Low-latency and zero-copy message passing protocols for SCI-based clusters

Fábio A. D. de Oliveira[1], Rafael B. Ávila[1], Marcos E. Barreto[1], Philippe O. A. Navaux[1],
[1] Institute of Informatics
Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre, RS, Brazil
E-mail: {fabreu,avila,barreto,navaux}@inf.ufrgs.br

Abstract—
This paper presents the design and implementation of three message passing protocols, whose development aimed at efficiently exploiting the high-performance capabilities of the SCI interconnect. These protocols are compared to the communication mechanisms adopted by MPI implementations for SCI clusters. The performance of the proposed protocols allows us to state that they are more efficient, in terms of latency and bandwidth, than the correspondent communication strategies employed by existing MPI implementations specifically designed for SCI-connected clusters.

Keywords— **message passing, high-performance networks, cluster computing, SCI.**

I. Introduction

In the last few years, clusters have raised as a platform for the execution of parallel applications, which is at the same time economically viable and technologically comparable to massively parallel processors (MPPs). Following the definition given in [Buy99], a cluster is referred to as a group of homogeneous PCs or workstations — possibly of the SMP kind —, interconnected by a system area network (SAN), like Myrinet [Bod95], Memory Channel [Gil96] and SCI (Scalable Coherent Interface) [IEE92].

The SCI technology, an IEEE standard since 1992, have turned out to be an interesting alternative as high-performance interconnect for clusters. SCI supports very low latencies for short messages — 2–3 μs —, high bandwidth and scalability, being comparable to the Myrinet network, undoubtedly the most widely used interconnect technology in the cluster computing scenario. The main difference between Myrinet and SCI lies on the way communication is done. Whereas Myrinet is essentially a message-passing network, SCI is a hardware-supported DSM (Distributed Shared Memory) platform, in the sense that each node of the cluster can directly get access to memory locations residing on another nodes. This behavior explains the low-latency inherent to SCI, since all communication can be done at user level, through simple CPU loads and stores, without the operating system intervention or the need of additional protocols.

As SCI is gaining acceptance in the cluster computing community, several efforts have been done to offer message-passing parallel programming environments for such architecture, including adaptations of established standards like MPI [MPI94] and PVM [Gei94]. Nowadays, there are two MPI implementations designed for SCI clusters: ScaMPI [Hus99], which is commercially distributed by Scali AS — one of the manufacturers of SCI network interfaces —, and SCI-MPICH [Wor99, Wor00], a freely available MPICH distribution, conceived at RWTH, in Aachen, Germany.

Besides MPI, the PVM standard was also ported to SCI clusters. SCIPVM [Zor99] and PVM-SCI [Fis97, Fis99] are the existing implementations of PVM on top of SCI network. Moreover, in the scope of the SISCI project [Gia98], CML [Her98], a low-level messaging layer, has been developed in order to serve as a basis for PVM and MPI implementations on top of SCI clusters.

In spite of all these efforts, most of the message passing protocols that underlie the aforementioned programming environments are unable to provide a full performance exploitation of the SCI network. PVM-SCI makes no distinction between sending a short or a large message; furthermore, it uses interrupts for signaling the arrival of a message at a destination node. Not surprisingly, the interrupt mechanism increases considerably the latency, mostly for short messages, and the utilization of a traditional `memcpy` routine for message transfers limits seriously the maximum achievable bandwidth. SCIPVM, in turn, follows a somewhat different approach. It adopts two strategies for message transfers, depending on the message size. Short messages are sent by means of explicit `memcpy` over SCI shared segments, whereas large messages are transmitted through DMA. Nevertheless, SCIPVM is also based on interrupts as the mechanism for signaling the arrival of a message and accordingly it cannot obtain a performance near the potential of the SCI technology. The low-level messaging layer CML, as well as PVM-SCI and SCIPVM, lacks of more elaborated communication protocols, but it is clear better than the PVM implementations. CML reduces the latency for exchanging short messages, avoiding the interrupt mechanism, and it transfers large messages via DMA; despite this improvement, however, CML still makes poor use of the SCI high-performance capabilities.

Both MPI implementations for SCI, ScaMPI and SCI-MPICH, are clearly more efficient than the other available similar programming environments. They make use of three different communication protocols, according to the size of the message to be sent. This really special treatment of short and large messages allows the MPI implementations to obtain lower latency and higher bandwidth. However, the employed protocols still bring unnecessary overheads that make impossible to reach a even better performance.

These observations have led us to devise, propose and implement high-performance message-passing protocols specifically for SCI clusters, in order to reduce the overhead to a minimum and get as much as possible from the SCI network. On the one hand the careful use of the SCI adapter's stream buffers can provide very low latencies when dealing with short messages, on the other hand a zero-copy protocol for large messages can produce a rapid increase in the bandwidth and approach it to the limit imposed by the SCI hardware. In the following, we present the main ideas behind the proposed protocols and we compare them with the protocols used by ScaMPI and SCI-MPICH.

The remainder of this paper is organized as follows: in section II we present the main characteristics of SCI, explaining how this interconnect technology works and what are the implications of its behavior to the development of message-passing protocols; section III presents the design and implementation details of the proposed message-passing protocols; in section IV we evaluate the designed protocols, by comparing them with the protocols used by the existing MPI implementations for SCI; finally, section V brings the authors' conclusions and final remarks.

II. Relevant characteristics of the SCI network

If one is willing to find out effective solutions to the problem of high-performance message-passing over SCI, it is mandatory to completely understand some idiosyncrasies of such network. We comment below not only the way communication is carried out by SCI, but also the main issues that must be observed before pursuing efficient communication protocols.

A. Hardware and software platform

The presented work took place on a cluster composed of 4 SMP nodes. The SMP nodes are Dual Pentium-III 500 MHz, each with 256 MB of RAM and Intel BX-chipset. The SCI interconnect is done with PCI-SCI (32 bits, 33 MHz PCI bus) network interfaces, model D312 (distributed with Scali Wulfkits), equipped with SCI link controller LC2 and PSB revision D. The nodes run Linux with kernel 2.2.14 and Scali Software Platform version 2.0.2.

B. SCI global address space and shared-memory segments

In the SCI network, communication relies on shared-memory segments that belong to the 64-bits SCI global address space. The most significant 16 bits of an SCI address specify a node, and the remaining 48 bits address the local memory within that node.

The SCI network interfaces, together with the driver, establish the global address space in the following manner. For example, a given node creates a shared segment in its physical memory and exports it to the SCI network. Other nodes can now import this segment into their I/O address space; in order to do that, each SCI adapter has an address translation table which maintains the mappings between local I/O addresses and global SCI addresses. Further, processes running on the nodes can map a created DSM segment into their logical address spaces.

Once these mappings have been done, the inter-node communication may be carried out by simple CPU loads and stores into DSM segments mapped from remote memories. The SCI network interfaces transparently translate I/O bus transactions into SCI transactions and vice-versa; in other words, the communication is performed totally at user level, without the operating system intervention. The driver is only used for the establishment of DSM segments, not when communication is taking place. It is exactly this support to directly accessing remote memory that is responsible for the low latency nature of SCI and makes it a very interesting choice as high-performance network for clusters.

However, the implementation of efficient communication protocols for message-passing requires more than trivial loads and stores into shared segments. In the sequel, we point out some SCI characteristics that designers of communication protocols for such architecture must keep track of so that the best performance can be achieved.

C. *Remote reads* versus *remote writes*

In SCI, remote reads, i.e. loads from memory addresses residing on remote memory, are approximately ten times slower than remote writes. The problem is that every load operation stalls the processor until the data has arrived; in other words, today's processors are not able to generate non-blocking load operations from main memory. Although the SCI adapters can be configured to perform prefetching, this is not enough to hide the latency inherent to remote reads, because the PCI-bridge — the interface between the PCI bus and the system bus — causes a PCI transaction for every CPU load operation.

Fortunately, remote writes allow a considerably more efficient use of the PCI-bridge, which supports write-gathering. Such an operation amounts to write as much data as possible into a single PCI transaction. Therefore, remote write accesses reach a peak bandwidth that is ten times higher than

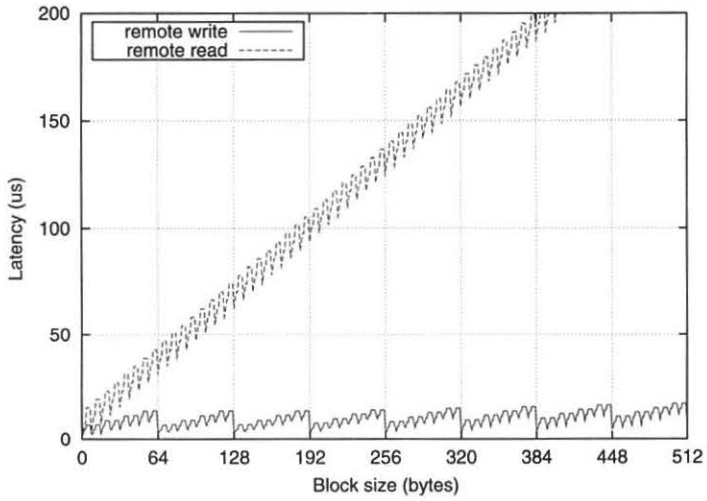

Fig. 1. Latency of SCI remote read and write.

the maximum bandwidth of remote reads.

Figure 1 shows the performance of SCI remote read and remote write, in terms of latency. Due to the observed performance disparity, it is of paramount importance that all data-structures handled by message-passing protocols are placed so that *write-only* protocols can be devised, in the sense that all accesses to DSM segments are necessarily done by means of CPU stores and never through loads. This behavior must be kept track of in the design of high-performance communication protocols for SCI.

D. Stream buffers of the SCI network interface

The PCI-SCI network interface has eight write stream buffers, each with 64 bytes [1]. When a contiguous block of bytes is being written onto a remote address, the block is divided into 64-bytes sub-blocks, and each sub-block is stored in a different stream buffer. In this way, a sub-block can be put onto a stream buffer while another sub-block is being sent to the SCI network, i.e., by combining the eight stream buffers we have a pipeline with eight stages. This technique, implemented by the SCI hardware, is referred to as *stream combining* [Rya96]. As soon as a stream buffer becomes full, it will be readily flushed and a corresponding 64-bytes SCI packet is put onto the network. So, a single SCI transaction corresponds to each completely filled stream buffer. Not surprisingly, the maximum payload of an SCI packet is 64 bytes.

However, a partially filled stream buffer is only flushed in three situations: by means of a command issued to the SCI network interface — explicit flush —; when the accessed address of remote memory is not consecutive to the last one — i.e., the last contiguous block of bytes is over —; or, finally, if the timeout associated with the stream buffer has expired. Furthermore, when a stream buffer not yet full is flushed, a number of 16-bytes SCI packets will be necessary to carry out the communication. For instance, writing 60 bytes onto a remote memory address requires four SCI transactions, whereas the same operation over 64 bytes will generate a single SCI transaction. This behavior explains why the latency curve for remote writes, shown in figure 1, drops abruptly for amounts of bytes multiple of 64 bytes. Also, that is the same reason for the sawtooth appearance of the curve, since the latency for transferring a given amount of data depends on the number of SCI packets — SCI transactions —, and this number does not vary proportionally to the amount of data, but is strictly related to the use of stream buffers. An increment of 64 bytes leads to an increment of one SCI packet, but if the amount of data is incremented by a value not multiple of 64 bytes, more than one SCI packet will be added.

In the light of these observations, it can be noted that to take full advantage of the *stream combining* technique the communication protocols should schedule remote memory accesses in such way that as few SCI transactions as possible are generated. The most intuitive solution is always transferring an amount of bytes multiple of 64, despite the message size from the user point-of-view, making the number of SCI transactions a minimum. Also, all buffers must be aligned on a 64-bytes boundary.

E. How to generate a write burst on the PCI bus

Another question concerning the design of message-passing protocols for SCI is how to carry out communication. At a first glance, the most intuitive way is to do it by means of the standard `memcpy` routine. Nevertheless, the traditional `memcpy`, present in the standard C library, is unable to generate a *write burst* on the PCI bus. To increase the maximum achievable bandwidth, it is necessary to find a way to gather as much data as possible into a PCI transaction, instead of a single word, before submitting it to the SCI adapter. By using MMX stores rather than traditional `memcpy` we can clearly increase the maximum bandwidth, as pointed out in figure 2. Notice that there is an increase from 49.94 Mbytes/s with conventional `memcpy` to 87.72 Mbytes/s with MMX stores. In both cases, observe that the maximum bandwidth is reached for 512 bytes, which is the point where the pipeline composed by the eight stream buffers (8x64 = 512) has all its stages with data.

III. The Proposed Message Passing Protocols

We have designed and implemented three different protocols: a minimal overhead and low-latency protocol, optimized for exchanging short messages; a general-purpose protocol; and a protocol that makes use of a zero-copy commu-

[1] The most recent model of adapters counts on 16 buffers with 128 bytes.

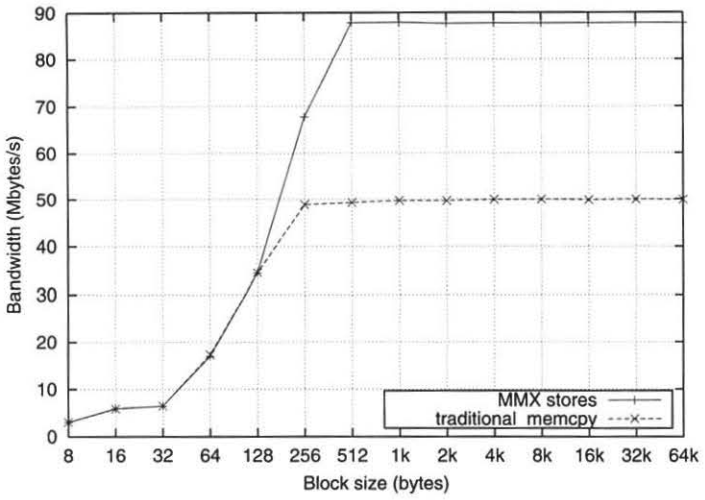

Fig. 2. Bandwidth of SCI remote write.

nication technique developed in order to increase the maximum achievable bandwidth for large messages. The resulting communication kernel is being used in DECK/SCI [Oli01], an environment for parallel programming for SCI clusters, originally developed for Myrinet ones [Bar00], that provides communication and multiprogramming services. DECK/SCI offers the abstractions of mail boxes and messages for communication, besides threads and semaphores for multiprogramming. Each thread can create one or more mail boxes to receive messages from threads residing on another nodes.

Although each protocol has its own specialization and peculiarities, all of them, in order to obtain the best performance, were implemented taking into account the SCI characteristics analyzed in the previous section. Additionally, all protocols avoid using interrupts for signalling the arrival of a message at a destination node; instead, the message-passing is based on polling, so that the latency can be kept low. Basically, the three developed protocols share a couple of characteristics, namely: polling-based message reception; *write-only* communication; use of MMX instructions for remote writes; transfer of blocks of bytes whose size is multiple of 64, in spite of the message length from the user point-of-view. Moreover, all protocols are *thread-safe* and *thread-aware*.

A. *"Protocol 1": short messages*

The latency for transferring short messages is particularly affected by unavoidable extra overheads like signalling of message arrival and flow control schemes. However, these overheads are of paramount importance to the correct operation of a message-passing protocol. For this reason, short message transfers are required to receive special attention from a message-passing library that is willing to ensure very low latency.

Hence, we have devised a special mechanism that optimizes the use of SCI network. This short messages oriented protocol utilizes a single 64-bytes SCI packet to send a message. The last byte of the packet payload contains an identifier that allows the receiver to get notified about the message arrival. In this way, a single remote write is sufficient to transmit the message and notify the receiver. This proposed scheme was in much inspired by the *valid flag* algorithm [Oma97].

Another advantage of the "protocol 1" is the fact that there is no need to explicitly flush the stream buffers, since it always sends 64 bytes, which is important to keep latency low. The message occupies the first 62 bytes of the packet; the 63th byte contains a sequence number, used for message ordering purposes; and the 64th byte is the message identifier used to notify the receiver, as already commented. Thus, the "protocol 1" is suitable to messages whose size ranges from 0 to 62 bytes.

As the message and its corresponding signalling flag are sent into a single SCI packet, it is guaranteed that the packet data arrives exactly in the order it was sent and, since the last byte of the packet is used for notification, when the receiver gets notified the message certainly was completely received.

For the correct work of "protocol 1", every mail box created during the execution of a parallel application reserves, within its SCI shared segment, a separate ring buffer to each process. Each position of a ring buffer maintains 64 bytes, used to store the packet data. Whenever a thread wants to send a short message — 0 to 62 bytes — to a given mail box, it must send a 64-bytes packet — based on the structure commented above — to the current write position related to the ring buffer reserved to the node which it is running on. After the message transfer, the sender thread, by means of a modulo operation, updates its write position as well as the identifier and sequence number of the message to be sent next time the communication primitive is invoked on the related mail box.

The receiver thread, in turn, polls the last byte of the current read position of each ring buffer, until a message has arrived. When the value stored into the last byte of the current read position of a given ring buffer equals to the next expected message identifier for that ring buffer, the receiver thread copies the message to the user buffer in local memory, if the sequence number also matches; at the end, it updates the current read position, as well as the next expected sequence number and message identifier for the appropriate ring buffer, by means of the same modulo operation as that performed by the sender thread.

Additionally, the receiver thread informs the sender about the current read position, by writing it into a predefined address within a previously established shared segment, used

for control purposes, owned by the sender, so that the sender can avoid the ring buffer overrun when sending messages. This is the way flow control is done.

B. *"Protocol 2": general-purpose mechanism*

The "protocol 2" is a more generic message-passing mechanism which can virtually deal with messages of arbitrary sizes. This protocol manages buffers that can store messages greater than those handled by "protocol 1". Again, every mail box reserves a different buffer to each process. The buffers of "protocol 2", in contrast to that of "protocol 1", are not logically divided into pieces of a given size; rather, the messages are contiguously copied into them. Related to each buffer, there is a location where the mail box owner expects a control packet that indicates a message have been transferred to the corresponding buffer.

Internally, the messages handled by "protocol 2" are composed of a header, that contains the message size, followed by the data. To send a message to a mail box, the sender thread first writes it into the buffer reserved to the process which the thread is running on. Before notify the receiver, it is mandatory to flush the SCI adapter's stream buffers, otherwise the signalling packet could be received while some SCI packets of the message are still in transit. This situation could take place because SCI does not ensure packet ordering. In order to overcome this undesirable behavior, "protocol 2" flushes the SCI adapter's stream buffers, waiting for the completion of all outstanding SCI transactions, and only after doing so the sender thread can safely notify the receiver by sending a 64-bytes control packet to the appropriate location within the shared segment of the mail box. Finally, the sender updates the current write position related to the "protocol 2" buffer reserved to the process which it is running on, as well as the sequence number and the identifier of the message to be sent next by means of "protocol 2".

In order to get a message from a mail box, the receiver thread polls all addresses where control packets are expected to be sent to. When a control packet arrives, the receiver thread reads from the proper buffer the message header, pointed by the current read position related to that buffer. After reading the size of the message, its data is copied to the user buffer present in local memory. Then, the receiver updates the current read position and the next expected sequence number and message identifier associated with the recently used buffer. Similarly to "protocol 1", the receiver thread informs the sender about its current read position, for flow control purposes.

Note that this flow control scheme, employed in both protocols, does not require that the sender waits for the information concerning the receiver read position, because communication is done through a remote write operation into a shared segment previously created and exported by the sender. Each process, during initialization, creates its own control segment and maps into its logical address space the control segments of all processes.

C. *"Protocol 3": zero-copy communication*

Although "protocol 2" may be used for exchanging messages of virtually any size, it limits seriously the maximum achievable bandwidth. The disadvantage of "protocol 2" is the fact that it only initiates moving the message from shared to local memory after the message has been completely transmitted. Specially for large messages, this constraint results in poor utilization of SCI bandwidth and cannot be tolerated when the main objective is to accomplish high-performance communication.

The most efficient communication libraries for SCI clusters developed so far, SCI-MPICH and ScaMPI, have two different protocols equivalent to protocols 1 and 2. Further, both MPI implementations adopt the same solution to the relative poor performance of their *eager protocol* — corresponding to our "protocol 2". In order to increase the bandwidth, SCI-MPICH and ScaMPI implement a third protocol, making use of a *rendez-vous mechanism*, the main idea of which is to interleave the message transmission and the copy of the message to the user buffer in local memory. In this way, through a handshaking scheme, the receiver is allowed to copy the message from shared to local memory while the message is still being sent.

Indeed, the mentioned *rendez-vous* protocol is effective in increasing the maximum achievable bandwidth. Nevertheless, it still relies on the message copy from shared to local memory, due to the semantics of the MPI receive primitives which impose that an user-allocated buffer be passed as argument to them.

In DECK/SCI API — the programming interface on top of the proposed protocols —, the message abstraction exists explicitly, being represented by a message object. As the programmer is required to utilize specific DECK/SCI primitives to manipulate messages — creation, packing, unpacking, etc. — and the message buffer is under control of DECK/SCI, it was possible to devise a zero-copy protocol to really increase the maximum bandwidth beyond the values obtained by MPI implementations and near to SCI limits. Of course, even though the message buffer is internally managed by DECK/SCI, the programmer can get its address and use it normally.

Following this idea, "protocol 3" is a zero-copy communication scheme, in the sense that there is no extra copy besides the message transmission. The message is directly sent to the user buffer, which resides on SCI shared memory. When the programmer creates a message, depending on the size passed as argument DECK/SCI will allocate the buffer on local or shared memory. The threshold to the tran-

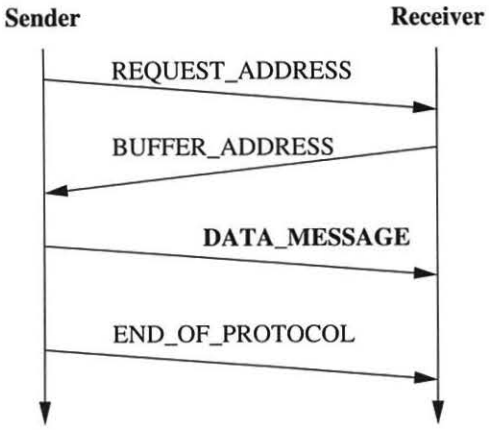

Fig. 3. The proposed zero-copy message passing protocol.

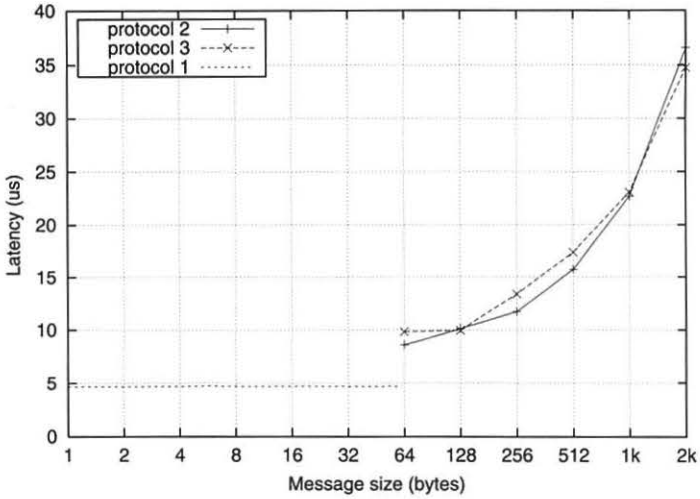

Fig. 4. Latency of protocols 1, 2 and 3.

sition from protocol 2 to 3 is configurable by changing the value of `DECK_MSG_BUF_LIMIT`. Usually, however, this parameter can remain untouched and the user does even not need to know about the multiple communication protocols of DECK/SCI.

As depicted in figure 3, "protocol 3" requires the exchange of some control messages before the actual data transfer. Firstly, the sender thread sends a control message requesting the address of the user buffer from the receiver. By doing polling, the receiver thread gets the request message and then informs the sender about the address which the message is supposed to be sent to. After that, the sender effectively sends the data message to the appropriate address and flush the SCI adapter's stream buffers, waiting for the completion of all outstanding SCI transactions. Finally, the sender signals the end of zero-copy communication by transmitting the last control message. Under the reception of such control message, the receiver can safely return from the communication primitive, as it is guaranteed that the data message was completely transmitted. Again, notice that the signalling message was sent after the flush of stream buffers, which is necessary to cope with reordering of SCI packets, as already stated.

It should be noted that the mail box abstraction remains valid, even when the zero-copy protocol is used. From the user point-of-view, messages are just posted to and retrieved from mail boxes.

IV. Evaluation of the Proposed Protocols

In this section, the performance of protocols 1, 2 and 3 is analyzed in terms of latency and bandwidth. All measures were done through a traditional ping-pong algorithm, with 1000 repetitions for each message size.

Figure 4 shows the latency of the designed protocols. The curves allow one to verify the specialization of each protocol. As can be seen, the latency of "protocol 1", suitable for short messages, is kept below 5 μs — 4.66 μs. When the "protocol 2" takes place, the latency is abruptly increased to 8.56 μs. This was expected, since "protocol 2" sends an additional SCI packet for signaling the message transfer at the destination node. In contrast, "protocol 1" always sends only one 64-byte SCI packet, which carries the message itself and the signaling flag, as stated before.

The latency of "protocol 3" begins with 9.82 μs. The overhead caused by the *handshaking* between sender and receiver has more impact in shorter messages, from 64 to 1024 bytes. From this point on, however, the latency of "protocol 3" is lower than that of "protocol 2", as the extra copy avoidance compensates the synchronization between sender and receiver. From the performance perspective, messages of 1024 bytes represent the ideal threshold for the transition from "protocol 2" to "protocol 3".

Figure 5 exhibits the bandwidth reached by each protocol. The curves of protocols 2 and 3 enforce that the zero-copy mechanism is really necessary to increase the peak bandwidth near to the SCI limit. "Protocol 3" achieves 84.12 Mbytes/s, whereas "protocol 2" is limited to only 62.54 Mbytes/s. This disparity lies on the fact that "protocol 2" waits for the complete message transfer before copying the message to the user buffer; in contrast, "protocol 3" directly sends the message to the user buffer.

Figures 4 and 5 show that multi-protocol solutions are really necessary when developing high-performance communication libraries for SCI, whose main aim is to obtain very low latencies for short messages and high bandwidth for large ones. This cannot be accomplished by just one protocol.

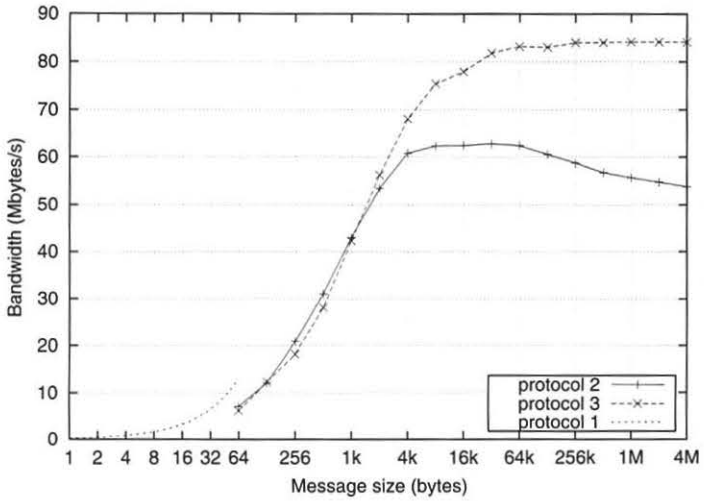

Fig. 5. Bandwidth of protocols 1, 2 and 3.

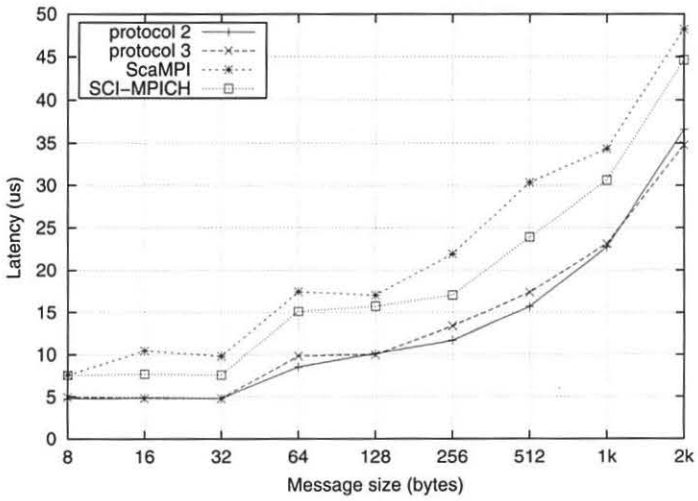

Fig. 7. Latency obtained by our protocols, SCI-MPICH and ScaMPI.

A. *Comparing the proposed protocols with ScaMPI and SCI-MPICH*

We have also compared, in terms of latency and bandwidth, protocols 1, 2 and 3 with the corresponding protocols used by the MPI implementations for SCI. Figures 6 and 7 present the latency for the three communication libraries.

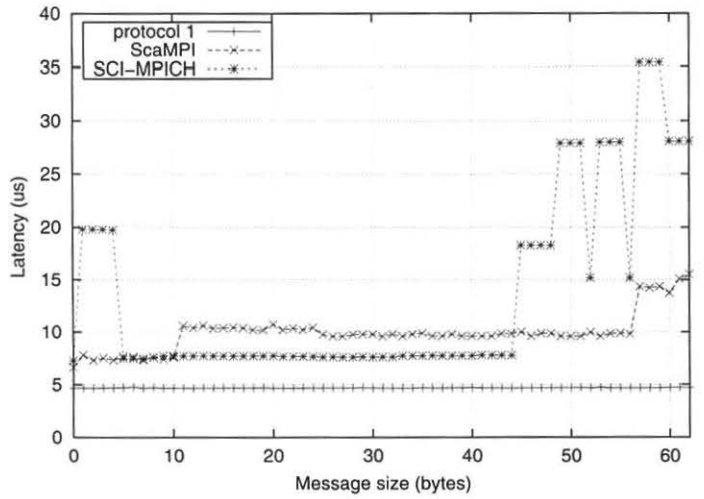

Fig. 6. Latency for short messages.

In figure 6, the latency for short messages is shown. It can be seen that the "protocol 1" is clearly more efficient than the equivalent protocols of the MPI implementations. While the minimal latency obtained by "protocol 1" is 4.66 μs, ScaMPI and SCI-MPICH got, respectively, 6.63 and 7.26 μs. Moreover, our communication kernel is the only one to keep the latency constant for messages ranging from 0 to 62 bytes. These results confirm that the devised mechanisms for "protocol 1" make a really efficient use of the low latency capabilities of the SCI technology.

Figure 7 shows the tendency of the latency curves for protocols 2 and 3 and also for the implementations of MPI, considering messages up to 2048 bytes. Both proposed protocols exhibit lower latency than that of the *eager protocol* of SCI-MPICH and ScaMPI, as can be seen.

Finally, figure 8 points out the bandwidth. One can notice that the maximum achievable bandwidth by the implementations of MPI is lower than that reached by our communication library. With zero-copy, we can get 84.12 Mbytes/s, whereas ScaMPI and SCI-MPICH obtained, respectively, 78.35 and 73.80 Mbytes/s with the *rendez-vous protocol*. "Protocol 2" is also more efficient than the equivalent *eager protocol* of the MPIs, which is adopted for messages up to 32 kbytes.

In short, we can say that all protocols designed and implemented in this work have presented better performance than the equivalent *short*, *eager* and *rendez-vous* used by ScaMPI and SCI-MPICH, according to the ping-pong measures we have done.

V. Concluding Remarks

As already commented, all protocols designed in this work are being currently used by our communication library, named DECK/SCI. From the performance evaluation we have done, it can be considered that DECK/SCI may represent an interesting alternative for programming SCI clusters. The comparison with existing MPI implementations, which are the most efficient communication libraries for SCI developed so far, has revealed that our programming environment is able to achieve better latency and bandwidth than the men-

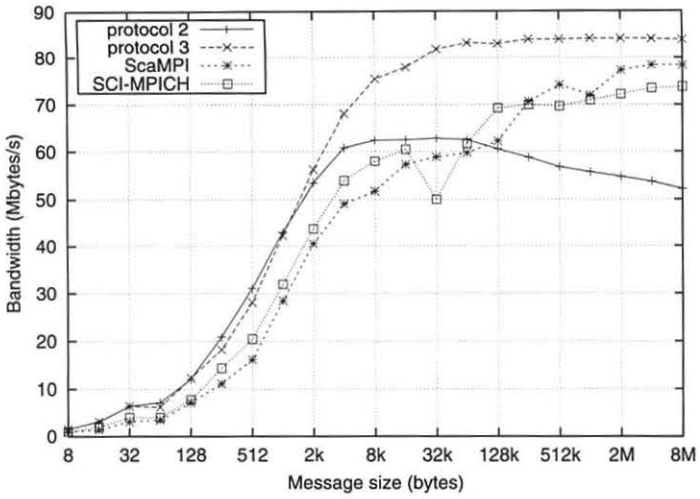

Fig. 8. Bandwidth obtained by the proposed protocols, SCI-MPICH and ScaMPI.

tioned implementations.

The gains obtained by "protocol 1", in terms of minimal latency, are really expressive. "Protocol 1" can reduce the minimal latency of ScaMPI and SCI-MPICH by 29.71 % and 35.77 % respectively. Also, the proposed zero-copy protocol achieves a peak bandwidth higher than that of the *rendez-vous* mechanism employed by the MPI implementations, obtaining 95.9 % of the maximum bandwidth supported by the SCI network, which is 87.71 Mbytes/s in our cluster. ScaMPI and SCI-MPICH have shown only 89.3 and 84.1 % respectively.

REFERENCES

[Bar00] BARRETO, M. et al. Implementation of the DECK environment with BIP. In: MYRINET USER GROUP CONFERENCE, 1., 2000, Lyon, France. **Proceedings...** Lyon: INRIA Rocquencourt, 2000. p.82–88.

[Bod95] BODEN, N. et al. Myrinet: a gigabit-per-second local-area network. **IEEE Micro**, Los Alamitos, v.15, n.1, p.29–36, Feb. 1995.

[Buy99] BUYYA, Rajkumar (Ed.). **High performance cluster computing**: architectures and systems. Upper Saddle River: Prentice Hall PTR, 1999. 849p.

[Fis97] FISCHER, M.; SIMON, J. Embedding SCI into PVM. In: EUROPEAN PVM/MPI USERS GROUP MEETING, 4., 1997, Cracow. **Proceedings...** Berlin: Springer-Verlag, 1997. p.177–184. (Lecture Notes in Computer Science, v.1332).

[Fis99] FISCHER, Markus; REINEFELD, Alexander. PVM for SCI clusters. In: HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI**: Scalable Coherent Interface: architecture and software for high-performance compute clusters. Berlin: Springer, 1999. p.239–248. (Lecture Notes in Computer Science, v.1734).

[Gei94] GEIST, Al et al. **PVM**: parallel virtual machine. Cambridge, USA: MIT Press, 1994.

[Gia98] GIACOMINI, F. et al. **Low-level SCI software requirements, analysis and predesign**. [S.l.]: ESPRIT Project 23174 — Software Infrastructure for SCI (SISCI), 1998.

[Gil96] GILLET, R. Memory Channel Network for PCI. **IEEE Micro**, v.16, n.1, p.12–18, Feb. 1996.

[Hel99] HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI**: Scalable Coherent Interface: architecture and software for high-performance compute clusters. Berlin: Springer, 1999. 490p. (Lecture Notes in Computer Science, v.1734).

[Her98] HERLAND, B. G.; EBERL, M.; HELLWAGNER, H. A common messaging layer for MPI and PVM over SCI. In: HIGH PERFORMANCE COMPUTING AND NETWORKING, 1998, Amsterdam. **Proceedings...** Berlin: Springer-Verlag, 1998. p.576–587. (Lecture Notes in Computer Science, v.1401).

[Hus99] HUSE, L. P. et al. ScaMPI—design and implementation. In: HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI**: Scalable Coherent Interface: architecture and software for high-performance compute clusters. Berlin: Springer, 1999. p.249–261. (Lecture Notes in Computer Science, v.1734).

[IEE92] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **IEEE standard for scalable coherent interface (SCI)**. New York: [s.n.], 1992. IEEE 1596-1992.

[MPI94] MPI FORUM. **The MPI message passing interface standard**. Knoxville: University of Tennessee, 1994.

[Oli01] OLIVEIRA, Fábio Abreu Dias de. **Uma biblioteca para programação paralela por troca de mensagens de clusters baseados na tecnologia SCI**. Porto Alegre: PPGC da UFRGS, 2001. Master's Thesis. (In Portuguese).

[Oma97] OMANG, Knut. Synchronization support in I/O adapter based SCI. In: INTERNATIONAL WORKSHOP ON COMMUNICATION AND ARCHITECTURAL SUPPORT FOR NETWORK-BASED PARALLEL COMPUTING, 1., 1997, San Antonio, Texas. **Proceedings...** Berlin: Springer-Verlag, 1997. p.158–172. (Lecture Notes in Computer Science, v.1199).

[Rya96] RYAN, Stein Jorgen; GJESSING, Stein; LIAAEN, Marius. Cluster communication using a PCI to SCI interface. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS, 8., 1996, Chicago. **Proceedings...** [S.l.: s.n.], 1996.

[Wor99] WORRINGEN, Joachim; BEMMERL, Thomas. MPICH for SCI-connected clusters. In: SCI-EUROPE, 1999, Toulouse, France. **Proceedings...** [S.l.: s.n.], 1999. p.3–11.

[Wor00] WORRINGEN, Joachim. SCI-MPICH: the second generation. In: SCI EUROPE, 3., 2000, Munich, Germany. **Proceedings...** [S.l.: s.n.], 2000. p.10–20. Organizado como *conference stream* do Euro-Par'2000.

[Zor99] ZORAJA, Ivan; HELLWAGNER, Herrmann; SUNDERAM, Vaidy. SCIPVM: parallel distributed computing on SCI workstation clusters. **Concurrency: Practice and Experience**, v.11, n.13, p.121–138, Mar. 1999.