Directed Point: A Communication Subsystem for Commodity Supercomputing with Gigabit Ethernet

Cho-Li Wang*, Anthony T.C. Tam, Benny W.L. Cheung, Wenzhang Zhu, David C.M. Lee

Department of Computer Science and Information Systems. The University of Hong Kong Pokfulam, Hong Kong.

In this paper, we study the practical issues on the design of a new communication subsystem, named Directed Point, on a server cluster with Gigabit Ethernet connection, with the goals of achieving high performance and good programmability. Our design exploits the gigabit network architecture and the operating system characteristics. We propose a realistic communication model which can be used to assess various design tradeoffs and to calibrate the performance results. Testing shows that Directed Point communication subsystem can achieve a 16.3 μs single-trip latency and 79.5 MB/s bandwidth. To achieve good programmability, we proposed an abstraction model that allows all inter-process communication patterns to be easily coded using the provided API. The API preserves the syntax and semantics of traditional UNIX I/O operations, making the proposed communication subsystem easy to use without long learning period.

Keywords: Gigabit Ethernet, Directed Pointed, low-latency communication, communication model, cluster

1. Introduction

Commodity supercomputing is one of the targets in building clusters. Being one form of message passing machines, the performance of clusters depends largely on the performance of the interconnection network and the communication software. Recently, the microprocessor clock speeds have approached gigahertz range. Such explosive growth of processor performance has greatly improved the computation. Nevertheless, it stresses the need of a higher speed communication subsystem that can achieve low-overhead calls to access the interconnect, as the performance of a parallel application depends largely on the performance of the interconnection network and the communication software.

Gigabit Ethernet is widely available and is becoming the commodity of the next generation LAN [3]. Gigabit Ethernet appears to be an ideal solution to the increasing demands placed on today's high-end server that operates at gigahertz clock rate. However, installing a Gigabit Ethernet adapter in an existing server generally won't yield the 10-fold

*This research was supported by Hong Kong RGC Grant HKU 10201701 and HKU University Research Grants 1023009.

performance boost over a fast Ethernet adapter. To achieve high performance, the communication software needs to minimize the protocol-processing overheads and resource consumption.

With the introduction of low-latency messaging systems, such as Active Messages (AM) [24], Fast Messages (FM) [16], BIP [17], PM [18], U-Net [25], and GAMMA [6], protocol-processing overheads induced in communication have been significantly reduced. Some of these messaging systems achieve low latency by pinning down a large area of memory as the send or receive buffers. This avoids the delay caused by the virtual memory system for mapping virtual address to physical address during the messaging stages. Such approach trades memory space for shorter communication latency. However, without a good flow control mechanism or an efficient higher level protocol, this type of messaging systems is usually not scalable. It may experience poor performance when a large number of concurrent communication channels need to be established during the program execution, because of the inefficient memory utilization. Some other messaging systems adopted the user-level approach, which allows moving data from user space to the network adapter without switching contexts or additional memory copy. This type of communication software may achieve shorter latency, however memory copy is sometimes inevitable to maintain the message integrity while extending the messaging system to develop a higher communication layer with more functions, such as the reliable support. Moreover, to avoid violating the OS protection mechanisms, these user-level solutions are usually restricted to have only one process using the communication system on a single host machine.

Besides the performance issue, *programmability* is also an essential goal for the design of communication subsystem. Adequate programmability means that programmer's parallel algorithm can be easily translated to parallel code through the provided API. This requres the provision of a communication abstraction model which can be used to depict various inter-process communication patterns exhibited during the program execution and a powerful yet ease-to-learn API for translating such patterns to program code. Many existing low-latency communication packages provide good performance, but they neglect the need of a simple and user-friendly API. Most of them just form their own programming interface, using complex data structures and syntax, thus making them difficult to use. They usually lack a good abstraction model for the high-level description of the communication algorithm, or an API that copes with the abstraction model and is easy to learn. For example, neither Active Messages nor Fast Messages provide standard set of operations like those used in the widely accepted MPI [14]. A message has to be received within a handler routine, which is specified by the send function. Programs are more easily prone to errors, especially for those with many communication partners and having multiple ways of message handling.

In this paper, we present a high-performance communication system, *Directed Point* (DP), with the goals of achieving high performance and good programmability. The DP abstraction model depicts the communication channels built among a group of communicating processes. It supports not only the point-to-point communication but also various types of group operations. Based on the abstraction model, all inter-process communication patterns are described by a *directed graph*. For example, a directed edge connecting two endpoints represents a unidirectional communication channel between a source process and a destination process. The application programming interface (API) of DP combines features from BSD Sockets and MPI to facilitate the peer-to-peer communication in a cluster. DP API preserves the syntax and semantics of traditional UNIX I/O interface by associating each DP *endpoint* with a *file descriptor*. All messaging operations must go through the file descriptor to send or receive messages. With the file descriptor, a process can access the communication system via traditional I/O system calls.

To achieve high performance with the consideration of generation scalability, DP is designed and implemented based on a realistic yet flexible cost model. The cost model captures the overheads incurred in the host machine and network hardware. It is used as a tool for both communication algorithm design and performance analysis. We consider data communication via the network as an extension to the concept of memory hierarchy. Thus, we abstract the communication event by means of local and remote data movements, and express all parameters by their associated cost functions. This model helps the design of DP adapt well to various speed gaps in processor, memory, I/O bus, and network technologies, thus achieves good generation scalability.

Based on the cost model, we propose various optimizing techniques, namely *directed message* (DM), *token buffer pool* (TBP), and *light-weight messaging call*. DP improves the communication performance by reducing protocol complexity through the use of DM, by reducing the intermediate memory copies between protocol layers through the use of TBP, and by reducing the context-switching and scheduling overhead through the use of light-weight messaging calls. DP allocates one TBP for each DP endpoint. It requires no common dedicated global buffers for storing incoming messages in the kernel space or user space. When a process needs to maintain a large number of simultaneous connections or multiple parallel programs are in execution, separate control of receive buffers avoids locking overhead. Moreover, the memory resource in a host machine can be efficiently utilized and can eliminate unnecessary memory copy as the message buffers are mapped to both the kernel space and the user space.

We have implemented DP for various networks, including Intel EEPro Fast Ethernet, Digital 21140A Fast Ethernet, Packet Engine G-NIC II Gigabit Ethernet, and FORE PCA-200E ATM. DP effectively streamlines the communication steps and reduces protocol-processing overhead, network buffer management overhead and process-kernel space transition overhead. The performance test of Directed Point shows low communication latency and high bandwidth as well as less memory resource consumption.

For the rest of the paper, we first introduce the Directed Point abstraction model in Section 2. Section 3 describes the architectural background and assumptions of our communication model, together with a layout of all model parameters. The performance enhancement techniques inspired by this cost model are discussed in Section 4. In Section 5, we evaluated and discussed the performance characteristics of two DP implementations, which are based on two different Ethernet-based technologies. In Section 6, we briefly studied and compared DP with other Gigabit communication packages, and finally the conclusions are given in Section 7.

2. Directed Point Abstraction Model

The communication traffic in a cluster is caused by the inter-process communication within a group of cooperating processes, which reside on different nodes to solve a single task. Various communication patterns are usually used in algorithm design, such as point-to-point, pair-wise data exchange, broadcast tree, total-exchange, etc. A communication abstraction model can be used to describe the inter-process communication patterns during the algorithm design stage. It also serves as a guide to implement the primitive messaging operations or API for the underlying communication subsystem. In this section, we describe the abstraction model adopted by Directed Point.

The Directed Point abstraction model [12] provides programmers with a virtual network topology among a group of communicating processes. The abstraction model is based on a *directed point graph* (DPG), which allows users to statically depict the communication pattern, and it also provides schemes to dynamically modify the pattern during the execution. All inter-process communication patterns can be described by a directed graph, with a directed edge connecting two endpoints representing a unidirectional communication channel between a source process and a destination process. The formal definition of the DPG is given below:

Let *DPG* = (*N*, *EP*, *NID*, *P*, *E*), where *N*, *EP*, *NID*, *P* and *E* are:

- ***N* (*Node set*)**: A subset of integer set, representing the nodes in a cluster.
- ***EP* (*Endpoint set*)**: A subset of integer set, representing endpoints of the directed edges.
- ***P* (*Process set*)**: The power set of *EP*, each element in *P* represents all endpoints created by a communicating process in a cluster. For example, P_i represents all the endpoints created by process i. A process set in DPG is usually shown as a circle; while the endpoint is shown as a vertex in the circle.
- ***NID* (*Node identification function*)**: *NID* is a function from *P* to *N*, representing the node in a cluster where a process resides. For simplicity, we write *NID*(P_i) as NID_i. The restriction on *NID* is that $\forall P_i, P_j \in P : NID_i = NID_j \rightarrow P_i \cap P_j = \phi$. This property ensures that no two processes in the same node share the same endpoints.
- ***E* (*Edge set*)**: $E = \{\langle i, m, j, n\rangle \mid (i \in P_a) \wedge (j \in P_b) \wedge (NID_a = m) \wedge (NID_b = n) \wedge (a \neq b)\}$ where i, j, m, n, a, and b are all integers, P_a and $P_b \in P$. We use the notation $<i,m>\rightarrow<j,n>$ to represent an edge $<i,m,j,n>$ in *E*, which is a communication channel for sending messages from the endpoint i of process a to an endpoint j of process b .

In the DP abstraction model, each node in the cluster is assigned a unique, known-by-all logical identity called the *Node ID* (NID), and every endpoint of the directed edge is labeled with another unique identity known as *Directed Point ID* (DPID). Thus we can uniquely identify a communication channel (the directed edge) by using the 4-tuple notation {*local DPID, local NID, peer DPID, peer node NID*}. The proposed model supports not only the point-to-point communication but also other types of group operations. For example, an endpoint can be used as the root of a broadcast tree or a destination point for a *reduce* operation. Below is an example to illustrate the usage of the DP abstraction model.

Given a DPG = (*N, EP, NID, P, E*), where

$N = \{1, 2, 3\}$

$EP = \{1, 2, \ldots\ldots, 256\}$

$P = \{P_1, P_2, P_3, P_4\}$

$NID_1 = 1, \quad NID_2 = 1, \quad NID_3 = 2, \quad NID_4 = 3$

$P_1 = \{1, 2, 3\}, \quad P_2 = \{5, 6\}, \quad P_3 = \{1, 2, 7, 8\}, \quad P_4 = \{3\}$

$E = \{ <2,1,2,2>, <1,2,3,1>, <1,1,5,1>, <8,2,3,3>, <6,1,1,1> \}$

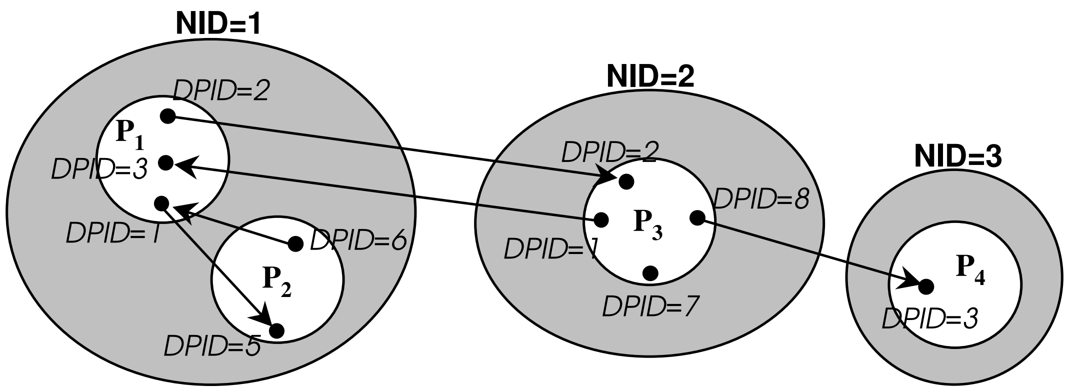


Figure 1. A simple example of the DP graph

Figure 1 shows the corresponding DP graph of the given example. A white circle represents a communication process, each vertex represents a communication endpoint, and a directed edge specifies a unidirectional communication channel between a pair of DP endpoints. From the function NID, we know that process 1 and process 2 are executed in node 1. There are five communication channels between these processes. For example, the channel $<1,1> \rightarrow <5,1>$ is from the endpoint 1 of process 1 to the endpoint 5 of process 2.

The DP graph provides a snap shot of the process-to-process communication. The inter-process communication pattern can evolve by adding a new endpoint within a process, adding a new edge between two distinct endpoints in different processes, deleting an endpoint as well as the edges linked to it, or deleting an edge between different endpoints. With these operations, any run-time inter-process communication patterns can be modeled. With the use of DPG, we abstractly depict the communication patterns and resource requirements of the parallel computation, and this greatly simplifies the programming task.

While capturing the design features presented above, the programming interface of DP is also simple to use. It follows the peer-to-peer communication model, providing functions such as *dp_open()*, *dp_read()* and *dp_write()*, which are analogous to the *open()*, *read()* and *write()* system calls used in the traditional BSD Socket communication. A summary

Table 1
The Directed Point Application Programming Interface

New System Call	
int *dp_open*(int *dpid*)	create a new DP endpoint
User-level Function Calls	
int *dp_read*(int *fd*, char ***address*)	read an arrived DP message
int *dp_mmap*(int *fd*, dpmmap_t **tbp*)	associate a token buffer pool (see Section 4.2) with a DP endpoint
void *dp_close*(int *fd*)	close the DP connection
Light-weight Message Calls	
int *dp_write*(int *fd*, void **msg*, int *len*)	send a DP message
int *dp_fsync*(int *fd*, int *n*)	flush n DP messages in the token buffer pool
void *dp_target*(int *fd*, int *nid*, int *dpid*)	establish a connection with the target endpoint

of the DP API is shown in Table 1. The DP API Layer consists of system calls and user level function calls, which are operations provided to the users to program their communication codes.

To provide better programmability, DP API preserves the syntax and semantics of traditional UNIX I/O interface by associating each DP endpoint with a file descriptor, which is generated when a DP endpoint is created. All messaging operations must go through the file descriptor to send or receive messages. The communication endpoint is released by closing the file descriptor. With the file descriptor, a process can also access the communication system via traditional I/O system calls.

The DP API provides a familiar user interface to application programmers, which can reduce the burden of learning new API. Moreover, programmers need not deal with the IP address and the port numbers of computing nodes anymore. Instead, the NIDs and DPIDs are used to specify every communication endpoint. By constructing the communication pattern in the form of a DP graph as illustrated above, one can easily translate the DP graph into application code without any knowledge of the IP or hardware address information.

Many other low-latency communication packages provide good performance, but they neglect the need of a simple and user-friendly API. Most of them just form their own programming interface, using complex data structures and syntax, thus making them difficult to use. For example, neither Active Messages nor Fast Messages provide explicit receive operations like those used in BSD Sockets. A message has to be received within a handler routine, which is specified by the send function. Programs are more easily prone to errors, especially for those with many communication endpoints and multiple ways of message handling. A programmer may specify the wrong handler, causing the program to exhibit unexpected behaviors. Similarly, GAMMA [6] also allows users to receive messages within a handler routine only. However, instead of putting the handler address into one of the send function arguments, the handler has to be specified at a special function, which is used to establish a communication channel between two GAMMA endpoints before communication takes place. Moreover, a GAMMA communication channel is specified using a 6-tuple: {*local node ID, local process ID, local port ID, dest node ID, dest process ID, dest port ID*}. This is more complicated than DP, where we use a 4-tuple description.

U-Net [25] adopts a peer-to-peer model with explicit user commands for sending and receiving messages between two endpoints through a channel. Although the send and receive operations only require users to specify the local endpoint and channel to be used, setting up endpoints and channels can be tedious. Programmers have to open a U-Net device, and use the returned file descriptor to create an endpoint. A channel is then formed by using the destination address of the U-Net device as input. Each procedure requires an explicit function call, reducing the user-friendliness of U-Net. On the other hand, the communication channel established in BIP [17] can be described using a 4-tuple notation {*local node ID, local tag ID, dest node ID, dest tag ID*}, just like DP does. However, a routing program and a configuration program needs to be executed before a user application is run, so as to determine the network topology and the number of nodes to be used for the application.

The programming interface of PARMA [13] is closely resembled to the BSD Socket interface. Therefore it adopts a client-server model and makes use of Unix system calls such as *bind(), accept(), listen(), read()* and *write()* for network communication. The extra socket layer and the system calls introduce extra overhead. In comparison, DP does not contain such a socket layer and hence eliminates this software overhead. Moreover, DP adopts a peer-to-peer model like the MPI, eliminating the need to perform binding and listening between endpoints of a communication channel. However, it is easy to build up this socket layer on top of DP in order to cope with user convenience and program compatibility issues [4].

3. Directed Point Cost Model

We propose a communication cost model that can be used as a versatile tool for performance analysis/evaluation and algorithm design. By providing a set of model parameters that captures the crucial performance characteristics of the cluster network together with the methodologies to derive those parameters, efficient but portable algorithms can be designed which are well-fitted to the underlying cluster domain.

A model, in general, is an abstract view of a system or a part of a system, obtained by removing the details in order to allow one to discover and work with the basic principle [10]. However, the diversity of computer architectures and the complexity of parallel applications require models to be used at various levels of abstraction that are highly related to the application characteristics. Such models should be developed for the relevant characteristics of the applications together with the characteristics of the architecture. Hence, with such a diverged domain of applications designated for parallel and cluster computing, e.g. regular and irregular problems, a simple, rigid model could not serve our needs.

In general, existing parallel models that focus on message-passing architecture, which include abstract architecture models (e.g. LogP [9], BSP [22]) and communication models (e.g. Postal [1]), usually assume reliable network, such that they treat sending a message as a send-and-forget [1] event. They also assume fully connected network with the exact architecture of the underlying communication network ignored. Communication is based on point-to-point semantics, with the latency between any pair of processors roughly the same time for all cases. These models provide an abstract ground for development. However, they have some drawbacks.

Under BSP model, parallel algorithm is portrayed as a sequence of parallel supersteps, which consists of a sequence of local computations plus any message exchange and follows by a global synchronization operation. With this restricted programming style, the overall usage may be affected. For the LogP model, it tends to be more network-oriented and simple. It uses four parameters to capture the cost associated with the communication events without limits to any programming style. However, its parameters neglect factors related to message size, communication load, and contention issue, which influence the communication latency in a large degree in real networks. An interesting feature of LogP model is the idea of finite capacity of the network, such that no more than certain amount of messages can be in transit from any processor or to any processor at any time, and any attempts to exceed the limit will stall the processor. However, the model does not provide any clear idea on how to quantify, avoid and take advantage of this information in algorithm design. The Postal model is similar to LogP model, with the exception of expressing the network more abstractly. The system is characterized by two parameters only, and this effectively reduces the dimension of analysis. Therefore, it facilitates communication analysis rather than for performance studies.

Most of the above drawbacks come from the tradeoff between simplicity and accuracy. The uses of those parameters are subjected to the target level of abstraction together with the application characteristics that we are going to work on. For instance, using a simple latency parameter may be good enough to capture the cost of the point-to-point communication, but is too simple for explaining the many-to-one or many-to-many issues, where contention problem may affect the communication performance.

In our model, a cluster is defined as a collection of autonomous machines that are interconnected by a switch-based network. We assume it is a packet-switched, pipelined network, and operates in a full-duplex configuration. Buffers are provided in the switches for temporary buffering, but the amount of buffers is assumed to be finite. All cluster nodes communicate via this switch-based network and assume to have the same local characteristics, such as computation power, memory hierarchy, operation system supports, and communication hardware. In our study, we assume that each node is equipped with one set of input and output channels, that is, it can simultaneously send and receive one data packet in one communication step.

A cost model is associated with our model parameters, which is focusing on the costs induced by moving data around, both locally and remotely. We consider data communication via the network as an extension to the concept of memory hierarchy, such as a movement from the remote memory region to the local memory region. So there may have two types of data movements involved in the parallel computation or in a communication event: a) remote data transfer and b) local data transfer. Our model intends to capture the cost of point-to-point communication and also contention overheads in various collective operations. Emphasis has also been made on the derivation of our model parameters by software approach, which is the key to the whole analytical process.

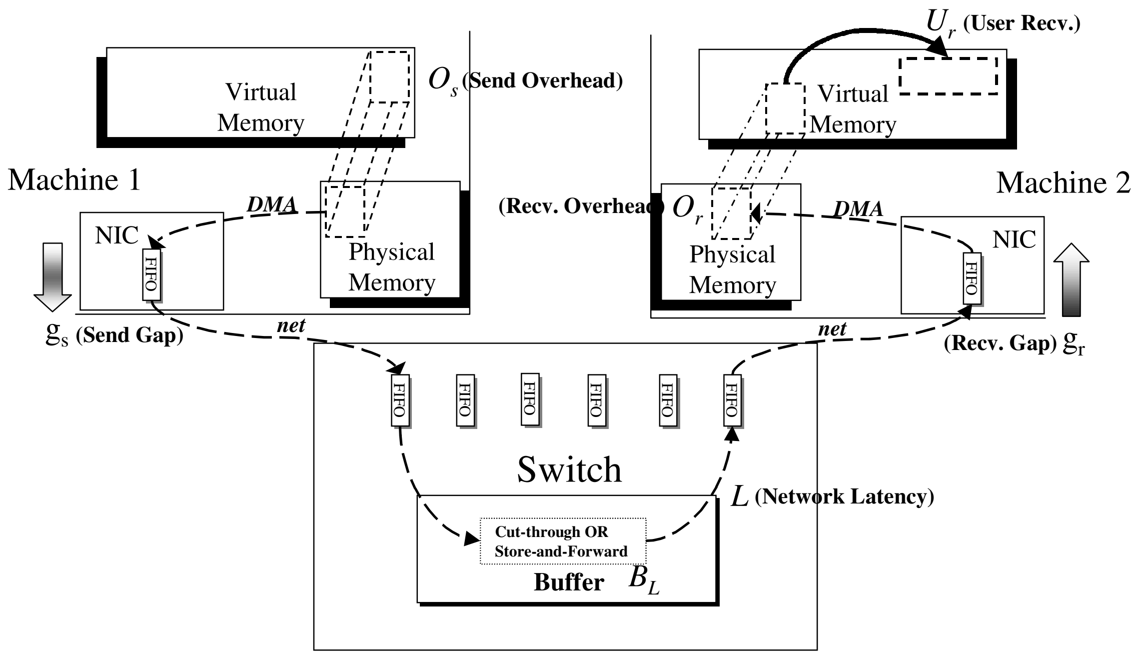


Figure 2. Model parameters affiliated with the remote data transfer

3.1. Remote Data Transfer

We abstract the data movement via the interconnection network by three phases, from the sender address space to the receiver address space. First, communication is initiated by a sender process, which injects data messages to the network during the *send phase*. Then the network delivers the messages to the remote node during the *transfer phase* of the communication. At the remote end, *receive phase* consumes the data and terminates the transfer event. We encapsulate all the overheads of these phases by a set of model parameters, and below is a detail description of individual parameters as well as using a schematic drawing (Figure 2) to show their correlation.

- *Machine size* p - This refers to the number of processes participating in the current event.

- *Send overhead* O_s - This parameter stands for the software overhead associated with the send process for sending an m-byte data packet. From the high level perspective, we view it as the time used by the user process to interact with the logical network interface, prepare the message, queue it to the send queue, and signal the network hardware. The overall cost reflects the processing speed of the host node, the efficiency of the memory subsystem, and the communication protocol in use. We model this parameter by a simple linear function, $O_s(m) = \kappa_s + \tau_s m$, where κ_s is the startup cost of this event which depends on the node processing power, m is the message length bounded by the range [1..MTU], and τ_s is the data transfer rate that depends on the efficiency of the memory subsystem. Subject to the communication protocol, e.g. under zero memory copy, this linear function can be reduced to a simple constant, i.e. $O_s(m) = \kappa_s$. As this is a synchronous event, we quantify this parameter by directly measuring the time engaged by the processor in handling those activities.

- *Inter-packet transmit gap* g_s - Owing to the difference in data movement speeds between the send and transfer phases, the network card is moving data packets with a confined capacity, which is captured by this parameter. For example, the performance of the PCI bus and the network technology are the major factors of this parameter. This inter-packet gap has two slightly different meanings with respect to different perspectives, but in general, it delineates the maximum network throughput available to the user process. From the user process perspective, it views the gap as the minimum service time of the network in transmitting consecutive data packets. Thus, sending data faster than this gap yields no performance gain, and the difference between O_s and g_s indicates the amount of CPU cycles available for the processor to do other useful computation. From the network perspective, it represents the maximum injection rate of the packets to the network. This parameter is delineated as $g_s(m) = g_1 + \tau_1 m$, where g_1 stands for the startup cost necessary to initiate the transfer and τ_1 reflects the available communication bandwidth provided by the I/O bus and the network. Due to the limited buffers in the send queue, if a sender generates messages faster than the network can dispatch, new message can only be accepted if the network has just finished servicing one. By quantifying this servicing time with respect to the message size, the required cost function can be obtained.

- *Network latency* L - This parameter represents the time used by the network in moving an m-byte data packet from the physical memory of the source node to the physical memory of the destination node. For example, from the send queue of the local machine to the receive queue of the remote machine. It is a network-dependent parameter, which encapsulates the performance of the host, I/O bus, the diameter between cluster nodes, the network topology and the network technology in use. If we model the network as a complete graph, both diameter and topology factors can be eliminated. In general, the value of L is subjected to the traffic loading at any particular instant in a real network. For example, when routing the packets through the network, conflicts take place if more than one packet access the same output line, and temporary buffering is needed. This delay affects the overall network performance perceived by the users. The amount of buffer memory inside the switch is assumed to be finite, thus, the network can sustain certain level of congestion rather than just as $\left\lceil \frac{L}{g_s} \right\rceil$ packets in the case of the LogP model. We model this phase by a bilinear function under the congestion-free condition, $L(m,p) = l(p) + m\tau_L(m,p)$, where $l(p)$ is a function representing the cumulative startup cost of this network transfer and $\tau_L(m,p)$ is the available network throughput. Both l and τ_L are a function of p. Routing a packet involves utilization of some central resources (e.g. buffer control unit and forwarding control unit), therefore, contention for resources may occur if more than one routing request happen concurrently. The extent of this contention is subjected to the switch internal architecture, and different routers may behave differently. Some networks may provide limited aggregate bandwidth, they cannot support too many communicating pairs at a time and contention arises. So the allocated network throughput to the transfer phase depends on the aggregate bandwidth of the network, the number of communicating pairs and the volume of the communication, thus, τ_L is a function of both p and m. To measure it by software means, we have to calculate it indirectly. Based on the fact that a pingpong test involves $2*(O_s+L+O_r+U_r)$ time units, if we know all other parameters, $L(m,p)$ can be derived. By artificially generating multiple concurrent pairs of pingpong nodes, the corresponding bilinear function could be obtained by using linear regression.

- *Inter-packet receive gap* g_r - This parameter stands for the minimum time interval between two consecutive receptions experienced by the receiving host, which is limited by the performance of the I/O bus and the network technology in use. Similar to the g_s parameter, it is used to delineate the maximum packet arrival rate delivered by the network. This parameter has two uses. First, the inter-packet receive gap reflects the CPU cycles available to handle arrived packets, so this information should be taken into consideration during protocol design. Second, this gap relates to the minimum service time of the switch in delivering packets, as we cannot receive more than one packet within that interval. We can adopt this information in scheduling communication events. Same as the g_s parameter, it is captured by a linear function, $g_r(m) = g_2 + \tau_2 m$. For simplicity, on a homogeneous cluster, we can generally assume $\tau_1 = \tau_2$, as both are related to the transfer capability of the network and the I/O bus. To quantify this parameter, we make use of the many-to-one communication pattern, and measure the minimum time perceived by the receiver in detecting the arrival of data packets.

- *Network buffer capacity* B_L - Limited resources are the major cause of congestion, which in turn, affect the delay experienced by the applications. In reality, congestion is a fact that we need to face with. The parameter B_L corresponds to the available buffers in a switch, which is a measure of the network tolerance of a switch in handling contention. For a single switch, we only have one B_L value, either it associates with the whole switch if it is a shared-buffered switch, or associates to a switch port if it is an input-buffered or output-buffered switch. By capturing the finite capacity of the network buffers, algorithm designers can calculate the network endurance, and avoid contention loss with appropriate communication schedule. To quantify this parameter, we perform a set of tests which flood the switch under different traffic loads, and record the percentage of packet arrival at the destination. Then, by mapping the data with the flow analysis equation [20], we can estimate the buffer capacity of a router switch.

- *Asynchronous receive overhead* O_r - This parameter captures the software overhead in handling incoming messages. An arrived message is handled by the kernel thread and does not involve the receiving process, so the reception is considered as an asynchronous event. It captures the costs of all kernel events including interrupt, memory copy and context switch, and its efficiency is affected by the processing speed of the processor and the communication protocol in use. In our model, we express it as a linear equation, $O_r(m) = \kappa_r + \tau_r m$. In which κ_r represents the minimal cost of this asynchronous event, such as interrupt cost, buffer management, and protocol overhead; while τ_r mainly reflects the speed of memory movement between different memory regions if needed. Observed that during the reception, system resources are consumed, therefore, other computations would be affected. Under experimental control, we first measure the normal execution time of a small but CPU-bounded code segment, and then we measure another run of this computation segment with message reception happening in the background. Then we estimate the increase in execution time induced by the message reception. Hence, we have indirectly measured the induced software overhead.

- *User receive overhead* U_r - Due to the asynchronous nature of the communication, the receiving process needs to find some means to check for data arrival, e.g. polling, block & wake-up by signal, or hybrid approaches; and consumes the data, e.g. copy to other memory segment. This parameter reflects the software overhead spent by the receiving process after arrival of messages. In most of the performance evaluation reports, due to the artificial nature of the benchmark programs, this parameter is of insignificantly low cost. However, in real parallel computing, this overhead reflects the performance loss due to improper coordination of communication events. For example, in a non-dedicated cluster environment, polling is a user-level event that is affected by the regular CPU scheduling policy. If the receiving process cannot be scheduled frequently to poll for its data, the overall performance may degrade a lot.

3.2. Local Data Transfer

- *Memory copy overheads* M_{ctc}, M_{ctm}, & M_{mtm} - Memory copy issue has been extensively studied in the past, and is being classified as a high overhead event. To avoid this overhead, most of the low-latency communication systems have removed it from their protocol stacks. However, in reality, memory copy operations cannot be avoided completely. To quantify these costs, we provide three memory copy parameters - M_{ctc}, M_{ctm}, and M_{mtm} to represent the costs induced by data movement between different memory hierarchies, such as the cache-to-cache, cache-to-memory, and memory-to-memory data movement.

For a homogeneous cluster, we can generally assume that $g_s \approx g_r$ and simplify the expression by $g = \max(g_s, g_r)$. To take advantage of the full-duplex communication, we assume that the cluster communication system must satisfy this condition, $(O_s + O_r + U_r) < g < L$. This assumption is generally true under the current CPU technologies and the adoption of low-latency communication. As a result, under no conflict, the one-way point-to-point communication cost (T_{p2p}) in transferring an M-byte long message between two remote user processes is:

$$T_{p2p}(M) \approx O_s + (k-1)g + L + O_r + U_r \tag{1}$$

where $k = \frac{M}{b}$, which corresponds to the fragmentation of an M-byte message to k data packets of size b bytes. For optimal performance, b usually stands for the maximum transfer unit of the underlying communication scheme.

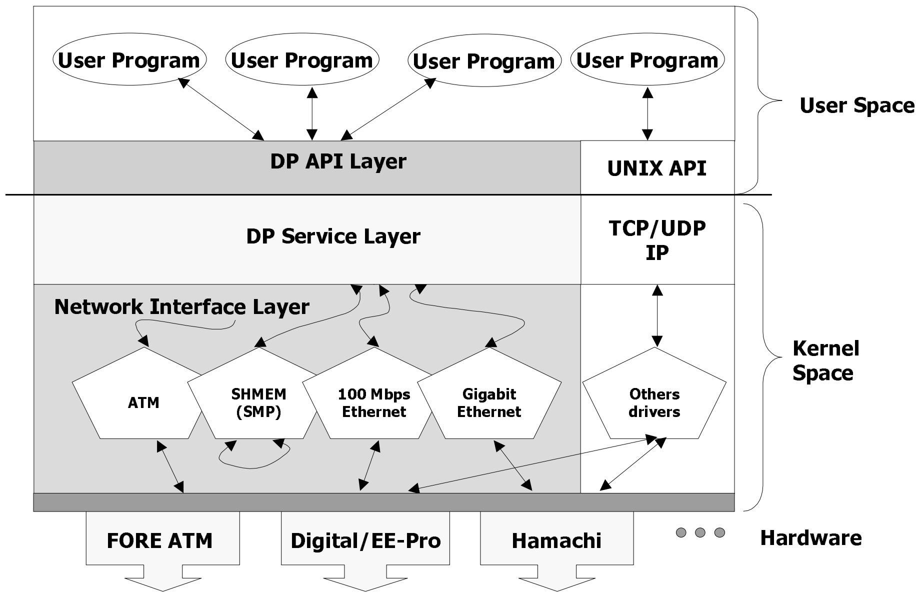


Figure 3. The architecture of DP

4. Directed Point System Architecture

The cost model discussed in the previous section not only serves as a means of performance analysis and evaluation, it also throws some light to improve the performance of the communication system architecture. Based on the cost model, we derive the Directed Point system architecture, which involves a number of performance enhancement techniques to be discussed in this section.

4.1. Overview of DP System Architecture

The system architecture of DP is shown in Figure 3. It consists of three layers: the DP Application Programming Interface (DP API) Layer in the user space, the DP Services Layer and the DP Network Interface Layer in the kernel space.

The DP Network Interface Layer consists of network driver modules. Most of the driver modules are hardware dependent. Each of them is an individual kernel module that can be loaded to and unloaded from the system without the need to recompile the whole kernel source tree. Multiple network interfaces can be loaded at the same time. Currently, supported network driver modules include Intel EEPro, Digital 21140A Fast Ethernet, Hamachi Gigabit Ethernet, and FORE PCA-200E ATM. We also develop the DP SHMEM module to support intra-node communication through shared memory. This layer is responsible for all hardware-specific messaging setup, and signaling the hardware to receive/inject messages from/to the network. The DP Services Layer implements services for passing the packets from user space to the network hardware, as well as delivering the incoming packets to the user space buffers of the receiving processes. This layer realizes the DP abstraction model and is hardware independent. Hence, to interact with

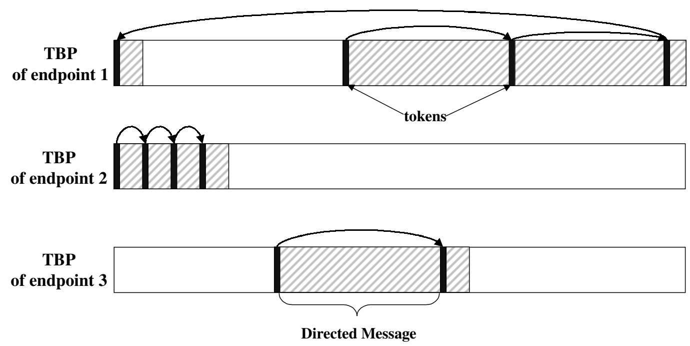


Figure 4. The data structure of the Token Buffer Pool (TBP)

the network hardware, the DP Services Layer must go through the DP Network Interface Layer.

The DP API Layer consists of system calls and user level function calls, which are operations provided to the users to program their communication codes. Full discussion of the DP API is provided in Section 2.

4.2. Light-Weight Messaging Techniques

DP is designed with the goals to achieve low communication latency and high bandwidth as well as minimizing the resource usage. We propose various techniques, namely, Directed Message, Token Buffer Pool, and Light-weight Messaging Call. They reduce protocol processing overhead, network buffer management overhead and process-kernel space transition overhead.

- ***Directed Message (DM)***: To decouple the processor with the network, and allow overlapping of computation with communication, DP divides a long message into multiple smaller packets for transmission. Each packet is called a *Directed Message*. It consists of a header and a data portion called the container. The header is constructed at the DP Service Layer and stores the destination NID and DPID, as well as the length of the container. The simplicity of DM packet requires very small packet processing time as compared to other complex protocols, and therefore reduces the O_s and O_r overheads. In addition, with Directed Message and the support of asynchronous communication, users have more control over the scheduling of communication, and efficient communication algorithms can be devised.

- ***Efficient Buffer Management***: DP provides a dedicated buffer at the receiving side to store the incoming messages. This is known as the *Token Buffer Pool* (TBP). TBP is a fixed size physical memory area dedicated to a single DP endpoint (as depicted in Figure 4). It is allocated when the communication endpoint is opened and freed when the endpoint is closed. The basic unit of storage in TBP is called token buffer. It is a variable-length memory chunk for storing the incoming DM packet. This design can improve the memory utilization as compared to the fixed length buffer used in other implementations. Each token buffer has a control header, called *token*. It is a data structure containing the length of a token buffer and linkage information to the next chained token buffer. The TBP is shared by the kernel and the receiving process through page remapping. This eliminates the delay caused by data copying from the kernel space to user space, and significantly reduces the user receive overhead (U_r).

- ***Light-weight Messaging Calls***: Most messaging procedures implemented in Directed Point are implemented in the kernel level. This provides maximum protection to the network resources. Moreover, the messaging calls to the kernel level routines are light-weight ones, through the use of Intel *x86 call gate*. Unlike traditional system calls which incur CPU exception, process rescheduling and context switching, the call gate provides a way to enter the kernel with no extra overhead. This light-weight system call mechanism reduces the software overhead of switching to/from kernel level, e.g. reduces the send overhead (O_s), and eventually, reduces the overall communication latencies.

The design of high-speed messaging should also consider the performance gaps between processor, memory, and the network. For example, a slow host machine, such as a Pentium PC, is hardly able to drive the 100 Mbits Fast Ethernet in full speed. Reducing the protocol handling overhead (O_s, O_r & U_r) and minimizing the memory copy costs can effectively improve the performance. Even though, if memory copy is required, DP tries to take advantage of the cache locality and avoid memory-to-memory copy. Similar situation can be observed while using a Pentium III machine to drive the Gigabit network. On the other hand, additional memory copy performed in a relatively high-speed host causes no performance degradation while driving a slow network.

We use our Gigabit Ethernet (GE) implementation as an example to illustrate the data movement journey adopted in DP. The Hamachi GE NIC uses a typical *descriptor-based bus-master* architecture [2]. Two statically allocated fixed-size descriptor rings, namely, the *transmit* (Tx) and *receive* (Rx) descriptor rings. Each descriptor contains a pointer to the host physical memory that stores incoming and outgoing messages. Figure 5 shows the messaging flow with respect to different components in DP using such descriptor-based network interface controller.

After preparing the message, the send process initiates the transmission by calling the *dp_write()* operation, which switches the execution control from the user space to kernel space. Now, operations (IOR) in the DP Service Layer are triggered to construct the Directed Message. To shorten the transmission delay, the Network Address Resolution Table (NART) is used to translate the NID to the network address. It contains the network addresses of all computing nodes in the cluster. Then the DM message is passed over to the DP Network Interface Layer, which directly deposits the message to the Tx descriptor ring and signals the network adapter to inject the packet to the network.

When the packet arrives, an interrupt signal is triggered by the network adapter. The interrupt handler calls the Message Dispatcher Routine (MDR) - a service in the DP Service Layer, examines the header of packet, locates the destination TBP based on the information stored in arrived DM, and copies the incoming message to a buffer at TBP.

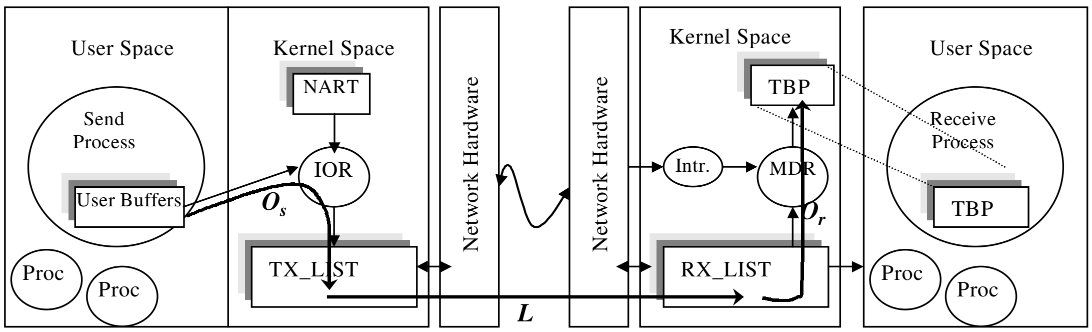


Figure 5. The messaging flow in DP

Since TBP is accessible by both kernel and user processes, the incoming message can be directly used by the user program.

DP allocates one TBP whenever a new DP endpoint is opened by the user program. It requires no common dedicated system buffers (such as *socket buffer* in BSD Socket) for storing incoming messages. Separate control of receive buffer avoids locking overhead when a process needs to maintain a large number of simultaneous connections. Moreover, the memory resource in a server can be efficiently utilized. The amount of memory needed is proportional to the number of endpoints created in the applications. The total memory consumption is roughly the same as the size of total incoming packets because of the use of variable-length token buffer.

In summary, DP improves the communication performance in the following ways: (1) by reducing protocol complexity through the use of DM, (2) by reducing the intermediate memory copies between protocols through the use of TBP, and (3) by reducing the context-switching and scheduling overhead through the use of light-weight messaging calls. Other low-latency messaging systems have also adopted similar performance enhancement techniques but using different strategies. For example, AM eliminates intermediate buffering at the receiving end by pre-allocating storage for the arriving data in the user program. AM reduces the scheduling overhead through the use of receive handlers, which interrupt the executing process immediately upon message arrival. GAMMA, an AM variant using Ethernet-based network, also inherited these techniques. It bypasses the TCP/IP protocol overhead and reduces the context-switching overhead through a small set of light-weight system calls and a fast interrupt path. Moreover, to ensure that messages can be moved directly from the adapter to the receiver buffer, it requires the user to explicitly pin down the receiver buffer before initiating the associated communication event. It also reduces the memory copy in the send operation, by moving the messages right away to the adapter's FIFO queue, without intermediate buffering in kernel space. On the other hand, FM also eliminates excess data copying, as the FM interface uses message streams to eliminate the need to marshal and un-marshal the messages to be communicated. Users do not need to perform explicit memory copy to assemble a message to be sent, even the message contents are not contiguous in the user memory. However, as mentioned, the handler approach for receiving messages, as adopted by these three packages, lacks programmability and portability as compared with DP.

For U-Net on ATM, performance is enhanced through removing the kernel from the critical path of sending and receiving messages. However, due to the absence of network co-processor on the Ethernet adapters, U-Net on FE has to go through the kernel space for OS protection. To reduce memory copy, it pre-allocates and pins down a fixed memory segment, called U-Net endpoint, which is structured to form a complex virtual network interface. During communication, users have to deposit and retrieve messages through this endpoint structure. Therefore, users are compelled to allocate a large memory segment right at the beginning, which is large enough to sustain the maximum load, but could only be released at the end of the application. Finally, PARMA aims at designing a greatly simplified protocol (known as PRP protocol) as compared to TCP/IP. By neglecting flow control and error recovery, PARMA succeeds in reducing the protocol overhead. However, the implementation of a Socket layer introduces certain amount of system call overhead.

5. Performance Analysis

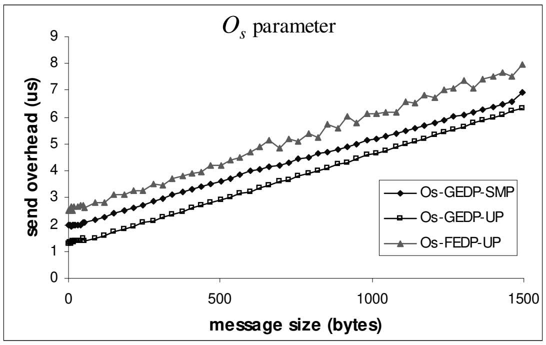
We have implemented Directed Point on two Ethernet clusters, one is a Fast Ethernet cluster (FEDP) and the other is a Gigabit Ethernet cluster (GEDP). The FEDP cluster consists of 16 PCs running Linux 2.0.36. Each node is equipped with a 450MHz Pentium III processor with 512 KB L2 cache and 128MB of main memory, and uses a Digital 21140A Fast Ethernet adapter for high-speed communication. The whole cluster is connected to a 24-port IBM 8275-326 Fast Ethernet switch which has 5 Gbps backplane capacity. For the GEDP cluster, it consists of 4 Dell PowerEdge 6300 SMP servers with 4 Pentium III Xeon processors sharing 1 GB memory. The Xeon processor consists of 512KB L2 cache and operates at 500 MHz. All servers are running on Linux 2.2.12 kernel and equip with one Packet Engine G-NIC II Gigabit Ethernet adapter, and are connected to the Packet Engine PowerRail 2200 Gigabit Ethernet switch, which has a backplane capacity of 22 Gbps.

To review the performance issues related to high-speed communication on clusters, we have performed a series of benchmark tests on these clusters. To achieve beyond-microsecond precision, all timing measurements are calculated by using the hardware time-stamp counters in the Intel Pentium processors. If applicable, all data presented in this section are derived from a statistical calculation with multiple iterations of the same benchmark routine. Each test is conducted with at least 200 iterations with the first and last 10% of the measured timing excluded. Only the middle 80% of the timings are used to calculate the average.

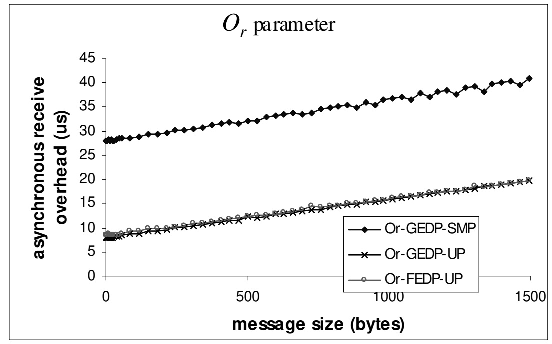
5.1. Latency with Performance Breakdowns

We analyze the performance of our DP implementations based on the communication model described in Section 3. As the model parameters represent some forms of software overheads and hardware latency, changes in communication hardware and software are being revealed by any changes in these model parameters. This gives us better insights on the performance impacts of various design choices.

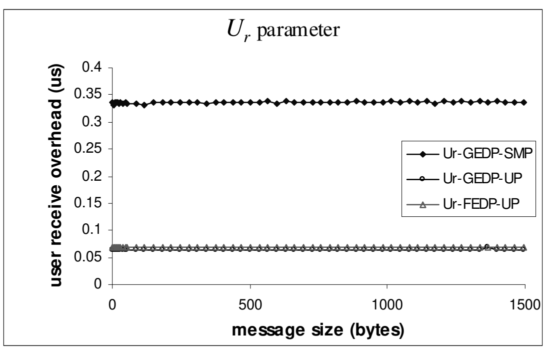
By executing the associated benchmark routines, we construct a set of model parameters



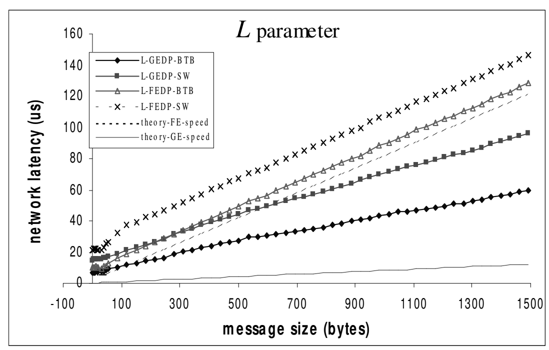
(a) Send overhead



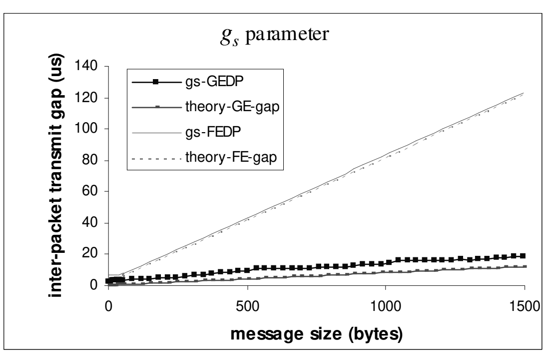
(b) Asynchronous receive overhead



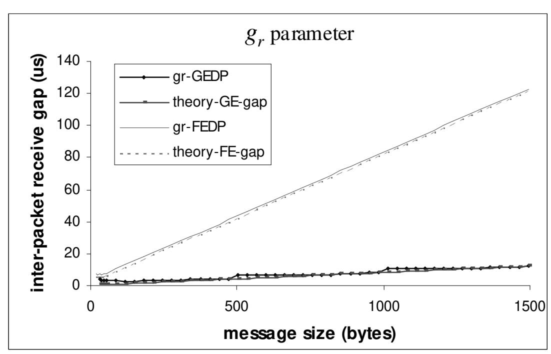
(c) User receive overhead



(d) Network latency



(e) Iner-packet transmit gap



(f) Inter-packet receive gap

Figure 6. Performance breakdown of two DP implementations - Fast Ethernet (FEDP) and Gigabit Ethernet (GEDP) expressed in the form of our model parameters.

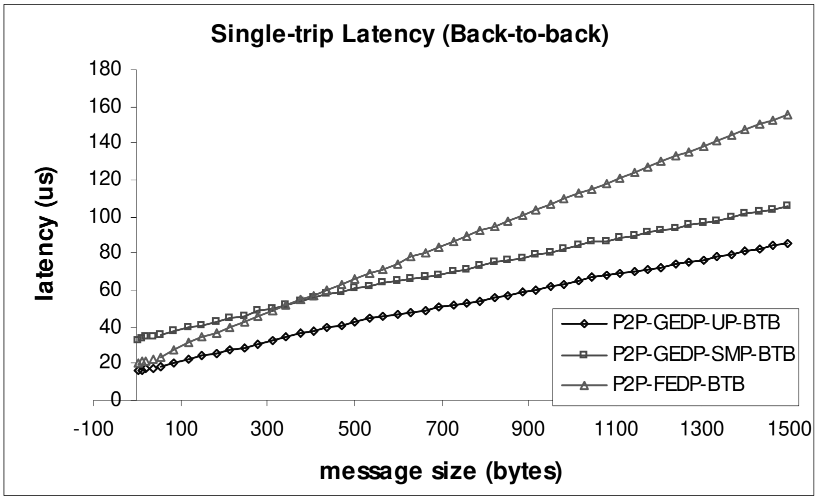


Figure 7. Single-trip latency performance with back-to-back setup

for the two clusters, as shown in Figure 6. In the figure, there are two sets of parameters for the Gigabit Ethernet implementation (GEDP), one is obtained when using an SMP kernel, i.e. with the SMP support for the Linux 2.2.12 kernel (GEDP-SMP), and the other is without SMP support (GEDP-UP), i.e. uni-processor mode on an SMP server. The purpose of this comparison is to reveal the differences in performance with respect to different OS modes and hardware platforms.

The O_s parameter reflects the time used by the host CPU to initiate the transmission while performing the *dp_write()* operation. Figure 6(a) shows the cost associated with the *dp_write()* operation. The 500MHz Xeon processor slightly performs better than the 450MHz Pentium III processor. We observe that the penalty of preserving the traditional Unix I/O abstraction in DP is the use of one-copy semantic in the send operation. However, by adoption of the Light-weight Messaging Call and the Directed Message protocol, we manage to minimize the send overhead and achieve good performance in driving the Gigabit network. For example, the cost to send a full-size Ethernet packet is less than 7 μs under the SMP OS, while the theoretical speed in delivering such an Ethernet packet under Gigabit performance is around 12.3 μs. With the SMP mode, there is an extra 0.5 μs overhead associated with it due to the use of locks for integrity control.

When examining on the O_r parameter - Figure 6(b), we find that SMP OS has an extra 20 μs overhead added on to this parameter, while both GEDP-UP and FEDP-UP have similar performance. This is also observed in the single-trip latency of the GEDP as shown in Figure 7, which is measured with the traditional *pingpong* test with back-to-back connection. There is a large performance gap appearing between the two OS modes. We conclude that this extra overhead is induced by the support of symmetric I/O and locking mechanism in the SMP kernel. Besides the SMP overhead, we also observe that the current architecture of the Gigabit Ethernet adapter has a limitation on the achievable performance. Due to the lack of intelligence network processor, data messages are not delivered to the user process directly and this requires a memory copy done in the interrupt handler. The one-copy cost together with the interrupt overhead would become a threat to the overall performance, e.g. the total interrupt cost for the full size packet is 19.7 μs under GEDP-UP.

Since the token buffer pool is accessible by the kernel and user process, the receiving process can simply check on this TBP for picking up and consuming the messages. As these are done in the user space, no kernel events such as block and wake-up signal are needed. Figure 6(c) shows the U_r cost of picking up a Directed Message directly from the TBP without any memory copy cost or system call overhead. Constant overheads, 0.34 μs, 0.06 μs and 0.07 μs were measured for GEDP-SMP, GEDP-UP, and FEDP, respectively.

Figure 6(d), (e) and (f) show three other network-dependent parameters, they are the network latency L, inter-packet transmit gap g_s and inter-packet receive gap g_r. To justify their relative performance, all parameters are compared with their theoretical limits. Looking at the FEDP data, we find that with modern PC or server hardware and low-latency communication system, we are able to drive the Fast Ethernet network with its full capacity. For example, the measured g_s and g_r for $m = 1500$ bytes is 122.75 μs and 122.84 μs, while the theoretical gap is 123.04 μs. This means that the performance of the host machine is faster than the speed of the Fast Ethernet network in all aspects. For the Gigabit Ethernet, due to the 10-fold increase in network speed, limitations within the host machine start to pop up. The graph with g_s-GEDP data (Figure 6e) shows that the network adapter cannot transmit data in full gigabit performance. The measured g_s for $m = 1500$ bytes is 18.76 μs but the theoretical gap is 12.3 μs. We have performed some preliminary investigation on this aspect, and the problem seems related to the PCI performance, even though our Dell server is coupled with a 64bit 33MHz PCI bus. A similar pattern also appears in the g_r-GEDP data, but is not as significant as that of the g_s parameter. The measured g_r for $m = 1024$ bytes is 10.6 μs but the theoretical gap is 8.5 μs. Part of the reason may be due to the difference in read and write performance of the PCI bus.

Lastly, when look at the L parameter, the calculated network latency of the GEDP with back-to-back connection is 6.9 μs for a 1-byte message, while the network latency of the FEDP with back-to-back connection is 9.9 μs for the same size message. We observe that the add-on latency by the GE hardware is much higher than that of the FE, when we compare the theoretical wire delay for the smallest packet size of the GE and FE, which are 0.67 μs and 6.7 μs respectively. For example, in Figure 6(d), the gaps between the network latency measurements with FE back-to-back and FE through switch, and between FE back-to-back and theoretical FE speed are almost constant, while the corresponding gaps on the GE platform seem to be increasing with the message size.

Figure 7 shows the latency results of the two DP implementations. To avoid add-on latencies from the switches, we connect two nodes back-to-back and measure their single-trip latencies. The GEDP-UP achieves single-trip latency of 16.3 μs for sending 1-byte message, while GEDP-SMP achieves 33.4 μs and FEDP achieves 20.8 μs respectively. From the above analysis, we obtain a set of performance metrics, which clearly delineate the performance characteristics of our DP implementations. In summary, the host/network combination of the FEDP implementation has the performance limitation on its network component. This is being observed by comparing the O_s, O_r, and U_r parameters with the g_s, g_r and L parameters. And since their performance characteristics satisfy this condition, $(O_s + O_r + U_r) < g < L$, we can directly adopt the previous defined point-to-point communication cost (T_{p2p}) whenever we want to evaluate on its long message performance. Moreover, the host/network combination of the GEDP implementation has the performance limitation not falling on the network component. For instance, the O_r parameter is higher than the g_s and g_r parameters for both GEDP-SMP and GEDP-UP, which means the performance bottleneck may fall on this region. Therefore, when predicting their long message performance, new point-to-point communication cost formulae are required. For example, the new cost formula for predicting the one-way point-to-point communication cost of the GEDP-UP implementation becomes:

$$T_{p2p-GEDP-UP}(M) \approx O_s + L + k(O_r + U_r) \qquad (2)$$

5.2. Uni-directional Bandwidth Test

In this section, we are going to explore the one-way bandwidth performance of our DP implementations with respect to different hardware and OS mode. Two sets of uni-directional bandwidth measurements are presented in Figure 8. To calculate the *raw* DP bandwidth, we measure the time to transmit 10 MB data from one process to another remote process, plus the time for the receive process to send back a 4-byte acknowledgment. By subtracting the measured time with the single-trip latency of a 4-byte message, we calculate the achieved bandwidth as the number of bytes transferred in the test divided by the result timing. We have implemented a simple Go-Back-N protocol on top of DP to provide flow control and support limited reliable communication. Since all the protocol works are done in the user space, it has add-on overheads to the O_s and U_r parameters. For example, the O_s-GEDP-SMP value for sending a full load packet is increased from 6.9 μs to 10 μs. To calculate the flow-controlled bandwidth of DP, we performed a set of tests similar to what we have done to obtain the raw DP bandwidth.

From the figure, we see that the maximum achieved bandwidth for GEDP is 79.5 MB/s, which is the raw DP performance measured under the SMP kernel. Under the UP kernel, the raw GEDP achieves only at most 75.2 MB/s. Despite the fact that the SMP kernel has a higher interrupt overhead, it has a better throughput than the UP kernel. This shows the advantage of sharing the token buffers between the kernel process and user process. Under the UP mode, the user process can only pick up its arrived messages after the interrupt thread returns, so the whole interrupt overhead is included in the delay calculation. But, with the SMP mode, the user process can check out its messages whenever it gets the CPU cycles and detects that there are arrived messages, even before the interrupt thread returns. For the FEDP-UP, the maximum achieved raw DP bandwidth is 12.2 MB/s, which is 97% of the Fast Ethernet performance, while the raw GEDP only achieves 63.6% of the theoretical gigabit performance. This shows that there are limiting factors in the host machines which hinder the GE performance. We have seen in Figure 6(e) that the network adapter cannot transmit data in full gigabit performance, by dividing the payload size with the corresponding g_s value, we have a good meter to estimate maximum performance we can achieve. Take the value of $g_s = 18.7\ \mu s$ at $m = 1500$ bytes as an example, we find that the maximum transmission throughput

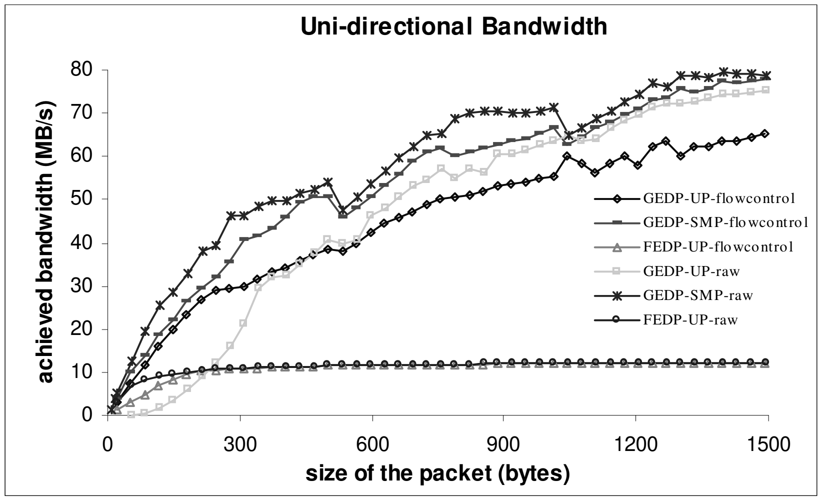


Figure 8. Uni-directional bandwidth performance

is around 80MB/s, which is closely matched with the above measurement. To reveal how much improvement we could achieve if we adopt a zero-copy semantic in the send path, we have done some tests that simulated a zero-copy send operation, (simply by removing the *memcpy()* operation and sending out garbage content). The resulting send gap (g_s) is approximately 16.4 μs for $m = 1500$ bytes, which corresponds to a bandwidth of 91.5 MB/s.

With flow control on, the FEDP performs as good as the raw performance for medium to large-sized messages. But for the GEDP, the higher protocol overhead does affect the overall performance, especially under the UP kernel mode. Our result shows that under the SMP mode, the maximum achieved GEDP bandwidth with flow control is 77.8 MB/s, with an average drop of 3.4% performance for the data ranged between 1K and 1.5K when compared with the raw speed. While for the performance under UP mode, the maximum achieved bandwidth with flow control is 65.2 MB/s and the average performance drop is 13% of the raw speed for the same data range. This further supports our argument that the performance of GEDP-UP is more susceptible to software overheads.

5.3. Bi-directional Bandwidth Test

Most networks support bi-directional communication and lots of communication patterns require concurrent send and receive operations to achieve optimal results, e.g. complete exchange operation, shift operation, tree-based broadcast, etc. We extend the tests used for uni-directional bandwidth to evaluate the communication performance of the bi-directional communication. During the experiment, two nodes are involved in each test, but they are both sender and receiver. To measure the raw bi-directional bandwidth of DP, both processes are synchronized by a barrier operation before starting the exchange. We measure the time spent by each process in exchanging 10 MB of data,

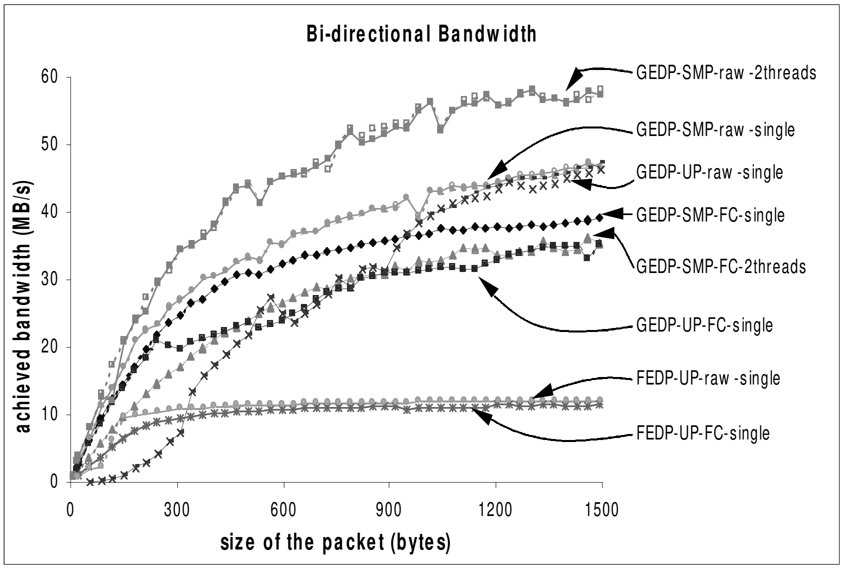


Figure 9. Bi-directional bandwidth performance

and calculate the bandwidth by dividing the exchange message size with the measured time. Similarly, we perform the same set of tests with the add-on reliable layer. When testing the bi-directional bandwidth on the SMP kernel, we also try to explore the effect of using multiple CPUs in driving the communication. We have performed a set of tests with two threads per process, which share the same DP endpoint, one thread takes up the job as the sender while the other acts as the receiver. All the experimental results are summarized in Figure 9.

For the GEDP, the best bi-directional performance is observed to be about 58 MB/s per process, which is measured on raw DP using multi-thread mode on SMP kernel. Comparing with the uni-directional bandwidth, we have a performance loss of 22 MB/s. We attribute this performance loss to the contention on the PCI bus as there are concurrent DMA transfers to and from the host memory. When compared with the single thread mode on GEDP-SMP and GEDP-UP, which only achieve 47 MB/s per process, we believe that the software overhead induced in the concurrent send and receive operations is the main cause of this performance loss. Therefore, with the add-on flow control (FC) layer that adds more software overhead, it is sensible to see that all GEDP-FC performance suffers more. However, it is surprising to find that the bi-directional performance of GEDP-SMP-FC with multiple thread support is worse than the single thread mode. This performance difference is coming from the extra memory contention and synchronization needed in accessing shared data structures on the reliable layer as both threads are concurrently updating those shared information. Finally, similar to the conclusion as appeared in the uni-directional benchmark, the performance of the FEDP on bi-directional communication has achieved a near-optimal result, which attains 12.16 MB/s per process on the raw bandwidth, and 11.7 MB/s per process with the add-on flow control support.

6. Related Works

In the past, several prototype cluster communication systems based on Gigabit networking technology have been built. The Genoa Active Message Machine (GAMMA) [6,8] is an experimental communication system on Fast Ethernet and Gigabit Ethernet. The GAMMA driver uses a mechanism derived from Active Messages called Active Ports [7]. Active Ports is the abstract communication endpoint inside processes to exchange messages with each other. The Active Ports differs from DP endpoint in that an active port is associated with the message handler designated by the receiver. It can achieve 9.5 μs latency and 93.7MB/s bandwidth by connecting two Pentium III PCs. However, the benchmark program reuses the same pre-pinned data buffer for the send and receive operations. As the send buffer has already been setup, the time spent in preparing of the frame header is not counted to the total time of the latency. Besides, in real applications, the reuse of same memory buffer may require the programmer to do their own data movement between the dedicate buffer and the user-space data buffers, which may introduce new overheads.

Fast Messages 2.x (FM 2.x) [11] is the core communication layer found inside the High Performance Virtual Machine (HPVM) package [5]. It has been implemented on Myrinet and Giganet, and runs on Windows NT. To send a 4-byte message, the measured latency is 9.63 μs on Myrinet and 14.7 μs on Giganet. For sending a 16-Kbyte message, the bandwidth reaches 100.53 MB/s on Myrinet and 81.46 MB/s on Giganet. The user interface of FM 2.x is quite different from DP. In DP, the API for the send and receive operations is quite straightforward, just involving one send and one receive function respectively. However, in FM 2.x, the send operation involves three functions - *FM_begin_message()*, *FM_send_piece()* and *FM_end_message()*, and the receive operation involves 2 functions - *FM_extract()* and *FM_receive()*, which may cause inconvenience for the users.

The GigaE PM [18] has been designed and implemented for parallel applications on clusters of computers on Gigabit Ethernet. It provides not only a reliable high bandwidth and low latency communication function, but also supports existing network protocols such as TCP/IP. By adopting a technique similar to the U-Net approach, PM provides a low-latency communication path through a virtual network object. This virtual network object encapsulates all the data structures needed to maintain a virtual channel between processes of the same parallel application, which include the fixed-size send and receive buffers and the associated queue structures. In its first implementation, it achieved 48.3 μs round-trip latency and 56.7 MB/s bandwidth on Essential Gigabit Ethernet NIC using Pentium II 400 MHz processor. GigaE PM II [19] has been implemented on Packet Engines G-NIC II for connecting Compaq XP-1000 workstations, each with 64-bit Alpha 21264 processor running at 500 MHz. The performance results show a 44.6 μs round-trip time for an eight-byte message. XP1000's four 64-bit CPU data buses, which support a 2.6 GB/s aggregate bandwidth, help GigaE PM II achieve 98.2 MB/s bandwidth for message length 1,468 bytes.

Virtual Interface Architecture (VIA) [23] is a novel communication architecture for clusters. It adopts the user-level communication paradigm and tries to standardize the interface for high-performance network technologies such as Gigabit networks. VI Architecture reduces network-related system processing overhead by creating the illusion of a dedicated network interface to multiple application programs simultaneously. Each VI represents a communication endpoint. Pairs of VIs can be logically connected to support point-to-point data transfer. M-VIA is a prototype software implementation of the VIA for Linux [15]. It supports the Fast Ethernet DEC Tulip chipsets as well as Packet Engines GNIC-I and GNIC-II Gigabit Ethernet cards. It is implemented as a set of loadable kernel modules and a user level library, and provides software doorbells with a fast trap for legacy hardware. The design of M-VIA adopts a highly modular approach similar to DP. M-VIA further abstracts the Kernel Agent defined in VIA specification into the M-VIA Kernel Agent and one or more M-VIA Device Drivers which are hardware dependent. Currently the evolution version of M-VIA, MVIA-2, is still under development. It will have much better support for high performance networks such as Giganet, Servernet II and Myrinet.

7. Conclusions

The design of DP exploits the underlying hardware architecture and operating system characteristics to effectively utilize the network and host system resources. We emphasize on the use of a realistic communication cost model so that designers can use it as a calibration tool to assess various design tradeoffs. The proposed communication model clearly delineates the characteristics of the communication network and allows the decoupling of communication overheads incurred in the host machine and the network, which in turn, allows a more controllable manner in scheduling communications.

Our messaging mechanisms significantly reduce the software overheads induced in the communication, as well as optimizing the memory utilization with an efficient buffer management scheme. Our implementation of the Directed Point communication system on two Ethernet-based networks achieves remarkable performance, e.g. the single-trip latency for sending 1-byte message is 16.3 μs and 20.8 μs on the Gigabit Ethernet and Fast Ethernet network respectively. With a standard PC, 97% of the Fast Ethernet network bandwidth is available to user applications, that corresponds to the achievable bandwidths of 12.2 MB/s and 24.32 MB/s for the uni-directional traffic and bi-directional traffic respectively.

Although DP achieves 63.6% of the Gigabit Ethernet bandwidth, (i.e. 79.5 MB/s), the performance evaluation result shows us that there are still rooms for us to further improve its performance, especially on the bi-directional bandwidth which is involved in many collective operations. For example, based on our communication model, it shows that the memory copies on the O_s and O_r parameters affect the overall performance in the Gigabit Ethernet network.

The secondary goal of DP is to achieve good programmability. A speedy communication system on its own does not guarantee to provide the efficient solutions to cluster applications. To address the programmability issue, DP provides an abstraction model and an API that has Unix-like I/O syntax and semantics. With the abstraction model, users can express those inter-process communication patterns on a directed point graph. This cleanly depicts the runtime system requirement of the target application and allows users to optimize their design strategies, e.g. allocating appropriate number of endpoints to realize this communication pattern. By providing an Unix-like I/O API, DP assists users in migrating their applications to the new environment, and facilitates the implementation of high-level communication libraries, e.g. MPI.

Indeed, the advance in the network technology has made the network capable of delivering packets in Gigabit speed or even higher. However, having the capability to deliver the packets in higher speed does not guarantee to achieve optimal performance while performing more complicated communication operations, such as many-to-one and many-to-many communications [21], since contention problems can happen in host node, network link and switch. The increasing growth of the network performance has stressed the need of a better resource utilization and system scalability on the design of contention-free collective communication algorithm.

REFERENCES

1. A. Bar-Noy and S. Kipnis, "Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems", in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 11-22, June 1992.
2. D. Becker, "A Packet Engines GNIC-II Gigabit Ethernet Driver for Linux", `http://beowulf.gsfc.nasa.gov/linux/drivers/yellowfin.html`
3. Cluster Computing White Paper, edited by Mark Baker, `http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/`
4. B. W. L. Cheung, C. L. Wang and K. Hwang, "JUMP-DP: A Software DSM System with Low-Latency Communication Support", in the *2000 International Workshop on Cluster Computing - Technologies, Environments and Applications (CC-TEA'2000)*, Las Vegas, Nevada, USA, June 2000.
5. A. Chien et al., "Design and Evaluation of an HPVM-based Windows NT Supercomputer", in The *International Journal of High-Performance Computing Applications*, Vol. 13, No. 3, Fall 1999, pp. 201-219.
6. G. Chiola and G. Ciaccio, "GAMMA: a Low-cost Network of Workstations Based on Active Messages", in *Proceedings of the 5th EUROMICRO workshop on Parallel and Distributed Processing (PDP'97)*, London, UK, January 1997.
7. G. Chiola and G. Ciaccio, "Active Ports: A Performance-oriented Operating System Support to Fast LAN Communications", in *Proceedings of Euro-Par'98, Southampton*, UK, September 1-4, 1998. LNCS 1470, Springer.
8. G. Ciaccio and G. Chiola, "GAMMA and MPI/GAMMA on Gigabit Ethernet", to appear in the *7th EuroPVM-MPI*, Balatonfured, Hungary, September 10 - 13, 2000.
9. D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation", in *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.
10. T. Heywood and C. Leopold, "Models of Parallelism", *Technical Report CSR-28-93*, Department of Computer Science, University of Edinburgh, 1993.
11. M. Lauria, S. Pakin and A. Chien. "Efficient Layering for High Speed Communication: Fast Messages 2.x.", in *Proceedings of the 7th High Performance Distributed Computing Conference (HPDC7)*, July 28-31, 1998.
12. C. M. Lee, A. Tam, and C. L. Wang, "Directed Point: An Efficient Communication

Subsystem for Cluster Computing", in *The International Conference on Parallel and Distributed Computing Systems (IASTED)*, October 1998.

13. P. Marenzoni, G. Rimassa, M. Vignali, M. Bertozzi, G. Corte and P. Rossi, "An Operating System Support to Low-Overhead Communications in NOW Clusters", in *Proceedings of Communication and Architectural Support for Network-Based Parallel Computing (CANPC97)*, San Antonio, Texas, February 1997.
14. Message Passing Interface Forum, University of Tennessee, Knoxville, "MPI: A Message-Passing Interface Standard", in the *International Journal of Supercomputing Applications*, Volume 8, Number 3/4, 1994.
15. M-VIA: A High Performance Modular VIA for Linux, National Energy Research Supercomputer Center at Lawrence Livermore National Laboratory, `http://www.nersc.gov/research/FTG/via/`
16. S. Pakin, V. Karamcheti, A. A. Chien. "Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs", in *IEEE Concurrency*, vol.5, (no.2), April-June 1997, pp.60-72.
17. L. Prylli and B. Tourancheau. "BIP: a New Protocol Designed for High Performance Networking on Myrinet", in *Workshop PC-NOW, IPPS/SPDP98*, Orlando, USA, 1998.
18. S. Sumimoto, H. Tezuka, A. Hori, H. Harada, T. Takahashi, and Y. Ishikawa, "The Design and Evaluation of High Performance Communication using a Gigabit Ethernet", in *International Conference on Supercomputing '99*, June 1999, pp. 243-250.
19. S. Sumimoto, A. Hori, H. Tezuka, H. Harada, T. Takahashi and Y. Ishikawa, "GigaE PM II: Design of High Performance Communication Library using Gigabit Ethernet", `http://pdswww.rwcp.or.jp/db/paper-J/1999/swopp99/sumi/sumi.html`
20. A. T. C. Tam and C. L. Wang, "Realistic Communication Model for Parallel Computing on Cluster", in the *First International Workshop on Cluster Computing*, August 11, 1999, pp. 92-101.
21. A. T. C. Tam and C. L. Wang, "Contention-Free Complete Exchange Algorithm on Clusters," to appear in *Cluster 2000 Conference,* Nov 28 - Dec 2, 2000, Technische Universität Chemnitz, Saxony, Germany.
22. L. G. Valiant, "A Bridging Model for Parallel Computation", in *Communication of the ACM*, August 1990, Vol. 33, No. 8, pp. 103-111.
23. VIA - Virtual Interface Architecture. `http://www.viarch.org`
24. T. Von Eicken, D. E. Culler, S. C. Goldstein and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", in *The 19th Annual International Symposium on Computer Architecture*, Gold Coast, Qld., Australia, May 1992, pp.256-266.
25. T. Von Eicken, A. Basu, V. Buch and W. Vogels, "U-Net: A User-level Network Interface of Parallel and Distributed Computing", in *Proc. of the 15th ACM Symposium of Operating Systems Principles*, vol. 29, (no.5), Dec. 1995, pp. 40-53.