J Supercomput (2006) 38:261–278
DOI 10.1007/s11227-006-8443-6

High-performance message striping over reliable transport protocols

Nader Mohamed · Jameela Al-Jaroodi · Hong Jiang · David Swanson

Abstract This paper introduces a high-performance middleware-level message striping approach to increase communication bandwidth for data transfer in heterogeneous clusters equipped with multiple networks. In this scheme, concurrency is used for the striping process. The proposed striping approach is designed to work at the middleware-level, between the distributed applications and the reliable transport protocols such as TCP. The middleware-level striping approach provides flexible, scalable, and hardware-, network-, and operating systems-independent communication bandwidth solution. In addition, techniques to enhance the performance of this approach over multiple networks are introduced. The proposed techniques, which minimize synchronization contention and eliminate the striping sequence header, rely on the features of a reliable transport protocol such as TCP to reduce some of the concurrent striping overhead. The techniques have been implemented and evaluated on a real cluster with multiple networks and the results show significant performance gains for data transfer over existing approaches.

Keywords Scalable network bandwidth · Network services · Socket · Parallel data transfer and striping

N. Mohamed · J. Al-Jaroodi (✉)
Electrical and Computer Engineering Department, Stevens Institute of Technology, Hoboken, NJ 07030
e-mail: jaljaroo@stevens.edu

N. Mohamed
e-mail: nmohamed@stevens.edu

H. Jiang · D. Swanson
Department of Computer Science and Engineering, University of Nebraska - Lincoln,
Lincoln, NE 68588-0115

H. Jiang
e-mail: jiang@cse.unl.edu

D. Swanson
e-mail: dswanson@cse.unl.edu

1 Introduction

Many research laboratories and organizations have multiple high performance heterogeneous machines for different application purposes. The heterogeneity in these machines usually lies in their difference in architecture, operating system as well as processing, memory, storage, and communication capacities. There are increasing interests in utilizing hardware and software resources available among a collection of existing heterogeneous systems to implement high-performance, scalable clusters capable of executing high performance applications such as multimedia, visualization, distributed data mining, and scientific simulations. These applications require scalable and high processing power, in addition to a reliable and scalable high-bandwidth communication infrastructure for high-volume and high-speed data access. Most clusters are equipped with multiple networks that connect all or some nodes. Each node has multiple network interface cards, of which each has a unique IP address. Most of the existing data transfer mechanisms on clusters use TCP/IP protocols, thus requiring a fixed IP address to identify the machine. This leads to the necessary requirement that a given application use a single IP address on each node to establish connections with other nodes, preventing the application from utilizing other available networks. This implies that the bandwidth available to any application is bounded by the bandwidth available on the single network interface associated with the assigned IP address. Thus, even if there are multiple networks connecting the nodes of the cluster, only one of them will be utilized for a given application.

There have been some efforts made to utilize multiple networks to enhance the communication performance. One important approach is the striping technique [4], which can be implemented at different network protocol levels by distributing incoming packets among the available network interfaces. One example of such efforts is channel bonding [2], which is implemented at the kernel level and provides high throughput for the applications. However, this technique is hardware-, network-, and operating system-dependant, and has some limitations in flexibility and requires some hardware and addressing configurations.

In this paper, we introduce a high-performance, middleware-level concurrent message striping technique that executes over reliable transport protocols such as TCP. This striping technique provides a scalable network bandwidth and portability solution for data transfer among heterogeneous systems. It provides a solution that reduces message transfer time and facilitates dynamic load balancing among the underlying multiple networks. In addition, we introduce some techniques to enhance the performance of this middleware-level concurrent striping implementation of message transfer over multiple networks. These techniques rely on the features of a reliable transport protocol such as in-order and guaranteed delivery of packets to significantly reduce some of the striping overhead.

In the rest of this paper, we start with a discussion of some background information on heterogeneous clusters, network striping, and channel bonding, in Section 2. Section 3 describes concurrent message striping over reliable transport protocols and measures its performance and overhead. Then, in Section 4, we introduce some techniques to reduce the concurrent striping overhead for the dual-channel case by utilizing the features of the reliable transport protocol. Section 5 discusses an extension of these techniques to reduce the striping overhead for situations with multiple channels (i.e. more than two networks). Experimental measurements and evaluations of the proposed approaches are included in Sections 3 through 5. Section 6 discusses related work and Section 7 concludes the paper.

2 Background

One constant factor shared by current heterogeneous clusters is the use of TCP/IP protocol, which provides a common base for any services to be provided on heterogeneous systems. Thus, allowing efficient utilization of the available multiple heterogeneous networks and facilitating better access to the networks and higher bandwidth. As an example, consider a scenario where a machine with a single Gigabit Ethernet (GE) card is connected through a high bandwidth network to another machine with multiple Fast Ethernet (FE) cards. Such a network connection will limit the achievable bandwidth to that of a single FE card, even though theoretical peak bandwidth on either machine is much higher.

Efforts are being made to utilize existing multiple networks to enhance the performance of communication. One important approach to provide such utilization is the striping technique [4]. Striping is well known and understood in with the context of storage systems, for example, the redundant arrays of inexpensive disk (RAID) architecture [10]. Network striping can be implemented at different network layers, such as physical, data link, network, transport, and applications layers. In addition, it can be implemented as network services used by applications requiring high network bandwidth. The network services are usually implemented at the middleware layer between the transport protocols and applications. Striping at lower network levels has been implemented in a number of research projects, such as the IBM project for striping PTM packets [15], and the Stripe, which is an IP packets scalable striping protocol [1]. However, these techniques are network and/or system dependent.

Another example of using striping in clusters is channel bonding [2], which is implemented at the kernel level and it provides high bandwidth and high throughput for the applications. In general, channel bonding cannot enhance the performance for very small messages that are less than the maximum transfer unit (MTU); however, it can increase the network throughput as well as reduce the transfer time for medium to large messages [5]. Since channel bonding is implemented in the lower network layers, it requires extra addressing and hardware configurations [3]. For example:

- All NICs on a node must be of the same type and have the same MAC address.
- It requires separate switches to connect the different NICs on the nodes (to avoid MAC confusion) or an advanced switch that can be segmented into virtual LANS.
- Communication from non-bonded nodes to bonded nodes is too slow, thus, all nodes on a cluster must be bonded. Therefore, all nodes must have same number of network interfaces connected to the same number of networks.
- Bonded NICs cannot be accessed individually, thus limiting their utilization. This limitation has significant impact on the performance of cluster applications, which require intensive but relatively small message communication.
- Channel bonding is available only for Linux systems with multiple Ethernet interfaces and networks.

These limitations make channel bonding less suitable for clusters with heterogeneous resources.

To overcome the limitations of existing scalable bandwidth solutions such as channel-bonding, MuniSocket was designed and implemented as a network service at the middleware layer. MuniSocket only needs to use standard API of reliable transport protocol. It overcomes many of the limitations imposed by existing techniques and provides desirable features. MuniSocket utilizes concurrent message striping at the middleware-level. Concurrent message striping at the middleware-level provides a scalable, portable, and flexible solution for increasing bandwidth among heterogeneous systems. It uses the standard TCP APIs that are

available in all types of operating systems to provide a scalable bandwidth and portable solution. One example of using MuniSocket is in information technology departments which have heterogeneous server machines such as AS/400, RS/6000, and Window NT. These servers have applications that collectively serve and support the company business objectives. Usually there is a need to make these servers to exchange data. At the same time, the communication bandwidth needs for these servers are vary. As a solution for this problem, we can install multiple network interface cards on each machine and use MuniSocket for communication among them, thus increasing usable bandwidth and throughput. More information about scalability and flexibility of MuniSocket can be found in [13].

3 Concurrent message striping over reliable transport protocols

Concurrent message striping at the middleware-level provides a scalable, portable, and flexible solution for increasing bandwidth among heterogeneous systems. It uses the standard TCP APIs that are available in all types of operating systems to provide a scalable bandwidth and portable solution. In the rest of the section, the architecture of MuniSocket and the operation of concurrent message striping over TCP are discussed in 3.1. In addition, performance and overhead of the concurrent striping are discussed in 3.2.

3.1 MuniSocket operation

MuniSocket is multithreaded, using kernel threads, with one sending and one receiving thread per communication channel, and a counter. MuniSocket divides large messages generated by the application into fragments that are then transmitted using any transport protocol. Multiple concurrent threads are used to process the fragments for transmission. A sending thread is responsible for handling the transmission of some fragments on a specific communication channel, while the receiving thread is responsible for handling received fragments and placing them into the appropriate place in a receiving buffer provided by the user.

The counter provides a sequence number that identifies a fragment in the user message. Each sending thread tries to take a number, i, from the counter, and sends both the sequence number i, as a header, and the corresponding data fragment in the sender memory to the corresponding receiving thread in the receiving end over the corresponding NICs and networks. In this approach, there are no extra data copies other than that made by TCP operations for both the sending and receiving sides. The transfer is done completely in parallel during sending at the sender side, transferring through the networks, and receiving and copying at the receiver side.

MuniSocket relies on the reliable transport protocol's features to handle many end-to-end issues such as reliability and flow and congestion control of the transferred message fragments [13]. In addition, it depends on the reliable transport protocol to divided a large fragment to a number of packets. Both flow and congestion control mechanisms of the reliable transport protocols are utilized to implement the load-balancing algorithm among the sending threads. The striping technique is used to enhance the performance on loaded and heterogeneous networks. This method is devised to balance the load for large messages so that slower networks will not slow down the overall message transfer. The message is fragmented into a large number of small sub-messages that are transmitted over the available channels, along with a fragment header containing the fragment number. Load balancing is achieved by having threads connected to lightly loaded networks process more fragments while threads connected to heavily loaded channels are blocked for longer periods by the congestion and

flow control mechanisms of the reliable transport service used. More information about implementation and operation of basic MuniSocket can be found in [13].

3.2 The performance and overhead of basic MuniSocket

To evaluate the performance gains of the basic implementation of MuniSocket, a number of experiments were conducted. The main program used to measure the achievable round-trip transfer time is the classical ping-pong benchmark [6–8], which sends messages back and forth between machines and determines the round-trip transfer times. All experiments in this paper were conducted multiple times (10 to 1000) and we took the average results. In addition, we repeat the previous process 10 to 50 times and we took the peak of the averages. This method allows us to avoid considering the results of experiments that are impacted by other load on the networks when we need not consider it. The frequency of repetition depends on the message size. For small message, we use high repetitions. All experiments were conducted on Sandhills, a 24-node cluster, where each node contains two AthlonMP 1.4 GHz processors with 256 KB cache and 1 GB RAM. The nodes were equipped with two Fast Ethernet (FE) interface cards that are connected to two FE networks. The experiments were designed to measure the round trip time (RTT) for the single network Socket and for MuniSocket using two FE networks, with the effective bandwidth being derived from the RTT as follows:

$$\textit{Effective Bandwidth}\ (\textit{Mbps}) \;=\; (8^{*}\ \textit{Message size}/10^{6})/(\textit{RTT}/2) \tag{1}$$

The communication performance in terms of effective bandwidth for TCP over FE (TCP100) and MuniSocket over two FE cards (MuniSocket 2 * 100) is shown in Fig. 1. The results show low effective bandwidth values for MuniSocket 2 * 100, as compared to TCP100, for messages smaller than 32 KB. However, as the message size increases the gain for MuniSocket 2 * 100 becomes evident and clearly shows the benefits of this method. The effective bandwidth reaches 186.4 Mbps for a message of size 4 MB. In addition, the CPU consumption (utilization) was measured during the transfer of a very large message. While the CPU utilization increased from 13.26% to 26.16% per processor, the effective bandwidth

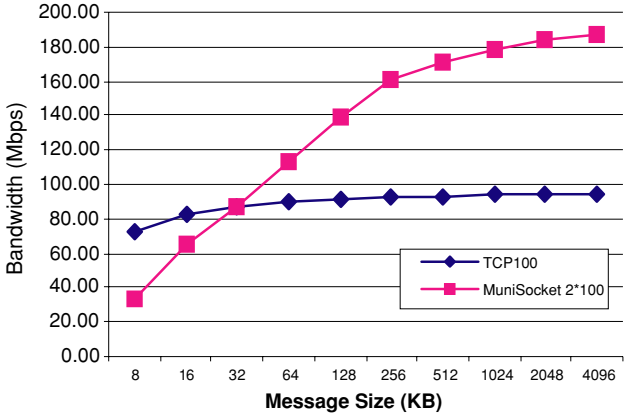

Fig. 1 Effective bandwidth using fast ethernet

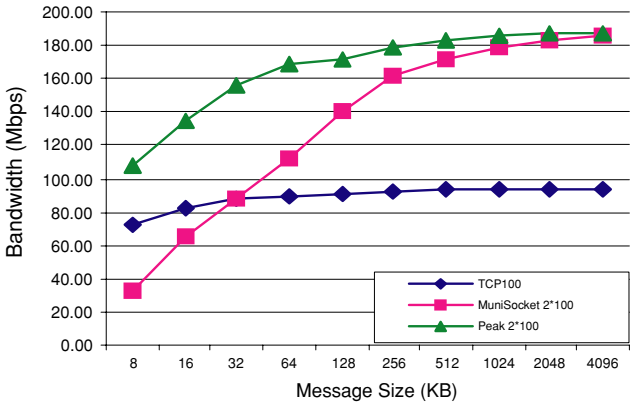

Fig. 2 Peak effective bandwidth using two fast ethernet networks

has also doubled. This makes MuniSocket useful in applications that need high and scalable bandwidth communications, but can afford to lose some processing time in return. Examples of these applications are ones that use blocking (synchronous) data transfer, data replications, and bulk data transfer.

The performance of concurrent message striping is dependent on two factors. The first is the efficiency of the protocol and the second is based on the performance of the system resources such as processor speed, memory and I/O bus bandwidths, and bandwidths of network interfaces and links. While the protocol is designed to utilize these resources, it induces reasonable but necessary overhead in message transfers. The overhead is generated from preparing fragment headers and thread contention on the shared counter, which is synchronized to maintain consistency. To measure the actual utilization of MuniSocket from available resources, a modified version of MuniSocket was written to transfer the messages by logically partitioning the messages into equal parts and transferring them over two unloaded FE interfaces. The modified version does not do any striping-related operations such as synchronization and does not add headers. All sending and receiving threads work independently and there is no need for a fragment counter. This modified version measures the maximum achievable performance (Peak $2 * 100$) on the available resources; the results are shown in Fig. 2.

The utilization of the basic MuniSocket protocol, normalized to the maximum achievable performance, was measured (see Fig. 3). Although MuniSocket has good performance for large messages, the current protocol overhead is the main contributing factor in message transfer time for small- to medium-size messages. The utilization is only 30% for 8 KB messages and increases slightly as messages grow larger. The performance does not get close to the peak until the message size reaches 0.5 MB. The main culprit in the overhead is the fragment headers and the sending threads contention on the fragment counter. The next two sections develop techniques to reduce both kinds of overhead.

4 Techniques for reducing concurrent striping overhead for dual-channel case

In this section, we propose two techniques to reduce the concurrent transfer overhead for the dual-channel case. The first focuses on reducing the resource contentions in the concurrent

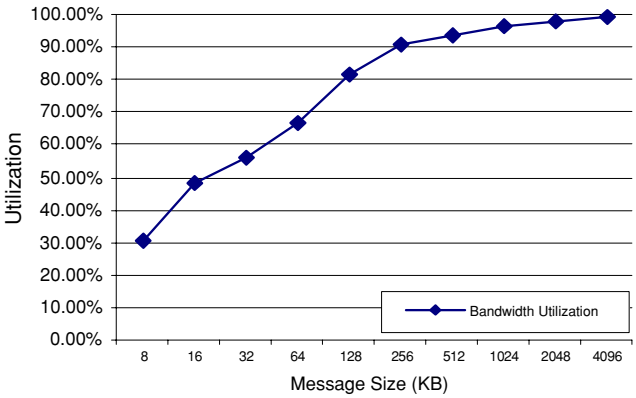

Fig. 3 Normalized bandwidth utilization of basic MuniSocket

striping process. The second technique relies on utilizing the functions of the reliable transport protocol to eliminate the need of fragment sequence headers.

4.1 The techniques

The shared counter of MuniSocket provides sequence numbers for the sending threads starting from 0 to $n-1$, with n being the number of fragments in the message. If a sender processes and sends a message that is divided into 8 fragments over dual networks with dynamic load changes, the fragment distribution can be of different combinations. For example, the first sending thread processes fragments f_0, f_3, f_4, and f_6, while the second sending thread processes fragments f_1, f_2, f_5, and f_7. Fragments are sent with their sequence numbers as part of the fragment header. Receiving threads process the header to get the sequence number to receive the fragment to the corresponding buffer position.

The first step to eliminate the counter synchronization contention is to change the fragment processing order. With the dual-channel case, for example, there are two sending threads at the sender side and two receiving threads at the receiver side. The first sending thread works from left to right on the message buffer while the second sending thread works from right to left, as illustrated in Fig. 4. The concurrent fragment sending process stops as soon as the two threads meet (indicating the end of message transfer). For example, with n fragments, the first thread sends fragments $f_0, f_1, f_2, \ldots, f_m$ in sequence and the second sending thread sends fragments $f_{n-1}, f_{n-2}, \ldots, f_{m+1}$ in sequence. The proposed technique ensures that the network with higher bandwidth processes more fragments, while the network with lower bandwidth processes fewer fragments. The value of m depends on the load and available bandwidth on both networks. Therefore, this method optimally solves the problem of load balancing between dual networks.

Fig. 4 Fragment processing direction by sender threads in dual networks

Since the sending threads process fragments from different directions, there is no need for the shared counter among the sending threads. Each sending thread maintains a separate counter. The first sending thread maintains an incremental counter that starts from zero, while the second maintains a decremental counter that starts from the last fragment number. An array of status values (one per fragment) can be used to synchronize the point where the sender threads stop processing more fragments. Each entry in the array represents a fragment status. The value 0 represents unsent fragment while value 1 represents in-transit or sent fragment. Each sending thread working from a different direction tries to get and change the fragment status value from the array. As soon as the entry is granted to a sending thread the value is changed to 1. Since both sending threads start from different directions for message fragments and the status array, they can stop as soon as they reach a sent fragment. In order to implement the fragment granting mechanism for the sending threads, a lock is used for each entry to maintain the consistency of the granting process. The number of locks is equal to the size of the array. Another option is to use the *test-and-set* method for the entries to guarantee atomicity.

Using a shared counter among sending threads to provide sequence numbers for fragments to be processed has a high possibility of contention and eventually high level of thread switching. Moreover, contention over the counter may occur every time the sending threads try to get a new fragment to process. In addition, the probabilities of contention increase further as the number of fragments is increased to achieve better load balancing. The status array can reduce threads switching and contention significantly. Using the status array the contention may happen only at the end when both sending threads compete for the last unsent fragment, in other words, when the two sending threads meet. In addition, using the status array, the fragments can be made as small as needed without increasing the contention overhead.

Since concurrent striping is done over reliable transport protocols, the fragments' first-in-first-out (FIFO) order is preserved for each sending and receiving thread pair. That is, if sending thread i on the first machine sends fragments f_0, f_3, f_4, and f_6 to the receiving thread i on the second machine, these fragments will be received in the same order f_0, f_3, f_4, and f_6. Although some fragments might be lost or corrupted, the underlying reliable transport protocol mechanism, such as TCP, maintains the reliability and FIFO order of fragment/packet delivery. The reliability mechanism ensures that there is no packet loss or corruption. The reliability and FIFO properties of a reliable transport protocol such as TCP eliminate the need for fragment sequence headers by ensuring that processing and sending of the fragments are done in sequence.

The technique can be implemented by making each sending and receiving thread pair maintain a counter. Each counter is duplicated at the sending thread and at the receiving thread. In addition, two control messages *Start* and *End* are used by the sending thread to synchronize the counters and to indicate the end of message transmission. The sending thread sends a control message *Start* that contains the first fragment number *firstFragement* and counter mode *counterMode* to the corresponding receiving thread. The value of *firstFragement* is the fragment number of the first fragment that the sending thread will be sending. The variable *counterMode* can take one of two values, either "*increment*" or "*decrement*". When *counterMode* is "*increment*" the counter increases (i.e., moving from left to right on the message buffer), and if it is "*decrement*" the counter decreases (i.e., moving from right to left on the message buffer). Using these values, sending-receiving thread pairs can synchronize their fragment counters without exchanging fragment sequence numbers. When processing a message each sending thread sends a *Start* message indicating its starting fragment number

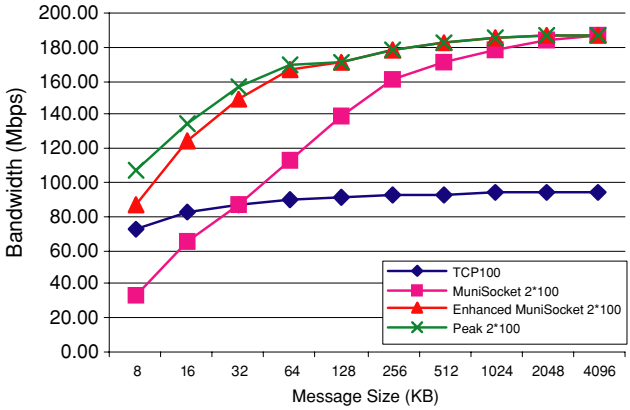

Fig. 5 Effective bandwidth of TCP, MuniSocket, and enhanced MuniSocket

and direction and when the sending threads meet, they both send the *End* message to indicate the end of message transmission.

4.2 The performance measurements

Experiments were conducted on the same cluster and using the same benchmark described in Section 3 to evaluate the performance of the Enhanced MuniSocket over two Fast Ethernet (FE) networks. With the proposed new mechanism in MuniSocket, the overall performance improved noticeably (see Fig. 5). From the experimental results we see the effect of removing the fragment header and eliminating the contention on the shared fragment counter. While the basic MuniSocket (MuniSocket 2 * 100) only achieved speedup with messages of size 32 KB and larger, Enhanced MuniSocket achieved speedup for messages of size 8 KB and beyond. With a message of size 8 KB, standard TCP socket on a single FE (TCP100) achieved a bandwidth of 72.26 Mbps and Enhanced MuniSocket achieved 87.37 Mbps effective bandwidth. Likewise, with a message of size 64 KB, TCP100 achieved 89.32 Mbps, basic MuniSocket achieved 112.75 Mbps, and Enhanced MuniSocket achieved 166.44 Mbps. The speedup of Enhanced MuniSocket for a 64 KB message is around 1.86 with respect to TCP100 and 1.47 with respect to basic MuniSocket. With a large message of size 2 MB, TCP100 achieved a bandwidth of 93.92 Mbps, the basic MuniSocket achieved 183.36 Mbps, and Enhanced MuniSocket achieved 186.41 Mbps. The speedup of Enhanced MuniSocket for a 2 MB message is around 1.98 with respect to TCP100 and 1.02 with respect to MuniSocket. This shows that the overhead due to fragment headers and thread contention is high at medium messages but largely hidden with large messages. Figure 6 shows the bandwidth utilization of both the basic MuniSocket and Enhanced MuniSocket, normalized to the maximum achievable peak. While the effective bandwidth of the basic MuniSocket approaches the maximum achievable peak with a message of size 0.5 MB, Enhanced MuniSocket can achieve the peak with a much smaller message of size 16 KB.

Another set of experiments was conducted to measure the performance of the load balancing mechanism in Enhanced MuniSocket. Two FE networks were used in these experiments. The first network had no load, while the second had some load generated by messages of size 32 KB being exchanged frequently between the two nodes. Figure 7 shows the average effective bandwidth achieved with the standard TCP socket, using the first (Unloaded TCP100)

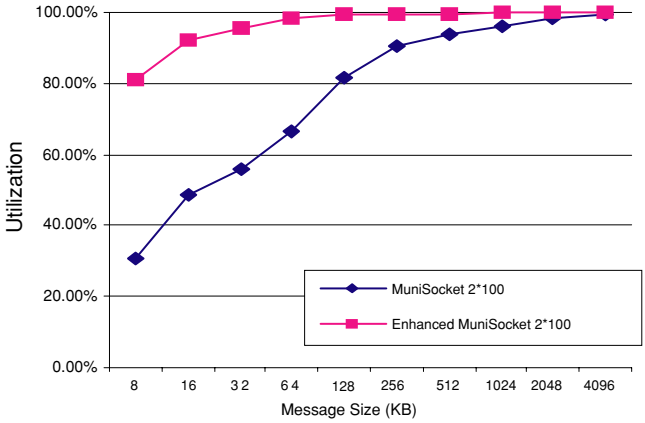

Fig. 6 Bandwidth utilization of MuniSocket and enhanced MuniSocket

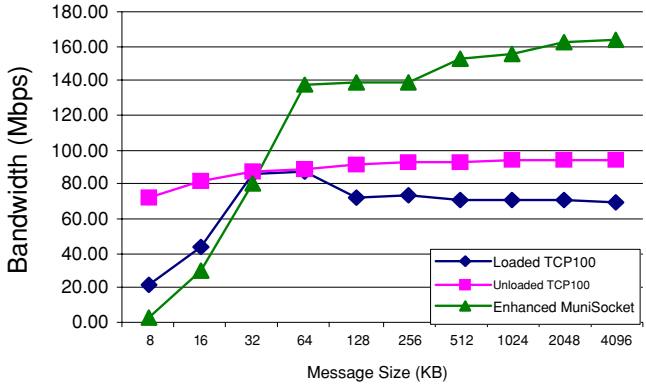

Fig. 7 Effective bandwidth of TCP and enhanced MuniSocket on loaded networks

and the second (Loaded TCP100) network, respectively. In addition, it shows the average effective bandwidth achieved with Enhanced MuniSocket that utilizes the available bandwidth in both networks. Enhanced MuniSocket is shown to achieve performance speedups with messages of size around 40 KB and larger, and to have dynamic load balancing ability. The results also show that while the loaded network provides less than 75 percent of its peak bandwidth, MuniSocket is still able to achieve high bandwidth gain with dynamic load balancing. In other words, with a 2 MB message, for example, an average bandwidth of 82.123 Mbps was obtained with either network interface when not using MuniSocket. However, using Enhanced MuniSocket, an average bandwidth of 161.84 Mbps was achieved using both interfaces simultaneously.

5 Reducing overhead for multiple-channel cases

The techniques described in Section 4 provide an efficient solution for removing fragment sequence numbers and reducing synchronization contention. However, the solution,

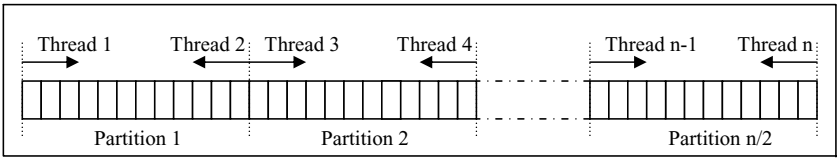

Fig. 8 Multiple network solution

as described above, only works for the dual-channel case. In this section, the technique is extended to scale to more than two channels.

5.1 The technique

For simplicity and without loss of generality, we will deal with an even number of networks. However, the same method described in this section can be easily adjusted for an odd number of networks. Assume that there are n networks and n sending threads, where n is even. The sending threads are divided into $n/2$ pairs of threads. In addition, the message is divided into equal-size fragments. The fragments can be further divided into $n/2$ partitions as shown in Fig. 8. Each pair of sending threads can handle one partition using the same technique as described in Section 4. The size of the partitions can be divided so that they are proportional to the total channel bandwidths or speeds of the associated threads. Any pair that finishes from their current partition can help other pairs in their partitions in a recursive way. Let us call the pair that finishes their current partition as a *free pair* while the pair that does not finish their current partition as a *busy pair*. Each *free pair* consists of an *incremental free thread*, which has worked from left to right and a *decremental free thread*, which has worked from right to left. Each *busy pair* also consists of an *incremental busy thread*, which works from left to right and a *decremental busy thread*, which works from right to left. The partition of the selected *busy pair* will be divided further into two parts, *left part* and *right part*.

After dividing the selected partition into two parts, one of the free threads is associated with the *incremental busy thread* to form a new thread pair for the *left part* while the second free thread is associated with the *decremental busy thread* to form another new thread pair for the *right part*. The *first free thread* starts to work in the *decremental mode* from right to left on the *left part* while the *incremental busy thread* continues its operation direction from left to right on the same part. In addition, the *second free thread* starts to work in the *incremental mode* from left to right on the *right part* while the *decremental busy thread* continues its operational direction from right to left on that part. This process is repeated until there are no other partitions needing help. In each recursive step, either a modified *Start* or *End* control message is sent. The modified *Start* control message indicates that the fragment processing will start from a different position with a different counter mode. The *End* control message indicates that there are no more fragments to be sent. The *End* control messages are only sent by the sending threads at the end of the message processing. Therefore, each sending thread may send multiple modified Start control messages; however, it sends a single *End* control message. All sending threads process the fragments until all are sent. For the case of an odd number of networks, the above method can be adjusted such that one partition is handled by one network. The other steps are also adjusted accordingly in a minor way.

To have better load balancing among the thread pairs processing their new parts, three greedy strategies are used. These strategies aim to reduce the number of recursive steps, thus reducing sequence numbers (modified *Start* messages) to be sent. Sending threads

handle the preparation and sending of the fragments through the network; therefore, thread-processing time is defined to include all the time involved in this process, including the fragment transmission over the network. Thus, a slow thread generally reflects a slow network associated with it. In the first strategy, where multiple *busy pairs* need help, we select the *busy pair* with the maximum number of unsent fragments in its partition to be helped. The second strategy associates the less busy thread (which sent fewer fragments than its partner) from the selected *busy pair* with the busier thread (which sent the most fragments than its partner) from the *free pair* and vice versa. This strategy takes into consideration the recent history of the network load by making each thread t_i maintain the number C(t_i) of fragments it processed. This number represents the contribution of the network/thread to the processing of the message fragments.

The third strategy divides the unfinished partition based on the recent progress of the four threads involved in the process. Let t_{il}, t_{dl}, t_{ir}, and t_{dr} denote the incremental thread on the *left part*, the decremental thread on the *left part*, the incremental thread on the *right part*, and the decremental thread on the *right part,* respectively. The threads t_{il} and t_{dr} are from the original *busy pair* that needs help while threads t_{dl} and t_{ir} are from the original *free pair*. The threads t_{il} and t_{dr} do not need to change their next fragment numbers. They continue their operations and directions. However, we need to assign new fragment numbers for threads t_{dl} and t_{ir} which are offering help. We assign the next fragment numbers for threads t_{dl} and t_{ir} based on the recent progress of the four threads involved in the process, we need to divide the unfinished fragments in the selected unfinished partition into two parts with sizes proportional to the recent speeds of their associated thread pairs. Let us define a new functions $P(t)$ to be the next fragment number that thread t needs to process. The number of unsent fragments in any part is the next fragment number that the decremental thread needs to send minus the next fragment number that the incremental thread needs to send plus 1.

$$Unsent\ fragments\ =\ P(t_{dr}) - P(t_{il}) + 1 \tag{2}$$

We need to divide this number of fragments into *left part* and *right part*, with sizes proportional to their threads' processing speeds. Let the recent contribution of the *left part* threads be represented by $C(t_{il}) + C(t_{dl})$ while the contribution of the *right part* threads be calculated as $C(t_{ir}) + (t_{dr})$. Therefore we can calculate the number of unfinished fragments for the *left part*, *U(left part)*, as:

$$U(left\ part) = unsent\ fragements\ \times\ \frac{Contibution\ of\ left\ part\ threads}{Contribution\ of\ the\ four\ threads} \tag{3}$$

$$U(left\ part) = (P\,(t_{dr}) - P\,(t_{il}) + 1) \times \frac{C\,(t_{il}) + C\,(t_{dl})}{C\,(t_{il}) + C\,(t_{dl}) + C\,(t_{ir}) + C\,(t_{dr})} \tag{4}$$

The value of *U* (*left part*) can be a real number but it should be truncated to an integer value.

We have:

$$Unsent\ fragments = U(left\ part) + U(right\ part) \tag{5}$$

From Eq. (5), we can calculate the unfinished fragments for the right part, *U*(*right part*), by the following equation:

$$U(right\ part) = unsent\ fragment - U(left\ part) \tag{6}$$

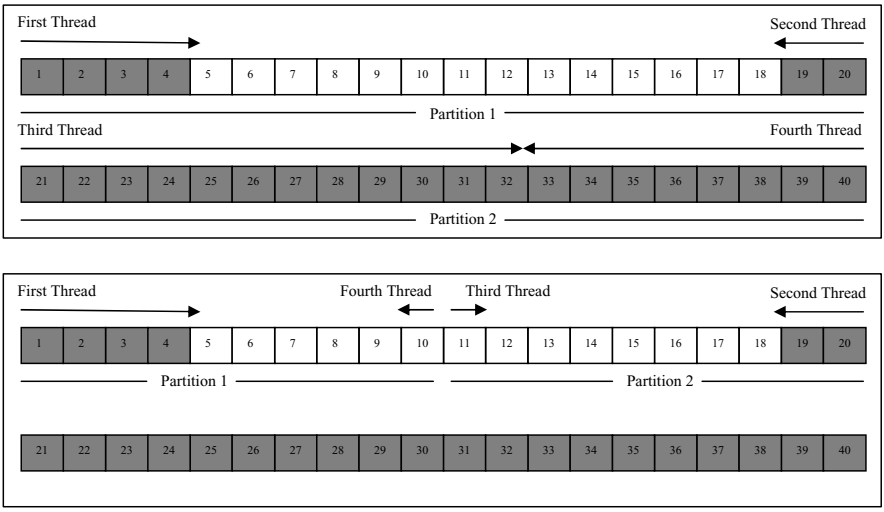

Fig. 9 (a) Top: a snapshot of four threads executing two partitions, the second partition is completed by the third and fourth threads while the first partition is still being processed. (b) Bottom: the rearrangement of partitions and threads when the third and fourth threads start helping the first and second. The third thread gets more fragments to process based on the partitioning heuristics

$$U(\textit{right part}) = (P(t_{dr}) - P(t_{il}) + 1) - U(\textit{left part}) \tag{7}$$

From these equations, the next fragment numbers for threads t_{dl} and t_{ir} can be calculated as:

$$P(t_{dl}) = P(t_{il}) + U(\textit{left part}) - 1 \tag{8}$$

$$P(t_{ir}) = P(t_{dr}) - U(\textit{right part}) + 1 = P(t_{dl}) + 1 \tag{9}$$

The threads t_{il} and t_{dr} do not need to change their next fragment numbers. They continue their operations and directions.

For example, a message of 40 fragments, as shown in Fig. 9(a), is transferred through four channels with equal latency and bandwidth. The message will be divided into two equal partitions and four threads will process it. The first and second threads work on the first partition while the third and forth work on the second partition. Consider a scenario where workloads on the networks are different such that the second partition has been sent while the first partition is still under processing. Now, consider the time immediately after the second partition is completed, the contributions of the first, second, third, and forth threads are 4, 2, 12, and 8, respectively, as shown in Fig. 9(a). Based on the second greedy strategy described above we will associate the forth thread with the first thread and third thread with the second thread, as shown in Fig. 9(b). This means that $t_{il} = t_1$, $t_{dl} = t_4$, $t_{ir} = t_3$, and $t_{dr} = t_2$. We have $C(t_{il}) = 4$, $C(t_{dl}) = 8$, $C(t_{ir}) = 12$, $C(t_{dr}) = 2$, $P(t_{il}) = 5$, and $P(t_{dr}) = 18$. We can calculate *U(left part)* from Eq. (4) *as follows:*

$$U(\textit{left part}) = \text{int}\left((18 - 5 + 1) \times \frac{4 + 8}{4 + 8 + 12 + 2}\right) = 6$$

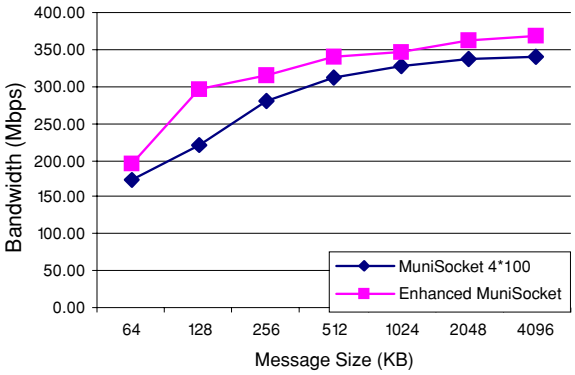

Fig. 10 Effective bandwidth of MuniSocket and enhanced MuniSocket on four fast ethernet networks

Thus we can calculate both $P(t_{dl})$ and $P(t_{ir})$ from Eqs. (8) and (9) by:

$$P(t_{dl}) = 5 + 6 - 1 = 10 \text{ and}$$
$$P(t_{ir}) = 10 + 1 = 11$$

5.2 The performance measurements and complexity analysis

Experiments were conducted on the same cluster and using the same benchmark described in Section 3.2 to evaluate the performance of the Enhanced MuniSocket over four FE networks. Two nodes were equipped with two additional FE interface cards each. Therefore, the two nodes each have four connected FE interface cards. Both basic MuniSocket (MuniSocket 4 * 100) and Enhanced MuniSocket (Enhanced MuniSocket) with four channels were experimentally evaluated. The results, in term of peek effective bandwidths, are shown in Fig. 10. The Enhanced MuniSocket provides better performance than the basic MuniSocket by a margin of 35% for messages of size 128 Kbyte and of 7.6% for messages of size 2 MB. These experiments also show that concurrent message striping in general provide a good scalability and flexibility solution to increase effective bandwidths for message transfers.

The synchronization contention on the shared counter and the overhead of sending the fragment sequence number headers are reduced significantly. While $O(n)$ sequence number headers need to be sent with the un-optimized concurrent message striping for a message with n fragments, only $O(1)$ *Start* control messages with startup sequence numbers and counter modes are needed with optimized concurrent message striping on dual-channel case. For the k-channel scenario, where $k > 2$ and $k << n$, we consider the best and average case scenarios. The best case is where all thread pairs finish their parts at the same time. In this case each thread needs to send only a single *Start* control message at the beginning. The complexity of the best case scenario is $O(k)$, Thus, for small k, the complexity is constant.

The average case analysis provides a good projection of the behavior of the proposed scheme in real-life situations in which the communication workload on the networks is dynamic. A simple approximation of average case scenario can be considered when the process of sending partitions consisting of j load-balancing rounds. In each round multiple recursive steps are performed by different thread pairs. In addition, in each round, we assume on average half of the thread pairs complete their fragments (thus completely sending full partitions) while the rest only complete on average half of their fragments (thus each remaining thread

pair still needs to send more fragments in the partition). This implies that a quarter of the total fragments are unsent at the end of the current round and still need to be processed in the following rounds. Therefore, j will be equal to $\log_4(n)$ rounds. In the first round, all k threads will each send a fragment sequence number header, while during the rest of the rounds only $k/2$ threads need to send sequence number headers in each round. This means that the number of *Start* control messages in the average case will be:

$$\textit{Number of Start Control Messages} = k + \frac{k}{2}((\log_4 n) - 1) \tag{10}$$

Since we have $k << n$, then $O(\log_4(n))$ *Start* control messages need to be sent in the average case. To verify our analysis, another set of experiments was conducted to count the number of *Start* control messages that are needed with the multiple-channel cases. In all experiments, a message of 2 MB was sent through four channels that represent four FE networks. The message was divided into 1024 equal size fragments where each fragment is 2 Kbyte. Two sets of experiments were conducted. In the first one, there is no other load on the networks. The average range of numbers of *Start* control messages for multiple 1000 runs was between 4 and 5. For the second set of experiments, the networks were randomly loaded. The average range of numbers of *Start* control messages for multiple 1000 runs in the second set of experiments was between 7.0 and 8.1. In both experiments, the variances of multiple results were very small, around 0.3. These numbers validate the above analysis. However, in the basic concurrent striping technique described in Section 3, 1024 fragment sequence headers need to be sent. In addition, the contention on the shared counter is reduced significantly by the same order.

6 Related work

Standard and advanced sockets provide either reliable transport services or unreliable transport services. Example of unreliable transport service is that provided by UDP socket. In [11] and [14], we proposed a UDP-based protocol for MuniSocket to utilize the existing multiple-network interfaces and single or multiple networks to transfer large messages in both local and wide area networks. In order to have a reliable transfer service with the UDP-based protocol, the protocol itself included a reliability mechanism over the unreliable multiple services. We introduced both reliability and fault tolerance mechanisms that utilize UDP. However, the reliability overhead makes the UDP-based multiple networks protocol perform well with bulk data transfers, while for small and medium messages, the reliability overhead becomes the dominant contributor to message latency. The UDP-based protocol also accommodates heterogeneous networks with different latency and bandwidth. In [12], we have developed analytical models for the performance properties of the combined heterogeneous networks.

GridFTP [9] uses multiple TCP streams to improve aggregate bandwidth over using a single TCP stream in wide area networks. In addition, GridFTP can transfer data from multiple servers where data is partitioned to improve performance. On the other hand, our approach uses concurrent message transfer to provide scalable bandwidth and load balancing for data-intensive applications on clusters connected by multiple interconnection networks. This is achieved by taking advantage of the physical and logical parallelism and the redundancy in interconnects. GridFTP and multiple streams use multithreaded solutions for multiple TCP streams. Using multithreading and striping for multiple TCP streams in wide area networks

can also benefit from utilizing the optimization method developed in this paper to reduce some of the overheads.

Some work [1, 15] has been done to minimize the overhead of striping at lower network layers. This work uses queues at the receiving ends of the channels to maintain synchronization between sender and receiver to reorder the packets. While our approach uses concurrent striping to utilize available networks to reduce the transfer time of a specific message generated by an application, other striping algorithms deal with differently sized packets coming from different applications to reduce the striping overhead and increase available network throughput. For example, [1] uses a distributed algorithm to maintain synchronization and a transformation of a fair queuing algorithm to provide fair load sharing among different packet sizes coming from upper layers.

Channel bonding [2], which uses striping at the data link (Ethernet) layer, does not use a sequence number. It relies on the higher layer protocols to handle the issues of packet ordering and reliability. Nevertheless, as mentioned earlier in Section 2, channel bonding has many limitations and hardware configurations that make it inflexible and introduce other forms of overheads. The main difference between MuniSocket and lower level striping is providing a mechanism to lower transfer time for specific message transmissions as opposed to higher throughput for all messages in the lower-level striping techniques. For example, small messages cannot benefit from striping; therefore, they can be transferred using standard communication mechanisms without the striping. Another difference from the striping technique, which distributes incoming packets from different applications among the available interface(s), is that our approach divides the messages generated by an application into fragments that are then transmitted in parallel using a reliable transport protocol.

An example of a channel bonding benchmark on a cluster called JAZZ [5] shows an increase in bandwidth from 86.8 Mbps to 160.4 Mbps for 1 MB message and from 86.88 Mbps to 162.08 Mbps for 4 MB message on two FE networks. Figure 11 shows the calculated speedup of two FE networks achieved in channel bonding (using JAZZ) compared to the speedup achieved by the Enhanced MuniSocket. The result shows that both channel bonding and the Enhanced MuniSocket cannot achieve any speedup with messages smaller than the maximum transfer unit (MTU). Channel bonding performs better for messages with size smaller than 12 KB, where the average speedup is 1.23. However, the Enhanced MuniSocket has a better performance for all messages larger than 12 K. Although MuniSocket

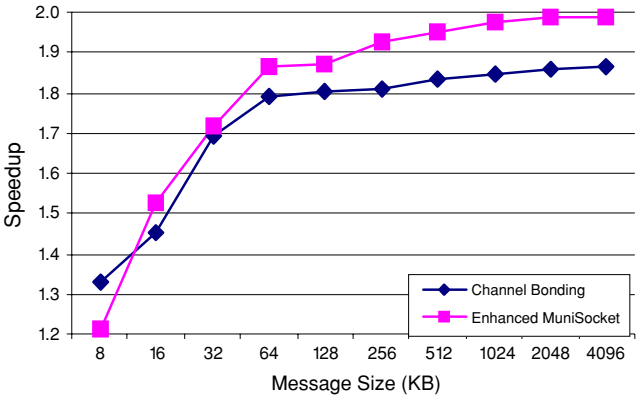

Fig. 11 Channel bonding and enhanced MuniSocket comparison

is implemented at a higher-level, it achieved 184.97 Mbps effective bandwidth for a 1 MB message, resulting in 1.97 speedup, and 187.09 Mbps effective bandwidth for a 4 MB message, resulting in 1.99 speedup, from standard TCP socket on a single network. In addition, MuniSocket is more scalable and flexible.

7 Conclusion

A high-performance, concurrent message striping approach to utilize the existing multiple network interfaces on clusters has been designed and discussed. This approach utilizes the existence of a reliable transport protocol and the physical redundancy in interconnects to provide a flexible and scalable bandwidth solution for heterogeneous clusters. This model also provides load balancing among available multiple network interfaces. In this approach, message fragmentation, transmission and reconstruction are performed in parallel. Moreover, the techniques were further optimized to reduce the overhead and enhance the performance.

In the basic MuniSocket implementation, concurrent message fragmentation provides a scalable bandwidth solution with some overhead due to the thread contention over the fragment counter and the added fragment headers. In this paper, we introduced techniques to eliminate these two sources of overhead. The first technique removes the synchronization contention over the shared counter by using a fragment status array that indicates whether the fragment has been processed (in-transit or sent) or not. The second technique eliminates the need for the sequence number headers by relying on the FIFO property of the underlying reliable transport protocol. The two techniques reduce the overhead of the implementation significantly, thus speeding up the message transfer. The experimental results show a substantial enhancement in transmission time, especially for medium sized messages. In the dual-channel case, the Enhanced MuniSocket implementation approached the peak bandwidth at a message size of 16 KB, while the basic implementation only got close to the peak at a message size of 0.5 MB. Furthermore, a technique to reduce threads contention and sequencing overhead in the multiple-channel case (more than two networks) was developed and evaluated. For a message with n fragments in average case, only $O(\log_4(n))$ control messages are needed to replace the n sequence headers in the basic concurrent striping approach. Similarly, the overall performance of the Enhanced MuniSocket was shown experimentally to be substantially better than the basic MuniSocket.

Although the Enhanced MuniSocket provides good performance results, there are more prospects for enhancements that would further improve the performance. One possible direction for improvement is to investigate the effects of varying fragment sizes on the performance and how to dynamically tune it for the best results. Another possible future direction is the study of the effects of the TCP buffer size on the performance. We envision that these enhancements, among others, will provide further improved performance and finely tuned execution. Moreover, we are currently working on providing a fully transparent implementation of MuniSocket (and the Enhanced MuniSocket) that can be used by any application originally coded to use regular sockets without any changes in the application's code. In addition, we plan to comprehensively compare middleware-level striping and low-level striping.

Acknowledgment This project was partially supported by a National Science Foundation grant (EPS-0091900), a Nebraska University Foundation grant (26-0511-0019), and an Academic Priority Grant of the University of Nebraska-Lincoln. We would also like to thank the members of the secure distributed information (SDI) group and the research computing facility (RCF) at the University of Nebraska-Lincoln for their continuous help and support. We would also like to extend our gratitude to the anonymous referees for their constructive comments and suggestions.

References

1. Adiseshu H, Parulkar G, Vargese G (1996) A reliable and scalable striping protocol. Comp. Communication Rev. 26:131–141
2. Beowulf Ethernet Channel bonding web page at http://www.beowulf.org/software/bonding.html, June 2002.
3. Configuring Ethernet Channel bonding, http://www.beowulf-underground.org/doc_project/BIAA-HOWTO/Beowulf-Installation-and-Administration-HOWTO-12.html
4. Brendan C, Traw S, Smith J (1995) Striping within the network subsystem. IEEE Netw. 22–29
5. Channel bonding benchmark results at http://www.fos.su.se/compchem/jazz/bond.html, June 2002.
6. Dongarra J, Hey T, Strohmaier E (1996) Selected results from the PARKBENCH benchmark. In Proceedings of the 2nd European Conference on Parallel Processing (Euro-Par '96), pp. 251–254,
7. Figueira S, Berman F (2001) A slowdown model for applications executing on time-shared clusters of workstations. IEEE Transactions on Parallel and Distributed Systems 12(6):653–670
8. Getov V, Hernandez E, Hey T (1997) Message-passing performance of parallel computers. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par '97) pp. 1,009–1,016,
9. GridFTP: Universal Data Transfer for the Grid, The Globus Project white paper. 2000. The University of Chicago and the University of Southern California. At http://www.globus.org/datagrid/deliverables/C2WPdraft3.pdf, and GridFTP Update. 2002. At http://www.globus.org/datagrid/deliverables/GridFTP-Overview-200201.pdf
10. Katz R, Gibson G, Patterson D (1989) Disk system architectures for high performance computing. In Proc. of the IEEE 77(12):1842–1858,
11. Mohamed N, Al-Jaroodi J, Jiang H, Swanson D (2002) A user-level socket layer over multiple physical network interfaces. In Proceedings of the 14th International Conference on Parallel and Distributed Computing and Systems pp. 810–815,
12. Mohamed N, Al-Jaroodi J, Jiang H, Swanson D (2003) Performance properties of combined heterogeneous networks. In the Proceedings of IPDPS 2003, International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'03), IEEE,
13. Mohamed N, Al-Jaroodi J, Jiang H, Swanson D (2003) A Middleware-Level Parallel Transfer Technique over Multiple Network Interfaces. ClusterWorld Conference and Expo, San Jose, California
14. Mohamed N, Al-Jaroodi J, Jiang H, Swanson D (2003) Scalable bulk data transfer in wide area networks. International Journal of High Performance Computing Applications 17(3)
15. Theoharakis V, Guerin R (1993) SONET OD-12 interface for variable length packets. The Second International Conference On Computer Communication and Networks