SNDBUF, &value, sizeof(value));
getsockopt(socket, SOL_SOCKET, SO_SNDBUF, &vaule, &size);
printf("Socket send buffer is set to %d \n", value);
```

The result shows that the buffer size is 16777216 (16 MB). This doubling of socket buffer sizes can be traced to the *sock_setsockopt()* kernel function in */linux-src/net/core/sock.c*, line 220-240 in kernel 2.4.20. In the kernel implementation, the socket buffer takes both the payload and protocol overheads into account, making the real buffer size smaller than what the application asked for. Considering this, kernel developers likely intentionally doubled the kernel socket buffer size for network applications, and set the actual usable buffer size for the application to exactly what it asked for, but the message shown by *getsockopt()* system call is twice that, which is the real memory size the kernel allocated. To make things less confusing, Hpcbench shrinks the user's buffer setting in half for Linux environments so that printed messages and log files for tests are consistent with the user's settings.

As we mentioned earlier, all TCP/IP data going through the kernel are in *sk_buff* structures, and all new *sk_buffs* are allocated dynamically in the kernel memory space. Consequently, a socket buffer is not a contiguous memory area when the socket is created. A Linux socket stores the kernel buffer size values (*rcvbuf* and *sndbuf*) in the *sock* structure, and uses two counters (*rmem_alloc* and *wmem_alloc*) to record the total size of created *sk_buffs* for sending or receiving processes. This counter is updated if an *sk-buff*

is free when a packet is sent out into network or is copied into user space. This keeps memory under control, and the total allocated memory for *sk_buffs* will never exceed the maximum socket buffer size.

Buffer size plays a more important role in TCP than UDP because there are retransmissions and flow and congestion controls in TCP communication, where the data will be held in the socket buffer until they are acknowledged by the other end. Theoretically, the optimal socket buffer size for TCP is twice the size of the Bandwidth Delay Product (BDP). Linux systems have a mechanism to adjust TCP window size and dynamically optimize buffer sizes during TCP communication. Three more parameters are also available for Linux TCP socket buffer auto-tuning (*tcp_mem, tcp_rmem and tcp_wmem* in */proc/net/ipv4*). Generally, it is not necessary to modify these values. Linux also has a function to cache TCP connection statistics and hold the threshold (*ssthresh*) of congestion control for every link. The cached values record the congestion status for a certain period of time. This functionality may have a negative effect for network benchmarking. We can flush all TCP statistics cached by the following command, with root privileges:

```
/sbin/sysctl –w sys.net.ipv4.route.flush=1
```

This ensures each connection has its own fresh TCP congestion control when a connection is established.

Data buffering in network communications consumes time and system resources. Data copying from the user space to the kernel space is extremely expensive for high throughput communications. To avoid this overhead, as we mentioned in Chapter 2, some proprietary technologies use zero-copy techniques to improve performance. In fact, this technique can also be deployed in Gigabit Ethernet, as shown in Figure 4-5.

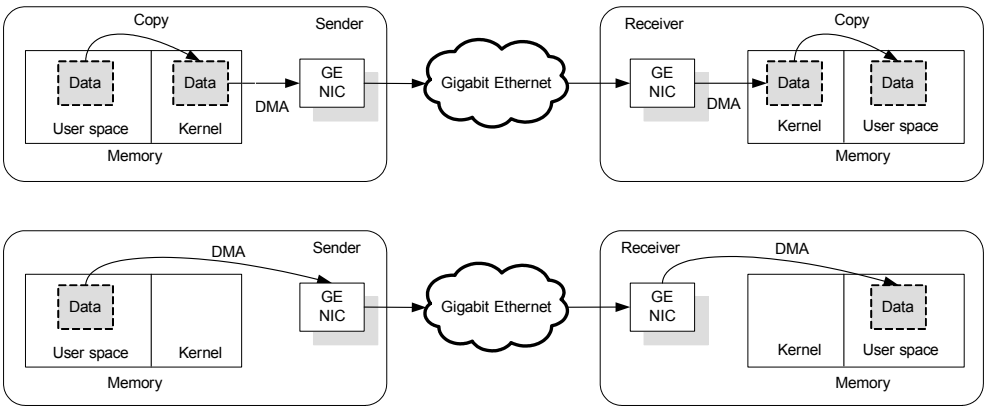

Figure 4-5 Illustration of Zero-copy Technique in Gigabit Ethernet

The key concept behind zero-copy for Gigabit NICs is memory mapping. In the sender, the memory (page) of data is mapped into the kernel space. The kernel goes through the usual TCP/IP procedures and creates protocol headers for the mapped data in its local space, and then triggers the NIC to send the data when the process is done. The NIC fetches the headers from the kernel space and payload from the user space using DMA (Scatter-gather DMA), creates frames, and sends them out. The receiver has similar but inverse operations. The real implementation for the receiver is harder than that of the sender. This is because the payload of each packet is supposed to be directly placed into user memory from the network card through DMA, but packet headers have to be first analyzed by the kernel to determine where the data should go.

Obviously, zero-copy networking needs the support of both the NIC and the operating system. Today, more and more Gigabit Ethernet adapters are able to support zero-copy techniques. Some are even more intelligent, capable of doing extra work, such as computing checksums in hardware and fragmentation. This trend will continue and grow in future 10 Gigabit Ethernet adapters, where the NICs play an even more active role in off-loading computations from the CPU and speeding up data transmissions.

Linux systems with kernel 2.4.x and later versions have a *sendfile()* system call that supports a zero-copy data transfer. Hpcbench is able to test and use this function. If the sender's NIC is zero-copy aware, we would expect a lower system load with higher throughput in such a case.

4.2 Network Communication and Kernel Interactions

In this section, we examine the interaction between the kernel and different communication protocols. We look at UDP, TCP and MPI communication on two different HPC systems: one based on 4-processor Alphas (*hammherhead*) and the other a system based on 4-processor Intel Xeons (*mako*). All tests were conducted on idle machines and repeated 10 times; the medians of results are presented for our discussion. We select medians instead of means because some attributes of interest in our study, such as CPU usage and workload distribution, are very different from test to test.

4.2.1 Communication on an Alpha SMP Architecture

Using Hpcbench and the system statistics that it reports, we examined UDP, TCP and MPI communication on *hammerhead,* which includes 28 HP Alpha ES40 (4x833MHz CPU) running the Linux 2.4.21 SMP operating system with an Alteon AceNIC (64bits/33MHz PCI) Gigabit Ethernet Card installed. The Gigabit Ethernet switch in *hammerhead* is Nortel's Passport 8600 series (refer to Figure 2-15 and Table 2-1).

4.2.1.1 UDP Communication

First, we tested UDP unidirectional communication between two nodes on *hammerhead*. Hpcbench was used to measure UDP throughput with its default settings: 5 seconds of test time, repeat 10 times, 1460-bytes datagram size, and 64KB system default send/recv socket buffer (we will examine other settings later):

```
[hh14 ~/hpcbench/udp]$ udptest –ch hh15 –o udp-result.txt
```

UDP stream test (5 Sec.)	Client (hh14)	Server (hh15)
Throughput (Mbps)	506.31	506.02
CPU Utilization	12%	18%
CPU load distribution (CPU 0-3)	0%, 14%, 34%, 0%	4%, 32%, 0%, 35%
Total Interrupt to kernel	41340	70048
Process time in user mode (sec)	0.06	0.02
Process time in system mode (Sec)	2.29	2.71
Process sent-datagram	216787	0
Process received-datagram	0	216787
Process sent-byte	316507592	0
Process received-byte	0	316507592

NIC sent-frame	217788	1
NIC sent-byte	325686720	74
NIC received-frame	9	217791
NIC received-byte	932	325686886
Interrupt generated by NIC	12596	41352

Table 4-1 UDP Unidirectional Communication Statistics

The results show that there was no UDP data loss at the application layer. In both ends, the number of sent/received datagrams reported by Hpcbench is a bit different from the statistics of the NIC that was traced from the kernel */proc* files. This is because the information read from */proc* files could not be accurately synchronized by the process, as we mentioned at the end of Section 3.5 in Chapter 3.

The average frame size was around 325686720/217788 (sender) ≈ 325686886/217791 (receiver) ≈ 1495-bytes in both sender and receiver, close to the expected 1502 bytes (a 1460-bytes payload + 20-bytes IP header + 8-bytes UDP header + 14-bytes Ethernet header = 1502 bytes).

UDP communication generated a 12% CPU load for the sender (client) and an 18% CPU load for the receiver (server). The workload was distributed to different CPUs in both sides. We noticed that the CPU distribution varied significantly for different tests, showing that the kernel randomly assigns the interrupts into different CPUs. The CPU cycles were mainly devoted to system mode in both sides. This can be verified by the process tracing: in a 5-second test, more than 50% process time was spent in system mode in both sender and receiver, and less than 2% process time was spent in user mode in both sides; the remaining time of was waiting for kernel processing.

The test results also show that the interrupt coalescence technique was used in UDP communications. In the sender, the NIC generated one interrupt to the kernel when sending out about 17 frames (217788/12596), while the receiver's NIC generated one interrupt for about 5 incoming packets (217791/41352).

The analysis of UDP communication with larger socket buffer will be discussed in Section 4.2.1.4.

4.2.1.2 TCP Communication

We then tested TCP communication using Hpcbench with its default settings for TCP (5-second test time, 10 repetitions, 64KB message size, default system socket buffers of 64KB for sending and 87380-bytes for receiving):

```
[hh14 ~/hpcbench/tcp]$ tcptest –ch hh15 –o tcp-result.txt
```

TCP stream test (5 Sec.)	Client (hh14)	Server (hh15)
Throughput (Mbps)	523.97	
CPU Utilization	21%	30%
CPU load distribution (CPU 0-3)	6%, 61%, 0%, 17%	0%, 64%, 53%, 2%
Total Interrupt to kernel	67595	70326
Process time in user mode (sec)	0.01	0.02
Process time in system mode (Sec)	1.19	3.34
Message size in bytes	65536	
Iteration of transferring message	4995	
Data size of each sending/receiving	8192	
Process total sent/recv-byte	327352320	
NIC sent-frame	226076	47820
NIC sent-byte	342273337	3156122
NIC received-frame	47825	226079
NIC received-byte	3156514	342273570
Interrupt generated by NIC	47086	49835

Table 4-2 TCP Unidirectional Communication Statistics in Alpha SMP Systems

(Client and server share some fields in the table as TCP is a reliable protocol)

From the results we found that CPU1 had the most workload in both the client and the server, and other CPUs were sharing the workload without an obvious pattern. As well, the server's system load was obviously greater than that of the client.

There was considerable return traffic from the server to the client in unidirectional tests. This was due to acknowledgements (ACKs). The server sent one acknowledgement to the client for about 4~5 incoming packets (226079/49835). This may imply the use of SACK. The average ACK frame size was around 66-bytes (3156122/47820). Considering a 20-bytes IP header, 20-bytes TCP header, 12-bytes timestamp (default setting in Linux) and 14-bytes Ethernet header, the ACKs were exactly blank TCP packets without payload. The average frame size (outgoing for the client and incoming for the server) was about 342273337/226076 ≈ 342273570/226079 ≈ 1514, equal to the maximum frame size

defined in the IEEE 802.3 Ethernet protocol. The overhead of TCP communication in this test is about (226076 * (20+20+12+14) + 3156514) / 327352320 ≈ 5.6%.

In this case we cannot distinguish whether the interrupts were generated by incoming or outgoing packets (frames). Putting them together, there were about 5-6 packets per interrupt in both the sender and the receiver.

Then we started a TCP communication session with a 1 MB socket buffer size in both ends:

```
[hh14 ~/hpcbench/tcp]$ tcptest –ch hh15 –o tcp-result.txt –b 1M
```

TCP stream test (5 Sec.)	Client (hh14)	Server (hh15)
Throughput (Mbps)	573.75	
CPU Utilization	23%	34%
CPU load distribution (CPU 0-3)	24%, 65%, 0%, 3%	5%, 65%, 0%, 66%
Total Interrupt to kernel	67060	68171
Process time in user mode (sec)	0.01	0.03
Process time in system mode (Sec)	1.35	3.54
Message size in bytes	65536	
Iteration of transferring message	5305	
Data size of each sending/receiving	8192	
Process total sent/recv-byte	347668480	
NIC sent-frame	240106	52015
NIC sent-byte	363515477	3432992
NIC received-frame	52023	240111
NIC received-byte	3433564	363515788
Interrupt generated by NIC	47165	48296

Table 4-3 TCP Communication with 1 MB Socket Buffer Size

From these results, we can see that with a bigger buffer size, the throughput and the system load increase a little. From similar calculations, the overall characteristics, such as the average frame size, average ACK packet size, SACK rate, and interrupt coalescence rate remained the same as those of tests with default socket buffer sizes.

4.2.1.3 MPI Communication

The default MPI implementation (MPICH 1.2.5.2) in all three Alpha clusters was linked to Quadrics' QsNet interconnect. To test MPI communication (MPICH) over Gigabit Ethernet, we built our own MPICH 1.2.5.2 packages based on the TCP/IP stack, defined

a p4 group file named *conf* that specifies two processes in two nodes (hh14 for the master process and hh15 for the slave process) for the test, and then passed the *–p4pg* argument to the *mpirun* script:

```
[hh14 ~/hpcbench/mpi]$mpirun –p4pg conf mpitest –c –o mpi.txt
```

MPI stream test (4.79 Sec.)	Master process (hh14)	Slave process (hh15)
Throughput (Mbps)	314.46	
CPU Utilization	19%	27%
CPU load distribution (CPU 0-3)	0%, 41%, 0%, 35%	11%, 53%, 14%, 29%
Total Interrupt to kernel	64511	66540
Process time in user mode (sec)	0.04	0.19
Process time in system mode (Sec)	1.79	2.80
Data size of each sending/receiving	65536	
Process total sent/recv-byte	188284928	
NIC sent-frame	134885	58886
NIC sent-byte	203794436	3913620
NIC received-frame	59002	134865
NIC received-byte	3915176	203792868
Interrupt generated by NIC	44018	45436

Table 4-4 MPI Point-to-Point Communication Statistics in Alpha SMP Systems

Except for the lower throughput, the characteristics of MPI point-to-point communication are similar to TCP communication. This was because our build MPICH over Gigabit Ethernet was based on underlying TCP/IP communication. The lower throughput of MPI communication was possibly due to extra communication control in our MPI implementation, where some of this control may overlap with low-layer TCP transmission controls. It may also have been affected by interrupt coalescence.

The average frame size of MPI communication from the master node to the slave (secondary) node is 203794436/134885 ≈ 203792868/134865 ≈ 1511-bytes. This is slightly less than that of the earlier TCP test. This may imply that there were some small packets for MPI communication. The average frame size from the slave node to the master node is 3915176/59002 ≈ 3913620/58886 ≈ 66-bytes, equal to the size of blank TCP packets. We assume that they were also TCP acknowledgements.

4.2.1.4 Performance Factors of Network Communication

There are many devices and software elements involved in network communications. For a Gigabit Ethernet environment, the main components participating in networking include the machine's operating system, memory, CPU, network card (NIC), and network switch. The bottleneck in the all of these components ultimately determines the overall throughput.

The memory used in modern computers is usually not the bottleneck. Even PC133 SDRAM that has been widely used in the PC market for years can support up to 8000Mbps throughput (64bit). The switch is also not likely a bottleneck for a network system in general, since data handling in a switch is much easier than in end hosts. Even in intelligent (layer 3-7) switches, all protocol processing is done in hardware instead of software, resulting in a very high packet per second rate.

From the experiments in the previous sections we found that the network communication throughput in Alpha systems measured by Hpcbench with default parameters was relatively low (less than 550 Mbps in UDP/TCP/MPI communications). What factors lead to such a low throughput in the Alpha systems? We need to do further experiments to discover the potential bottleneck.

We tested UDP communication with a larger socket buffer (1MB) and larger datagram size (4KB):

```
[hh14 ~/hpcbench/udp]$ udptest -ch hh15 -b 1m -o udp-result1.txt
[hh14 ~/hpcbench/udp]$ udptest -ch hh15 -b 1m -l 4k -o udp-result2.txt
```

UDP stream test (5 Sec.)	Client (hh14)		Server (hh15)	
	Datagram 1460 Byte	Datagram 4KB	Datagram 1460 Byte	Datagram 4KB
Throughput (Mbps)	1432.12	1599.17	650.26	588.40
CPU Utilization	18%	18%	21%	26%
CPU load distribution (CPU 0-3)	0%, 71%, 0%, 0%	0%, 72%, 0%, 0%	22%, 45%, 18%, 0%	37%, 57%, 5%, 4%
Total Interrupt to kernel	41400	41403	70330	70186
Process time in user mode	0.14 Sec.	0.07 Sec.	0.03 Sec.	0.01 Sec.
Process time in sys mode	4.86 Sec.	4.93 Sec.	3.24 Sec.	3.19 Sec.
Process sent-datagram	613181	244061	0	0
Process received-datagram	0	0	279548	89870
Process sent-byte	895242832	999669792	0	0

Process received-byte	0	0	408138652	368103456
NIC sent-frame	279866	294113	6	7
NIC sent-byte	419906259	411703255	682	1008
NIC received-frame	8	11	279861	294110
NIC received-byte	512	1108	419904790	411701565
Interrupt generated by NIC	12680	12682	41635	41474

Table 4-5 UDP Unidirectional Communication with 1MB Socket Buffer

In contrast to Table 4-1, the UDP communication statistics recorded vary more widely than that observed with a default 64KB socket buffer. Network throughput had significantly increased 30% when the socket buffer size increased from 64KB to 1MB. When a socket buffer is small (e.g. 64KB), it will fill more quickly, and then block the application (benchmark) from sending more data (pending on *write(), sendto()* or other system calls). Thus, the communication bottleneck is possibly due to the limitations of small kernel socket buffers. When the socket buffer is large enough, this barrier disappears, resulting in higher throughput.

When the socket buffer size increased to 1MB, in addition to the increase in throughput, we observed considerable data loss during the test. We further observed that the UDP data loss was due to the local system, and not because of the network. There was about a 54% datagram lost (1MB socket buffer, 1460-bytes datagram), before the data was sent out of the network interface card. The results table shows no evidence that this data loss was caused by the network or the switch (the data sent from the client's NIC were equal or close to the data received from the server's NIC); instead, statistics show that the kernel or network card dropped the UDP datagram.

With a large socket buffer, the UDP protocol processing capability in the kernel may become a bottleneck instead. If the kernel cannot handle all of the data arriving from the socket buffer, it may have no choice but to drop the data. Another possible bottleneck is the network interface card. When it is not able to cope with all of the packets arriving from the kernel, it may also have to discard some of the packets. Since the kernel is working closely with the network card's driver, we simply consider that kernel operations were responsible for the data loss.

When the datagram size was increased to 4KB (with the same socket buffer size of 1MB), we see that the local throughput (recorded by the UPD sender application that returns immediately regardless if the data was sent or lost) increased too, but the network throughput decreased, resulting in a very high data loss rate of 63% ((999669792-368103456)/999669792). The reason for a higher local throughput with a larger datagram size is due to a reduction in the number of system calls required, such as *write()* or *sendto()*, for sending the same amount of data (from the application to the kernel socket buffer). In our 5-second test examples, the application (Hpcbench) had sent 895242832 Bytes of data to the kernel in the case of 1460-bytes UDP datagrams, and this number had increased to 999669792 for a 4KB UDP datagram size.

To determine why a larger datagram size leads to lower network throughput in our tests, we first look at how a UDP datagram traverses the network. Each UDP datagram is treated as a single unit during transmission in the network. When the DF flag (Do not Fragment) is not set, the datagram will be fragmented into smaller packets to traverse the network if its size is greater then the MTU. If the flag is set and the size exceeds the MTU for the network, network devices will discard the datagram and return an ICMP error message of "Fragmentation needed but DF set". In the receiver, all fragmented packets are reassembled back to a single datagram and then copied into the application. The datagram is discarded by the kernel if it is not complete, even if only a single packet belonging to it was lost. The theoretical maximum datagram size is 64KB in IPV4. In most network environments, including *hammerhead*, the MTU size was 1500-bytes, implying a maximum 1472-bytes UDP payload size for each packet (1500 - 20-bytes IP header - 8-bytes UDP header). So, a 4KB UDP datagram would be fragmented into 3 packets to fit the 1500-bytes MTU size and then sent out over the network. In this case, there would be some packets transmitted that did not completely fill an entire MTU, because 4KB is not a multiple of the number 1472. This can be verified by the average frame size of the sender: in the 1460-bytes socket buffer case, the average frame size is 419906259/279866 ≈ 1500 Byte, close to 1502 (1460 + 20-bytes IP header + 8-bytes UDP header + 14-bytes Ethernet header); in the 4KB socket buffer case, the average frame size is 411703255/294113 ≈ 1401 Bytes, close to 1407 (4096/3 + 42-bytes header).

When some small packets are transmitted, this can lead to smaller network throughput when the other conditions remain the same. In a 5-second test, there was a total of 419904790 Bytes of data that reached the server for 1460-bytes datagrams, and this number decreased slightly to 411701565 for 4KB datagrams.

Moreover, if some packets belonging to a large datagram are lost, all packets of this datagram will be dropped. In our 4KB datagram example, each datagram will be sent out in three separate packets, and all 4KB of data will be dropped in the server if any of these three packets is lost. This situation was observed to have occurred during our tests. Packet loss during the client's kernel processing caused other packets to be unable to be reassembled back to a complete datagram, even though they had successfully reached the server. This is clearly shown in the network statistics: in the case of 1460-bytes datagrams, the server received 419904790-bytes of data from the network interface, and the process (application) finally retrieved back 408138652-bytes of data. This ratio is about 97.2%, equal to 1460 (payload) / 1502 (payload+headers); while in the 4KB datagram case, this ratio decreased to 89.4% (368103456/411701565), implying that some data was discarded at the server. This loss can be computed by the packets and datagrams the server received: the server received a total number of 89870 4KB datagrams, corresponding to 269610 (89870x3) packets with a 1500-bytes MTU. Consequently, as many as 24500 (294110-269610) packets were discarded by the server's kernel due to incomplete datagrams. This is another reason that explains how larger datagrams may lead to lower network throughput when there is data loss during the communication.

In both cases (with both datagram sizes), the main data loss is due to kernel processing. From recorded process time information, we can also arrive at this conclusion. When the socket buffer is large, more than 95 percent of process time was used in system mode and the remaining was in user mode. The system was overwhelmed by UDP transmission processing. The speed limitations of the system caused considerable data loss during this process.

We focus on the UDP communication in our analysis because UDP communication is connectionless, has less overhead, and is closer to raw IP packets than TCP and other

connection oriented protocols. There is no extra interferential control between the two end systems in our test, so UDP communications can more closely reflect the characteristics and behavior of the network. For throughput tests, TCP and MPI communication never exceeded 600 Mbps, no matter how we configured test parameters.

Three issues are important to achieving maximum throughput for UDP communication: (1) minimizing application (benchmark) overhead, (2) maximizing the kernel resource for network processing; (3) sending full packets to fit the path MTU (payload = MTU – protocol headers). For instance, fewer system calls have less application overhead (condition 1); a larger socket buffer allows the kernel to spend more resources on networking (condition 2); data transmission with a packet size slightly less than the MTU can reduce the protocol overhead and improve the communication efficiency (condition 3). Out tests with a 1460-bytes payload and 1MB socket buffer size closely matched these conditions, and thus the measured throughput of about 650Mbps is close to the maximum achievable throughput of the *hammerhead* systems.

From our analysis, we conclude that the maximum achievable throughput of 650 Mbps is limited by the UDP sender's processing capabilities. The network switch may support higher throughputs because there was no obvious data loss caused by the switch in all of the tests according to the statistics reported by Hpcbench. To verify this, we can direct two senders' traffic into one receiver, to check whether the overall network throughput measured by the server is significantly greater than 650Mbps, as shown in the below picture.

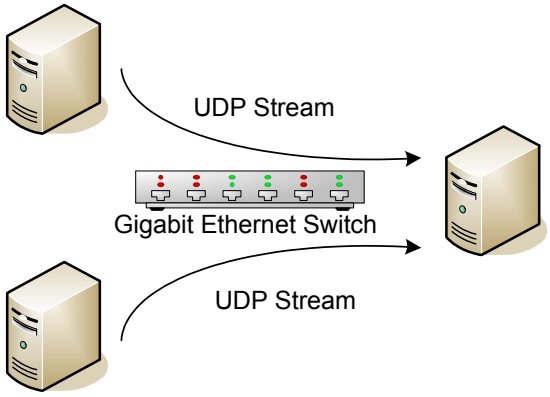

Figure 4-6 Two Simultaneous UDP Stream into One End Station

We conducted this test using two machines (clients) to send UDP data to one machine (server) simultaneously in *hammerhead*:

```
[hh14 ~/hpcbench/udp]$ udptest -ch hh15 -b 1m -t 10 -o udp-1.txt
[hh16 ~/hpcbench/udp]$ udptest -ch hh15 -b 1m -t 10 -o udp-2.txt
```

UDP stream test (10 Sec.)	Clients (hh14, hh16)		Server (hh15)	
	hh14	hh16	Connected to hh14	Connected to hh16
Throughput (Mbps)	1433.40	1409.27	448.63	446.38
CPU Utilization	21%	21%	41%	41%
CPU load distribution (CPU 0-3)	0%, 83%, 0%, 0%	0%, 83%, 0%, 0%	34%, 74%, 18%, 38%	34%, 74%, 18%, 38%
Total Interrupt to kernel	74535	74510	139251	139441
Process time in user mode	0.28 Sec.	0.28 Sec.	0.05 Sec.	0.04 Sec.
Process time in sys mode	9.72 Sec.	9.72 Sec.	5.83 Sec.	5.83 Sec.
Process sent-datagram	1229978	1206566	0	0
Process received-datagram	0	0	384328	382991
Process sent-byte	1795766452	1761584932	0	0
Process received-byte	0	0	561117452	559165432
NIC sent-frame	557858	556923	5	5
NIC sent-byte	837457288	836052794	726	731
NIC received-frame	13	17	767166	768186
NIC received-byte	916	1168	1151479279	1153009899
Interrupt generated by NIC	25320	25308	90046	90178

Table 4-6 Two UDP Stream Test Simultaneously on an Alpha Cluster

Two client processes were started at the same time by hand, which consequently may not be accurately synchronized. To reduce this effect, we carried out the tests for longer periods of time (10 seconds).

The overall throughput directed to the server was about (837457288+836052794)*8/10 ≈ 1339 Mbps. The network link to the server was saturated and packets were dropped by Gigabit Ethernet switch. The overall network throughput measured by server was about 895 Mbps (448.63+446.38). This proves our assumption that the bottleneck in the Alpha system is in the sender machine, not the server or the switch.

All of the above experiments were executed in purely idle machines. When the systems are under a heavy load, network performance should degrade accordingly. We selected four busy nodes in *hammerhead* to test their unidirectional throughput: two were working in user mode, and the other two were working in user space with a lower priority

(maximum nice value of 19 in two nodes). Tests were repeated ten times and the average results are shown in the following table.

Test node hh10 and hh25 had 100% CPU-load in user mode (pre-test and post-test). Test node hh15 and hh20 had 100% CPU-load in user and nice mode (pre-test and post-test). All results were tested by Hpcbench with default settings within 10 minutes.				
Protocol	hh10→hh25	hh25→hh10	hh15→hh20	hh20→hh15
UDP	5.017 Mbps	5.334 Mbps	92.498 Mbps	92.872 Mbps
TCP	6.290 Mbps	7.069 Mbps	63.330 Mbps	64.694 Mbps
MPI	3.101 Mbps	3.217 Mbps	38.105 Mbps	39.273 Mbps

Table 4-7 Network Protocol Communications with Busy Machines

As we can see, network throughputs in Gigabit Ethernet dropped to less than 10 Mbps when the system had heavy applications running. Even though the applications had been set at a lower priority, the achievable throughputs were still much lower than normal. We tested two relatively busy machines with 50% to 80% CPU loads, where there usually existed at least one CPU with low usage in the 4-processor Alpha servers, and a throughput of about 500Mbps was achieved for both UDP and TCP communication. However, throughput decreased dramatically to less than 300Mbps when CPU load was greater than 90%, where all CPUs had a heavy load. This shows that network performance also depends on system's workload, and high network throughput is only available with sufficient system resources.

In HPC environments, computational resources tend to be controlled by some kind of a job management system, which utilizes available CPU power as much as possible to service submitted jobs, and usually ignores network communication requirements. We often found in other testing that many compute nodes have 100% CPU usage in our test-bed; most are doing computational jobs (Refer to Appendix B). In such cases, network communication over Gigabit Ethernet will likely perform extremely poorly.

4.2.2 Communication on an Intel Xeon SMP Architecture

The *mako* cluster consists of 8 nodes of Intel Xeon 4-processor SMP systems with the Linux 2.4.20-8smp operating system. The network components in *mako* include Broadcom NetXtreme Gigabit Ethernet and an HP ProCurve 2800 switch, as shown in Figure 2-15. In this section, we examine end-to-end protocol communications over Gigabit Ethernet in this cluster in a similar manner as our testing with the Alpha SMP system in the previous section.

4.2.2.1 UDP Communication

We selected two idle nodes *mk1* and *mk2* for UDP communication tests using Hpcbench with its default parameters:

```
[mk1 ~/hpcbench/udp]$ udptest –ch mk2 -o udp.txt
```

UDP stream test (5 Sec.)	Client (mk1)	Server (mk2)
Throughput (Mbps)	956.98	956.76
CPU Utilization	13%	16%
CPU load distribution (CPU 0-3)	15%, 0%, 38%, 0%	38%, 25%, 0%, 0%
Total Interrupt to kernel	83017	410289
Process time in user mode (sec)	0.16	0.12
Process time in system mode (Sec)	2.50	1.65
Process sent-datagram	409670	0
Process received-datagram	0	409659
Process sent-byte	598116772	0
Process received-byte	0	598100712
NIC sent-frame	410672	4
NIC sent-byte	617039761	616
NIC received-frame	4	410661
NIC received-byte	616	617023195
Interrupt generated by NIC	82153	409486

Table 4-8 UDP Unidirectional Communication Statistics in Intel Xeon SMP Systems

Interestingly, the UDP throughput measured using Hpcbench with default settings could achieve 956Mbps in *mako*. We found an 11-datagram loss out of about 400,000-datagrams sent during the test at the application layer. The average frame size was about 617039761/410672 ≈ 617023195/410661 ≈ 1502-bytes in both the client and the server, matching the number of 1460 (payload) + 8 (UDP header) + 20 (IP header) + 14 (Ethernet header) = 1502. Considering UDP/IP and Ethernet overhead (including the preamble and the CRC trailer), the real throughput over the network was about

(1460+20+8+26)*956/1460 ≈ 992 Mbps, nearly the theoretical maximum throughput of a Gigabit Ethernet.

From statistics available from the NIC, we observe there were about 5-packets per interrupt in the sender, but no interrupt coalescence in the receiver. A lack of interrupt coalescence in the receiver implied that there was confidence in the system being able to handle a large number of interrupts. In our experiments, there were more than 80,000-interrupt/sec in the receiver during the test, and the CPU utilization was only about 16%, distributed to CPU0 and CPU1, while CPU2 and CPU3 were completely idle. CPU performance benefited from Intel's high CPU clock rate and other architectural considerations. All machines in *mako* had 4x3GHz CPUs. On the contrary, the nodes in *hammerhead*, *deeppurple*, and *greatwhile* had CPUs with a clock rate of less than 1GHz.

4.2.2.2 TCP Communication

We also started TCP communication tests using Hpcbench with its default settings:

```
[mk1 ~/hpcbench/tcp]$ tcpptest –ch mk2 -o tcp.txt
```

TCP stream test (5 Sec.)	Client (mk1)	Server (mk2)
Throughput (Mbps)	941.052	
CPU Utilization	22%	22%
CPU load distribution (CPU 0-3)	64%, 0%, 24%, 0%	87%, 0%, 0%, 0%
Total Interrupt to kernel	201863	406477
Context switches	30561	437966
Process time in user mode (sec)	0.01	0.07
Process time in system mode (Sec)	1.23	2.76
Message size in bytes	65536	
Iteration of transferring message	8971	
Data size of each sending/receiving	8192	
Process total sent/recv-byte	587923456	
NIC sent-frame	391814	195674
NIC sent-byte	594606442	13696481
NIC received-frame	195659	391834
NIC received-byte	13696411	594652848
Interrupt generated by NIC	201256	405873

Table 4-9 TCP Unidirectional Communication Statistics in Intel Xeon SMP Systems

The table shows that the achievable TCP throughput was about 1.7 percent less than that of UDP. The sender CPU load increased to nearly double that observed during UDP tests.

This may be because the TCP sender had a control window and had to work with ACK data. From the table, we can see that the number of ACK packets was about half of the test packets with frame size of 1369411/195659 ≈ 13696481/195674 ≈ 70-byte, 4 more bytes than 40 (TCP/IP headers) + 12(timestamp) + 14(Ethernet header) = 66-byte. The data frame size is around 594606442/391814 ≈ 594652848/391834 ≈ 1518-byte, 4 more bytes than 1500 (MTU) + 14 (Ethernet header) = 1514-byte. It is possible that the NIC driver on *mako* machines takes the 4-bytes CRC for Ethernet frames into account.

4.2.2.3 MPI Communication

The default MPI implementations (MPICH and LAM/MPI) in the Intel Xeon cluster (mako) were all linked to Myrinet. Once again, to test MPI communication (MPICH) over Gigabit Ethernet, we built our own MPICH 1.2.5.2 (the same as that of Alpha systems) packages based on TCP/IP stack, and defined a p4 group file named *conf* that specifies two processes in two nodes (mk1 for the master process and mk2 for the slave process) for the test. We then tested MPI communication (MPICH) with Hpcbench's default settings:

```
[mk1 ~/hpcbench/mpi]$ mpirun –p4pg conf mpitest –c –o mpi.txt
```

MPI stream test (4.97 Sec.)	Master process (mk1)	Slave process (mk2)
Throughput (Mbps)	932.055	
CPU Utilization	19%	24%
CPU load distribution (CPU 0-3)	76%, 0%, 0%, 0%	97%, 0%, 0%, 0%
Total Interrupt to kernel	190124	399308
Context Switches	73351	120122
Process time in user mode (sec)	0.10	0.39
Process time in system mode (Sec)	2.81	3.96
Data size of each sending/receiving	65536	
Iteration of transferring message	8840	
Process total sent/recv-byte	579338240	
NIC sent-frame	396894	185949
NIC sent-byte	592235616	13039878
NIC received-frame	185912	396974
NIC received-byte	13038020	592333662
Interrupt generated by NIC	189517	44828

Table 4-10 MPI Point-to-Point Communication Statistics in Intel Xeon SMP Systems

We can see there are many results similar to those observed in TCP communication. The throughput was slightly less than TCP's, while more system and user time was spent. As with TCP, the return traffic in MPI communication (*mk2* to *mk1*) had an average frame size of 70-bytes, and the average frame size from the client (mk1) to the server (mk2) was about 1518-bytes; both of them are equal to TCP's. This is natural since our MPICH implementation is based on underlying TCP communication. One difference between MPI communication and TCP communication is that only a single CPU, CPU0, participated in MPI communication.

4.2.2.4 Performance Factors of Network Communication

To examine the impact of different parameters, as on the Alpha systems, we tested UDP communication with a larger socket buffer (256KB, the maximum in *mako*) and a larger datagram size (4KB) in the Intel Xeon cluster:

```
[mk1 ~/hpcbench/udp]$ udptest -ch mk2 -b 256k -o udp-result1.txt
[mk1 ~/hpcbench/udp]$ udptest -ch mk2 -b 256k -l 4k -o udp-result2.txt
```

UDP stream test (5 Sec.)	Client (mk1)		Server (mk2)	
	Datagram 1460 Byte	Datagram 4KB	Datagram 1460 Byte	Datagram 4KB
Throughput (Mbps)	956.95	957.65	956.62	957.02
CPU Utilization	12%	18%	20%	16%
CPU load distribution (CPU 0-3)	12%, 0%, 36%, 0%	0%, 72%, 0%, 0%	50%, 0%, 30%, 0%	61%, 0%, 0%, 2%
Total Interrupt to kernel	83092	88735	409997	438716
Process time in user mode	0.13 Sec.	0.10 Sec.	0.08 Sec.	0.04 Sec.
Process time in sys mode	2.38 Sec.	2.89 Sec.	2.04 Sec.	1.65 Sec.
Process sent-datagram	409656	146127	0	0
Process received-datagram	0	0	409645	146127
Process sent-byte	598096332	598532128	0	0
Process received-byte	0	0	598080272	598532128
NIC sent-frame	410658	439389	4	4
NIC sent-byte	617018677	616442599	613	608
NIC received-frame	4	7	410647	439387
NIC received-byte	613	811	617002110	616442125
Interrupt generated by NIC	82124	87759	409872	437882

Table 4-11 UDP Unidirectional Communication with 1MB Socket Buffer

Results show that network throughput does not change dramatically, which is as we expected, since the measured throughput with the default socket buffer size had already been very close to the theoretical bandwidth of Gigabit Ethernet. However, the local throughput almost remained unchanged in all cases with different socket buffers and datagram sizes. This is totally different from how the Alpha system behaved, where the local throughput could be much higher than the network throughput and a large number of UDP datagrams were dropped in the kernel for such cases. This difference may come from the network drivers of the two network interface cards, since the Linux kernel of these two systems (Linux 2.4.21 SMP in Alpha and Linux 2.4.20 SMP in Intel) was not significantly different. Another reason may be due to the New Application Program Interface (NAPI).

Since kernel 2.4.20, Linux implemented a NAPI for network devices. NAPI deploys a new interrupt mitigation mechanism that combines hardware and software interrupts, and polling techniques. Some experiments and research show that the NAPI was able to efficiently improve network performance and lower system load for network subsystems. NAPI is compatible with legacy NIC drivers, but the new functionality is disabled in this case. All drivers for network interface cards have to be rewritten to support the NAPI functionality. We noticed that the Broadcom Tigon3 Gigabit Ethernet cards were used in the *mako* Intel system, whose driver (*/Linux-kernel/driver/net/tg3.c*) supported the NAPI, while NICs in the Alpha systems did not support NAPI. It is possible that the new features of NAPI result in the excellent network performance in the Intel Xeon system, and lead to different interactions between the kernel and network interface cards. Or both faster CPU and NAPI in the Intel system contribute to the better performance.

The IEEE 802.3x standard specifies the transmission flow control between the sender machine and network devices, such as switches, to prevent the sender from flooding the network. This frame based flow control is independent of TCP's flow controls, and is only effective in layer 2. The sender's network interface card and its driver should follow this standard, and the sender will never generate data traffic greater than the bandwidth of the (Gigabit) Ethernet. Thus the 1Gbps data rate is the maximum throughput for a sender machine in a Gigabit Ethernet environment. Inside the sender, different drivers of network cards can have a different impact on the Linux kernel. In Alpha systems, the

NIC did not block the sender from sending, even when the data transfer rate was greater than the throughput that the NIC could support (some data was discarded by the NIC to resolve this issue). Consequently, the kernel and the application may not be aware of the underlying NIC's maximum throughput, which is determined by the NIC itself and Ethernet protocol limits as well, so the local throughput reported by application was much higher than the 1 Gbps. On the other hand, in the Intel Xeon system, the Tigon3's driver seemed to block the kernel from sending when the data rate exceeded Gigabit Ethernet's bandwidth. In this case, no matter how fast the system is, the local throughput measured by the application is constrained by the NIC's transmission limit, which will never pass the bandwidth of Gigabit Ethernet.

We also tested the case of two UDP sessions to a single end system in the Intel Xeon system:

```
[mk1 ~/hpcbench/udp]$ udptest -ch mk2 -o udp-1.txt -t 10
[mk3 ~/hpcbench/udp]$ udptest -ch mk2 -o udp-2.txt -t 10
```

UDP stream test (10 Sec.)	Clients (mk1, mk3)		Server (mk2)	
	mk1	mk3	Connected to mk1	Connected to mk2
Throughput (Mbps)	956.01.	956.83	498.11	488.15
CPU Utilization	15%	21%	22%	22%
CPU load distribution (CPU 0-3)	28%, 0%, 0%, 30%	0%, 83%, 0%, 0%	63%, 24%, 0%, 0%	63%, 25%, 0%, 0%
Total Interrupt to kernel	165629	165775	844080	844774
Process time in user mode	0.27 Sec.	0.25 Sec.	0.14 Sec.	0.12 Sec.
Process time in sys mode	4.37 Sec.	4.35 Sec.	2.01 Sec.	1.96 Sec.
Process sent-datagram	818496	819207	0	0
Process received-datagram	0	0	426516	418020
Process sent-byte	1195002732	1196040792	0	0
Process received-byte	0	0	622711932	610307772
NIC sent-frame	819505	820210	13	15
NIC sent-byte	1232732395	1233802649	1742	2147
NIC received-frame	11	6	857163	846368
NIC received-byte	1466	759	1288283755	1272012684
Interrupt generated by NIC	163952	164108	842673	843381

Table 4-12 Two UDP Stream Test Simultaneously on the Intel Xeon Cluster

There was a slight time drift for the two processes because they were simultaneously started by hand. In a 10-second test, the overall throughput sending to the server was

about (1232732395+1233802649)*8/10 ≈ 1973 Mbps. Only about half of the packets were received by the server (1288283755 and 1272012684, reported by the server's two processes respectively), and the rest were dropped by the Gigabit Ethernet switch. The overall network throughput measured by the server was about 986 Mbps (498+488). The system load in both the client and the server were similar to what was observed with a single communication session.

4.2.3 Summary and Comparison

From the experiments, we clearly see that the systems based on Intel Xeon Architecture performed much better than the systems based on the Alpha Architecture, specifically, with lower system load and higher throughputs. The interrupts coalescence technique (reception) was used in the Alpha system but not in the Intel Xeon system. So we also expect that the network response in the Intel Xeon system is faster than in the Alpha system.

From the analysis of UDP communication with larger socket buffer size and datagram size, we can conclude that in the Alpha system, the sender was the bottleneck during the communication, and there was significant data loss with large socket buffer because the UDP sender was not fast enough to process the data and discarded it instead. In contrast, there was no an obvious bottleneck in the Intel Xeon system, where all UDP, TCP, and MPI communication could reach a very high throughput close to the limits of Gigabit Ethernet's bandwidth.

Network communication consumes considerable system resources, especially CPU cycles. High network throughput introduces a large number of interrupts into the system. When back-end machines were extremely busy (with a 100% CPU load), the network performance could significantly degrade, and the real throughput in this case could drop to 1%, as compared to the measurement of idle machines in the Alpha clusters.

4.3 Blocking and Non-blocking Communication

Socket communication is blocking by default on most Unix platforms. Sometimes, however, we need non-blocking I/O for multiplexed communications. In this section, we

have a brief analysis of these two communication models, and test their throughput in the Intel Xeon system and the Alpha system.

For blocking socket receiving operations, the system waits for incoming data from the network; when packets arrive, the system copies the data into a kernel buffer, processes the protocol operations, and then copies the payload into an application buffer. During the waiting and processing, the application is blocked waiting for the input system call to complete, and the invocation returns when data is copied into the application buffer.

For blocking socket sending operations, the application copies data from the user space into the kernel socket buffer and then returns the size of how much the data had been copied. When the kernel buffer is full, in TCP communications, the process is put to sleep until the socket buffer is available. For UDP communications, the operations may differ between different operating systems, where the kernel may flush the socket buffer, drop the datagram, or wait like TCP does. The traditional BSD socket implementation does not really have a UDP send socket buffer, but only specifies the maximum datagram size that can be written into a socket. Each datagram is just passed on to the next stack, and it may be lost if the lower layer processing is not fast enough to keep up with the rate of creation of UDP datagrams in the socket layer. In Linux, UDP sending system calls may block if the socket send buffer is full (no room for *sk_buff* allocation), as was the case in UDP communication with a 64KB default socket buffer in the Alpha system we discussed in Section 4.2.1.4.

In Unix platforms, for non-blocking socket communication, all sending and receiving system calls, such as *write()*, *sendto(), read(), recvfrom(),* and so on will return immediately, even if there is no successful data transfer between the application and the kernel.

Our previous experiments and discussions were based on blocking communications. To compare the differences, we conducted additional experiments on the Alpha and Intel Xeon systems. The results are listed in Table 4-13.

Experiments showed that, on both Intel and Alpha clusters, unidirectional throughputs were almost the same for blocking and non-blocking TCP and MPI communications, although non-blocking communications had a higher CPU load. These results are not

surprising. Although blocking and non-blocking applications have different interactions with the kernel, the underlying TCP/IP stacks control network communication, and the achievable throughput is determined by the network itself and the kernel networking implementation. The different CPU loads came from the application level. For non-blocking throughput tests, Hpcbench kept sending and reading data (system calls) until the condition (e.g. test time or transferred data) was satisfied. Since each sending or receiving invocation returned immediately for non-blocking I/O, the total number of system calls was much higher than that of blocking communication for the same amount of data transferred, even although *select()* was used to access the ready network handles or descriptors (select() itself is also an expensive system call.)

Experiments conducted on two idle nodes on the Alpha cluster (*greatwhite*) and the Intel Cluster (*mako*), with 64KB message size and system default socket buffer. Means of 10 repetitions.

Protocol	Test Mode	Throughput (Mbps)		Sender CPU load		Receiver CPU load	
		Intel	Alpha	Intel	Alpha	Intel	Alpha
TCP	Unidirectional Blocking	941.5	519.8	22%	21%	22%	29%
	Unidirectional Non-blocking	941.5	517.5	38%	38%	34%	37%
	Bidirectional Blocking (ping-pong)	854.9	422.3	22%	21%	22%	21%
	Bidirectional Non-blocking	1827.8	721.7	47%	39%	45%	39%
MPI	Unidirectional Blocking	932.3	312.2	20%	20%	24%	27%
	Unidirectional Non-blocking	932.1	311.8	24%	26%	37%	32%
	Bidirectional Blocking (ping-pong)	796.6	287.1	26%	26%	26%	23%
	Bidirectional Non-blocking	960.8	316.5	31%	34%	35%	37%
UDP	Bidirectional Multiplexing I/O	1867.2	775.6	34%	27%	32%	27%

Table 4-13 Blocking vs. Non-blocking Communication

In contrast, TCP bidirectional throughputs varied significantly from blocking to non-blocking communications, because both ends participated in data sending and receiving.

On the Intel cluster, the non-blocking TCP communication throughput was almost double that of blocking communication, implying that the network system worked well in full-duplex mode. This can also be verified by UDP bidirectional tests. On the Alpha cluster, the bidirectional non-blocking TCP throughput increased about 40% over the blocking communication.

MPI communications, however, did not show such a radical change. This may because of an inefficient MPI implementation of MPICH1.2.5.2 for non-blocking communication, or it is not well tuned for the system. The MPICH (linked to TCP/IP stack) in both systems was compiled by *gcc* and was installed with default settings. In next chapter, we will see that MPICH-GM over Myrinet performs much better in the same system, and that MPI non-blocking bidirectional throughput is almost the double of its unidirectional throughput over Myrinet.

4.4 UDP and TCP Throughput

In this section, we present results of analyzing UDP and TCP throughput for inter-cluster and intro-cluster communication with different parameters. MPI communication throughput was not discussed because cross-cluster MPI communication was not available (the inner nodes did not support ssh or rsh services for remote clusters). We only focus on blocking, unidirectional stream experiments, because they directly show the network behavior with less benchmark overhead and other effects. Our experiments include both intra-cluster and inter-cluster communications in our test-bed. Since the routing settings in the Intel Xeon system (*mako*) can only route to one Alpha cluster (hammerhead), we only tested the other three Alpha clusters, *greatwhite*, *deepurple*, and *hammerhead*. For intro-cluster tests, we did experiments in two idle nodes in *greatwhite*. For inter-cluster tests, we examined the communication between *greatwhite* and *deeppurple*, which was connected via optical fibre (1KM distance); communication between *greatwhite* and *hammerhead* with long distance optical fibre link (150KM) was also tested. Note all these three clusters used the same kind of network devices (Alteon AceNIC, HP Passport 8600 switch) and software (Linux 2.4.21).

4.4.1 UDP Communication

We have discussed some issues related to UDP communication throughput in Section 4.2.3.4. We conduct further experiments in this section to determine the maximum achievable throughput in our test-bed associated with socket buffer size, datagram size, and different connections. We chose 1KB, 1460-byte, and 4KB datagrams for our UDP tests, with four different socket buffer sizes: 10KB, 100KB, 1MB, and 10MB. Table 4-17 shows the averages (means) of test results.

Unidirectional UDP throughput test inside and between *greatwhite*, *deeppurple* and *hammerhead* (Mbps). Mean of ten replications.					
Datagram (Byte)	Link	Socket buffer			
		10KB	100KB	1MB	10MB
1024	gw→gw	165.11	485.22	575.30	574.79
	gw→dp	162.27	471.38	556.33	559.54
	gw→hh	162.31	459.07	557.15	568.03
1460	gw→gw	177.75	557.43	649.83	647.75
	gw→dp	177.68	541.88	628.19	630.91
	gw→hh	177.59	539.45	636.77	638.83
4KB	gw→gw	147.64	549.46	586.02	583.32
	gw→dp	146.55	540.11	539.13	541.22
	gw→hh	147.30	536.34	538.54	537.49
40KB	gw→gw		567.09	1.15	1.13
	gw→dp		564.46	1.16	1.13
	gw→hh		565.78	1.15	1.14

Table 4-14 Intro/Inter-cluster UDP Communication Throughput

From the results, we have several observations. First, the throughput increased when the datagram size increased from 1KB to 1460-byte, but it dropped when the datagram size increased from 1460-bytes to 4KB; when datagram size increased to 40KB and the socket buffer size was large (1MB and 10MB), the network throughput decreased to be almost

negligible. Second, throughput increased when socket buffer size increased, but remained unchanged for larger socket buffer sizes once it was large enough. Third, the throughput varied a bit for different links, but the difference was not significant.

To explain the first case, we look at the packet size for the different datagrams. The MTU size in our test-bed was 1500-byte, implying the maximum 1472-bytes UDP payload for each packet. When datagram size increased from 1024-bytes to 1460-byte, the total transmitted packets became less for the same amount of data transferred and the system overhead was reduced, resulting in higher throughput. Throughput decreased when datagram size increased from 1460-bytes to 4KB As we discussed in Section 4.2.2.4, two reasons could explain this. The 4KB datagram had to be fragmented into 3 packets to fit the MTU size before transferring in the network, resulting in some smaller packets requiring transmission.

Furthermore, the whole datagram would be discarded if there was a single packet loss, i.e. one packet (less than 1473-byte) lost would lead to 4096 bytes of data lost. When datagram size increased to 40KB, the situation became worse. One datagram was separated into at least 28 packets, and the total 40KB of data would lost if any of these 28 packets were lost. When the socket buffer size was not too large, 100KB in our example, the bottleneck was the socket buffer itself (the application was blocked from sending when the socket buffer was full), so there was not much data loss during kernel processing and the network throughput was expectedly high. However, when the socket buffer was large enough, 1MB and 10MB in our case, the socket buffer limit was eliminated, and the UDP data was randomly dropped when the data transferring from the application exceeded the kernel's (or NIC's) capabilities. When data loss caused by the sender itself was considerable, there were few complete datagrams that could be reassembled back in the server. The system log files for these tests verify our assumption. We found that the actual network throughput measured by the server's NIC was greater than 500 Mbps for both cases (1MB and 10MB socket buffers for 40KB datagrams), but the application in the server only reported a few of datagrams actually received. Similar test conducted in the Intel Xeon system did not show such a behavior and the throughput remained around 950 Mbps when both the socket buffer size and datagram size were large (Section 4.2.2.4).

Without considering the effects of data loss, the kernel was able to process more packets in one round with larger socket buffers, so the application overhead of the sending process was reduced, and the throughput could increase with a larger socket buffer in general. When the maximum throughput was reached, larger socket buffers were useless. We can see this trend from the results. The throughput increased dramatically with buffer size increasing from 10KB to 100KB, and there was relatively little change when socket buffer size increased from 100KB to 1MB, and from 1MB to 10MB as well.

The results table shows that the maximum UDP throughput was about 630~650 Mbps for both intro-cluster and inter-cluster communication, which was measured in the case of a 1460-bytes datagram size with a large socket buffer size (1MB and 10MB). Similar throughputs between inter-cluster and intro-cluster UDP communication shows that the fibre optic network was able to provide sufficient bandwidth for Gigabit Ethernet communications over long distance.

4.4.2 TCP Communication

For TCP throughput tests, we chose 10KB, 100KB, 1MB, and 10MB socket buffer sizes with three different message sizes: 100KB, 1MB ,and 10MB. Tests were conducted in the same way as the UDP tests. Table 4-18 shows the test results.

Unidirectional TCP throughput test inside and between *greatwhite*, *deeppurple* and *hammerhead* (Mbps). Means of ten replications.

Message Size	Link	Socket buffer			
		10KB	100KB	1MB	10MB
10K	gw→gw	108.34	513.86	568.13	587.71
	gw→dp	88.79	495.32	565.79	572.33
	gw→hh	12.30	152.04	527.51	535.40
100K	gw→gw	119.85	515.47	574.22	589.43
	gw→dp	98.41	504.30	570.44	573.27
	gw→hh	13.82	157.65	541.89	549.20
	gw→gw	117.27	510.82	573.15	590.54

10MB	gw→dp	98.22	504.41	567.08	567.18
	gw→hh	13.85	155.33	534.67	550.14

Table 4-15 Intro/Inter-cluster TCP Communication Performance

In contrast to UDP, TCP throughput dropped significantly for cross-cluster communication when the socket buffer size was small, especially in long distance communication between *greatwhite* and *hammerhead*: with a 10KB socket buffer, UDP could achieve more than 140 Mbps throughput in any datagram size, while TCP throughput was less than 14 Mbps for all message sizes. This is likely due to TCP's flow and congestion control. As we pointed out in Section 4.1.1, the data sending process is controlled by the TCP window. To guarantee reliable delivery of data, the TCP active window shrinks during sending, and no more data will be sent when the TCP window closes. So the theoretical maximum throughput is window-size/RTT-time if there is enough bandwidth for the data transfer. In our example, the network RTT time between *greatwhite* and *hammerhead* was about 2.9ms (refer to Section 4.6), the maximum throughput for a 10KB socket buffer (actual usable buffer is only 5KB in Linux) is about 5*1024*8/0.0029 ≈ 14.12 Mbps, agreeable with our test results. For 100KB socket buffer, this value is about 5*1024*1024*8/0.0029 ≈ 144.63, also close to the results observed.

For a "long fat pipe" link, the optimal buffer size setting should be at least twice of Bandwidth Delay Product (BDP), or Bandwidth*RTT (Section 4.1.1). The connection between *greatwhite* and *hammerhead* is via DWDM optical fibre with a 1Gbps bandwidth. To achieve the maximum throughput in this link, the socket buffer size should be at least $(1*10^9/0.0029)/8$ = 362.5KB. For Linux and Hpcbench, this value should be 365.2 * 2 = 725KB. Our experiments with 1M and 10MB socket buffers satisfied this condition. In such configurations, TCP throughputs by different links varied slightly.

We can also observe that the message size had little impact on measured throughput since TCP is a byte-stream protocol. The achievable maximum TCP throughput in our tests was about 590 Mbps on the cluster *greatwhite*, 570 Mbps between *greatwhite* and *deeppurle*, 550 Mbps between *greatwhite* and *hammerhead*.

4.5 Network Communication Latency

Another important attribute of high performance networks is network latency. In our Gigabit Ethernet environment, network latency is a measure of the time taken for a tiny packet to travel from the sending application through the network adapter, over the communication link, through the destination's network adapter and into the receiving application. So, it is affected by a number of factors: operating system, protocol overhead, characteristics of the network devices, such as NICs and switches, network congestion, the physical distance traveled. Most benchmarks, including Hpcbench, can only measure the Round Trip Time (RTT) instead of the one way network latency. Figure 4-7 shows the RTT measurement model.

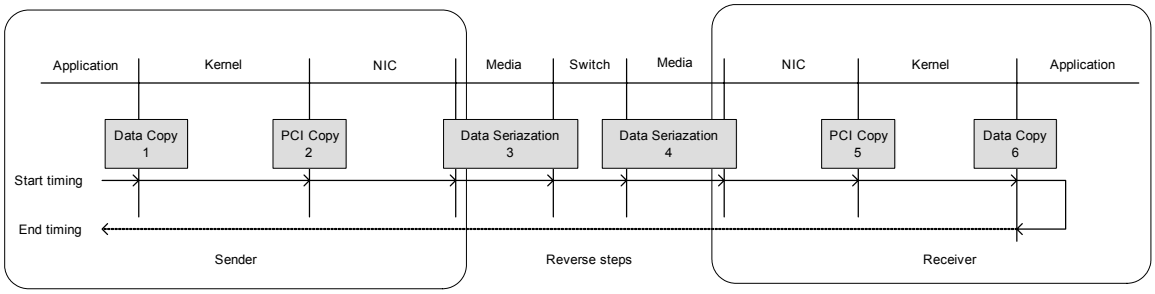

Figure 4-7 Round Trip Time Distribution

(Times in boxes are depending on packet length)

High throughput does not necessarily imply low latency, and vice-versa. Throughput may be increased by adding more data channels and running packets through them in parallel. The latency factor, however, is usually not so easy to decrease. Some applications are very sensitive to network latency, such as interactive communication sessions. In HPC environments, the compute nodes may need to exchange small data packets frequently. In such a case, network latency plays a crucial role.

We tested the network latency in the Alpha systems, including the intra-cluster and inter-cluster communications. The results are shown in Table 4-16.

ICMP RTT was tested by the default *ping* utility. Other RTT tests were measured by Hpcbench. All results were the median of 10 trials with 64-bytes RTT message size.
VLAN: over private IP, directly linked by switches in SHARCNET.
Internet: communication over Internet (gw1 and hh1 include two NICs connected to SHARCNET and Internet).

Protocol	gw1-gw1 (Local)	gw1-gw10 (VLAN)	gw1-dp1 (VLAN)	gw1-hh1 (VLAN)	gw1-hh1 (Internet)
ICMP RTT	0	0	0	2.924 ms	3.907 ms
UDP RTT	87.15 µs	245.37 µs	322.36 µs	3.128 ms	4.792 ms
TCP RTT	102.05 µs	251.78 µs	352.88 µs	3.174 ms	4.853 ms
MPI RTT	174.58 µs	358.91 µs	367.46 µs	3.266 ms	

Table 4-16 RTT Tests between Different Alpha Systems

Notice we only conducted tests on the Alpha clusters, since the routing setting in *mako* cluster disabled the communication between *mako* and greatwhite, *mako* and *deeppurple*. As well, the MPI communication between *greatwhite* and *hammerhead* was only available on their master nodes (gw1 and hh1). The results showed that the roundtrip time latency for TCP/IP communication inside the cluster of *greatwhite* was about 240 µsec, about 320 µsec between *greatwhite* and *deeppurple*, and then rose to the millisecond level between *greatwhite* and *hammerhead*. The dramatic increase was due to the long distance (150KM) of the connection between these two clusters.

We then evaluated the network latency between two nodes inside an Alpha cluster (*hammerhead*). All tests were repeated 10 times and the average results are shown in Figure 4-8. We observe that the UDP and TCP RTTs share a similar but not smooth shape of delay function, and the MPI RTTs were higher than those of TCP and UDP. For tiny packets, the RTTs of TCP and UDP were about 230-250 µsec, and MPI RTTs started from about 350 µsec. The curves for all protocols are not linearly smooth because of the impact of interrupt coalescence.

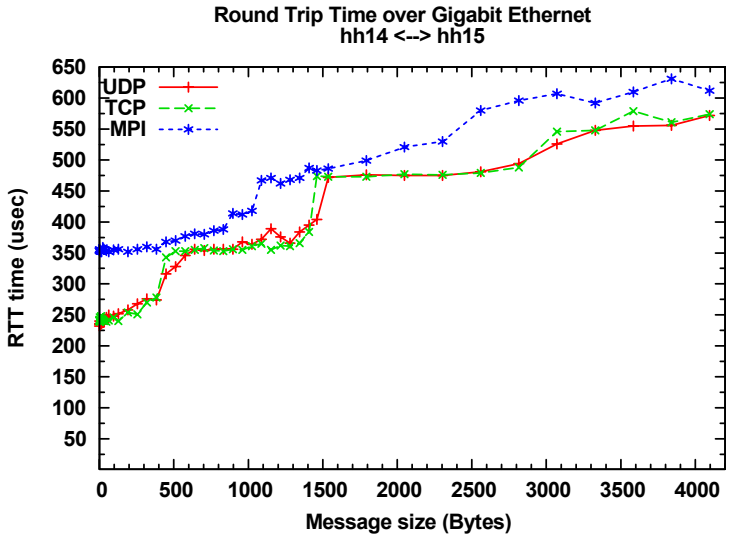

Figure 4-8 RTT Test over Gigabit Ethernet on an Alpha Cluster

We conducted similar experiments in the Intel Xeon cluster (*mako*), where there was no interrupt coalescence for packet reception (recall Section 4.2.2). From the results (Figure 4-9) we can see that all UDP, TCP and MPI RTTs in *mako* were lower than 80 µsec when the message size is less then 128-bytes, which are much lower than those of the Alpha system. As well, all three curves are very smooth, and the measured RTTs were linear to small message sizes. Figure 4-10 shows these slope functions. There is a clear change of the curves around the 1500-bytes message size. This is because the data greater than MTU size was fragmented into a second packet and then sent to the network. The slope functions tell us the communication latency characteristics with different protocols. For instance, consider the UDP's RTT function: y=0.0536x+54.7, the fixed delay (54.7µsec) is caused by protocol overhead such as the execution time for certain instructions, and the delay associated with data size is caused by data copying and transferring in the network.

From these experiments, we can conclude that the interrupt coalescence technique introduces extra network delay, resulting in a longer response time for network communications. This has a negative effect for time-sensitive parallel computation, although it helps to lower system load.

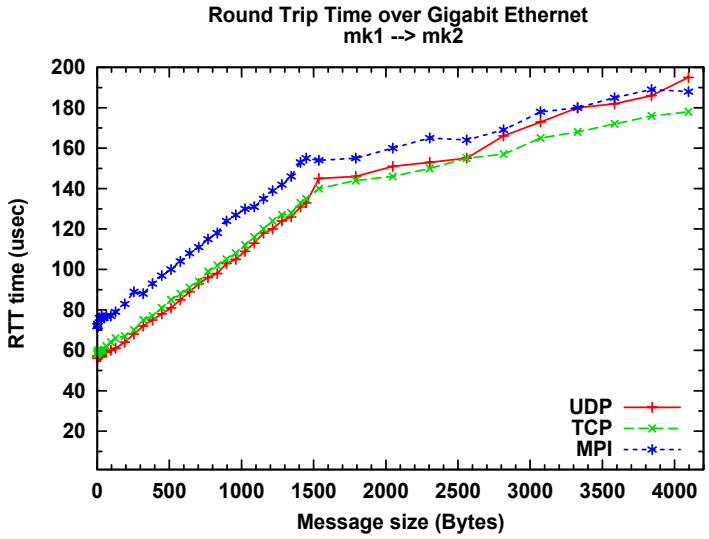

Figure 4-9 RTT Test in Mako

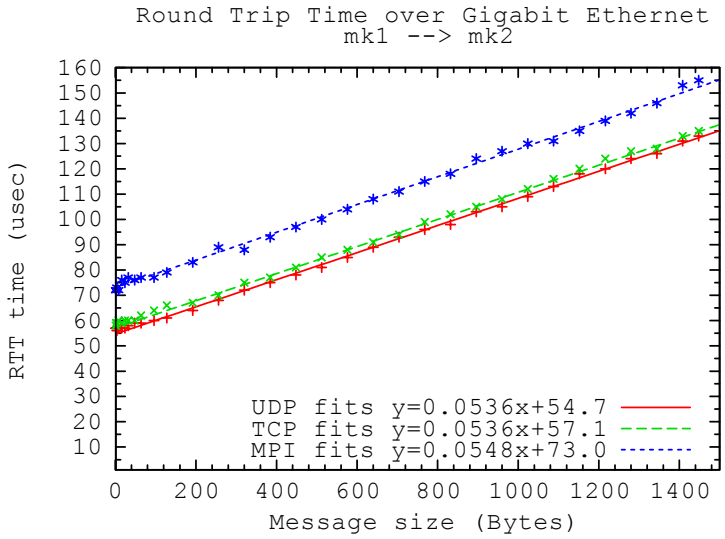

Figure 4-10 Slope Functions of RTT vs. Message Size

Although many people likely associate high performance networks with high throughput (bandwidth) systems, the attribute of low latency also plays a key role in optimizing network communication performance. From our experiments, the Intel Xeon system, which performed much better than the Alpha system, still had about a 60 µsec RTT, which may still not satisfy some latency-sensitive parallel applications. The relatively high latency of Gigabit Ethernets is one of the reasons that some proprietary technologies, such as Myrinet and QsNet, are more commonly used in many extremely high

performance computing clusters. We will have more discussions about Myrinet and QsNet in next chapter.

4.6 Summary

In this chapter, we studied the Gigabit Ethernet technology and its performance. We conducted a wide variety of experiments in Alpha SMP and Intel Xeon SMP clusters, to estimate and measure the network behavior of Gigabit Ethernet. There are several observations and conclusions based on the results reported by Hpcbench:

- High network throughput requires plenty of system resources. Lack of system resources, such as heavy CPU load, may lead to poor network performance.
- The interrupts coalescence technique can be used in Gigabit Ethernet adapters to reduce system load; however, it will increase response times and lead to higher network latency. In our test-bed, the Alpha systems enabled interrupts coalescence, and its intra-cluster UDP/TCP roundtrip time was about 240 μsec; the Intel Xeon system did not apply interrupts coalescence, and its UDP/TCP roundtrip time was slightly less than 60 μsec.
- The bottleneck of a Gigabit Ethernet is usually the back end machine, not the switch. In the Alpha system, the sender is the bottleneck. We found a lot of data lost during the UDP end-to-end communication with sufficient socket buffer sizes, but there was no frame loss in the network. The switch dropped the frames only when multi-linked data delivering to one end exceeded Gigabit Ethernet's bandwidth.
- Without transmission controls, UDP communication has lower protocol overhead than TCP, hence UDP communication can reflect network characteristics more closely. The maximum achievable UDP throughput should be always greater than maximum TCP throughput when the communication parameters are well tuned.
- MPI communication over Gigabit Ethernet is usually built on top of TCP, and thus has more protocol overhead than UDP and TCP. In the Alpha systems, MPI throughput was only around 300Mbps. The poor MPI performance may be possibly due to the inefficient implementation of MPICH.

- A maximum UDP throughput can be achieved with three important conditions: full MTU size frames, sufficient socket buffers, and less application overhead, i.e. we should configure the payload for each sending close to MTU size, a large socket buffer size such as 1MB, and minimize the system calls in an application.
- Different network cards and their drivers result in very difference network behavior. In the Alpha clusters, for UDP communication, the sender's local throughput could be much higher than the actual network throughput when socket buffer was large, and the gap between them showed that there was significant data loss inside the kernel of sender. On the other hand, this situation never happed in the Intel Xeon cluster; the sender's throughput in the application layer always matched the network throughput, and there was no UDP datagram loss in the kernel.
- In a long distance and high bandwidth connection, sufficient socket buffers are extremely crucial for TCP communication. To achieve maximum TCP throughput, the socket buffer size should be at least twice of the Bandwidth Delay Product (BDP).
- The Intel Xeon system performed much better than the Alpha system. The Intel Xeon system (*mako*) could easily deliver 955 Mbps for UDP, 940 Mbps for TCP and 930 Mbps for MPI with default system settings. On the contrast, on the Alpha system (*hammerhead*), these three numbers were around 505 Mbps, 525 Mbps, and 315 Mbps respectively. The maximum UDP throughput in Alpha system (*hammerhead*) was about 650 Mbps measured by Hpcbench. One key reason that the Intel Xeon system had better performance came to its faster CPUs that were able to handle the heavy network processing. Better processors usually lead to higher network throughput.

With the above analysis, we showed how Gigabit Ethernets behave in different situations, how network communications interact with the Linux operating system, and how we can optimize the system's performance.

Chapter 5

Performance of Myrinet and Quadrics Interconnects

In our SHARCNET test-bed, besides Gigabit Ethernet, three Alpha clusters also included a Quadrics' QsNet interconnect, and the Intel Xeon cluster included Myrinet. Application communication over these interconnects is done using MPI libraries provided by vendors for parallel computing. TCP/IP communication over these two proprietary high speed interconnects is currently unavailable (Not set up yet). In this chapter, we look at MPI communication over these proprietary interconnects.

Since MPI implementations built for these interconnects were a high level communication middleware, only a very few parameters can be set. For instance, the underlying "socket" buffer is not configurable in the application layer. We tested MPI point-to-point blocking communication using *MPI_Send()* and *MPI_Recv()*, and non-blocking communication using *MPI_Isend()* and *MPI_Irecv()* function calls.

5.1 MPI Communication Performance

To evaluate the performance of Myrinet and QsNet in our test-bed, we tested their MPI communication throughput and round trip time. All experiments can only be conducted inside each cluster, since Myrinet and QsNet are tightly coupled interconnects and it is impossible to launch an MPI communication session between two clusters over these high speed interconnects. We selected two idle nodes in *mako* to test the Myrinet communication, and two idle nodes in hammerhead to test QsNet communication for our discussion. All MPI communication is based on the same version of MPICH (MPICH-GM).

5.1.1 Myrinet

We have seen MPI communication over Gigabit Ethernet in *mako* in Section 4.2.2.3, with throughput about 930 Mbps. We now test MPI communication over Myrinet. By default, the *mako* cluster has two MPI implementations installed: MPICH-GM 1.2.5 and

LAM/MPI 7.0.6, and both are linked to the Myrinet interconnect. We used Hpcbench to measure the MPICH communication throughput between mk1 and mk2 (defined in the conf file) over Myrinet with message size increasing exponentially from 1-bytes to 128MB (2^{27}):

```
[mk1 ~/hpcbench/mpi]$ /usr/local/mpich-gm/bin/mpirun -p4pg conf mpitest
-e 27 -o mpi.txt
```

The above command tests unidirectional blocking communication. We also evaluated non-blocking and bidirectional communication. All tests were repeated 10 times, and the means are shown in Figure 5-1.

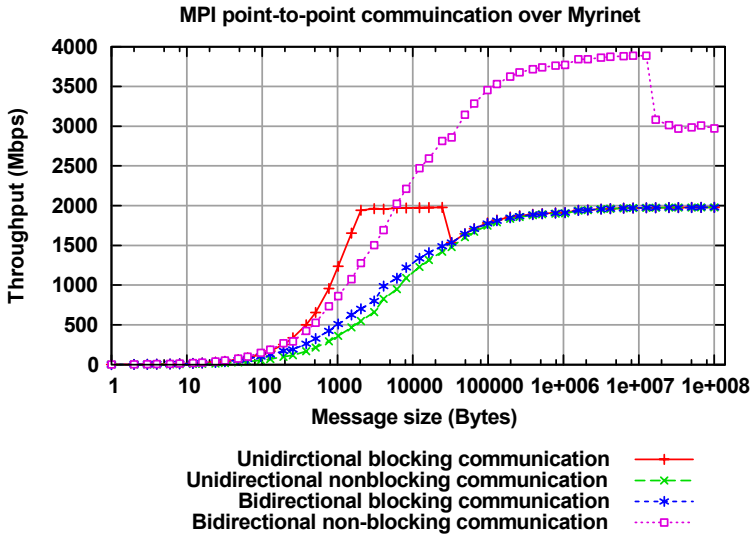

Figure 5-1 MPI Point-to-Point Communication Throughput over Myrinet

Results show that Myrinet maintained a sustained unidirectional throughput around 1950 Mbps when message size was not too small. Unidirectional blocking communication (*MPI_Send/MPI_Recv*) throughput had a sharp jump, reaching its maximum throughput 1976 Mbps with a small 1KB message size, and had a clear drop for message size increasing to about 30KB. There was also a throughput slump for bidirectional non-blocking communication when message size increased from 12MB to 16 MB. These unstable behaviors may come from implementation details of Myrinet or the MPICH-GM libraries.

The maximum bidirectional non-blocking throughput reached 3885 Mbps with a message size of 12MB, exactly the double of the maximum unidirectional throughput, showing that Myrinet supported full duplex communications very well.

We then tested the network latency (Round Trip Time) of Myrinet on *mako* using Hpcbench (the file defines the master process mk1 and the slave process mk2):

```
[mk1 ~/hpcbench/mpi]$ /usr/local/mpich-gm/bin/mpirun -p4pg conf mpitest
-A 1 -o mpi-rtt-1.txt
```

The above command evaluates the MPI Round Trip Time, based on blocking communication (*MPI_Send/MPI_Recv*), ten times with a message size of 1 Byte. We used a shell script to scan the RTTs for different message sizes using Hpcbench. Figure 5-2 shows the means of results of ten repetitions.

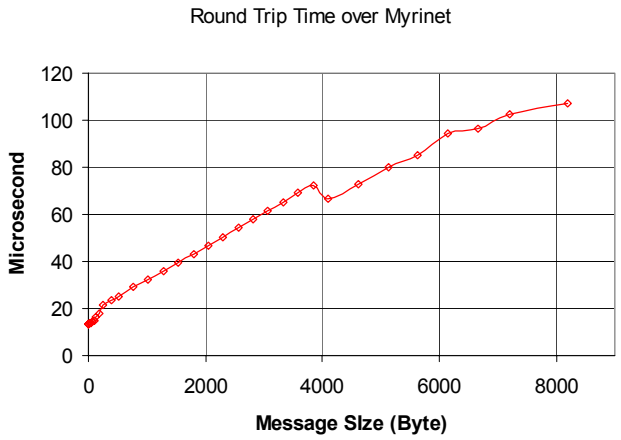

Figure 5-2 MPI Point-to-Point Communication Round Trip Time over Myrinet

The MPI RTTs over Myrinet were less than 14 µsec for small size messages (1-16 Bytes), implying less than 7 µsec of one way network latency for MPI communication. This is a tremendous improvement when compared to the Gigabit Ethernet we studied. The curve is linearly smooth for message size smaller than 4KB, and there is a clear drop after that. As in TCP/IP communication, when a second packet has to be sent when the message size is greater than the MTU size, there is a slight degradation in performance.

5.1.2 Quadrics' QsNet

In our test-bed, three Alpha clusters (*hammerhead*, *greatwhite,* and *deeppurple*) included both Gigabit Ethernet and QsNet interconnects. MPI communication over Gigabit Ethernet in the Alpha systems was examined in the Section 4.2.1.3, which showed a poor throughput around 310 Mbps. In this section, we test MPI point-to-point communication over QsNet.

The default MPI implementation in these three clusters was MPICH 1.2.5 linked to the QsNet interconnect. We first used Hpcbench to measure the MPICH communication throughput between hh14 and hh15 over QsNet with the message size increasing exponentially from 1-bytes to 128MB (2^{27}). The *mpirun* script is disabled on the Alpha clusters since they use Platform's LSF queuing system; consequently, we submit the following to LSF:

```
[hh1 ~/hpcbench/mpi/]$ bsub -n 2 -extsched "nodes=2" -m "hh14 hh15" -q
short prun -n 2 -N 2 mpitest -e 27 -o mpi.txt
```

The above command tests unidirectional blocking communication. We also tested non-blocking and bidirectional communication. All tests were repeated by 3 times, and the averages are shown in Figure 5-3.

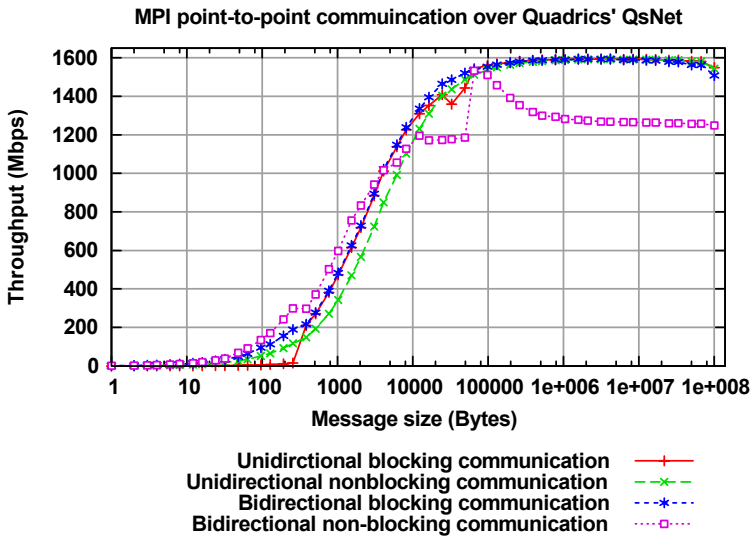

Figure 5-3 MPI Point-to-Point Communication Throughput over QsNet

The experiments showed that the maximum throughput for unidirectional communication was sustained at about 1600 Mbps when the message size was greater than 100KB. All curves change smoothly except the bidirectional non-blocking communication, showing

MPI communication over QsNet was stable. Compared to Myrinet, the bidirectional non-blocking communication throughput on QsNet did not behave well, and was less than the unidirectional throughput with a message size larger than 10KB. This tells us that the QsNet in the Alpha system does not support full duplex communication very well. Considering the bottleneck in UDP communication on the Alphas, the reason for low bidirectional non-blocking throughput may be due to the relatively slow Alpha servers, which might not be fast enough to handle the heavy system load introduced by the asynchronous I/O communication.

We then evaluated the RTTs of MPI communication over QsNet using Hpcbench. The tests were conducted between *hh14* and *hh15* on *hammerhead*, and were repeated 10 times. The averages of the results are shown in Figure 5-4.

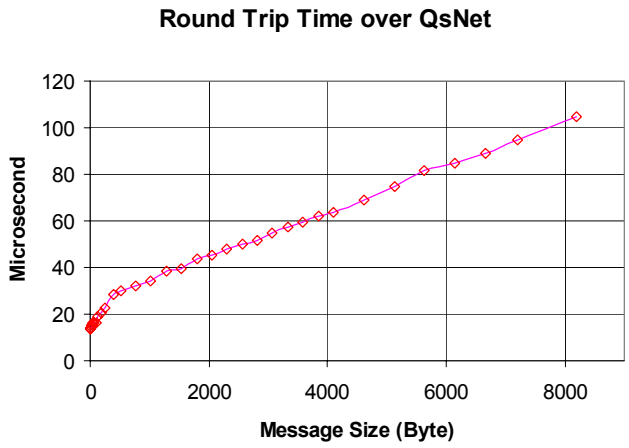

Figure 5-4 MPI Point-to-Point Communication Round Trip Time over QsNet

As in Myrinet, the network latency of QsNet was very good. The MPI RTTs over QsNet were also less than 14 μsec for tiny messages (1 Byte to 32 Byte). The curve is relatively linear, showing that the response time of MPI communications was stable.

5.2 A Comparison to Gigabit Ethernet MPI Communication

From the experiments, we know that Myrinet and QsNet performed much better than Gigabit Ethernet in both achievable throughput and network latency. If we look at the statistics of Myrinet and QsNet communication (Table 5-1), we observe that the CPU load during the tests was purely in user mode, and there were no CPU cycles from system

mode for both the sender and receiver. This may be due to the fact that both of these two interconnects employ a zero copy design for message passing.

Generally speaking, both technologies use a global virtual memory concept, where message access (passing) to remote hosts is done by pointing to a special virtual memory area mapped into their local network interface cards (e.g. Elan in QsNet), which are able to transfer the data from the application to the interconnect network by analyzing the virtual memory address that the application referenced. There are no system calls involved in the network processing (no socket created, for instance), and the function calls for network communication are directly linked to the driver of network interface card, without bothering the kernel to process the communication protocols. So there are no interrupts of the kernel involved in network communication. All data exchanged moves from user space through the NICs directly. This can lead to very high speed communication and dramatically reduces system overhead. At the same time, since the application runs completely in user space, each network communication session can only be served by a single CPU in an SMP system without the load distribution support.

MPI point-to-point communications with message size of 1MB. Tests were conducted in two idle nodes in *mako* (Myrinet) and *hammerhead* (QsNet).

Test mode	Myrinet			QsNet		
	Throughput (Mbps)	Sender CPU load	Receiver CPU load	Throughput (Mbps)	Sender CPU load	Receiver CPU load
Unidirectional Blocking	1914	25%	25%	1596	25%	25%
Unidirectional Non-blocking	1911	25%	25%	1594	25%	25%
Bidirectional Blocking	1913	25%	25%	1596	25%	25%
Bidirectional Non-blocking	3784	25%	25%	1307	25%	25%

Table 5-1 Statistics of MPI Communication over Myrinet and QsNet

Experimental results reflect these designs. In both Myrinet and QsNet, there is only one CPU participating in each communication for both the sender and receiver. The results also show that this CPU was completely occupied (100% load) during the communication, resulting in a 25% overall CPU load for 4-processor SMP systems (Table 5-1).

As we have seen in the previous section, the throughput of bidirectional non-blocking communication in Myrinet was about the double of its unidirectional non-blocking throughput, but this did not occur with QsNet. We now can give a further explanation. Since there is only a single CPU participating in networking for each session in these Zero-copy technologies, the achievable throughput is more affected by speed limit of each CPU in an SMP system. Compared to a TCP/IP communication workload that was dynamically distributed to all four CPUs, for the high-speed and low-latency QsNet, the relatively slow Alpha servers (4x833MHz CPUs in *hammerhead*) appeared to not have enough CPU power to handle heavy bidirectional non-blocking communication.

Unlike Gigabit Ethernet's star network topology, Myrinet and QsNet use a complex interconnection structure, such as QsNet's fat-tree topology. This may improve the throughput for one node with several links at one time. In Gigabit Ethernet, no matter how many connections exist in one host, its maximum throughput of incoming and outgoing traffic is limited to 1 Gbps theoretically. We examined the capacity of multiple-link communication over Myrinet in the *mako* cluster. Figure 5-5 shows the experiment configuration and Table 5-2 shows the test results.

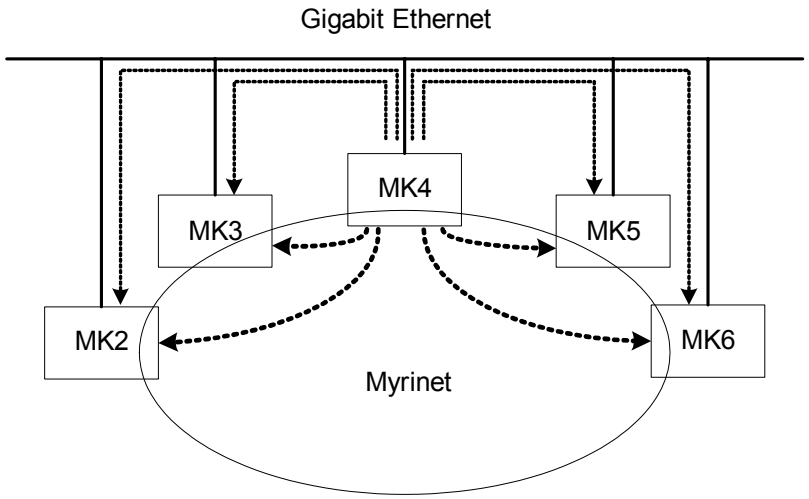

Figure 5-5 Multilink Communication over Gigabit Ethernet and Myrinet

MPI stream tests with multiple links simultaneously. Sender mk4, 1MB message size, blocking communication. One link: mk4→mk3; Two links: mk4→mk3, mk4→mk5; Four links: mk4→mk2, mk4→mk3, mk4→mk5, mk4→mk6; Six links: mk4→mk2, mk4→mk3, mk4→mk5, mk4→mk6, mk4→mk7, mk4→mk8				
Test mode	Gigabit Ethernet		Myrinet	
	Overall throughput	Sender CPU load	Overall throughput	Sender CPU load
One link	930 Mbps	20%	1915 Mbps	25%
Two links	938 Mbps	22%	1970 Mbps	50%
Four links	942 Mbps	23%	1980 Mbps	100%
Six links	940 Mbps	24%	1978 Mbps	100%

Table 5-2 Throughput of Multiple Connections on Gigabit Ethernet and Myrinet

Although the total throughput increased a bit with more connections, it never passed a limit for both Gigabit Ethernet and Myrinet. For MPI point-to-point unidirectional communication, this limit was about 940 Mbps for Gigabit Ethernet and around 1980 Mbps for Myrinet.

There is no kernel control (system calls) for communication over Myrinet, and the applications were just trying to send as much data as possible to the Myrinet NIC. In our throughput tests with Hpcbench, this consumed as much of the CPU power as possible. When the number of connections is larger than the number of CPUs, the system will be fully loaded and the interconnection link is saturated at the same time. This is a side effect of zero-copy technique. For MPI communication over Gigabit Ethernet based on TCP/IP, the CPU load did not increase dramatically. As we pointed out in Section 4.2.2.4, the traditional interrupt-driven TCP/IP communication over Ethernet has a transmission control mechanism between the kernel and NIC driver, and a control between the NIC and the network switches or routers (IEEE 802.3x), thus the applications will not overwhelm the system resources if all the applications are trying to utilize as many network resources as possible. However, unlike some web-based services (e.g. ftp server) where many processes extensively transferring a large amount of data simultaneously,

applications on HPC systems usually are computational complex, so the situation of 100% CPU usage only for the network communication does not likely occur. Myrinet and QsNet were no doubted a better interconnect for HPCs than Gigabit Ethernet.

5.3 Summary

In this chapter, we analyzed the performance of two popular proprietary interconnects, Myrinet and Quadrics' QsNet. From the experimental results, both interconnects showed a high throughput point-to-point communication and a low network latency. Myrinet in the Intel Xeon system could deliver about 2000 Mbps unidirectional throughput and 3900 Mbps bidirectional non-blocking throughput; QsNet in the Alpha system gave about 1600 Mbps unidirectional and bidirectional throughput.

During the high throughput communication tests, there were no interrupts generated by both Myrinet and QsNet, and CPU load was totally in user mode for network communications. This zero-copy transmission model is much efficient than traditional interrupt-driven communication model, resulting in lower system load and higher network throughput.

Both Myrinet and QsNet had very low network latency. Their measured MPI Round Trip Times for tiny messages were less than 14 μsec. This is much lower than that of Gigabit Ethernet (about 60 μsec in the Intel Xeon system and 240 μsec in the Alpha system). Low network latency is crucial for some time-sensitive applications. In HPC clusters, all compute nodes use the same file system, so there are usually a lot of short messages. High latency can degrade network performance in situations relying on small messages, even though the network supports a very high throughput. Although Myrinet and QsNet were deployed in SHARCNET, the NFS systems used were based on Gigabit Ethernet and cannot take advantage of these proprietary technologies to improve performance due to the critical issue of communication latency. A possible upgrade is to enable TCP/IP communication over these two interconnects, and deploy the NFS servers to the domain of Myrinet and QsNet, then we would take more advantages from the high performance Myrinet and QsNet.

Chapter 6

Conclusions and Future Work

6.1 Thesis Summary

In this thesis, we presented a comprehensive study of high performance networks in HPC clusters. The motivation for this thesis is the question of how well the network system of commodity clusters performs, and what factors can affect this network performance. The analyses of Gigabit Ethernet, Myrinet and QsNet, were based on experiments conducted within SHARCNET.

Chapter 2 provided background research on HPC systems, including the history of HPC, networking, message passing, storage deployment, and so on. Detailed descriptions of the various components of our experimental test-bed environment were also provided.

A survey of current popular network measurement tools was presented in Chapter 3, which led to the motivation for a new network benchmark to analyze HPC networks. To address this need, we developed the Hpcbench tool set, and discussed its implementation in this chapter as well. In particular, its communication model, timers, synchronization issues, UDP measurement pitfalls, and other key issues for accurately evaluating network performance were discussed.

The network performance of Gigabit Ethernet was investigated in Chapter 4. Theoretical studies included TCP/IP properties, interrupts coalescence, jumbo frames, and zero-copy techniques. Experimental research was based on UDP, TCP and MPI communication tests on Alpha and Intel Xeon systems. The network characteristics and performance factors for both systems were extensively analyzed. MPI communication over Myrinet and Quadrics' QsNet was studied in Chapter 5, with a comparison of Myrinet and Gigabit Ethernet.

6.2 Contributions and Results

A comprehensive Linux-based network benchmark, Hpcbench, has been developed for HPC environments. Hpcbench accurately measures the network latency and throughput of high performance networks. In addition, it is capable of tracing the system information during this testing, which is very helpful for us to understand network behavior. With kernel information, not only can we measure this network performance, but also we can discover the bottleneck(s) of the system, why achievable throughput is low, and why latency is unacceptable.

We conducted a set of experiments on the SHARCNET Alpha and Intel Xeon clusters. The results showed that system architecture, configuration, workload, drivers of network interface cards, and other factors can significantly affect the network performance of these cluster environments.

We analyzed the communication performance of Gigabit Ethernet in an Intel Xeon SMP system and Alpha SMP systems. The experiments included UDP, TCP and MPI communications. The results showed that the Intel Xeon system performed extremely well, and provided a sustained high throughput between 930-960 Mbps for all three communication protocols. The system load for both sender and receiver was less than 25% in all cases, while the receiver usually had a higher CPU load than the sender for the communication of these protocols. The Intel Xeon system also showed a very good support for UDP and TCP full duplex communication, which offered almost double the throughput of unidirectional throughput.

The communication over Gigabit Ethernet in the Alpha systems gave a smaller throughput in our tests, with about 650 Mbps maximum UDP throughput, and 590 Mbps maximum TCP throughput. The MPI communication over Gigabit Ethernet in Alpha system performed poorly, showing around 300 Mbps unidirectional throughput. With the analysis of the kernel statistics, we found the bottleneck of UDP communication in Alpha system was the sender machine. There was significant data loss during the UDP communication when the socket buffers were sufficiently large. Experiments also showed an 890 Mbps UDP throughput could be achieved by sending two UDP streams into one end system in the cluster.

Gigabit Ethernet's network latency in Intel Xeon system was also less than that the Alpha system. The TCP/UDP Round Trip Time (RTT) on the Intel Xeon cluster started from 54-56 μsec for tiny packets, while this number increased to about 240 μsec on the Alpha cluster. In addition, the curve function of RTT versus message size was linearly smooth for the Intel Xeon system, but this did not hold for the Alpha system. One major reason was that the interrupts coalescence technique was used in the Alpha system. Interrupts mitigation could reduce the system load for networking, but introduced extra communication delay.

The proprietary Myrinet and QsNet technologies delivered higher throughput and lower latency than Gigabit Ethernet. For MPI (MPICH) communication, Myrinet could provide 1980 Mbps unidirectional throughput and about 3800 Mbps bidirectional non-blocking throughput; QsNet could offer maximum 1600 Mbps unidirectional throughput but gave less than 1500 bidirectional non-blocking throughput. This is possibly due to the insufficient CPU power of the systems tested.

Both Myrinet and QsNet had very low network latencies of less than 14 μsec MPI Roundtrip time for small message sizes. Myrinet and QsNet used the zero copy technique to improve the efficiency of network communication, which eliminates the heavy interrupts problem for traditional communication models.

High throughput communication consumed considerable system load and required sufficient system resources. Network performance would dramatically decrease when sufficient system resources were unavailable. Experiments showed that in the Alpha system, the achievable network throughput for UDP, TCP and MPI communication could shrink to less than 10Mbps in a Gigabit Ethernet environment when the system was busily handling computational jobs.

6.3 Future Work

There are several interesting areas of further work. First, currently the tracing of MAC layer statistics in Hpcbench only works on Gigabit Ethernet and TCP/IP-based networks. For those proprietary technologies that are not TCP/IP-based, such as Myrinet and QsNet,

it is possible to trace the information of the network interface card with vendor-dependent APIs. This extension can be part of our future work.

We would like to do further research to investigate why MPICH over Gigabit Ethernet performs so poorly in the Alpha system, where the maximum one-way throughput was only about 320 Mbps. Although the MPI implementations over Myrinet and QsNet were developed and well tuned by the interconnect vendors, cross-cluster parallel computing cannot work upon these tightly-coupled interconnects directly. The inter-cluster message passing for MPI applications is only available through Gigabit Ethernet in the current network architecture of SHARCNET, and many other cluster environments. In this case, the vendor-independent MPI implementation based on TCP/IP plays a critical role for distributed cluster computing. Understanding its performance deficiencies is therefore very important.

We would also like to investigate the relation between network performance and computational performance. For instance, in a Gigabit Ethernet cluster, different network interface cards can lead to different network throughput and latency. How does the computational ability of a HPC cluster change (such as the FLOP measurement) with different network behavior introduced by different NICs? It is important to conduct this kind of test so that we can have a guideline to build a commodity HPC system, although this kind of experiment can be costly to set up.

References

[1] W. Richard Stevens. *TCP/IP Illustrated Volume 1: The Protocols*. Addison-Wesley, 1994.

[2] G.R. Wright and W. Richard Stevens. *TCP/IP Illustrated Volume 2: The Implementation.* Addison-Wesley, 1995.

[3] W. Richard Stevens. *Unix Network Programming 2nd Edition Volume 1: Networking APIs.* Addison-Wesley, 1998.

[4] W. Richard Stevens. *Advanced Programming in the Unix Environment.* Addison-Wesley, 1992.

[5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel 2nd Edition.* O'Reilly, 2002.

[6] Michael Beck, et al. Linux *Kernels Internals 2nd Edition.* Addison-Wesley, 1997.

[7] Moshe Bar. *Linux Internals.* McGraw-Hill, 2000.

[8] Raj Jain. *The Art of Computer Systems Performance Analysis*, John Wiley & Sons INC, 1991.

[9] Thomas Sterling. *Beowulf Cluster Computing with Linux*. The MIT Press, 2002.

[10] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems (Volume 1)*. Prentice Hall PTR, 1999.

[11] Stan Openshaw and Ian Turton. *High-performance computing and the art of parallel programming*. Routledge, 2000.

[12] MPI Implementation and Documentation, http://www-unix.mcs.anl.gov/mpi/.

[13] MPI Forum. http://www.mpi-forum.org.

[14] The TOP500 Supercomputers list. http://www.top500.org.

[15] 10 Gigabit Ethernet Alliance. http://www.10gea.org.

[16] InfiniBand Trade Association. http://www.infinibandta.org.

[17] The PCI Special Interest Group (SIG). http://www.pcisig.com.

[18] Fabrizio Petrini et al. *The Quadrics Network (QsNet): High-Performance Clustering Technology*. IEEE Micro, January-February 2002.

[19] Quadrics' home Page. http://www.quadrics.com.

[20] Brent N. Chun et al. *Virtual Network Transport Protocols for Myrinet*. Technical report. UC Berkeley, 1998.

[21] Myricom's home page. http://www.myri.com.

[22] SHARCNET home page. http://www.sharcnet.ca.

[23] Jack J. Dongarra and Piotr Luszczek and Antoine Petitet, The LINPACK Benchmark: Past, Present, and Future, Concurrency and Computation: Practice and Experience. 2003.

[24] T. Delaitre, et al. *Publishing and Executing Parallel Legacy Code Using an OGSI Grid Service*. ICCSA (2) 2004.

[25] The Condor project. http://www.cs.wisc.edu/condor/.

[26] LSF job management system. http://www.plotform.com.

[27] Maui and Moab job management systems. http://www.supercluster.org.

[28] Portable Batch System (PBS). http://www.openpbs.org, http://www.pbspro.com.

[29] IBM SAN Redbook (Introduction to Storage Area Networks). http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg245470.html?Open

[30] HP SAN Design Reference Guide. http://h18006.www1.hp.com/products/storageworks/san/documentation.html.

[31] IBM OpenSource Distributed Lock Manager project. http://oss.software.ibm.com/dlm.

[32] PVFS file system. http://www.parl.clemson.edu/pvfs/.

[33] Redhat's GFS project. http://sources.redhat.com/cluster/gfs/.

[34] Frank Schmuck and Roger Haskin. *GPFS: A Shared-Disk File System for Large Computing Clusters.* Proceedings of the Conference on File and Storage Technologies, 28-30 January 2002.

[35] Lustre file system. http://www.clusterfs.com.

[36] Matrix server. http://www.polyserve.com/.

[37] DataPlow's SAN file system. http://www.dataplow.com/.

[38] Patrick khoo. *Ethernet Storage Trends and Overview*. Data Storage Institute. October 2002.

[39] IP storage discussion website. http://www.iscsistorage.com.

[40] Ping Guan, et al. *A Survey of Distributed File Systems*. Technical Report. University of California, 2000.

[41] T10 Technical Committee (information about I/O interfaces). http://www.t10.org.

[42] Coda file System. http://www.coda.cs.cmu.edu.

[43] AFS file system. http://www.openafs.org.

[44] Kevin Lai, Mary Baker. *Measuring Link Bandwidths Using a Deterministic Model of Packet Delay*. Proceedings of ACM SIGCOMM 2000, August 2000.

[45] Pathchar measurement tool. ftp://ftp.ee.lbl.gov/pathchar/.

[46] Pathload and Pathrate measurement tools. http://www.pathrate.org.

[47] Ntop measurement tool. http://www.ntop.org.

[48] Nettimer measurement tool. http://mosquitonet.stanford.edu/~laik/projects/nettimer/.

[49] Nuttcp tool. ftp://ftp.lcp.nrl.navy.mil/pub/nuttcp/.

[50] Udpmon measurement tool. http://www.hep.man.ac.uk/u/rich/net/.

[51] Netperf measurement tool. http://www.netperf.org.

[52] Iperf measurement tool. http://www.iperf.org.

[53] Netpipe measurement tool. http://www.scl.ameslab.gov/netpipe/.

[54] Tcpdump utility and packet capture library. http://www.tcpdump.org.

[55] High resolution timer project. http://sourceforge.net/projects/high-res-timers/.

[56] Web100 project. http://www.web100.org.

[57] Netlogger toolkit. http://www-didc.lbl.gov/NetLogger/.

[58] C.Jin, et al. *Fast TCP: From Theory to Experiments*. IEEE Communications Magazine, Internet Technology Series, April 2003.

[59] Sally Floyd. *HighSpeed TCP for large congestion windows*. Internet Draft, 2002.

[60] Piyush Shivam, et al. *Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing*, SC2001 November 2001.

[61] Christian Kurmann, et al. *Speculative Defragmentation – Leading Gigabit Ethernet to True Zero-Copy Communication*. Cluster Computing 4, 2001.

[62] Alteon Networks White Paper. *Extended Frame Sized for Next Generation Ethernets*. 1999.

[63] Matt Mathis's MTU discussion. http://www.psc.edu/~mathis/MTU/.

[64] Raj Jain. *Error Characteristics of Fiber Distributed Data Interface (FDDI)*. IEEE Transactions on Communications, No. 8, August 1990.

[65] R. Hughes-Jones, et al. *Performance Measurements on Gigabit Ethernet NICs and Server Quality Motherboards*. First International Workshop on Protocols for Fast Long-Distance Networks, February 2003.

[66] DataTAG's how to achieve Gigabit speeds with Linux. http://datatag.web.cern.ch/datatag/howto/tcp.html.

[67] Miguel Rio, et al. *A Map of the Networking Code in Linux Kernel 2.4.20*. Technical report. DataTAG, March 2004.

[68] R. Prasad, et al. *Effects of Interrupt Coalescence on Network Measurements*. Proceedings of Passive and Active Measurement (PAM) Workshop, April 2004.

[69] Vern Paxson. *Measurements and Analysis of End-to-End Internet Dynamics.* Ph.D. Thesis. UC Berkeley, April 1997.

[70] Roelof Jonkman. *NetSpec: Philosophy, Design and Implementation*. Master Thesis. University of Kansas. 1998.

[71] John Enok Vollestad. *A high performance cluster file system using SCI*. Master Thesis. University of Oslo, Autumn 2002.

[72] Jon Postel. *User Datagram Protocol*. RFC 768, 1980.

[73] Jon Postel. *Internet Protocol*. RFC 791, 1981.

[74] Jon Postel. *Transmission control protocol – protocol specification*. RFC 793, 1981.

[75] John Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896, 1984.

[76] Van Jacobson, et al. *TCP Extensions for High Performance*. RFC 1323, 1992.

[77] Philip Almquist. *Type of Service in the Internet Protocol Suite*. RFC 1349, 1992.

[78] Matt Mathis, et al. *TCP Selective Acknowledgment Options*. RFC 2018, 1996.

[79] W. Richard Stevens. *TCP slow start, congestion avoidance, fast retransmit and fast recovery algorithms*. RFC 2001, 1997.

[80] Kathleen Nichols, et al. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474, 1998.

[81] Mark Allman, et al. TCP Congestion Control. RFC 2581, 1999.

[82] Sally Floyd, et al. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 2582, 1999.

[83] Sally Floyd, et al. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. RFC 2883, 2000.

[84] L.S. Brakmo and L.L. Peterson. *TCP Vegas: End to End Congestion Avoidance on a Global Internet*. IEEE Journal on Selected Areas in Communications, Vol. 13, No. 8, October 1995.

[85] Sally Floyd, et al. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168, 2001.

[86] Sally Floyd. *HighSpeed TCP for Large Congestion Windows*. RFC 3649, 2003.

Appendix A. Hpcbench Benchmark

A.1 Overview

Hpcbench is a Linux-based network benchmark evaluating the high performance networks such as Gigabit Ethernet, Myrinet and QsNet. Hpcbench measures the network latency and achievable throughput between two ends. Hpcbench is able to log the kernel information for each test, which includes the CPU and memory usage, interrupts, swapping, paging, context switches, network cards' statistics, etc.

Hpcbench consists of three independent packages that test UDP, TCP and MPI communications respectively. A kernel resources tracing tool "sysmon" is also included, whose output is similar to that of vmstat, but has more information of network statistics.

Hpcbench comprises about 8000-line C and MPI code, optimized for Linux systems. Hpcbench's official website is: http://hpcbench.sourceforge.net, where its source code and some test examples are freely available.

Figure A-1 shows the communication architecture of Hpcbench.

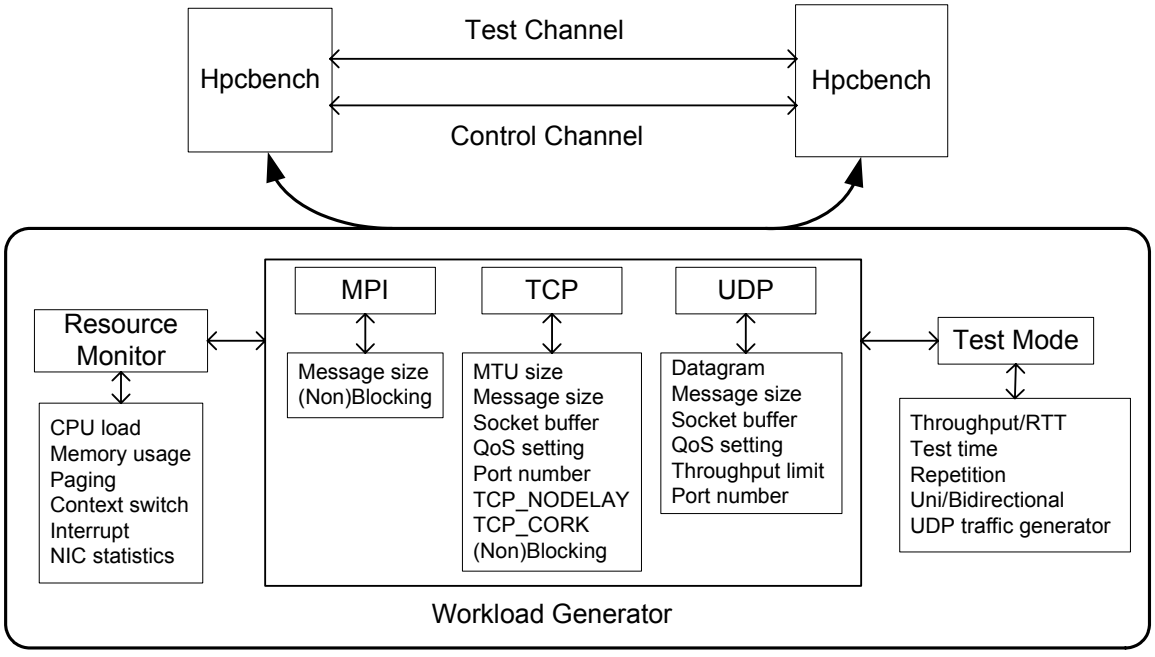

Figure A-1 Communication model of Hpcbench

A.2 Features

A.2.1 UDP Communication Test:

- Microsecond resolution
- Roundtrip time test (UDP ping)
- Throughput test
- Unidirectional and Bidirectional test
- UDP traffic generator (can run in single mode)
- Fixed size and exponential test
- Log throughputs and process resource usage of each test
- Log system resources information of client and server (Linux only)
- Create plot configuration file for gnuplot
- Configurable message size
- Other tunable parameters:
    - Port number
    - Client and server's UDP socket buffer size
    - Message size
    - Packet (datagram) size
    - Data size of each read/write
    - QoS (TOS) type (Pre-defined six levels)
    - Test time
    - Test repetition
    - Maximum throughput restriction (Unidirectional and UDP traffic generator)

A.2.2 TCP Communication Test:

- Microsecond resolution
- Roundtrip Time test (TCP ping)
- Throughput test
- Unidirectional and Bidirectional test
- Blocking and non-blocking test
- Fixed size and exponential test
- Linux sendfile() test
- Log throughputs and process resource usage of each test
- Log system resources information of client and server (Linux only)
- Create plot configuration file for gnuplot
- Configurable message size
- Other tunable parameters:
    - Port number
    - Client and server's TCP socket buffer (window) size
    - Message size
    - Data size of each read/write
    - Iteration of read/write
    - MTU (MSS) setting
    - TCP socket's TCP_NODELAY option setting
    - TCP socket's TCP_CORK option setting
    - QoS (TOS) type (Pre-defined six levels)
    - Test time
    - Test repetition

A.2.3 MPI communication Test:

- Microsecond resolution
- Roundtrip Time test (MPI ping)
- Throughput test
- Unidirectional and Bidirectional test
- Blocking and non-blocking test
- Fixed size and exponential test
- Log throughputs and process resource usage of each test
- Log system resources information of two processes (nodes) (Linux only)
- Create plot configuration file for gnuplot
- Tunable parameters:
    - Message size
    - Test time
    - Test repetition

A.3 Hpcbench Usage and Options

Hpcbench includes four packages. Each of them can work independently. UDP and TCP benchmarks are used in pairs, and you should start the server process before the client process. MPI benchmark must work with a MPI implementation. All these three tools measure end-to-end performance in a network. Sysmon is a Linux-based system resource monitoring tool, functioning like vmstat/iostat with more information about the network statistics. The directory structure of Hpcbench:

```
hpcbench/  --> Readme and Makefile
hpcbench/udp/ --> UDP measurement tool
hpcbench/tcp/ --> TCP measurement tool
hpcbench/mpi/ --> MPI measurement tool
hpcbench/sys/ --> Linux system resource monitoring tool
hpcbench/testscript/ --> Scripts to scan a set of parameters for
UDP/TCP/MPI communication
```

A.3.1 UDP Communication Measurement

Two executables, udpserver and udptest, will be created in the directory after compilation. You should start the server process at first, except udptest(client) runs as UDP traffic generator. The command line options:

UDP server usage: udpserver [options]

$ udpserver [-v] [-p port]

- [-p port] Port number for TCP listening (0 picked by system), 5678 by default.
- [-v] Verbose mode. Disable by default.

UDP client usage: udptest -h host [options]

$ udptest -h host [-vacdeiP] [-p port] [-A rtt] [-b buffer] [-B buffer] [-m msssage] [-q qos] [-l datagram] [-d data] [-t time] [-r repeat] [-o output] [-T throughput]

- [-a] UDP Round Trip Time (RTT or latency) test.
- [-A rtt-size] UDP RTT (latency) test with specified message size.
- [-b buffer] Client UDP buffer size in bytes. Using system default value if not defined.
- [-B buffer] Server UDP buffer size in bytes. The same as cleint's by default.
- [-c] CPU log option. Tracing system info during the test. Only available when output is defined.
- [-d data-size] Data size of each read/write in bytes. The same as packet size by default.
- [-e] Exponential test (data size of each sending increasing from 1 byte to packet size).
- [-g] UDP traffic generator (Keep sending data to a host). Work without server's support.
- [-h host] Hostname or IP address of UDP server. Must be specified.
- [-i] Bidirectional UDP throughput test. Default is unidirection stream test.
- [-l datagram] UDP datagram (packet) size in bytes (< udp-buffer-szie). 1460 by default.
- [-m message] Total message size in bytes. 1Mbytes by default.
- [-o output] Output file name.
- [-p port] Port number of UDP server. 5678 by default.
- [-P] Write the plot file for gnuplot. Only enable when the output is specified.
- [-q qos] Define the TOS field of IP packets. Six predefined values can be used for this setting:
    - 1: (IPTOS)-Minimize delay 2: (IPTOS)-Maximize throughput
    - 3: (DiffServ)-Class1 with low drop probability 4: (DiffServ)-class1 with high drop probability
    - 5: (DiffServ)-Class4 with low drop probabiltiy 6: (DiffServ)-Class4 with high drop probabiltiy
- [-r repeat] Repetition of tests. 10 by default.
- [-t time] Test time constraint in seconds. 5 by default.
- [-T throughput] Throughput constraint for UDP generator or throughput test. Unlimited by default.
- [-v] Verbose mode. Disable by default.

NOTE: Input (except -T) supports the postfix of "kKmM", 1K=1024, 1M=1024x1024. Throughput constraint option (-T): 1K=1000, 1M=1000000.

With plot option (-P), when an "output" file is specified, an "output.plot" file will also be created for plotting. Use "gnuplot ouput.plot" to plot the data. With CPU option (-c), when an "output" file is specified, "output.c_log" and "output.s_log" files store the system information of client and server, respectively.

UDP Round Trip Time (latency) test is just a UDP version of "ping". RTT is too short to be measured in HPC environments, so we repeat RTT test many times and get the average of RTTs.

A UPD throughput test is done when both of the conditions are satisfied: message size AND test time. So the actual size of sent message could be greater than the message size you specify if the test time is large.

In UPD throughput tests, message size (-m option) specifies the total amount of data to be sent. Messages are actually sent by small pieces (defined by -d option) that

must be smaller than datagram (packet) size. In exponential tests, the sending size increases exponentially from 1 byte to the datagram (packet) size; while in the fixed-size tests, the size of each sending is always the same as datagram (packet) size. Most systems have a 64KB maximum size limit of UDP datagram (packet).

UDP traffic generator keeps sending UDP packets to a remote host that is unnecessary running as server. Better to pick an unused port for this test. You can specify the throughput to be sent (-T option). Be aware that this test may affect target host's performance.

If CPU and system monitoring option (-c) is defined, both client and server's CPU and memory usages (Maximum 8 CPUs supported for SMP systems), network interface statistics and its interrupts to each CPU will be recorded. Currently this option is only available for Linux system.

Examples

1. Start server process
[server] $ udpserver

2. Start client process

Example 1: [client] $ udptest -ah server
UDP Round Trip Time (latency) test.

Example 2: [client] $ udptest -h server
UDP throughput test with default set of parameters (Port: 5678, test-time: 5, test-repeat: 10, Message-size: 1Mbytes, packet-size: 1460, send-size: 1460)

Example 3: [server] $ udpserver -p 3000
[client] $ udptest -vh server -p 3000 -b 1M -m 10m -l 20k -t 2 -r 20 -o output.txt
Repeat throughput tests by 20 times with communication port of 3000; store results in "output.txt"; buffer-size: 1MB, message-size: 10MB, test-time: 2 Seconds, packet-size: 20KB

Example 4: [client] $ udptest -eP -h server -b 100k -o output.txt
Exponential throughput test for buffer size of 100KB, writing output and plot file.

Example 5: [client] $ udptest -gh abc.com -T 10M -t 30
Keep sending UDP data to the target host by 10Mbps throughput for 30 seconds.

A.3.2 TCP Communication Measurement

Two executables, tcpserver and tcptest, will be created in the directory after compiplation. You should start the server process at first. The command line options:

TCP server usage: tcpserver [options]

$ tcpserver [-v] [-p port]

- [-p port] Port number for TCP listening (0 picked by system), 5677 by default.
- [-v] Verbose mode. Disable by default.

TCP clinet usage: tcptest -h host [options]

$ tcptest -h host [-vanicCNP] [-p port] [-A rtt-size] [-e exponent] [-b buffer] [-B buffer] [-q qos] [-M MSS] [-d data] [-m message] [-r repeat] [-t time] [-f sendfile] [-I iteration] [-o output]

- [-a] Test the TCP Round Trip Time (RTT). Ignore all other options if defined.
- [-A test-size] TCP RTT test with specified message size.
- [-b buffer-size] TCP buffer (windows) size in bytes. System default if not defined.
- [-B buffer] Server UDP buffer size in bytes. The same as cleint's by default.
- [-c] CPU log option. Tracing system information during the test. Only availabe when output is defined.
- [-C] Turn on socket's TCP_CORK option (avoid sending partial frames). Disable by default.
- [-d data-size] Data size of each read/write in bytes. The same as packet size by default.
- [-e n] Exponential tests with message size increasing exponentially from 1 to 2^n.
- [-f sendfile] Sendfile test. Memory mapping is used to reduce the workload. Disable by default.
- [-h host-name] Hostname or IP address of server. Must be specified.
- [-i] Bidirectional UDP throuhghput test. Default is unidirection stream test.
- [-I iteration] Iteration of sending/receiving for each test. Auto-determined by default.
- [-m message-size] Message size in bytes. 65536 by default.
- [-M MSS-size] Maximum Segent Size in bytes (MTU-40 for TCP). System default if not defined.
- [-n] Non-blocking communication. Blocking communication by default.
- [-N] Turn on socket's TCP_NODELAY option (disable Nagel algorithm). Disable by default.
- [-o output] Output file name.
- [-p port-number] Server's port number. 5677 by default.
- [-P] Write the plot file for gnuplot. Only enable when the output is specified.
- [-q qos] Define the TOS field of IP packets. Six predefined values can be used for this setting:
    - 1: (IPTOS)-Minimize delay 2: (IPTOS)-Maximize throughput
    - 3: (DiffServ)-Class1 with low drop probability 4: (DiffServ)-class1 with high drop probability
    - 5: (DiffServ)-Class4 with low drop probabiltiy 6: (DiffServ)-Class4 with high drop probabiltiy
    - [-r repeat] Repetition of tests. 10 by default.
- [-t test-time] Test time in seconds. Disable if iteration is sepcified. 5 by default.
- [-v] Verbose mode. Disable by default.

NOTE: Input supports the postfix of "kKmM", 1k=1024, 1M=1024x1024.

With plot option (-P), when an "output" file is specified, an "output.plot" file will also be created for plotting. Use "gnuplot ouput.plot" to plot the data. With CPU option (-c), when an "output" file is specified, "output.c_log" and "output.s_log" files store the system information of client and server, respectively.

The TCP RTT (latency) test is just a TCP version of "ping". RTT is too short to be measured in HPC environments, so we repeat RTT test many times and get the average of RTTs. In the TCP tests, message size (-m option) specifies the amount of data to be sent each time.

The iteration of sending/receiving for a test time (-t option) is determined by an evaluation test, so the actually test time could vary slightly. In exponential test, message size increases exponentially from 1 byte to a large number (-e option). Be aware that there is a minimum number of iteration, and the test time might be much greater than what you specify if the message size is very large.

If CPU and system monitoring option (-c) is defined, both client and server's CPU and memory usages (Maximum 8 CPUs supported for SMP systems), network interface statistics and its interrupts to each CPU will be recorded. Currently this option is only available for Linux system.

Examples

1. Start server process
[server] $ tcpserver

2. Start client process

Example 1: [client] $ tcptest -ah server
TCP Round Trip Time (RTT) test. A TCP version of ping.

Example 2: [client] $ tcptest -h server
TCP blocking stream test with default set of parameters, verbose off, no result writing.

Example 3: [server] $ tcpserver -p 3000
[client] $ tcptest -vn -h server -p 3000 -b 100k -m 10m -t 2 -r 20 -o output.txt
Repeat non-blocking stream tests by 20 times with communication port of 3000. Buffer size: 100K, message size: 10M, test time: 2 Seconds, store results in "output.txt".

Example 4: [client] $ tcptest -e 20 -vh server -b 100k -o output.txt
Exponential stream test for buffer size of 100 KB with verbose mode with message size increasing exponentially from 1 Byte to 1 MByte (2^20).

A.3.3 MPI Communication Measurement

To compile the MPI benchmark (mpi directory), you need a MPI implementation installed, such as MPICH or LAM/MPI. And you should define the MPI compiler in the makefile. Most MPI implementations have a script named mpicc to do this job. An executable file mpitest will be created when compilation is finished. To run the

mpitest, you should run the program with another script named mpirun, or submit the job to a high level queuing systems like RMS and LSF. For MPICH, the command line options of mpitest:

MPI test: mpirun -np 2 mpitest [options]

$ mpirun -np 2 mpitest [-acinP] [-A size] [-e exponent] [-m message] [-o output] [-r repeat] [-t time]

- [-a] Round Trip Time (latency) test. Disable by default.
- [-A RTT-size] Specify the message size in bytes for RTT (latency) test.
- [-c] CPU log option. Tracing system information during the test. Only available for Linux systems.
- [-e n] Exponential tests with message size increasing exponentially from 1 to 2^n. Disable by default.
- [-i] Ping-pong (bidirectional) test. Stream (unidirectional) test by default.
- [-m message-size] Message size by bytes (1M by default). Disable in exponential tests.
- [-n] Non-blocking communication. Blocking communication by default.
- [-o output] Write test results to a file. Disable by default.
- [-P] Plot file for gnuplot. Only enable when the output is specified. Disable by default.
- [-r repeat] Repeat tests many times. Disable in exponential tests. 10 times by default.
- [-t test-time] Specify test time by seconds. 5 seconds by default.

NOTE: Input supports the postfix of "kKmM". 1k=1024, 1M=1024x1024.

Examples:

Example1: $ mpirun -np 2 mpitest
Throughput stream test with default parameters.

Example2: $ mpirun -np 2 mpitest -e 20
Exponential stream (unidirectional) test, message size from 1 byte to 2^20 (1M) bytes.

Example3: $ mpirun -np 2 mpitest -c -m 10m -Po output.txt
Throughput stream test with 10MBytes message size, write result/plot files, log system info.

Example4: $ mpirun -np 2 mpitest -ni -m 100k -t 3 -r 10
Nonblocking ping-pong test. Message-size: 100KBytes; test-ime: 3 seconds; repeat 10 times.

Example5: $ mpirun -np 2 mpitest -a -o rtt.txt
MPI Round Trip Time (latency) test. Write the result to file "rtt.txt".

To use own machine file (MPICH):

$ mpirun -np 2 -machinefile <machine file> mpitest [options]

or use the p4 procgroup file (MPICH):

$ mpirun -p4pg <p4 procgroup file> mpitest [options]

There are sample machinefile and p4pg files in mpi directory. To submit your job to queuing system such as LSF, refer to the test-mpi.lsf and test-mpi.sh scripts in testscript directory.

A.3.4 SYSMON – Linux System Resource Monitor

Sysmon is a lightweight Linux-based system resource tracing tool. Although it's not a benchmark, it's very helpful to trace what is happening in the kernel level during the benchmarking. The output includes the CPU and memory usage, swapping, paging and context switches information, interrupts kernel received, and each network interface's statistics, which includes interrupts to kernel, packets and bytes that received and sent in a specified interval.

Sysmon usage: sysmon [options]

$ sysmon [-bhkwW] [-i interface-name] [-r repeat] [-t test-interval] [-T test-time]

- [-b] Background (daemon) mode. Only valid when write option is defined.
- [-h] Printout this help messages.
- [-k] Kill the sysmon background process (daemon). Disable by default.
- [-w] Write all results to a file. Disable by default.
- [-W] Write statistics of each network device to separate files. Disable by default.
- [-i interface-name] Define the network device name (e.g. eth0). Monitor all if no interface defined.
- [-r repeat] Repetition of monitoring. 10 times by default.
- [-t test-interval] The interval (sample time) between each tracing in seconds. 2 seconds by default.
- [-T test-time] The duration of system monitoring in minutes. Valid only write option defined.
- [-o output] Specify the output (log) filename. Implies the write option.

NOTE: Default log file has format of hostname-start-time.log if write option (-w) is defined and output option (-o) is not defined. If separate write option (-W) is defined, besides the overall log file "output", each network interface has its itself log file with name like "output.eth0". This smaller log file is more readable than the lengthy overall log file.

You can use command "/sbin/ifconfig" to check the network devices and their names in your computer. Possible names: eth0, wlan0, elan0, etc. If your system has some strange NIC names, you can define them with constant NETNAME in util.h. We use the name of "loop" for loopback address.

Examples:

Example1: $ sysmon
Monitor all network devices. Output has a very long format if the computer has several network cards.

Example2: $ sysmon -r 100 -t 1 -i eth0 -o net.log
Only monitor the first Ethernet card, repeat test 100 times with time interval of 1 second, write results to net.log

Example3: $ sysmon -bw -i eth0 -t 600 -T 10080
Log every 10 minutes for one week, run process in background (daemon).

Appendix B. Network Statistics of Clusters

We used the *sysmon* utility of Hpcbench to trace the network statistics over Gigabit Ethernet in hammerhead and deeppurple for a whole week (Auguest 4th to 11th, 2004). The statistics of hammerhead is shown in Table B-1. The results show that node hh24 had much more data transfer than others during the one week monitoring. We traced the log files and found there was a steep jump of data transmission in case of insufficient memory. When the memory usage was greater than 95% and swapping occurred, the amount of data transferred (in and out) during the period was significantly larger than normal cases. It's possible that the data swapped into or out of NFS server when the local virtual memory was inadequate for some applications consuming lots of memory such as Gaussian.

Network statistics of hammerhead from Aug. 4th-11th. hh1-0 was the Internet link and hh1-1 was connected to local cluster. Sample rate: 10 Minutes.

Node	CPU load (%)			Received Data				Sent Data			
	Avg.	Min	Max	Total (GB)	Avg. (Kbps)	Min (Kbps)	Max (Mbps)	Total (GB)	Avg. (Kbps)	Min (Kbps)	Max (Mbps)
hh1-0	1.52	0.70	25.6	695	9.64	0.75	0.71	9095	126.15	0.01	1.09
hh1-1	1.52	0.70	25.6	42648	591.54	45.84	59.14	35443	491	52.08	58.92
hh2	N/A										
hh3	65.6	0.10	100	402758	5586.28	5.57	89.65	89266	1238.13	4.17	31.83
hh4	58.3	0.10	100	149064	2067.54	4.52	90.20	17242	239.15	3.59	9.73
hh5	84.7	0	100	2965	41.13	2.83	5.82	15638	216.90	2.16	9.70
hh6	59.5	0.1	98.3	2583	35.20	3.38	18.13	8954	124.20	2.47	19.49
hh7	84.6	25.0	100	855	11.87	2.87	1.03	2892	40.12	2.12	1.72
hh8	55.5	0	100	960	13.32	1.43	1.03	7852	108.92	0.77	2.02
hh9	75.4	0	100	5535	76.77	1.63	2.86	5906	81.93	1.34	2.38
hh10	59.2	0.1	100	16705	231.71	4.08	4.29	12313	170.79	2.97	11.60
hh11	N/A										
hh12	85.0	0	100	20008	277.52	3.94	6.46	16080	223.04	3.21	1.46
hh13	72.0	0	98.0	2103	29.18	1.67	4.25	2714	37.65	1.37	10.98
hh14	80.7	0	100	1493	20.71	1.67	9.09	2334	32.38	1.38	0.88
hh15	77.2	0	100	165542	2296.09	1.68	88.21	27903	387.02	1.42	6.59
hh16	65.4	0	99.7	63009	873.94	1.34	86.31	15474	214.64	0.72	5.31
hh17	63.6	0	100	5789	80.30	3.55	15.15	11412	158.29	2.55	32.40
hh18	66.5	0	100	96324	1336.03	2.83	87.82	11036	153.08	2.10	5.82
hh19	71.8	0	100	177108	2456.50	1.36	92.61	26394	366.10	0.77	5.86
hh20	76.4	0	100	163673	2270.16	1.42	85.54	11203	155.39	0.75	6.21
hh21	81.7	0	100	4760	66.03	2.04	1.12	2647	36.72	1.86	1.37
hh22	61.8	0	99.9	1190	16.51	2.76	0.74	21479	297.92	1.85	1.31
hh23	37.8	0	99.8	158357	2196.42	3.11	87.14	14334	198.82	2.45	6.40

hh24	32.3	0	99.9	648506	8994.82	2.87	122.03	609426	8452.78	2.12	42.63
hh25	83.5	0	100	474	6.59	1.63	0.55	513	7.12	1.36	0.21
hh26	80.2	0	100	342	4.75	1.64	0.60	304.	4.23	1.36	0.42
hh27	N/A										
hh28	0	0	1.7	238	3.30	1.22	0.70	148	2.06	0.61	0.03

Table B-1 Network Statistics of Hammerhead (Augest 4th – Augest 11th, 2004)

Mainly there were three kinds of traffic in the Gigabit Ethernet network: disk I/O between processes and NFS server (all users' home directory was mounted by NFS), management traffic by LFS, RMS, etc. and user access whose data traffic mainly went to the master node. Table B-2 and B-3 show the network statistics of deeppurple in one week (August 4th - August 11th) and one day (August 8th). Since there was no user logging into the deeppurple on Auguest 8th (Sunday), the data traffic was due to parallel computing and communication of cluster controlling. Figure B1-B12 show the plotting of statistics of all nodes in deeppurle. We can see all compute nodes were extremely busy (100% CPU load) in the whole week. This would lead to a very low throughput from our earlier analysis.

The burst of curve in the figures might be the result of the launch of some processes that copied a big amount of data from disk (NFS) into local memory, or the termination of some processes that copied results from memory into disk (NFS). For example, dp4 might probably started a parallel job as a master node around 4:00 AM on August 8th (outgoing data were greater than incoming data since then), and dp2 and dp3 might probably started a job as slave nodes (outgoing data were less than incoming data since then). Because the data passing (in/out) in dp2 is roughly the double of dp3's, we may assume the running processes in dp2 were double of dp3's. The average data flow in the day was less than 10Kbps for all compute nodes, showing that the jobs didn't need a lot of data exchanging, nor did work closely with NFS file system.

Network statistics of deeppurple from Aug. 4th – 11th. dp1-0 was the Internet link and dp1-1 was connected to local cluster. Sample rate: 5 Minutes.											
Node	CPU load (%)			Received Data				Sent Data			
	Avg.	Min	Max	Total (GB)	Avg. (Kbps)	Min (Kbps)	Max (Mbps)	Total (GB)	Avg. (Kbps)	Min (Kbps)	Max (Mbps)
dp1-0	1.4	0.8	20.7	474	6.58	1.40	0.09	384	4.83	0	1.51
dp1-1	1.4	0.8	20.7	3182	44.15	23.37	2.46	3008	41.73	23.40	2.50
dp2	99.8	46.8	100	418	5.80	3.54	1.52	443	6.15	2.62	0.68
dp3	99.8	0	100	360	5.00	2.18	1.52	2804	38.90	1.79	39.78

dp4	99.7	0	100	3246	45.03	2.09	2.48	142924	1982.38	1.70	119.17
dp5	99.8	0	100	830	11.52	3.52	1.51	10902	151.22	2.62	3.70
dp6	99.9	0	100	516	7.16	5.94	0.73	739	10.25	5.36	4.03
dp7	99.9	64.8	100	304	4.22	2.12	1.25	2135	29.62	1.76	6.89
dp8	99.9	65.6	100	755	10.48	3.57	1.52	8536	118.39	2.64	10.75
dp9	99.9	0	100	291	4.04	2.10	0.77	3101	43.01	1.69	4.43
dp10	99.9	63.0	100	593	8.23	2.11	1.52	7975	110.62	1.75	5.20
dp11	100	83.2	100	402	5.58	2.80	1.51	2775	38.49	2.13	6.24
dp12	99.9	63.3	100	594	8.25	2.10	1.50	9904	137.38	1.69	2.59

Table B-2 Network Statistics of Deeppurple (Augest 4th – Augest 11th, 2004)

Network statistics of deeppurple on Aug. 8th. Dp1-0 was the Internet link and dp1-1 was connected to local cluster. Sample rate: 5 Minutes.											
Node	CPU load (%)			Received Data				Sent Data			
	Avg.	Min	Max	Total (GB)	Avg. (Kbps)	Min (Kbps)	Max (Kbps)	Total (GB)	Avg. (Kbps)	Min (Kbps)	Max (Kbps)
dp1-0	1.45	1.40	6.20	61	6.03	5.31	14.07	20	2.03	1.41	43.36
dp1-1	1.45	1.40	6.20	377	36.79	24.64	1940.79	320	31.21	24.89	145.56
dp2	100	100	100	41	4.02	3.91	5.69	31	3.03	2.92	5.02
dp3	100	100	100	23	2.28	2.18	3.94	20	1.94	1.84	3.93
dp4	99.6	0	100	24	2.42	2.18	14.01	34	3.37	1.83	7.99
dp5	100	99.1	100	42	4.09	3.91	13.49	31	3.09	2.90	11.25
dp6	100	100	100	64	6.28	5.95	8.22	62	6.11	5.38	10.29
dp7	100	100	100	23	2.30	2.12	8.36	20	1.95	1.77	3.95
dp8	100	100	100	41	4.03	3.79	5.92	31	3.04	2.84	5.26
dp9	100	100	100	23	2.32	2.19	3.95	27	2.65	1.84	5.43
dp10	100	99.7	100	23	2.31	2.18	10.75	20	1.97	1.82	8.47
dp11	100	100	100	32	3.16	2.82	4.80	25	2.51	2.14	4.47
dp12	100	100	100	23	2.30	2.18	8.49	20	1.97	1.76	4.14

Table B-3 Network Statistics of Deeppurple (Augest 8th, 2004)

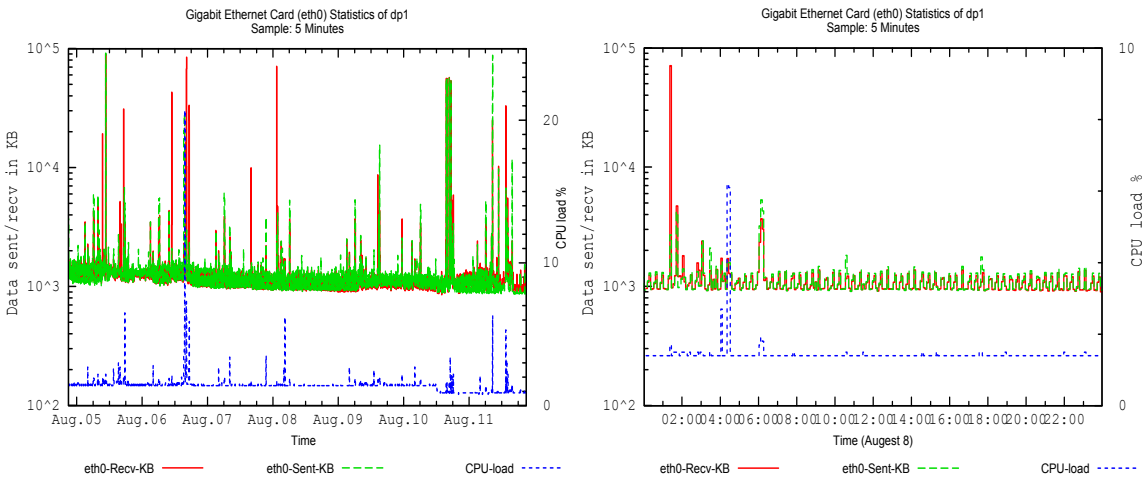

Figure B-1 Network Statistics of dp1

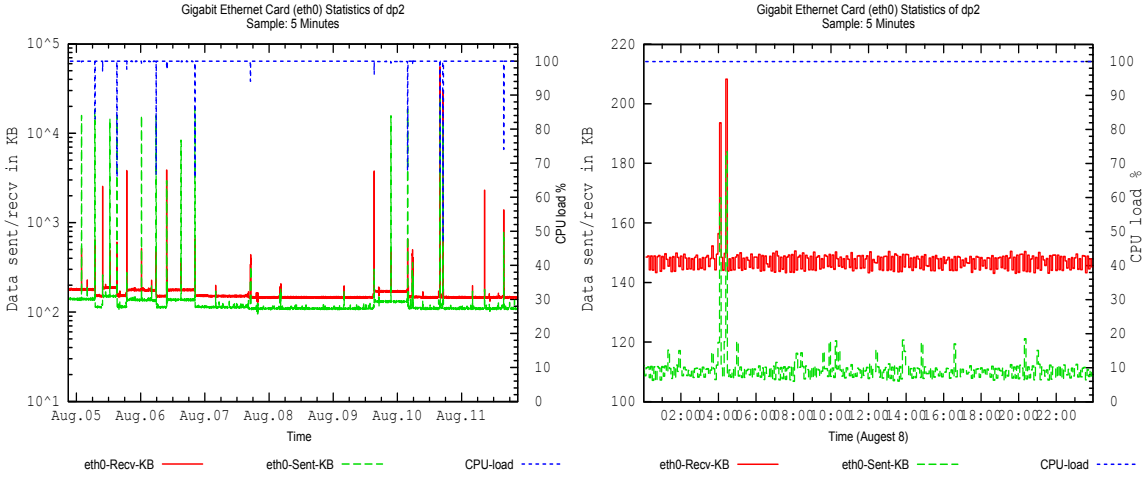

Figure B-2 Network Statistics of dp2

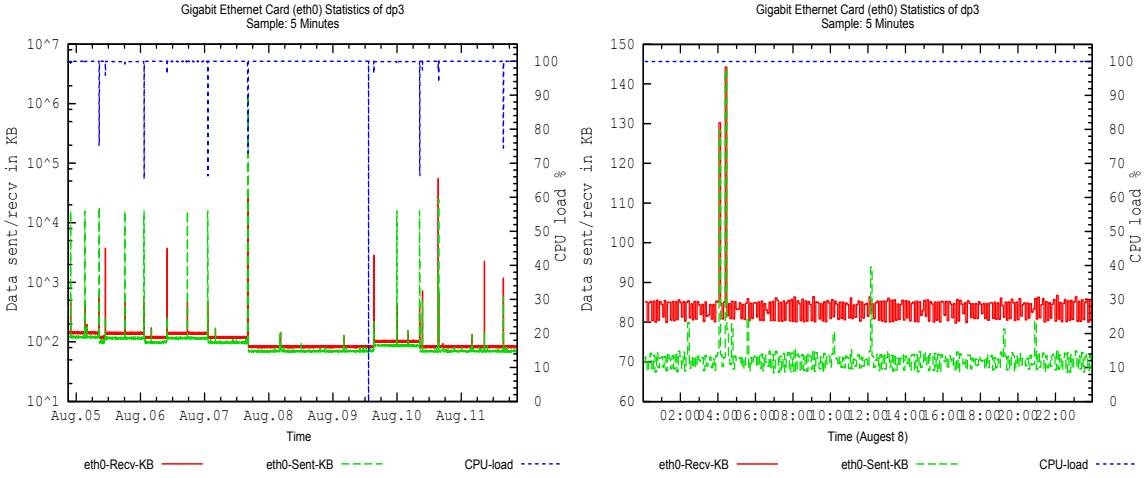

Figure B-3 Network Statistics of dp3

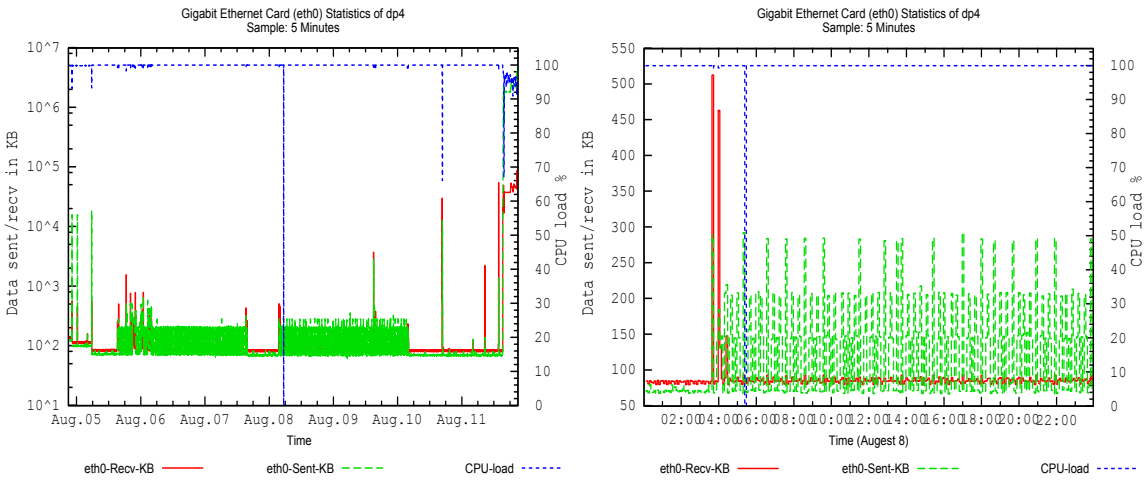

Figure B-4 Network Statistics of dp4

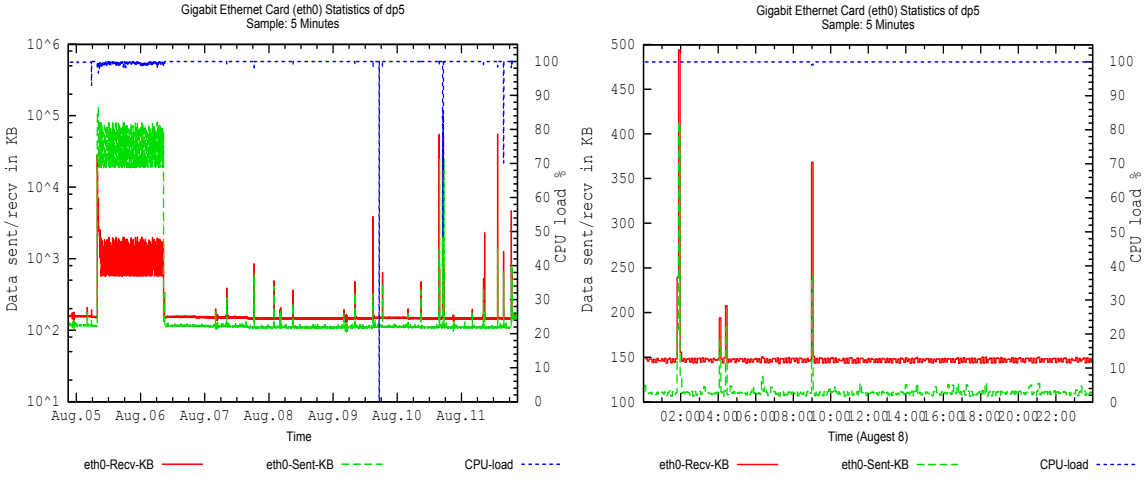

Figure B-5 Network Statistics of dp5

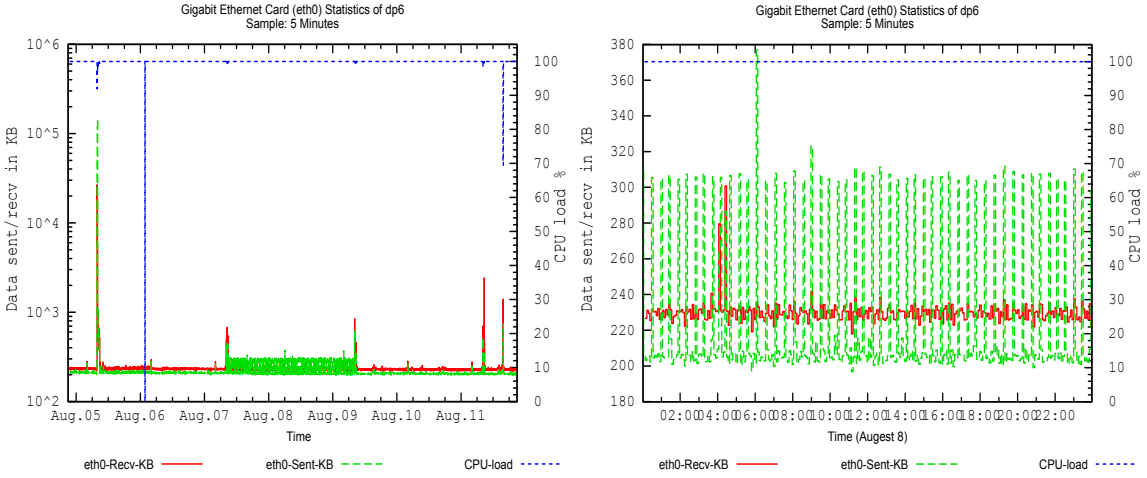

Figure B-6 Network Statistics of dp6

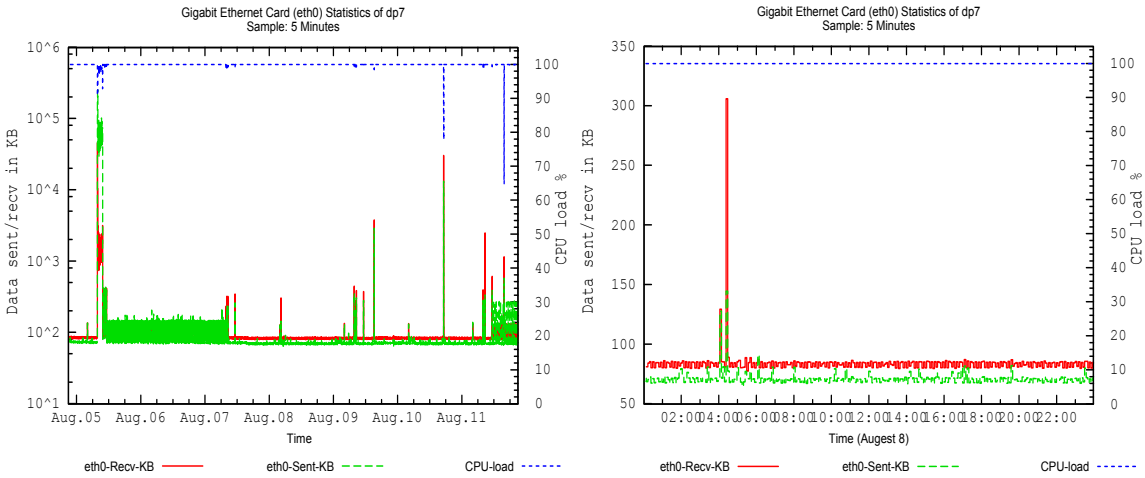

Figure B-7 Network Statistics of dp7

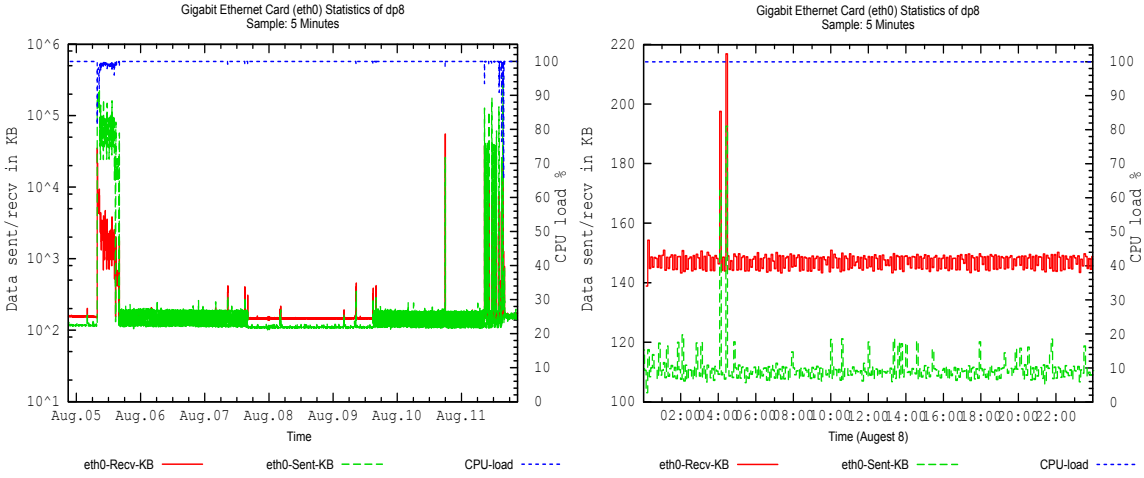

Figure B-8 Network Statistics of dp8

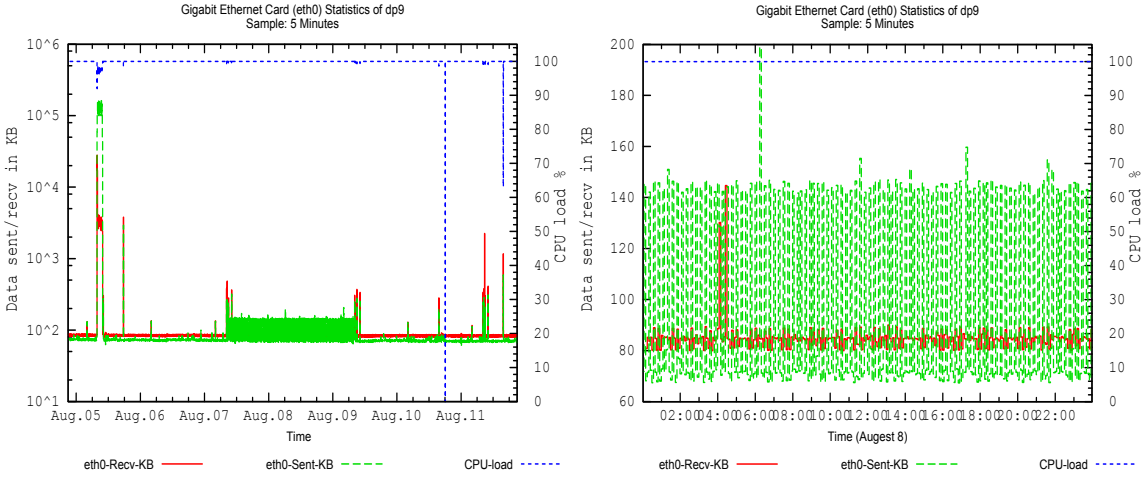

Figure B-9 Network Statistics of dp9

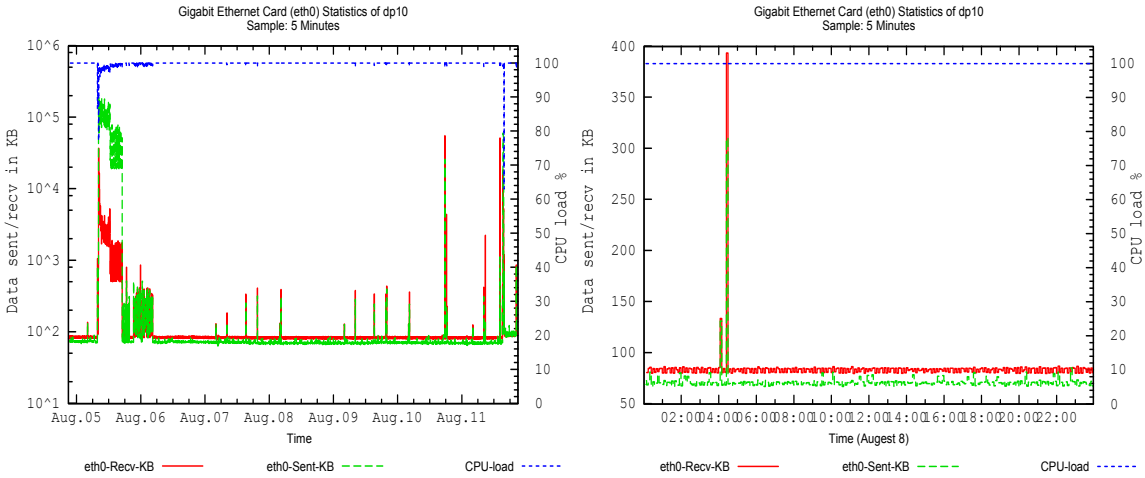

Figure B-10 Network Statistics of dp10

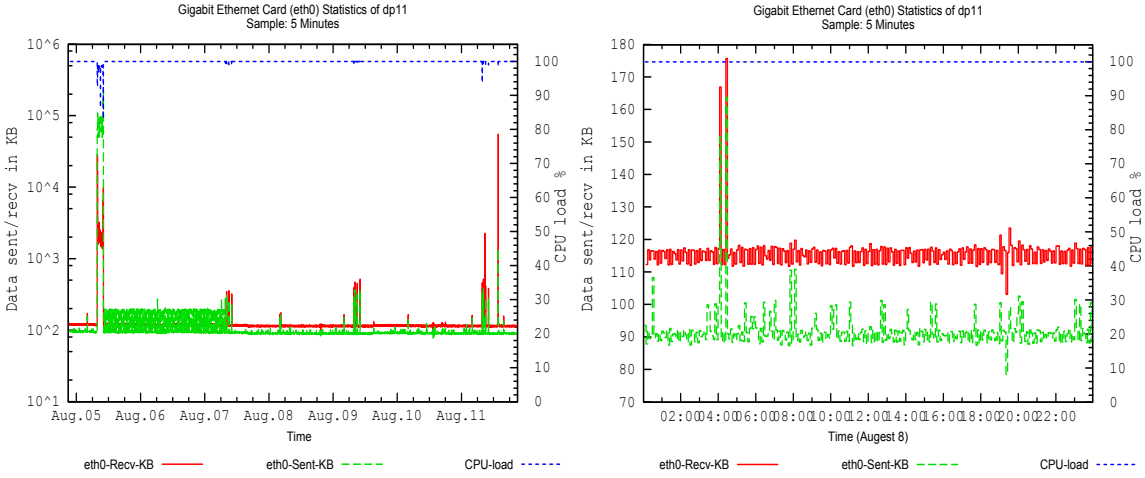

Figure B-11 Network Statistics of dp11

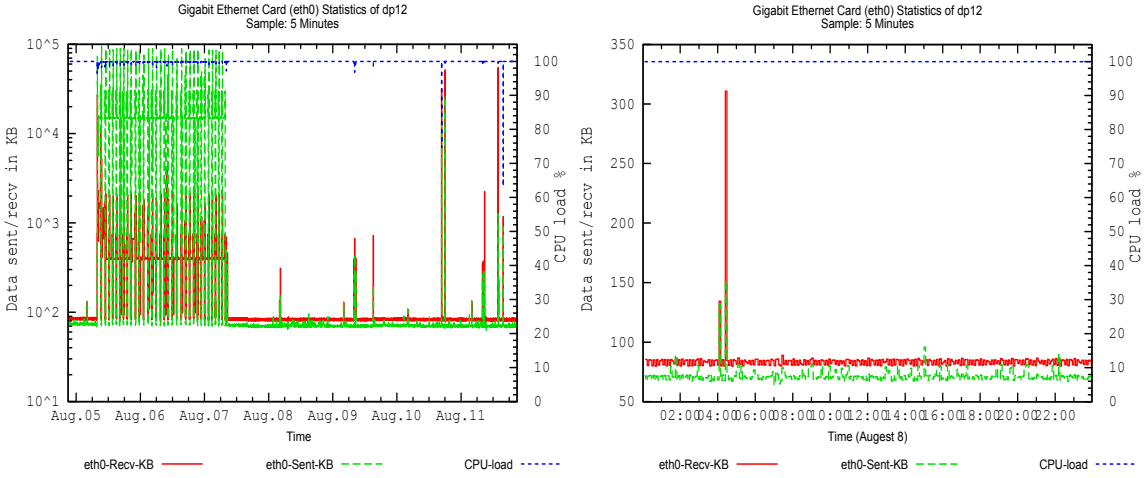

Figure B-12 Network Statistics of dp12