SCI-PVM: Parallel Distributed Computing on SCI Workstation Clusters *

Ivan Zoraja
Department of Electronics and Computer Science
University of Split, 21000 Split, Croatia
zoraja@split.fesb.hr

Hermann Hellwagner
Institut für Informatik, Technische Universität München
D-80290 München, Germany
hellwagn@informatik.tu-muenchen.de

Vaidy Sunderam
Department of Math and Computer Science
Emory University, Atlanta, GA 30322, USA
vss@mathcs.emory.edu

Abstract

Workstation and PC clusters interconnected by SCI (Scalable Coherent Interface) are very promising technologies for high performance cluster computing. Using commercial SBus to SCI interface cards and early system software and drivers, a two-workstation cluster has been constructed for initial testing and evaluation. The PVM system has been adapted to operate on this cluster using raw device access to the SCI interconnect, and preliminary communications performance tests have been carried out. Our preliminary results indicate that communications throughput in the range of 3.5 MBytes/s, and latencies

*Research supported by the Applied Mathematical Sciences program, Office of Basic Energy Sciences, U. S. Department of Energy, under Grant No. DE-FG05-91ER25105, the National Science Foundation, under Award Nos. ASC-9527186 and ASC-9214149, and the German Science Foundation SFB342.

of 620 μs can be achieved on SCI clusters. These figures are significantly better (by a factor of 3 to 4) than those attainable on typical Ethernet LAN's. Moreover, our experiments were conducted with first generation SCI hardware, beta device drivers, and relatively slow workstations. We expect that in the very near future, SCI networks will be capable of delivering several tens of MBytes/s bandwidth and a few tens of microseconds latencies, and will significantly enhance the viability of cluster computing.

1 Introduction

Cluster computing refers to the use of interconnected collections of workstations as a unified concurrent computing resource. Such network-based computing platforms have become widely prevalent, and have evolved into very effective and viable environments for high performance scientific computations, general purpose data processing, replicated web and internet servers, and for commercial and business applications. Typically, these computing environments are based upon hardware consisting of a collection of (homogeneous or heterogeneous) workstations interconnected by a local area network, and software libraries and tools that offer different types of programming models. The most common software frameworks are those that present a process-oriented message-passing interface, e.g. PVM [1] or MPI/P4 [6]. While the computers and software in clusters have established a pattern of growth and enhancement, the interconnection networks are only now undergoing major changes, with the advent of various technologies such as ATM, FDDI, fast Ethernet, DQDB networks, and SCI.

The Scalable Coherent Interface (SCI) standard is a point-to-point technology that offers backplane bus services and that can, in fact, serve as a high speed interconnect for LAN-based high-performance cluster computing [3]. An IEEE standard since 1992 [4] [5], SCI proposes innovative solutions to the bus signaling and bottleneck problems and to the cache-coherence problem, and is capable of delivering 1Gbit/s bandwidth and latencies in the 10 μs range in local networks. These characteristics, coupled with the potential for inexpensive hardware implementations, make SCI very attractive for clusters, especially since most of the other high-speed interconnect technologies such as ATM and FCS only improve current bandwidth and latency by one order of magnitude or less. SCI, on the other hand, is designed to be two orders of magnitude superior, both in terms of latency and bandwidth. Further, the SCI architecture is scalable, and the bus-adapter model can be realized in an inexpensive manner.

In order to perform an early exploration of the potential for SCI-based clusters, we have undertaken an experimental project to implement and test a message passing system on SCI. The system chosen is PVM, a widely used standard software framework for parallel and distributed computing on networked computing environments. PVM was selected because of its high portability and reliability, and the existence of a very large user base and application pool. Our project goals are to extend and enhance PVM, in several modular phases, to operate over an SCI interconnection network, and to measure its effectiveness in terms of functionality and performance. We have completed the first major phase of this project, and our experiences and results are described in this paper. We include some background material on SCI, a description of the key features of our implementation, performance results, and a discussion on our findings.

1.1 The SCI Standard

The Scalable Coherent Interface (SCI) emerged from an attempt in the late 1980s to design and standardize a very high performance computer bus ("Superbus") for the next generation of high-end workstations and multiprocessor servers. To address scalability and signaling limitations, a distributed solution was adopted, and is currently manifested as SCI. The SCI interconnect technology has a number of attractive characteristics and capabilities, including:

- Scaling: SCI supports traditional computer bus-like services, yet in a fully distributed manner, enabling scalability and alleviating the need of central clocks.
- SCI is based on unidirectional, point-to-point links only, avoiding any back-propagating flow control – allowing high speeds, large distances, and differential signaling. Currently, the standard defines 1 Gbit/s and 1 Gbyte/s link bandwidth, using bit serial and 16-bit parallel transmission, respectively. With parallel fiber optic links, 2 Gbytes/s transmission rates over distances of up to 100 meters have been demonstrated [7].
- SCI is based on split-transaction protocols and packet switching. Multiple independent transfers can be in transit in a SCI system simultaneously, emanating from different nodes or pipelined into the network from a single node. The standard specifies that each node may have up to 64 outstanding transactions.
- SCI uses a 64-bit addressing model, with the most significant 16 bits addressing a SCI node, and the remaining 48 bits representing the address offset within the node. Thus, up to 64k nodes can be included in a SCI system. A SCI node may be a full desktop or multiprocessor machine, a processor (together with

its caches), a memory module, a switch, an I/O controller or I/O device, or a bridge to other buses or interconnects. A standard node interface structure is defined by SCI.

- The SCI standard particularly supports distributed shared memory (DSM) systems. Nodes may share memory segments across node boundaries and perform remote memory accesses employing hardware mechanisms only. Since no operating system or network protocol software is involved, this type of communication has very low latencies (in the low μs range), even In a distributed system. Atomic read-modify-write transactions (e.g. compare&swap) are specified to support efficient synchronization when DSM is used in a multiprocessor system.

- To support local caching of remote data, SCI provides sophisticated cache coherence mechanisms based on a distributed directory scheme (sharing lists). It must be noted that these mechanisms are *optional* and do not sacrifice performance of accesses to non-cacheable memory regions.

- The basic topological building block of a SCI network is a small ring (a ringlet), containing a few (for instance, eight) nodes. Using ringlets, arbitrary network and switch topologies can be constructed for large systems, e.g. rings of rings, but multistage interconnection networks as well.

- SCI defines additional elaborate protocols for e.g. broadcast transactions, ring bandwidth allocation (the SCI equivalent to bus arbitration), and error detection and containment. Packets are protected by CRCs and, in case they are lost or transmitted with errors, are recommended to be re-sent using hardware mechanisms. All protocols are designed to avoid deadlocks, provide fairness, and guarantee forward progress.

Owing to its unique characteristics, SCI can be applied to a wide range of interconnect problems in the design of high performance computing systems. For example, RAM interfaces, which link memory closely to processors for very high memory bandwidth required by future microprocessors, can exploit SCI. Similarly, in tightly coupled, cache coherent multiprocessors, SCI typically interfaces to and extends the *processor-memory* bus/interconnect of the nodes to connect them into a scalable shared-memory cluster (SMC) e.g. as in the Convex Exemplar SPP 1000/1200. SCI is also effective in data acquisition systems and I/O subsystem interconnects.

SCI can also be very valuable in situations not traditionally in the realm of hardware buses. In loosely coupled, non-cache coherent compute clusters, SCI typically attaches to the *I/O* bus to interconnect nodes in a way similar to a traditional LAN. This permits very high speed clusters and facilitates efficient concurrent computing in network environments. Apart from SCI's higher bandwidth, the major difference is that SCI can be used to set up DSM segments among participating nodes, thereby providing the low latency communication option mentioned above. Boards to interface SCI to standard I/O buses like the SBus or PCI are or will soon be available on the market, e.g. from Dolphin Interconnect Solutions (see [8]). These allow off-the-shelf workstations or even PCs to be clustered into high performance parallel computing devices.

2 The SCI Experimental Testbed

The testbed for SCI software development that was used in this project is a SCI "ring" of two SPARCstation-2 (SS-2) machines, interconnected by commercial 25 MHz Dolphin SBus-SCI adapter cards and SCI copper cable from Dolphin Interconnect Solutions, Oslo, Norway[8]. The device driver for this product (V1.4a, May 1995) supports two elementary software interfaces.

The *raw channel interface* (message-passing) provides system calls for establishing and controlling connections between processes on different machines. Data can then be transferred across connections using `write` and `read` system calls. Raw I/O channels are simplex; separate unidirectional connections are created if both communicating processes wish to send and receive messages from each other. The SCI adapter uses programmed I/O for messages smaller than 64 bytes, or if the message is not aligned on a 64-byte boundary. For larger messages (if properly aligned), a DMA engine conducts the data transfer and enables high throughputs to be achieved. It is emphasized by the vendor that, to achieve low latencies, only a single intermediate kernel buffer (at the receiver side) and thus only a single intermediate copy operation is involved in a data transfer.

The *shared-memory interface* consists of system calls for constructing shared-memory segments as well as setting up address mappings in the interface so that remote segments can be accessed transparently. Processes can include shared segments in their virtual address spaces using the `mmap` system call. Processes can move data transparently from the local memory of one machine to local memories on other machines by simple assignment operations onto shared segments. This is effected by the DMA engine; no system call or intermediate kernel buffer is involved in this kind of data transfer. The SCI links are nominally rated at 1 Gbit/s bandwidth, while the interface cards deliver sustained application throughput of 10 Mbytes/s.

It must be noted that the device driver functionality is far from complete. For the raw channel interface, only a minimal and slow implementation of the `poll` system call is available. The shared memory interface lacks atomic, SCI-based read-modify-write transactions to build efficient locks or semaphores across node boundaries. No fast signaling mechanism over SCI is available. These deficiencies have a severe impact on the implementations of the packages and, in particular, on their performance.

3 Implementation

The PVM system provides a general library-based message passing interface to enable distributed memory concurrent computing on heterogeneous clusters. Message exchange as well as synchronization, process management, virtual machine management, and other miscellaneous functions are accomplished by this library in conjunction with PVM daemons that execute on a user defined host pool and cooperate to emulate a parallel computer.

Data transfer in PVM is accomplished with the `pvm_send` and `pvm_recv` calls that support messages containing multiple datatypes but require explicit `pvm_pk<type>` and `pvm_upk<type>` calls for buffer construction and extraction. Alternatively the `pvm_psend` and `pvm_precv` calls may be used for the transfer of single data type buffers. Both methods account for data representation differences, and both use process "taskid" and message tags for addressing and discrimination. In addition, a runtime option allows message exchange to use direct process-to-process stream (i.e. TCP) connections, rather than transfer data via the daemons as is the default. Therefore, `pvm_psend` and `pvm_precv` with the "direct routing" option is the fastest communication method in PVM. Our project to implement PVM over the SBus-SCI interface consists of several phases:

- Implement the equivalent of `pvm_psend/pvm_precv` as a non-intrusive library over SCI using raw I/O interface (DMA), i.e message passing. Complete this implementation to incorporate heterogeneity, fragmentation/reassembly, Fortran stubs, optimization and fine tuning.

- Incorporate the use of SCI shared memory for data transfers into this implementation. Tune this implementation by utilizing shared memory communication or message passing, respectively, where most efficient.

- Implement the native PVM *send/receive* calls with intelligent decision making to utilize SCI when available and to fall through to default protocols otherwise.

- Investigate the feasibility and desirability of implementing PVM control functionality and the collective communication routines over SCI.
- Enhance the functionality and performance of the library using a threads-based parallelism model.
- Port the above to PC-based compute clusters interconnected via the forthcoming PCI-SCI interface card.

In the initial phase of this project (the first two action items), `pvm_scisend` and `pvm_scirecv` counterparts to direct routed `pvm_psend` and `pvm_precv` have been designed and implemented as a non-intrusive library to PVM. In the interest of compatibility, the argument lists and semantics of `pvm_scisend` and `pvm_scirecv` have been taken to be identical to *psend/precv*; thereby programs may be trivially translated to use either of the two mechanisms. The argument list for `pvm_scisend` consists of the destination taskid (multicast is not currently supported), a message tag, a typed buffer, the number of items to be sent, and a coded datatype indicator:

```
int info = pvm_scisend (int tid, int msgtag, char *buf, int len,
            int datatype)
```

The routine `pvm_scisend` is asynchronous, so computation on the sending processor (i.e. return of control to the invoking process) can resume as soon as the user send buffer is safe for reuse. This routine does not affect the active send buffer and the messages sent by this routine can be only received by the `pvm_scirecv` routine. The first five parameters to `pvm_scirecv` are identical, with the exception that wildcards are allowed for the source taskid and message tags; in addition, there are three output parameters for the return of the actual taskid, message tag, and message length:

```
int info = pvm_scirecv (int tid, int msgtag, char *buf, int len,
            int datatype, int *atid, int *alen, int *atag)
```

The routine `pvm_scirecv` is a blocking call; therefore, the invoking process is suspended until a message matching the user specified tid and msgtag arrives on one of the SCI-PVM connections. If the message has already arrived then `pvm_scirecv` returns immediately with the message. This routine also does not affect the active receive buffer. Return values from both calls are identical to those for *psend/precv*.

3.1 Implementation Model

The SCI-PVM library presents an API and is the application task's interface to the local PVM daemon and to other tasks. As depicted in Figure 1, the SCI-PVM library consists of two functional parts: the message queue (and associated routines) to store received messages, and the communication driver to choose the route and send outgoing messages. The route via the PVM daemon is used to exchange control messages, and also for regular messages in case the SCI connection is not operational or not installed. This "fallthrough" is accomplished by utilizing the `pvm_send/pvm_recv` PVM calls, using a special message tag in a reserved range. To distinguish regular messages from control messages, additional internal tags are used.

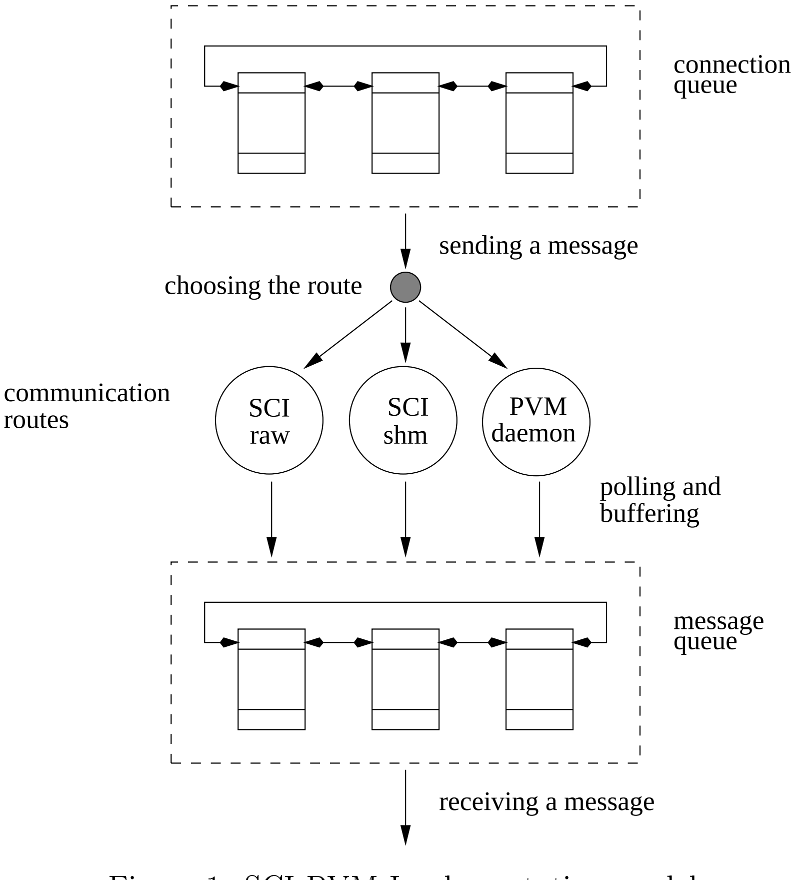

Figure 1: SCI-PVM Implementation model

For the actual data transfer over SCI channels, the SCI-PVM library transparently employs whatever SCI communication mechanism is most efficient. For large data transfers, raw channel access to the SCI interconnect (i.e. message passing using the `write/read` system calls) is chosen in order to achieve high throughput. Small messages, for which low latency is of primary importance, are transferred via a shared memory segment, at the user level. Performance evaluation has indicated that a threshold of 1024 bytes is best suited to switch between these SCI routes. Based on this parameter, the sender chooses the route and transmits messages by writing directly to the active end of the connection. For each message, a header containing the specific information for the particular route is first sent to permit the destination to prepare for reception; the actual message is then written, with fragmentation carried out when needed.

Implementation of the `pvm_scirecv` function involves accepting connections, and subsequently reading headers and data from the passive end of the connection. In addition, the receiver library performs multiplexing among many incoming SCI-PVM communication routes (channels), performs discrimination based on source and message tags, and also reads in and buffers early messages. To preserve PVM source/tag filtering semantics and to avoid retries and possible deadlocks, all incoming messages are first read in and buffered whenever the `pvm_scisend/pvm_scirecv` libraries get program control.

The connection queue keeps track of SCI established connections between the two tasks, as well as connections between the tasks that do not have SCI capability (i.e. card not installed or operational). Each descriptor contains the peer's PVM task identifier, the SCI node number, the SCI kernel buffer key, and the SCI shared memory segment key. It also contains a pointer to the shared memory segment, its size, a file descriptor and a pointer to the message being currently received. Negative values in the SCI node number field indicate the receiving connection descriptor or the sending connection descriptor to a task without SCI interconnect.

The message queue stores messages until they are moved to the user receive buffer. All information needed to reassemble the message are kept in the message structure, which is filled from the header of the message and contains the sender PVM task identifier, a message tag, a data type, a pointer to the message buffer, a pointer to the current fragment and a flag indicating if the message is completely received. When the message is complete, the message structure is removed from the connection queue and other messages from the same sender can now be received.

3.2 Communication Model

The SCI-PVM routines have been designed to operate on *any* node, even on those without an SCI interconnect. Therefore, as depicted in Figure 2, six logical communication routes must be handled. SCI channels are used whenever possible. At the first transmission attempt (using `pvm_scisend`) for a specific source-destination pair, a special connection request message is sent via the default PVM mechanism. The corresponding `pvm_scirecv` either accepts or rejects this connection request.

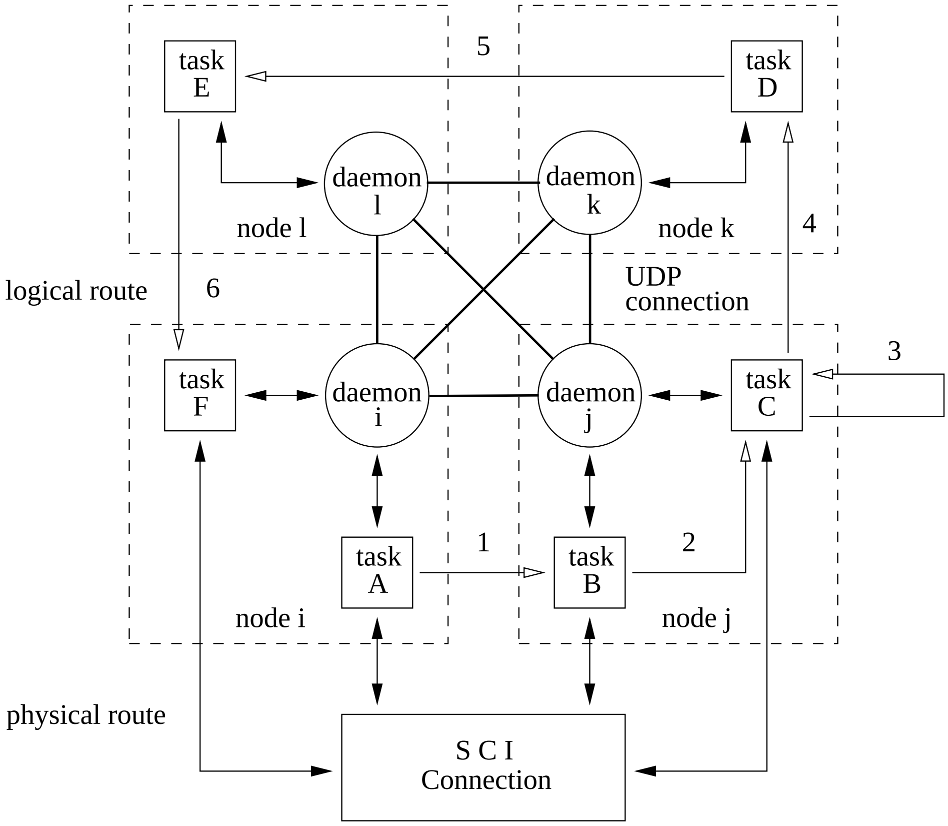

Figure 2: SCI-PVM Communication routes

When both tasks have SCI interconnects, the receiver (or passive task) allocates a port number, creates a shared memory segment, maps the shared memory segment into its virtual address space, creates a connection descriptor, and sends an acknowledgment message to the requester (i.e. the sender or initiator of the connection). This control message contains the SCI node identifier, the SCI port number, and the SCI shared memory key. The requester does not wait for an acknowledgment; it polls all SCI-PVM routes and buffers new incoming messages from other tasks in the message queue. On receipt of the acknowledgment message, the sender connects to the remote SCI node, maps the SCI shared memory segment into its virtual address space, creates its connection descriptor, and sends the message using one of the SCI interfaces.

If for any reason there are problems obtaining resources, i.e. file descriptors, the message will be routed through the PVM daemon. In such a circumstance, the sender does not request a SCI connection. If the sender successfully obtains SCI resources but the receiver does not, the connection is rejected by sending an appropriate control message back to the requester. However, an optimistic approach is adopted – subsequent SCI requests will repeatedly attempt to establish a connection over the SCI interconnect. Finally, in the case that two tasks simultaneously try to establish an SCI connection to each other, a simple convention (lower PVM task identifier is passive) is used to avoid a potential deadlock situation.

With reference to Figure 2, the message passing mechanisms for various other cases are now clarified. Tasks B and C reside on the same machine; message exchange between them is performed over the PVM daemon. When a message is to be sent to oneself (task C), only a local copy from the user send buffer to the message queue suffices. If C attempts a SCI connection to D, the latter will reject the request with an appropriate message, indicating that SCI interconnect is not installed on that node. In this instance, the sender will create a special descriptor for this connection and will refrain from future attempts to set up a SCI channel to this receiver. When a processor does not have SCI interconnect, a connection attempt is not requested; the message is immediately sent over the PVM daemon. This is the case when task D sends a message to E, or when task E sends a message to F. To improve performance for long bursts of sustained message sending, the last receiver task identifier is kept by the sender, so that these messages can be sent without consulting the connection queue.

3.3 Raw Channel Route

The SCI-PVM library uses the SCI driver `write` and `read` system calls for the transfer of large messages. The `write` system call copies the content of the user send buffer to the remote kernel receive buffer, and the `read` system call reads data from this kernel buffer (from the reader's point of view it is local) to the user receive buffer. When data are properly aligned, a DMA conducts data transfer; otherwise, programmed I/O is performed.

In the current (1996, Solaris 2) version of the SCI driver, a DMA transfer takes place only when three conditions are met: (a) the user send buffer must be aligned at a 64-byte boundary address; (b) the size of the message must be a multiple of 64 bytes; and (c) the start address in the remote kernel buffer must also be aligned on a 64-byte boundary. To ensure that these conditions are fulfilled, our implementation divides the message into three parts, and writes it, as shown in Figure 3.

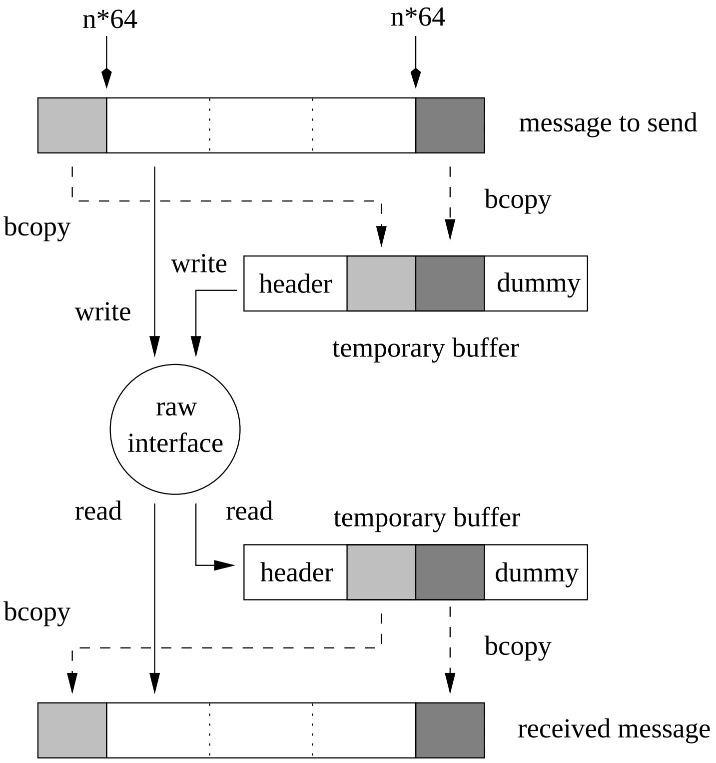

Figure 3: Raw channel data transfer

To permit the destination to prepare for reception, a header containing the message tag, the number of items to be sent, a data type indicator, the preample (the part of the message to the first alignment), the residue, their sizes, and some padding, is first sent. At the other end of the connection, the receiver polls all receiving file descriptors in the connection queue. On receipt of the header it creates a message structure and fills it with information from the header, allocates a buffer for the message, copies the two (unaligned) parts of the message into the buffer, and labels the message “not complete”. After receiving the header, the receiver does not wait for the rest of the message; rather it continues polling on all SCI-PVM routes.

In a SCI-PVM implementation catering to parallel programs where each process communicates with multiple partners, polling must be artificially implemented to avoid indefinite delays and deadlocks. Because of the limited space of the kernel buffer (128 kbytes), writes could potentially block, perhaps unnecessarily holding up the sender. In this version, we have circumvented this problem by marking the SCI file descriptors as non-blocking and fragmenting the message in pieces of 42 kbytes. Instead of waiting for the reader to empty the buffer when it is full, the writer does also polling on all incoming routes and buffers early messages. This is combined with an exponential backoff scheme – delaying the writer by successively increasing amounts of time. A potential deadlock occurring when two tasks are sending large messages to each other at the same time is prevented using this method. An alternative method would have been to use tokens.

Completely received messages are moved from the connection queue and reside only in the message queue, wherefrom they are copied to the user receive buffer. If only SCI raw channel interfaces were used, small messages (less than 128 bytes) could be piggybacked onto headers to avoid multiple system calls and context switches.

3.4 Shared Memory Route

Data transfer over the SCI raw channel interface suffers from one major drawback; the transmission latency for small messages is very high. This is primarily caused by the system calls and context switches involved in every data transfer. Also, aligning the send buffer entails additional copy overhead in the library, and the need to transfer dummy bytes decreases performance as well. In order to improve performance for small messages, shared memory segments are used to emulate message passing. As depicted in Figure 4, each shared memory segment contains variables to ensure mutual exclusion, variables for the management of buffer usage, and the message store. Two solutions, described below, have been implemented for synchronization.

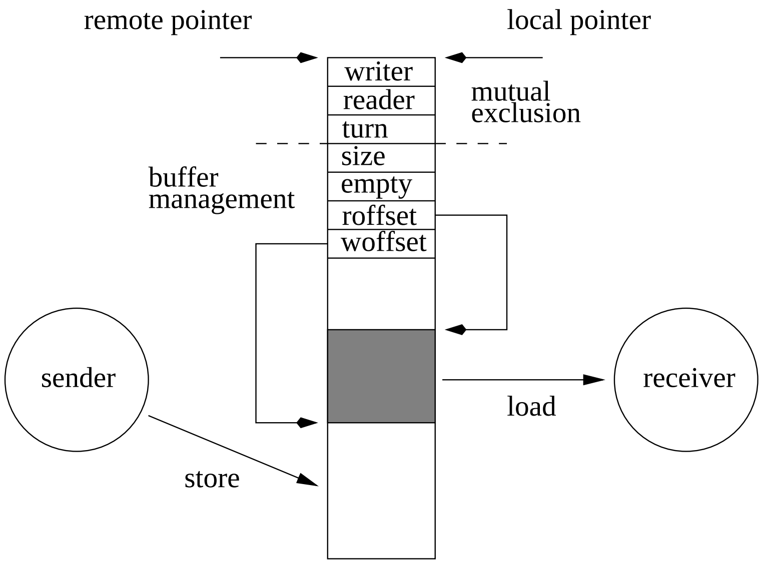

Figure 4: Shared memory data transfer

The first solution to achieve mutual exclusion between two processes uses an integer variable *turn* and requires that two processes strictly alternate in entering the shared memory segment. The variable *turn* keeps track of whose turn it is to enter the segment. Entering the segment, the process first examines *turn*. If it is its turn, the process enters the segment; if it is not, the process continuously examines (spinning on) *turn*, waiting for its value to be changed. When leaving the segment the process sets *turn* and allows the other to enter the segment. Using this solution, the sender writes a message in the segment and waits for the receiver to read it. When the message is read the sender is allowed to write the next message. No buffer management variables are needed in this scheme; however, it works only when both processes are equally matched in terms of speed.

Using Peterson's solution [11] to achieve mutual exclusion, the number and size of the messages are only limited by the size of the shared segment. The *writer* and *reader* variables indicate the interest of the processes to enter the segment. If only one process is interested in entering the segment, it indicates that by setting its interest variable and placing its process number (writer 0, reader 1) into the variable *turn*, and enters the segment because the other has not shown an interest by setting its variable. The variable *turn* arbitrates when both processes try to enter the segment at the same time. The process who first stores its process number into variable *turn* will win the conflict and enters the segment. The other loops until the first one leaves and resets its interest variable.

In both solutions processes waste processor time, spinning on the variable before they are allowed to enter the shared segment. In the case of the writer, the spinning is done on the remote segment, and interferes with the reader's local memory accesses. Both previous algorithms can be improved to reduce remote spinning, by introducing additional local variables on the writer's side, that can be accessed by the reader. Also, due to the lack of efficient SCI-based locking mechanisms, other synchronization primitives are very expensive to implement.

The current SCI-PVM implementation uses Peterson's solution to achieve mutual exclusion. Sending and receiving messages using the shared memory route is similar to the scheme used by the raw channel route. The only difference is that no alignment is needed, and so the header does not contain any part of the message. Message and queue management are also done in a similar fashion. When the shared segment is full, the sender does not wait for the receiver to empty it but does polling on all incoming SCI-PVM routes, buffers early messages and takes action in connection establishment requests. On the other end of the channel, the reader does not wait for the message to be completely sent, rather then does the polling as well. Polling on the shared memory route is simply accomplished by checking shared variable *empty*.

3.5 PVM Daemon Route

This implementation uses the PVM daemon to exchange control messages, as well as regular messages when the SCI connection is broken or not installed at all. The sending and receiving scheme is the same as in the other cases. The header and the message are sent using `pvm_send` and `pvm_pk<type>` calls with a message tag in a reserved range. Polling is done by using the `pvm_nrecv` call; the `pvm_bufinfo` call gives the PVM task identifier of the sender. To distinguish control messages from regular ones, additional internal tags such as, "connection request", "connection acknowledge", "connection rejected" and "SCI not installed" are used.

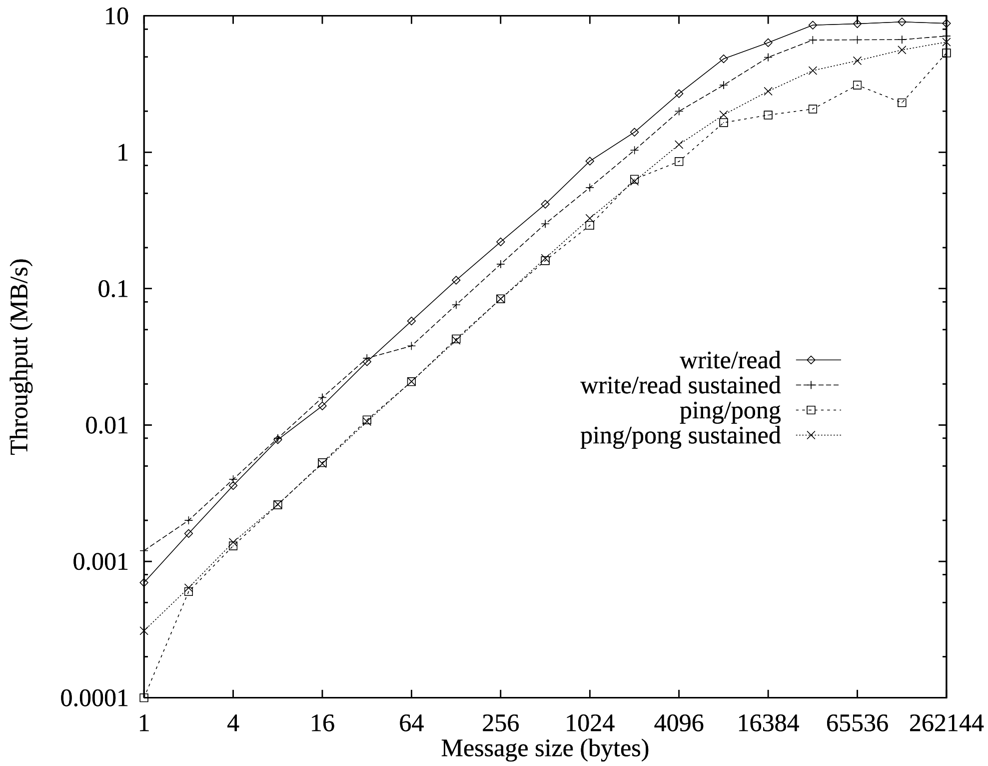

Figure 5: Raw channel throughput

4 Performance Results

The primary objective of the performance studies was to compare direct data transfer over SCI using `pvm_scisend` and `pvm_scirecv` routines and Ethernet using `pvm_psend` and `pvm_precv` routines with the direct route option. The latter are the fastest communication mechanisms available in standard PVM. However, in order to provide some insight into the performance potential and some artifacts of the SBus-SCI adapter as well, results for the raw channel and shared memory interface (without the PVM API) are also included.

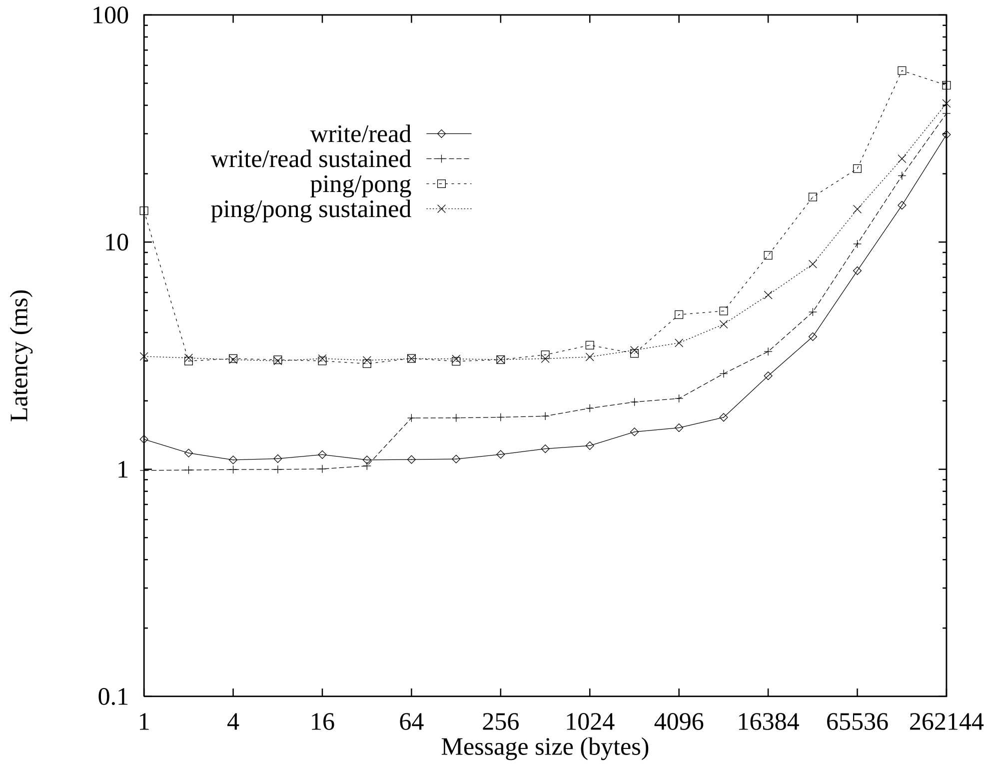

Figure 6: Raw channel latency

The raw channel interface was tested using the *write* and *read* system call to the SCI driver, and throughput performance results are given in Figure 5; Figure 6 depicts latency results for the same test. The measurement was done using a simple *timer-write-timer* program on the sender's side and a *timer-read-timer* program on the receiver's side for one and one thousand iterations (sustained performance). Messages are properly aligned, and the message size was chosen to be a multiple of 64 bytes, but the message header was not sent. For large messages, provided the requisite conditions are met, throughput for one iteration of up to 9.5 MBytes/s was observed. Sustained throughput for the same test of up to 7.1 MBytes/s could also be achieved. For comparison, if one of the conditions is not fulfilled, the throughput peaks at only 160 kbytes/s. For small messages, a minimum latency of 1.1 ms using one iteration could be achieved. The raw channel interface was also tested using a roundtrip measurement

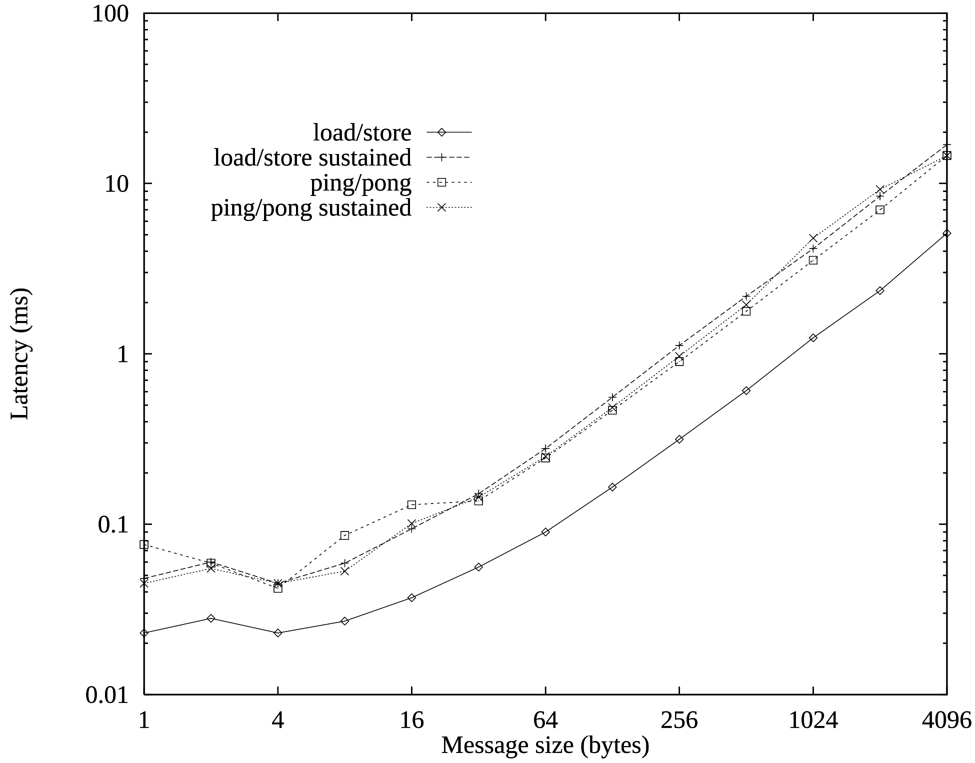

Figure 7: Shared memory Latency - alternation

program, consisting of a simple *timer-write-read-timer* program on the sender's side and a *timer-read-write-timer* program on the receiver's side (ping-pong) for one and one thousand iterations, too. Maximum throughput of 5.7 MBytes/s for one iteration and maximum throughput of 7.1 MBytes/s for sustained ping-pong message exchange could be attained. Minimum small message latency was 3 milliseconds.

The shared memory interface was tested, according to Figure 4 using *load* and *store* instructions for both strict alternation and Peterson's solution implementations of the synchronization mechanism. Latency results are shown in Figures 7 and 8. Simple *timer-store-timer* and *timer-load-timer* programs were used for one and one thousand iterations. The minimum latency achieved with the first algorithm was 23 μs, while that attainable with Peterson's solution was 92 μs. Using *timer-store-load-timer* and *timer-load-store-timer* (ping-pong) programs a minimum latency of 48 μs with the first algorithm and 193 μs by the second were observed. For large messages, the difference between the two algorithms diminishes, and maximum throughput for both peaks at about 800 kbytes/s. The better performance of the strict alternation algorithm was achieved because of the nature of the test program where the speed of

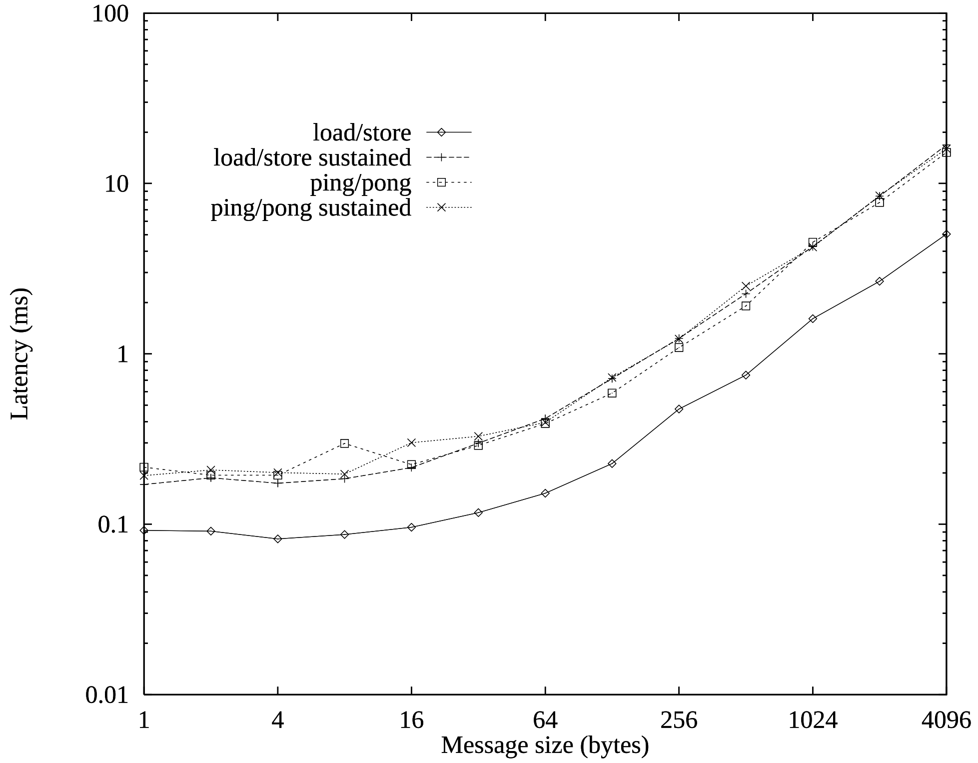

Figure 8: Shared memory latency - Peterson's algorithm

the writer and reader is equal, an unrealistic condition in real applications.

For the final throughput comparison test, simple sustained *timer-send-timer* and *timer-receive-timer* programs were used. Figure 9 shows results with several variants of the *send/receive* functions, message lengths and interconnect media. Experiments were conducted with one thousand iterations of sending and receiving messages. For large messages, sustained throughput over 3.4 Mbytes/s could be attained – a three to four-fold improvement over the fastest Ethernet implementation. However this is only about 50% of the performance that SCI can sustain using the write and read system calls (7.1 MBytes/s). Clearly, performance is diminished by the improved functionality of the SCI-PVM library (e.g. multiplexing, buffering, additional header transmission). However, major causes for the degradation are the deficiencies of the first-generation of the SCI interface hardware and software, such as the critical alignment requirement and an incomplete implementation of the `poll` system call.

The communication route over shared memory segments is clearly superior to the raw channel interface for small data transfers, up to 1 kbyte. Although not shown in our graphs, this effect was even more pronounced when fewer messages (i.e.

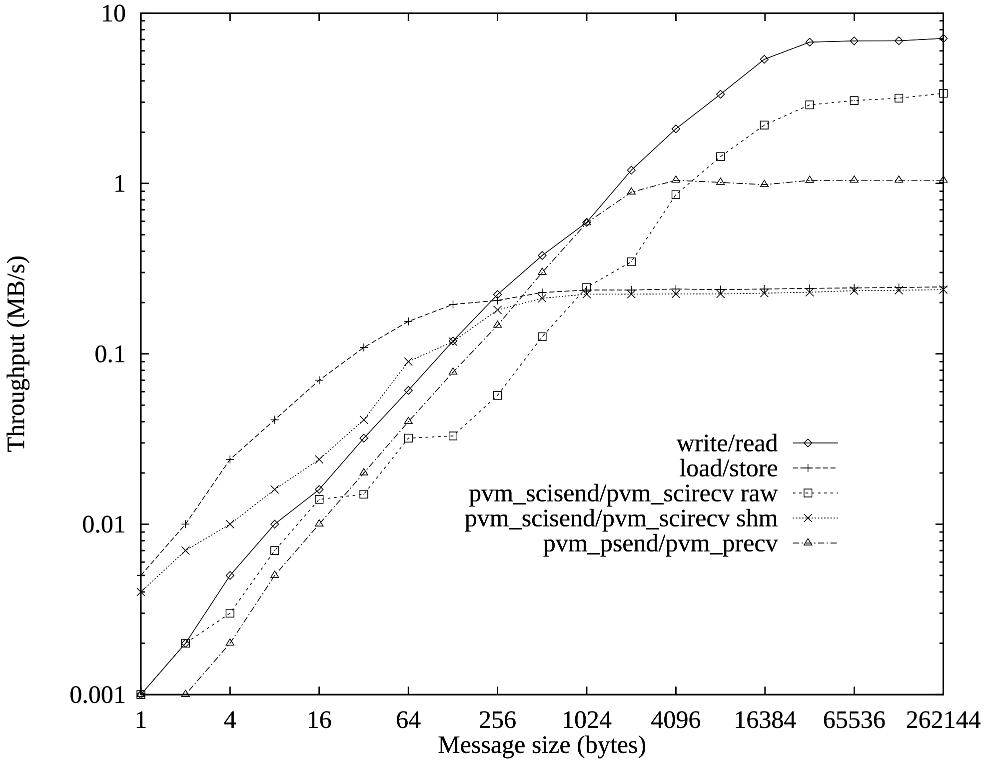

Figure 9: Sustained throughput - comparison

smaller number of benchmark iterations) are sent. Therefore, the SCI-PVM library transparently uses the shared memory route for the messages up to 1024 bytes, and the raw channel route for larger messages. In a ping-pong program, based on shared-memory transfer, latencies of about 620 μs were observed. The maximum throughput and minimum latency comparison results for the previous tests are shown in Figures 10 and 11.

5 Discussion

While the implementation scheme is, for the most part, straightforward, several significant issues should be considered – most pertaining to the current version of the SBus-SCI device drivers. The first, and most critical, is the incompletely implemented `poll` system call for the SCI device (support for this has not been "official", a beta version of the driver was made available). The driver does not implement "pollwakeup" as a result of a "data ready" interrupt, and the "pollwakeup" will only

MB/s	ping	ping sust.	ppong	ppong sust.
shm alter.	*0.871*	*0.247*	*0.292*	*0.281*
shm Pet.	*0.812*	*0.243*	*0.269*	*0.256*
raw	*9.523*	*7.110*	*5.746*	*7.103*
SCI-PVM raw	*7.354*	*3.385*	*2.860*	*2.793*
SCI-PVM shm Pet.	*0.605*	*0.224*	*0.261*	*0.246*

Figure 10: Maximum throughput - comparison

μ sec.	ping	ping sust.	ppong	ppong sust.
shm alter.	*23*	*45*	*48*	*50*
shm Pet.	*92*	*171*	*194*	*193*
raw	*1099*	*1101*	*2917*	*3036*
SCI-PVM raw	*1200*	*1200*	*3569*	*3715*
SCI-PVM shm Pet.	*262*	*386*	*621*	*630*

Figure 11: Minimum latency - comparison

take place as a result of a driver timeout function. In the ping-pong program, in the case when both drivers doing the `poll` system call at the same time, performance can be substantially decreased. This problem can be circumvented by marking the SCI file descriptors non-blocking and attempting to read, with the result values used to determine readiness.

The given throughput results are in line with what was recently reported for 100 Mbits/s ATM LANs [9] [10] whereas our latency figures are significantly better. While the results so far are promising, they do not reach the full potential of the SCI technology. Problems and shortcomings of the first-generation products, in particular the device driver, still limit both ease of use and performance of the SCI interconnect. At the time of writing, Dolphin has indicated that an SBus-2 adapter card for SCI will soon become available and will significAntly increase the bandwidth available to applications (30Mbytes/s throughput). Improvements in device driver functionality and performance will also become available.

6 Further Work

The design and implementation of PVM extensions and enhancements to permit operation over SCI networks have proven to be both successful and enlightening. We have understood in greater depth, a number of issues concerning the SCI interface and operation methods, as well as factors relating to the implementation of message passing over SCI. Our performance results have been very encouraging, especially considering that only early versions of the hardware and drivers were available. These positive experiences strongly motivate further development, and we intend to continue to pursue the PVM-SCI project along several dimensions.

One of the first enhancements to be undertaken is a multithreaded implementation of PVM-SCI. Currently, the necessity to poll for incoming requests on SCI channels is disruptive to local processing, and independent threads of control will lead to a more efficient and elegant implementation. On a related subject, synchronization between SCI nodes interacting via shared memory is also expensive, due to the nature of mutual exclusion implementation. We will investigate schemes to alleviate the busy waiting or remote machine disruption that is currently required.

A third modification involves changing both the PVM daemon and library to utilize SCI when possible for control functions within the PVM system. Similarly, re-implementing the native PVM message passing calls, particularly those that deal with "direct routed" messages, to use SCI when possible, would be of great benefit. The latter two extensions are straightforward to implement, but require changes to the PVM system — as emphasized earlier, our first phase experimental SCI-PVM system was intended to be completely non-intrusive. Finally, some long term work, depending on the evolution and availability of SCI hardware interfaces, will be undertaken to extend SCI-based cluster computing to personal computers.

References

[1] V. S. Sunderam, G. A. Geist, J. J. Dongarra, and R. Manchek, "The PVM Concurrent Computing System: Evolution, Experiences, and Trends", *Journal of Parallel Computing*, **20**(4), pp. 531-546, March 1994.

[2] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, "PVM - Parallel Virtual Machine: A Users Guide and Tutorial for Networked Parallel Computing", *M.I.T. Press*, November 1994.

[3] Gustavson, D.B., Li, Q., "Local-Area MultiProcessor: the Scalable Coherent Interface", *SCIzzL, The Association of SCI Local-Area MultiProcessor Users, Developers, and Manufacturers* (1994).
Contact: `dbg@sunrise.scu.edu`.

[4] Gustavson, D.B., "The Scalable Coherent Interface and Related Standard Projects", *IEEE Micro* **2**(2) (1992) pp. 10–22.

[5] IEEE Std 1596-1992, "Scalable Coherent Interface", *IEEE CS Press* (1993).

[6] W. Gropp, E. Lusk, N. Doss, A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard", *Parallel Computing*, (to appear) 1996.

[7] D. R. Engebretsen, D. M. Kuchta, R. C. Booth, J. D. Crow, and W. G. Nation, "Parallel Fiber-Optic SCI Links", *IEEE Micro* **16**(1), pp. 20–26, Feb. 1996.

[8] Dolphin Interconnect Solutions, "1 Gb/s SBus-SCI Cluster Adapter Card", *White Paper*, 1994. `http://www.dolphinics.no`.

[9] Sheue-Ling Chang, David Hung-Chang Du, Jenwei Hsieh, Rose P. Tsang, and Mengjou Lin, "Enhanced PVM Comunications over a High-Speed LAN", *IEEE Parallel and Distributed Technology* **3**(3),20-32 (Fall 1995).

[10] Honbo Zhou and Al Geist "Faster Message Passing in PVM", *ACM Conf. on High Speed Networks, ACM Press* (1995).

[11] Andrew S. Tanenbaum. "Modern Operating Systems". *Prentice-Hall* (1992).