Impact of On-Demand Connection Management in MPI over VIA*

Jiesheng Wu* Jiuxing Liu * Pete Wyckoff† Dhabaleswar Panda*

*Computer and Information Science
The Ohio State University
2015 Neil Avenue
Columbus, OH 43210
{wuj, liuj, panda}@cis.ohio-state.edu

†Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
Phone: 614 247 7956
pw@osc.edu

Abstract

Designing scalable and efficient Message Passing Interface (MPI) implementations for emerging cluster interconnects such as VIA-based networks and InfiniBand are important building next generation clusters. In this paper, we address the scalability issue in the implementation of MPI over VIA by on-demand connection management mechanism. *The* on-demand connection management *is designed to limit the use of resources to what applications absolutely require. Thus, the MPI implementation ensures that resource usage scales only as demanded by the application itself, not the underlying system.*

We address the design issues of incorporating the on-demand connection mechanism *into an implementation of MPI over VIA. A complete implementation was done for MVICH over both cLAN VIA and Berkeley VIA. Performance evaluation on a set of microbenchmarks and NAS parallel benchmarks demonstrates that the* on-demand mechanism *can increase the scalability of MPI implementations by limiting the use of resources as needed by applications.*

Furthermore, performance evaluation also shows that the on-demand mechanism *delivers comparable performance as the* static mechanism *in which a fully-connected process model exists in MVICH over cLAN VIA. It even performs better for certain applications. It performs better in MVICH over Berkeley VIA on Myrinet. These results demonstrate that the* on-demand connection mechanism *is a feasible solution to increase the scalability of MPI implementations over VIA- and InfiniBand-based networks.*

*This research is supported in part by Department of Energy's Grant # DE-FC02-01ER25506 and an NSF Grant #EIA-9986052.

1 Introduction

In recent years, clusters of workstations have become both a popular and powerful environment on which to do parallel processing due to the tremendous improvement in network hardware and protocols, and the ever decreasing cost of commodity components. On these cluster systems, it is crucial to have an efficient communication system that can make effective use of the capability of the underlying network hardware and deliver the actual performance of the hardware to applications.

User-level communication protocols such as AM [32], VMMC [6], FM [22], U-Net [31, 33], LAPI [29], and BIP [24] were developed to attempt to achieve this performance objective by bypassing the operating system and eliminating intermediate copies in the critical communication path. To achieve high performance promised by these protocols, it is important that the network interface cards (NIC) have the ability to perform certain aspects of the protocol processing, offloading the work from the host CPU.

The Virtual Interface Architecture (VIA) [10, 8] has been developed to standardize these protocol efforts and to make these features available in commercial systems. VIA defines an abstraction of network protocols and hardware capabilities that provides applications with direct access to the network, eliminates intermediate data copies using remote direct memory access (RDMA), and thus increases host processor application productivity. Since its introduction, different software and hardware implementations of VIA have become available. Berkeley VIA [7], Giganet VIA [11], Servernet VIA [28], M-VIA [1], and FirmVIA [5] are some of the prominent implementations.

More recently, the InfiniBand [14] architecture has been proposed to provide the next generation high-performance communication architecture for both inter-processor communication and I/O systems. InfiniBand incorporates many

of the features of the VI architecture and draws on research efforts from high performance networked I/O. In the next several years, VIA/InfiniBand will be the standard architecture for high performance computing systems, servers, and clusters.

On these cluster systems, MPI [19] has become the de facto standard for developing portable parallel applications. Several MPI implementations built on VIA are currently available: MPI/Pro [27] based on GigaNet cLAN; MVICH [17], a port of the generic MPICH implementation on M-VIA, GigaNet VIA and Servernet VIA; and LAM/MPI [18] on M-VIA. We have also recently retargeted MVICH to Mellanox's InfiniBand card [16, 2]. Several complex design issues have been addressed in the literature describing these efforts and in overview papers [25]. However, the aspect of connection management has not yet been explored in detail.

MPI 1.2 does not specify a connection model. It assumes that all processes are fully connected after initialization. VIA, though, is connection-oriented, such that for any pair of processes that will communicate, a VI endpoint must be created on each node and a connection between these two VI endpoints must be established beforehand. These connections between processes must be handled explicitly by the MPI library itself when implemented on the VI Architecture. In MVICH, for instance, each process creates $N-1$ VI endpoints and then establishes $N-1$ connections to other processes statically during MPI_Init(), where N is the number of processes in the MPI application. Thus, after initialization, there is a fully-connected network among all participating processes from the point of view of the VI layer. The total number of VI endpoints is $N \times (N-1)$, with half that many connections. This connection management mechanism is simple to implement; however, there are both scalability and performance problems in this *static connection mechanism*:

1. In large systems, the time to establish and to destroy a fully-connected process network is considerable and significantly affects the time to start and to terminate a parallel application. This is because connection setup is typically a costly operation with operating system involvement.

2. Beyond the basic connections required for message passing, connections are likely to be needed for other operations, such as for I/O operations on a parallel file system and for debugging. The number of connections supported in a specific VIA system serves as a hard limit to scaling, since each VI endpoint and connection consumes resources from the VIA library, the device driver, and the NIC, which are limited. Thus, a fully-connected parallel application can easily exceed this limitation with only a moderate task size.

3. The logically fully-connected process network requires a physically fully-connected network, which requires a considerable number of links and switches [35].

4. In an MPI implementation based on VI, each VI is associated with certain internal buffers and pre-preposted descriptors [8]. The amount of these resources for each process would ideally be a function of an application's communication pattern when scaling to large systems. However, in the *static connection mechanism*, it is a function of the number of processes in the application, regardless of what the application really needs. In reality, many large-scale scientific parallel applications do not require a fully-connected process model. Table 1 lists the average number of communication destinations per process in several applications [30]. Consequently, a large amount of resources is never used in these applications for the *static connection mechanism* approach. For example, if each VI is associated with a 120 kB buffer as in MVICH, the total amount of unused memory for the NAS benchmark [20] Conjugate Gradient (CG) application on a 1024 node cluster is 119 GB using the *static connection mechanism*. We should note that due to the restriction of VIA communication, these buffers have to be pinned down in physical memory. Therefore, a large amount of memory is essentially unused. This not only wastes valuable resource but also adversely affects the system performance.

Table 1. Average number of distinct destinations per process in several large-scale applications.

App	Number of Processes	Average number of distinct destinations
sPPM	64	5.5
	1024	< 6
SMG2000	64	41.88
	1024	< 1023
Sphot	64	0.98
	1024	< 1
Sweep3D	64	3.5
	1024	< 4
Samrai 4	64	4.94
	1024	< 10
CG	64	6.36
	1024	< 11

5. The number of VIs and connections may have an impact on performance [4] of the underlying VIA communication systems, even if the endpoints are never

used after setup. Figure 1 shows this impact on Berkeley VIA over Myrinet. Thus, it is worthwhile to eliminate all unused VIs and connections in the implementation of MPI on a VIA-based communication system.

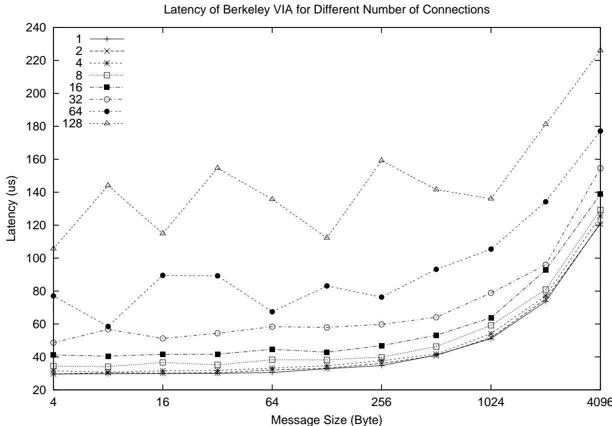


Figure 1. Latencies in BVIA as a function of the number of active VIs.

An alternative approach that reduces the number of VIs and connections required by a large application is to establish connections when they are needed. We call this approach the *on-demand connection mechanism*. In this approach, the creation of two VI endpoints and establishment of a connection between them are on a per-use basis for any pair of processes, and undertaken only when it is known that they need a connection to pass messages.

In this paper, we focus on designing and implementing an *on-demand connection mechanism* in MPI over VIA. Detailed performance evaluations are also presented. The main contributions of this paper are as follows:

- We have addressed the issues in incorporating the *on-demand connection mechanism* into an MPI implementation on VIA-based communication systems.

- We have implemented this *on-demand connection mechanism* in MVICH over cLAN VIA and over Berkeley VIA on Mryinet.

- We have shown that by using the *on-demand connection mechanism*, the resource consumption of parallel applications, such as the NAS parallel benchmarks, is dramatically reduced.

- We have conducted a serial of performance evaluations. Performance results of both microbenchmarks and the NAS parallel benchmarks show that there is very little performance degradation after incorporating the *on-demand connection mechanism* in MVICH over cLAN VIA. The performance even increases for certain applications. The performance over Berkeley VIA on Myrinet is better than *static mechanism*.

The rest of the paper is organized as follows. Background and related work are presented in section 2. Section 3 describes the design issues of incorporating the *on-demand connection mechanism* into a generic MPI implementation, while section 4 presents our implementation of the on-demand connection mechanism in MVICH. The performance results are presented in section 5, followed by conclusions and future work in section 6.

2 Background and Related Work

The Virtual Interface Architecture (VIA) [8] specifies the interface between high performance network hardware and computer systems. This architecture defines mechanisms to eliminate much of the protocol processing overhead and intermediate data copies by providing user applications a protected and directly accessible network interface called the Virtual Interface (VI). Many papers offer detailed overviews of VIA; here, we only present information on VIA related to connection management.

A VI is a bi-directional communication endpoint. To use a VI, a point-to-point channel must be established between two VIs, in a process known as connection setup. Each channel supports the traditional two-sided send/receive model, as well as a one-sided remote memory access model. Each VI consists of a send queue and a receive queue. A process can submit a request, in the form of a descriptor, to either queue to facilitate a data transfer. The send/receive semantics require a one-to-one correspondence between send descriptors and receive descriptors. Receive descriptors must be posted with the address of the destination buffer before the arrival of the expected message; otherwise, the message is dropped. This requirement, consequently, has a significant impact on the implementation of MPI over VIA. Particularly, a certain number of descriptors with message buffers must be allocated and preposted in the receive queue for each VI. Note that these buffers must be pinned down in physical memory.

VI connection establishment is typically a costly operation with operating system involvement. Initially in version 0.95 of the VIA specification, only a *client-server connection model* [8] was defined. The server side waits for incoming connection requests and then either accepts them or rejects them based on the attributes associated with the remote VI. Since VIA specification 1.0, a *peer-to-peer model* has been included as well. In this model, either of the two process to be connected can initiate the connection by calling a VIA peer-to-peer connection setup function. The connection is established after both of them have called the connection setup function.

There is some literature which analyzes the VI architecture and its ability to support parallel and distributed communication in general, and MPI implementations in particular. The inherent costs of using VI primitives to implement Active Messages and Split-C was analyzed in [3], while [27] described efficiency features of the MPI/Pro architecture gained by exploiting certain characteristics of the VI architecture. Also, [18] presented how to build a substrate over VIA to ease the implementation of LAM/MPI, and [13] compared the performance of Myrinet and GigaNet and their respective impact on an MPI implementation. Their results indicated that the implementation of MPI is crucial for system performance. In [9], performance of MPI/Pro over cLAN and ScaMPI over SCI was compared, while [21] compared performance of LAM/MPI, MPICH and MVICH on a Gigabit Ethernet network. In [34], Wong *et al.* present a study of the scalability of the NAS Parallel benchmarks from the aspect of resident working set and communication performance. None of these works studied the connection scalability issues of the implementation of MPI over VIA and the impact of connection management on application performance.

Brightwell *et al.* [25] analyze the scalability limitations of VIA in supporting the CPlant runtime system as well as any high performance implementation of MPI. The authors of this paper claim that the *on-demand connection mechanism* is not a good approach to increase the scalability of the MPI implementation by qualitative analysis; however, their analysis does not consider the impact of the number of connections on the underlying VIA communication systems and the impact of the allocation of large memory buffers on system performance. We argue that with efficient design and implementation, the *on-demand connection mechanism* can achieve comparable performance. And it does not require an extra thread to make progress as well as keeping the same determinism, predictability, and fairness as the *static mechanism*.

3 Design of On-demand Connection Management

In the *on-demand connection mechanism*, the creation of VI endpoints and the establishment of a connection between two VI endpoints are performed strictly on a per-use basis for any pair of processes, and undertaken when they pass messages for the first time. Although conceptually simple, it is not trivial to incorporate it into current MPI implementations. In this section, we present the design issues in *on-demand connection mechanism*. We also discuss how to maintain progress and MPI communication semantics.

3.1 Threading vs. Polling

The *on-demand connection mechanism* requires that an MPI process should be able to handle communication requests and connection requests simultaneously. The process cannot just block for communication, or just block for connection. It must be ready to handle both. Since VIA itself doesn't provide such capability, this problem must be addressed explicitly in the MPI implementation.

Two alternatives can be used to solve the problem. The first one is to use a separate thread to handle all connection requests. The main thread is dedicated to communication and computation. In this way, the communication progress can be ensured. However, a separate thread will incur quite large overhead and the context switch between the main thread and the connection thread may degrade application performance. Some MPI implementations, like MPI/Pro, are multi-threaded. For these implementations, it may be possible to use the existing thread to handle connection requests. However, many MPI implementations, such as MPICH, are based on a single thread. These MPI implementations may not even be thread safe. Incorporating the *on-demand connection mechanism* into them by adding a separate thread will be very difficult.

Another method is using polling to handle both communication and connection requests. In a polling based approach, the process checks periodically to see if there are pending communication or connection requests. This matches quite well with single threaded MPI implementations such as MPICH and it has very little overhead. In this paper, we have chosen such an approach and implemented it for MVICH on top of both cLAN VIA and Berkeley VIA.

3.2 Client/Server vs. Peer-to-Peer Connection Model

VIA provides two connection models: client/server and peer-to-peer. In theory both model can be used to implement *on-demand connection mechanism*. However, we have found that the peer-to-peer connection model is generally better for the *on-demand connection mechanism*.

In the client/server connection model, one process acts as the server and the other acts as the client. The two processes have different actions during the connection setup procedure. Consequently, we have to be careful choosing the client and the server; otherwise deadlock situations may occur. The asymmetry of the client/server model also makes the implementation awkward.

In the peer-to-peer model, both processes involved in the connection setup behave similarly. And the order which of the two processes perform these actions first doesn't matter. This matches quite well with the MPI communication because either the message sender or the receiver can initiate

the communication setup. The symmetry of the model also makes the implementation task easier.

In Berkeley VIA, only peer-to-peer connection model is provided. In cLAN VIA, both models are provided. However, as we shall see later, the performance of peer-to-peer connection is much better than client/server in cLAN VIA.

3.3 Progress Rule

The progress rules of MPI [19] are both a promise to users and a set of constraints on implementors, though different interpretations seem to be possible. MPICH [12] follows a loose interpretation in which all pending communication requests are progressed by the library every time a communication function is called. On the other hand, MPI/Pro [27] makes progress of all messages independently of the sequence of the user process calls by using an additional thread to facilitate progress, complying with a more strict interpretation of the MPI progress rule. In incorporating the on-demand connection mechanism into MPI for the VI Architecture, it is important to maintain this progress rule.

The *on-demand connection mechanism* can be incorporated into the above two typical implementations of MPI without violation of their respective progress semantics. The requirement of a "server" thread waiting to establish connections [25] depends on which interpretation of the MPI progress rule is provided. In MPICH, no thread is needed to maintain the loose interpretation of progress. A peer-to-peer connection request can be considered as another type of nonblocking communication requests. It can be progressed by the library every time a communication function is called. In particular, the connection setup can be done in the first blocking communication request if the initial send and receive requests are blocking operations. Or it can be done in other following calls if the first request is a nonblocking request.

In the case that there is a progress thread as implemented in MPI/Pro, this progress thread can take care of connection requests as well as communication requests. No extra control thread is required.

3.4 Pre-posted Send Requests

In the *on-demand connection mechanism*, one important scenario must be carefully examined for both correctness and message delivery order. This scenario is that in which an MPI issues multiple nonblocking communication requests before the corresponding connection is established. For the VI Architecture, any requests posted into the Send Queue of a VI which is not yet connected are discarded, which would result in MPI message loss. To prevent this, *pre-posted send requests* must be stored if the corresponding connection is not yet established and handled in order later when connections are available.

Attention also must be paid to the order of processing these pre-posted send requests. MPI specifies that messages are *non-overtaking*: if a sender sends two messages in succession to the same destination, and both match the same receive, the second message cannot be received until after the first. Without violation of the MPI message order rule, each VI must have its own first-in-first-out (FIFO) queue to store these pre-posted send requests. Thus, when a connection is established, pending requests in the associated VI FIFO queue must be processed in order.

Note that all receive requests and send requests issued after the related connection is established can be handled directly.

3.5 Message Reception with MPI_ANY_SOURCE

MPI allows a special parameter to be specified as the source host in a receive which serves to match a message issued from any sender. Since the receive may potentially match a message issued from any sender, the receiver may need to establish connections with all other processes.

This communication pattern exhibits a mismatch with the peer-to-peer connection model on which the on-demand connection mechanism depends. The only solution is to issue peer connection requests to all other processes in the specified communicator upon encountering a receive from MPI_ANY_SOURCE. The receiver will then have an established connection with the process with which it will eventually communicate. In this solution, each process has the same probability to established connection and communicate with the receiver as long as it wants. The receiver can receive messages from all processes in the group. For example, the receiver may first receive a message from A, the connection request from B can arrive before the second message from A arrives, then a connection can be established between B and the receiver. The first message from B may or may not arrive before the second message from A.

Note that there is no problem with fairness in this scenario since any nondeterminism is inherent in the application itself. If the receiver chooses to use MPI_ANY_SOURCE, and there happen to be multiple senders which could issue messages that might all match the receive, MPI offers no ordering guarantees, nor any concept of fairness in this case. As messages arrive, including data and connection requests, they will be processed in order and matched against the receive queue.

3.6 Semantics of MPI Communication Modes

MPI defines four communication modes: standard, buffered, synchronous and ready. They differ in whether a matching receive is required in order to start a send, and whether a send operation completes independently or if it requires that a matching receive has taken place. Only the buffered mode send is *local* since its completion does not depend on the occurrence of a matching receive. Send operations in the other three modes are *non-local*: successful completion of a send operation may depend on the occurrence of a matching receive. (The system has the option of buffering some sends and declaring local completion.) The *on-demand connection mechanism* has no impact on the non-local send modes. It does not change the *local* semantics of a buffered send either. This means that the semantics of MPI communication modes are maintained after adding the *on-demand connection mechanism* into the implementation of MPI.

4 Implementation

We implemented the *on-demand connection mechanism* for MVICH on top of both GigaNet cLAN VIA and Berkeley VIA on Myrinet. MVICH is a freely available port of MPICH over several VIA implementations. All modification for incorporating the *on-demand connection mechanism* occur in the ADI layer [12].

Unlike the original MVICH implementation with static connection management, there is no VI creation and VI connection setup in the MPI low-level initialization routine, MPID_Init(). Instead, a VI is created and a peer-to-peer connection request is issued during the processing of the first communication request. Before the connection for a VI is established, all send requests on that VI are stored in a FIFO. Thus, on the sending side, modification is made in the device contiguous send routines, MPID_IsendContig() and MPID_IssendContig(). It first checks whether the local VI and its connection for the destination process have been created and setup. If not, it creates a VI and issues a peer-to-peer connection request to try to make a connection between the local VI and a destination VI. The send request is stored into the pre-posted send request FIFO queue for that VI. If the VI has been created, but a connection is still not available, only the request is stored. If the connection is available, the request is processed normally.

On the receiving side, all calls go through a routine in the VIA receive interface, MPID_VIA_Irecv(), and the same modification as that on the sending side is made there, except that there is no extra queue needed to store receive requests when the connection is not yet established. In the case that the receive request uses MPI_ANY_SOURCE, though, the receiver tries to create a VI for each process in the specified communicator if the VI does not exist and further issues a peer-to-peer connection request.

MVICH adheres to the weak form of the MPI progress rule in that message progress is guaranteed only when user processes call the MPI library. This is achieved by running a common device check routine, MPID_DeviceCheck(), as part of most of the MPI library calls. Essentially, MPID_DeviceCheck() is the function in MVICH which handles all message progress. Modification is thus made to MPID_DeviceCheck() to maintain both connection progress and message progress. Because the peer-to-peer connection model is similar to nonblocking point-to-point message passing, MPID_DeviceCheck() can check all pending connection requests in a non-blocking manner. When a connection is established and the FIFO of the local VI is not empty, requests in the queue are processed in order by the progress routine.

Our implementation of the *on-demand connection mechanism* keeps the same communication semantics and progress guarantees as the original MVICH implementation using the *static connection mechanism* except for a small difference in the *non-local* semantics of the standard send mode. In the static mechanism, the standard mode send is *non-local* and the successful completion of a send operation may depend on the occurrence of a matching receive, for example, when flow control credits are used up for short messages or a long message switches to a rendezvous protocol. In the *on-demand connection mechanism*, even if flow control credits are available for short messages, the completion of short messages still depends on whether the connection can be established. That is, the completion of pre-posted send requests for short messages at the source process depends on whether the receiver has planned to communicate with the sender. This is not a problem in any correct MPI program because the receiver can always find a chance to issue a connection request to the sender in any communication requests destined for the sender. Note that this also complies with the original interpretation of the MPI progress rule.

5 Performance Results

To evaluate the performance of *static connection mechanism* and *on-demand connection mechanism*, we conducted a series of performance measurements using a set of micro-benchmarks and NAS parallel benchmarks.

5.1 Experimental Setup

Our experimental testbed is a cluster system consisting of 8 Dell Power Edge 6400 nodes connected by GigaNet cLAN and Myrinet. We use cLAN 1000 Host Adapters and cLAN5300 Cluster switches. LANai 7.0 Adapters are used

for Myrinet. Each node has four 700MHz Pentium III Xeon processors, built around the ServerWorks ServerSet III HE chipset, which has a 64-bit 66MHz PCI bus for all experiments. Thus, there are actually 32 CPU processors in total. These nodes are equipped with 1GB of SDRAM and 1MB L2-level cache. The linux kernel is 2.2.17.

5.2 Scalability

One of our main objectives is to increase scalability of MPI implementations on the Virtual Interface Architecture. As mentioned earlier, implementing the *on-demand connection mechanism* in MVICH enables the implementation of MPI over VIA to limit the use of resources to what applications absolutely require. Table 2 lists the average number of VI endpoints created in each process in *static mechanism* and *on-demand mechanism* for tests in this paper (due to space limit, some of them are not presented and can be found in [15]). As shown in this table, the utilization of resources associated with VIs is very low with *static mechanism*. With increase of the size of applications, it can be expected that the utilization becomes much lower. This is true for most parallel applications in which the number of communicated processes does not grow in proportion with the size of applications, instead it remains constant or grows with log-scale. *On-demand mechanism* eliminates all unused VI endpoints, connections and their related resources.

Table 2. Average Number of VIs and Resource Usage in Each Process with *static connection mechanism* **and** *on-demand connection mechanism***.**

App	Size	Ave. number of VIs		Resource Utilization	
		static	*on-demand*	*static*	*on-demand*
Ring	16	15	2	***0.13***	***1.0***
	32	31	2	***0.06***	***1.0***
Barrier	16	15	4	***0.27***	***1.0***
	32	31	5	***0.16***	***1.0***
Allreduce	16	15	4	***0.27***	***1.0***
	32	31	5	***0.16***	***1.0***
Alltoall	16	15	15	*1.0*	*1.0*
	32	31	31	*1.0*	*1.0*
Allgather	16	15	5	***0.33***	***1.0***
	32	31	6	***0.19***	***1.0***
Bcast	16	15	4	***0.27***	***1.0***
	32	31	5	***0.16***	***1.0***
CG	16	15	4.75	***0.32***	***1.0***
	32	31	5.78	***0.19***	***1.0***
MG	16	15	15	*1.0*	*1.0*
	32	31	15	*1.0*	*1.0*
IS	16	15	15	*1.0*	*1.0*
	32	31	31	*1.0*	*1.0*
SP	16	15	8	***0.53***	***1.0***
	36	35	9.83	***0.28***	***1.0***
BT	16	15	8	***0.53***	***1.0***
	36	35	9.83	***0.28***	***1.0***
EP	16	15	4	***0.27***	***1.0***
	32	31	4.75	***0.15***	***1.0***

Furthermore, decreasing the number of VIs and connections in the communication system can relieve the issue of performance scalability in the underlying communication system, such as Berkeley VIA, which has performance penalty to maintain a large number of VIs and connections. The fewer VIs and connections used, the better performance the NIC and the system can deliver.

5.3 Latency and Bandwidth

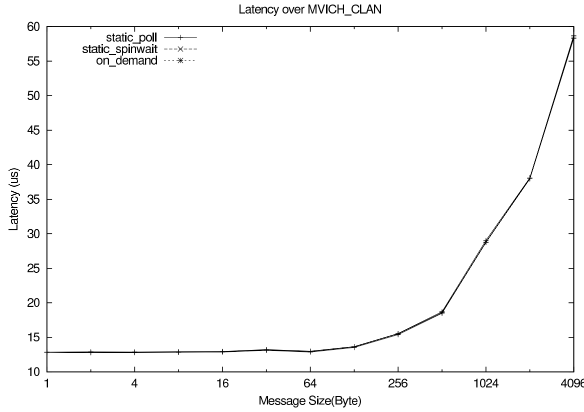
Generally there are two methods to complete a request: one is to infinitely poll the status of the request and another one is to wait for the completion. Particularly, the default method in MVICH over cLAN VIA is to first check 100 times and then go to wait if the request does not complete (*spinwait*). Since cLAN VIA does not use polling to implement wait, this has an impact on the performance of MVICH when the spincount (the default value is 100) is changed to a considerablely large number to make requests done in the spin step (here we call it *polling*). Thus, two cases are considered with the *static connection mechanism* for the performance measurement on cLAN VIA in the following several subsections: *static polling* and *static spinwait*. While in Berkeley VIA, wait operation is implemented by an infinite loop to check status. Thus, there is no difference between *polling* and *spinwait*. That is, MVICH on Berkeley VIA always uses polling to wait for any request completion. Thus, there is only one case, referred as *static polling*, with the *static connection mechanism* on Berkeley VIA.

Figures 2(a) and 2(b) show the latency results of MVICH on cLAN and Berkeley VIA, respectively. *Polling*, *spinwait* and *on-demand* achieve same performance. In these latency and bandwidth tests, any request can be done in spin step in *spinwait*. Thus, there is no difference between *polling* and *spinwait* on cLAN VIA. Similar results are shown in Figure 3 for bandwidth. Since the default threshold from the eager protocol to the rendezvous protocol is 5000 bytes, a jump happens around 5000 bytes in the test of bandwidth. This indicates that a threshold greater than 5000 is expected to deliver better performance.

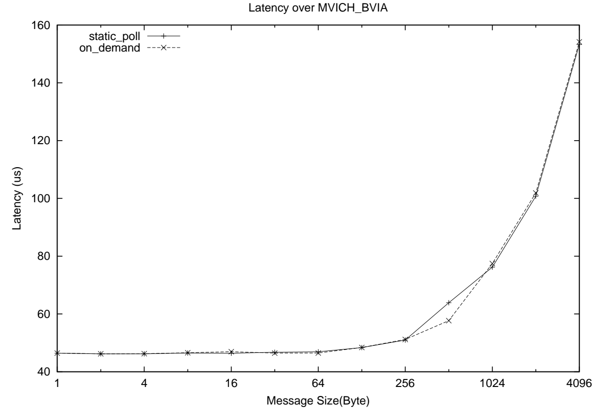
5.4 Collective Communications

In this subsection, we report the performance results of *MPI_Barrier()* and *MPI_Allreduce()* using the *static connection mechanism* and the *on-demand mechanism*. As mentioned earlier, two cases, *static polling* and *static spinwait*, are considered with *static connection mechanism* on cLAN VIA.

- *MPI_Barrier:* To measure the latency of a barrier operation, each process repeats 1000 barrier operations

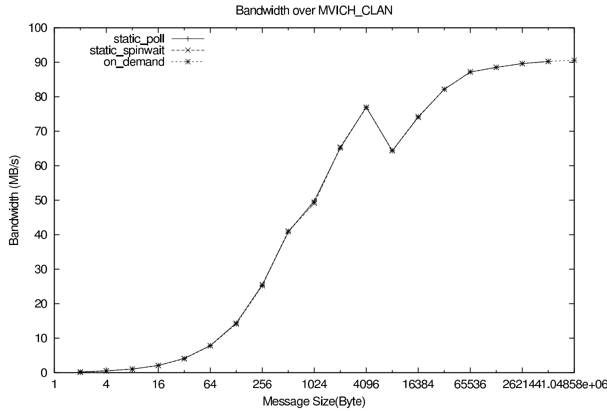


(a) MVICH on cLAN VIA

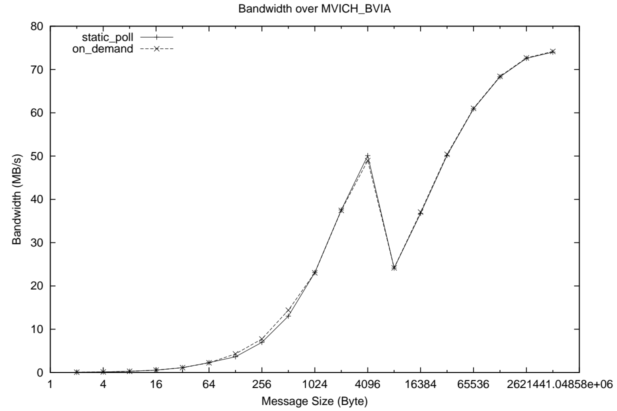


(b) MVICH on Berkeley VIA

Figure 2. Latency of MVICH on cLAN VIA and Berkeley VIA.



(a) MVICH on cLAN VIA



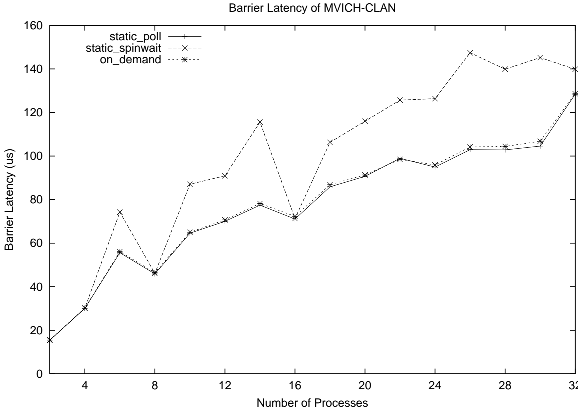
(b) MVICH on Berkeley VIA

Figure 3. Bandwidth of MVICH on cLAN VIA and Berkeley VIA.

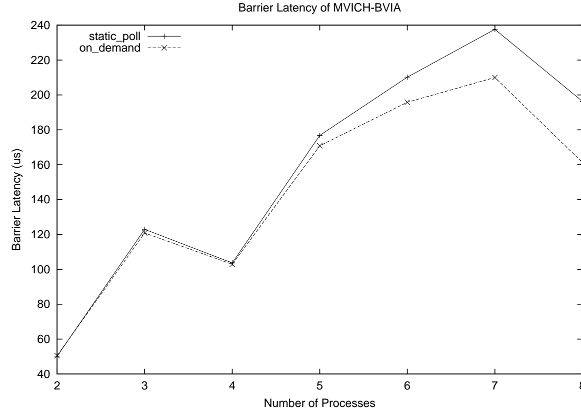
and then reports the average latency. One of the nodes gathers these latency values and computes their average value as the latency of a barrier operation. Figure 4(a) shows the latency results with different numbers of processes. If the number of processes is not a power 2 number, fluctuation occurs since extra steps are needed for nodes which are not in the binomial tree [26]. As seen in Figure 4(a), the *on-demand mechanism* can achieve same results as the *static mechanism* using *polling*. If *spinwait* is used, there is a high probability that some processes cannot complete a request in the spin step. When these processes go to wait step, big overhead occurs. Thus, *spinwait* is no good for barrier operation.

Figure 4(b) shows the barrier latency of MVICH over Berkeley VIA. Because the number of VIs and connections has impact on performance of the Berkeley VIA, the *on-demand mechanism* can achieve better performance. For example, the latency of barrier on 8 nodes is 161 microseconds using the *on-demand mechanism*, while 196 microseconds using the *static mechanism*. The number of VIs is 3 in the former case, and 7 in the latter case.

- *MPI_Allreduce:* This is one of the most frequently used operations [30] in large scientific applications. The test program we used is the *llcbench* [23] benchmark suite. The operation in MPI_Allreduce is *MPI_SUM*. Originally, this program repeats these collective operations multiple times and then each process reports its own average latency. Only the latency on the



(a) MVICH on cLAN VIA



(b) MVICH on Berkeley VIA

Figure 4. Latency of Barrier in MVICH on cLAN VIA and Berkeley VIA.

master process (process 0 as default) is reported as the latency of the allreduce operation. In our test, the master process gathers all values from others and outputs the average value as the latency shown in Figure 5.

On cLAN VIA, the *on-demand mechanism* can achieve the same performance as the *static mechanism* using polling with negligible degradation. If *spinwait* is used, performance of MPI_Allreduce is worse. The same reason as in the barrier operation can be applied here.

On Berkeley VIA, the reduction of VIs in the *on-demand mechanism* is beneficial. It can be seen that the *on-demand mechanism* achieves better performance than the *static mechanism*.

Similar results are achieved for other collective operations. Overall, *on-demand mechanism* delivers the same performance as *static polling*, while both perform better than *static spinwait*.

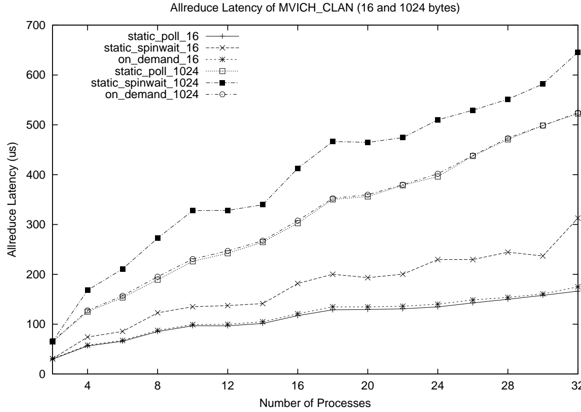
5.5 Results of NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB) have been widely used to objectively measure the performance of highly parallel computers and to compare their performance [20]. The NPB suite consists of a set of 8 programs: EP, FT, MG, CG, IS, LU, SP and BT. There are three standard problem sizes, Class A, B and C for small, medium and large size problems, respectively. The benchmark results are given by the CPU time, total operation count (Mop/s) and process utilization (Mop/s per process). Mop/s is derived from the CPU time and the total operations in the benchmarks. In this paper, we show the results of MG, CG, IS, SP and BT only. We use the CPU time as comparison metrics. Because we are interested in the impact of *on-demand mechanism*, the normalized CPU time is presented in this paper, in addition the CPU time.

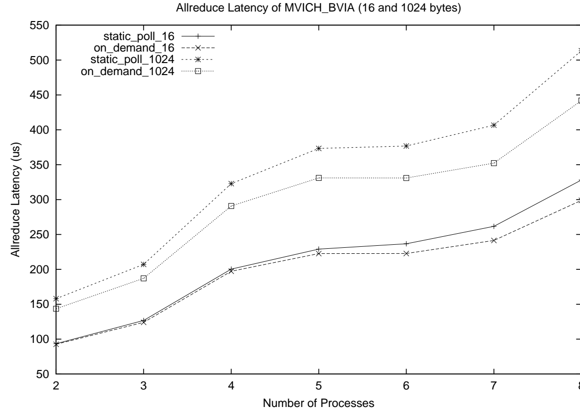
Figure 6 shows the performance of 5 NPB programs with different program sizes and numbers of processes. Our testbed has 32 processors, the largest number of processes tested in CG, MG, IS is 32, while only 16 in SP and BT since they require a power-of-two number of processes. For MVICH over Berkeley VIA, we could not run more than one processes on each node. Thus, the largest number of processes supported for Berkeley is 8. The x axis lists the tested combinations of the class and The number of processes for each program, and the y axis represents the normalized CPU time for clearer comparison. For example, in Figure 6(c), C.32 means the CG program runs on 32 processors using the Class C size.

It should be noted that in those benchmarks, the time for establishing connections is not included with *static connection mechanism*. In *on-demand mechanism*, part of connections are established during the MPI initialization (MPI_Init has communication) and others are established during the execution of applications. That is, part of connection time is included in the CPU time reported by the NAS benchmarks with *on-demand mechanism*. This connection overhead can be amortized by all communication operations on that connection. When there is much communication, the performance improvement by the reduction of VIs and connections and other related resources can win up. This is a common pattern found in all benchmarks, as seen in Figure 6 and 7.

MG results are shown in Figure 6(a). MG uses fine-grain communication. Its spatial decomposition of data provides nice communication pattern in the nearest-neighbor region. *Barrier, allreduce and bcast* are called in MG be-



(a) MVICH on cLAN VIA



(b) MVICH on Berkeley VIA

Figure 5. Allreduce Latency in MVICH on cLAN VIA and Berkeley VIA.

sides send/recv. Thus, the *on-demand mechanism* and *static polling* can achieve better performance over *static spinwait* for MG. Figure 6(a) also shows that *on-demand mechanism* can deliver better performance compared to *static polling*. This may be related to our modification in the communication code path.

IS benchmark mainly relies on *allreduce, alltoall* and *alltoallv* (a vector version of the all-to-all operation). IS is communication bound. For the B class with 16 processes a total amount of 1920 MB must be transfered at each all-to-all exchange [9]. Thus, as shown in Figure 6(b), IS can benefit from the better performance of collective operations in *static polling* compared to *on-demand mechanism*. Similarly they both perform better than *static spinwait*.

Figure 6(c) shows the CG performance. We can see that the *on-demand mechanism* can achieve comparable performance to the *static mechanism* when *polling* is used. The average percentage of performance degradation is less than 2%. It performs better than the *static mechanism* when *spinwait* is used.

Figure 6(d) shows the performance of SP and BT. It can be observed that the *on-demand mechanism* can deliver comparable performance as the *static polling*.

Figure 7 shows the performance of IS, CG, EP, SP and BT of MVICH over Berkeley VIA. The *on-demand mechanism* performs better than the *static mechanism*. This is attributed to the performance improvement with the reduction of VIs and connections. Note that even there is same number of VIs with *static and on-demand mechanisms* in the IS benchmark, *on-demand mechanism* still performs better. This is because the number of VIs gradually increases as needed with *on-demand mechanism*.

Table 3 represents the actual CPU time.

5.6 Initialization Time

As mentioned earlier, the *static mechanism* can be implemented in two ways: client-sever and peer-to-peer models. cLAN VIA supports both models, while Berkeley VIA only supports peer-to-peer model. We measured the time for finishing MPI_init function in MVICH over cLAN VIA and Berkeley VIA, respectively. The initialization time shown is an average of the initialization time from all processes. The client-server implementation and the peer-to-peer implementation over cLAN VIA are both measured.

Figure 8(a) shows the initialization time over cLAN VIA. Current implementation of the client-server model in MVICH is a serialized version, in which connections are established in order regardless of the arrival order of connection requests from peer processes, while the peer-to-peer scheme can avoid this serialization by checking each potential connection request. Thus, time for establishing a fully-connected VI network among all participating processes in the client-server model is much higher than other two. Note that in the *on-demand mechanism*, it also uses peer-to-peer model. However, it does not create a fully-connected network. Thus, the initialization time is lower than the peer-to-peer implementation of the *static mechanism*.

Figure 8(b) shows the initialization time in MVICH over Berkeley VIA. Similarly, the initialization time is lower than the peer-to-peer implementation of *static mechanism*. This time has an impact on the wall clock time for the execution of applications.

6 Conclusions and Future Work

The scalability of an implementation of MPI over VIA is one of crucial issues in cluster systems connected by VIA-

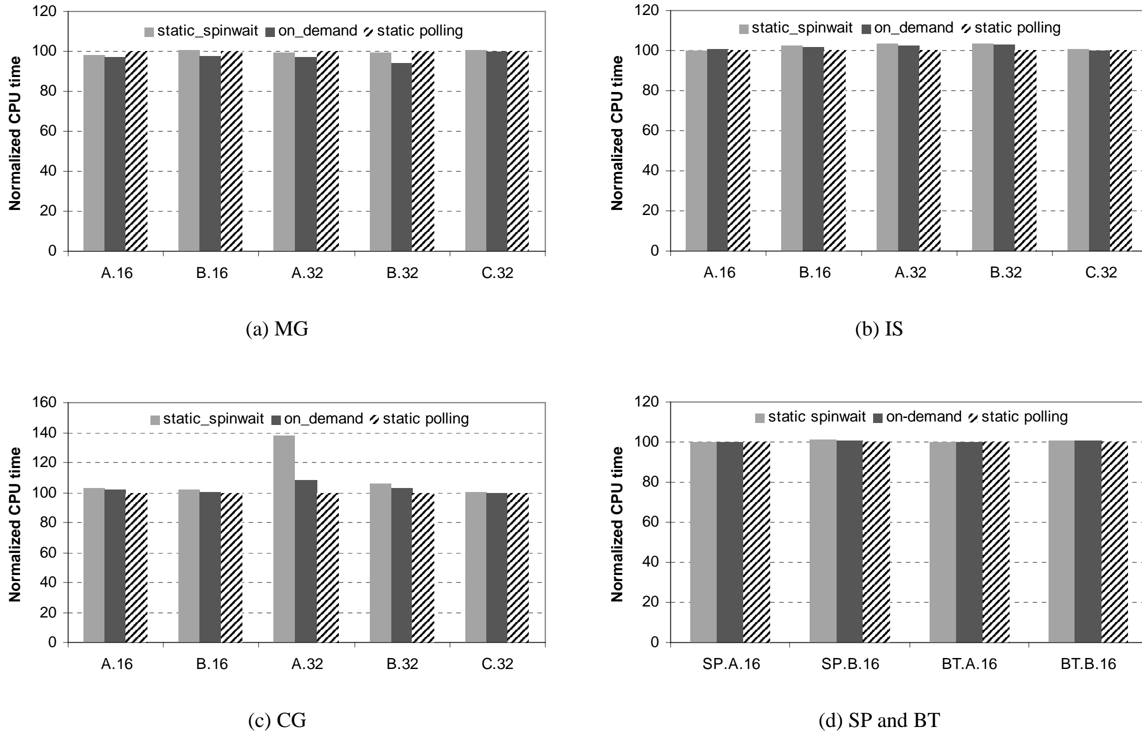


Figure 6. Performance results of NPB programs with *static connection management* **and** *on-demand connection management* **in MVICH over cLAN.**

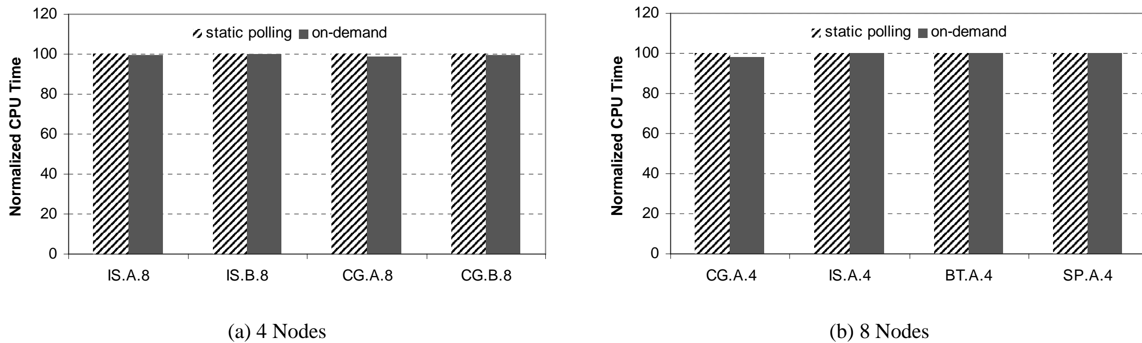
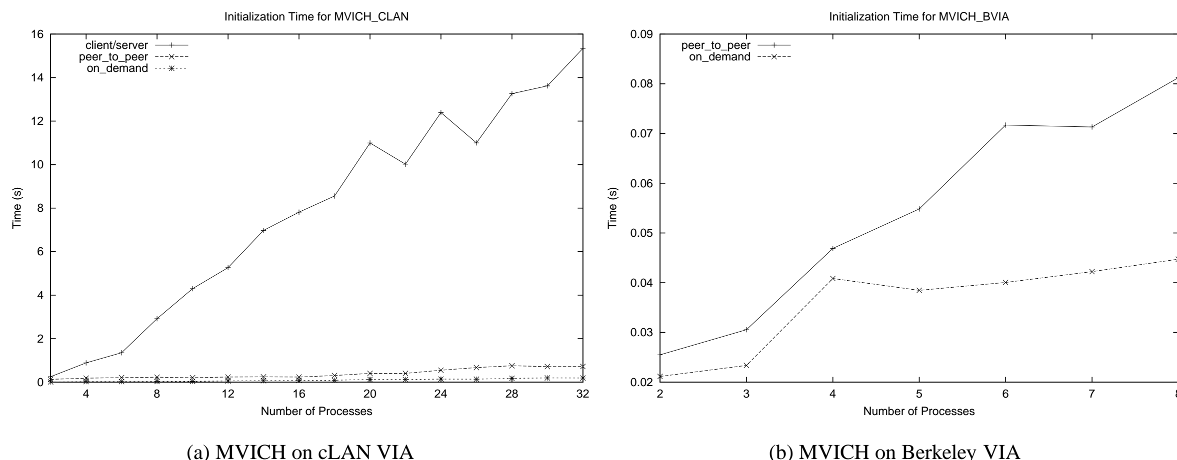


Figure 7. Performance results of NPB programs with *static connection management* **and** *on-demand connection management* **in MVICH over Berkeley VIA on Myrinet**

based networks. Since InfiniBand has many characteristics in common with VIA and with VIA-based I/O specifications such as Next Generation I/O (NGIO), this issue will continue to exist along with next-generation InfiniBand hardware. In this paper, we addressed the design issues of incorporating the *on-demand connection mechanism* into an implementation of MPI over VIA. A complete implementation was done for MVICH over cLAN VIA and over Berkeley VIA on Myrinet. Performance evaluation on a set of microbenchmarks and NAS parallel benchmarks demonstrates that the *on-demand mechanism* can limit the use of resources to what applications absolutely require. Thus, the MPI implementation ensures that resource usage scales only as demanded by the application itself, not the underly-



(a) MVICH on cLAN VIA

(b) MVICH on Berkeley VIA

Figure 8. Initialization time in MVICH on cLAN VIA and Berkeley VIA.

ing system.

Furthermore, performance evaluation also shows that the *on-demand mechanism* delivers comparable performance for collective operations based on MVICH over cLAN VIA and better performance over Berkeley VIA on Myrinet. Performance results of the NAS parallel benchmarks show that the *on-demand mechanism* performs better than the *static mechanism* over Berkeley VIA on Myrinet. The *on-demand mechanism* can deliver comparable performance for these benchmarks to *static mechanism* using polling over cLAN VI over cLAN VI. It even performs better for certain benchmarks.

We also addressed how to design and implement the *on-demand mechanism* to maintain MPI semantics, determinism, predictability, and fairness. We believe that the *on-demand mechanism* is a feasible solution to address one important current scalability limitation in the implementation of MPI on VIA-based networks.

Truly large-scale application performance evaluation is a natural extension of this work. Combination of on-demand connection establishment and dynamic flow-control on each VI connection is another planned work.

Acknowledgments

We thank Darius Buntinas and fellow students in the NOWlab for the many profitable discussions they have undertaken with us.

References

[1] M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via/.

[2] Mellanox Technologies. http://www.mellanox.com.

[3] A. Begel, P. Buonadonna, D. Gay and D. Culler. An Analysis of VI Architecture Primitives in Support of Parallel and Distributed Communication. *to appear in Concurrency and Computation: Practice and Experience*, 2002.

[4] M. Banikazemi, J. Liu, S. Kutlug, A. Ramakrishna, P. Sadayappan, H. Sah, , and D. K. Panda. VIBe: A Microbenchmark Suite for Evaluating Virtual Interface Architecture (VIA) Implementations. In *IPDPS*, April 2001.

[5] M. Banikazemi, V. Moorthy, L. Hereger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP Switch-Connected NT Clusters. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 33–42, 2000.

[6] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro*, pages 21–28, Feb. 1995.

[7] P. Buonadonnaa, A. Geweke, and D. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of the Supercomputing (SC)* , pages 7–13, Nov. 1998.

[8] Compaq, Intel, and Microsoft. VI Architecture Specification V1.0, December 1997.

[9] F. S. Daniel. Comparing MPI Performance of SCI and VIA. In the Proceedings of SCI-Europe 2000, August 2000.

[10] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.

[11] Emulex Corp. cLAN: High Performance Host Bus Adapter, September 2000.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[13] J. Hsieh, T. Leng, V. Mashayekhi, and R. Rooholamini. Architectural and performance evaluation of giganet and myrinet interconnects on clusters of small-scale SMP servers. In *Supercomputing*, 2000.

[14] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.

Table 3. Actual CPU Time of NAS Parallel Benchmarks using *static connection mechanism* **and** *on-demand connection mechanism***.**

MIVCH on cLAN VIA			
	static-spinwait	on-demand	static-polling
CG.A.16	4.58	4.56	4.47
CG.B.16	155.37	152.95	152.64
CG.A.32	3.97	3.1	2.87
CG.B.32	132.49	128.97	125.5
CG.C.32	290.01	287.55	289.25
MG.A.16	4.62	4.57	4.7
MG.B.16	21.81	21.23	21.69
MG.A.32	3.91	3.82	3.94
MG.B.32	18.4	17.37	18.48
MG.C.32	154.7	153.66	153.9
IS.A.16	1.5	1.51	1.5
IS.B.16	6.71	6.7	6.57
IS.A.32	1.31	1.29	1.26
IS.B.32	5.7	5.68	5.52
IS.C.32	25.23	25.06	25.06
SP.A.16	100.46	100.61	100.47
SP.B.16	531.51	528.24	525.62
BT.A.16	183.17	183.46	183.04
BT.B.16	826.64	824.06	820.92

MVICH on Berkeley VIA			
		on-demand	static polling
IS.A.8		1.98	1.99
IS.B.8		8.29	8.29
CG.A.8		6.36	6.44
CG.B.8		203.24	205.01
CG.A.4		10.76	10.96
IS.A.4		3.7	3.69
BT.A.4		552.13	552.1
SP.A.4		419.45	420.14

[15] Jiesheng Wu, Jiuxing Liu, and Dhabaleswar K. Panda. Impact of On-Demand Connection Management in MPI over VIA. Technical Report, CIS depart., The Ohio State University, April 2002.

[16] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. Designing Clusters with InfiniBand: Early Experience with Mellanox Technology. Submitted for publication, March 2002.

[17] L. L. N. Laboratory. MVICH: MPI for Virtual Interface Architecture, August 2001.

[18] Massimo Bertozzi, Marco Panella, Monica Reggiani. Design of a VIA based communication protocol for LAM/MPI suite. In *9th Euromicro Workshop on Parallel and Distributed Processing*, Sept. 2001.

[19] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[20] NASA. NAS Parallel Benchmarks.

[21] H. Ong and P. A. Farrell. Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network. 4th Annual Linux Showcase and Conference, Atlanta, October 2000.

[22] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[23] Philip J. Mucci. LLCbech. http://icl.cs.utk.edu/projects/llcbench/, July 2000.

[24] L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. *In Proceedings of the International Parallel Processing Symposium Workshop on Personal Computer Based Networks of Workstations*, 1998. http://lhpca.univ-lyon1.fr/.

[25] R. Brightwell and A. Maccabe. Scalability limitations of via-based technologies in supporting mpi. In the Proceedings of the Fourth MPI Developer's and User's Conference, March 2000, March 2000.

[26] Robert can de Geijn, David Payne, Lance Shuler and Jerrell Watts. A Streetguide to Collective Communication and its Application. http://www.cs.utexas.edu/users/rvdg/pubs/streetguide.ps, Jan 1996.

[27] Rossen Dimitrov and Anthony Skjellum. An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. http://www.mpi-softtech.com/publications/default.asp, 1998.

[28] ServerNet. http://www.servernet.com.

[29] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with lapi - a new high performance communication library for the ibm rs/6000 sp. *International Parallel Processing Symposium*, March 1998.

[30] J. S. Vetter and F. Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *IPDPS*, April 2002.

[31] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.

[32] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.

[33] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of Hot Interconnects V*, Aug. 1997.

[34] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the nas parallel benchmarks. In *In the Proceedings of Supercomputing'99*, 1999.

[35] Xin Liu and Andrew Chen. Performance Evaluation of a Hardware Implementation of VIA. Technical Report. http://www-csag.ucsd.edu/papers/hpvmfm-p.html, June 1999.