DECK-SCI: High-Performance Communication and Multithreading for SCI Clusters

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

César A. F. De Rose
Catholic University of Rio Grande do Sul
High Performance Research Center
Porto Alegre, RS, Brazil
derose@cpad.pucrs.br

Abstract

This paper presents the design and implementation of DECK-SCI, a multithreaded communication library that fully exploits the high-performance capabilities of the SCI technology. We compare DECK-SCI, in terms of performance, to a commercially distributed MPI implementation and to a freely available MPICH distribution, both specifically designed for SCI clusters.

Keywords: cluster computing, message passing, multithreading, high-performance networks, SCI.

1 Introduction

The growing interest in cluster computing in the past decade has motivated many advances in both hardware and software related to computer networking, noticeably in the field of communication technologies. In order to enable clusters to face existing supercomputers in terms of performance, high-speed communication technologies such as Myrinet [3] and Gigabit Ethernet [12] have been developed. To our understanding, Myrinet is currently the most widely used of such technologies, due to factors like low latency, high bandwidth, scalability and accessible price. In addition, the nature of Myrinet stimulates the use of message passing, which is a well known and accepted programming paradigm for parallel programming.

One communication technology comparable to Myrinet in terms of performance, scalability and cost is SCI. Though established as an IEEE standard since 1992, only recently has SCI gained attention from the cluster computing community with the availability of PCI-based interfaces for PCs, namely the Dolphin [5] and Scali [19] interfaces. The main difference between Myrinet and SCI lies on the nature of communication: instead of message passing, SCI provides hardware-controlled distributed shared memory, thus allowing a node to directly access a portion of memory on another node.

Several projects are devoting research to the achievement of programming tools for SCI, both based on shared memory and message passing. Following the same tendency, we present in this paper the implementation of DECK for SCI clusters. DECK is a parallel programming environment developed at UFRGS which supports multithreading and communication. Currently, DECK has been implemented for both Ethernet and Myrinet. Our intention in porting DECK to SCI, besides the primary motivation of turning it into a solution for the programming of SCI clusters, is to provide a combined programming model with both fine- and coarse-grain parallelism, which should be suitable to minimize the NUMA characteristic of SCI clusters, as shall be commented later.

The rest of this paper is structured as follows: Section 2 presents an overview of DECK, and the details of its implementation for SCI are shown in Section 3; in Section 4 we present a performance evaluation of DECK-SCI with both raw and application-measured results; Section 5 presents some information on related work, and finally Section 6 brings the authors' conclusions and final remarks.

2 Overview of DECK

DECK (*Distributed Execution and Communication Kernel*) [2] is a parallel programming environment whose objective is to provide resources to allow the development of irregular applications, through the combination of multithreading and communication.

Figure 1 shows the general structure of DECK, which is organized in two levels: the lower layer, called μDECK, which corresponds to the platform-dependent part (operating system and hardware) and is responsible for the support

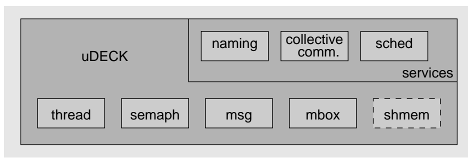

Figure 1. Internal structure of DECK.

of multithreading, synchronization and basic communication mechanisms; and the upper layer, referred to as a *service layer*, which provides additional services (e.g. naming, collective communication) and is platform-independent, in the sense that such services rely only on μDECK primitives.

In the lower layer, DECK defines its four basic abstractions: threads, semaphores, mail boxes and messages. Threads and semaphores are used for multiprogramming and synchronization, and follow a conventional Pthread-like semantics. Mail boxes and messages are intended for inter-node communication. Messages can be *posted in* and *retrieved from* mail boxes (equivalent to ordinary *send* and *receive*); these can be given names in order to be fetched by remote nodes and thus initiate communication.

3 Design and implementation of DECK-SCI

The main goal of DECK-SCI is to provide an environment for the development and efficient execution of CPU-demanding parallel applications over SCI clusters, ensuring a useful exploitation of the underling architecture. Particularly, DECK-SCI was designed to guarantee communication performance near to SCI hardware limits, accomplishing very low latency for short messages and high bandwidth for large ones.

3.1 Shared segment management

As the SCI network is based on shared memory, the first decision for the design of DECK-SCI concerns the way shared memory segments are handled in order to allow the implementation of message-passing primitives. There are a number of available APIs that make possible the establishment of SCI shared segments among the nodes of a cluster, namely the SCI driver [18], SISCI API [8], SMI [6] and YASMIN [21]. The first two are low-level APIs, whereas SMI and YASMIN are more complex libraries following the SPMD model, with a bunch of additional services developed for shared-memory programming of SCI clusters.

Since DECK-SCI needs total control over its own runtime environment, there are actually only two valid options for shared segments management: the SCI driver and SISCI API. We chose to use SISCI — Software Infrastructure for SCI —, a specification of standard primitives for SCI programming, proposed by a group of partners from both the academia and industry. It presents a set of low-level primitives and, at the same time, it is more comfortable than the SCI driver. In fact, SISCI encapsulates the driver functions and additionally supports direct access to the hardware. DECK-SCI makes use of the SISCI implementation available in the SSP (Scali Software Platform), a software package distributed with Scali Wulfkits [19], upon which our cluster is constructed.

Within DECK-SCI, SISCI is used whenever there is the need to: initialize the SCI network; create a shared segment; make a previously created segment available to all nodes of the cluster; establish a connection to an already available remote segment; map into the logical address space of a DECK process a local segment, or a remote one to which a connection has been established; flush the stream buffers of an SCI network interface.

3.2 SCI global address space

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. For this, 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 adapters 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.

3.3 The proposed communication protocols

As SCI-MPICH [23, 22] and ScaMPI [11], we have designed and implemented three different protocols for DECK-SCI: 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 communication technique developed in order to in-

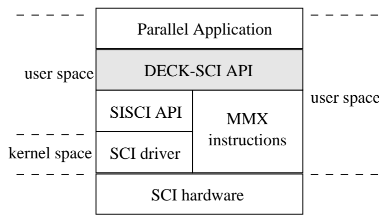

Figure 2. Execution of parallel applications with DECK-SCI.

crease the maximum achievable bandwidth for large messages.

Despite the specialization and peculiarities of each protocol, all of them, in order to obtain the best performance, were implemented taking into account some idiosyncrasies of the PCI-SCI network interfaces, namely the performance difference between remote loads and remote stores and the exploitation of the network interfaces' stream buffers. Although the support for direct memory access allows one to conceive simple message-passing communication schemes, the implementation of efficient communication protocols on top of SCI requires more than trivial loads and stores into shared segments.

DECK-SCI 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 protocols of DECK-SCI share a couple of characteristics: 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, in order to optimize the use of the stream buffers. Figure 2 illustrates the layers involved in running parallel applications with DECK-SCI.

Depending on the message length, DECK-SCI transparently chooses the appropriate protocol for carrying out communication. In fact, programmers do not even need to know that there are multiple message-passing protocols.

3.4 "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. 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 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 the 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 [14].

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 by DECK-SCI to 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 get notified the message certainly was completely received.

For the working of "protocol 1", every mail box created during the execution of a parallel application reserves, within its shared segment, a separate ring buffer to each DECK 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 and 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.

3.5 "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 DECK-SCI 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 in DECK-SCI, 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, DECK-SCI 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, named control segment. Each DECK-SCI process, during initialization, creates its own control segment and maps into its logical address space the control segments of all processes.

3.6 "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, since DECK-SCI is targeted at high-performance communication.

The most efficient communication libraries for SCI clusters developed so far, SCI-MPICH and ScaMPI, have two different protocols equivalent to DECK-SCI protocols 1 and 2. Further, both MPI implementations adopt the same solution to the relative poor performance of their *eager protocol* — corresponding to the "protocol 2" of DECK-SCI. 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 API, the message abstraction exists explicitly, being represented by a message object. As the programmer is required to utilize specific DECK 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, DECK-SCI "protocol 3" implements 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 transition 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 com-

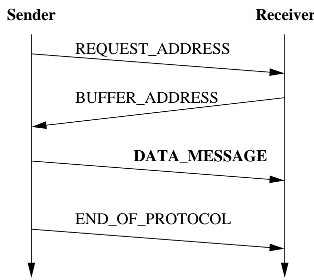

Figure 3. DECK-SCI zero-copy protocol.

munication 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, 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.

During DECK-SCI initialization, each process creates and exports a shared segment for storing messages to be received through "protocol 3". The message buffers are allocated on this segment instead of local memory depending on the message size.

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.

4 Performance evaluation

All results presented in this section were obtained in 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) adapters, 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.

The results concerning latency and bandwidth were measured by means of a traditional ping-pong algorithm. For each message size, we have made 1000 repetitions.

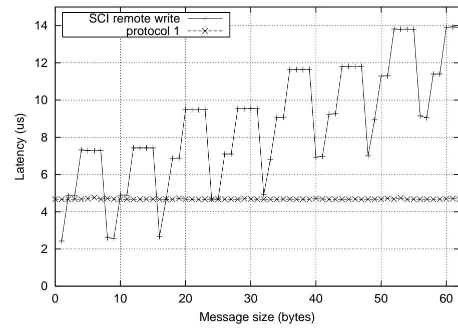

Figure 4. Latency of "protocol 1".

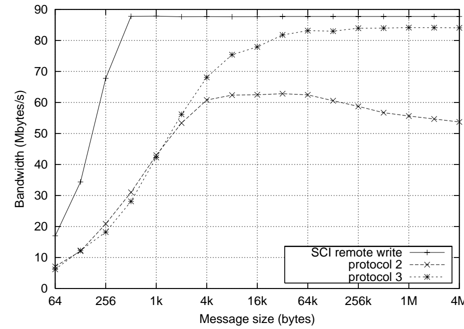

Figure 5. Bandwidth of protocols 2 and 3.

4.1 Evaluation of DECK-SCI communication protocols

In the following, we evaluate the efficiency of the three DECK-SCI communication protocols, comparing their performance to that of raw communication over SCI. These results allow us to verify the impact of the proposed mechanisms — flow control, message ordering, message signalling —, pointing out the overhead inherent to the protocols.

Figure 4 shows the latency of "protocol 1". Observe that DECK-SCI is able to keep the latency below 5 μs for messages ranging from 0 to 62 bytes. In contrast, the latency obtained by raw communication over SCI is not constant and it is higher than that of "protocol 1", except for 1, 8 and 16 bytes. Although, at a first glance, it seems strange the better performance of "protocol 1", even when compared to raw communication, we expected these results, since "protocol 1" always sends 64 bytes, in spite of the actual message length, in order to optimize the use of stream buffers. Remember that stream buffers can optimize the communication for an amount of data multiple of 64.

These results reveal that DECK-SCI really accomplishes very low latency for short messages, namely 4.66 μs. The

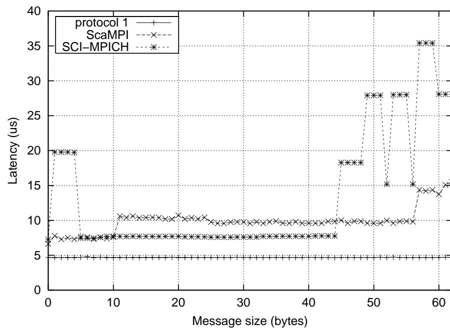

Figure 6. Latency for short messages.

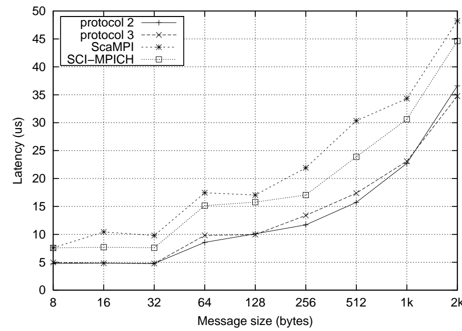

Figure 7. Latency obtained by DECK-SCI, SCI-MPICH and ScaMPI.

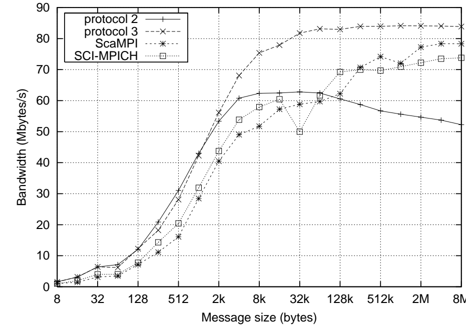

Figure 8. Bandwidth obtained by DECK-SCI, SCI-MPICH and ScaMPI.

optmized use of the stream buffers compensates the extra overhead caused by flow control.

Figure 5 shows the achievable bandwidth for protocols 2 and 3, as well as the bandwidth of raw communication over SCI. As can be seen, thanks to the zero-copy mechanism, "protocol 3" reaches a maximum bandwidth close to the that of raw remote write, exhibiting 84.12 Mbytes/s, which represents 95.9% of the maximum bandwidth that can be reached in this architecture — 87.72 Mbytes/s.

"Protocol 2" is unable to increase the bandwidth beyond 62.81 Mbytes/s, due to the extra copy of message, as already explained. Notice that for messages up to 1024 bytes, the performance of "protocol 2" is a little bit better than that of "protocol 3". From this point on, however, the extra copy avoidance compensates the handshaking between sender and receiver.

4.2 DECK-SCI, SCI-MPICH and ScaMPI

We have compared, in terms of latency and bandwidth, DECK-SCI with the existing implementations of MPI for SCI clusters. Figures 6 and 7 present the latency for the three communication libraries.

In figure 6, the latency for short messages is shown. It can be seen that the protocol for short messages of DECK-SCI is clearly more efficient than the equivalent protocols of the MPI implementations. While the minimal latency obtained by DECK-SCI is 4.66 μs, ScaMPI and SCI-MPICH got, respectively, 6.63 and 7.26 μs. Moreover, DECK-SCI is the only library 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 of DECK-SCI and for the implementations of MPI, considering messages up tp 2048 bytes. Both DECK-SCI 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 for the three communication libraries. One can notice that the maximum achievable bandwidth by the implementations of MPI is lower than that reached by DECK-SCI. With zero-copy, DECK-SCI got 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 DECK-SCI 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.

4.3 Results obtained with applications

Besides the raw performance evaluation of DECK-SCI, we have run and measured the performance of three common applications in the HPC community: Mandelbrot fractal generation [13], Laplace's Equation [16] and POV-Ray [15]. The Mandelbrot and Laplace applications have been implemented by our group; POV-Ray has been ported

from the MPI implementation available in Scali's SSP. All of them have been implemented at the High Performance Research Center (CPAD), PUCRS/HP. In all cases, DECK-SCI was configured to use the zero-copy protocol for messages greater than 8 kbytes.

The first application calculates the full Mandelbrot set and draws a two-dimensional picture of it. It iterates the Mandelbrot function for every pixel on the picture, and sets the color of the pixel according to the iteration where the orbit "escapes", with a maximum iteration value of 17500. Our parallel version uses a master/slave model where the master is responsible for distributing parts of the picture to be calculated and for displaying the already calculated parts. The picture is partitioned in horizontal slices and slaves keep requesting new slices to calculate until all slices are ready. Slaves communicate only with the master in a 1:n pattern.

The Laplace's Equation application calculates the temperature in a slab of a hypothetical homogeneous material completely insulated on the edges. Initially, the slab is thoroughly at one uniform temperature and a heat source is applied to one of the borders. Laplace's Equation is used to solve this problem, being applied by means of the Gauss-Seidel iterative method, with maximum error of 10^{-3}. The surface of the slab is divided in square sections, where each intersection is a point in a grid. The finer the grid, the more accurate the approximation, and the larger the problem. Our parallel version of the application partitions the image in rectangular regions, assigning each region to a node.

POV-Ray (Persistence Of Vision Raytracer) is a three-dimensional rendering engine. The program derives information from a file containing the description of a scene (objects, textures, lights and point of view) simulating the way the light interacts with the objects in the scene to obtain a three-dimensional realistic image (procedure known as ray tracing). Each pixel of the resulting image may be calculated from the scene description without knowledge of neighbor points. MPIPovray 3.01 is a parallel implementation of the above application that divides the image to be calculated in horizontal slices, mapping each slice to one slave process. A master process is dedicated to the image partitioning, slices distribution and screen drawing. Slave process don't wait for the calculation of the whole image slice and send ready screen lines to the master to allow real time drawing of the already calculated points.

Figure 9 shows the execution times obtained with both DECK-SCI and ScaMPI for the described applications. All the results correspond to using the 8 processors available in our cluster; DECK-SCI makes use of two threads for calculations, and with MPI we have run two processes on each node[1]. In any case, the first node always holds an additional thread/process to act as a manager for the application (e.g. to deliver and collect calculated sections in the Mandelbrot algorithm).

[1]For this reason, we could not include MPICH-SCI in the comparison, since the version installed in our cluster does not support SMP.

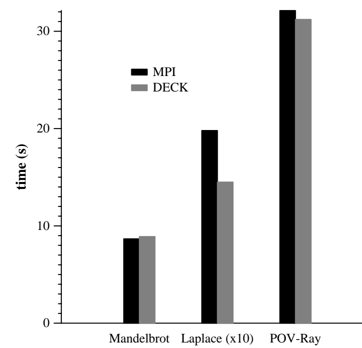

Figure 9. Results obtained for the three applications.

The graphs present very good results for DECK-SCI, being able to achieve practically the same execution times as MPI for Mandelbrot (8.9s DECK, 8.67s MPI) and POV-Ray (31.22s DECK, 32.13s MPI), and a significant 27% reduction for Laplace's Equation (145s DECK, 198s MPI). We believe this gain is due to the size of messages in Laplace (larger than 8K), entering the region of the zero-copy protocol. Anyway, in general, we consider that DECK-SCI is able to achieve at least the same performance of currently available MPI implementations for SCI.

5 Related work

In order to enable application programmers to exploit the benefit of SCI's shared address space model without detailed hardware knowledge, several standardized APIs like MPI and PVM are also available for SCI systems. We have shown two implementations of MPI, namely ScaMPI and MPICH-SCI, and there are two others for PVM: SCIPVM [24] and PVM-SCI [7]. Another effort targeted at message passing for SCI is CML [9], a low-level API designed to efficiently support implementations of MPI and PVM for SCI.

Parallel languages like Split-C have also been ported to SCI systems. Split-C [4] is a parallel extension of the C programming language that provides bulk transfer operations as well as a global address-space abstraction, the later being mapped directly over SCI's shared address space.

Threads are in widespread used as a model for concurrent programming and are able to efficiently exploit the multiple processors of a SMP node. Because threads communicate over shared memory, the multithreaded model should be efficiently implemented over a SCI system and would be a very interesting alternative for porting sequen-

tial programs to SCI cluster machines. SISCI-Pthreads [20] follows this direction emulating a Pthread-compliant SMP machine over the SCI system. Sthreads [17] also provides threads across the nodes of a SCI cluster but the placement mechanism is not transparent.

6 Concluding remarks

In this paper we have presented the design and implementation of DECK for SCI clusters; from the performance evaluation presented in Section 4, it can be considered that DECK may represent an interesting alternative for the programming of SCI clusters. The comparison with existing implementations of MPI for SCI has revealed that DECK-SCI is able to achieve significantly better latency and bandwidth, in terms of raw performance, than the mentioned implementations. In a different situation, with the implementation of three scientific applications, DECK-SCI presented performance at least equivalent to that of MPI, reaching up to 27% difference in the case of Laplace. From these results we consider that our main goal has been achieved, namely to provide an efficient implementation of DECK for SCI.

One can notice a concrete effort to supply a broad range of programming tools for the SCI platform, but to our understanding there is still a gap concerning an API with support for a hybrid programming model. Being a SCI system a NUMA machine, with different access times for local and remote memory, we believe that a API with a combined support for message passing and multithreading gives the programmer the needed functionality to exploit all the potential of a SCI based cluster, under different application demands. Some APIs like ScaMPI and MPICH-SCI are thread-safe, but not thread-aware. That means that processes may use threads locally but the application interface is not aware of such functionality. In DECK-SCI we have tried to achieve a programming environment where multithreading can be fully integrated with communication both at the system and the application level.

References

[1] *Proc. of HPCN'98*, volume 1401 of *Lecture Notes in Computer Science*, Amsterdam, 1998.

[2] M. E. Barreto. DECK: Um ambiente para programação paralela em agregados de multiprocessadores. Master's thesis, PPGC da UFRGS, Porto Alegre, 2000.

[3] N. Boden et al. Myrinet: A gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[4] D. Culler et al. Parallel programming with Split-C. In *Proc. of SuperComputing'93*, Portland, Oregon, Nov. 1993.

[5] Dolphin interconnect solutions web. Available at http://www.dolphinics.no, Apr. 2000.

[6] M. Dormanns, W. Sprangers, H. Ertl, and T. Bemmerl. A programming interface for NUMA shared-memory clusters. In *Proc. of HPCN'97*, volume 1225 of *Lecture Notes in Computer Science*, pages 698–707, Vienna, Austria, Apr. 1997. Springer.

[7] M. Fischer and J. Simon. Embedding SCI into PVM. In *Proc. of the 4th European PVM/MPI Users Group Meeting*, volume 1332 of *Lecture Notes in Computer Science*, pages 177–184, Cracow, 1997.

[8] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B. D. Johnsen, H. Kohmann, R. Nordstrøm, and P. Werner. Low-level SCI software requirements, analysis and predesign. Technical report, ESPRIT Project 23174 — Software Infrastructure for SCI (SISCI), May 1998.

[9] B. G. Herland, M. Eberl, and H. Hellwagner. A common messaging layer for MPI and PVM over SCI. In *Proc. of HPCN'98* [1], pages 576–587.

[10] G. Horn and W. Karl, editors. *Conference Proceedings of SCI-Europe '99*, Toulouse, France, Sept. 1999.

[11] L. P. Huse, K. Omang, H. Bugge, H. Ry, A. T. Haugsdal, and E. Rustad. ScaMPI—design and implementation. In H. Hellwagner and A. Reinefeld, editors, *SCI: Scalable Coherent Interface: Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *Lecture Notes in Computer Science*, pages 249–261. Springer, Berlin, 1999.

[12] IEEE. Gigabit ethernet. IEEE P802.3z, 1997.

[13] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. E. Freeman and Company, New York, 1982.

[14] K. Omang. Synchronization support in I/O adapter based SCI. In *Proc. of CANPC'97*, volume 1199 of *Lecture Notes in Computer Science*, pages 158–172, San Antonio, Texas, 1997.

[15] Persistence of vision(tm) ray tracer. Available at http://www.povray.org.

[16] W. H. Press et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University, Melbourne, second edition, 1994.

[17] E. Rehling. Multithreading for SCI-clusters: Yasmin and the Sthreads library. In Horn and Karl [10].

[18] S. J. Ryan. The design and implementation of a portable driver for shared memory cluster adapters. Research report 255, Department of Informatics, University of Oslo, Dec. 1997.

[19] Scali homepage—scalable Linux systems—affordable supercomputing. Available at http://www.scali.com, Apr. 2000.

[20] M. Schulz. SISCI-Pthreads: SMP-like programming on an SCI cluster. In *Proc. of HPCN'98* [1], pages 566–575.

[21] H. Taşkın. Synchronisationsoperationen für gemeinsamen speicher in SCI-clustern. Diplomarbeit, Universität GH Paderborn, Paderborn, 1998.

[22] J. Worringen. SCI-MPICH: The second generation. In *Proc. of SCI Europe 2000*, pages 10–20, Munich, Germany, 2000. Held as a conference stream of Euro-Par'2000.

[23] J. Worringen and T. Bemmerl. MPICH for SCI-connected clusters. In Horn and Karl [10], pages 3–11.

[24] I. Zoraja, H. Hellwagner, and V. Sunderam. SCIPVM: Parallel distributed computing on SCI workstation clusters. *Concurrency: Practice and Experience*, 11(13):121–138, Mar. 1999.

View publication stats