SCI–VM: A flexible base for transparent shared memory programming models on clusters of PCs

Martin Schulz
schulzm@in.tum.de

Lehrstuhl für Rechnertechnik und Rechnerorganisation, LRR–TUM
Institut für Informatik, Technische Universität München

Abstract. Clusters of PCs are traditionally programmed using the message passing paradigm as this is directly supported by their loosely coupled architecture. Shared memory programming is mostly neglected although it is commonly seen as the easier and more intuitive way of parallel programming. Based on the user-level remote memory capabilities of the Scalable Coherent Interface, this paper presents the concept of the SCI Virtual Memory which allows a cluster-wide virtual memory abstraction. This SCI Virtual Memory offers a flexible basis for a large variety of shared memory programming models which will be demonstrated in this paper based on an SPMD model.

1 Introduction

Due to their excellent price–performance, clusters built out of commodity–off–the–shelf PCs and connected with new low–latency interconnection networks are becoming increasingly commonplace and are even starting to replace traditional massively parallel systems. According to their loosely coupled architecture, they are traditionally programmed using the message passing paradigm. This trend was further supported by the wide availability of high–level message passing libraries like PVM and MPI and intensive research in low–latency, user–level messaging [17, 12].

Next to the message passing paradigm, which relies on explicit data distribution and communication, a second parallel programming paradigm exists, the shared memory paradigm. This paradigm, which offers a global virtual address space for sharing data between processes or threads, is preferably utilized in tightly coupled machines like SMPs that offer special hardware support for a globally shared virtual address space. It is generally seen as the easier programming model, especially for programmers that are not used to parallel programming, but it comes with the price of higher implementation complexity either in the form of special hardware support or in the form of complex software layers. This prohibits their widespread use on cluster architectures as the necessary hardware support is generally missing.

In order to overcome this gap and provide a bridge between tightly and loosely coupled machines, we utilize the Scalable Coherent Interface (SCI) [3, 13] as the interconnection technology. This network fabric, which is standardized in the IEEE standard 1596–1992 [14], utilizes a state of the art split transaction protocol and currently offers bandwidths of up to 400 MB/s. The base topology for SCI networks are small ringlets of up to 8 nodes which hierarchically can be organized to configurations of up to 64K nodes.

The core feature of SCI based networks is the ability to perform remote memory operations through direct hardware distributed shared memory (DSM) support. Figure 1 gives a general overview of how this capability can be applied. The basis is formed by the SCI physical address space which allows addressing of any physical memory location on any connected node through a 64 bit identifier (16 bit to specify the node, 48 bit to specify the physical address). From this global address space, each node can import pieces of remote memory into a designated address window within the PCI address space using special address translation tables on the SCI adapter cards. After mapping the PCI address space into the virtual memory of a process, the remote memory can be directly accessed using standard user–level read and write operations. The SCI hardware forwards these operations transparently to the remote node and, in case of a read operation, returns the result. Due to the pure hardware implementation avoiding any software overhead, extremely low latencies of about 2.3 μs (one way) can be achieved.

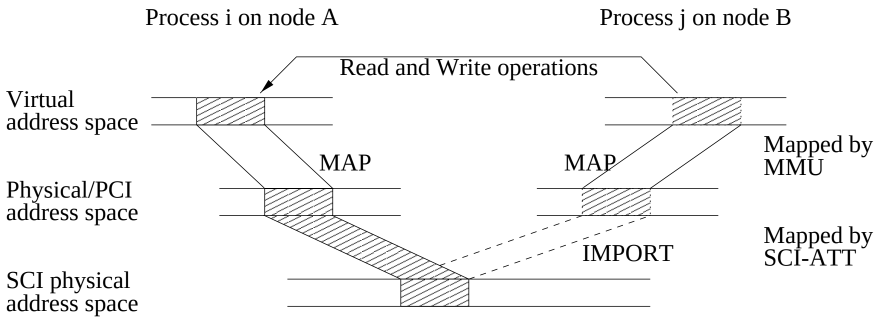

Fig. 1. Mapping remote memory via SCI from node A into the virtual address space of process j on node B.

Unfortunately, this hardware DSM mechanism can not directly be utilized to construct a true shared memory programming model as it is only based on sharing physical memory. It misses an integration into the virtual memory management of the underlying operating system. Extra software mechanisms have to be applied to overcome this limitation and to build a fully transparent global virtual address space. The SCI Virtual Memory layer, presented in this paper, implements these concepts on a Windows NT platform forming the basis for any kind of true shared memory programming on clusters with hardware DSM support.

On top of this virtual memory, it is then possible to implement a large variety of shared memory programming models with different properties customized for the needs of the programmers. This can range from distributed thread libraries over SPMD–style programming libraries, as presented later in this paper, to models with explicit data and thread placement. This flexibility forms a thorough basis for the application of the shared memory paradigm in cluster environments and therefore opens this promising architecture to whole new group of applications and users.

The reminder of this paper is organized as follows: Section 2 gives a brief overview of some related work. Section 3 introduces the concept and design of the SCI Virtual Memory a software layer that provides a global virtual address space which forms the basis for the cluster wide shared memory. Section 4 introduces one possible programming model supported by the SCI-VM. This paper is then rounded up by a brief outlook into the future in Section 6 and by some concluding remarks in Section 7.

2 Related work

Work on shared memory models for clusters of PCs is mostly done with the help of pure software DSM systems. 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, the Lazy Release Consistency [7]. However, unlike in the SCI-VM approach, where the sharing of the memory is transparently embedded into a global abstraction of a process, TreadMarks requires the programmer to explicitly specify the shared segment. The same is also valid for any other currently existing software DSM package, although they vary with respect to their APIs, consistency models, and usage focus. Examples for those kind of systems that have been developed for the same platform as the SCI Virtual Memory, Windows NT, are Millipede [6], Brazos [15], and the SVMlib [11].

Besides these pure software DSM systems that just deal with providing a global memory abstraction on a cluster of workstations, there are also projects that provide an execution model in the form of a thread library on top of the constructed global memory. One example for this are the DSM-Threads [10], which are based on a POSIX 1003.1c conforming API. This thread package, based on the FSU-Pthreads [9], allows the distribution of threads across a cluster interconnected by conventional interconnection networks. However, it also does not provide complete transparency. It therefore forces the user to modify and partly rewrite the application's source code.

Only a few projects utilize similar techniques than the SCI Virtual Memory and try to deploy SCI's hardware DSM capabilities directly. In the SciOS project [8], a global memory abstraction is created using SCI's DSM capabilities to implement fast swapping to remote memory. The University of California at Santa Barbara is also working on providing transparent global address space with the help of SCI [5]. Their approach, however, does not target a pure programming API, but rather to a hybrid approach of shared memory and message passing in the context of a runtime system for Split-C [4].

3 Global virtual memory for SCI based clusters

The main prerequisite for shared memory programming on clusters is a fully transparent, global virtual address space. This section describes a software layer that provides such a cluster-wide memory abstraction, the SCI Virtual Memory or SCI-VM. This forms the basis for any kind of shared memory programming model on top of an SCI based PC cluster.

3.1 Design goals for a transparent global memory abstraction

The design and the development of the SCI-VM was governed by several design goals and guidelines. They mostly originated from the initial project goal of creating an SMP-like environment for clusters as well as from the idea of providing a flexible basis for a large variety of shared memory programming models.

In order to fulfill these requirements, one of the main goals was to enable the SCI-VM to provide a fully transparent global address space hiding the distribution of the memory resources from the programmer. This includes the distribution and global availability of any static variables that are provided within the original process image. As a final result, the SCI-VM has to provide every process on each node with exactly the same view onto the complete memory system as only this allows the illusion of a single virtual address space that spawns the whole cluster.

Another important goal for the SCI-VM is the implementation of a relaxed consistency model together with the appropriate routines to control it. The SCI-VM should not fix a certain model, though, but rather provide mechanisms that allow higher software layers to adapt the model to their needs and requirements.

The implementation of the SCI-VM should be directly based on the HW-DSM provided by SCI rather than deploying a traditional SW-DSM system with all its problems like false sharing and complex differential page update protocols. Only this enables the SCI-VM to benefit from the special feature and the full performance of the interconnection fabric. Additionally, the implementation of synchronization primitives should directly utilize atomic transactions provided by SCI to ensure greatest possible efficiency.

3.2 The concept of the SCI-VM

As already mentioned in Section 1, SCI's hardware DSM support cannot directly be used to create a global virtual abstraction as it is required according to the goals stated above. For this endeavor, the remote memory capabilities have to be merged with well–known mechanisms from traditional DSM systems. The memory is distributed on the granularity of pages and these distributed pages are then combined into a global virtual address space. In contrast to pure software mechanisms though, no page has to be migrated or replicated. All remote pages are simply mapped using SCI's HW-DSM mechanisms and then accessed directly.

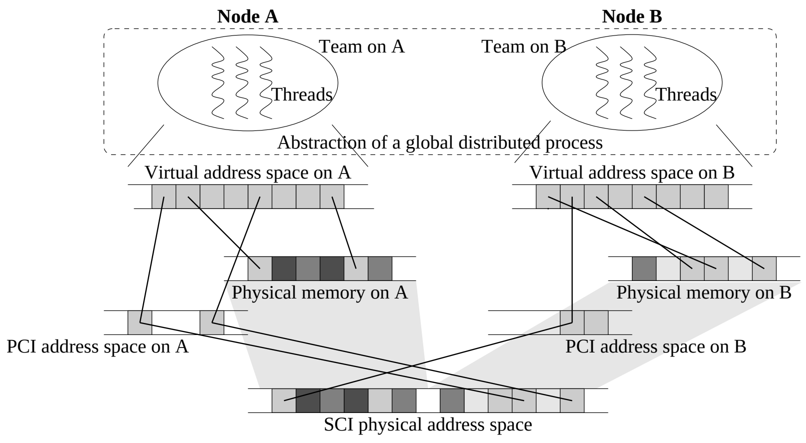

Fig. 2. Principle design of the SCI Virtual Memory

This concept is further illustrated in Figure 2 for a two–node system. In order to build a global virtual memory abstraction, a global process abstraction has to also be built with team processes as placeholders for the global process on each node. These processes are running on top of the global address space which is created by mapping the appropriate pages from either the local physical memory in the traditional way or from remote memory using SCI HW-DSM.

The mapping of the individual pages in done in a two–step process. First, the page has to be located in the SCI physical address space from where it can be mapped in the PCI address space using the address translation tables (ATT) of the SCI adapter cards. From there, the page can be mapped with the help of the processor's page tables into the virtual address space. Problematic is the different mapping granularity in these two steps; while the latter mapping can be done at page granularity, the SCI mappings can only be done at the basis of 512 KByte or 128 pages segment. To overcome this difference, the SCI-VM layer has to manage the mappings of several pages from one single SCI segment. The mappings of the SCI segments themselves will be managed with an on-demand, dynamic scheme very similar to paging mechanisms in operating systems.

3.3 Technical challenges of the SCI-VM design on Windows NT

The design discussed above presents several interesting implementation challenges. The most severe is related to the integration of the concepts presented above into the underlying operating system, in this case Windows NT and its virtual memory manager (VMM). Windows NT does not provide the capabilities to map individual page frames into the virtual address space of a process nor to intercept exceptions like the page fault before the VMM treats it. Currently both issues can only be solved by bypassing the OS and directly manipulating the hardware (the page table and the interrupt vector respectively). Experiments have shown that this approach, applied carefully, does not impair the stability of the operating system. However, we are looking for ways of cleanly integrating our concepts into the Virtual Memory Management of Windows NT to improve the robustness of the SCI-VM implementation.

Also, the integration of the SCI-VM concept into the currently existing hard- and software infrastructure imposes problems as it is mainly intended to provide support for sharing large, pinned segments of contiguous memory. To solve this problem, the hardware here also has to be addressed directly through the physical abstraction layer interface provided within the SCI device drivers.

A special problem is caused by the requirement of sharing the global variables within a program to complete the transparency for the user. These variables are part of the process image that is loaded at process creation time and are therefore treated by the operating system in a special way. In order to transparently share them, it is necessary to determine their location within the image file and to distinguish them from global variables used within the standard C++ runtime system as those must not be shared among the cluster nodes. To accomplish this, the concept of sections in Windows NT object files can be used. All global variables are located by the C++ code generator in sections called ".bss" or ".data". In all object files of the application, these sections have to be renamed with the help of a special patch program using a new, by default unused section name. After linking these modified object files with the unmodified runtime system, this renamed section in the executable containing all global variables can easily be identified (Figure 3.1).

In order to share them across all nodes, a global memory is allocated at program start in the standard SCI-VM manner and the initial information from the image file is copied into the global memory segment. From there it can be mapped into the virtual location of the global variables covering the information from the image file and replacing it with its globally shared counterpart (see also Figure 3.2).

3.4 Caching and consistency

An additional problem is imposed by the fact that, although the SCI standard [14] defines a complex and efficient cache coherency protocol, it is not possible to cache remotely mapped memory in standard PC based systems. This is caused by the inability of PCI based cards to perform bus snooping on the system bus. However, caching is necessary to overcome the problem of the large latencies involved when reading transparently from remote memory (around $6\mu s$ per access). The only solution here is to apply a relaxed consistency model that allows to enable caching while coping with the possible cache inconsistency.

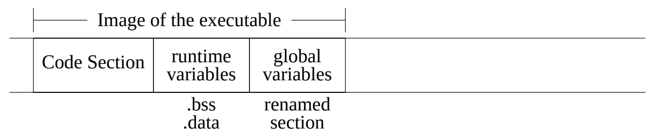

Figure 3.1: Address space of modified executable image

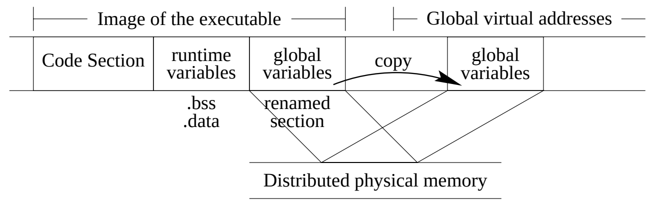

Figure 3.2: Address space after distributing the global variables

Fig. 3. Distributing an application's global variables.

In order to allow for the greatest possible flexibility, the SCI-VM layer does not enforce a specific cache consistency protocol, but merely provides mechanisms that allows applications or higher level programming models to construct application specific consistency protocols. For this purpose, the SCI-VM offers the functionality to flush the current memory state of the local node to the SCI network and to synchronize the memory of individual nodes with the global state by invalidating all local buffers including the caches. This functionality is implemented by routines that allow to control both the SCI adapter card with its internal buffers and the CPU's memory model by providing routines to flush the local caches and write buffers.

3.5 SCI-VM implementation

The main part of the SCI-VM is currently implemented in user level. This part is responsible for the setup of the SCI-VM, the distributed memory allocation, and the appropriate mappings from SCI to PCI and from PCI to the virtual address space. It is supported by two kernel mode drivers: the SCI device driver and a special VMM driver. The latter one was especially developed for the SCI-VM and helps in manipulating the virtual memory management. It provides the appropriate routines to map individual pages through maintaining the page tables, to intercept the page fault exception, and to configure the cacheable area to enable caching of the PCI address window used to map remote memory. This driver also includes cleanup routines that remove all modifications to the process's virtual memory configuration at process termination. This is necessary to maintain the stability of Windows NT.

4 Programming models based on the SCI-VM

The cluster–wide virtual memory abstraction provided by the SCI-VM offers the basis for the implementation of various shared memory programming models. Each of these models can differ in the manner of how threads of activities are created and controlled, in the memory consistency model, and in the degree of control the programmer has regarding the locality properties of underlying virtual memory.

4.1 An SPMD model on top of the SCI-VM

This section introduces in detail one example of these programming models that are based on the functionality of the SCI-VM. This specific model realizes a very simple concept of parallelism by allowing one thread of activity based on the same image file per node. Due to this execution mode and due to the synchronous manner of allocating global memory resources, which will be discussed below, this programming model can be classified as a Single Program Multiple Data (SPMD) model. Within the actual parallel execution, however, this synchronism is not fully enforced; it is possible to implement independent threads of control on each node as long as the restrictions mentioned above are observed.

4.2 Allocating shared memory

The SPMD programming model offers a central routine to allocate shared memory. It reserves a virtual address segment that is valid on all nodes within the cluster and performs the appropriate mappings of interleaved local and remote memory into this address range. The memory resources are thereby distributed in a round robin fashion at page granularity, i.e. at a granularity of 4 KBytes (on x86 architectures). This results in an even distribution of the memory resources across the cluster while trying to avoid locality hot spots.

This allocation routine has to be called by all nodes within the cluster simultaneously. It is therefore the main constraint that causes the SPMD properties of the programming model. This is currently necessary to request memory resources from all nodes during the allocation process. In the future, this constraint will be eliminated by a more complex SCI-VM implementation which allows to preempt the execution on remote nodes to request memory.

The allocation itself is a three step process. In the first step, the contribution of the local node to the new global segment is determined and an appropriate amount of local physical memory is allocated. In the second step, each node enters the physical locations of all local pages together with its node ID into a global list of pages. After the completion of this list, a virtual address segment is allocated at the same location on each node and all pages in the global list are mapped into this newly created address segment. Local pages are mapped directly from physical memory, whereas remote pages are first mapped into the PCI address range and from there into the virtual address space. Upon completion of the page mappings on all nodes, the allocation process is completed and the new virtual address is returned to the caller.

4.3 Synchronization

Shared memory programming models need mechanisms to coordinate accesses to shared data and to synchronize the execution of the individual threads. For this purpose, the SPMD SCI-VM programming model provides the following three mechanisms: global locks without any association to data structures, cluster–wide barriers, and atomic counters that can be used for the implementation of further custom synchronization mechanisms.

All of these mechanisms are implemented based on atomic fetch and increment transactions provided directly in hardware by the SCI adapter cards in the form of designated SCI remote memory segments. Read accesses to these segments automatically trigger an atomic fetch and increment. This transaction increments the value at the memory location specified through the read transaction atomically and returns the original value of the memory location as the result of the read operation.

While the atomic counters can directly be implemented using the atomic SCI transactions, locks and barriers need additional concepts. Locks are implemented using two separate counters, a ticket counter and an access counter. Each thread trying to acquire a lock requests a ticket by incrementing the ticket counter. The lock is granted to the thread if and only if the value of the ticket counter equals the value of the access counter. On release of the lock, the thread increments the access counter and thereby grants the next waiting thread access to the lock.

The implementation of barriers utilizes one global atomic counters that is incremented by each thread on entrance into the barrier. When the counter reaches a value that is divisible by the number of threads in the cluster, all threads have reached a barrier and are now allowed to continue. To distinguish multiple generations of one barrier an additional local, non-shared counter is maintained. This counter helps keeping track of the number of times the barrier has been executed.

Currently, all waiting operations required by locks and barriers are implemented using simple spin waits. This is not a major problem as there is currently only one thread per node allowed. In the future, however, this will be replaced using blocked and spin-blocked waits with a tight integration into the thread scheduling policies to allow more flexibility.

4.4 Coherency model

The SPMD programming model utilizes a relaxed consistency model to enable the caching of remote memory. Most of the synchronization mechanisms are implicitly hidden within the synchronization operations to ease the use of the programming models by hiding the relaxed consistency model from the programmer.

Barrier operations perform a full synchronization of all memories within the cluster. This is achieved by performing a flush of all local buffers (on the processor, in the memory system, and on the network adapter) to the network together with a complete invalidation of all caches and local buffers.

Locks provide a semantics similar to release consistency; after acquiring a lock, the local memory state is invalidated, and before the unlock or release operation, all local buffers are flushed to network to force the propagation of the local memory state to remote nodes. This semantics guarantees that accesses to shared data structures, which are guarded by locks to implement mutual exclusion, are always performed correctly in a consistent manner.

In addition to this implicit synchronization, the SPMD programming model also provides a routine for explicit synchronization. This routine performs a full flush and invalidation of the local memory on the node it has been called. However, this routine should only be necessary in special cases when working with unguarded data structures as all other cases are already covered by the implicit synchronization mechanisms.

4.5 Implementation

The SPMD programming model described in this section is implemented as a Win32 DLL which has to be linked to all images on all nodes. The startup of the SCI-VM and the SPMD data structures is automatically done from the main routine of that DLL without requiring an interaction with the application. This routine initializes all internal mappings and synchronizes the individual processes on all nodes.

Also, the termination of the application is controlled by the main routine of the DLL which is executed after the termination of the application. At this point, the DLL removes all mappings created during the runtime of the application and frees all memory resources that have been allocated.

5 Experiments and results

In order to evaluate the concept presented above, we implemented a few numerical mini kernels and ported an existing shared memory based volume rendering application using the described SPMD programming model. All codes utilized the shared memory in a fully transparent fashion without any locality optimizations. Also, the relaxed consistency model was applied only through the implicit mechanisms described above without requiring any code modifications.

5.1 Experimental setup

All of the following experiments were conducted on our SCI cluster in a two node configuration. This reduced version of our cluster was chosen due to current limitations in the implementation. Future version will overcome this and are expected to scale to distinctly larger cluster configurations. Each compute node in the cluster is a Pentium II (233 MHz) based PC with a 512 KByte L2 cache. The motherboards of these systems are based on the Intel 440FX chipsets. We deployed the PCI–SCI adapter cards, Revision D from Dolphin ICS [2]. These cards are equipped with the Link Controller LC-2 which allows to operate the SCI network at a raw bandwidth of up to 400 MB/s.

The base operating system for all experiments was a standard Windows NT 4.0 (Build 1381, SP 3). The driver software for the SCI adapter cards was also supplied by Dolphin ICS through the SCI Demokit in version 2.06.

5.2 Results: Numerical mini kernels

The first experiments where conducted using three numerical mini kernels: linear sum, standard dense matrix multiplication, and one iteration of the successive over-relaxation (SOR) method. All of these kernels are applied to one contiguous virtual memory segment in which the respective data structures are mapped.

For each of these codes, three different experiments were performed: a parallel version on top of the global memory provided by the SCI-VM, a sequential version on top of local memory to get a baseline comparison for the speed–up values, and a sequential version on top of global memory to get information about the overhead connected to using the global memory abstraction. Based on the results from these experiments, three key values were computed: speed–up as the ratio of parallel to sequential execution on local memory, fake speed–up as the ratio of parallel to sequential execution on global memory, and overhead as the ratio of sequential execution on local to global memory. The results for two different sizes of the virtual segment are presented in Table 1.

	Memory size	Local seq.	Speed–up	Fake s.u.	Overhead
Linear sum	256 KByte	4128 μs	0.92	1.50	63.01 %
	1024 KByte	16549 μs	1.06	1.63	53.56 %
Matrix multiplication	256 KByte	4912 ms	1.97	1.98	0.48 %
	1024 KByte	42731 ms	1.89	1.99	5.2 %
SOR iteration	256 KByte	296 ms	1.53	1.71	11.1 %
	1024 KByte	1225 ms	1.58	1.73	5.3 %

Table 1. Performance results for the numerical mini kernels.

The linear sum performs poorly; applied to small problem sizes, no speed–up, but rather a slight slow–down can be observed. Only when applied to larger problem sizes a small speed–up is possible. This can be explained through the fact that this algorithm traverses the global memory range only once and therefore does not utilize any temporal locality through the caches. Only the implicit prefetching of cache lines and the prefetching abilities that are implemented within the SCI hardware help the performance. The result is a rather high overhead and consequently a low speed-up. In addition, the linear sum is an extremely short benchmark which causes the parallel version of the algorithm to infer significant overhead which can be seen in the rather low fake speed–up value. This can only be improved by applying the algorithm to larger data sizes.

The matrix multiplication algorithm, however, behaves almost optimally by achieving a nearly perfect speed–up. This can be achieved by the high exploitation of both spatial and temporal cache locality which is documented in the extremely low–overhead number. When applying the scheme to larger data sizes, however, this overhead increases as the working set no longer fits into the L2 cache. Due to this, an increased number of cache misses has to be satisfied through the SCI network which causes additional overhead. This also leads to a slight decrease of the overall speed–up.

Also, the third numerical mini kernel, the SOR iteration, shows a very good performance. This is again the result of an efficient cache utilization for temporal and spatial locality. In contrast to the matrix multiplication, however, it is not necessary to keep the whole virtual address segment within the cache, but rather only a small environment around the current position within the matrix. Due to this, the algorithm does not suffer in performance when applied to data sizes larger than the L2 cache. It even benefits from the reduced overhead due to the larger data size resulting in a higher speed–up.

5.3 Results: Volume rendering

While the experiments using the numerical mini kernels provides detailed information about the raw performance and the problems of the SCI-VM and the SPMD model implementation, it also necessary to perform experiments using large complex applications to evaluate the full impact of the SCI-VM. For this purpose we utilized a volume rendering code from the SPLASH-II suite [18]. It was ported to the SPMD programming model by providing an adapted version of the ANL macros. The actual volume rendering code itself was not modified.

To evaluate this application the same experiments were conducted as described above for the numerical kernels. The results are summarized in Table 2. The values correspond to rendering times of single images. Pre– and postprocessing of the data was not included in the measurements. The data set used for all experiments is the standard test case provided together with the SPLASH distribution and has a raw size of roughly 7 MB. The work sharing granularity is set to blocks of 50 by 50 pixels to achieve the optimal tradeoff between management overhead and load balancing.

Sequential execution (local memory)	3522 ms
Sequential execution (global memory)	4136 ms
Parallel execution (global memory)	2381 ms
Speed–up	1.49
Fake speed–up	1.74
Overhead	17.43 %

Table 2. Performance results for the volume rendering application.

Also this complex application which handles large data sets exhibits a good performance on top of the transparent virtual address space. It is possible to achieve a speed–up of about 1.5 due to a low overhead caused by utilizing the transparent memory of only about 18 %. Most of the overhead prohibiting a larger speed–up is caused by the management and locking overhead of a central work queue which is a common bottleneck in any shared memory environment. This experiment shows that the concepts presented in this paper can be directly and efficiently applied to a large existing shared memory code without a major porting effort. Future experiments will extend this to further applications from different domains and will evaluate the SCI-VM in a large variety of scenarios.

6 Future work

The numbers above show that the SCI-VM concept together with the SPMD programming model allows to run parallel shared memory codes on clusters of PCs in a fully transparent fashion with good performance. However, the performance is not optimal, a price that has to be paid for the transparency. Further research will therefore investigate on adding optional mechanisms to the programming model that allow the programmer to incrementally optimize the data distribution. This enables a gradual transition of codes from a fully transparent model, which is easier for the programmer, to a more explicit model with more locality control for the programmer. This will be beneficial for porting existing codes to clusters as well as for implementing new applications using the shared memory paradigm because implementation and locality optimization can be treated as two orthogonal phases.

A second focal point of future research will lie on the development and implementation of other programming models for the SCI-VM that overcome the limitations of the current SPMD–like model. Of special interest will be the development of a distributed thread package on the basis of the POSIX thread API [16], which features full transparency in an SMP–like manner, and its counterpart, a completely explicit programming model which forces the programmer to manually place data and threads on specific nodes. On the basis of these two most extreme models, the tradeoffs between total transparency with its ease of use and full control with the possibility of ideal optimizations will be evaluated.

7 Conclusions

Clusters of commodity PCs have traditionally been exploited using applications built according to the message passing paradigm. Shared memory programming models, which are generally seen as easier and closer to sequential programming, are only available through software DSM systems. Currently, however, these systems lack performance and/or transparency. With the help of the SCI Virtual Memory presented in this work, it is now possible to create a fully transparent global virtual address space across multiple nodes and operating system instances. This is achieved by merging hardware DSM mechanisms for physical memory with page based techniques from traditional software DSM systems to provide the participating processes on the cluster nodes with an equal view onto the distributed memory resources.

On top of the global virtual address space provided by the SCI-VM, it is then possible to implement various programming models. One example of such a model, an SPMD–style programming environment, has also been presented here. It allows the programmer to easily develop parallel application in a synchronous manner which are then executed in top of a global address space. This programming model also provides appropriate synchronization mechanisms for shared memory programming in the form of global locks, barriers, and atomic counters. In order to allow the greatest possible efficiency, a relaxed memory consistency model is applied. The coherency is maintained using synchronization mechanisms which are, invisible for the programmer, implicitly implemented within the synchronization mechanisms in release consistency style.

This paper also presented the results of several experiments using this SPMD programming model on top of the SCI-VM to implement three numerical mini kernels. This evaluation shows, in most cases, a good performance with significant speed–ups. In some cases it is even possible to achieve a nearly optimal speed–up. Further experiments prove that also large applications, like the presented volume rendering code, can efficiently take advantage of the shared memory programming model. Without any locality optimization and with a minimal porting effort a speed–up of about 1.5 is achieved on a two node system.

In summary, the SCI-VM is the key component to open the cluster architecture to the shared memory programming paradigm which has been traditionally the domain of tightly coupled parallel systems like SMPs. Together with appropriate programming models on top, it will ease the programmability of clusters and allow to utilize the large number of existing shared memory codes directly in cluster environments.

Acknowledgments

This work is supported by the European Commission in the Fourth Framework Programme under ESPRIT HPCN Project EP23174 (Standard Software Infrastructure for SCI based parallel systems – SISCI). The SCI device driver is kindly provided in source by Dolphin ICS and the driver software to augment the Windows NT memory management has been implemented by Detlef Fliegl (LRR–TUM, TU–München).

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

3. 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.
4. 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.
5. M. Ibel, K. Schauser, C. Scheiman, and M. Weis. High-Performance Cluster Computing Using SCI. In *Hot Interconnects V*, August 1997.
6. A. Itzkovitz, A. Schuster, and L. Shalev. Millipede: a User-Level NT-Based Distributed Shared Memory System with Thread Migration and Dynamic Run-Time Optimization of Memory References. In *Proceedings of the 1st USENIX Windows NT Workshop*, August 1997.
7. P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Rice University, January, 1995.
8. P. Koch, E. Cecchet, and X. de Pina. Global Management of Coherent Shared Memory on an SCI Cluster. In *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 51–57, September 1998.
9. F. Müller. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of USENIX*, pages 29–42, January 1993.
10. 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.
11. S. Paas, M. Dormanns, T. Bemmerl, K. Scholtyssik, and S. Lankes. Computing on a Cluster of PCs: Project Overview and Early Experiences. In W. Rehm, editor, *Tagungsband zum 1. Workshop Cluster Computing*, number CSR-97-05 in Chemnitzer Informatik–Berichte, pages 217–229, November 1997.
12. S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 1997.
13. 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.
14. 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.
15. E. Speight and J. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Workshop*, August 1997.
16. 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.
17. T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.
18. S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH–2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.