SISCI — Implementing a Standard Software Infrastructure on an SCI Cluster*

Michael Eberl, Hermann Hellwagner, Martin Schulz
Institut für Informatik, Technische Universität München
{eberl,hellwagn,schulzm}@informatik.tu-muenchen.de

http://wwwbode.informatik.tu-muenchen.de/Par/arch/smile/sisci/

Bjarne G. Herland
Parallab, University of Bergen
Bjarne.Herland@ii.uib.no

http://www.parallab.uib.no/projects/sisci/

Abstract

To enable the efficient utilization of clusters of workstations it is crucial to develop a stable and rich software infrastructure. The ESPRIT Project SISCI will provide two widely used message–passing interfaces, MPI and PVM, as well as a POSIX compliant, distributed thread package (Pthreads) on multiple SCI–based clusters. This paper features motivation and background on this projects as well as details of the two core components: the common messaging layer and the Pthreads package.

1 Introduction

As PC clusters with high–speed interconnection networks are emerging as a cost–effective alternative to large scale parallel systems, the question of their programmability has increasingly become important to resolve. The individual machines must be tied together and represented to the user as a single machine. In contrast to the traditional massively parallel machines, this proves more difficult to accomplish in a cluster environment, as the machines are not as tightly coupled and the operating systems are not geared towards clustering. The SISCI Project [26] (Standard Software Infrastructure for SCI-based Parallel Systems), funded by the European Commission as an ESPRIT HPCN project, attacks this problem focusing on one promising interconnection approach, the Scalable Coherent Interface (SCI) [22].

This interconnection solution is one of various new local–area or system–area networks that have made cluster-based computing attractive. However, it clearly differs from most other networks, like Myrinet [2] or Fast Ethernet, as it is not based on explicit communication via messages. Data is transfered using implicit communication via remote memory accesses. Every node can map remote memory segments from any other node within the cluster into its own address space and operate on it like on local memory. The actual communication is triggered by the SCI adapter card on reads from and writes to remote memory segments. This technique not only allows high throughput rates, but also low latencies of under 3 microseconds (one-way) as communication can be performed without any protocol overhead or system calls.

*This work is supported by the European Commission in the Fourth Framework Programme under ESPRIT HPCN Project EP23174 (SISCI).

Within the SISCI project, several well-known parallel programming models will be implemented on this new architectural concept, exploiting the possibilities offered by SCI. They will provide a familiar and transparent access to the cluster and will allow for easy ports of already parallelized code to the new environment. In order to open the clusters to a broad range of applications, message passing programming models in the form of the widely used libraries MPI [24] and PVM [25], as well as a shared memory programming model represented by a POSIX 1003.1c [23] compliant thread package will be provided.

These programming models are challenging to implement in a cluster environment. Especially for the POSIX thread package, much work must be done to abstract the physically distributed cluster memory into one global address space maintaining total transparency. This problem is further increased by the fact that the package is supposed to be implemented as a pure library approach without any special compiler support or the need to change the original application source code.

Offering these two programming paradigms, message passing and shared memory, efficiently on one single system opens this system to a whole new group of users and applications. The programmer will decide which model to use according to the needs of the application; it will no longer depend on the architecture of the underlying system. This forms a stable basis for any future software development and will promote SCI–based parallel systems as an attractive platform for low–cost parallel computing.

The remainder of this paper is organized as follows. An overview of the Scalable Coherent Interface (SCI) as the enabling technology for this project is given in Section 2. Section 3 motivates the need for a standardized software infrastructure and gives further background information on the project. Sections 4 and 5 describe in more detail two of the key components of the SISCI project, the common messaging layer for MPI and PVM and the Pthreads package. Section 6 offers a brief overview of related work, followed by some concluding remarks in Section 7.

2 The Scalable Coherent Interface (SCI)

SCI is a recent interconnect standard [22] that specifies hardware and protocols to tightly connect up to 64k nodes (e.g. workstations or PCs, multiprocessors, bus bridges, network interfaces, I/O adapters) into a high–speed network [9, 10]. SCI defines computer–bus–like services, but, in contrast to busses, offers fully distributed solutions for their realization. The most notable of these services are a single physical 64–bit address space across SCI nodes and related transactions for reading, writing, and locking memory locations in this hardware–based distributed shared memory (DSM). Transactions for efficient messaging as well as for fast event notification or synchronization are also specified.

Optional distributed cache coherence protocols for the SCI DSM have been developed so that SCI systems with NUMA as well as CC–NUMA characteristics can be built. Examples for the former are local–area compute clusters for parallel processing built from standard workstation or PC nodes, as used in SISCI. The latter systems are represented by the HP/Convex Exemplar, Data General NUMALiine, and Sequent NUMA–Q multiprocessors.

SCI avoids the physical limitations of computer busses by employing unidirectional point–to–point links only. Thus, in contrast to bus–based symmetric multiprocessors, the size and number of processors of SCI systems are not limited by bus length and bus speed. The links can be made fast and their performance can scale with improvements in the underlying technology. SCI initially specifies link bandwidths of 1 Gbit/s using serial links and 1 Gbyte/s using parallel links over small distances.

At the logical layer, SCI defines packet–based, split–transaction protocols. SCI specifies up to 64 outstanding transactions per node, allowing for each node to pipeline multiple packets into the network. The interconnect bandwidth can thus be utilized efficiently, and the latencies of e.g. DSM accesses can be hidden effectively for the node.

The basic building blocks of SCI networks are small rings (ringlets). Large systems are built of rings of rings, rings interconnected via SCI switches, or multistage interconnection networks that can be constructed from ringlets. Further, a standardized interface of a node to the SCI network is proposed, with an incoming and an outgoing SCI link per node. The protocols defined for this basic model provide for fair bandwidth allocation and network administration; they are deadlock–free and robust. Maintenance of data integrity and error detection is primarily done in hardware, for instance by hardware checksum computation and by timeout logic that supervises outstanding transactions and performs retries in case of errors.

The major benefit of SCI networks and protocols is that communication can be performed *at user level* via the hardware DSM, by load and store operations into memory regions mapped from remote memories. This translates into latencies in the low microseconds range in a cluster environment; see e.g. [18, 13] and [20, 21]. For high–volume communications, the SCI messaging transactions can be employed which use DMA and SCI hardware protocols for data transfers. Throughput close to the bandwidth provided by the network interface can be achieved in this manner, e.g. [18, 13].

The SCI protocols do not guarantee in–order delivery of transactions. If this is an issue for software atop of SCI hardware, implementation–specific facilities must be provided and used to ensure correct implementation of message passing or shared–memory communication.

The SISCI project uses current SCI hardware from Dolphin Interconnect Solutions: the SBus/SCI adapter card [4] and PCI/SCI adapter board [5]. These boards plug into the I/O bus (SBus or PCI) of off–the–shelf workstations or PCs and allow them to be connected into SCI compute clusters. The adapters and associated driver software offer two interfaces and communication facilities over the SCI DSM: shared–memory operations and DMA–based messaging.

Each node can create shared memory segments in its I/O address space and export them into the SCI network. Other nodes import these DSM segments into their I/O space. A process may further map DSM segments into its virtual address space and, from that point on, use standard load and store instructions to access shared, potentially remote memory. The SCI adapter cards translate I/O bus transactions into SCI transactions (and vice versa) and, using on–board address translation tables, local I/O addresses into SCI addresses (and vice versa). Further, they participate in SCI protocol processing. None of the cards currently supports (remote) lock transactions.

The functionality, the two–stage address translation scheme (virtual address — I/O address — SCI address), and the performance of the SBus/SCI interface boards are described e.g. in [13]. Throughputs of up to 25 Mbytes/s have been achieved with this card. Further information on the PCI/SCI interface cards is given in [20, 21]. The PCI/SCI card introduces memory barrier operations and the concept of *streams*. A memory barrier on a node ensures that all DSM operations issued by that node have finished. A stream allows to combine multiple consecutive SCI accesses to consecutive addresses to be combined into larger packets (up to 64 bytes). PCI/SCI interfaces employing multiple streams have been measured to achieve throughput of up to 70 Mbytes/s.

3 SISCI Background

As discussed above, SCI is a promising interconnect technology for building high–performance, cost–effective parallel computing platforms out of "commodity off–the–shelf" workstation or PC nodes. While hardware (SCI adapter cards and switches) and basic device driver software for SCI compute clusters have been on the market for about three years, standard parallel programming APIs like MPI, PVM, or a thread library, have not been available. These APIs would support a wide range of parallel applications and enable application porting to SCI clusters with relative ease.

The ESPRIT HPCN project SISCI aims at establishing this missing link. The project develops appropriate software support for parallel programming on SCI–based clusters, by implementing and evaluating the following formal and *de facto* standard software environments:

- the Message Passing Interface communication library (MPI);
- the Parallel Virtual Machine parallel programming system (PVM);
- a POSIX compliant, distributed thread package (Pthreads) that will transparently operate across machine boundaries and provide a system–wide virtual address space.

The project pursues an approach to provide this software on multiple hardware platforms from possibly multiple vendors, and on multiple operating systems. Therefore, in the interest of portability and reusability of the programming environments as well, the project will also define and implement a standardized low–level SCI programming layer. This layer is to hide specific details of SCI hardware and device driver. It offers functionality such as setting up, exporting, and mapping DSM segments, initiating efficient bulk data transfers via the DMA facility, and initializing and monitoring the SCI network. In case a parallel application requires highest efficiency, this low–level API can also be used directly; still, portability among the different platforms of the project is being retained.

On top of these standard parallel programming APIs, a number of demanding parallel applications will be ported or developed, respectively, and evaluated. The applications are from diverse fields such as computational fluid dynamics (CFD), structural analysis, synthetic aperture radar (SAR) imaging, and large–scale data acquisition.

Figure 1 depicts the layered structure of the SISCI cluster and software products. It must be noted that MPI and PVM will be based on a common message passing layer that will make efficient use of the SCI DSM and communication facilities. The figure also shows a new kernel–level component, the SCI Virtual Memory Manager, that extends the virtual memory manager of the underlying operating system to established the global virtual address space for the SISCI Pthreads. These two basic components will be dealt with in the rest of this paper.

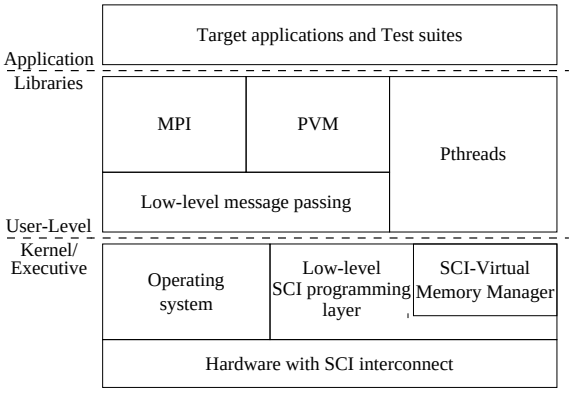

Figure 1: The architecture of the SISCI project.

Two different cluster architectures are being used in SISCI: (1) SPARC–based workstations with Dolphin SBus/SCI adapters and the Solaris operating system; (2) Pentium II–based high–end PCs with Dolphin PCI/SCI interface cards and Windows NT. Systems integration and cluster administration issues are also being addressed in the project.

4 Common Messaging Layer for MPI and PVM

The SISCI project develops the two widely used message passing interfaces, MPI and PVM for SCI clusters. MPI will be implemented on top of Unix and PVM on top of Windows NT. Despite the different operating systems, both message passing interfaces are to be built on top of SCI distributed shared memory. This enables the use of a common message passing layer based on shared memory segments and allows the efficient implementation of MPI and PVM on top of it.

Message passing interfaces on top of SCI DSM are designed to offer SCI's good communication performance, especially the low message latency, to application software. Short messages[1] are sent and received by direct accesses to shared memory segment. Additional copy operations and the overhead for setting up network DMA hardware are avoided wherever possible. However, for long messages this overhead becomes negligible and the SCI DMA hardware is used to relieve the CPU and to achieve high bandwidth.

Designing a common message passing layer (CML) for MPI and PVM requires a detailed understanding of MPI and PVM functions. Close analysis of their semantics gives slight differences in most routines while only a small set of routines are identical. Despite these differences, a basic CML can be designed to form a base for the implementation of both MPI and PVM.

4.1 Terminology

In order to explain the slight semantic differences between MPI and PVM it is necessary to introduce some terminology at this point.

A message has *arrived* (at the receiver) if it is available in the messaging layer at the receiver. The term to *match a message* is used when an arrived message is paired with a receive call. A message that has been matched can not be matched again. The term to *post a receive* describes the action where the messaging layer is informed that a message of a given type is expected to arrive. A posted receive will immediately be matched if the message had arrived already.

The term *unexpected message* is used to describe a message that has arrived, but no matching receive has been posted. A message consists of two parts: a *header* which identifies the message, and the *body* which is the contents of the message. A message is identified by an *address*, which is the triplet (sender, tag, communicator[2]).

4.2 Functionality of the CML

The main semantic differences between MPI and PVM are in receiving messages. One good example is the non–blocking receive, present in both MPI and PVM. In MPI, the non–blocking receive is called once, followed by a wait–for–completion call. When the latter returns, the message is available in the user buffer. In PVM, the non–blocking receive simply returns if the message is not available, like a non–blocking probe. The user has to call the routine repeatedly until the message is available, at which point it is matched and copied to the system receive buffer. Thus, in MPI, a receive call will always match exactly one message, in contrast to PVM, where multiple receive calls may fail until one succeeds and the message is received.

Furthermore most PVM "receive" calls (other than `pvm_precv()`) do not receive any data into user buffers, but only the PVM "unpack" functions can do so. Hence, in PVM receiving a message is alway split into two functions: the receive calls and the unpack call.

[1] The threshold delimiting "short" and "long" messages can be adjusted at configuration time and is set to 1024 bytes by default.

[2] For PVM the communicator is merely a constant value.

The conclusion is that in both MPI and PVM receive calls may or may not match a message, depending on whether the call was non–blocking and whether the message has already arrived. But the problem at what time messages are matched and data is received is solved differently in the two interfaces. The solution for the CML is to provide the fundamental functionality for MPI and PVM: A function called `common_match()` that matches a message and a second call `common_copy()` that will get the message out of the system buffer into user memory. These functions can be called from MPI and PVM as required.

Other functionality of the CML includes:

- A blocking receive function, useful for both MPI and PVM, that matches and receives a message into a user buffer.
- A non–blocking send function that can be used "as is" by MPI; together with a wait–for–completion function, it forms a blocking send function for PVM.
- A search function that seeks for arrived messages.
- A cleanup function that keeps the DSM read buffer from getting filled up and allows further messages to be received.
- Queues to store posted receives as well as unreceived messages and outstanding DMA transactions.
- A separate thread that maintains the outstanding DMA queue to support transfers of long messages.

4.3 Implementation of the CML

Basic Structure and the Receive Ring Buffer

Figure 2 shows the basic structure of the DSM message–passing routines. This structure consists of a part for sending messages (left, on node A) and a part for receiving messages (right, on node B).

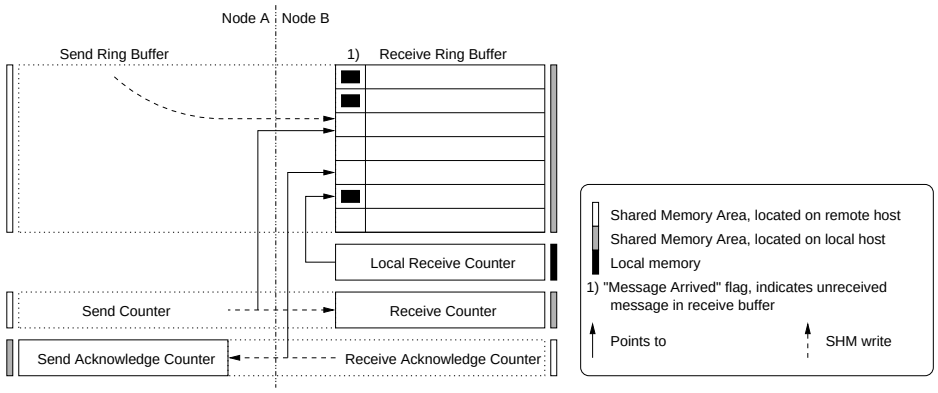

Figure 2: SCI DSM send and receive buffers.

The gray marking aside of an area means that it is local physical memory shared with another node via SCI, the white side–bars on the opposite side of the drawing mark the respective counterparts of these areas: mapped memory areas that physically reside on a remote node. This means that the gray and white marked areas always have the same contents as their counterpart on the opposite side of the drawing. All memory accesses to a

gray area are treated as regular memory accesses, while read and write operations to white areas are transformed into remote accesses to their counterpart. Variables marked black reside in regular local memory which is not shared.

The send and receive buffers are organized as ring buffers. Every ring buffer offers a number of message *slots* consisting of space for the message header and the contents of a short message. Every slot has a *Message Arrived* flag which is set by the sender and cleared when the message has been matched by the receiver[3].

Relying solely on this flag for detecting free message slots and newly arrived messages would not guarantee correct message ordering and could also lead to race conditions. Therefore two counters control the ring buffers: the *Send/Receive Counter* and the *Send/Receive Acknowledge Counter*. As depicted in figure 2, the *Send Counter* and the *Receive Counter* (as well as the acknowledge counters) are only different names for the same value.

The sender writes messages to the slot pointed to by *Send Counter* and then increases this counter. The *Send Acknowledge Counter* marks the last free slot that may be used by the sender. The receiver, when matching a message, checks all slots starting from *Local Receive Counter* until *Receive Counter* for newly arrived messages (*Message Arrived* flag set).

When the first message at *Local Receive Counter* is matched, this pointer is increased to the next unmatched message. The *Receive Acknowledge Counter* is increased as soon as the message has been received and the slot can safely be reused.

The Unexpected Message Queue

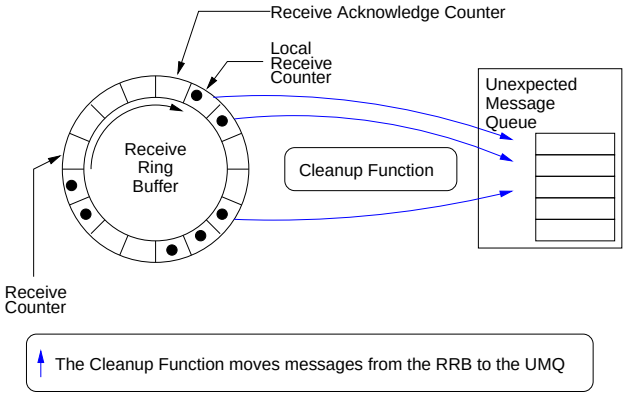

Figure 3: Receive Ring Buffer and Unexpected Message Queue.

In order to avoid deadlocks in case the *Receive Ring Buffer* (RRB) is filled up with unexpected messages while a blocking receive is waiting, the receiver can call a cleanup function that moves messages from the RRB to the *Unexpected Message Queue* (UMQ). Redundant copying of messages is prevented by cleaning the RRB as late as possible (see figure 3).

When matching a message, the UMQ has to be checked first to see whether the message already has been moved from the RRB. Next, the RRB is checked in case the message arrived after the last cleanup.

The Posted Receive Queue

`MPI_Irecv()` is a mechanism to post a receive request enabling the underlying message engine to copy the message into the user buffer as soon as it arrives. To facilitate this, the

[3]Note that a matched message still stays in the Receive Ring Buffer (RRB) and is removed from there later.

Posted Receive Queue (PRQ) is added to the CML. For every message found in the RRB, the match function first checks the PRQ. If no match is found in the PRQ, the match function tries to match the message that was intended to be received by the calling function. In this manner correct ordering of messages is ensured even when allowing posted receive requests.

Long Message Transfer

Long messages are transfered using a combination of both available message passing mechanisms: the message header is sent via DSM, while the bulk data is transfered through a DMA mechanism. The CML guarantees that the correct message order is always preserved, thus long messages and their headers will not be mixed up. The DMA mechanism is handled by separate tasks (threads) processing the queues of outstanding requests. The common send and receive functions enqueue the requests and receive a handle that can be used to check whether the transfers have been completed.

4.4 Summary

The common message passing layer for MPI and PVM provides the means for efficiently implementing MPI and PVM on top of SCI shared memory and SCI DMA transfers. Both mechanisms are used, depending on which one is more efficient. Transfers of short messages avoid unnecessary memory copy operations: every message is copied once into the network and once out of the network. Only in case of unexpected messages jamming the Receive Ring Buffer, additional memory operations have to be done in order to prevent temporal blocking of the sender.

For long message transfers we relieve the CPU from copying large amounts of data into the network. By using SCI DMA transfers, threads/tasks are not blocked and the bulk bandwidth of the SCI hardware can be exploited.

5 SISCI–Pthreads

Besides the previously mentioned message passing approach, the SISCI project will also feature a shared memory programming model. A POSIX compliant thread package [23] will be implemented on the SISCI cluster on top of Windows NT. It will provide complete transparency to the user, allowing the execution of Pthread applications without recompiling or even rewriting the application. As a pure library approach, it will be sufficient to simply link the library to the code and execute it on the target platform.

5.1 Motivation and Objectives

Unlike message passing, shared memory with multithreading has the advantage that its programming model does not differ far from the conventional sequential one. It is therefore simpler to learn, and applications are easier to write or to port. This is mainly achieved by the fact that the programmer only has to deal with one address space that spans all participating machines. Conceptually, there is no distinction between local and global data; all data can be directly accessed by any processor. Hence, there is also no need for data redistributions allowing easier and more implicit implementations of load and work distribution. However, this ease of programmability comes at a price; the complexity of the implementation of such a system is rather large as each processor needs fine grain access to all the data. Due to this, shared memory programming models can normally be found on tightly coupled systems like symmetric multiprocessors (SMPs), whereas loosely coupled machines, like clusters, are programmed using message passing.

The SISCI Pthreads project has the goal of forming a bridge between these two worlds incorporating the best of both: an easy–to–use programming model in the form of globally

shared memory on top of a cost efficient cluster environment consisting of "commodity off–the–shelf" components. The main challenge is to create a fully transparent memory that is shared across all participating machines. Each thread running within a Pthread application must be provided with the same view on the memory; the data distribution must be hidden completely.

5.2 Architectural Concept

SCI offers hardware support for a shared–memory concept allowing transparent remote memory accesses within a cluster. However, it is not powerful enough to provide the capabilities needed to implement a globally shared memory. Instead, it provides mechanisms similar to inter–process shared memory as known from workstations. To achieve the necessary intra–process shared memory, additional software mechanisms known from traditional software DSM approaches have to be applied. By combining them with the remote memory operations offered by SCI, a new concept, called the SCI Virtual Memory, can be created featuring intra–process shared memory. In contrast to traditional software DSM systems, however, data does not have to be migrated or replicated within the system, eliminating the problem of false sharing and the cost of a multiple writer consistency protocol. Remote accesses can be satisfied directly via the SCI network allowing for an efficient implementation of a globally shared memory in a cluster environment.

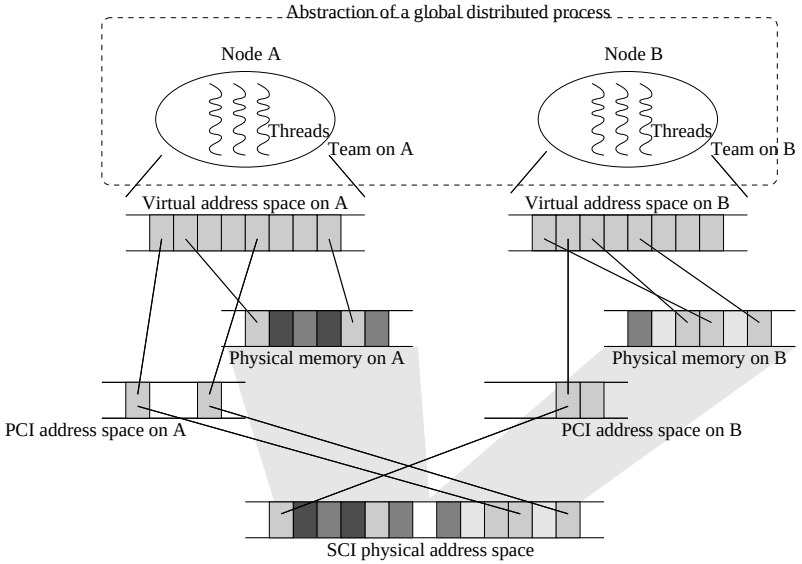

Figure 4: SCI Virtual Memory.

The architectural concept of the SCI Virtual Memory can be seen in figure 4. Like in software DSM approaches, the memory is distributed at the granularity of pages. A thread can access data in pages in the same node using standard memory accesses. In order to access data in remote pages, the page has to be mapped via SCI into the local address space first, before the access can be carried out. Once accomplished, memory accesses to remote pages can be done like local ones. The SCI hardware transparently forwards the memory request to the remote node without any further user interaction.

The basis for this mapping is formed by the SCI physical memory. It comprises the physical memory on all nodes and forms a global physical address space. From this address

space, pages can be mapped into the local PCI address space. This is done using the address translation tables of the SCI card. From there, the page is mapped into the thread's virtual address space using the memory management unit of the local processor.

These mappings can normally not be done statically at the beginning of the application's execution. The address translation tables in the SCI adapters as well as the PCI address space are limited resources and are normally not large enough to store all information needed for the whole runtime simultaneously. To solve this problem, a dynamic scheme has to be applied, that creates mappings on demand, similar to swapping pages into main memory by the virtual memory manager.

5.3 Implementation Issues

This architectural concept introduces some problems for a concrete implementation. First, in order to be able to perform dynamic mapping at any time, the physical location and state of any page in the system has to be known to any node. This can only be achieved by maintaining a global page directory. This task is further complicated by the fact that pages can be swapped to disk. To prevent this for pages that are mapped from remote nodes, these pages have to be temporarily locked. As this can potentially cause an unfair memory consumption by the Pthread application, the working set of pages has to be actively monitored and controlled. In the case that too many pages are locked by the Pthread application, some pages must be released to free memory for other applications.

Besides these pure implementation issues, some performance issues also have to be taken into account in order to achieve good performance. A central issue is the initial page distribution. It can be done statically, e.g. in a round robin fashion, or dynamically based on runtime information. Other performance related issues include the implementation of fast mappings and low management overhead for keeping the global page information consistent.

On top of the implementation of SCI Virtual Memory, a global abstraction of a process can be created representing the Pthread application. All threads created during the application's runtime are distributed onto the participating nodes. In order to be able to host the threads on one node, they are grouped together into teams and executed in a team process context. This fact, however, is hidden from the threads; they are not aware of this and execute in the same way as they would execute on a standard SMP system.

5.4 Summary

In summary, the SISCI Pthreads offer an SMP–like programming model on SCI–based clusters of PCs. They introduce a second programming paradigm for loosely coupled systems, allowing the port and the execution of new applications that haven't been run on clusters before. In particular, programs utilizing fine grain parallelism will gain some advantages from this new execution environment.

6 Related Work

Software environments for clusters of workstations is an area of intense research, but also shows a wide diversity with respect to programming models and interconnection technologies. The SMiLE project [11] at the Technische Universität München is focusing on SCI [22] as the underlying technology. Other projects dealing with SCI can be found at several institutions worldwide including the University of Paderborn [21], the University of Oslo [18] , the University of California at Santa Barbara [12], and the University of Florida [6]. The main focus of the majority of projects is primarily message passing and its efficient implementation on top of SCI shared memory. Only a few projects, like Split-C [13] at

UCSB, evaluate SCI as the base for globally shared memory programming environments. These projects have to be seen in the same context as the SISCI Pthreads.

Also work on software infrastructure environments or high performance virtual machines is done at several places. Of special interest and closely related to the described workpackages of the SISCI project are the Illinois High Performance Virtual Machines [3] and the Millipede project [14] as both are using Windows NT as the underlying operating system. The first one implements several programming libraries, including the Illinois Fast Messages [19] and MPI [24] while the latter constructs a cluster–wide shared object space and is intended as a low–level programming model or basis for a runtime system of a parallel programming language.

Work related to the development of the Common Message Layer for MPI [24] and PVM [25] can be found at Mathematics and Computer Science Division at the Argonne National Laboratory. Together with the implementation of the MPICH library [8], a standard implementation of the MPI specification, the differences between MPI and PVM have been investigated [7].

The Pthreads project utilizes mechanisms and basic concepts from pure software DSM approaches. A well–known representative of this group is the TreadMarks [1] implementation. Here, shared pages are replicated across the cluster and synchronized with the help of a complex multiple–writers protocol. To minimize the cost of maintaining the memory consistency, a relaxed consistency model is applied, Lazy Release Consistency [15]. However, unlike in the SISCI Pthread approach, where the sharing of the memory is transparently embedded into a global abstraction of a process, TreadMarks require the programmer to explicitly specify the shared segment. The same is also valid for the DSM-Thread [17] approach, which like the SISCI approach, is based on a POSIX compliant [23] API. This thread package, based on the FSU-Pthreads [16], allows the distribution of threads across a cluster interconnected by conventional interconnection networks, but does not provide complete transparency. It therefore forces the user to modify and partly rewrite the application's source code.

7 Conclusions and Future Work

As clusters of PCs are getting more commonplace, software developers are demanding standardized APIs to efficiently create portable code for this new architecture. The SISCI project attacks this problem by providing a complete software infrastructure on SCI based clusters. The project features a large variety of activities reaching from work on low–level APIs up to a large variety of applications from different areas. Two of the integral components of the project have been shown in detail: the Common Message Layer and the SISCI Pthread package.

The first one will be used to implement both MPI and PVM on a common base sharing as much functionality as possible. As shown here, this goal can be achieved without sacrificing performance based on a detailed knowledge of MPI and PVM function semantics. The next steps in this development will be further extensive testing of the existing CML implementation and the realization of MPI and PVM on top of it.

The second, the SISCI Pthreads, introduce a pure shared memory programming model into the world of cluster computing, as it is traditionally from tightly coupled systems like SMPs. It provides the user with a new view of the system using globally shared memory in cooperation with threads in contrast to message passing, the traditional programming model for loosely coupled distributed memory machines. By merging SCI remote memory operations and software mechanisms from traditional DSM systems, the necessary globally shared memory can be achieved while still providing complete transparency. This concepts opens up several challenging implementation issues that still have to be researched further. Next steps in the project also include the implementation of a first base version of the SCI Virtual Memory as well as the selection of a comprehensive benchmark suite.

In summary, the software infrastructure created by the SISCI project will feature the two mostly used parallel programming models, message passing and shared memory in cooperation with multithreading, on a single architecture. By having both programming paradigms on one system, each user has the chance to choose the right programming interface for their application needs and personal preference. This will not only ease the programmability and porting code onto SCI based clusters, but also form a stable platform for any further software development on this architecture.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, February 1995.

[2] N. Boden, D. Cohen, R. Felderman, J. Seizovic A. Kulawik, C. Seitz, and Wen-King Su. Myrinet: A Gigabit–per–Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[3] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Proceedings of SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[4] Dolphin Interconnect Solutions, AS. *Dolphin SBus–2 Cluster Adapter Card*, September 1996.

[5] Dolphin Interconnect Solutions, AS. *PCI–SCI Cluster Adapter Specification*, May 1996. Version 1.2.

[6] A. George, W. Phillips, R. Todd, and W. Rosen. Multithreading and Lightweight Communication Protocol Enhancements for SCI–based SCALE Systems. In *Proceedings of the 7th International SCI Workshop*, March 1997.

[7] W. Gropp and E. Lusk. Why are PVM and MPI so Different? Technical Report PREPRINT ANL/MCS-P667-0697, Mathematics and Computer Science Division, Argonne National Laboratory, June 1997.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High–Performance, Portable Implementation of the MPI Message Passing Interface Standard. Technical report, Argonne National Laboratory, Mississippi State University, 1996.

[9] D. Gustavson. The scalable coherent interface and related standards projects. *IEEE Micro*, 12:10–22, February 1992.

[10] D. B. Gustavson and Q. Li. Local-Area MultiProcessor: the Scalable Coherent Interface. In S. F. Lundstrom, editor, *Defining the Global Information Infrastructure: Infrastructure, Systems, and Services*, pages 131–160. SPIE Press, 1994.

[11] H. Hellwagner, W. Karl, and M. Leberecht. Enabling a PC Cluster for High Performance Computation. *SPEEDUP-Journal*, 11(1), 1997.

[12] M. Ibel, K. Schauser, C. Scheiman, and M. Weis. Implementing Active Messages and Split-C for SCI Clusters and Some Architectural Implications. In *Sixth International Workshop on SCI-based Low-cost/High-performance Computing*, September 1996.

[13] M. Ibel, K. Schauser, C. Scheiman, and M. Weis. High-Performance Cluster Computing Using SCI. In *Hot Interconnects V*, August 1997.

[14] A. Itzkovitz, A. Schuster, and L. Shalev. Supporting Multiple Parallel Programming Paradigms on Top of the Millipede Virtual Parallel Machine. In *Proceedings of the Workshop on High–Level Programming Models and Supportive Environments*. IEEE, April 1997.

[15] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Rice University, January, 1995.

[16] F. Müller. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of USENIX*, pages 29–42, January 1993.

[17] F. Müller. Distributed Shared–Memory Threads: DSM–Threads, Description of Work in Progress. In *Proceedings of the Workshop on Run–Time Systems for Parallel Programming*, pages 31–40, April 1997.

[18] K. Omang and B. Parady. Performance of Low-Cost UltraSparc Multiprocessors connected by SCI. In *Proceedings of Communication Networks and Distributed Systems Modeling and Simulation*, pages 109–115, January 1997.

[19] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 1997. To appear.

[20] S. J. Ryan, S. Gjessing, and M. Liaaen. Cluster Communication using a PCI to SCI Interface. In *IASTED 8th International Conference on Parallel and Distributed Computing and Systems*, Chicago, Illinois, October 1996.

[21] J. Simon and O. Heinz. SCI Multiprocessor PC Cluster in a Windows NT Environment. In *Workshops im Rahmen der 14. ITG/GI-Fachtagung Architektur von Rechensystemen*, pages 189–199, Rostock, Deutschland, September 1997.

[22] IEEE Computer Society. *IEEE Std 1596–1992: IEEE Standard for Scalable Coherent Interface*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, August 1993.

[23] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating Systems Interface (POSIX) — Part 1: System Application Interface (API)*, chapter including 1003.1c: Amendment 2: Threads Extension [C Language]. IEEE, 1995 edition, 1996. ANSI/IEEE Std. 1003.1.

[24] WWW:. MPI – The Message Passing Interface Standard
. http://www.mcs.anl.gov/mpi/index.html, December 1996.

[25] WWW:. PVM – Parallel Virtual Machine
. http://www.epm.ornl.gov/pvm/, December 1996.

[26] WWW:. SISCI
. http://www.parallab.uib.no/projects/sisci/, August 1997.