Shared Memory Programming on NUMA–based Clusters using a General and Open Hybrid Hardware / Software Approach

Martin Schulz

Institut für Informatik
Lehrstuhl für Rechnertechnik und
Rechnerorganisation

Shared Memory Programming on NUMA–based Clusters using a General and Open Hybrid Hardware / Software Approach

Martin Schulz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. R. Bayer, Ph.D.
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. A. Bode
	2.	Univ.-Prof. Dr. H. Hellwagner, Universität Klagenfurt / Österreich

Die Dissertation wurde am 24. April 2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28. Juni 2001 angenommen.

Abstract

The widespread use of shared memory programming for High Performance Computing (HPC) is currently hindered by two main factors: the limited scalability of architectures with hardware support for shared memory and the abundance of existing programming models. In order to solve these issues, a comprehensive shared memory framework needs to be created which enables the use of shared memory on top of more scalable architectures and which provides a user–friendly solution to deal with the various different programming models.

Driven by the first issue, a large number of so–called SoftWare Distributed Shared Memory (SW–DSM) systems have been developed. These systems rely solely on software components to create a transparent global virtual memory abstraction on highly scalable, loosely coupled architectures without any direct hardware support for shared memory. However, they are often affected by inherent performance problems and, in addition, do not solve the second issue of the existence of (too) many shared memory programming models. On the contrary, the large amount of work done in the DSM area has led to a significant number of independent systems, each with its own API, thereby further worsening the situation.

The work presented within this thesis therefore takes the idea of SW–DSM systems a step further by proposing a general and open shared memory framework called HAMSTER (Hybrid-dsm based Adaptive and Modular Shared memory archiTEctuRe). Instead of being fixed to a single shared memory programming model or API, this framework provides a comprehensive set of shared memory services enabling the implementation of almost any shared memory programming model on top of a single core. These services are designed in a way that minimizes the complexity for target programming models making the implementation of a large number of different models feasible. This can include both existing and new application or application domain specific programming models easing both the porting of given and the parallelization of new applications.

In addition, the HAMSTER framework avoids typical performance problems of SW–DSM systems by relying on so–called NUMA (Non–Uniform Memory Access) architectures which combine scalability and cost effectiveness with limited support for shared memory in the form of non–cache coherent hardware DSM. Their capabilities are directly exploited by a new type of hybrid hardware/software DSM system, the core of the HAMSTER framework. This Hybrid–DSM approach closes the semantic gap between the global physical memory provided by the underlying hardware and the global virtual memory required for shared memory programming enabling applications to directly benefit from the hardware support.

On top of this Hybrid–DSM system, the HAMSTER framework defines and implements several independent and orthogonal management modules. This includes separate modules for memory, consistency, synchronization, and task management as well as for the control of the cluster and the global process abstraction. Each of these modules offers typical services required by implementations of shared memory programming models. Combined they form the HAMSTER interface which can then be used to implement shared memory programming models without much effort.

This capability is proven through the implementation of a number of selected shared memory programming models on top of the HAMSTER framework. These models range from transparently distributed thread models all the way to explicit put/get libraries and also include various APIs from existing SW–DSM systems with different relaxed consistency models. It therefore covers the whole spectrum of shared memory programming models and underlines the broad applicability of this approach.

The presented concepts are evaluated using a large number of different benchmarks and kernels exhibiting the performance details of the individual components. In addition, HAMSTER is used as the basis for the implementation or port of two real–world applications from the area of nuclear medical imaging, more precisely the reconstruction of PET images and their spectral analysis. These experiments cover both the porting of an already existing shared memory application using a given DSM API and the parallelization of an application from scratch using a new, customized API. In both cases, the system provides an efficient platform resulting in a very scalable execution. These experiment, therefore, prove both the wide applicability and the efficiency of the overall HAMSTER framework.

Acknowledgments

I would like to take this opportunity to express my deepest gratitude to the following people, for I realize that this work would not have been possible without their help, guidance, and support.

First, I am indebted to my two advisors, Prof. Dr. A. Bode, chair of LRR–TUM, and Prof. Dr. H. Hellwagner, now at the University of Klagenfurt. Prof. Hellwagner initially took me in as his student and after his move to Klagenfurt, Prof. Bode took over without hesitation. Together they provided me with an excellent research environment and gave me the greatest possible freedom to pursue my work and interests. In addition, despite their busy schedules, they both always had an open ear for me.

I would also like to direct my special thanks to Dr. Wolfgang Karl who, in his function as leader of the architecture group at LRR, significantly contributed to the excellent research environment. In addition, his support and the many long, informal discussions and helpful comments regarding my work were invaluable, especially in times when I seemed stuck.

I would also like to thank my many colleagues at LRR–TUM; I enjoyed very much working with them and the many discussions we shared over coffee. Especially I would like to mention my office mates over the years: Phillip Drum, Michael Eberl, Detlef Fliegl (who also contributed significantly to the Windows NT driver for the SCI-VM and provided personal system administration support), Günther Rackl, and Christian Weiß. Additionally, I would like thank Jie Tao, whose pleasant disposition and enthusiasm made it a pleasure to work with her.

Our secretaries, especially Mrs. Eberhardt, Mrs. Wöllgens, and Mrs. Brunnhuber, deserve my special thanks for their help in maneuvering through the many administrative obstacles ranging from contract issues to business trip applications. Without them, I would still be sitting here in a pile of paper work. I would also like to thank Klaus Tilk, our system administrator, for keeping our systems alive and healthy.

I do not want to forget to mention Prof. Dr. A. Chien and the members of his Concurrent Systems Architecture Group (CSAG) from the University of Illinois at Urbana–Champaign (now at the University of California at San Diego). During my stay in this group and during my Master's work there, they introduced me to a scientific style of working from which I still very much profit.

The European Union/Commision deserves credit for the funding it provided for the majority of my work. This was done mainly within the ESPRIT projects SISCI (EP 23174) and NEPHEW (EP 29907). In addition, the many partners Europe–wide involved in these projects provided an additional source of inspiration and support. Special thanks go to Dolphin ICS, most specifically to Kåre Løchson, Hugo Kohmann, Torsten Amundsen, and Roy Nordstrøm, who were involved not only in these ESPRIT projects, but also technically supported our work within SMiLE.

The work within NEPHEW also brought me in contact with the Clinic for Nuclear Medicine at the “Klinikum Rechts der Isar”. From this cooperation I received valuable input for the evaluation part of this work. This would not have been possible without help from Frank Munz, Sibylle Ziegler, and Martin Völk and I am very thankful for the time and energy they invested.

On a more personal note, I am indebted to my friends and my whole family who were always at my side providing constant encouragement and support. I would especially like to thank my good friend, Dr. Johannes Zimmer, for all our long talks and for his prolific help with LaTeX. I would also like to express my deepest gratitude to my parents. From early on, they sparked and encouraged my interest in learning and provided unconditional support in all of my endeavors, both financially and (most important) spiritually. Last, but certainly not least, I would like to thank the love of life, my wife Laura, who not only took the time to review my thesis and to give valuable comments regarding language and style, but who also supported me throughout the entire time and with her whole heart.

Martin Schulz
April 2001

Contents

List of Figures

List of Tables

Chapter 1

Motivation

Even with today's capabilities of modern microprocessors, a whole range of applications exist which are too computationally intensive in order to be solved by a single processor within a reasonable amount of time. Only by spreading the work across multiple processors can the computational power necessary for these kind of applications be aggregated. This has led to the development of parallel architectures, but has also raised the question of their programmability; how can applications organize and manage multiple activities on different processors and how can information be shared among them?

For this purpose, a variety of different parallel programming paradigms has been proposed, each providing a different approach on how to specify the intended parallelism. The two most common among them are the message passing and the shared memory paradigm; the first is based on explicit messages being sent between (in principle) independent threads or tasks on different processors while the latter one enables a single application running across all nodes to share data implicitly through a common global address space and uses additional mechanisms for explicit synchronization.

Each of these two paradigms has its strengths and weaknesses. It depends on the target application and/or on the user's preference and previous experience which paradigm is more suited for a specific parallelization. In general, however, the shared memory paradigm is seen as easier and more intuitive since it is based on a single global address space. This is very close to the sequential model, which is familiar to virtually any programmer, and hence easy to learn and understand. It also provides a simple and straightforward parallelization without major code changes, in most cases. In addition, the use of a global address space also eliminates the need for an explicit data distribution across nodes, which normally requires major changes in global data structures. This has to be seen in contrast to message passing programming models which do require an explicit data distribution across the utilized processing nodes and hence often lead to major code changes during the parallelization of an application. The strongholds of message passing, on the other hand, have to be seen in the direct mapping of the programming paradigm with its concept of processing nodes and messages being sent between them to the underlying hardware. In most cases, this ensures a more predictable performance and allows for easier performance optimizations as the programmer has to take the distributed nature of the underlying system into account from the beginning.

A free choice of a certain paradigm is in reality often restricted by external circumstances, mostly the missing support for one of them on a particular platform. This problem can actually be found on most of the current parallel architectures or systems and hence severely limits their flexibility since it introduces additional complexity for the users. They need to adjust their codes to the subset of available programming models rather than being able to choose the models most appropriate for their target application.

Researchers in the field of parallel programming have therefore focused for a long time on eliminating this problem by providing comprehensive software infrastructures for parallel architectures, which include efficient implementations of programming models from both major paradigms on top of a single system. This thesis is a contribution to these efforts and aims at providing shared memory programming for cluster architectures with minimal hardware support in the form of a NUMA interconnection fabric, traditionally the domain of message passing models. More specifically, this work concentrates on clusters connected with the Scalable Coherent Interface (SCI) [75], an IEEE–standardized [92] state-of-the-art System Area Network (SAN) with NUMA characteristics, and provides a working prototype for these architectures.

It goes, however, one step beyond the traditional work done in this field in the respect that not only a single, new shared memory programming model is being implemented on top of the target architecture, but rather a flexible environment is created that enables the implementation of almost any shared memory programming model on top of a single shared memory core. The result is a general and open shared memory programming environment for NUMA–based clusters called HAMSTER (Hybrid dsm–based Adaptive and Modular Shared memory archiTEctuRe). This environment provides a maximum of flexibility to the users since it does not require them to adjust their codes to a specific shared memory programming model or API, but instead enables them to choose the ones appropriate for their applications. This significantly increases the attractiveness of shared memory programming on NUMA–based clusters and hence represents an important step towards the envisioned complete software infrastructure for these architectures.

1.1 Current Deficiencies of the Shared Memory Paradigm

Despite the advantages of the shared memory paradigm with regard to easier programmability, the message passing paradigm currently dominates the area of High Performance Computing (HPC). For one, this can be attributed to the restricted availability of shared memory programming models on large scale systems; while message passing models, mostly represented through libraries like MPI [152] or PVM [60], have been successfully implemented on almost all common parallel architectures, programming models based on the shared memory paradigm are mostly restricted to tightly coupled machines like SMPs, which provide only a limited scalability. Only these machines feature the hardware mechanisms which enable the creation of a consistent global virtual address space.

Such support is generally not available on more scalable, loosely coupled architectures. Therefore, shared memory programming models can only be implemented using software emulation mechanisms, as is done in SoftWare Distributed Shared Memory (SW–DSM) systems. They create the illusion of a global address space and enable the user to utilize this illusion directly in their applications. These systems, however, face several inherent performance problems which are unavoidable in pure software solutions. Examples include false sharing due to large sharing granularities or complex differential update protocols responsible for packing and sending data between nodes [143]. Even though vast amounts of research worldwide have been devoted to these performance problems, no sufficient solution has been found, thereby confining SW–DSM systems mainly to the realm of research and lacking broad acceptance.

Besides these architectural deficiencies, the use of shared memory programming is further hindered by the fact that a large variety of different shared memory programming models and APIs have been developed and are present on different systems. This reaches from pure thread models, as can be found in SMPs, to explicit one–sided communication libraries and also includes all the different APIs from the individual SW–DSM systems. This greatly reduces the portability of shared memory codes since they often need to be retargeted to different APIs when being moved from one machine to another. While this porting is certainly less complex than porting codes between paradigms, it still requires a significant amount of work, increases cross–platform code maintenance, and hinders code reuse. In addition, it steeply increases the learning curve for users since they have to deal with the subtle details in both syntax and semantics of a large variety of models. All of this lowers the acceptance of shared memory programming and hinders its wider use among the HPC user community.

In summary, these deficiencies render the shared memory paradigm unsuitable for many problems and application areas, especially those generally classified as HPC. Only in certain areas, mainly those dealing with small number of processors and dynamic task structures, the shared memory paradigm has gained significant acceptance and is widely available in the form of thread libraries. In most other areas, however, message passing is dominant. This is mainly due to the standardization towards only a few portable libraries and to their straightforward implementation, even on loosely coupled architectures.

1.2 The Approach

The work presented here aims at overcoming these shortcomings connected with shared memory programming and to provide users with a choice of the appropriate programming paradigm for their application needs, even on loosely coupled architectures. This is done by providing a general and open framework for shared memory programming. This framework is based on the following three principles:

1.2.1 Benefiting from the Rise of NUMA Architectures

As briefly mentioned above, an implementation of shared memory programming systems on loosely coupled architectures without any hardware support possesses some inherent performance problems. This work focuses therefore on architectures that offer some support for shared memory programming models, but maintain the scalability of loosely coupled architectures, as is found in NUMA (Non Uniform Memory Access) systems. More specifically, this work focuses on clusters of PCs connected by using the Scalable Coherent Interface (SCI) [75, 92] which enables a direct remote memory access to any physical memory location in hardware. This is accomplished using a global physical address space created by the network which can be used as the basis for shared memory programming.

In general, NUMA architectures, whether they are cluster–based or implemented as more integrated Massively Parallel Processor (MPP) systems, require only limited hardware resources for their remote memory access capabilities and hence can be implemented in a straightforward manner. In addition, they are not based on a centralized component or global maintenance protocol and therefore show very favorable scaling properties. On the software side, the remote memory access capabilities can be exploited for both shared memory programming and high performance messaging [51], making these kinds of platforms very attractive for a broad range of users and application domains. Due to these properties, all kinds of NUMA architectures are becoming increasingly popular and could rise to become the dominating architecture in the near future, thereby making work in this direction very promising.

1.2.2 Exploiting Hybrid–DSM Techniques

In order to exploit such NUMA architectures in a loosely coupled scenario, like it is given in cluster architectures, a new type of DSM system must be developed. On one side, it has to directly exploit the benefits provided by the NUMA hardware, while on the other side, a strong software component needs to ensure a global view onto the distributed system across the individual nodes and distinct operating system instances. This results in a hybrid DSM scheme, which merges the hardware DSM support found in NUMA systems in the form of remote memory access capabilities with the global memory management of traditional software DSM systems. In such a system, any communication is efficiently performed directly in hardware without any protocol overhead and it thereby forms a bridge between SW–DSM systems on cluster architectures on one side and complex hardware supported shared memory systems on the other.

For the implementation of a hybrid DSM system, one major challenge needs to be solved: the gap between the global physical memory provided by the NUMA system and a global virtual memory abstraction required for shared memory programming needs to be closed. This is achieved in tight cooperation with the underlying operating system through a global memory management component. This software module extends the concept of virtual memory to all participating nodes within the system. The result is a global virtual memory abstraction backed transparently by distributed physical resources which can then directly be used for any kind of shared memory programming.

1.2.3 Towards an Open Architecture for Shared Memory

Such a hybrid DSM system alone, however, has only the ability to solve one of the problems mentioned above, namely the performance problems of DSM systems in loosely coupled environments. The second problem, the availability of (too) large a number of different

APIs and programming models would remain, if not further worsened, because the new hybrid DSM would introduce yet another API. On the other side, it is also illusory to work towards a unified programming model since it would be impossible to convince people to retarget applications and systems towards a new API based only on its claims to be “better” or “unified”.

Therefore, another approach has to be found to help users deal with the abundance of programming models in a manner that allows them to avoid extra porting or maintenance efforts for their applications. The solution for this problem proposed by the work presented here and implemented in HAMSTER (Hybrid–dsm based Adaptive and Modular Shared memory archiTEctuRe) consists of a comprehensive framework which enables the availability of multiple models on top of a single system. This system is designed in a way that additional programming models can easily be added without requiring much implementation complexity thereby making it feasible to create as many programming models as required.

With this approach, the user is no longer bound to the one programming model provided by the target system, but has the option to choose a model fitting to the particular application. This can either be an existing programming model or API adopted from another platform to ease the porting of existing applications or a new programming model specifically deployed to support a special class or domain of applications. In both cases, the user is alleviated from tedious porting and implementation efforts thereby significantly easing the use of shared memory environments.

1.3 Contributions

This work provides the following main contributions:

- **Modular framework for arbitrary shared memory programming models**

  The HAMSTER system, as presented in this work, provides the ability to implement almost any kind of shared programming model on top of a single core. With that, it enables users to work with different models on a single system at the same time without having to worry about obtaining, installing, and maintaining a large number of models or porting a given application to the particular model available on the target machine, which is often a cumbersome task. This system therefore allows users to choose the programming model most suited to the application needs for an efficient implementation rather than to the hardware and software constraints on the target platform.

  This capability has been tested within this work based on several selected programming models. These models range from full thread libraries to explicit shared memory programming models and also include the emulation of various APIs from traditional SW–DSM systems. Each of these programming models was implemented with only limited complexity proofing both the feasibility and the flexibility of the pursued approach.

- **Orthogonal support for shared memory services**

  This ability to support almost any programming model on top of a single framework is created by providing a large variety of shared memory services. These services include facilities for memory, consistency, synchronization, and task management and are organized in several distinct and independent modules. The modules themselves are organized in a way that ensures a full orthogonality between them, i.e. each service can be used without unwanted side effects. This guarantees the required flexibility for the easy and safe implementation of shared memory programming models on top of these services.

  Each of these management modules has been implemented in a way ensuring a maximum of flexibility and performance. Special care in this respect has been given to synchronization and consistency enforcing mechanisms. Both have been implemented in a very lean manner taking direct advantage of present hardware mechanisms, thereby avoiding unnecessary overheads and providing a rich functionality. This enables the user of HAMSTER to create the necessary synchronization constructs and consistency models for the intended target programming models in an efficient and straightforward manner without high implementation complexity.

- **Hybrid–DSM core for NUMA–based clusters**

  In order to guarantee an efficient performance for all programming models implemented on top of HAMSTER, the framework includes a new type of DSM system, a hybrid hardware / software DSM system. It directly exploits the hardware capabilities of the underlying NUMA architecture without losing the two main advantages of loosely coupled architectures, scalability, and cost effectiveness. It provides the necessary support to transform NUMA support for a global physical memory into a global virtual memory abstraction suitable for the implementation of shared memory programming models.

  A prototype based on these concepts has been implemented on top of clusters of PCs interconnected by the Scalable Coherent Interface (SCI) [75, 92]. This prototype shows both the feasibility and the efficiency of the approach presented in this work. The overall concepts, however, are not restricted to this specific NUMA architecture alone, but can be principally be applied to any NUMA–like architecture and hence represents a general approach for shared memory programming on these kinds of architectures.

- **Detailed performance evaluation**

  This work includes a detailed performance assessment of the overall system using several applications and computational kernels from various areas as well as an evaluation of the individual components used to create the shared memory services and management modules. This allows a detailed insight into the system characteristics and provides useful hints for future users of the system.

In summary, this work provides a comprehensive framework which aims at overcoming the current problems of shared memory programming for loosely coupled architectures with NUMA characteristics. With its ability to support arbitrary shared memory programming models, it guarantees a high degree of flexibility and openness. Additionally, these properties are achieved without sacrificing performance since the overall system is based on an efficient Hybrid–DSM core, capable of directly exploiting the advantages of NUMA architectures.

1.4 Thesis Organization

The thesis is organized as follows: Chapter 2 introduces the fundamental concepts of this work from both the programming model and the architectural point of view and briefly discusses the context from which it originates. The work itself, the HAMSTER framework, is introduced in Chapter 3 together with the discussion of its fundamental design guidelines and principles.

The following chapters discuss the various components of the HAMSTER system in detail: Chapter 4 presents the hybrid hardware / software distributed shared memory system, followed by the discussion of the various shared memory services provided by several independent modules in Chapter 5. These services are then used to form shared memory programming models, which is shown in Chapter 6.

In Chapter 7, the overall system is evaluated using two larger applications from the area of nuclear medical imaging. Based on these codes, both HAMSTER's capabilities to support new codes on top of special programming models and the porting of existing codes using standard programming models have been evaluated proving the efficiency and wide applicability of the HAMSTER framework.

In conclusion, Chapter 8 offers a brief outlook on future work and closes with some final remarks.

The work is completed with a short manual containing information on how to run applications using HAMSTER in Appendix A and with a small example of a HAMSTER–based programming model in Appendix B.

Chapter 2

Background

Before going into the details of the shared memory framework for NUMA–based clusters proposed in this thesis, it is necessary to look at a few basic concepts. This chapter first discusses the principles of shared memory from both the programmer's and the architectural point of view and presents the problems associated with different approaches. It then introduces a NUMA–based cluster interconnection fabric — the Scalable Coherent Interface (SCI) — as a suitable tradeoff between hardware support for shared memory programming and architectural scalability and provides a detailed overview of its history, current status, and its use in both the commercial world and the research community. The chapter ends with a brief description of the context of this work.

2.1 Shared Memory Programming

Shared memory programming generally refers to parallel programming models that exhibit a global virtual address space which can then be used to share data implicitly among multiple tasks potentially located on different processors. Synchronization between the individual tasks within a parallel application is accomplished by using additional explicit mechanisms specific to the programming model. Additionally, the task model, i.e. the control over the concurrent activities within a parallel application, varies from model to model and can range from a static task assignment to processors to a fully dynamic scheme with task creations and destructions during the runtime of the application.

Compared to other paradigms, this basic approach is often seen as more intuitive and easier to learn since it eliminates the need for explicit communication and data distribution. Rather data is shared in the global address space and can be accessed from there by any node at any time. This simplicity makes the shared memory paradigm especially suitable for programmers with little experience in parallel programming or for those lacking a formal computer science background.

2.1.1 Definition of Shared Memory Programming Models

The exact connotation of the term "Shared Memory" is influenced by the direction from which it is seen: when looking at it from an architectural view, it normally denotes the ability of the architecture to access global data, while from a programming model point of view, the sharing of data is emphasized. In the latter context, often the term "Shared

Variable Programming" [33] is used as a synonym.

In the context of this work, the term "shared memory programming model" refers to parallel programming models which exhibit a single global virtual address space to all tasks within the parallel program. This address space is then used for any communication between the tasks and does not require the use of any explicit communication or copy routines. Besides this basic definition of memory behavior, no further restrictions to the type of used programming model is set forth. This refers especially to synchronization mechanisms and the task structure offered by the programming model. Their definitions are intentionally left open to include a wide range of models and hence to increase the scope of this work.

2.1.2 Available Models and APIs

Based on this general definition, many different types of shared memory programming models and systems exist. The following list provides an overview of the major subtypes:

- **Thread libraries**

  The most typical representatives of shared memory programming models are thread libraries. Such libraries are present in any modern operating system and are mostly used to exploit the capabilities of SMPs. Examples for thread libraries are *POSIX threads* [219] as implemented e.g. in Linux, *Solaris™ threads* [148], a slight variation for SUN Solaris™ platforms, *Win32 threads* [154] available in Windows NT™ and Windows 2000™ , and *Java™ threads* [104] as part of the standard Java™ Development Kit (JDK).

  Next to these operating system inherent implementations of thread packages, also several standalone thread libraries exist, mostly from the research community. Examples for this are the FSU threads [163] or the ACE thread package [191].

- **Distributed Shared Memory systems**

  Distributed Shared Memory (DSM) systems aim at emulating a shared memory abstraction on loosely coupled architectures in order to enable shared memory programming despite missing hardware support. They are mostly implemented in the form of standard libraries and exploit the advanced user–level memory management features present in current operating systems. In recent years, a large number of such systems, each with its own API, have been developed by the research community, but only one, the TreadMarks system [5], has made it to the product stage. Examples for research systems are IVY [136], the predecessor of all DSM systems, Munin [29], Brazos [209], Shasta [189], and Cashmere [213]. A more comprehensive list and a detailed description of DSM systems can be found in Chapter 4.1.

- **Macro packages for parallel programming**

  Special macro packages can also be used for the construction of parallel shared memory applications, although they do not represent full programming models. They merely provide an easy and high–level way to specify the intended parallelism in the target applications and provide the appropriate preprocessor rules to transparently transform these into intermediate applications based on a second, often platform specific shared memory programming model at a much lower level of abstraction. The most prominent example for this kind of approach are the ANL or *parmacs* m4 macros, which are used within the SPLASH [206, 233] benchmark suite and exist for a variety of different shared memory platforms.

- **Parallel Programming Languages with global address space**

  In addition to the library–based programming models discussed so far, parallel programming languages based on the shared memory paradigm also exist. The most prominent example among them is High–Performance Fortran (HPF) [236], a data parallel language, which allows the specification of parallelism based on shared arrays. A compiler then transfers the program into a message passing code which can then be executed on the target machine. The programmers, however, do not need to deal with the translated code and therefore maintain the shared memory view on their application at all times. Due to these properties, HPF has gained acceptance among the Fortran and scientific computing community and is available on most MPPs.

  Besides HPF, also further parallel shared memory languages, again mostly research–based, exist. An examples for this is ZPL [139], developed and maintained at the University of Washington, a data–parallel array programming language.

- **Program Annotation Packages**

  Between library based models and full shared memory programming languages, a third principal approach exists which is based on code annotations. A quite renowned approach in this area is OpenMP [172], a newly developed industry standard for shared memory programming on architectures with uniform memory access characteristics. In contrast to HPF , OpenMP is based on task parallelism and focuses mostly on the parallelization of loops.

  OpenMP implementations use a special compiler to evaluate the annotations in the application's source code and to transform the code into an explicitly parallel code which can then be executed. In contrast to full parallel languages, annotated codes can still be compiled with standard compilers which ignore the annotations and produce normal sequential codes. Hence a common code basis of sequential and parallel versions can easily be maintained.

- **Explicit shared memory**

  Lying on the border of the definition of shared memory programming models are explicit shared memory programming models. In such systems, accesses to remote data are no longer fully transparent, but are normally executed through special remote memory access routines provided by the programming model which copy the data to the intended remote location. Accesses to local data however are still carried out directly without the need to call specific routines. An example for such an explicit model is the so called “shmem” library provided by SGI and Cray for the MPP machines [98].

One step further than the Cray shmem library are approaches like Linda [230] or Java™ Spaces [158]. They no longer provide even a transparent access to any part of the data, but rather require users to explicitly call routines for data storage and retrieval at any time. However, a global address space of some kind is also present in these approaches, making them at least related to shared memory.

2.1.3 Shortcomings of Shared Memory Programming

Despite the advantages of shared memory programming discussed above, this paradigm has by far not reached the same level of acceptance as the message passing paradigm, especially in the area of High Performance Computing (HPC). This can be attributed to mainly two shortcomings inherently connected with shared memory programming:

Abundance of Different APIs without a Dominant Standard

One shortcoming is already directly visible from the pure size of the list of examples of shared memory programming models presented above. This abundance of different models severely restricts both code portability and reuse since codes and code parts frequently have to be ported from one shared memory programming model to another. In addition, each programming model may have subtle differences in both syntax and semantics with regard to their APIs thereby causing ports to be cumbersome and error prone. This results in a steep learning curve for programmers dealing with more than one system or programming model.

Limited Scalability of the Underlying Architectures

The second shortcoming comes from an architectural point of view. Shared memory programming normally requires hardware support to be present in the underlying hardware architecture. Traditionally, such support is only available in tightly coupled systems, such as Symmetric MultiProcessors (SMPs). These architectures, due to their need for a tight coupling between the individual processors, generally exhibit rather unfavorable scaling properties and hence provide only a limited amount of aggregated computational power limiting their use.

2.2 Architectural Support for Shared Memory

In order to further illustrate this shortcoming of limited scalability and to point towards approaches to overcome it, it is necessary to briefly talk about the various types of memory organizations in existing architectures and their support for shared memory programming. This will help understand the implications introduced by hardware shared memory support

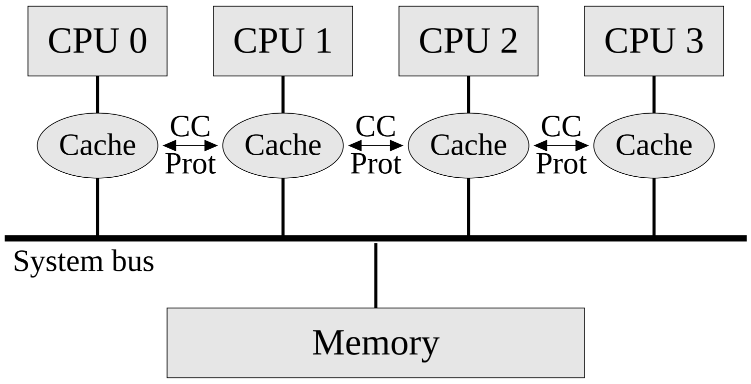


Figure 2.1 Memory organization of UMA architectures.

with respect to scalability and performance and will allow to judge the suitability of existing architectures for shared memory programming.

To structure this discussion, a simple and straightforward classification of architectures based upon their type of global memory access is used. Based on this scheme, three major types can be identified:

- **UMA – Uniform Memory Access architectures**
  In this class of architectures, any memory location can be reached from any processor with the same access characteristics in terms of cost (i.e. uniform)

- **NUMA – Non Uniform Memory Access architectures**
  Like UMA systems, these architectures allow any processor to access any memory location directly, but the access characteristics varies with respect to the access latency.

- **NORMA – NO Remote Memory Access architectures**
  In this class of architectures no direct hardware support is available to access memory locations other than those local to a processor.

2.2.1 Full Hardware Support within UMA Systems

The first class of architectures discussed in more detail are the UMA (Uniform Memory Access) architectures. Their general structure is shown in Figure 2.1. A number of processors, potentially with additional caches in the memory path, are connected to a single global memory through a common system bus. Such busses are normally only able to cover short distances due to their electric properties, which leads to systems with all processors installed within one box. Such systems are therefore often referred to as tightly coupled systems.

This organization is currently widely used in Symmetric MultiProcessor (SMP) machines. In these machines several processors of the same type and speed are coupled to a single system. In addition, no one of these processors has a special role; instead all nodes are configured in a fully symmetric way[1]. Typical configurations range from 2 or 4 processors in commodity PC–based system [38] to up to 16 or 32 processors in high–end servers, like the SGI Power Challenge™ [58] and the SUN Enterprise systems [251]. .

As any processor is connected to the global memory through the system bus, it can directly access any memory location resulting in a full hardware support for shared memory. In the presence of caches, SMP machines also provide an appropriate cache coherency protocol [175], which automatically guarantees a coherent state of all involved caches. This maintains their transparency known from single processor systems also in such a multiprocessor scenario. On the software side, these systems are normally equipped with the appropriate operating system support for multiple processors, especially with respect to a global virtual memory management. The shared memory can therefore be easily exploited by the users without the need for any additional software complexity.

The major drawback to these architectures is their limited scalability. Any access to the memory has to be performed across a single bus, leading to a major bottleneck with rising numbers of processors. This can be seen with respect to both performance, due to bus contention, and space, due to the limited distances such buses are able to cover. In recent years, a significant amount of effort has been invested in the area of high–end servers to reduce this bottleneck, leading e.g. to the PowerPath™ bus system [58] used in the SGI systems. However, the principal problems remain. Due to this, the number of processors within an SMP system with which a system is still able to provide sufficient performance will always be limited.

The only way to overcome this problem is to leave the bus–based design and deploy new, more scalable interconnection technologies which do not suffer from the contention problem. An example for such a novel approach can be found in the SUN Starfire architecture [34] which couples up to 16 boards with 4 Sparc processors each using point–to-point links through a highly efficient crossbar backplane. Such a design, however, requires a very complex hardware in order to keep the total memory latency both uniform and at an acceptable level.

2.2.2 Improving Scalability with CC–NUMA

This scalability problem of SMPs with UMA organization has led to the development of CC–NUMA machines. In this class of architectures, whose typical layout is depicted in Figure 2.2, again any processor has direct access to any memory location in the whole system. This memory, however, is now distributed to the individual processor nodes and no longer only available at a single location. This introduces a distinction between local and remote memory, i.e. memory on the same or on another processor node than the accessing processor. This distinction with respect to memory location consequently also leads to a distinction in memory access types, which can also be local or remote and therefore lead to different (non–uniform) memory access times.

[1]The only exception is normally visible at boot time, where one processor takes the role of a master and controls the further initialization of the remaining processors.

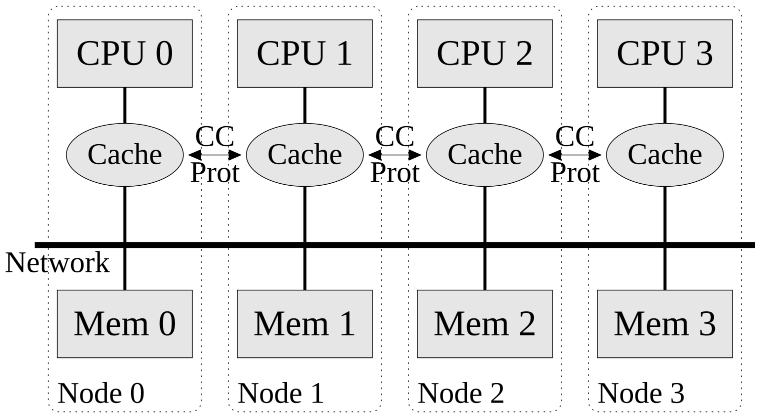


Figure 2.2 Memory organization in CC–NUMA architectures.

This concept of CC–NUMA architectures enables the construction of systems with a larger number of processors since it allows a more flexible packing with individual nodes each with its own memory and replaces the single common bus with a more flexible inter–node interconnection fabric. To further raise the number of processors, CC–NUMA machines often use a hybrid scheme with SMP nodes with a small number of processors sharing a local memory connected by a CC–NUMA interconnection fabric.

Like in the UMA case, a cache coherence protocol (hence the name CC–NUMA) is deployed to maintain the transparency and consistency of the caches in the system. Its implementation, however, is normally much more complex than in a UMA scenario because it has to take the increased latencies for remote accesses and the potentially higher number of involved CPUs and caches into account. This has led to new approaches in the area of cache coherency protocols [175, 86, 202].

The concept of CC–NUMA machines (sometimes referred to as S2MP: Scalable Symmetric MultiProcessing) has gained quite a bit of popularity in the last few years and has led to both research prototypes, like the FLASH architecture [128], and a number of commercial developments which are used for both server and scientific computing applications. Commercial examples include the SGI O2000/O3000™ series [131], which couples Dual SMP nodes to a CC–NUMA architecture, the Exemplar™ [35] machines by Hewlett Packard (formerly by Convex) with its 8–way SMP nodes connected by four independent cross–node links, and the Numa–Liine™ [37] by Data General coupling 4–way SMP nodes based on the PC architecture using a CC–NUMA interconnection fabric. These systems allow processor numbers in the range of 32 to 128 processors and hence significantly exceed the typical number of processors in current SMP systems.

Together with appropriate software mechanisms, present in proprietary operating systems or extensions thereof, which are available on all of these machines, the concept of CC–NUMA architectures, like their UMA counterparts, enables users to directly exploit shared memory programming in a fully coherent global address space. The distributed nature of the underlying memory resources, however, requires the user to take this distribution into account and to optimize with respect to data locality, which leads to additional complexity in using such systems for parallel processing.

2.2.3 Transitions Towards Pure NUMA Systems

A problem of these CC–NUMA architectures is their often quite complex cache coherency protocol. While extremely useful for the user, as it maintains the cache transparency known from single processor systems, it again forms a limit for the scalability of the overall system. This is caused, for one, by the complexity of the protocol which can lead to a performance problem and, secondly, by the requirement of tightly coupling the individual nodes from a logical point of view to allow the maintenence of a global state.

This observation has inspired non–CC–NUMA architectures without a global cache coherency protocol, which will simply be referred to as NUMA architectures in the following. Such systems are capable of supporting a larger number of processors since the only hardware requirement imposed by a NUMA system is the ability to perform loads and stores to remote memory locations. The implementation of this capability requires only very little hardware complexity enabling a simple and cost effective system design. This reduced complexity on the hardware side, however, leads to a higher complexity demand in software as applications have to compensate for the missing hardware coherency. This can, however, even lead to a higher performance compared to CC–NUMA machines, as the coherency control can be adapted to the application needs instead of using a general cache coherency protocol.

Examples for this kind of architecture are the Toronto NUMAchine [70], a research prototype based on a hierarchical interconnection topology, and the T3D/E™ line [235, 120, 124] by Cray Inc. (now part of Tera Computer Company). Especially the latter one has had a significant commercial success. It allows the coupling of up to 2048 processors in a three dimensional torus topology. Due to the programming complexity discussed above, these machines are mostly programmed using message passing libraries like PVM [60] or MPI [152], which have been optimized for this NUMA architecture. A direct exploitation of a pure shared memory programming model is only available on this system in the form of an explicit shared memory programming model using a put/get semantics since such a model is capable of hiding the missing hardware coherence. This can be done because any communication is performed with explicit calls and these points in the execution can then be used to ensure the correct memory coherency behavior.

2.2.4 Shared Memory for Clusters

On the other end of the architectural spectrum with respect to shared memory support are the so–called NORMA (NO Remote Memory Access) architectures which offer no direct hardware support for shared memory programming. Systems in this class of architectures are often built from independent nodes of either single processor or small SMP type systems, as depicted in Figure 2.3. The nodes are then interconnected using a general purpose network like Ethernet. In contrast to the architectures discussed so far, this network is normally not attached to the main system bus, but rather through an I/O bus and a separate

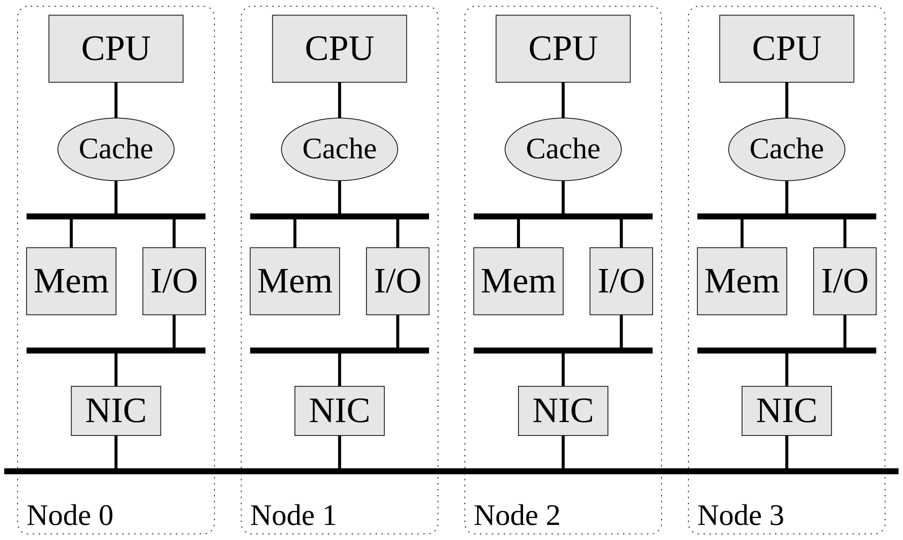


Figure 2.3 Memory organization in clusters or NoWs with NORMA characteristics.

Network Interface Card (NIC). Due to the ability to use standard single–processor or small SMP nodes as the basic building blocks, these machines are often referred to as Network of Workstations (NoWs), Cluster of PCs (CoPs), or Pile of PCs (PoPs) depending on the type of nodes used.

Due to the missing support for shared memory, any communication in this kind of system has to be performed in the form of explicit messages. These messages can then be transfered through the NIC to the network and require an explicit receive on the target node. Consequently, these architectures are by default programmed using pure message passing paradigms mostly in the form of standard libraries, since this directly matches the underlying hardware facilities.

Despite the lack of hardware support for shared memory programming, the advantages of this paradigm have motivated researchers to provide a global memory abstraction purely in software for clusters and NoW architectures. This has led to so–called Distributed Shared Memory (DSM) systems, which either use the page swap mechanisms in modern virtual memory systems or instrumented applications to track user accesses to global data. The information gathered is then forwarded to other nodes using explicit communication mechanisms. Therefore, such approaches allow shared memory programming in NORMA environments despite the missing hardware support and are in principle widely applicable.

Until recently, these approaches have been limited to a certain extent by the network performance available in these systems. However, with the rise of high–speed System Area Networks (SAN), like the Scalable Coherent Interface (SCI) [75, 92], Myrinet [20], ServerNet [82] , or GigaNet [237], this situation has greatly improved. These technologies allow a direct user–level access to the network interface without the usual protocol stack and therefore enable a low latency and high bandwidth communication.

2.3 NUMA for Clusters: The Scalable Coherent Interface

Among these systems, the Scalable Coherent Interface (SCI) [75] has a special role. It not only represents a state-of-the-art System Area Network (SAN), but also allows remote memory accesses. It therefore forms a bridge between cluster architectures and NUMA systems by maintaining the scalability and cost effectiveness of NORMA systems while providing NUMA–like remote memory access capabilities. SCI is therefore well suited for providing both message passing and shared memory programming on clusters. The latter is explored in detail in this work.

2.3.1 History and Principles of SCI

The work on the Scalable Coherent Interface originated in the late 80s from the Futurebus+ project [93] which aimed at the specification of a successor for the Futurebus [90]. During this work, it became evident that the traditional bus–based approach for large–scale multiprocessor architectures would no longer be capable of keeping up with the rising speed of coming generations of microprocessors and would need to be replaced by a more scalable concept. On the other hand, the services provided by bus–based systems were to be preserved in order to keep the familiar environment and to enable an easy interface to existing and future bus systems. This led to the specification of SCI, which was finished in 1991 and became the formal IEEE/ANSI standard 1596 in 1992 [92].

It specifies the hardware interconnect and protocols to connect up to 65536 SCI nodes, including processors, workstations, PCs, bus bridges, and switches, in a high–speed network. SCI nodes are interconnected via point-to-point links in ring–like arrangements or are attached to switches. The logical layer of the SCI specification defines packet–switched communication protocols using a state-of-the-art split transaction scheme. It decouples request and response for a remote operation, each being transfered in a separate packet. This enables every SCI node to overlap several transactions and allows for latencies of accesses to remote memory to be hidden. Optionally, the SCI standard defines a directory–based cache coherence protocol which enables the construction of CC–NUMA architectures.

The Scalable Coherent Interface specification, however, does not represent the end of the road. Its technological principles have been used on other successor projects. The most visible result is the Serial–Plus project (IEEE P2100), which was formerly known as Serial Express or IEEE 1394.2. Its intent is to provide a scalable extension for IEEE 1394 [94] installations, also known as Firewire™ , an interconnection technology used in various consumer products allowing a fast and easy data transfer between digital imaging equipment like digital cameras.

2.3.2 Shared Memory Support in SCI

The central feature of SCI, which distinguishes it from any other mainstream cluster–oriented interconnection fabric, stems directly from the original starting point of SCI, the direct replacement of buses while maintaining bus–like services. It contains the ability

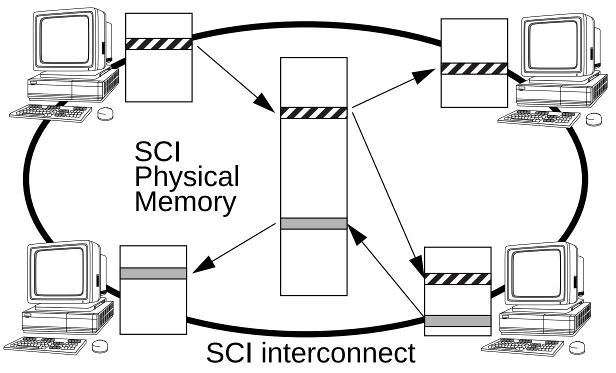


Figure 2.4 SCI HW–DSM using a global address space (from [74]).

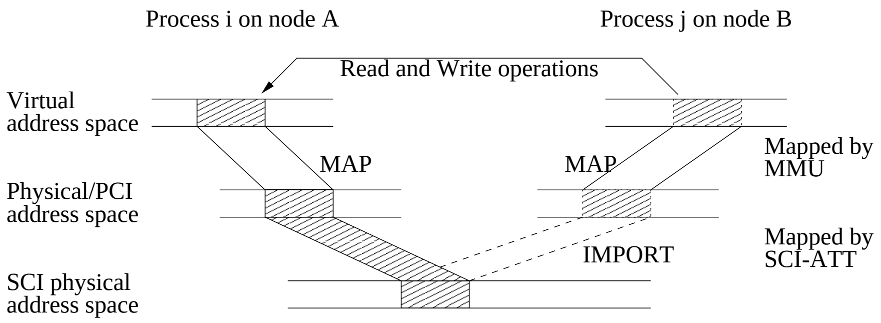


Figure 2.5 The various mapping levels in PC based SCI systems.

to support remote memory accesses — both reads and writes — directly in hardware by providing a Hardware Distributed Shared Memory (HW–DSM). For this purpose, SCI establishes a global physical address space which allows to address any physical memory location throughout the system. Any address within this SCI physical address space has a length of 64 bit, with the higher 16 bit specifying the node on which the addressed physical storage is located and with the lower 48 bit specifying the local physical address within the memory on that particular node.

Nodes can access this global physical address space and hence any physical memory location within the whole system by mapping parts or segments of this memory space into their own memory. Once such a mapping has been established, users can issue standard read and write operations to those mapped segments and these are then translated by the SCI hardware into accesses within the global address space and forwarded to the intended remote memory. A simplified picture illustrating this concept is shown in Figure 2.4. It shows four PC nodes and the SCI physical memory together with a few example mappings.

While the SCI standard defines this basic principle and also specifies the protocol used to transfer the data across the network, it purposely omits any specification regarding the actual remote memory mapping process needed to access the SCI physical address space. This process is very system and architecture dependent and needs to be adjusted to each platform. On PC–based cluster architectures, which form the focus of this work, the concrete implementation of SCI is more or less dictated by the I/O subsystem available in current PCs. Therefore, SCI adapters intended for PC architectures generally come in the form of PCI adapter cards and implement a PCI–SCI bridge.

The concrete mapping process in such a PCI–based SCI system is shown in Figure 2.5. Any physical memory location within the system is potentially available within the whole cluster since it can be addressed through the global SCI physical memory. Parts of this address space can then be mapped into the PCI or I/O space controlled by the PCI–SCI bridge on a particular node (in the figure on the right side) using a mapping table on the SCI adapters, the so–called Address Translation Table (ATT). Each entry in this table is responsible for a given part of the whole I/O space and allows the specification of (almost) arbitrary parts of the SCI physical memory to be visible within this part of the I/O space. The granularity of these mappings is dictated by the PCI–SCI adapter and ranges, depending on the adapter type and configuration between 512 Kbyte and 4 Kbyte. Due to constraints with regard to the number of available ATT entries and the total amount of mappable memory, however, for newer generation adapter cards this granularity is typically set to values around 64 Kbyte and is hence significantly larger than the page granularity.

In order to give user processes access to these mapped memory resources, a second level of mappings into the virtual address space of the target processes needs to be performed. This mapping is purely local and does not affect the SCI implementation. In the PC architecture the I/O address space, including the part used by the SCI adapter for remote memory mappings, is an extension of the physical address space used for the local memory. Hence, the appropriate parts containing the remote memory are mapped into a virtual address space in the same manner as local memory by using the processor's Memory Management Unit (MMU).

Once both of these mapping levels have been established, processes can access the mapped virtual memory regions with standard read and write operations. On nodes on which physical memory is locally available (left node in the figure) these accesses are executed in a conventional manner, while on nodes on which the physical memory has been mapped via SCI (right node in the figure) the memory access will result in an access to the I/O space, from where the SCI adapter forwards it to the remote node according to the appropriate ATT entry. The process issuing the memory access, however, does not see the difference between local and remote accesses except for their latency, thereby resulting in the intended transparent NUMA–like shared memory behavior of SCI.

This SCI shared memory support has two major shortcomings: for one, the global address space provided by SCI is solely based on physical addresses since only locations in the physical memories of any node can be addressed. This limitation is due to the fact that within the PC architecture no I/O bus exists, which directly uses virtual addresses (in contrast to e.g. the Sparc Architecture with its SBus system [91]). As a consequence, no physical page can be accessed safely, which is not pinned into memory, as it could otherwise be evicted by the operating system at any time. Consequently, the SCI shared memory support can not directly be used to implement a global virtual address space as it is required by shared memory programming models. The user rather has to rely on the sharing of separate pinned memory segments which do not provide the desired transparency since they have to be explicitly allocated and accessed.

A second shortcoming in PC–based SCI systems is the missing cache coherency between the individual nodes. Even though the SCI standard [92] defines a versatile cache coherency protocol, it can not be used together with PCI–SCI bridges since the PCI bus [253] does not allow the snooping of memory transactions. As a result, current PCI–SCI implementations disable any caching of remote memory locations in order to avoid cache inconsistencies across nodes. This, however, has a severely negative impact on the read performance.

2.3.3 Commercial Success of SCI

SCI is currently used in a large variety of scenarios and products. Virtually all of them are based on the technology of a single SCI manufacturer, Dolphin Interconnect Solutions (ICS) [239], despite some efforts by the company Interconnect Systems Solutions (ISS) [241] to provide an alternative source for SCI–based technology. After some initial success, however, these efforts seem to have been abandoned.

The core for any SCI–based system is the so called Link Controller (LC) chip, which currently exists in its fifth generation. Its speed has been steadily increased over the years and has reached 800 MB/s on each link in the latest generation, the LC 3 [9]. It therefore provides a level of performance already very close to 1 GB/s, which was the original goal during the SCI standardization process. Based on this component, Dolphin offers several SCI bridges to various other system busses, including the SUN SBus [91], the PCI bus [253], and PMC [95]. In addition, Dolphin also provides several types of switches which allow the creation of larger systems.

Besides these products from Dolphin, several other companies either use or benefit from Dolphin's SCI technology and offer further SCI–based products. The small Norwegian company SciLab [250] offers a sophisticated SCI tracing and trace analysis tool called SCIview [208], Siemens has developed an optical SCI link called Paroli (now marketed by Dolphins ICS) capable of bridging larger distances than the standard copper based links, and Lockheed Martin has designed a large scale SCI switch.

Additionally, several OEM vendors use the SCI chips and adapters provided by Dolphin to implement their own systems and deliver these as turn–key solutions. SUN Microsystems uses SBus or PCI adapters for SUN Enterprise™ clusters [157] to enable a high–performance fault tolerance solution and Siemens integrates SCI bridges into their R600™ mainframes [244] for clustering and enhanced I/O aiming at the parallel database market. While these systems use SCI without the optional cache coherency protocol included in the standard, other vendors have used this option and have created SCI–based CC–NUMA machines. Data General successfully distributes its NUMA–Liine™ [37], which is based on commodity multiprocessor PC boards with a special access for SCI to the memory bus to enable the maintenance of cache coherency, a concept also used by IBM (formerly Sequent) for their NUMA–Q™ systems [89]. Also, the (now obsolete) Exemplar™ system, introduced by HP/Convex, uses SCI links based on Dolphin hardware to connect its SMP

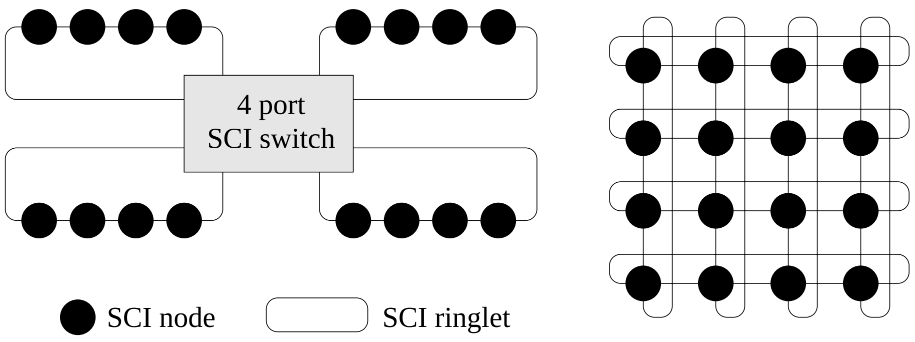


Figure 2.6 Switched SCI ringlets (left) vs. two dimensional torus (right).

hypernodes to a full CC–NUMA system [35].

The most common use of SCI, however, which will also be solely considered for the remainder of this work, is the use of SCI in commodity PC clusters based on the PCI–SCI bridges or adapters [138] directly developed and marketed by Dolphin. Along with the continued development of the SCI link controllers, these boards have also lived through a number of generations. During the timespan of the project presented in this work, the evolution has gone from the D308, still with LC 1 and for 32–bit/33 MHz PCI busses, over the D310, with the improved LC 2 chip, to the D320, the first 64–bit PCI card. Just recently, but not in time to be used for this work, the line of adapters was extended to the D330 series, carrying the new LC 3 and capable of using 64–bit/66 MHz PCI busses. These PCI adapters allow very high–performance end–to–end communication with latencies under $2\mu s$ and a bandwidth of up to 85 MB/s with standard 32–bit/33 MHz PCI busses and up to 250 MB/S (DMA communication) with new 64–bit/66 MHz PCI systems.

By default, all of these adapters can be connected in unidirectional ringlets, the basic topology of all SCI–based systems. Since this approach, however, can only be applied up to a certain limited number of nodes before a saturation of the network is reached (typically at 8–10 machines, depending on the load generated by the nodes), additional switches are required to allow the connection of multiple independent SCI ringlets. For this purpose, two different approaches are available in today's systems, resulting in two fundamentally different topologies (shown in Figure 2.6). These are the use of traditional switches connecting ringlets of several nodes and the creation of multidimensional tori using small interdimensional switches within each node.

The first option can be realized by using external switching components from Dolphin. The current generation of switches, in form of the D515 switch [15], is based on the LC 2 and contains 4 ports per switch plus 2 extension ports. Using the latter ones, the switch can either be expanded to a stacked switch with up to 16 ports or configured to a non–expandable 6 port switch. Each port can be connected to a ringlet with arbitrary numbers of nodes, but again the bandwidth limitations of ringlets apply. Internally, the D515 is based on multiple busses delivering the required cross–section bandwidth.

The next generation of switches, in form of the already announced D535 [142], will change this design to a full crossbar, which is expected to result in a greater overall performance and better scalability. This switch, which will be based on the LC–3, can also be expanded to much larger configurations than the D515 switch enabling configurations with more than 128 ports.

The second option is the creation of multidimensional tori by using small SCI ringlets in all dimensions [23]. In such a configuration, each node/adapter will be connected to each dimension and uses a small switch integrated into the SCI adapter to provide cross–dimensional packet transmissions. Technically, this solution is realized with separate daughter boards which are plugged into the base PCI–SCI bridge and provide additional SCI link controllers. Currently up to three dimensional tori with up to 10–12 nodes in each dimension can be created, resulting in potentially very large systems.

This latter option has been developed together with Scali AS [249], a Norwegian company specialized in “Affordable Supercomputing”, i.e. the construction of turn–key solutions for high–performance computing based solely on commodity components. Together with the appropriate software infrastructure, the so–called Scali Software Platform (SSP) [190], which includes both high–performance communication support in the form of the optimized MPI implementation ScaMPI [85, 84] and comprehensive system management and monitoring support, these systems have found a wide use in both academia and industry. The concept has also been adopted by Siemens and is now marketed as the Siemens HPC–Line^{TM} [69].

2.3.4 Other SCI–based Research Activities

Along with these commercial developments, a significant amount of research has been done worldwide focusing on SCI. This has led to a very active research community with participants from many different areas of science and has manifested itself in a comprehensive book published about the current state of SCI [75], the establishment of a conference series since 1998 [181, 106, 81], and an international working group supported by the ESPRIT programme of the European Union [245] as well as several different ESPRIT projects [240].

Early work has concentrated mostly on the IEEE standardization efforts of the SCI protocol which has led to the ANSI/IEEE Standard 1596–1992 [92] and on SCI’s low–level performance as it has been done e.g. at University of Oslo within the SALMON project [170], at SINTEF [80], and at the University of Wisconsin [203]. With the rise of other high performance system area networks (SANs), additional studies e.g. at the Medical University of Lübeck [177], the ETH Zürich [127], and the University, California at Santa Barbara [87] have aimed at comparing and contrasting their performance with SCI.

These performance oriented efforts have then been complemented by intensive efforts to form a suitable software environment for the efficient use of SCI–based clusters. As a first step in this direction, several approaches have targeted the design of suitable low–level APIs which led to several competing approaches: an IEEE standardization effort headed by the physics community entitled PHYSAPI [140, 141], a widely portable interface definition [64] developed within the SISCI ESPRIT project [207, 50] and adapted by Dolphin as their primary API for all platforms, and the driver interface used by Scali systems for efficient MPI support [185]. All three provide approximately the same functionality, i.e. the creation and mapping of SCI shared memory segments and their competing existence has led to a significant amount of confusion in the SCI community. By now this situation has significantly improved as the SISCI API has established itself as the main interface for SCI due to its wide availability and direct support by Dolphin. The PHYSAPI approach, on the other hand, seems to have been abandoned and for Scali–based systems, a SISCI compatible emulation is now available, allowing the execution of SISCI–based codes without changes.

Low–level APIs alone, however, only provide a basic abstraction of SCI's hardware capabilities and are not directly suited as the basis for future application developments. This has led to a thorough investigation of the various issues involved in creating high–performance communication libraries [186, 51, 61, 88, 84]. These studies illustrated the use of direct user–level access to the network adapter to eliminate the protocol stack overhead and the efficient use of the special hardware DSM mechanisms for improved communication performance. The result is a broad range of SCI communication libraries. Examples are ports of the Active Message specification [225] done at the Technische Universität München [49] and the University of California at Santa Barbara [87]. In addition, various fast socket implementations at different levels of the operating system have been undertaken: underneath the TCP/IP stack as a general network driver from the University of Paderborn [218], replacing the TCP/IP stack but still in kernel mode ensuring full protection from the University of Copenhagen [72], and at user level exploiting the full performance benefits of SCI from the Technische Universität München [76].

In addition, higher–level message passing layers, which are widely used, have been targeted in order to provide a wide support for the transition of existing applications to SCI–based clusters. As part of these efforts, SCI–optimized versions of PVM [60] have been implemented and evaluated at the University of Paderborn [55] and the Technische Universität München [52], and MPI [152] has been adopted to SCI by the RWTH–Aachen [234], the Technische Universität Chemnitz [68], and by Scali AS [249] within their previously mentioned ScaMPI system [85]. Also a port of the CORBA architecture has been undertaken (by INRIA [178]) enabling the convenient use of SCI in coarse–grain object–oriented environments.

Several projects also target the use of shared memory programming for SCI–based cluster systems. Examples for this are the Madeleine / PM2 environment [10, 8] developed at the ENS Lyon, the NOA system from INRIA [151], a DSM system implemented at Lund University [114], the HPPC–SEA DVSM library [40] from the Politecnico di Torino, and the SVMlib [174] from the RWTH Aachen. These approaches, however, only exploit SCI as a fast interconnection system by implementing traditional Distributed Shared Memory (DSM) systems (see also Chapter 4.1). This omits SCI's potential to directly support shared memory in hardware and hence fails to fully exploit the advantages of this interconnection technology.

Despite this, only very few projects follow this path since it is connected with complex implementation and system integration issues. The SciOS system [122, 123], jointly implemented by the University of Copenhagen and INRIA, depends mainly on traditional SW–DSM mechanisms but enables the use of direct SCI mappings in scenarios where this is appropriate. The concrete type of implementation used for a particular page, however, is fully hidden from the user through a common interface which is designed after the System V shared memory system. The Yasmin system [25, 180] from the University of Paderborn, on the other hand, provides a simple global memory abstraction directly relying on the shared memory facilities of SCI, but on the basis of coarse grain segments rather than at page level. Additionally, it does not include any shared memory optimizations such as the ability to cache remote memory. These deficiencies can cause some severe performance problems, as shown later in this work.

The SCI Virtual Memory (SCI-VM) system [198, 193, 195], which forms the core for the implementation discussed in this work and will be explained later in detail, solves these deficiencies and therefore represents the first approach which directly exploits the hardware DSM features of SCI in an efficient and usable way. In addition, the SCI-VM is not a monolithic system but is rather integrated in the context of a larger, highly adaptable framework for shared memory programming, the HAMSTER system[145]. This system will also be introduced in much more detail throughout the remainder of this thesis.

Most of the projects mentioned above, however, investigate only one side of the overall picture, i.e. only message passing or shared memory programming models. Only few of them attempt a full integration of both paradigms into a single environment: the SMiLE project [109, 74] at the Technische Universität München, which forms the greater context of this work and is explained in more detail in Section 2.3.5, the SISCI project [207, 50], an ESPRIT project combining the efforts from several partners from both the middleware and the application area, and the SciOS/SciFS system developed in a joint project between the University of Copenhagen and INRIA [123, 72] providing both System–V shared memory and socket communication.

With the software environments for SCI–based clusters maturing, SCI has started to move more into the industrial field and is now used in a variety of different real–world application scenarios. Examples for these, which have been done in cooperation with research institutions and are therefore publicly known, are the deployment of SCI–based clusters in clinics for computationally intensive algorithms in nuclear imaging procedures [113, 200] (conducted at the Technische Universität München, partly within the ESPRIT project NEPHEW [242]), the use for historic film restoration (implemented within NEPHEW as well as within another ESPRIT project called DIAMANT [246]), the port of the commercial state-of-the-art fluid dynamics code TfC/CfX [252] (AEA technologies [247] in cooperation with the SISCI project [207, 50]), the use of SCI–based systems for a Synthetic Aperture Radar (SAR) processing application [24] (realized by Scali [249]), a shared memory–based implementation of the GROSMOS 96 Molecular Dynamics Code (from the RWTH–Aachen [45]), and the exploitation of SCI–based high–performance communication for the optimization of electric fields for high voltage transformers [222] (a joint project between the Technische Universität München and ABB Research in Heidelberg). This list is, however, by far not complete and is only a brief compilation of a few interesting examples of known projects.

Besides these industrially relevant codes, large scale SCI–based applications can also be found in basic research for particle physics. In this area, large scale experiments are designed which deliver a huge amount of raw data. This data then needs to be received, processed, filtered, and stored, requiring high–performance network solutions. Currently many different interconnection technologies are investigated including the Scalable Coherent Interface. Work in this direction is conducted at the University of Heidelberg targeting a custom solution for a SCI–based data processing farm [227] as well as at the European Laboratory for Nuclear Research (CERN) and the Rutherford Appleton Laboratories (RAL) utilizing mainly commodity components [14, 13].

Next to these software related projects, a few academic projects also deal with developments on the hardware side. These efforts can be roughly split into two main streams: the development of SCI adapter cards and the design and implementation of hardware tracer and monitoring equipment. Work in the former category aims at implementing adapter cards with special features not available in main stream SCI adapters and for providing access to internals for further research. The first is pursued by the Technische Universität Chemnitz for the investigation of user–level DMA in a VIA [36] manner [221], while the latter drives the development of the SMiLE adapter card at the Technische Universität München [2, 1].

The SMiLE card is also used as the basis for the development of a hardware monitor [107, 79, 78]. This monitor facilitates the observation of the network traffic with little intrusion overhead and the generation of memory access histograms across the complete SCI physical address space. The information acquired using this monitor can then be used to optimize applications [108].

In addition to this performance aspect of monitoring, the SMiLE hardware monitor has also been designed to support deterministic debugging [79]. This is enabled through a feedback line from the monitor to the SMiLE adapter which allows, upon recognition of certain memory access patterns or events, to hold packets in the buffer RAM. This can be used to ensure the repetition of memory access patterns in consecutive debug runs and hence a clean debugging process.

A second hardware monitoring approach is being undertaken at the Trinity College Dublin [147]. In contrast to the SMiLE monitor mentioned above, this approach enables the gathering of complete traces in large on board memories, which can then be used for a post mortem, off–line analysis. For this purpose, the traces are transfered into a relational database system enabling complex queries across a full trace.

While all research discussed so far deals with real systems, several groups have also based their work on comprehensive simulations of SCI systems. This approach allows more flexibility with regard to the concrete target system and SCI implementation and hence enables the investigation of new or enhanced features. The University of Wisconsin has implemented a simulation environment with the goal to investigate SCI–based systems with thousands of processors, the so–called Kiloprocessor systems [115]. The Polytechnical University of Valencia implemented a VHDL–level simulation allowing the evaluation of new hardware features [204] and at the University of Oslo an OPNET–based [159] simulation environment [182] has been developed which aims at an accurate modeling of SCI–

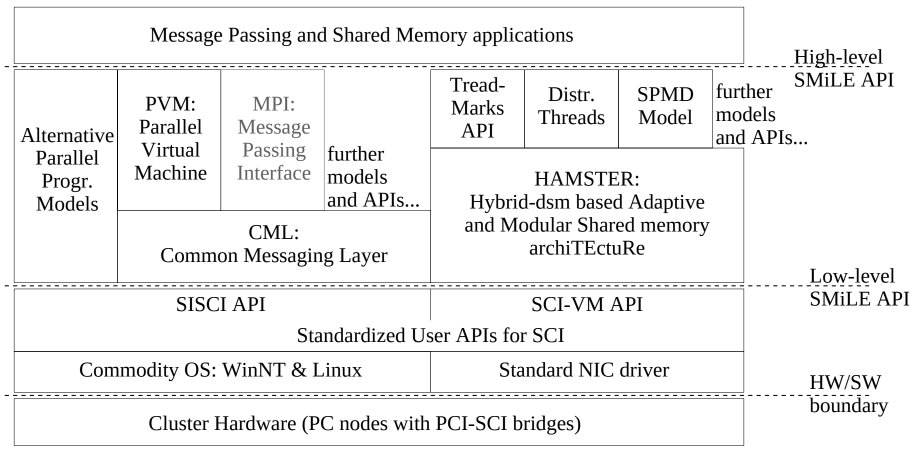

Message Passing and Shared Memory applications
High-level SMiLE API
Alternative Parallel Progr. Models
PVM: Parallel Virtual Machine
MPI: Message Passing Interface
further models and APIs...
Tread-Marks API
Distr. Threads
SPMD Model
further models and APIs...
CML: Common Messaging Layer
HAMSTER: Hybrid-dsm based Adaptive and Modular Shared memory archiTEctuRe
Low-level SMiLE API
SISCI API
SCI-VM API
Standardized User APIs for SCI
Commodity OS: WinNT & Linux
Standard NIC driver
HW/SW boundary
Cluster Hardware (PC nodes with PCI-SCI bridges)

Figure 2.7 Overview of the SMiLE software infrastructure.

based systems for performance prediction. In addition, the latter system has also been used to study the impact of improved flow control as well as new fault tolerant routing schemes for SCI networks [183].

2.3.5 HAMSTER as Part of the SMiLE Project

As already mentioned above, the HAMSTER project presented in this work is carried out in the larger context of a clustering project at LRR-TUM[2] with the name *Shared Memory in a LAN-like Environment* (SMiLE), which reminds of the hardware DSM capabilities of SCI. This base mechanism is used within the project to implement highly efficient programming abstractions adhering to many different programming paradigms.

The SMiLE project broadly investigates clusters based on this interconnection technology using both hardware and software approaches. This led to the design and the implementation of an own PCI–SCI adapter [2] and a hardware monitor for the on–line observation and evaluation of memory access patterns across the HW–DSM [111, 217]. In the area of software the SMiLE efforts have led to the development of a comprehensive software infrastructure (shown in Figure 2.7) featuring programming models from both paradigms, message passing and shared memory. In addition, other less widely used paradigms have also been investigated including the graphical representation of dataflow and communication by PeakWare [112], a product of the French company *Sycomore* [243], and a multithreaded scheduling environment (MuSE) [133, 134] based on the principles of dataflow and task stealing.

The main cores of the SMiLE software efforts, however, have to be seen in the area of the two predominant parallel programming paradigms: message passing and shared memory. The first one can be implemented efficiently and in a straightforward way using the given HW–DSM capabilities. It is represented in SMiLE with various low–level user-level

[2]Lehrstuhl für Rechnertechnik und Rechnerorganisation at the Technische Universität München

message passing layers [51] including the implementation of the Active Message concept [225], a fast socket library omitting the expensive TCP/IP stack [229], and a general messaging layer, called the Common Messaging Layer or CML [51]. On the high level side, it is completed by a port of the Parallel Virtual Machine (PVM) [60] delivering the performance benefits of the CML to PVM–based applications [52].

The efforts in the area of shared memory with the establishment of a general, programming model independent shared memory concept [145] are presented in this work and will be described in full detail in the following chapters. It consists of a distributed shared memory core, the SCI Virtual Memory or SCI-VM [193, 198] (see Chapter 4), which establishes a global virtual memory abstraction across cluster nodes and a set of management modules. These can then be used for the implementation of a large number of different programming models allowing the easy adaptation of the overall system to new requirements and existing applications which guarantees a high degree of flexibility.

In summary, the SMiLE efforts allow the efficient exploitation of the clusters interconnected by SCI using a large variety of different programming models and paradigms. This allows both the easy porting of existing codes and the efficient and flexible development of new applications and hence represents an important step towards a full exploitation of SCI–based clusters.

2.4 Summary

The discussion of shared memory programming in this Chapter has shown the large diversity of programming models and the problems induced by it. The portability of shared memory codes is severely limited as ports from one programming model to another are frequently necessary and the learning curve for programmers is increased as they have to be familiar with several different models. Also from an architectural point of view, the shared memory paradigm is associated with problems since shared memory programming models are only supported in hardware on tightly coupled systems with a limited scalability and software emulations on more scalable loosely coupled architectures, like clusters, have proven to be associated with performance problems.

Nevertheless, the advantages of the shared memory paradigm, in terms of programmability and easier parallelization, remain and justify the further examination of their adoption for loosely coupled architectures. In order to avoid the performance problems mentioned above, however, some hardware support should be used for the creation of a global virtual memory. An example for systems with such properties are NUMA systems, which allow every node to access remote memory on any other node directly and without the need for any software involvement. However, this occurs without the enforcement of cache coherency. On the other side, NUMA systems maintain many of the advantages of traditional loosely coupled architectures, such as good scaling properties and a potentially good price/performance ratio. NUMA architectures therefore represent a good tradeoff between the need to support shared memory in hardware and to provide sufficient scalability at acceptable cost, making them worthwhile to be studied in more detail.

The Scalable Coherent Interface (SCI) [75], an IEEE standardized interconnection fabric [92], allows the creation of NUMA systems based on clusters of commodity PCs. Using this technology, the work presented here establishes a comprehensive shared memory programming framework called HAMSTER (Hybrid–dsm based Adaptive and Shared memory archiTEctuRe) which creates a transparent and programming model independent global resource abstraction. It therefore allows the efficient use of shared memory applications on NUMA–based architectures and hence forms a bridge between CC–NUMA architectures and traditional clusters.

Chapter 3

HAMSTER: Hybrid–dsm based Adaptive and Modular Shared memory archiTEctuRe

As can be seen from the discussion in the previous chapter, shared memory programming on top of loosely coupled architectures still faces some major obstacles. The two main problems are the abundance of available programming models without a dominating standard in sight and the existing efficiency problems inherently associated with traditional software approaches. The HAMSTER system, the core of this work, approaches both of these problems in a novel way. HAMSTER, which stands for Hybrid–dsm based Adaptive and Modular Shared memory archiTEctuRe, allows users to implement a large variety of different programming models on top of a single efficient DSM core. This core is designed and implemented for NUMA architectures with hardware DSM capabilities, but without the need to provide either global memory/cache coherence or a specialized global operating system. This ensures that the system remains also applicable to loosely coupled NUMA architectures, like clusters, and therefore profits from their scalability.

On top of this global memory abstraction, the HAMSTER framework then offers a large number of shared memory services available to the user to implement any kind of shared memory programming model. This can include both existing models to allow the direct porting of existing codes and new application field specific models to ease the implementation of new applications. This system, with its flexibility with regard to programming models, therefore allows true shared memory programming on scalable NUMA architectures to directly benefit from their hardware support for remote memory accesses.

3.1 Related Work

A system like HAMSTER with the clear intent of supporting any kind of parallel programming model of a certain class on top of a single core has, to our best knowledge, not been previously attempted. This may stem from the fact that such an approach seems only beneficial in the area of shared memory programming models, as its motivation is taken from the observation that presently no clear standard for shared memory programming exists which is suited for all kind of architectures and application domains[1]. In other areas, the situation is different with one or a few standards being clearly accepted, like PVM [60] and MPI [152] in the area of message passing.

However, even though HAMSTER presents a novel approach as far as the whole system is concerned, its individual components are connected to a significant amount of related work from many different areas. This reaches from the previously mentioned Software Distributed Shared Memory (SW–DSM) systems, to the large amount of existing shared memory programming models, from the architectural support in NUMA–like system to operating system issues connected to form a global resource abstraction, and from issues related to distributed multithreading to global performance monitoring and load balancing. These issues are discussed in the various “State of the Art” sections in this work included in the chapters of each subcomponent.

3.2 HAMSTER Design Goals

Before going into the discussion of the system’s details, it is necessary to discuss the main goals governing the design of the HAMSTER system. This will further illustrate the purpose and intended use of the system and will set the stage for the extensive coverage of the system’s implementation in the following chapters.

3.2.1 Main Characteristics

The main idea behind the HAMSTER system is the possibility to allow users to implement as many different programming models as possible or necessary with a minimum of complexity on top of a single efficient core. In order to achieve this goal, the design of HAMSTER needs to exhibit a few key characteristics discussed in the following sections.

Flexibility

One of the prime high–level design goals for such a system is flexibility. Since its intention is to form the framework for large numbers of different, not necessarily a priori known, programming models, it needs to be able to adapt to varying prerequisites and requirements. This is important not only with respect to the services provided by HAMSTER to build programming models, but also with respect to the surrounding environment, which can greatly change depending on the target applications.

Multi–operating system support

As part of this flexibility, it is also advantageous for such a system to support multiple operating systems. This increases the available number of target platforms and therefore

[1]It should be mentioned that there are some standardization approaches also for shared memory, most prominently OpenMP [172] and HPF [236]. However their focus is either on specific application domains or special architectural subclasses (mostly on UMA machines).

further eases the portability of applications. In addition, it also allows the porting of various programming models between the supported operating systems at no extra cost, thereby further widening the total application platform available on top of the system.

No bias towards any particular model

In order to fulfill the goal of enabling the implementation of almost any shared memory programming model on top of the target system, it needs to be designed without any restriction for, or bias towards, a particular programming model. This would either limit the number of possible target programming models or increase the complexity of such implementations. One of the most critical points in this direction is the task or execution model of programming models, as these can vary greatly between both programming models and operating systems.

The fulfillment of this goal leads to a non–monolithic system without strong ties between the individual functionalities exposed by the system. It has more the character of a large tool box of services available at the digression of the user, but without interfering with each other. This guarantees a maximum of freedom in the implementation of new programming models and does not favor any particular one.

Easy retargetability

In addition to the already discussed flexibility to implement a large number of shared memory programming models, it is also necessary to guarantee a low complexity for such implementations. It has to be easy to retarget the system to a new programming model or API. Only this will allow users to have several different programming models on top of HAMSTER concurrently and hence to fully exploit the strongholds of the overall system. This will make it worthwhile to use application field specific APIs or programming environments for a small number of applications.

This goal can be achieved by providing a large number of services for different scenarios with many options and parameters. By specializing these services, a large percentage of the intended target programming model can then be designed in a way that leaves only very limited functionality to be implemented from scratch for each new model. The total effort for a new programming model will therefore be low making the creation of many concurrent programming models feasible.

Efficient use of available hardware resources

So far the guidelines have only covered the aspects of multi–programming model support and easy programmability. It is, however, equally important to provide each programming model implemented in such a system with a maximum efficiency by exploiting the available hardware resources as directly as possible. In the context of this work, which focuses on the use of clusters of PCs with a NUMA–like interconnection technology, this translates to the efficient and direct exploitation of such a NUMA system for the creation of a global virtual memory abstraction. Any memory access should be executed directly in hardware either through the local memory system or by the NUMA network without any further software intervention. This eliminates the typical software overhead associated with network protocols allowing for a maximum of performance.

Besides the efficient exploitation of the underlying hardware in order to achieve an efficient global virtual memory abstraction, any other mechanism needed for the implementation of shared memory programming models should be implemented as close to the underlying architecture as possible. Of special importance hereby are the synchronization and consistency enforcing mechanisms, as these are frequently used during the execution of shared memory programs. Therefore, their efficiency has a large potential impact on the overall system performance.

3.2.2 Target Architectures

As previously discussed in Chapter 2, this work targets the rising class of NUMA architectures since they provide a good trade–off between hardware support for easy programming and efficient communication in coordination with good scalability properties. More specifically, this work concentrates on SCI connected clusters of PCs because this specific choice adds the ability to construct such a system purely from easily available commodity–off–the–shelf (COTS) components at an optimal price/performance ratio.

The SMiLE Cluster

The work within the HAMSTER project has been done on the SCI–based clusters established within the SMiLE project. It also serves as the experimental platform for most of the experiments described in the following section (unless otherwise noted) and will therefore be introduced in the following in detail.

At the moment, this cluster consists of six identical nodes, even though certain experiments may only use a subset. All nodes are based on dual Xeon™ processor nodes (450 MHz) interconnected using the commercial PCI–SCI adapters available from Dolphin ICS. The exact configuration of the individual nodes is shown in Table 3.1. Each node is installed with Windows NT™ 4.0 (SP5) and SuSE Linux (kernel version 2.2.5). The topology of the SCI network used within the SMiLE cluster setup can be varied between a single ringlet and a fully switched version. Both options are depicted in Figure 3.1. While the first one is realized simply based on a single PCI–SCI adapter per node with no extra equipment necessary, the second option requires an SCI switch. For this purpose, the six port switch D515 from Dolphin ICS is used.

Both options have their distinct advantages and disadvantages: while the ringlet version is simple to build and helps to cut cost in deploying SCI, its bandwidth is limited as all nodes share the same ringlet link. In addition, this setup is not fault tolerant since a single link failure breaks the ring and therefore disconnects all nodes. Both of these drawbacks are resolved in the switched system design; each node works on its separate link allowing

[2]The D320 is the latest of the cards used in the current system and is by default used in the following experiments. Some older experiments, however, have been done using other adapters. This is then specifically marked in the discussion of the respective experiments.

Component	Description
Processor	Intel Xeon™ 450 MHz
Number of CPUs per node	2
L1 data cache size (line size)	16 Kbyte (32 bytes)
L2 data cache size (line size)	512 Kbyte (32 bytes)
Main memory	512 Mbyte (ECC)
Disk subsystem	SCSI–II & IDE
Disk capacity	2 GB & 10-30 GB
Control network	Intel EtherExpress Pro (Fast Ethernet)
System area network	PCI–SCI adapter D320[2]

Table 3.1 Configuration of the SMiLE cluster nodes.

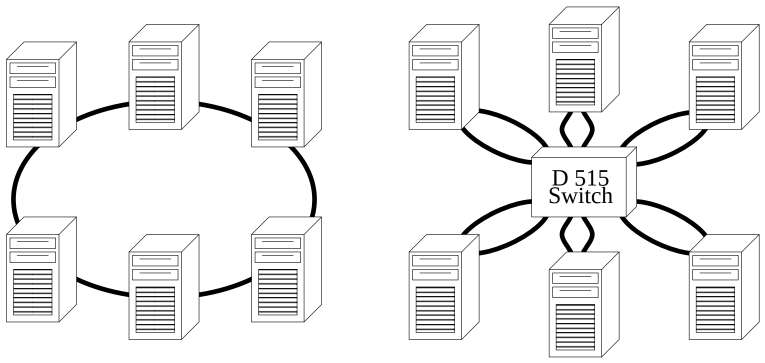


Figure 3.1 SCI topologies used for the SMiLE cluster: ringlet (left) or switched (right).

it to exploit the full bandwidth capabilities of SCI and providing simple fault tolerance on the connection level. On the downside, however, the switch solution has proven to be less reliable due to hardware problems and adds extra latency to any transmitted packet as it passes through the switch [66]. For this reason, and because the ringlet setup has been available longer, all following experiments have been conducted using the ringlet topology.

Portability to Other Architectures

Even though this work has been done within the SMiLE project and therefore implemented for and on top of SCI–based clusters, the concepts presented in this work are not restricted to this specific technology. It presents a general concept on how to use the capabilities and core mechanisms of general (non necessarily cache–coherent) NUMA architectures to efficiently support true shared memory programming on such architectures. With these properties, the proposed system represents a general approach for a NUMA–based global shared memory programming environment and will therefore (hopefully) also find its application beyond SCI.

3.3 The HAMSTER Layers

Based on the guidelines set forth above, the HAMSTER framework has been designed. Figure 3.2 gives an overview of the complete system. It is built on top of a cluster of commodity–off–the–shelf PC hardware running both Linux and Windows NT™ . The cluster is interconnected using SCI [75, 92] which provides high bandwidth and low latency communication in combination with remote memory access capabilities exhibited in the form of HardWare DSM (HW–DSM). This hardware capability, which creates a NUMA architecture, in combination with an additional software component responsible for the memory setup and control, can be used to create an efficient hybrid hardware/software global memory abstraction with NUMA characteristics, the SCI Virtual Memory or SCI-VM [195, 193]. This technique is very similar to the VI approach [36] used in the message passing world, as all communication can be handled in hardware without any OS or protocol overhead.

With this DSM system as the basis, HAMSTER provides several independent modules for the core services needed by shared memory programming models. Most significant are the modules for memory, synchronization, and consistency management. The first one provides mechanisms for a controlled memory allocation and includes both the option to transparently distribute the memory and to control the memory placement at allocation time. This flexibility can be used to migrate SMP applications to a HAMSTER–based system using the transparent memory allocation followed by an incremental memory layout optimization using distribution hints.

The modules for coherency and synchronization management [196] provide all services required to implement a coherent and race–free application. This includes mechanisms to control the relaxed memory consistency of the SCI-VM and typical shared memory synchronization constructs like locks and barriers. In order to allow the greatest possible flexibility, these two modules are kept orthogonal to each other, i.e. synchronization does not include a consistency guarantee and vice versa. The decision on how to combine these two modules is left to the programming model implementation and can be tailored to its specific needs.

In addition to these pure shared memory service modules, a separate module is responsible for the task management, i.e. the distributed creation of activities and their control. This also includes maintaining a global activity counter as well as the controlled termination of an application. The latter is especially important for the implementation of dynamic task models as they are present in multithreading environments.

Due to the inherent connection between the concepts of virtual memory and processes, the creation of programming models with a global virtual memory also requires some form of global process abstraction. In the HAMSTER system, this is realized with the help of a cluster control component that provides a global execution environment. It's tasks include the cluster configuration (i.e. the identification of participating nodes) and the creation and management of a global node name space. In addition, the cluster control component also provides a simple message passing mechanism for control messages, an instrument useful for almost any implementation of a programming model.

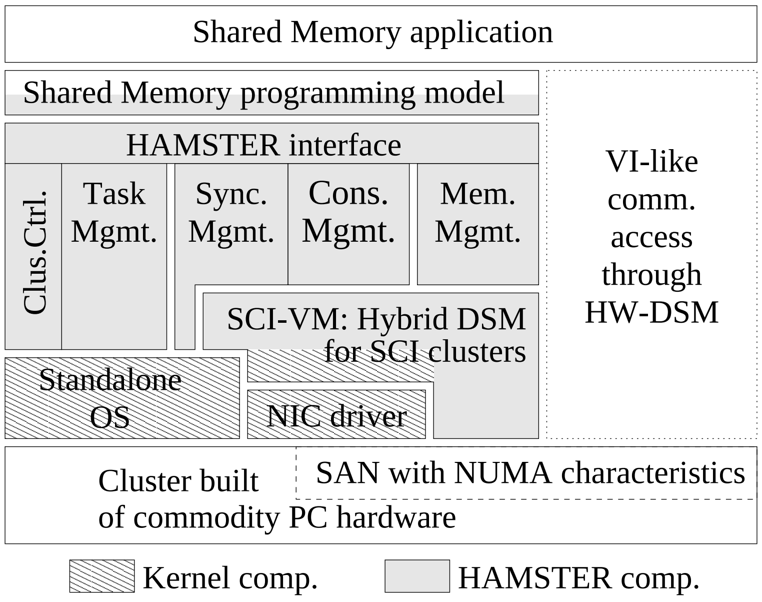

Shared Memory application
Shared Memory programming model
HAMSTER interface
Clus.Ctrl.
Task Mgmt.
Sync. Mgmt.
Cons. Mgmt.
Mem. Mgmt.
SCI-VM: Hybrid DSM for SCI clusters
Standalone OS
NIC driver
VI-like comm. access through HW-DSM
Cluster built of commodity PC hardware
SAN with NUMA characteristics
Kernel comp.
HAMSTER comp.

Figure 3.2 The HAMSTER framework for SCI–based clusters of PCs.

The services exported by all of these modules are combined in one single interface, depicted in Figure 3.2 as the “HAMSTER interface”, providing all mechanisms needed for the implementation of shared memory models in a structured and flexible way. This allows the implementation of such models with minimal amount of effort and code. The final applications, however, have no direct contact to the HAMSTER environment. They run fully transparent on top of the respective programming model and its API, using the benefits of the HAMSTER infrastructure without having to use yet another shared memory API.

The result is an open and extensible framework for shared memory programming on loosely coupled NUMA architectures. It combines a maximum of flexibility for applications running on top of it with a direct and low–overhead access to the interconnection network.

3.4 Underlying Principles

Before describing the individual components of the HAMSTER system in the next chapters, first a few common principles visible throughout the whole systems are introduced here.

3.4.1 The Principle of Orthogonality

One of the key design principles of the HAMSTER system is the orthogonality between the individual management modules introduced above. This means that no module is conceptually based on any other module or restricts the usage and/or the services provided by any other module. This guarantees a maximal amount of flexibility for any higher layer within the HAMSTER framework, as it prevents any inter–module constraints and side–effects. Higher layers can then use any HAMSTER service independently at the finest possible granularity to form the intended target programming model.

This orthogonality is achieved by clearly separating the services needed to implement shared memory programming models into the respective management modules. This task is not always straightforward as many typical constructs for shared memory programming are anchored in more than one module. An example for this, which will be discussed in greater detail in Chapter 6.3, is the combination of synchronization constructs and memory consistency enforcing constructs, as is typically used in relaxed consistency models. In such cases like this, the required functionality is broken up and independent services are implemented in each module. These can then be combined in an easy and straightforward way based on the level of and depending on the requirements of the individual programming models. On the other side, this approach also allows any higher layers to use any of the services alone without this combination resulting in the intended amount of flexibility.

In addition, this approach also allows an easier and safer implementation of the modules themselves, as they can be treated as their own individual entities. Module extensions and ports to new platforms can therefore be made faster as well as safer, thereby further increasing the flexibility of the system itself.

This principle of orthogonality is clearly visible throughout the complete design and implementation of the individual management modules. Chapter 5, which describes the modules in detail, will discuss this further and illustrate the influence of this overall design principle.

3.4.2 Resource and Performance Monitoring

In addition to providing the capabilities needed to implement various shared memory programming models, it is also necessary to enable resource and performance monitoring on top of the framework. For this purpose the HAMSTER system includes an implicit monitoring system. Each component collects statistics regarding various parameters during the runtime of an application and allows higher layers (potentially all the way up to the user if implemented by the programming model) to query them through a monitoring interface. This can then be used to assess the key parameters describing the behavior of the executed application and can help to improve its performance either through implicit or explicit optimization. The individual parameters and the respective interface will be discussed along with the individual components in the following chapters.

Without such an implicit monitoring approach, the user would be required to annotate each application separately in order to get some performance feedback. In addition, only a restricted amount of information could be collected using this approach, as access to internal mechanisms and/or parameters is not possible at the application level. The implicit approach hence provides a more reliable and more accurate performance data collection mechanism without the burden of any extra programming complexity for the programmer.

In addition to an application centered performance evaluation, which is discussed above, this HAMSTER inherent statistics collection also allows the utilization of system centered performance monitoring and steering using an on–line monitoring framework for shared memory systems, as it is currently envisioned within the SMiLE project [111, 110]. This allows the collection and evaluation of performance data independent of any programming model or application through an independent system. Such a system can then be used, in combination with performance data drawn from other sources, to visualize the system's behavior using on–line visualization tools and/or to automatically tune the performance using an adaptive runtime system cooperating with the respective HAMSTER modules.

It should, however, be mentioned that any performance monitoring can cause some overhead that is simply wasted in scenarios in which no performance data is needed. To reduce this impact, the implementation of the statistics collection mechanisms within HAMSTER is done in a very lean and careful way keeping the inferred overhead as low as possible. In scenarios, however, where even this is not enough, the user is given the option to turn any performance monitoring completely off using a compile–time switch during the compilation of HAMSTER. This could be useful for production environments without any automatic performance evaluation in which the absolute maximum in performance is needed; in general, however, the collection of performance statistics is not harmful and therefore enabled by default.

3.5 Summary

In order to tackle the two major problems currently associated with shared memory programming, the abundance of different shared memory programming models and APIs and the performance problems in loosely coupled systems, a comprehensive framework for shared memory programming is required. This work presents such a framework, the HAMSTER system, which enables the construction and use of almost any programming model on top of a single core. It targets commodity clusters built of PC components interconnected with SCI, a NUMA enabling SAN, and has been designed according to the following design principles: flexibility with regard to programming model and application needs, support of multiple operating systems, no bias towards a particular programming model, and easy retargetability to new programming models enabling the creation and maintenance of as many programming models as useful for a certain application scenario, as well as the direct use of the NUMA features of the underlying architecture in order to eliminate many of the typical DSM performance problems.

The result is a strongly modularized and therefore open and flexible framework which offers the necessary services to enable the construction of potentially any shared memory programming model. These services, which will be discussed in great detail in the next chapters, are grouped into orthogonal modules, each responsible for one main service class: memory, synchronization, consistency, and task management. These modules are complemented by a separate cluster control module responsible for the cluster management and the global process abstraction.

All of these services are combined into a unified interface, called the HAMSTER interface, which can then be used for the creation of arbitrary programming models without major complexity, as will be demonstrated later. Applications can then be implemented using these HAMSTER–based programming models while being completely unaware of the underlying shared memory framework. This ensures a full portability of existing shared memory codes without code modifications to HAMSTER–based platforms which creates the intended flexibility. This prevents the problems arising from the existence of a large variety of shared memory programming models without having to impose a new unified programming model, normally a long and tedious process.

Chapter 4

SCI Virtual Memory: Creating a Hybrid–DSM System

The main prerequisite for shared memory programming is a fully transparent, global virtual address space. This section describes a software layer that provides such a memory abstraction on top of SCI connected clusters of PCs, the SCI Virtual Memory or SCI-VM. In contrast to existing systems, it enables the user to directly benefit from the NUMA hardware, present in SCI–based systems. This is achieved by a software component closing the gap between the global physical memory provided by SCI in the form of HardWare Distributed Shared Memory (HW–DSM) and the required global virtual memory. The result is a high performance hybrid hardware/software solution in which any data transfer can be executed directly and transparently in hardware and only management and control functionality remains in software.

4.1 State of the Art

The idea of providing a global virtual memory abstraction for clusters and thereby enabling the use of shared memory programming models for NORMA architectures has inspired researchers for almost two decades. Due to the missing hardware support in this kind of architecture, however, a comprehensive software system is required to compensate for this deficiency. This has led to the development of the so–called SoftWare Distributed Shared Memory (SW–DSM) systems which enable a software emulation of the intended global memory abstraction. The following section provides an overview of their basic principles and the wide variety of existing DSM systems.

4.1.1 Idea and Concept of DSM Systems

The basic idea behind any DSM system is that a software component is used to track any memory access to globally shared data of an application. This enables the software to keep a copy of the global memory state and to determine when information needs to be communicated between nodes in order to maintain the global virtual memory abstraction. This translates the implicit communication present in the shared memory paradigm in the form of global memory accesses to explicit communication and hence can be conducted using some form of message passing. This approach is therefore suitable for all kinds of architectures including clusters and NoWs and allows the implementation of shared memory programming models on these architectures.

The key question in this approach is how to keep track of the memory accesses issued by an application without having to modify the application itself since this would destroy the intended transparency of the shared memory abstraction. The mechanism deployed for this purpose in most current systems is the ability of modern processors and operating systems to support page–based protection mechanisms. Based on these, global data areas can be protected which leads to the respective page faults on any access to the data. A DSM system can use page fault handlers to get a hold of these events and issue the necessary communication; on read faults data has to be brought to the faulting node while write faults lead to the marking of the page as dirty and/or the forwarding of the written information to other nodes. Due to their basic property of relying on the protection of individual pages, systems based on this approach are often referred to as page–based DSM systems.

The first system adhering to these principles is Ivy [136] (Integrated shared Virtual memory at Yale) which was implemented by Kai Li and published in his PhD thesis in 1986 [135]. It was the starting point for many DSM systems which took this basic principle and added new features or aimed at optimizing their performance. The result is a large number of different DSM systems [169, 96], all with the same intent to bring shared memory programming to NORMA architectures. The remainder of this section will present these systems along with their properties and thereby provide an overview of the current state of DSM research.

4.1.2 Memory Consistency Models for DSM Systems

The main problem connected with the SW–DSM approach is the high and fine grain communication demand created by the forwarding of global memory accesses. Therefore, a large amount of research has been invested in finding ways to reduce the communication in DSM systems. One of the most promising ways, which is by now used in any state-of-the-art DSM system, is a relaxation of the memory model [30]. It allows to defer the communication of update information. This optimization has, however, an impact on the application executed on top of such a system since information written by one thread is not immediately visible by other threads. This needs to be taken into account during any application development in order to guarantee a predictable and correct execution.

The role of memory consistency models

In order to give application programmers a formal way to reason about the correctness of their codes, the behavior of memory systems can be defined using the concept of memory consistency models [86]. They precisely characterize the behavior of the respective memory subsystem by clearly defining the order in which memory operations perform in the specific memory system. As a result, users are provided with a set of constraints containing this information. The users can then build their application based on these constraints resulting in a safe and predictable execution of the application.

The concept of memory consistency models and the idea of their relaxation is not only

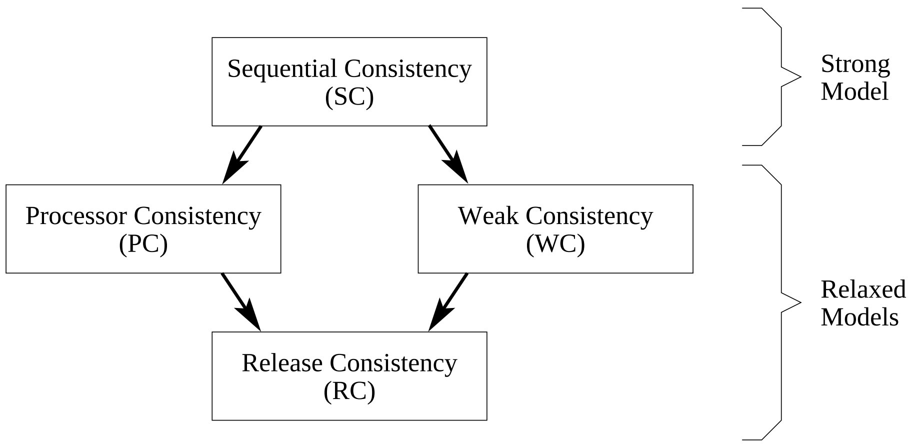


Figure 4.1 Classification of memory consistency models; arrows point from stronger to more relaxed models [86, 169].

relevant for DSM systems, but also applies to any multiprocessor shared memory environment. This has led to a significant amount of work in this area targeting both hardware and software approaches [3, 137, 161].

General classification

A general and intuitive classification of memory consistency models can be found in [86]. It distinguishes models into four main types, as depicted in Figure 4.1: Sequential Consistency (SC), Processor Consistency (PC), Weak Consistency (WC), and Release Consistency (RC). These models and their relation to each other are discussed in the following.

- **Sequential Consistency** ([129])

  Sequential Consistency (SC) was first described in [129] and is defined as follows: “A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”. While this model provides for the programmer a very intuitive and easy–to–follow memory system behavior, it can have severe consequences in terms of performance. The strict ordering requires memory systems to propagate updates early and prohibits many optimizations in both hardware and compiler. Due to this, several approaches have been proposed to relax the constraints of Sequential Constraints with the goal to improve the overall performance while limiting the impact on the programmer.

- **Processor Consistency** ([67])

  This issue is addressed by relaxed consistency models which weaken the strict ordering constraints imposed by SC. A first example for such a model is Processor

Consistency (PC). Under this model, *Reads* are allowed to bypass previous *Write* operations from other processors, while all *Write* operations are executed in program order. However, both operations execute in order with respect to all other operations from the same processor. This results in a scheme in which operations performed within a node behave identical to SC, but global operations are relaxed.

- **Weak Consistency Models** ([47, 205])

  Another way to relax Sequential Consistency is taken by Weak Consistency (WC) Models. They divide memory accesses into standard and synchronizing accesses. While the latter ones are guaranteed to be sequentially consistent across the whole system, weak consistency models allow standard accesses to be reordered in relation to each other. The exact reorder rules differ from one weak model to another. Examples are the DSB model [47], which is named according to its authors Dubois, Scheurich, and Briggs, and TSO (Total Store Order), which is used within the Sparc architecture [205]. A complete description of these models can be found in [86]. These models, however, require programmers to explictly take these consistency models in their applications into account by introducing the synchronizing accesses.

- **Release Consistency** ([63])

  The Release Consistency (RC) Model continues on the idea of Weak Consistency by further dividing the synchronizing memory accesses into *Acquires* and *Releases*. Any standard shared memory access can be performed in arbitrary order, unless all *Acquire* accesses have been completed before any standard memory access and all accesses have been completed before a *Release* access. This results in a scheme in which an *Acquire* informally allows the access to shared data from a particular node and ensures that all data is up-to-date, while the *Release* relinquishes this access right again and ensures that all memory updates have been properly propagated. By separating the synchronization accesses in this way, this model efficiently allows optimizations like buffering and pipelining and thereby forms the basis for a significant performance increase.

Release Consistency in DSM systems

Due to this large potential impact on the performance of the overall system, Release Consistency models have been widely adopted in DSM systems. Depending on the actual implementation of the consistency protocol within a specific DSM system, a few subtypes can be distinguished which differ with regard to when communication is performed and where data is located.

- **Eager Release Consistency**

  With Eager Release Consistency (ERC), updates are propagated to remote nodes as early as possible, i.e. at the time of a *Release* operation. An example for a system using ERC is Quarks [215], a system aiming at providing a lean implementation of DSM avoiding complex high–overhead protocols. ERC is very suited for this approach since, due to the early transmissions, only very little state needs to be preserved thereby reducing the work necessary for bookkeeping.

- **Lazy Release Consistency**

  In contrast, Lazy Release Consistency (LRC) [116] performs any communication of write updates as late as possible when the data is actually needed, i.e. during an *Acquire* operation. While this allows unnecessary transactions to be avoided in cases where released data is not used or overwritten by subsequent *Acquire* operations, it leads to an increased amount of work to be done for bookkeeping. This stems from the fact that released data has to be stored on the local node for future *Acquire* operations until all nodes have requested the data or it is overwritten. Despite this increased complexity, LRC has proven its capabilities in a variety of systems, including TreadMarks [5], Shasta [189], and Munin [29].

- **Home–based Lazy Release Consistency**

  The Home–based Lazy Release Consistency (HLRC) model [179] is a slight derivation of LRC and only differs in the fact that each shared data page is statically associated with a home location, instead of being unbound as in the standard LRC case. This again allows some further optimizations with regard to both application performance and overhead reduction through reduced bookkeeping [256].

Except some minor differences [116], all of these implementation options provide the application programmer with the same relaxed consistency model and hence with the same consistency constraints but their different implementations have an impact on the application performance. However, none of these protocol implementations of RC is generally faster than the others. The concrete performance strongly depends on the individual applications and their memory access patterns [256, 119].

New models from the area of DSM systems

The consistency models discussed above with their various protocol implementations are dominating the area of DSM systems. Only few other projects investigate in new, more relaxed consistency models with new consistency constraints exposed to the application. The most prominent examples are *Scope* and *Entry Consistency*, which are briefly discussed below.

- **Scope Consistency**

  Scope Consistency [97] is a generalization of Release Consistency. It is based on the observation that in an RC implementation *Acquire* operations guarantee the visibility of data written before the last *Release* operation performed in the system. Scope Consistency relaxes this constraint by grouping *Acquire* and *Release* operations and by restricting the constraint above to only operations within the same group. This implicitly restricts the visibility to regions of memory protected by consistency operations within the same group, the so–called consistency scopes. As a result of this, a scope consistency compliant DSM implementation is enabled to decrease the amount of data that needs to be transfered during consistency operations reducing the DSM overhead and providing higher overall performance.

This type of consistency model is e.g. used in JiaJia [83] and Brazos [209], both state-of-the-art SW–DSM systems. JiaJia is implemented for UNIX based clusters and deploys a home–based version of Scope Consistency, while Brazos has been developed for Windows NT™ based architectures and supports active multithreading.

- **Entry Consistency**

  In contrast to the implicit relation between consistency operations and data to be kept consistent used in Scope Consistency, Entry Consistency [21] as it is implemented in Midway [22] is built upon an explicit scheme. The user associates each lock and its related consistency operations with regions of memory; only these regions of memory are then kept consistent by operations of this lock.

  While this scheme allows a more precise definition of the data which needs to be transfered or updated during the individual operations and helps to avoid unnecessary communication, it is burdened by the fact that programmers have to explicitly specify the associated memory regions for each lock, often a complex and cumbersome task.

Support for multiple coherency models and protocols

As already mentioned above, no single consistency model or consistency protocol implementation has proven unrestricted superiority above the others. It strongly depends on the individual application and its memory access pattern as well as on the properties of the underlying architecture. This has inspired work towards DSM systems capable of supporting multiple consistency models within one system [6, 30]. These are then adaptable to the actual constraints and behavior of the target application and can therefore deliver optimized performance. Examples for systems with these capabilities are the Coherent Virtual Machine (CVM) [117], a framework for multiple consistency models, ADSM [160], a Release Consistent system enabling the user to choose among different protocols on a per page basis, and Munin [29, 17], which is based on Lazy Release Consistency but allows users to influence many of the protocol parameters.

4.1.3 Alternative Approaches

Beyond the optimization approaches for page–based DSM systems through various memory consistency models, DSM research has also focused on further developments. One direction for this are efforts to decrease the distribution and sharing granularity. Page–based systems are in this respect by their nature always fixed to full pages. In the case of the x86 architecture [99] this amounts to 4096 bytes. This large and fixed granularity often has an unfavorable impact on the overall performance of applications. It increases the chance of false sharing, i.e. the colocation of unrelated data within a single sharing unit (in this case within a page) and hence can lead to unnecessary communication if multiple processors access these independent parts of data.

A reduction of the distribution and sharing granularity, however, can only be achieved by abandoning the page–based implementation and by replacing it with a new mechanism capable of transparently tracking the memory accesses of applications. One approach for this is the annotation of load and store operations to global data directly in applications. This information again provides the complete state of the global memory to the DSM system and enables it to create a global virtual memory abstraction. In order to still maintain the transparency of this abstraction for the user, the required program annotation can not be part of the source code itself, but rather has to be introduced into the final application without user interaction. This can be done with the help of binary instrumentation tools which insert the necessary hooks for the memory control into the final application binaries without any further user intervention.

Examples for systems built according to this type of memory subsystem are the Shasta system [189] and CRL [102]. Using this approach, they are enabled to keep track of every memory access independently (at least in principle) and can therefore adapt the sharing granularity to the application needs without any restrictions.

A similar approach in the same direction has been undertaken by the Midway system [254], although used here in combination with page–based mechanisms and in cooperation with a special compiler responsible for adding code enabling a fine granular write detection. This is done by setting dirty flags, which are then evaluated before the actual data transfer. The result is a hybrid solution which reduces the sharing granularity without having to rely on a complex full software scheme as the systems above.

Another approach to reduce the sharing and distribution granularity is the use of an object–oriented approach in combination with a global object space. There, objects are used as the basic unit of sharing and hence the granularity is automatically adapted to the application's data structures. In addition, objects also provide a natural way to track data accesses since objects need to be dereferenced and located in the global object space. Examples for such a systems are Millipede [100], an object–oriented DSM approach for Windows NT™ –based clusters, and Orca [11], a parallel programming language based on shared objects.

Besides these performance oriented research directions, some projects also aim at overcoming functional deficiencies of traditional DSM approaches. One of them is that DSM systems generally only support homogenous environments requiring each node in the system to be of the same architecture and often also of the same speed. The Stardust [28] and the Mairmaid system [255] try to overcome this by explicitly targeting heterogenous cluster and NoW architectures. This leads to several problems, most severely the potentially different data formats on different architectures. The two systems use different approaches to solve this problem: while Stardust relies on explicit user annotations, the Mairmaid system uses implicit type specifications exhibited through the loader and linker facilities of the underlying operating system. Both approaches provide the DSM system with type definitions for any shared data and thereby enable a transparent conversion.

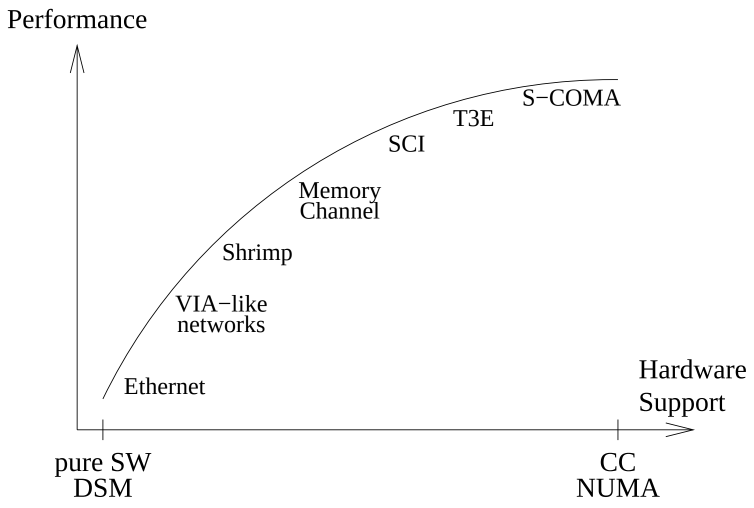


Figure 4.2 Hardware complexity vs. Performance (after [201]).

In addition to the support for heterogenous architectures, the Stardust system also contains mechanisms for application checkpointing [27]. This allows the persistent storage of the complete state of the application including the global memory footprint which can be reactivated after a system or node failure. The Stardust system therefore is part of the class of recoverable DSM systems. A more complete overview over this class of systems can be found in [162].

4.1.4 Exploiting Existing Hardware Support

The DSM systems discussed so far rely in their implementation solely on communication based on standard protocols like UDP/IP or TCP/IP. While this decision offers a clean and portable approach for general loosely coupled systems, it fails to sufficiently exploit the available capabilities of novel architectures which often feature additional hardware mechanisms. These mechanisms could be exploited to increase the efficiency of DSM systems.

Figure 4.2, which is created after [201], illustrates this approach. With rising hardware complexity in the underlying target architecture, DSM systems implemented on top of them are expected to gain performance (if the hardware capabilities are properly exploited). The spectrum of possible hardware support can range from standard cluster technologies, i.e. with no special hardware support at all, all the way to full CC–NUMA systems which include the DSM support in both their hardware and their global operating system, as is the case in the SGI O2000/O3000™ series [131]. This spectrum also includes all kinds of system area networks (SAN) with user–level communication and NUMA–based architectures, which are the focus of this work.

Several projects have tried to exploit such additional hardware support to implement DSM systems. The most prominent examples among them are SHRIMP [19, 18] and

CASHMERE [212, 125]. In SHRIMP, a custom–designed network adapter with an automatic global update functionality is used, while Cashmere utilizes DEC's Memory Channel [65], a high–performance SAN with remote write capabilities. In both cases, the remote memory access capabilities can be used to implement update operations in a very direct and efficient way [213, 16] which has proven to be advantageous for the overall system performance.

Both systems, however, miss one important feature — remote reads. This leads in both cases to an implementation that exhibits the same base concepts as in traditional DSM systems: information has to be brought to nodes using explicit communication at page granularity. Only updates can be performed in a more direct way using the hardware support, although still in a manner suited for the page–based information retrieval. Such an approach therefore leads to relaxed consistency models in a more or less traditional, protocol–oriented fashion even though they have to be especially adapted to the implicit remote updates.

NUMA architectures, as they are e.g. present in the form of SCI–based clusters (see Chapter 2.3), provide the ability for both remote write and read operations and hence have the capabilities to overcome the need for explicit communication. Their use, however, is also connected with some implementation complexity, as the remainder of this chapter will show. Therefore, many DSM systems have been implemented for these architectures based on the traditional page–based mechanisms, but exploit the NUMA capabilities for a fast messaging subsystem or for an easier global memory management. Examples for these kind of systems based on SCI are Madeleine [10], NOA [151], a DSM system developed by the Lund University in Sweden [114], the HPPC–SEA DVSM library [40], and the SVMlib [174].

Only few projects aim at a direct utilization of SCI's NUMA capabilities for a DSM system. The SciOS system, developed in cooperation between the University of Copenhagen and INRIA [122, 123], provides a global System-V memory abstraction which is mainly based on a traditional page–based SW–DSM approach, but also enables the use of direct NUMA in some special cases. In contrast, the Yasmin system [25, 180] implements a global memory abstraction based only on SCI's NUMA capabilities. However, it merely resembles a thin mapping library for SCI remote memory and is restricted by the large granularity of the SCI mappings (see Chapter 2.3.2). In addition, the system does not contain any mechanisms enabling the use of caches, resulting in the caches being turned off, or allowing the control of the memory coherence leading to increased application complexity.

The system discussed in this work, the SCI Virtual Memory or SCI-VM [195, 198], takes the idea of hardware support one step further by directly relying on the NUMA–like shared memory support in SCI and avoiding a software scheme based on page replication of differential update protocol. This helps avoid typical performance problems of traditional DSM system like false sharing and extensive protocol overhead. In addition, the missing cache coherency in the system is compensated based on the concept of relaxed consistency models which have been adapted for the use on NUMA–based systems. Therefore, the SCI-VM represents a novel approach in the DSM area closing the gap between systems with modest hardware support in the implementation of the DSM protocols and CC–NUMA systems with full hardware and operating system support, as can be seen e.g. in the SGI O2000/O3000™ [131] or the HP/Convex Exemplar™ [35].

4.2 The SCI-VM Design

The SCI Virtual Memory (SCI-VM) is based on two fundamental building blocks, the SCI support for hardware DSM in NUMA fashion and the concepts for a global memory management borrowed from traditional software DSM systems. By combining these two concepts, the system is capable of merging the global physical memory provided by SCI with the global virtual memory abstraction required for shared memory programming. The result is a new hybrid hardware/software DSM system taking full advantage of the SCI hardware facilities.

4.2.1 Building Block 1: SCI–based Hardware–DSM

One of the main design goals of the SCI-VM is the direct utilization of 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 [143]. Only this exploitation of HW–DSM enables the SCI-VM to benefit from the special features and the full performance of the interconnection technology. Additionally, the implementation of synchronization primitives should directly utilize atomic transactions provided by SCI to ensure greatest possible efficiency.

4.2.2 Building Block 2: Software–DSM Systems

Unfortunately, SCI alone can not provide a global virtual memory abstraction as required by shared memory programming models. Both its hardware and software components target only the utilization of large, contiguous, and permanently pinned memory segments. In order to overcome these limitations and to reach a fully transparent implementation of a global virtual address space, the SCI remote memory capabilities have to be augmented by concepts and mechanisms well known from traditional software DSM systems, like data distribution at page granularity, on–demand access to remote pages, and a relaxed consistency model [169].

4.2.3 Combining Both Building Blocks to the SCI-VM

Together, the building blocks described above allow the formation of a transparent virtual address space. The memory resources are distributed at the granularity of pages and these distributed pages are then combined into a single global virtual address space. In contrast to purely software based systems though, no page has to be migrated or replicated. All remote pages are simply mapped using SCI's HW–DSM mechanisms (described in Chapter 2.3.2) and then accessed directly. Due to the large amount of necessary mappings (in the worst case one for each remote page), these mappings are handled on demand with a similar concept as realized in paging mechanisms of modern operating systems. The SCI-VM

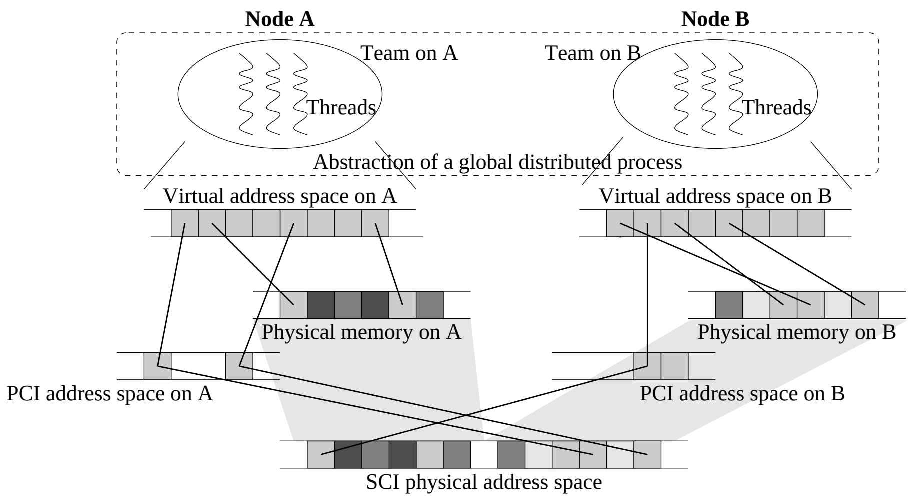


Figure 4.3 Principal design of the SCI Virtual Memory.

therefore represents a cluster–aware extension of an operating system's virtual memory management.

This concept is further illustrated in Figure 4.3 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 on each node as placeholders for the global process. 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's HW–DSM.

The mapping of the individual pages is 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 mapping granularity depends on the configuration of the PCI–SCI adapter and can vary from 4 Kbyte or 1 page segments to 512 Kbyte or 128 pages segments. As this setting is done during the initial setup of the adapter cards and can not be influenced during the runtime of an SCI–based application, the SCI-VM layer has to overcome this difference. For this purpose, it manages the mappings of several pages within one single SCI segment. The mappings of the SCI segments themselves are managed with a dynamic, on-demand scheme very similar to paging mechanisms in operating systems.

4.3 Static vs. Dynamic Memory Management

The implementation of the basic design above can be either be done by using static mappings during the memory allocation time or by deploying a fully dynamic memory management scheme. Both exhibit the same functionality to the application, but have different properties with respect to implementation complexity and resource management. Therefore, even though this choice is transparent to the rest of the HAMSTER framework, both options are introduced and discussed below.

4.3.1 Static Memory Mappings

The first option to implement the SCI-VM is to create the necessary mappings in both the network's HW–DSM mapping tables and in the processor's page table during the allocation of a piece of global memory and then keep them untouched until the application is terminated. This results in a static scheme in which any memory location within the global address space is accessible at all times. This also includes the necessity to pin down any physical memory backing the global virtual address space and thereby preventing it from being swapped out to secondary storage. Otherwise, the static mappings via SCI would point to invalid memory locations resulting in unprotected accesses to physical pages not belonging to the SCI-VM controlled process.

Allocation procedure

The core of such an implementation can be found in the actual allocation routine since any physical memory is allocated and all mappings are created there. This is done in a five step process, which is depicted in Figure 4.4. First, the necessary piece of virtual address space is reserved. This is done equally by all nodes using a globally consistent counter that guarantees that all nodes are provided with the same piece of virtual memory in their respective teams. In the next step, the amount of memory (in terms of numbers of pages) needed by the local node to contribute to the global memory abstraction are computed and then allocated in step three.

In step four, a globally shared table is used to store an entry for each page contained within the new virtual address space. Each node then enters the physical addresses along with its own network ID for all pages it contributes to the newly allocated virtual address range. During this step, the decision is made on how the physical memory is to be distributed among the participating nodes. By default, the SCI-VM uses a round–robin scheme for the page placement, but this can be overwritten by the caller of the allocation routine and replaced by a new distribution policy. More details on this are provided in Chapter 5.1.3. The step is finished by a global barrier across all nodes to ensure the writing of the page table is complete and all information from all nodes has been entered.

After the completion of the barrier, all nodes have access to this page table with a complete description of any page within the newly allocated global virtual address space. They can now start to map all pages into their location within the local team's virtual address space. During this, any page located on the local node is simply mapped using the MMU while any remote page is first mapped via the network mapping mechanisms and then by the MMU.

At the end, each team has the complete new piece of virtual address space mapped and accessible for the rest of the process runtime. No further management calls have to be

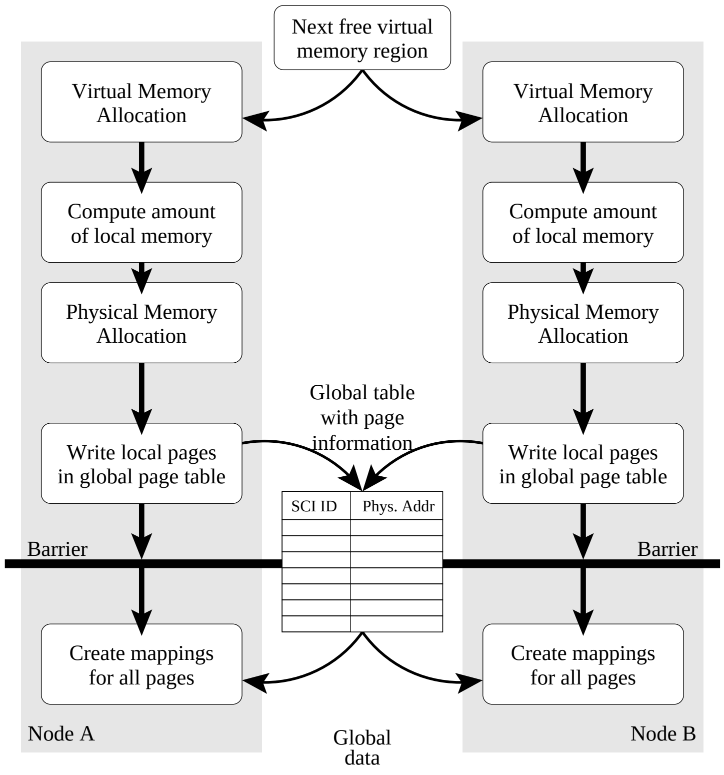

Next free virtual memory region
Virtual Memory Allocation
Virtual Memory Allocation
Compute amount of local memory
Compute amount of local memory
Physical Memory Allocation
Physical Memory Allocation
Global table with page information
Write local pages in global page table
Write local pages in global page table
SCI ID
Phys. Addr
Barrier
Barrier
Create mappings for all pages
Create mappings for all pages
Node A
Global data
Node B

Figure 4.4 Static memory allocation procedure.

made; the new memory can be accessed from the allocation time on by simply using standard read and write operations and any access to remote memory is transparently relayed to the respective node by SCI.

Implementation requirements

One of the main advantages of this static SCI-VM scheme is that the implementation complexity is very low and also the required operating system services for such an implementation are very limited. Basically only two key functions need to be provided: the allocation of pinned physical memory and the mapping of arbitrary physical pages into the virtual address space of a process at page granularity. These two services suffice for the implementation of the static scheme; the allocation is required to allocate physical pages on the local node that are prepared to be accessed in a safe way via SCI and the mapping routine is used to merge local and remote pages into a single virtual address space for the local team.

Besides these operating system services, a direct access to the hardware mappings within the SCI adapter are also needed to allow an access to remote memory resources at the granularity of single ATT entries. This should be part of the respective network device driver to ensure a clean cooperation of the SCI-VM with other applications using the network device at the same time.

Tradeoffs

This static implementation scheme closely matches the memory allocation scheme of many shared memory applications. A large piece of memory is allocated during the initialization of the application, maintained and accessed during the whole runtime, and not freed before the end of the application. In addition, many applications, especially those from the area of scientific or numerical computing, aim at not using any operating system driven paging since this normally has a large negative impact on performance and therefore do not require a support for memory paging. Together with its low implementation complexity, this scheme therefore provides a good tradeoff point and was therefore chosen for the first implementation of the SCI-VM.

4.3.2 Dynamic Global Memory Management

The static scheme presented above, however, misses a few characteristics that are desirable in order to provide a more efficient resource management and more flexibility in page placement. This can only be achieved by tightly integrating the SCI Virtual Memory into the virtual memory management of all operating system instances in the whole cluster. This leads the way to a global virtual memory management across node boundaries with a global resource management. Such a system solves all of the shortcomings described above in connection with the static scheme.

Global virtual memory management

Even though this dynamic memory management scheme provides the same functionality to the user, its internal structure is radically different. In contrast to the static scheme, the allocation procedure is very lean and contains only a small part of the overall implementation. It is only responsible for reserving a new piece of virtual memory and protecting it using the virtual memory management facilities. It does not perform any allocation or mapping of memory resources at this time.

The actual functionality of the SCI-VM under this scheme is executed at runtime. Accesses to the global virtual address space create page faults which will be handled by the system. Figure 4.5 shows the main activities carried out by the SCI-VM in case of such a fault event: first it is tested whether the faulting address actually belongs to the virtual memory region under SCI-VM control. If not, the fault is handed back to the operating system and from there to the default handler. If the address is part of the SCI-VM, the home node of the faulting page is queried either using a global page information structure or based on a static page distribution that was specified during the allocation process.

If the local node is the home node, the page can be treated as any conventional page of the local memory and hence the control is handed back to the operating system for further page fault handling. In case a page from a remote node has been accessed and no mapping to that page has been established yet, the request is sent to that particular node. This node is then responsible for ensuring that the requested page is available in memory using the same principle mechanism. Once this has been accomplished, the page can be mapped on

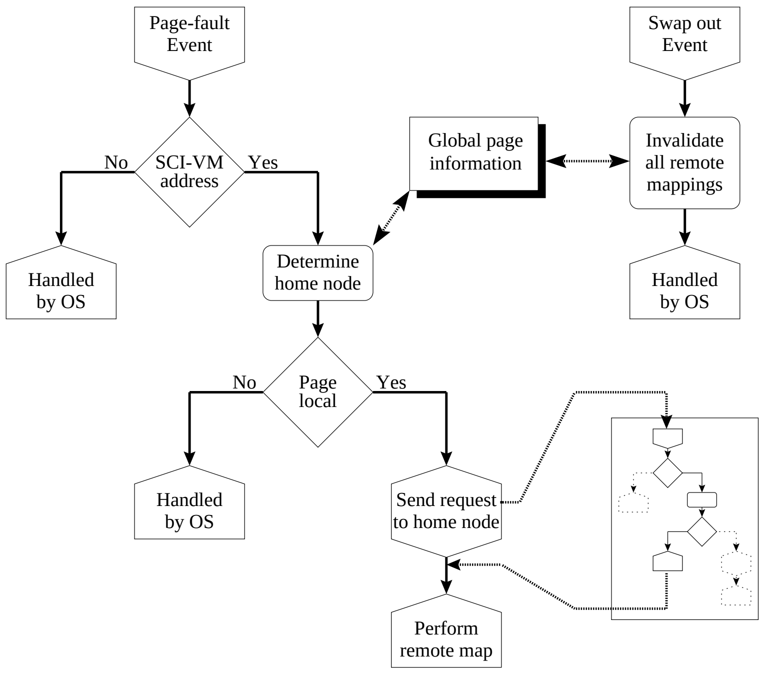


Figure 4.5 Activities triggered by a page fault or swapping event.

the node on which the fault occurred, allowing full access from that particular point of time on.

As all allocation and mapping functionality is performed by the operating system, all pages will be allocated from the pool of pageable pages. This leads to the problem that pages might be swapped out to disk while SCI memory mappings still exist from other nodes. This would leave them stale and risk wrong write accesses to memory not participating in the execution of the HAMSTER system. In order to solve this, a dynamic scheme also has to create a suitable hook into the virtual memory management layer that allows it to be notified on page–out events of SCI-VM pages. This can then be used to invalidate all SCI mappings to the page that is about to be evicted from the local memory allowing a safe execution of the application even without this page being present.

Implementation requirements

In contrast to the static approach, the dynamic memory management requires a much tighter integration into the operating system and therefore also a more versatile interface to core services of the virtual memory management. Besides the actual mapping mechanisms that are also required by the static scheme, the dynamic scheme needs to be able to insert hooks for callbacks on events like page faults and page swap–out. While the first one is given in most operating system and is normally also available in user mode (in UNIX systems in the form of signals), the latter event is generally not available. Therefore an additional kernel extension needs to export this functionality in order for the dynamic scheme to be implemented.

In addition to these kernel oriented requirements, a dynamic scheme also relies on a much more complex communication infrastructure. Unlike the static scheme, which basically only requires the existence of a user–level barrier at allocation time, the dynamic option requires communication at arbitrary times during the program execution at both user and kernel level. This further increases the complexity of the implementation.

Tradeoffs

The discussion above has shown that the main drawback of the dynamic scheme is the rather high implementation complexity, which comes from the necessary deep integration with the operating system and the required additional kernel component. On the positive side, the dynamic scheme enables a full integration into the operating system's virtual memory management and therefore allows the utilization of swappable memory as well as an on–demand page allocation and node assignment. These advantages lead to a more efficient resource utilization without the need to split the physical memory in pinned SCI segments and standard physical memory. In addition, it eases the implementation of dynamic and adaptive page placement schemes since the decision where to place a certain page can be performed at runtime and can take observations about the application's runtime behavior into account.

4.4 Underlying Task and Execution Model

Independent of the implementation, the design of the SCI-VM directly imposes a specific task and execution model on the final system. Each node has to execute one team process with one or multiple threads. These processes are then combined into a global process abstraction visible to the user of the SCI-VM. In order to achieve such an abstraction of a single process, any team is also required to load and execute the same binary. Only this provides a consistent address space layout across all nodes, being one of the prerequisites to form a transparent global process abstraction.

Based on this task model, an SCI-VM based application is started by executing the binary on each node in the cluster. During the initialization of the application, the individual processes are transformed into teams and merged into the intended global process abstraction. Initially, each team executes one thread of activity that also continues the execution of the binary on each node after the initialization. This initial thread per node can then be used to spawn further threads and to carry out the actual computation across the complete system.

Even though the SCI-VM is based on such a rigid and static task model, it can serve as the basis for more complex and dynamic models which will be described in Chapter 5 and Chapter 6. It does therefore not restrict the flexibility of the HAMSTER system with respect to the intended ability to implement shared memory programming models with potentially quite different task models.

4.5 Integrating the SCI-VM with Existing SCI–based Platforms

The approach presented above represents a direct extension of the virtual memory concept from node local memory resources to global ones. Its implementation therefore requires a tight integration into the existing software infrastructure. Specifically, the SCI driver infrastructure responsible for the SCI address translation mechanism and the virtual memory management of the underlying operating system need to be considered in this endeavor.

4.5.1 SCI Driver Integration

As described in Section 2.3 users, by default, get access to SCI through a special low–level user API called SISCI API [64]. Within this API, shared memory management is done on the basis of individual coarse grained segments, each with its own address space. This is not sufficient for a concept like the SCI-VM. Here, fine-grained mappings at page granularity and the inclusion of mappings into a given virtual address spaces are required. In addition, the memory management within the SISCI API is always done with global operations in order to guarantee a global visibility and availability of all segments. This, however, is very costly and induces a high overhead. This is also not desirable in the SCI-VM concept.

Due to these reasons, an implementation of the SCI-VM requires a more direct access to the hardware on the SCI adapter cards responsible for remote memory mappings. In order to allow for this, a special extension has been integrated into the existing driver infrastructure provided by Dolphin, more specifically the Interconnect Resource Manager (IRM), the lower–level driver with direct access to the hardware. This extension, called *Direct ATT extension*, allows programs to directly allocate, configure, and release ATT entries as well as to directly control the remote memory mappings. As those operations obviously are potentially unsafe since they allow to access arbitrary memory locations throughout the cluster, they are implemented as a kernel level API and only for testing and prototype purposes exported into user level.

This extension is also implemented in a way that other programs running on top of the SCI driver infrastructure are not influenced by an SCI-VM application. This ensures a full interoperability of the HAMSTER environment with the existing infrastructure and also enables HAMSTER itself to use the existing user–level parts of the SCI driver stack for its own purposes, like configuration segments and interrupt management.

4.5.2 Operating System Integration/Extension

The integration of the SCI-VM into the virtual memory management of the underlying operating system imposes further difficulties. The mechanisms offered by the public APIs of current operating systems do not provide the page–level granularity and low–level access required for the SCI-VM, even though the necessary routines have to be present as internal mechanisms within the operating system's core. It is therefore necessary to use these internal mechanisms or add additional functionality to the operating system kernels to guarantee a full integration into the virtual memory management. As this is very dependent on the concrete underlying operating system, the low–level part of the SCI-VM implementation also varies greatly between the different supported platforms.

Windows NT/2000™

The first version of the SCI-VM was implemented on top of Microsoft's Windows NT™ 4.0 operating system [146]. As is the case with most commercially available operating systems, the source code is not freely available, thereby imposing a significant challenge for the implementation of systems like the SCI-VM. It prohibits a direct access to low–level mechanisms of the kernel beyond the exported and documented API for kernel–level drivers which is not designed to allow a fine–grain access to memory mappings or to enable callbacks on important system events like page faults or disk swap requests.

Therefore, the dynamic version of the SCI-VM can not be realized on top of this platform, and the implementation of the static version can also not be done in a clean cooperation with the operating system due to the missing mapping facilities. The only way to overcome these shortcomings is to bypass the operating system and to perform the necessary mappings at page granularity directly at processor level building on the mechanisms of the CPU's memory management unit [99].

This approach, however, has some severe consequences for the OS. As it is completely bypassed and therefore unaware of the manipulations, its stability and robustness is impaired due to conflicts between the SCI-VM and the virtual memory management of the underlying operating system. An important design guideline used throughout the complete SCI-VM implementation is therefore the hiding of modifications done by the SCI-VM from the operating system and the use of distinct resources wherever possible. Based on several experiments using the current version of SCI Virtual Memory concepts, Windows NT™ has proven to exhibit an acceptable stability enabling extensive experiments. The critical point during the execution of a program using direct page mappings is its termination, especially with several DLLs[1] involved. While freeing a process's memory resources, Windows NT™ under certain circumstances also attempts to free the resources controlled by the SCI-VM it is not aware of, thereby leading to an unpredictable system behavior.

Linux – Static memory mappings

Even with access to the source as is possible on the second supported platform Linux, the implementation of a dynamic version with full support for a pageable, global memory and dynamic remote page requests, is very complex. Therefore, a static version has been implemented, however in contrast to the first scenario, in cooperation with the operating system rather than working around it. Any page mapping required by the SCI-VM is done using low–level routines of the Linux kernel and therefore does not cause any conflict. The access to these routines is enabled through a special kernel driver module developed for the static version of the SCI-VM which exports this functionality into user–mode. As a result, the static version of the SCI-VM on top of Linux, even though not fully integrated into the

[1]Dynamic Link Libraries – Windows NT™ 's counterpart to shared libraries

memory management as a dynamic SCI-VM solution, represents a stable implementation of the concepts discussed above.

Linux – Dynamic global memory management

Using the open source approach of Linux, a fully dynamic version of the SCI-VM concepts can also be implemented. This endeavor is further simplified, as the virtual memory management of the Linux kernel allows to set callbacks for various events, including the swapping of pages [184] . Together with the available mechanism to register callbacks for page faults, all necessary prerequisites for the implementation are available.

In order to ease the implementation, parts can be implemented in user level. Looking back at Figure 4.5, which provides an overview of the internal procedures that need to be implemented, the dynamic scheme can be split into two parts: the handling of page faults leading either to the creation or a swap–in of a page and the invalidation of mappings in case of a swap–out event by the operating system. While the latter requires a kernel–level implementation since the swap out callback is only available inside the kernel, the former part can easily be implemented in user level using the application level fault facilities.

Currently, the dynamic scheme for Linux is still under development and will be available soon. Therefore, all following experiments are based on the static schemes for both Windows NT™ and Linux. However, as explained earlier, the memory characteristics of both schemes are the same from the viewpoint of applications running on top. This guarantees that the results of the following experiments give a clear indication of the overall system performance independent of the concrete implementation of the SCI-VM.

4.6 Memory Coherency of the SCI-VM

The concept described above directly relies on the hardware DSM provided by SCI and therefore any application running on top of it is heavily influenced by its performance. Hence, it is a fundamental necessity to optimize the SCI memory performance. Similar to the traditional DSM approaches discussed above, this can also be done within the SCI-VM by applying relaxed consistency models. These allow the reduction of the amount of network transactions, or more accurately in this context remote memory accesses, and therefore have the potential to improve the performance of applications running on top of the global memory abstraction.

The mechanisms available to reach this goal are, however, fundamentally different. As discussed above, the traditional DSM systems control the complete virtual memory and all communication to achieve the global memory abstraction. Therefore any memory optimization is also implemented in software giving the DSM system the full control over the communication carried out within the system. In a Hybrid DSM approach as seen within the SCI-VM, however, communication is done implicitly by the hardware without software influence. While this increases the efficiency of the communication by reducing overhead, it also limits the possible optimizations that can be applied to hardware mechanisms available in the memory pipeline of the underlying architectures. These mechanisms are introduced below, together with their system–wide impact on the memory coherence of the SCI-VM and how to deal with it to guarantee a correct program execution.

4.6.1 Memory Optimization

Both the SCI adapter and the underlying PC architectures offer possibilities for such optimizations. They all deal with transparent buffering or prefetching of remote data and therefore have the potential to reduce the network traffic significantly.

Additional buffering in the SCI adapters

The PCI–SCI adapters by Dolphins ICS [138] offer a large range of parameters to control the memory traffic for both read and write operations. By default, these settings are set in a very conservative way to minimize network traffic and to optimize for message passing communication. It provides, however, several features that can be used to improve the performance of a distributed virtual memory system.

For write operations, the Dolphin adapters introduce the concept of streams which allow write accesses to consecutive addresses to be merged into longer network transactions. The current version of the adapters provide eight concurrent stream registers, allowing for up to eight series of accesses to be collected independently.

Most important for read operations are speculative buffering and aggressive prefetching. Both techniques allow implicit prefetching of consecutive address ranges in hardware by the SCI adapter card [44]. Enabling speculative buffering initiates the transfer of whole 64 byte packets and speculatively holds this data in the stream buffers on the Dolphin SCI card to be used by future read requests. The aggressive prefetching utilizes a similar technique but fetches the next consecutive 64 byte block and loads the data into another stream buffer in the same manner. Obviously, both techniques increase the network traffic on the SCI network and could result in the transport of potentially unused data. Nevertheless, they can improve the overall performance of the system, as can be seen in the experiments presented at the end of this chapter.

Caching of remote memory

These settings alone, however, are not enough to guarantee a high performance of the overall system. Especially the long latency of read operations has a drastic impact on the overall performance of shared memory applications. One way to reduce these latencies is to buffer or cache the result of remote reads on the local node avoiding many remote memory accesses and their latency penalties. This can be achieved by enabling the standard processor caches for remote memory regions, an option normally disabled due to the missing cache coherency in current PCI–SCI bridges (see also Chapter 2.3.3).

In the x86 architecture based on Intel CPUs, the cache settings are controlled using the memory type range registers (MTRRs) of the processor [99]. With these registers, up to eight individual physical memory regions (including the I/O space of the PCI bus used by SCI adapters for remote memory mappings), can be defined and associated with different

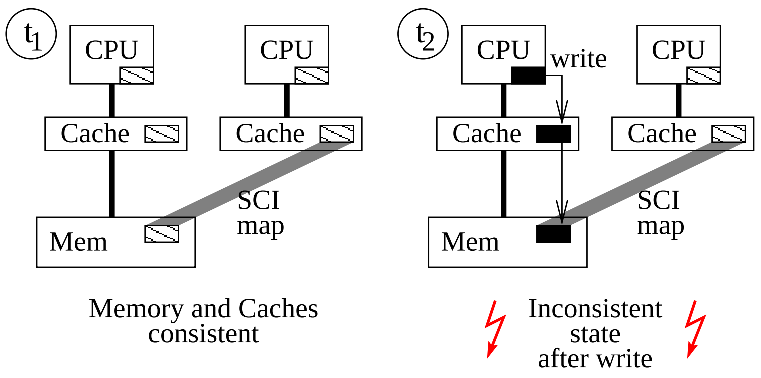


Figure 4.6 Potential cache inconsistency when caching remote memory.

cache types. For this purpose, the Pentium II™ processor offers five different cache types ranging from uncacheable and strong ordering to write back caching with speculative loads and weak ordering. To provide the best performance, a PC's main memory is normally set to use the latter option. This setting for SCI memory, however, causes problems on the PC bus system because the PCI bus bridge does not allow write back operations from caches to be delayed. Exactly this can happen though when an SCI transaction fails and has to be retried by the SCI hardware. The result is a deadlock on the PCI bus and a halt of the whole machine [101]. To prevent this, only write through caching can be used for the SCI address range.

4.6.2 Impact on Memory Coherency

All of these optimizations of the memory system influence the memory coherency of the overall system. This stems from the fact that remote data is stored in various buffers throughout the whole system without being held consistent. Especially visible is this impact with the caching of remote memory in absence of a hardware cache coherence protocol. As shown in Figure 4.6, updates on local memory are not propagated to remote caches leaving them with stale values. To avoid any negative impact on the execution and the correctness of the application in such scenarios, it is necessary to deploy additional software mechanisms controlling the memory consistency.

These mechanisms need to provide the functionality to manage the utilized buffers and caches in a way that allow the control over their contents. This means that it has to be possible to flush and/or invalidate these buffers at any time and with that to either propagate local data through the network to its destination or to delete stale data on the local node. All of these mechanisms are provided within HAMSTER within the so–called consistency management module which is discussed on more detail in Chapter 5.2.

In summary, this leads to a straightforward extension of the hybrid character of the SCI-VM to the memory coherency control. While the actual memory model is still fully implemented in hardware, the control is done in software. This hybrid scheme differs significantly from the consistency enforcement used in SW–DSM systems; there the actual communication and hence the propagation of memory state is triggered by the DSM software at well–defined times. In the hybrid scheme discussed here, on the other hand, the communication is performed implicitly by the system without user control. The consistency enforcing mechanisms merely resemble memory barriers guaranteeing that communication has been completed at code positions requiring a consistent view on global data.

4.6.3 Towards Relaxed Consistency Models

By utilizing the memory optimizations discussed above and by combining them with the appropriate usage of the consistency enforcing mechanisms, the SCI-VM is capable of implementing a large variety of different consistency models. However, not all consistency models discussed in the current literature (see Chapter 4.1.2) can be implemented due to restrictions inherently embedded in the underlying hardware architecture.

The most severe restriction is implied by SCI's property not to guarantee in–order packet arrival. Therefore, write and read operations can overtake each other destroying the consistency constraints of models, which directly connect synchronization aspects with individual read and write operations. Examples for such consistency models are e.g. *Sequential* and *Processor Consistency* [86]. They can therefore not be implemented on top of the SCI-VM. Less restrictive approaches, like e.g. *Weak Consistency* [86], which introduce separate transfer and consistency operations can, however, be implemented within the SCI-VM, as the distinct consistency operations provide the necessary hooks to integrate the consistency enforcing mechanisms.

One further restriction is connected to consistency models which relate consistency to individual memory regions, like e.g. *Entry Consistency*. Such a scheme can not directly be transfered to the SCI-VM because the necessary invalidation or flush of the buffers can only be performed globally and not on selective address regions. It is, however, possible to provide a consistency model with the same guarantees to the application by replacing any selective flush or invalidation by a global one. While this does not exploit the full potential of optimizations possible with such a model, it still effectively provides a suitable execution environment for the respective applications.

Besides these hardware imposed restrictions, the SCI-VM system is in principle capable of implementing any kind of consistency model. In addition, the concrete consistency model is not fixed by the SCI-VM itself, but can rather be determined independently for every target programming model implemented on top of HAMSTER. This satisfies the main design goals of HAMSTER and contributes to the intent of enabling the implementation of as many programming models as possible.

4.7 SCI-VM Performance

As the SCI-VM forms the common core for any programming model implemented within the HAMSTER framework, its performance is crucial for any application utilizing the framework. This chapter therefore provides a first insight into its performance focusing on the low–level aspects and the impact of the memory optimizations discussed above.

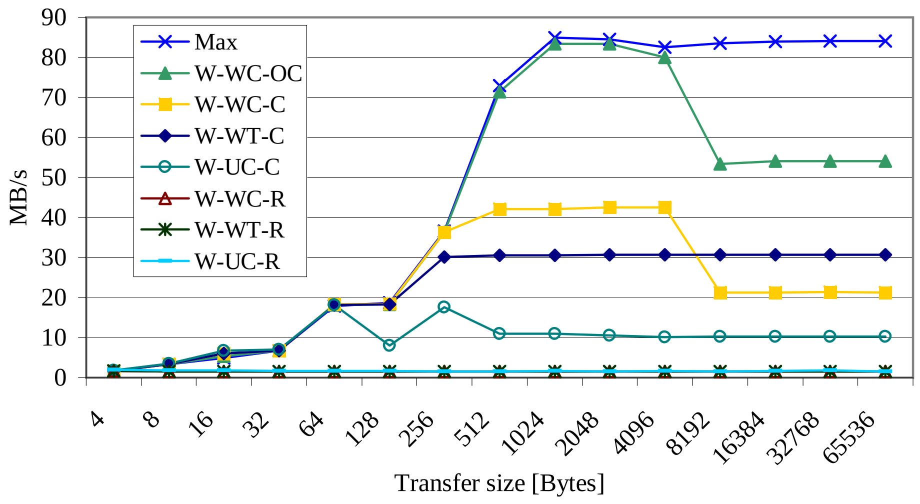


Figure 4.7 Write performance on SCI-VM memory.

Further experiments will be presented in the following chapters further highlighting the performance details of this core component.

4.7.1 Basic Memory Performance

The first set of experiments assesses the raw bandwidth available on remote memory allocated through the SCI-VM. This is done using various memory regions with different cache policies and with two different memory access patterns: consecutive and random accesses. The result for write operations is shown in Figure 4.7. In this figure, lines denoted as WT have been measured using write through caching (necessary to enable read caching), while WC denotes write combining and UC stands for uncached (the latter two do not contain any read caching). The letter at the end indicates the memory access pattern with C for contiguous and R for random accesses. In addition, the figure contains two further curves: Maximal and W-WC-OC. The former one stands for the peak bandwidth available on the cluster and was measured using the official SCI benchmark program provided by Dolphin. The memory model used for this experiment is also write combining, the default model of the IRM and the SISCI API. For the W-WC-OC curve the setting equals the W-WC-C curve, but the transfer loop includes an optimization in the form of write cache flushes which has proven beneficial to an optimized transfer rate and is also used within the Dolphin benchmark.

It should be noted though that this optimization is only effective in combination with write combining. Other cache or memory types do not profit from it; it sometimes is even counterproductive. In addition, it is also clear that shared memory applications can not be expected to perform such an optimization, as any access to global memory is transparent with regard to the physical location of the data. The latter curve hence represents only a theoretical experiment to contrast the performance to the theoretical peak.

This comparison shows equal behavior for transfer sizes of up to 4096 bytes (a full page) and reaches bandwidth values close 85 MB/s. For larger transfer sizes, the performance measured with HAMSTER drops significantly, even though both experiments are conducted on the same memory type and use the same optimizations. The difference comes from the fact that Intel CPUs [99] have two different write combining types, one set through the page tables and one set via the MTRRs. The latter, which is used by default within the IRM and the SISCI API and hence also for the Dolphin benchmark, does not infer any restrictions at page boundaries allowing for a smooth bandwidth curve. Within the HAMSTER system, however, this option can not be used as it would prohibit any caching of remote memory. Therefore, it is necessary to use page table based write combining which leads to the observed anomalies at page boundaries. A similar behavior is also visible without the use of the write cache flush optimization (W-WC-C), but at an overall lower performance.

This anomaly is limited to write combining and not present in the write–through and uncached cases. In the former one, a peak bandwidth of about 30 MB/s can be reached due to a limited support for write combining present in this cache mode, while the latter one achieves only a sustained throughput of about 10 MB/s. Higher transfer values can be achieved, however, for some of the smaller data transfer sizes which lead to favorable PCI burst transactions.

All of the measurements discussed so far relate only to consecutive addresses being accessed during the transfer. This is, however, quite untypical for shared memory applications as those do not use the memory for simple data transfers, but rather for random accesses to their data structures. In order to provide a more realistic scenario taking this behavior into account, the curves marked with R at the end show the bandwidth values for accesses to random remote locations. In such scenarios, no PCI burst or write gathering within the SCI adapter can be used, leading to N individual write operations. The bandwidth therefore stays roughly constant over all data transfer sizes at about 1 MB/s.

Using the same scenarios as above, also the read performance has been measured. For these experiments, additionally the SCI read optimizations for buffering and prefetching have been enabled for the WC and WT cases. The results for this are shown in Figure 4.8 and exhibit no large differences between the individual curves. In contrast to the write scenario, the consecutive and the random access behave almost identical since read operations are always blocking and therefore offer no possibility for pipelining.

Of special interest is the curve for cached accesses. Their performance is significantly lower for very small accesses since each access triggers the fetching of a full cache line. With rising consecutive transfer sizes, this can be utilized, leading to the best performance for transfer sizes equal to or larger than the cache line size of the first level cache (32 bytes). In the random case, cache lines can not be reused effectively, leading to the lower performance. With rising numbers of transactions, cached data can be reused leading to increased performance.

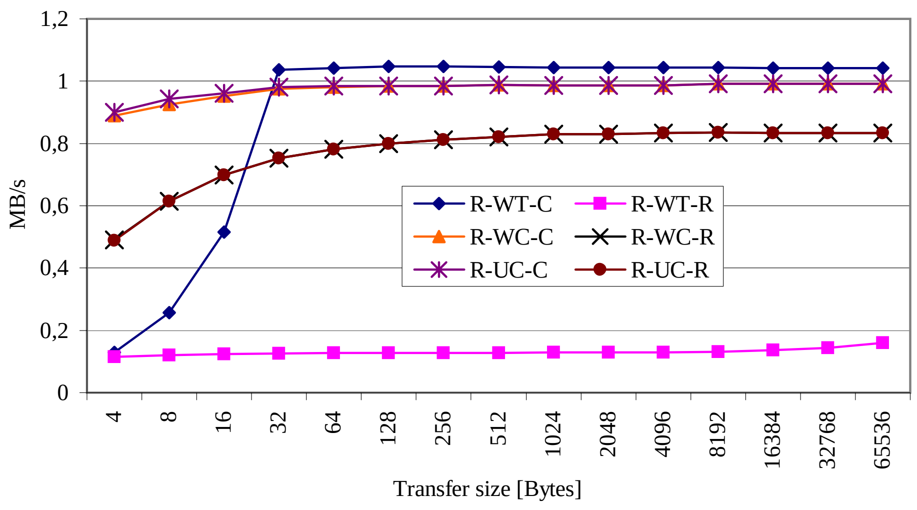


Figure 4.8 Read performance on SCI-VM memory.

4.7.2 Effect of Memory Optimizations

The bandwidth experiments discussed above, however, only indicate the raw performance of the underlying network infrastructure and do not allow a direct prediction of application performance. Hence, further experiments using kernels and applications are required. As a first step in this direction, the memory optimizations discussed above have been evaluated using two sets of experiments using small numerical kernels: an array sum and a matrix multiplication. The experiments were conducted on an older cluster in a reduced configuration consisting of two PCs with 233 MHz Pentium II™ processors running Windows NT™ 4.0, and connected via Dolphin's PCI–SCI cards (D308, PSB Rev.B). In all experiments, only one of the two nodes is used for computing; the second node can be seen as a memory server supplying remote memory pages.

Evaluating prefetching capabilities: array sum

The first experimental kernel computes the sum of a 256 Kbyte array by using a standard loop. All data is accessed sequentially and no data is reused. This experiment shows the worst case behavior of a transparently distributed program since 50% of all data is remote. In addition, all remote data is accessed and therefore needs to be transferred.

The program was tested in several versions with different optimizations: both SCI optimization techniques, speculative buffering and aggressive prefetching, have been applied individually and combined each with both caching disabled and enabled. The results can be seen in Table 4.1. As expected, the unoptimized version of the program shows a dramatic overhead as the number of remote reads compared to local reads and computation is extremely high. The code therefore runs more than a factor of 60 slower.

Significant performance gains can be achieved by enabling the prefetching capabilities of the SCI adapter cards. Especially in this example, prefetching techniques are extremely

	without caching		with caching	
	total [ms]	overhead	total [ms]	overhead
Local memory / baseline	-	-	2.6	0.00%
No SCI optimizations	164.2	6152.52%	23.1	801.05%
Speculative buffering	25.5	869.44%	13.8	437.62%
Aggressive prefetching	279.8	10555.07%	20.9	716.17%
Both optimizations	16.2	518.40%	4.8	88.31%

Table 4.1 Performance data for low–level experiments: array sum algorithm.

	without caching		with caching	
	total [ms]	overhead	total [ms]	overhead
Local memory / baseline	-	-	62.2	0.00%
No SCI optimizations	15875.2	25414.67%	67.9	9.43%
Speculative buffering	12426.6	19872.20%	65.9	6.21%
Aggressive prefetching	28261.2	45321.63%	68.3	10.09%
Both optimizations	21776.2	34898.94%	65.3	5.35%

Table 4.2 Performance data for low–level experiments: matrix multiplication algorithm.

effective as the program traverses all memory in a linear fashion. The data also clearly shows that aggressive prefetching alone does not help at all; it is even counterproductive. This is most probably due to the fact that data that is being prefetched can not be utilized directly by the program since it needs other data first. The prefetching, therefore, becomes a pure waste of bandwidth. This scenario changes with buffering activated. Now all necessary data that is used before the data from the prefetched block is needed is being loaded together with the first load in a 64 byte memory range. Due to this the prefetched data can actually be utilized and the program performance is increased compared to only applying speculative buffering.

Enabling caching of remote memory also boosts the performance significantly. This effect is achieved by implicit prefetching of whole cache lines. It is, however, less effective than speculative buffering and aggressive prefetching using the SCI hardware because the PCI–SCI adapter triggers prefetching of consecutive blocks whereas the prefetching through caching is limited to single cache lines. However, combining the two techniques provides the best performance as the two prefetching levels efficiently complement each other. The result is an overhead factor of less than 100% which leaves room for speedups when the computation is distributed as well.

Temporal and spatial locality: matrix multiplication

The second experiment implements a standard matrix multiplication. Both source matrices and the result matrix are distributed. The total working set of this program is also about

256 Kbytes and therefore still fits into the L2 cache of the Pentium II™ . This program also exhibits a rather large number of remote reads compared to the amount of local computation.

The same experiments as with the code above were conducted and the results are shown in Table 4.2. It can clearly be seen that using distributed memory via SCI causes a severe performance degradation when it is used transparently without caching. Unlike in the sum experiment, not even the SCI optimizations are able to improve the performance to a point that a parallel execution on several nodes would deliver a speedup. Comparing the SCI optimizations with each other, a similar picture than with the sum code can be seen. Speculative buffering alone improves the performance drastically, while aggressive prefetching decreases performance due to increased network traffic. Unlike in the sum experiment, however, both optimizations combined perform worse than even the unoptimized case. This is due to the fact that a large part of the memory accesses in the matrix multiplication code do not exhibit a single, large linear stride through the memory. Aggressive prefetching therefore does not help significantly because the stream buffers on the SCI adapter cards that hold the prefetched data are reused for other remote reads before the prefetched data can be used.

Only after enabling caching for remote shared memory segments, an acceptable performance can be achieved. With any of the SCI optimization levels, the overhead is lower or equal 10%, i.e. utilizing the global SCI-VM only costs 10% of the total program execution time, thereby leaving room for excellent speedups in systems that utilize the computational power of several computing nodes. This can be attributed to the fact that the matrix multiplication algorithm exhibits a large degree of both temporal and spatial locality and therefore significantly benefits of caching.

Like in the case discussed above, speculative prefetching again improves the performance and decreases the overhead factor to around 6%. Aggressive prefetching alone, like in the non–cached case, again reduces the performance, but combined with speculative prefetching the optimal performance with an overhead of less than 5.5% is reached. This would allow for very good speedups in clusters of several computing nodes and shows that this concept is feasible and has the potential to exhibit good and scalable performance.

4.8 Remaining Problems and Challenges

The SCI-VM used within HAMSTER is currently implemented as a prototype. While this system provides the complete functionality needed for an experimental evaluation of the overall system, a few issues remain open. These are briefly discussed below.

4.8.1 Transparency Gaps in the Underlying Hardware

As the SCI-VM directly relies on the underlying SCI hardware, it is capable of directly exploiting its capabilities, but is also bound by its technical limitations. The main one stems from the fact that the SCI Virtual Memory is designed with the assumption of a fully transparent hardware DSM system. This transparency, however, is not fully present in the current SCI implementations [26]; both the PCI subsystem of current PC architectures and the PCI–SCI bridge can be the source of transaction failures.

It has to be mentioned, though, that this transparency gap is in both cases, for the PCI bus and the SCI bridge, are not a principal but merely a pure implementation problem. Both are based on standards, which guarantee the transparency expected by the SCI-VM, but the current hardware does not fulfill these standards. The most common problems include deadlock situations on the PCI bus resulting in packet loss and buffering problems in the SCI link chip, again resulting in unrecoverable packet loss. During the last years, these problems have constantly decreased due to newer PCI chipsets (currently the Intel BX/GX chipsets seem to be the least affected) and newer SCI generations (especially the new link chip generation with the LC-3 is expected to solve many of these problems).

However, until this development is completed and has led to hardware fully adhering to their respective standards, the SCI-VM and with it the HAMSTER system, has to live with these problems. As a result, the stability of the overall system is affected since shared memory programming models are not capable of tolerating such transmission errors due to the implicit nature of the communication[2]. The HAMSTER system is therefore at the moment only an experimental system exploring the shared memory capabilities of hardware DSM architectures without the option for a production quality system in the immediate future. The hope is, however, that this will change with either further advances in hardware development guaranteeing the transparency promised by the respective standards or with ports to other NUMA architectures.

4.8.2 Full Operating System Integration

As described above, the current version of the SCI-VM is based on the static memory management scheme. While this provides the intended experimental platform to evaluate the overall system, it has some limitations with respect to the resource management. One of the currently open issues is therefore to complete the integration of the SCI-VM into the virtual memory management of the OS and therefore the elimination of the problems mentioned above.

4.9 Summary

Distributed Shared Memory systems are one of the main prerequisites for the implementation of shared memory programming models on architectures without direct hardware support. They allow the emulation of a global virtual memory based on software mechanisms. Since their introduction with the IVY system [136] in the mid 80’s, a large variety of systems have been developed exploring the different aspects associated with them. This includes the exploration of various relaxed consistency models for the reduction of inter–node communication requirements as well as the investigation of additional hardware support for

[2]In contrast to explicit communication mechanisms, as given e.g. in message passing systems, where the successful completion of individual transmissions can be controlled and the transmission can be repeated in the case of an error.

their optimization. The latter one, however, has mostly been restricted to interconnection fabrics with remote write or update functionality, but without remote read capabilities.

By using a NUMA–based interconnection fabric, namely the Scalable Coherent Interface (SCI), this work can go a step further and deploy the HW–DSM provided by SCI directly for the creation of a global virtual memory. This HW–DSM alone, however, is not enough because it only relies on physical addresses and not on a virtual address space as required for a shared memory programming model. Therefore an additional software component is required. The result is a new type of DSM system called SCI Virtual Memory or SCI-VM, merging SW– and HW–DSM in a hybrid fashion: any communication is carried out directly in hardware without any protocol overhead while the memory and system management remains in software.

As this concept directly relies on the underlying HW–DSM, it is important to optimize its performance. For this purpose, various buffering and caching mechanisms available on both the local system and the SCI network adapter can be enabled. Experiments have shown that this has the potential to significantly increase the performance of applications running on top of the SCI-VM. These optimizations, however, have an impact on the memory coherency presented to the user since data is stored in the various buffers and updates are delayed or invisible due to stale cache contents. Users therefore are required to take this into account within their applications and use memory consistency enforcing mechanisms to guarantee a correct program executions. This leads the way to relaxed consistency models, as they are known from traditional SW–DSM solutions, However, based here on a radically different hardware/software implementation.

Chapter 5

HAMSTER Management Modules

The HAMSTER system is designed to accommodate as many shared memory programming models as possible. In order to accomplish this goal, the system provides a number of services beyond the pure shared memory abstraction implemented by the SCI Virtual Memory or SCI-VM. These services are grouped into a number of modules, including modules for memory and consistency management, synchronization, and task control. In addition, a separate cluster control module provides the base services needed for cluster management and for the creation and maintenance of the global process abstraction. This chapter introduces these modules, their functionality, and also discusses their relation to each other.

5.1 Memory Management

The first module discussed here is responsible for the management of the globally shared memory. It directly relies on the SCI Virtual Memory described in Chapter 4 and provides its capabilities to form a global virtual memory to higher layers allowing them to allocate and control this global memory. The control hereby ranges from memory coherency type selection to locality annotations during allocation time.

In addition, this module provides some control over static data contained within the application's binary. This data is by default not shared since it is allocated during the initialization of the application by the operating system without the chance for the SCI-VM to gain control of this data. The memory management module therefore provides mechanisms to identify this memory area and to make it available within the cluster in an either implicit or explicit way, as dictated by the intended target programming model.

5.1.1 State of the Art

Memory management for NUMA architectures, both with and without cache coherency, has been investigated in a variety of projects and products. Their main goal has been the improvement of data locality since a good data locality is the key requirement for good application performance on NUMA machines. Therefore, most CC–NUMA machines, which use their own operating system or extension thereof to provide a global system and memory management, contain some sort of automatic and transparent data locality optimization. Examples for this can be found in the IRIX operating system [39] for the

O2000/O3000™ series by SGI [131], which enables transparent thread and data migration, and in the FLASH architecture [128], which uses a sampling of the cache performance counters in the processors to optimize the data layout at runtime [223].

Also for pure NUMA machines, a significant amount of work in this direction has been done. The DUnX system (Duke University nX2) provides a general framework for the evaluation of different NUMA placement and migration policies [105]. It has shown that no single policy is optimal for all applications and the best scheme is highly dependent on each application's memory access pattern and locality behavior. A similar conclusion is also drawn by Bolosky et.al. [53] by using a trace driven simulation of several existing NUMA management schemes.

This observation has inspired work on application driven locality optimizations, which enable programmers to influence data placement decisions. Using this approach, semantic knowledge from the application can be used directly for the optimization process, but it also results in more coding complexity for the programmer. To keep this impact at a minimum, novel programming systems have been developed which aim at providing higher–level programming constructs for the easy specification of data locality. An example for this is the Tornado [59] NUMA operating system. This work, which is implemented for the Toronto NUMAchine [70], uses an object–oriented abstraction for this purpose. Programmers can form groups of objects, so–called cluster objects, and base the locality specification on these application and data set specific entities.

Data locality optimizations are also present in SW–DSM systems, however, mainly in an explicit form. This means that the programmer is required to either explicitly place data parts on certain nodes or to manually initiate data migrations. Only few systems, mainly those which combine DSM principles with a higher–level programming model, enable an implicit data distribution. The COOL system [32, 31], an extension of C++ for parallel programming based on a global memory abstraction, allows programmers to specify affinities between data and tasks to processing nodes in the form of optional hints. With their help, the locality of COOL applications can be optimized resulting in more efficient program execution.

Another example for such a system is the Global Array toolkit [168], an explicit shared memory programming environment. It is intended for applications working on dense matrices and bases its locality and data distribution policies on the programmer's data specifications on parts of matrices. Any matrix under control of Global Array is available from any node, but in contrast to true shared memory programming models users have to explicitly use library calls for data accesses. This allows a much easier software implementation and also renders this approach feasible in Wide Area Network (WAN) scenarios [167].

5.1.2 Functionalities of the Module

The functionality of this module is closely related to the SCI Virtual Memory layer discussed in Chapter 4. It allows to directly control the memory allocation process and the virtual memory regions that are part of the global memory abstraction. The routines provided by the module for this purpose are listed in Table 5.1. The main routine, *scivm_alloc*,

API call	Description
scivm_alloc[2]	Allocate a chunk of global virtual memory
scivm_getGlobalMem	Reserve a piece of configuration memory
memMod_shareStaticImpl	Implicit distribution of static application data
memMod_shareStaticExpl	Explicit distribution of static application data
memMod_getStaticAddr	Query addresses for explicit distribution
memMod_getStatistics	Query the current status of the collected statistics
memMod_resetStatistics	Query the statistics and reset them

Table 5.1 API of the memory management module.

deals with the allocation of global memory. It can be used in two different modes: while the first one allows to allocate global memory in a transparent fashion, i.e. in a round-robin manner at finest possible granularity[1], the second one gives the caller some control on how the newly allocated memory will be distributed among the nodes participating in the global process abstraction of the SCI-VM. This can help to improve performance in cases where the memory access pattern of the application is known, as discussed later in this chapter. In addition, the latter mode also allows higher layers to request certain memory coherency guarantees in the form of coherency types, which will also be discussed in more detail below.

Besides the allocation of new dynamic memory in a heap–like fashion, the memory management module also allows some control over the existing static data, which is part of the application's executable and provided at load time by the operating system's loader mechanism. Here both an implicit scheme, i.e. hiding the existence of node local versions of this data by extending the global memory abstraction, and an explicit scheme, i.e. providing explicit access to the individual static data segments on all nodes, are offered to higher layers. The appropriate distribution and coherence type, if any at all is needed, can then be chosen by the intended target programming model.

In addition, the memory management module is also gathering statistical information about the status of the global memory and therefore contributes to the HAMSTER monitoring interface mentioned in Chapter 3.4.2. Table 5.2 lists the information recorded during the application execution. This information can then be queried by higher layers within the HAMSTER infrastructure, either a specific programming model or an appropriate tool environment [110, 111]. The appropriate routines are also listed in Table 5.1.

5.1.3 Influencing the Memory Layout

As mentioned above, the memory management module gives higher layers the option to influence the distribution of the physical memory backing the newly allocated global virtual

[1]On currently supported systems consisting of commodity PC hardware this normally is page (4 Kbyte) granularity.

[2]The prefix *scivm* is used instead of *memMod* due to historic reasons.

[3]This value is always zero if the static application memory is shared neither implicitly nor explicitly.

Variable	Purpose
numAlloc	Number of times *scivm_alloc* called on local node
staticSharedImpl	Static application data shared implicitly
staticSharedExpl	Static application data shared explicitly
virtMem	Total amount of memory under the control of the SCI-VM
allocMem	Amount of global memory requested by the local node
physMem	Amount of local physical memory used by the SCI-VM
allocStatic	Size of static memory under SCI-VM control[3]

Table 5.2 Statistical information collected by the memory management module.

Pattern	Parameters	Description
TRANSPARENT[4]	—	Distribute memory at finest possible granularity
FULLDIST	*node*	Use physical memory on *node*
BLOCKDIST	*node*	Distribute the memory in blocks put block 0 on *node*
CYCLEDIST	*node,size*	Distribute chunks of *size* in a round robin fashion, put chunk 0 on *node*

Table 5.3 Memory distribution types available for locality annotations.

memory during the allocation process. This is done on the basis of locality annotations that are passed to the system along with memory allocation requests. The newly allocated physical memory used to back the requested global virtual memory will then be allocated as specified in the annotation.

The user currently has the option of four different basic distribution types. These are listed together with their respective parameters in Table 5.3 and visualized in Figure 5.1. With these four types, most of the commonly needed distribution options can be described. In addition, it is planned to allow the user to hand tune the distribution by providing the system with a concrete distribution list containing the information about where each page of the newly created memory region should be located. Such an option would, however, be very cumbersome to use and is intended more for the implementation of higher–level programming models deploying parallelizing compilers or other means of code generation.

It is worth noting that these extra locality parameters are here referred to as annotations as they merely guide the allocation of resources among the cluster but do not influence the actual functionality of the system. No matter which annotation is passed to the memory management module, the system always returns with a newly allocated piece of virtual memory accessible from any node within the system; merely the access times to the memory can vary as different memory layouts lead to different locality properties. This mechanism can be used for an easy and safe incremental performance optimization;

[4]This option is also used if no locality annotation is provided at allocation time.

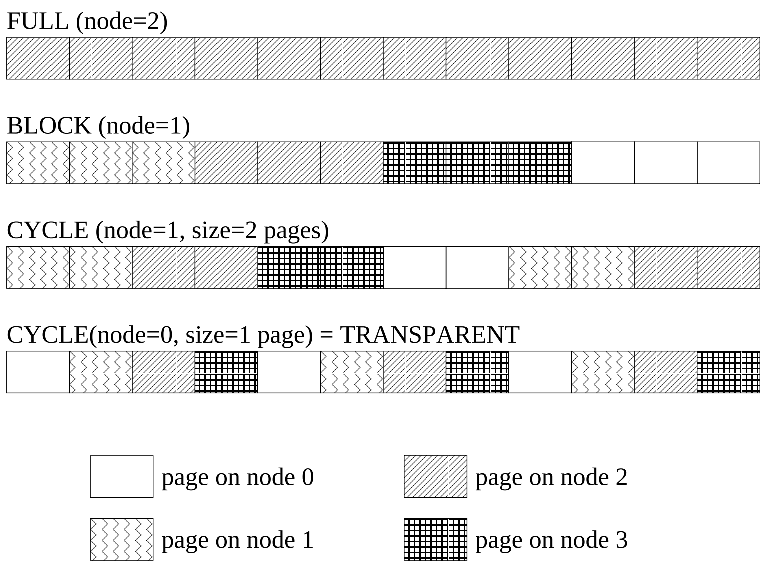


Figure 5.1 Examples of memory distribution types (based on a system with 4 nodes).

the application can first be ported to a HAMSTER–based system just using the transparent allocation mechanisms. Once this is completed and the correctness of the code has been verified, the performance can then be tuned by introducing locality annotations at the memory allocation points in the code. This no longer changes the functional behavior of the code, but instead allows for a fine tuning of the memory layout adhering to the application's memory access pattern.

Especially codes based on large regular data structures, like dense matrices, and with regular and static task distributions can profit significantly from locality annotations. This is illustrated in the following using two codes with these characteristics, more specifically a code performing an iterative Successive OverRelaxation (SOR) and a code implementing a Gaussian Elimination with pivoting. The pseudo code for both kernels is shown in Figure 5.2. It can be seen that, in both cases, each node is responsible for a subpart of the total matrix. This property can be used to optimize the data layout by distributing the data in a way that all matrix rows are located on the node responsible for the respective row. In the SOR code, this leads to a regular block distribution and for the Gaussian elimination to a block–cyclic distribution with block sizes equal to the size of single rows.

Table 5.4 shows the performance for both codes with two different matrix sizes each using both a transparent and the optimized data layout discussed above. The data shows a significant improvement that is gained by optimizing the codes through locality optimizations. Without optimizations, the performance is very poor and does not lead to any speedup while the optimized versions show the expected speedup. The speedup of the SOR code is thereby slightly higher than the one of the Gaussian elimination since the latter one requires a more fine grain parallelization and induces more global communication caused by the global pivoting.

```
Split matrix in #nodes blocks

for all iterations
   for all rows i in block rank
      for all columns j in row i
         compute matrix element (i,j)
      end for
   end for

   global barrier
end for
```

```
for all rows i
   if (i modulo #nodes) = rank
      compute pivot column
   end if

   global barrier

   for all rows j greater than i
      if (j modulo #nodes) = rank
         compute all elements in row j
      end if
   end for

   global barrier
end for
```

Figure 5.2 Pseudo code for the SOR code (left) and the Gaussian elimination (right).

	SOR code		Gauss code	
Matrix size	512x512	1024x1024	512x512	1024x1024
Seq. execution	1.36 s	5.58 s	4.83 s	40.23 s
Par. execution	78.16 s	1033.4 s	44.95 s	341.19 s
Speedup	0.0174	0.0054	0.1075	0.1179
Opt. execution	0.43 s	1.57 s	2.11 s	13.33 s
Speedup	3.16	3.55	2.289	3.018
Improvement factor	181.8	658.2	21.3	25.6

Table 5.4 Impact of locality annotations on dense matrix codes (on 4 nodes).

The good performance of the optimized codes is also visible in the speedup numbers of these codes on up to 6 nodes, as shown in Figure 5.3. Both exhibit a significant speedup across all configurations used, with the better overall performance for the SOR code due to the reasons already discussed above. In addition, both codes perform better on larger matrix sizes, as can be expected due to the decreased impact of the parallelization overhead.

It should be noted, however, that such a steep increase in performance after applying appropriate locality annotations is not common among all applications. Only such applications with regular memory access patterns and data structures like dense matrices are suited for such a kind of optimization. In addition, the codes presented here are mainly based on small arithmetic operations on only global data without any computation performed on local data, thereby further increasing the potential impact of locality annotations. On the other hand, applications with heavily irregular and unpredictable memory access distributions and/or local computations are either no candidates for such static distributions or do not require them in order to provide sufficient performance.

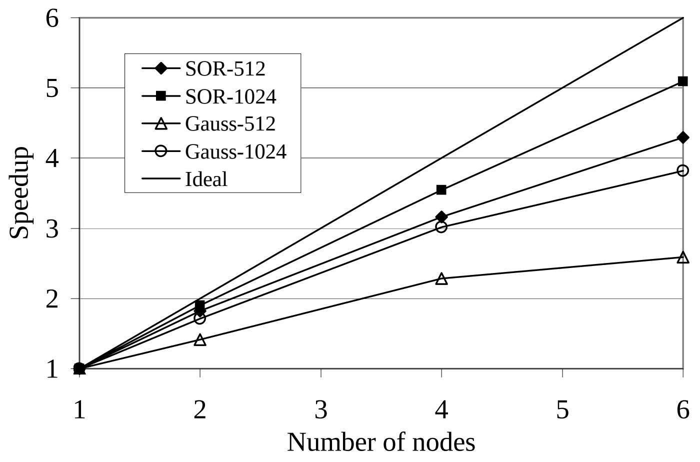


Figure 5.3 Speedup of dense matrix codes using locality annotations.

Type	Description
NONE	No buffers or caches activated
BUFFERED	Activate write buffers or read prefetch buffers
CACHED	Activate caches
BOTH	Combine the two types above (default)

Table 5.5 Memory coherency types.

Other prerequisites for the use of annotations are that the memory access pattern has to be known a priori, i.e. before the execution of the application, and the programming model used by the application has to export this capability to the application programmer. In the case where one of these prerequisites is not fulfilled, the locality annotations of the memory management module can obviously not be used. In order to still allow for some locality optimizations, a dynamically adaptable runtime system is necessary which is capable of detecting areas with bad locality and removes them by appropriate data and/or thread migration in cooperation with the underlying SCI Virtual Memory. Such a system is currently under development within the SMiLE project based on a hardware monitoring approach [110].

5.1.4 Controlling the Coherency of Global Memory

As mentioned in Chapter 4, the SCI Virtual Memory enables non coherent caching of remote memory with a non–CC NUMA architecture. This, however, does not mean that remote memory should always be cached; for some regions of global memory, e.g. configuration address space or memory regions with mainly write accesses, it is beneficial to avoid caching and hence to decrease the amount of necessary consistency enforcing mechanisms at programming model or application level.

For this purpose, the memory management module allows the specification of so–called memory coherency types, i.e. the coherency guarantees that are expected from a certain piece of memory. Table 5.5 shows the main attributes available for this purpose. Each of these can be used for both read and write statements. The strongest coherence type disallows any caching, i.e. the utilization of any processor cache, or buffering, i.e. the deployment of write and read prefetch buffers as well as potentially available streaming mechanisms. This coherence type can then be incrementally weakened by enabling either buffering or caching, or both.

Any of the above choices is implemented within the memory management module in close cooperation with the SCI-VM by enabling or disabling the appropriate resources in the system's hardware. This can range from mechanisms in the network adapter to caching policies within the CPU. It depends therefore strongly on the capabilities of the underlying system and can lead to situations in which the requested coherence type can not be provided by the system, but only a more coherent memory type can be offered to higher layers. One typical example for such a scenario is the caching for write instructions, which would lead to write–back caching . In the PC architecture, this causes timing problems on the PCI bus [101] and leads to a deadlock of the complete system, effectively crashing the machine.

In addition, certain guarantees may not be implementable without any side effects on the memory coherency of other memory areas which are either already allocated or will be allocated at a later time. An example for this is the buffering of both reads and writes within SCI adapters. While the latest adapters allow the usage of such buffering mechanisms on a per ATT basis and can therefore be specifically tailored to only influence specific memory regions, older adapters only allow these options to be set globally influencing the whole global memory within this system.

To allow higher layers control over these two scenarios, an additional argument can be passed to the memory management module specifying the behavior in these situations. Table 5.6 lists all options for this argument. In cases where *EXACT* is used, only memory allocation requests will be granted and executed which can be handled in a way exactly adhering to the requested specification and without any global side effects. This can be weakened by *ALLOW_STRONGER* or *ALLOW_GLOBAL* in any one of the two directions or both at the same time using *BOTH*. In case such a weakening of the memory coherency is performed by the system, the caller of the allocation routine is informed about this on return of the successful memory allocation.

Combined, these two additional arguments controlling memory coherency give any higher layer using them full control over the allocated memory without requiring any external knowledge about the underlying system. The memory management module is aware of the constraints on the target system and matches them with the requested attributes for newly allocated memory. This mechanism provides a maximum of comfort with the least restrictions on the available coherency types.

Type	Description
EXACT	Only allow exact matches
ALLOW_STRONGER	Also include stronger memory coherency types, but no global side effects
ALLOW_GLOBAL	Also allow global side effects while setting coherency type, but no stronger types
ALLOW_BOTH	Both of the above (default case)

Table 5.6 Parameters to control the handling the coherency type request.

5.1.5 Dealing with Static Application Data

Any of the mechanisms discussed above applies only to memory allocated through and under the control of the SCI Virtual Memory. The memory area reserved for the static application data, i.e. the data implicitly contained in the binary, is not affected by this and therefore only available locally on each node. It is solely controlled by the local operating system instance and allocated by the application loader and/or the dynamic linker. A global view of this data is therefore not possible with the mechanisms presented so far.

While this is sufficient for many shared memory programming models, especially for the programming models exposed by most software DSM systems, some programming models require access to this static application data. The best example for this are standard thread programming models present in almost all current operating systems. In these models, all data, static and dynamic, is shared between the individual threads. When implementing such a model on top of HAMSTER, the static memory needs to made available to any other node.

The HAMSTER memory management module provides the necessary base functionality to realize this requirement and therefore opens the HAMSTER system also to those programming models. It should be noted though, that this functionality is not applied by default in order to maintain to maximum amount of flexibility. Any programming model implemented within HAMSTER has the choice on whether static data should be accessible to other nodes, and if so, how it should be available. For this purpose, two basic schemes are offered: an implicit and an explicit distribution. Both are explained in more detail below.

Identifying static application data

Before being able to distribute the static data, it is first necessary to identify the appropriate memory regions. Hereby, first the actual data part of the application's memory footprint needs to be found and then this data part needs to be further subdivided into static data allocated for the application, i.e. static data from the actual application modules, and data allocated for the runtime system. Only the former part should be affected by any routine within the memory management module dealing with static application data, as this is part of the actual application. The runtime data needs to be kept local and separated on each

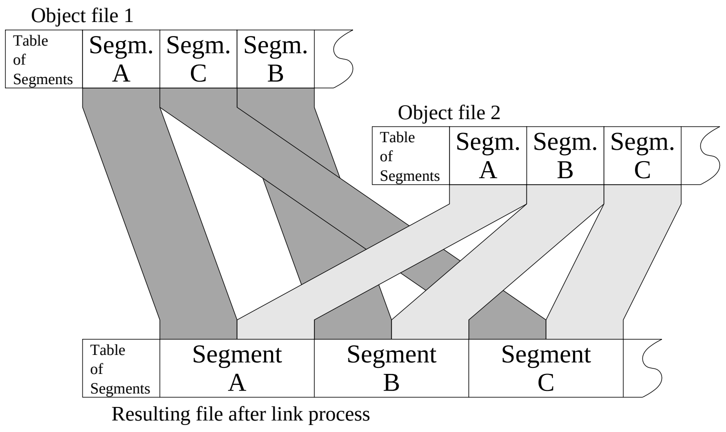


Figure 5.4 Segments within the file format and linking process.

node at all times, as the application execution is performed on separate operating system instances and therefore also separated runtime instances, which should not interfere with each other to ensure a correct program behavior.

The implementation of this identification process is based on the basic executable file format used in both Linux and Windows NT™ . In both operating systems, any executable or object file is formed by a series of segments, each with a specific content, e.g. code or data, and a unique identifier or name. In addition, each file is preceeded by a header containing a list of all segments within the respective file. In this list, all information needed to identify and locate the segment within the file is given.

An executable is created during the link process from various binary object modules, as depicted in Figure 5.4. During this linking process, the segments of all modules with identical identifier are merged into a single segment. At application load time, this complete binary is copied into the virtual memory address space of a newly created process. Using the header information, the data segments can be identified and also exactly located.

In order to further distinguish between static data belonging to the application and the data belonging to the runtime system, a closer look needs to be taken on the origin of the data. While the application static data has its sources in the data segments of the application object modules compiled by the user, the runtime data is added by the linker from separate runtime modules thereby to creating a complete application. To keep these two sources apart, a small patch utility has been developed which is capable of marking data segments of object files by slightly changing their names. This is accomplished by simply modifying the header containing the list of segments of the respective object file.

By applying this patch tool to any application object and marking each data segment therein in the same way, a new executable is created with two different sets of data segments, one marked, and one unmarked, as shown in Figure 5.5. While the latter one now contains all data of the runtime system which should not be touched by the memory management module, the former solely contains the static application data available for a further treatment by HAMSTER.

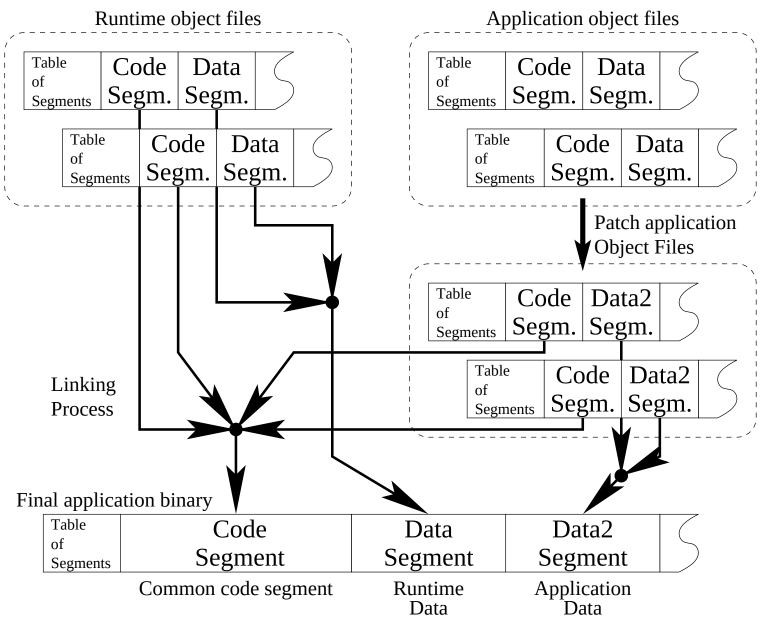


Figure 5.5 Linking process after patching the application's object files.

Providing implicit access to static data

Once the static application data has been identified in the way described above, the memory management module is capable of distributing it using either an implicit or an explicit scheme. In the implicit case, the static data is distributed across nodes and then made available at the original location of the static data. This effectively puts the memory area containing the static data under the control of the SCI Virtual Memory and thereby shares it among all nodes. This completes the transparency of the full virtual address space and allows the implementation of programming models relying on such a fully transparent memory, like the thread models briefly mentioned above[5].

The actual distribution is done in a three step process, as depicted in Figure 5.6. First, a global piece of memory with the same size as the memory area containing the static application data is allocated using the standard SCI-VM mechanisms. In the second step, the static application data is copied into the global memory area making it available to any node. The choice of the node from which the data is copied does not have any influence since the data is assumed to be the same at application start. In addition, it is also expected that the routine providing this implicit distribution is called before any changes to the static application data occurs.

In the third step, the actual mapping of the newly created global memory region into the original location of the static application data is performed. During this process, the initial virtual memory mappings are replaced by the mappings pointing to the memory region created by and under the control of the SCI-VM. Once these mappings, of which some

[5]More details on the thread programming models can be found in Section 6.4.

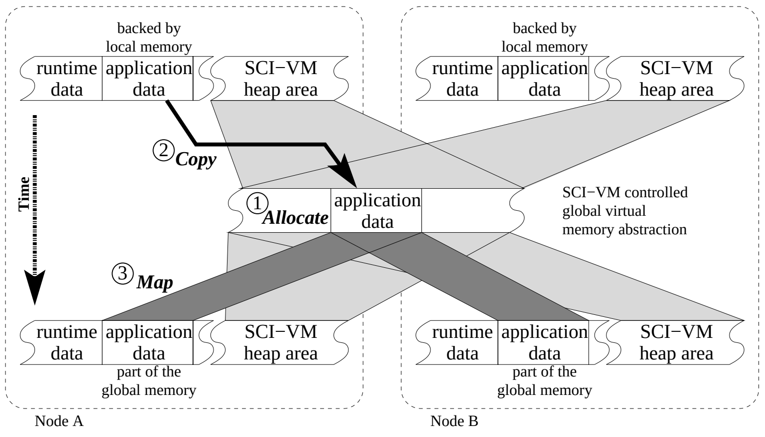


Figure 5.6 Implementing the implicit access scheme to static application data.

might be remote, are established, the application will access the globally shared copy of the static application data instead of the initial local version during its further execution.

This approach, however, is connected with some inherent implementation challenges. During the mapping, the memory management module needs to replace the virtual memory mappings of the virtual memory area containing the static application data. As this mapping was initially created by the operating system during the load time of the application, conflicts with the operating system can occur. This is especially valid for operating systems like Windows NT™ , which due to missing access to the source code do not allow a clean unmap of the virtual memory regions before applying the new mappings. Such an implementation has therefore proven, despite an implementation as proof of concept, as too unreliable for extensive experiments.

With an open source operating system like Linux, on the other side, a clean unmapping of virtual memory previously under the control of the operating system, can be accomplished, resulting in a stable system.

Providing explicit access to static data

Besides this implicit distribution, some programming models also require a more explicit distribution and access semantics to the static data. This means that the separation of the individual static segments is preserved while still allowing any node access to any other node's static data. An example of a programming model relying on this capability is the *put/get* model discussed in Section 6.5.1. In this model, the virtual address spaces are kept separate and the user is provided with primitives to copy data to (*put*) and from (*get*) any memory location on the other nodes' virtual memory[6].

The implementation of the explicit scheme is done in a two step process, as depicted in Figure 5.7. First, all physical page frames backing the static application data on all the nodes need to be pinned down in order to prevent them from being swapped out. This is

[6]More details about this programming model are listed in Section 6.5.1

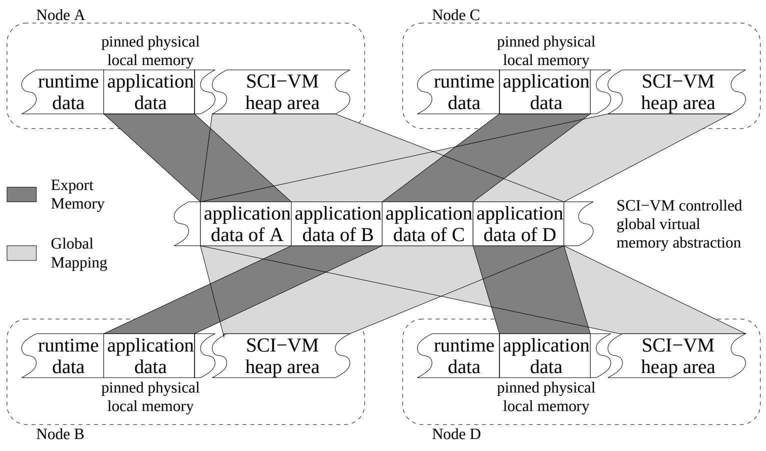


Figure 5.7 Implementing the explicit access scheme to static application data.

necessary to guarantee a safe and secure access from other nodes via the interconnection network. Once this is done, the memory area can be mapped via SCI to a newly created address space segment within the virtual address space controlled by the SCI-VM. This mapping is done in an all–to–all fashion, making every static memory area of every node available to any other node. The target mapping addresses are then made available to the upper layers allowing access to all the static memory areas of the whole system.

In contrast to the implicit scheme, this approach imposes fewer problems with the operating systems, as mappings initially done by the operating system are not overwritten during the mapping process. However, also here a direct access to operating system's memory management is necessary to perform the pinning of the physical page frames backing the application's static data on each node. Therefore, also the explicit scheme is only available in Linux since the available source code enables the implementation of this operation.

5.2 Consistency Management

As shown in Chapter 4.6, it is necessary to weaken the memory coherency in order to achieve acceptable performance. The SCI Virtual Memory therefore offers such capabilities which can be controlled through the memory management module as seen above. This control at allocation time alone, however, is not sufficient to allow both an efficient and correct application behavior on weakly coherent memory, as the memory state during the execution of an application is undefined and therefore most likely leads to an incorrect execution.

To compensate for this, additional mechanisms have to be provided that enable such a control through the application or the programming model. Using these mechanisms, a consistent memory state can be enforced whenever and wherever necessary. This enables applications to work on a consistent memory whenever required and to tolerate inconsistent, but potentially faster memory whenever consistency is not required for a correct execution. The consistency module within HAMSTER provides such mechanisms suited for all available coherency types. These are then at the disposal of any higher layer to be used for the required application and/or programming model consistency management.

5.2.1 State of the Art

Software controlled cache coherence in NUMA architectures has already been investigated in a few projects. Within Platinum [41] a coherent memory abstraction for a NUMA architecture is created using the virtual memory management to detect read or write accesses causing potential cache inconsistencies in coordination with a global page state information. A similar approach is taken in [176] and [126]. Both, however, do not exist for real systems and are evaluated only using simulation.

A different approach to deal with potential cache inconsistencies has been undertaken within the Shared Regions project [188], which defines a high–level abstraction to group and commonly manipulate memory parts. Based on this abstraction, it provides mechanisms to invalidate or flush remote memory regions and uses these mechanisms in coordination with synchronization primitives to guarantee a safe and reliable memory abstraction [187]. This approach avoids additional overhead for maintenance and hence represents a lean and easy–to–implement solution. This basic principle is therefore also used within this work to enforce a global cache coherency, however, in a more general and flexible framework independent of any higher–level data or programming abstraction.

5.2.2 Enabling the Control of the SCI-VM Memory System

As already discussed in Chapter 4.6, it is necessary to utilize relaxed consistency models in order to cope with the missing hardware cache coherence mechanism. They have to be capable of controlling the various buffers and caches in the system, the source of potential inconsistencies. The mechanisms necessary for this can roughly be divided into two categories: mechanisms to ensure that data has been propagated to other nodes and is not still being held in a local buffer and mechanisms to invalidate potentially stale data forcing the application to fetch new data from remote locations whenever appropriate.

The first group results in a full flush of all write buffers on the local node. Once called, the flush operation will block until all transactions currently stored in these buffers have been completed and the data contained within them has reached its final destination. This includes both the processor write buffers and those on the network interface. In the case of the SCI adapters used for the current version of the HAMSTER system, this affects the stream buffers implemented on the PCI–SCI bridges, as they hold data before their transmission in order to perform an aggregation of write transactions on consecutive addresses.

The second group, the read invalidation, simply invalidates all copies of remote data present on the local node, guaranteeing that the next read access to a particular memory location fetches its data from the physical storage of the data and does not shortcut to potentially outdated local data held in caches or other buffers. Like in the case of the write flush discussed above, resources on both the processor and the network interface are affected; the complete cache hierarchy within a node and any buffer used for read buffering or prefetching within the SCI interconnection fabric has to be invalidated.

It is important to note that all of these mechanisms have a purely local effect since they just deal with flushing or invalidating local buffers and caches. None of the routines require a remote node to be involved and therefore can be used fully asynchronously within the whole system. In addition, their implementation is quite straightforward and does include any explicit communication or other form of synchronization. On the downside, however, this property leads to the fact that these mechanisms alone do not suffice for a global memory synchronization required for a safe and reliable program execution. For this purpose they need to be combined with global synchronization mechanisms resulting in relaxed consistency models, which will be discussed later in this work.

5.2.3 Minimizing the Global Impact

These consistency enforcing mechanisms should be used with great care, because they affect the whole memory system on the local node and hence have a drastic impact on the performance of any application. Especially the invalidation of the local caches can affect the application, as not only the memory locations with stale data are invalidated, but rather the whole memory range containing remote data. This also leads to the invalidation of still valid data. This problem is further increased on architectures that do not allow partial cache invalidations, like the Intel x86 (TM) architecture [99]; there the complete cache, including the local data, needs to be invalidated on each write flush operation.

Due to these performance implications, it is important to avoid any unnecessary cache invalidation as far as possible. In this context, unnecessary invalidations can be classified as those, which do not lead to new data being fetched from remote locations and therefore invalidate only up–to–date data. In applications which are solely based on the consistency enforcing mechanisms discussed in this section[7] these can be avoided by observing remote write flushes. Only these indicate that a remote node purposely intended to make its local writes available and expects its data to be present on all other nodes. Therefore, any cache invalidation without a prior remote write flush since the last invalidation on the local node will not produce any new data. It is therefore unnecessary and can be omitted.

To implement this scheme it is necessary to provide a global time stamp identifying the temporal relation between cache flushes and write invalidations. Using the special mechanisms of SCI in form of atomic *fetch and increment* operations, such a time stamp mechanism can easily be implemented based on a global system–wide counter, in the following denoted as *Global Activity Counter (GAC)*. Each read invalidation or write flush is tagged with such a time stamp by reading this counter followed by an atomic incrementation. Before a cache invalidation, these time stamps are checked to detect unnecessary invalidations, which are then omitted.

[7]As opposed to applications or programming models which also utilize several memory coherency types with additional implicit guarantees.

More precisely, an optimized write flush consists of the following steps:

- Perform actual write flush
- Get new time stamp from the *GAC*
- Save the time stamp in the global, system–wide variable *Last Global Flush (LGF)*

A read invalidation uses this information in the following manner to avoid unnecessary invalidations:

- Check whether there has been a flush since the last invalidations by comparing the time stamps in *LGF* and in the node–local variable *Last Local Invalidation (LLI)*
- If *LGF* has an earlier time stamp than *LLI*, the invalidation is unnecessary and can be omitted
- Get new time stamp from the *GAC* and store it in *LLI*
- Perform actual read invalidation

An effect of this optimization is that the purely local operations are turned into global ones by requiring the use and maintenance of global state. The impact of this, however, is quite low as is visible from the performance numbers discussed later in this section, as only very few global operations need to be performed. In addition, those can be implemented using the underlying NUMA architecture and its remote memory access capabilities without the need to interrupt the execution on any other node. This ensures that the implementation of the optimization scheme does not lead to a drastically more complex code and/or a reduced performance.

5.2.4 Introducing Scope

The optimization described above creates an implicit relation between flushes and invalidations. Only invalidations following prior flushes need to be performed. So far, this relates to any flush within the whole system. This observation can be used to further optimize the routines of this module. By binding sets of flush and invalidate operations into groups, the preceeding write flush responsible for triggering an invalidation can be restricted to only those being in the same group as the invalidation in question.

These groups, which are also known as *Consistency Scopes* [97], implicitly restrict consistency enforcing mechanisms only to those parts of the global data which is protected by the consistency operations within the respective group. Read invalidations only guarantee the visibility of new data from remote nodes which have been flushed to the system within the same scope. This is sufficient for many application scenarios since read invalidations are often used to guarantee the visibility of new data in specific regions. In this case, the invalidation is carried out within the same scope as the flush associated with updates to this region of interest, avoiding unnecessary cache invalidations in cases in which no data of interest has been written by any remote node.

Using this extended scheme, the scoped version of the write flush is performed as follows:

- Perform actual write flush
- Get new time stamp from the *GAC*
- Save the time stamp in the global variable *LGF*, which is distinct for each scope (*LGF–scoped*).

A scoped read invalidation then uses this information in the following manner:

- Check whether there has been a flush since the last invalidations by comparing the time stamps in *LGF–scoped* and *Last Local Invalidation (LLI)*
- If *LGF–scoped* has an earlier time stamp than *LLI*, the invalidation is unnecessary and can be omitted
- Get new time stamp from the *GAC* and store it in *LLI*
- Perform actual read invalidation

Scopes have to be allocated by the user of this module and can then be supplied to the consistency enforcing routines as a parameter. However, it has to be possible to disable the scope optimization to guarantee full flexibility. A compatability to the simpler optimization scheme discussed above can be achieved by just using a single global scope, which is provided by the system itself. This leads to the exact same system behavior because the decision on whether to perform a read invalidation will again be based on the globally last write flush.

5.2.5 Functionalities of the Module

The complete functionality of the module is shown in Table 5.7. The main routines exported are the already mentioned read invalidation and write flush operations. In addition, a separate routine combines these two into a single memory synchronization routine. All of those routines are based on the idea of scopes introduced above which are passed to the routines as an argument. The last routine in this module enables the allocation of further scopes which can then be used with any of the other operations.

In order to ease the use of the scope concept in programming models without consistency scopes, the consistency management module introduces two additional constants (see also Table 5.8). They allow the specification that no scope should be used, i.e. the no consistency enforcing mechanism is avoided, or that a single global scope has to be applied, which is based on all consistency operations in the overall system.

API call	Description
consMod_syncRead	Invalidate all read buffers including caches (i.e. *Acquire* operation)
consMod_syncWrite	Flush all write buffers (i.e. *Release* operation)
consMod_sync	Both routines above merged together
consMod_allocScope	Allocate a new consistency scope

Table 5.7 API of the consistency management module.

Type	Functionality
CONSMOD_NOSCOPE	Always perform flush or invalidation
CONSMOD_GLOBAL_SCOPE	Perform flush or invalidation if necessary with respect to last operation
Allocated scopes	Perform flush or invalidation if necessary with respect to last operation on the same scope

Table 5.8 System defined scopes.

As the memory management module, the consistency module also collects on-line data to contribute to the overall HAMSTER monitoring interface. It delivers detailed information of both the numbers of consistency operations called by the higher layers and their associated delay times. The concrete information recorded is listed in Table 5.9. It purposely includes only data about the individual consistency mechanisms and no combined data reflecting the performance of a particular consistency model since the implementation of such a concrete model is the task of the programming model and should not be restricted by this module.

Variable	Purpose
numAcq	Number of calls to *consMod_syncRead*
timeAcq	Sum of time spent in *consMod_syncRead*
maxAcq	Maximal time spent in *consMod_syncRead*
numAcq	Number of calls to *consMod_syncWrite*
timeAcq	Sum of time spent in *consMod_syncWrite*
maxAcq	Maximal time spent in *consMod_syncWrite*
numFull	Number of call to *consMod_sync*
numScope	Number of scopes allocated

Table 5.9 Statistical information collected by the consistency module.

Operation	Execution time	Scoped time	Overhead
Full enforcement of coherency	481.97 μs	490.47[8]μs	8.35 μs
Invalidate read buffers	470.85 μs	475.36[8]μs	4.48 μs
Flush write buffers	8.60 μs	12.19 μs	3.57 μs

Table 5.10 Cost of consistency enforcing mechanisms (measured on one dual processor node of the SMiLE cluster).

5.2.6 Low–level Performance

The cost associated with the consistency enforcing routines is shown in Table 5.10. In the left column, the times for the standard unoptimized versions are listed, in the middle column the times for the routines together with the global operations necessary to manage scopes, and in the right column the resulting scope management overhead. The numbers show that the execution time of the routines is quite low due to their low–level and hardware oriented implementation. The cost for the write flushing is significantly lower than the cost for the read invalidation because only resources with a limited capacity, i.e. the processor write buffers and the SCI stream buffers, are affected. For the read invalidation operation, on the other side, all caches on the local node (in the case of an SMP node the caches from all processors) need to be flushed affecting significantly more resources.

It should be noted that the latter routine can in addition incur a slowdown of code executed after the read invalidation statement as caches and other read buffers have to be reloaded. This is not included in the actual cost portrayed in the figure. The impact of this depends on the application and its memory access pattern and can only be evaluated in a case by case process.

As also can be seen in the table, the overhead associated with the scoped versions of the routines is very low and just reflects the time needed to compute and store the time stamps. This has to be seen contrasted to the potentially large performance improvement by avoiding cache flushes. The concrete benefit, however, can again be only evaluated based on concrete applications.

5.2.7 Realizing Consistency Models in HAMSTER

The mechanisms introduced above can be used in the realization of a concrete memory consistency model, either for a specific programming model or for an individual application. However, as memory consistency between several nodes or threads is by nature an issue of global state, routines with only a local effect, as those offered by the HAMSTER consistency module, are clearly not sufficient for the implementation of a full memory consistency model. They need to be combined with appropriate synchronization constructs enforcing a time relation between individual invocations of consistency enforcing mechanisms. Together they can then form a complete consistency model.

[8]The numbers reflect the time needed for an invalidation operation that is not classified as unnecessary and therefore performed.

This approach is also taken by any DSM system (see related work on DSM systems in Chapter 4.1). Those, however, mostly statically merge consistency and synchronization mechanism, essentially hard–wiring a specific consistency model. The HAMSTER framework on the other side, with its intention to provide the ability to implement various different programming models each with a potentially different consistency model, does not implicitly perform such a merge between consistency and synchronization management. This is left to the implementation of the programming model on top of the management modules. A discussion of this, together with a description on how specific consistency modules are implemented, is included in Chapter 6.3 by means of specific programming models.

5.3 Synchronization Management

Any programming model based on the shared memory paradigm requires explicit synchronization primitives since the access to the global shared memory, and therefore any communication, is by default not synchronized. This can be seen in contrast to pure message passing models in which any message represents an implicit synchronization point; further synchronization is therefore not required in these models.

The synchronization module discussed in this section contains the necessary primitives to enable this required synchronization. It contains both high–performance implementations of typical primitives present in many programming models and low–level mechanisms to enable programming models with specific requirements to implement their own synchronization constructs. This two–sided approach provides the necessary performance and ease–of–use as well as the required flexibility expected from a system like HAMSTER.

5.3.1 State of the Art

The issues involved in the implementation of this module have to be seen from two different views. On the one side are the requirements imposed on the functionality of the synchronization module from potential programming models of HAMSTER and on the other side are the implementation aspects of the individual primitives.

With respect to the former view, the large variety of different programming models also leads to a large number of different synchronization primitives. Thread APIs normally include some sort of mutual exclusion in the form of locks and a wait and notification mechanism. In the case of POSIX threads [219], these constructs are part of the thread implementation and specification, while the Win32 API separates them from the actual thread calls and rather defines a system–wide synchronization framework allowing these mechanisms to act beyond the scope of a single process [155].

Some of the common thread APIs include further synchronization constructs, like the multiple reader/single writer locks present in the Solaris™ threads [148] or the concept of monitors [216] available in Java™ [104]. In addition, many shared memory programming models offer barriers as an easy–to–use tool to facilitate a synchronized application execution. Last, but not least, some programming models also rely on flag–based synchronization, i.e. on synchronization information being propagated simply by updating designated flag variables.

Regarding the implementation, any current processor architecture includes atomic read–modify–write primitives. These allow the implementation of efficient locks and other synchronization constructs on hardware shared memory multiprocessors in a very scalable manner [232, 150].

In distributed scenarios, more complex protocols have to be developed to compensate for the missing atomic update operations. In most cases, the only atomic operation available are load and stores which cause the implementation of synchronization mechanisms, especially locks, to be quite complex. First solutions for this scenario have been proposed by Dijkstra [43] and Lamport [130]. Further work has tried to optimize the synchronization by deploying token–based schemes and/or complex decentralized schemes based on various communication topologies. [103] provides a comparison of a few of these approaches. In addition, quite a bit of work in this direction has recently been done within the DSM–Threads project [165, 226].

A different approach to provide efficient synchronization to distributed architectures proposed by [153] is the extension of atomic hardware primitives. Using simulation, this work describes the opportunities for an efficient implementation of synchronization at the presence of such global atomic operations and proves their usability.

Most of these primitives are already present in the SCI Standard [92] as special packets and newer adapter generations also implement some of them in an easy–to–use fashion. Nevertheless, few projects have focused on utilizing this potential by implementing synchronization concepts for SCI–based systems. Within the prototype system SALMON [170, 171] at the University of Oslo, a survey of possible implementations of synchronization in the contexts of explicit messages and locks for mutual exclusion has been done. Both are based on simple read and write operations, not exploiting the potentials of the atomic operations offered by SCI, because those were not yet available in the SCI adapters at that time. In addition, no barriers have been included in the survey omitting this important synchronization mechanism for many programming models.

Within the HPPC–SEA DVSM system [40], which is intended to provide an easy–to–use interface for SCI–based architectures, locks have been implemented directly based on SCI's remote memory mechanisms using a centralized, token–based scheme. Also here, none of the atomic operations available in SCI have been used and the concept of barriers is missing.

5.3.2 Functionalities of the Module

The synchronization module encapsulates all synchronization mechanisms available to the programming models implemented within the HAMSTER framework. This includes both efficient implementations of typical shared memory synchronization constructs like locks and barriers, but also additional low-level mechanisms which can be used to implement further synchronization constructs for certain programming models. The implementation of this module itself is based on the local operating system interfaces available for syn-

API call	Description
syncMod_allocLock	Allocate a global lock
syncMod_lock	Acquire a global lock
syncMod_unlock	Release a global lock
syncMod_allocBarrier	Allocate a barrier
syncMod_barrier	Perform a barrier over one thread per node
syncMod_activityBarrier	Perform a barrier over all global threads
syncMod_fixedBarrier	Perform a barrier over a number of threads specified as a parameter
syncMod_allocCounter	Allocate a global counter
syncMod_allocCounterBlock	Allocate a block/array of counters
syncMod_incCounter	Increment a global counter and return the old value
syncMod_getCounter	Query the value of a global counter
syncMod_setCounter	Set the value of a global counter
syncMod_intSetCallback	Register interrupt callback
syncMod_intTrigger	Trigger a remote interrupt

Table 5.11 API of the synchronization management module.

chronization management and on a low–level access for the direct exploitation of SCI's features. It also profits from some configuration mechanisms within the SCI-VM and uses them for its own configuration and storage of global state.

The core routines provided by this module are listed in Table 5.11. They can be grouped into four sets: locks, barriers, global counters, and interrupts. The first three groups consist for one of the appropriate allocation routines and on the other side of routines providing the actual functionality. The last group, the interrupt routines, does not require an explicit allocation routine because all interrupts are established automatically during the module initialization. Therefore, the interrupt interface consists of only two routines: one to register a callback function and one to trigger a (potentially remote) interrupt.

In the following section, the individual functionalities of these four groups will be introduced, together with implementation details. Also the base performance for the individual synchronization constructs is presented.

In addition to the actual synchronization constructs, the synchronization management module also includes a monitoring component designed for on-line collection of statistical data. The information delivered by this component is shown in Table 5.12. It includes data about both locks and barriers and, similar to the consistency module, records both the number of calls to these routines and the time spent within them. The latter information is of special importance since these times mostly correspond to waiting times while either acquiring a lock or performing a barrier. A high time during these operations often indicates a serious performance bottleneck, which should be examined further.

Variable	Purpose
numLock	Number of global lock operations
numUnlock	Number of global unlock operations
numLockWait	Number of times a thread had to wait during a lock
timeLock	Sum of time spent during lock operations
maxLock	Maximal time spent during lock operations
numBarrier	Number of barriers executed
numBarFirst	Number of times a local thread has reached a barrier first
numBarLast	Number of times a local thread has reached a barrier last
timeBar	Sum of time spent during barrier operations
maxBar	Maximal time spent during barrier operations
numCounter	Number of global counters allocated
numCountInc	Number of incrementations of global counters
numIntOut	Number of interrupts generated on local node
numIntIn	Number of interrupts triggered on local node
timeIntIn	Time spent in interrupt handlers on local node

Table 5.12 Statistical information collected by the synchronization management module.

5.3.3 Implementation Aspects and Performance of Locks

One of the most important synchronization construct for shared memory programming are locks. They can be used to protect certain code or data regions by only allowing one thread at a time to acquire a specific lock. All other threads that try to acquire the same lock at the same time are blocked until the thread holding the lock has released it again. This provides the basis for the implementation of a mutual exclusion structure as it is often used to protect data structures from being manipulated by more than one thread at a time. Such a locking mechanism is provided by the HAMSTER environment through a simple interface. It consists of three routines: one to allocate a lock, one to acquire a lock, and one to release it again.

For an efficient implementation of such a locking mechanism on top of SCI, the atomic transactions provided by SCI can be applied. These transactions can perform read–modify–write operations on remote memory in an atomic manner without the possibility of races. In the Dolphin adapter used in this work, one of these transaction types is made available to the programmer through a special kind of remote memory mapping. When applied, a read operation through this kind of mapping triggers an atomic fetch and increment operation, i.e. increments the read memory location and at the same time returns its old value.

Using this atomic operation, a simple ticket scheme can be applied to implement locks (similar to [150]). Two counters are used for this purpose, one called the ticket and one called the access counter. When a thread attempts to acquire the lock, it increments the ticket counter using the atomic operation and gets the old value returned, which is saved as the ticket for the thread to enter the lock. After this, the thread continuously polls on the access counter until its value equals the ticket value. If this is the case, the lock is considered

	Simple Lock	Secure Lock	Overhead
Lock time	4.28 μs	5.43 μs	1.15 μs

Table 5.13 Cost of locking operations.

taken by the requesting thread. During the unlock operation, the thread releasing the lock increments the access counter and therefore grants the next thread access to the lock. This locking algorithm is not only efficient due to its simple implementation, but also provides fair access to the lock for any thread since it provides a true FIFO scheduling.

Unfortunately, this algorithm comes with a problem. Due to the current implementation of the PCI bridges and the SCI adapter cards, the execution of such an atomic increment operation can fail. This can lead to the loss and then to the repetition of a failing transaction resulting in an increment by two or more. In this case, which only happens rarely and can normally only be observed during stress conditions on the SCI network, the lock will get into a state in which the algorithm considers it taken even though no thread has acquired it. As a consequence, the application will deadlock.

In order to avoid this problem, a more complex scheme has to be employed that is secure and can not be harmed by atomic update errors. For this purpose, an additional reservation buffer with one slot for each potential lock acquiring thread is introduced next to the two counters. This buffer is then used during the execution of the lock operation to mark and order requests for the lock.

The actual locking scheme is based on the insecure method described above and also relies on tickets to be requested by the individual threads. It also relies on the fact that the atomic increment operation at least always returns a strictly higher value than the originally stored value, as in case of an error the transaction is carried out twice and only the result value from the second execution is returned.

When arriving at a lock, the requesting thread first acquires a ticket using an atomic operation (just as above in the insecure case) and writes the acquired ticket into the reservation buffer. Then the ticket is compared to the access counter and if the two values are equal, the thread has successfully acquired the lock. If the values are not equal, two things could have happened: either the value is incorrect due to an error or the value is correct and the thread really has to wait as the lock is currently taken by another thread. To distinguish these two cases, the minimal ticket value in the complete reservation buffer is computed and considered the new access counter. The waiting thread now continuously polls this minimal value. If no other thread is waiting, this minimal value will equal the own ticket and the thread can safely acquire the lock. In the other case, in which the lock is taken by a second thread, the minimal value will show the ticket of the other thread and the acquiring thread will not be able to continue.

A thread releasing a lock will, besides incrementing the access counter, also erases its ticket from the reservation buffer and therefore automatically triggers a new minimal ticket value in this buffer. This unblocks the next waiting thread currently polling on this minimal value and allows the continuation of its execution.

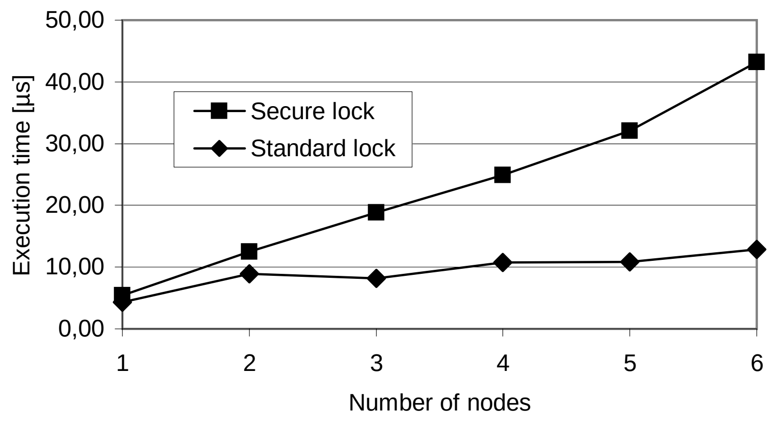


Figure 5.8 Comparing the two locking algorithms under contention.

The cost of the two locking algorithms is shown in Table 5.13. The numbers represent the time needed to do a full lock and unlock cycle together with a simple arithmetic operation. In both cases the actual locking time is very low and therefore causes only a small overhead for the application. Due to the more complex algorithm of the secure lock, the cost of this locking method is slightly higher. The difference, however, is only a little more than 1 μs. This small impact comes from the fact that in this experiment all locking operations are carried out by one thread without any lock contention to evaluate the pure locking overhead. In this case, only very little atomic update errors happen and there is never any waiting time for acquiring the lock. Hence, in almost all cases, the lock can be acquired directly without having to poll on the reservation slots thereby resulting in this small overhead.

The situation changes when locks are used under contention, as is shown in Figure 5.8. In this experiment, a lock–increment–unlock cycle was done concurrently by up to six nodes with a large number of iterations and the numbers shown are the total execution times divided by the number of iterations. It shows that the standard method performs significantly better, as the secure method in this contention scenario has to frequently fall back to the slow algorithm of finding the minimal ticket over all reservation slots. Nevertheless, without the guarantee of a working atomic increment, this overhead has to be taken into account in order to ensure a correct program execution. Due to this, in the remainder of the work only the secure version is used within the HAMSTER framework.

The experiments presented so far only discuss locks in scenarios with one process or thread per node. The results shown in Figure 5.9 complete this picture with experiments using several processes or threads per node. As the base performance, the graph shows the data for one thread per node (marked "Single team/thread"). The data points denoted as "Multi Team" present the performance of the same experiment, i.e. a lock–arithmetic operation–unlock cycle, for two processes on each node (which are dual processor SMP nodes). It can be clearly seen that this introduces a significant overhead which can mostly be attributed to the mutual influence between the two processes on the same node due to

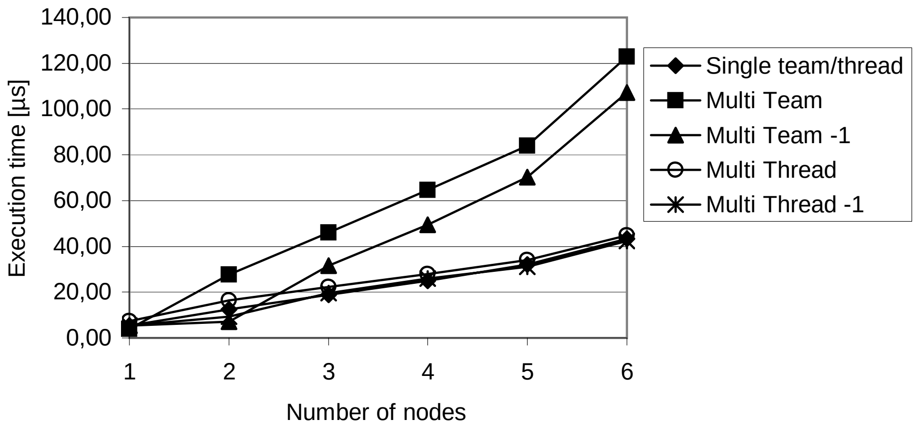


Figure 5.9 Lock performance under contention with secure algorithm in different SMP scenarios.

their global polling for the lock. This observation is further supported by the data shown as "Multi Team -1", which presents the data obtained with two processes on N-1 nodes and only one on the remaining node. These data points are roughly the same as in the "Multi Team" case with one node less. This indicates that this one extra process, which is located alone on a node, does not influence the overall performance significantly.

Based on the outcome of these experiments, it is obvious that such a concurrent polling of two processes on the same node should be avoided in order to decrease this drastic overhead. In the case of multiple threads, this can be done without much additional coding complexity by using local operating system locks in addition to the global locks. On locking, each thread first acquires the local lock and then the global one. This ensures that only one thread per node actively polls for the same global lock, while any other thread on the same node competing for the lock is blocked by the operating system trying to acquire the local lock.

The two lines marked as "Multi Thread" in the graph of Figure 5.9 show the performance results for this combination of local and global locks. The overhead is significantly reduced to a point where the difference between the performance of a lock with a single or multiple threads per node becomes negligible. Consequently, the performance with one node being loaded with one thread less also behaves almost identical.

5.3.4 Implementation Aspects and Performance of Barriers

Besides locks, a second very typical shared memory synchronization construct exists: barriers. These can be used to defer the execution of a set of threads until all of them have reached the same point of execution. This is done by all threads calling the barrier function which does not return until all threads included in a certain set of threads have called it with the same parameters. After that, the execution of all threads participating in the barrier can be resumed.

Barriers are often used to organize regular computations in an SPMD style. In these kind of programming models, all threads of an application execute the same computation on distinct sets of data. Barriers are used to separate different phases or iterations within a computation to hold the execution at the same level.

The main parameter of any barrier function is the number of threads which need to be synchronized. To provide the greatest possible flexibility, the HAMSTER synchronization module provides three different barrier routines with different ways to specify the number of threads:

- **Standard barrier**: Execute a barrier over all nodes with one thread per node. This is often used to coordinate management tasks which need to be executed on all nodes, like the distributed allocation of shared memory.
- **Activity barrier**: The HAMSTER environment maintains a counter for all running activities or threads. This barrier type is using this counter to synchronize all currently running activities.
- **Fixed barrier**: Execute a barrier over the amount of nodes specified as an extra argument. This routine takes care of all remaining cases in which the number of threads can not be determined implicitly.

All of these routines are based on the same core consisting of a single routine with the number of threads to wait for as the argument. Individual front ends are used to compute this number and call the core routine to execute the actual barrier.

In contrast to the locking scheme described above, it is not helpful to rely on the SCI atomic transactions for this routine. Due to the natural contention of a barrier, atomic transactions are likely to fail resulting in an insecure execution. In order to avoid this, an alternative has been developed that is based purely on standard remote memory operations. Further, only local polling is deployed to reduce the stress on the PCI–SCI bridge and to optimize the efficiency of the barrier mechanism. For this purpose, each node allocates its own small segment of local memory, from now referred to as local synchronization memory, and then maps these memory segments from all nodes into its own virtual address space. For each barrier that is being allocated, the synchronization routine reserves an integer slot for each node in each node's synchronization memory. On entrance to the barrier, each thread marks its arrival in the corresponding slot on every node. This is accomplished by using remote writes through SCI. Once completed, each thread only scans its local memory until all threads have signaled their arrival in the corresponding slots of the local synchronization memory. Once all threads have arrived, the barrier is reset and all threads resume their operation.

Figure 5.10 shows the performance of this barrier algorithm in the same scenarios as already used for the evaluation of the locks. The data points denoted as "Single team/thread" show the behavior when executed on up to six nodes with one thread per node. The execution of a barrier takes under 11 μs and only increases slightly when executed on more nodes. The algorithm is therefore very scalable with regard to performance, however, at

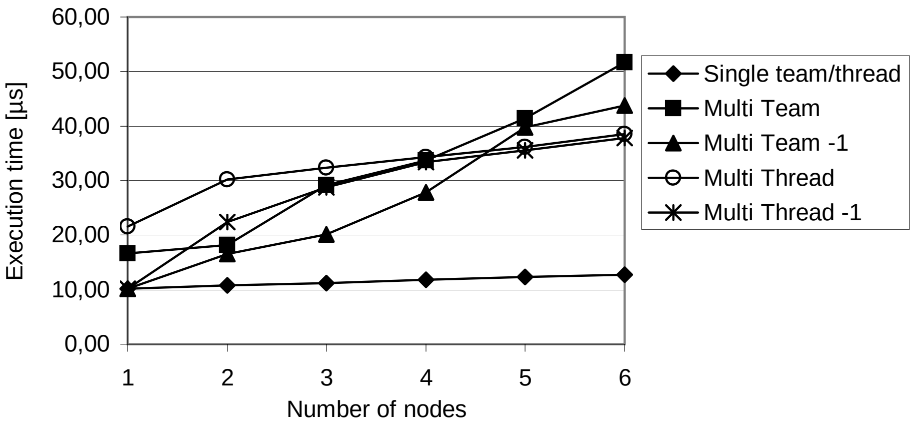


Figure 5.10 Barrier performance on up to 6 nodes.

the cost of memory and mapping resource requirements on each node being linear in the number of nodes.

The algorithm in the version above is only suited for one thread of execution in each process. It needs to be augmented to be able to work in SMP environments. To ensure the greatest possible efficiency, this is done in a two step process: a local barrier synchronizing all threads on one node and a global barrier for the inter–node synchronization. The first thread on each node to reach the barrier executes the global barrier, while any further thread simply blocks until the main barrier is completed. Before the blocking, the main barrier thread is informed of the arrival of another thread at the barrier by incrementing a local barrier counter. After the completion of the barrier, the main barrier thread releases all registered threads on the same node allowing them to continue their execution.

The two lines in Figure 5.10 marked as “Multi Thread” show the time needed for the execution of this barrier scheme using either two threads on each node or two threads on N-1 nodes and one thread on the remaining node. The data indicates that the overhead in this barrier version required for the registration and unblocking of the additional threads is initially higher than in the “Multi Team” cases, but that with with rising numbers of nodes, the increase in time spent for the barrier is again very modest resulting in a more scalable behavior. In addition, it is worth noting that in case one node is executed with only one thread, the barrier shows a similar behavior as in the case of one thread (and therefore also one node) less. The barrier for such a single thread on an additional node is therefore basically free.

Figure 5.10 also contrasts the performance of barriers with multiple threads per node to multiple processes per node each with one thread. This data is shown in the graph marked as “Multi Team”. With a small number of nodes, a barrier in this scenario performs better than the multithreaded counterpart, while in configurations with larger number of nodes the situation is reversed. This is likely to be attributed to the fact that the multithreaded version basically performs a barrier with one thread per node and just has to pay an (almost) constant overhead for registering and releasing additional threads while in the multi–team

	Time [μs]
Interrupt trigger	15.17
Interrupt ping–pong $/2$	94.95

Table 5.14 Execution time of interrupts in HAMSTER.

case each node performs two barriers with active polling leading to a higher overall system load and therefore to a performance degradation.

5.3.5 Global Counters and Interrupts

The two synchronization constructs described above, locks and barriers, are sufficient for the implementation of any arbitrary synchronization scheme. However, it can be beneficial to make further base mechanisms available to both ease and optimize the implementation of further synchronization constructs.

For this purpose, the synchronization modules exports two further low–level mechanisms, which are based directly on the capabilities of SCI: global counters using the SCI atomic transactions and (potentially remote) interrupts. Their implementation is therefore very lean and does not result in additional complexity. While the global counters are simply an abstraction of the atomic *fetch & inc* operations which already have been used to implement the locks, the interrupts are directly based on the interrupt implementation of the SISCI API [64] provided by Dolphin (see Section 2.3). This API, however, has been extended for the various software layers within the SMiLE project to support a more efficient interrupt management — the original version, as is currently distributed by Dolphin, does not support a user–level signaling mechanism; instead, active waiting on incoming interrupts is required. With the help of the mentioned SMiLE extension, interrupts can be delivered in a fully asynchronous way. This version is also used for the synchronization module described here.

Table 5.14 and Figure 5.11 show the performance characteristics of these two mechanisms. In both cases the performance is very good, which reflects their low–level implementation close to hardware and system software. Interrupts can be triggered on the local node with an impact of about 15 μs and a remote node is notified in under 100 μs.

With regard to the performance data for the atomic increments, the value for one node reflects the raw read latency of SCI, which is around 4 μs. With increasing numbers of nodes and therefore more concurrent processes performing an atomic increment operation, this latency is slightly increased due to contention on the network. The overall impact, however, is quite modest.

5.4 Task Management

Besides the services that allow the control over the actual shared memory, additional mechanisms to control the task or execution model are required. These are merged into the task

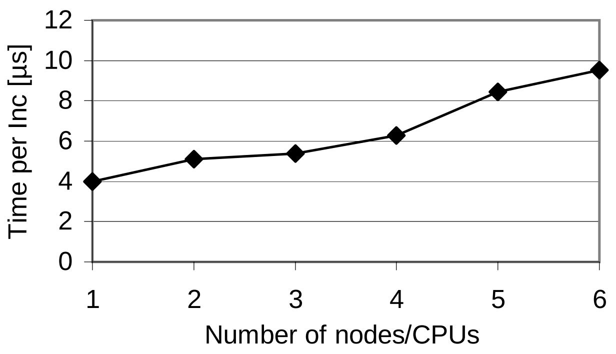


Figure 5.11 Performance of atomic counters under varying numbers of nodes.

module discussed in this section. Its functionality, however, is kept very lean in order to fulfill the goal of being unbiased towards any specific programming model.

This design choice is especially visible in the fact that the task module purposely does not provide mechanisms to create and/or control threads. Due to the subtle differences with regard to syntax and semantics in the thread APIs of the different target operating systems, the implementation of such a mechanism would either result in an emulation of one thread API on the other operating system (which would in addition either be very complex or semantically not complete) or in a subset of common thread functionalities. In both cases it would nearly be impossible to implement a cluster–wide extension of one of the native thread APIs.

For this reason, the implementation of thread services is left to the actual programming model implementation (see also Chapter 6.4). The task module, however, provides support services needed to guarantee a clean cooperation with the global process abstraction and its associated teams as described in Chapter 4. This can be used to implement any task structure that goes beyond the native one of one thread per node exported by the SCI-VM.

5.4.1 State of the Art

Several projects provide a global thread management. Typical examples for this class of systems are Nexus [56], Chant [73], PM2 [7] , and Mosix [12]. They provide the ability to create and manage distributed activities or threads and are mostly combined with transparent thread migration mechanisms. Those systems, however, do not include a global memory abstraction and are therefore not suited for implementing applications using the shared memory paradigm. They are rather used as runtime systems for higher–level programming environments.

5.4.2 Functionalities

The functionality of the task module, which is listed in detail in Table 5.15, can roughly be split into two groups: routines for the registration of threads and routines available for thread placement within an SMP. Both are intended to support multithreaded programming

API call	Description
taskMod_incNumberOfActivities	Register a new local thread
taskMod_decNumberOfActivities	Deregister a local thread
taskMod_queryNumberOfActivities	Query the total number of threads
taskMod_queryNumberOfLocalActivities	Query the number of local threads
taskMod_assignToCPU	Bind thread to given local CPU
taskMod_assignToNextCPU	Bind thread to next local CPU

Table 5.15 API of the task management module.

models that go beyond the simple model on which the SCI-VM is based.

The first group of routines allows user threads to be registered and deregistered with the HAMSTER framework. This is used to keep track of the total number of activities or threads on the local node as well as globally. At the beginning of the program execution, all initial user threads started by the SCI-VM, i.e. one on each node, are already registered. Any further thread created by the programming model layer needs to be registered by incrementing the activity counter. In the same manner, this counter needs to be decremented in the case of the termination of a thread. In addition, the task management module provides two routines to query the current number of activities on the local node and in the total system. This information can then for example be used for load balancing or thread placement decisions.

The second group of routines allows to optimize the behavior of multithreaded programming models when executed on SMP nodes by pinning the execution of threads to processors. This has the potential to improve the overall system performance as unnecessary thrashing in the assignment of threads to processors by the scheduler is eliminated. The feature to pin threads to processors, however, is not available on all platforms. To still ensure the portability of programming models, these routines are available on all platforms, with or without the functionality. As these routines do not directly influence the execution, but are only used for optimization purposes, this approach provides a portable execution across platforms.

As with all the other management modules, the task management module also implicitly collects some statistical data and thereby contributes to the HAMSTER monitoring system. However, as the functionality of the module is purposely kept small, the amount of data which can be collected is also limited. The concrete parameters are listed in Table 5.16. In contrast to the data in the other modules, the data for the last two parameters shown in the table is not accumulated during the runtime of the program, but rather represent the current status of the system: the number of activities or threads both global and local which are currently registered in the system (as it can also be queried using *taskMod_queryNumberOfLocalActivities* and *taskMod_queryNumberOfActivities*) and the current system load. These values are retrieved and updated at the time the statistics from the task module are queried by the user or a higher layer and returned together with the rest of the statistical data from this module.

Variable	Purpose
numAssigned	Number of times *taskMod_assignToNextCPU* called on local node
numAct	Number of global current activities
numLocAct	Number of local current activities
locLoad	Current load on the local node

Table 5.16 Statistical information collected by the task management module.

5.5 Towards a Global View: The Cluster Control Module

In contrast to the actual management modules discussed so far, the last module, the cluster control module, takes a special role. It is not designed and implemented in an orthogonal way to the other modules, but can rather be seen as a common support layer. It is responsible for controlling the configuration of the target system and for providing any other HAMSTER component, from the SCI Virtual Memory to the programming model, with the necessary information about it.

It also stands in a very close relationship to the task module as both deal with the organization of activities within the system — the task management module at thread and the cluster control module at process level.

5.5.1 State of the Art

Due to this close relationship between the cluster control and the task management module, any work related to the task module mentioned in Chapter 5.4.1 can also be seen as related to the cluster control module. In addition, the cluster control module in its function to establish a global machine abstraction is also comparable to the wide area of so called *Single System Image* (SSI) systems. These systems aim at hiding the distributed nature of the underlying hardware from the users by presenting them with a single frontend. The extent of this abstraction as well as the concrete intention behind it, however, varies greatly between individual SSI systems.

A few commercial systems have been developed to allow easy system management and batch processing and therefore feature special software installation and remote control tools together with a job management system. Examples for this are the Scali Software Platform (SSP) [190] and the “Raisin” Package by the Company Alinka [238]. These systems, however, do not include a global resource abstraction for applications since the behavior of applications on the target system is not influenced.

The latter issue is targeted by distributed operating systems which aim at extending the services offered by a single operating instance transparently to a distributed system. Approaches in this direction can be found both in industry, e.g. in Solaris MC™ [121], and academia, e.g. in the Tornado [59] and the Sprite operating system [173]. They com-

API call	Description
clusctrl_getNodeNum	Query the rank of the local node
clusctrl_getNodeCount	Query the number of nodes in the complete system
clusctrl_getNodeID	Query the SCI node ID of a node
clusctrl_getLocalScale	Query the maximal concurrency on the local node
clusctrl_getNodeScale	Query the maximal concurrency on a particular node
clusctrl_sendMessage	Send an asynchronous RPC–like command

Table 5.17 API of the cluster control module.

bine a global thread management with distributed operating system services and therefore complete the single system abstraction at operating system level.

5.5.2 Functionalities

The core of the functionality contained within the cluster control module is designed to deliver extensive information about the system configuration to any other layer. The routines for this purpose, which are listed in Table 5.17, allow the querying of the number of nodes involved in the current system and the rank of the local node within the system as well as the retrieval of the network ID (in this case the SCI ID), which can differ from the rank. In addition, special routines are provided to query the maximal concurrency on a particular node. This number, in the following called scale of a node, indicates the maximal number of concurrent activities suited for the particular node and can be used to indicate SMP configurations on certain nodes. It should be noted, however, that the scale of a node does not directly imply the number of running activities, but merely represents a hint for multithreaded programming models on how many threads can be started on a particular node without overloading it.

5.5.3 Asynchronous RPC-like Communication Service

Besides these configuration routines, the functionality of this module also includes a simple messaging mechanism allowing the remote execution of routines. This facility is both used by the HAMSTER modules themselves and is available for the implementation of programming models. Even though this mechanism seems contradictory to the shared memory paradigm of any target programming model, it is often required, or at least useful, for their implementations as it enables the exchange of small messages for configuration and/or status update purposes.

The messaging engine in the cluster control module is implemented as an asynchronous, non–blocking Remote Procedure Call (RPC). Using the routine *clusctrl_sendMessage*, a sender can specify the address of a routine on the receiver node that will be executed on arrival of the message. The system is built for small amounts of payload data which will be delivered to the target routine as a parameter. This style of communication is simplified by the fact that any node is required by the SCI-VM to load and execute the same binary.

Therefore, any potential target routine is always available on any node at the same location and can hence be specified by its address. This omits the need for an elaborate naming scheme or centralized lookup service.

In its current state, the messaging mechanism is implemented on top of standard sockets using TCP/IP over Ethernet. This choice was made as this mechanism is normally used only during the configuration phase of either the management modules or any programming model. The performance of this mechanism was therefore not a major issue resulting in the choice of a less complex and more robust implementation. The actual send and receive mechanism, however, is encapsulated into a single submodule leaving the option for an easy port.

5.5.4 Control Over the Global Process Abstraction

The exported routines, however, do not cover the complete functionality of the module. Unlike the other modules discussed so far, it also includes implicit functionalities which are executed transparent to the other layers. One of the core points of this is the management of the global process abstraction. Such an abstraction is required to support the global virtual memory abstraction implemented within the SCI-VM introduced in the previous chapter.

The cluster control module is responsible for establishing the teams used by the SCI-VM, their coordination, control, and communication. This forms a global environment which, together with the global memory abstraction, allows to transparently host threads on any node without significantly influencing or restricting their execution.

This global process abstraction is, however, currently not complete. Especially the area of I/O is at the moment not covered resulting in the fact that any I/O operation issued by a thread running within HAMSTER will only be executed on the node local to the calling thread without any global coordination. In order to overcome this, a transparent forwarding mechanism for I/O operations in coordination with a global resource management would be required virtually extending the complete operating system to the cluster environment. This would have been far beyond the scope of this work, but might prove worthwhile to pursue in the future.

5.5.5 Startup Control

As already described in Chapter 4.4, an SCI-VM based application is initialized by executing the same binary compiled on top of a HAMSTER–based programming model on all participating nodes. This, however, does not include the correct identification of these nodes nor the initial establishment of connection between them. These tasks are taken care of by the cluster control module implicitly at program initialization time and this information is then provided to any other component of HAMSTER — including the SCI-VM — through the interface described above.

The task of node identification and system setup is based on a configuration file which has to be provided by the user. It is read and evaluated by this module during program initialization. In order to keep the complexity for the user as low as possible, the syntax is kept extremely simple and easy–to–use. Two examples for such a file are shown in Figure

```
#name    SCI ID   Scale        #name    SCI ID   Scale

smile1   4        2            smile1   4        1
smile2   8        2            smile1   4        1
smile3   12       2            smile2   8        1
smile4   16       2            smile2   8        1
smile5   20       2            smile3   12       1
smile6   24       2            smile3   12       1
                               smile4   16       1
                               smile4   16       1
                               smile5   20       1
                               smile5   20       1
                               smile6   24       1
                               smile6   24       1
```

Figure 5.12 HAMSTER configuration file for complete SMiLE cluster consisting of 6 dual SMP nodes in SCI ringlet configuration — left: exploiting SMPs with multiple threads; right: exploiting SMPs with multiple teams per node.

5.12. It is a plain ASCII text file and contains a line for each node participating in the global process abstraction (plus arbitrary numbers of comment and empty lines). For each node three parameters have to be provided: the DNS node name, the network ID (in this case the SCI ID), and the scale of the particular node.

It is also possible to list nodes several times in the configuration file. In this case, several teams (equal to the number of listings in the configuration file) are started on the respective node. Each of these teams then behaves like an independent node participating in the HAMSTER system. An example for this can also be seen in Figure 5.12 which shows both ways to specify the SMiLE cluster used for all experiments, on the left side using the scale specifier and on the right side using multiple node entries to specify SMP node configurations. This second option is introduced to support SMP nodes also for programming models that do not directly exploit multithreading within a team. This extends the applicability of the HAMSTER system and allows SMP clusters to be used both for true multithreading programming models as well as programming models designed for only one thread per team.

In both cases, the order in which the nodes are listed in the configuration file determines their rank in the system. The top listed node will be assigned the rank zero, the second one rank one, and so forth. In the case, multiple entries are used for a node, the team started first on that particular node will receive the lowest rank on that particular node, the second team the second lowest and so forth until the last thread, which then receives highest rank in the configuration file for that particular node. The first node in the configuration file will also receive a special role as a master node during the initialization phase and should be called last. During later stages this special role will be dropped.

The cluster control module also ensures that all nodes are running before any further

HAMSTER mechanism is invoked. For this purpose, the execution of the application is handed to the cluster control module at the beginning which blocks the execution until all nodes listed in the configuration file have started executing and joined the global process abstraction. This is necessary to guarantee that the complete global process abstraction is in place and provides the associated functionality before any further code is executed that relies on this abstraction and might cause fatal errors in the case of its absence.

5.5.6 Clean Termination

Not only the startup process requires special treatment by the cluster control module, but also the termination of the global process abstraction needs to be considered. In stand–alone operating systems, a process is normally considered terminated as soon as its last thread terminates. This semantics has to be extended to the complete cluster, requiring the teams on all nodes participating in the global process to terminate if (and only if) the last thread on any node is being terminated. Such an accurate termination policy is of special importance for the implementation of programming models based on a dynamic task management, as is given in any thread API.

To implement this scheme, it is not only necessary to have an overview of the total number of threads currently running within the system, which can be obtained using the mechanisms of the task management module discussed above, but also to keep teams currently without user threads alive since their resources are still needed to contribute to the overall global process and the node still has to be available to potentially host new threads. This is achieved through additional communication threads created by the cluster control module, guaranteeing the number of threads per team registered with the operating system always to be at least one. This ensures that no team will be evicted by the operating system on any node prematurely.

In the case the activity counter is decreased to zero, no other activity is running within the whole system and the complete global process has to be terminated. This is done, again under the control of the cluster control module, by terminating the communication threads after a global node barrier to ensure a synchronous termination across the cluster. This will then delete the last thread in each team which results in the termination of the individual team processes and thereby the complete global process abstraction.

5.6 Summary

In order to fulfill the goal of enabling as many shared memory programming models as possible or necessary, the HAMSTER framework provides a large number of shared memory services. These are grouped in independent management modules, each providing a specific set of services. These modules are implemented in a fully orthogonal manner without any cross–dependencies ensuring the openness and flexibility of the overall system.

The memory management module contains mechanisms for a flexible NUMA memory management. It not only facilitates the allocation and management procedures for a global shared memory, but it also provides an interface giving the programmer control over data placement and coherency parameters of newly allocated memory based on optional annotations. In addition, it contains mechanisms for the distribution of static application data completing the transparency required by some shared memory programming models.

The coherency management modules provide all mechanisms necessary to enforce the memory consistency needed for a safe and reliable application execution. Together with the synchronization module, which exports optimized implementations of low–level synchronization constructs such as locks and barriers, it can be used to create relaxed consistency models. This will be illustrated in more detail in the next chapter.

Besides these core shared memory services, services regarding the task and global process management are also necessary to complete the requirements for an implementation of shared memory programming models. This functionality is encapsulated in the task management and the cluster control module. The latter one is also responsible for any kind of global cluster and process management and therefore functions as a central management component in the overall framework.

Combined, the services exported by the different management modules provide a comprehensive infrastructure for the implementation of shared memory programming models. The issues related to this endeavor will be explained in detail throughout the next chapter.

Chapter 6

Implementing Programming Models on Top of HAMSTER

The modules described in detail in the last chapter can be used to build almost any shared memory programming model. The interface available for this purpose is discussed in this chapter followed by the description of the most important implementation aspects encountered in different programming models. The chapter is then rounded up by a complete overview of all existing programming models on top of HAMSTER and a discussion about their implementation complexity.

6.1 The HAMSTER Interface

The HAMSTER interface forms the connection point between the collection of services provided by HAMSTER and the programming models using them for their implementation. This interface, however, is no rigid monolithic block, but should merely be seen as a toolset collected from several independent parts. From a high–level point of view, this toolset can be split into two main groups: the core HAMSTER services provided by the modules introduced in the last chapter and a small set of auxiliary services for additional support.

6.1.1 HAMSTER Services

The main part of the HAMSTER interface is formed by the actual core HAMSTER services exported through the various management modules discussed in Chapter 5. This collection of routines allows programmers to implement and control most aspects of shared memory programming models.

6.1.2 Multiplatform Timing Support

Besides these core services for the actual implementation of shared memory programming models, the HAMSTER interface provides additional services and functionalities available to any higher layer. One example of such an add-on service is a timing module providing platform independent support for timing. This includes both the emulation of routines typically used on one platform — like *times()* or *gettimeofday()* — on any other supported platform and the implementation of own timing routines based on high precision timing mechanisms available in the underlying architectures.

This timing module further helps to increase portability of both programming models and applications between different platforms. It eliminates the need to adapt to the various different timing mechanisms on the individual platforms which traditionally is a very tedious task. In addition it forms the basis for easy and fair cross platform performance studies.

6.1.3 Using the HAMSTER Interface

All services provided by HAMSTER are available as standard C bindings and can be used by any programming system capable of using such a binding, even though during the course of this project only standard C or C++ environments (gcc for Linux and Microsoft's Visual C++™ for Windows NT™) have been used. The only prerequisite is that the complete final binary is static in the sense that the layout of its virtual address space after it has been loaded by the OS is constant. Only this allows the creation of identical copies of the process image on several nodes, which is necessary for the SCI-VM and the memory management module to work properly and to guarantee a fully transparent global memory abstraction.

In order to utilize the services available through the HAMSTER interface, the programming model layer has to include the respective header files for the individual modules containing them. Those are themselves based on a small set of global header files with system wide type and constant definitions. Together they form the complete basis for the implementation of any programming model, which then in turn exports its own API to the final user application.

6.2 A First Model: Single Program – Multiple Data (SPMD)

The first programming model implemented within the HAMSTER framework is a programming model that implements a straightforward Single Program – Multiple Data or SPMD model. Its concrete API and usage is explained in Appendix B. In this model a fixed number of threads equal to the number of nodes in the system is executed. All threads work in principle on the basis of the same code (Single Program), but each thread has its own part of the data to work on (Multiple Data). As an exception to this rule special code segments (normally not as a part of the computational core and mostly dealing with initialization, post processing, or I/O) might be performed only by a single master thread. The programming model, however, does not enforce the SPMD programming style; nothing prevents the user from diverting from it and implementing more irregular code constructs, but this is depreciated and normally requires a significantly higher coding complexity. Therefore, all codes based on the SPMD discussed in this work stick to this particular programming style.

During the execution, data regions can be allocated and shared between the nodes. These regions are normally used to store the application's working set while support and management variables are often stored in node local resources like a thread's stack. The main synchronization method in this model are barriers since they separate phases during the program execution and often help to separate the work in distinct iterations and/or different sections of the data. Locks are only very rarely used, because they contradict the pure SPMD style due to their nature of mutual exclusion which allows only one thread to perform work during this execution.

The SPMD model was chosen as the first test programming model for the HAMSTER system as it has a straightforward and simple task model that does not introduce any further difficulties for testing and tuning while still allowing a complete evaluation of the memory management, synchronization, and consistency modules. For the same purpose, it also exports most of the features of these modules directly to the user. This, however, is done in a way that allows users to easily ignore advance features relying on default parameters. As a result, the SPMD model combines both simple and advanced HAMSTER mechanisms and hence serves as an ideal testing ground for the HAMSTER system, especially for the lower management and memory abstraction layers.

6.3 Support for Various Consistency Models

In order to fulfill the goal of supporting almost any arbitrary shared memory programming model, HAMSTER also needs the ability to support the various consistency models present in other systems. This is especially important when supporting HAMSTER–based APIs of existing DSM systems since this area of research has produced a large variety of such models. This section discusses how the mechanisms provided by the HAMSTER modules can be used to create these consistency models, including an in–depth discussion of the two popular consistency models *Release Consistency (RC)* and *Scope Consistency (ScC)*[1].

6.3.1 State of the Art

Relaxed consistency models have been present for a long time in SW–DSM systems, as already discussed in Chapter 4.1.2. Two prominent examples among them are Release Consistency [116], implemented in systems like TreadMarks [5], CVM [117], Shasta [189], and Munin [29], and Scope Consistency, implemented in Brazos [209] and JiaJia [83]. The main intention for their deployment is their ability to reduce the amount of communication necessary for the maintenance of the global memory abstraction since the number and size of updates can be reduced.

The actual performance impact of the different models, however, is strongly dependent on the target application and its memory access pattern as well as the behavior of the concrete consistency protocol implementation. For this reason, few projects aim at providing the ability to support either several given or even user–defined consistency protocols. In latter case the respective systems provide a set of basic DSM functionalities and/or a set of parameters allowing the definition of new protocols. Examples for systems with such

[1]The abbreviation ScC is used instead of SC to distinguish Scope Consistency from Sequential Consistency.

a multiprotocol support are the CVM framework [118], in the ADSM system [160], and Munin [29, 17].

While the flexibility of these systems is quite comparable with the intentions within the HAMSTER framework, the inherent restrictions for the different approaches are radically different. The HAMSTER system draws its restrictions from the underlying hardware architecture and the memory coherency types realized on it; software implementations of multiprotocol systems impose restrictions from the selection of exported functionalities and/or parameters as well as from the underlying base SW-DSM systems.

6.3.2 Combining Consistency and Synchronization

In most DSM systems with relaxed consistency (see Sections 5.2.1 and 4.1), the consistency management is inherently connected with synchronization constructs. This is especially valid for constructs guaranteeing a mutual exclusion, like locks. Here, most programming models combine an *Acquire* operation of some kind with a *Lock* call and the corresponding *Release* with the corresponding *Unlock*. This kind of combination guarantees a consistent view on any memory accessed only from within critical sections. In addition, barriers are often combined with a full synchronization of the underlying memory system, i.e. *Acquire* and *Release* together, resulting in a consistent memory between program phases separated by barriers.

However, by hardcoding this combination of synchronization and consistency control, the resulting system is bound to one particular relaxed consistency model and therefore only exhibits a restricted amount of flexibility. While this is tolerable for DSM systems intended for application programmers, it does not suffice for a system like HAMSTER, which aims at the implementation of arbitrary shared memory programming models. Hence, in the HAMSTER system, the synchronization module does not include any consistency enforcing mechanisms and vice versa, guaranteeing the full orthogonality between these two modules.

Instead, the connection between synchronization and consistency management is left to the implementation of the programming model, the layer above the individual management modules. This has the intended advantage that each programming model can select its own consistency model and combine the appropriate synchronization mechanisms. In addition, it opens the possibility for the implementation of application specific consistency models without much additional cost. In summary, this approach provides a maximum of flexibility and induces no restrictions to the programming models implemented on top of this system.

6.3.3 Implementing Release Consistency

As a first example of how to connect the mechanisms from the synchronization module with the consistency management routines, the following section provides a description on how to implement Release Consistency.

Consistency conditions

When using Release Consistency, all memory operations are divided into synchronizing and non–synchronizing operations. The first group is further split into so–called *Acquire* and *Release* constructs. The first ones are used to gain permission to access shared data, while the second ones are used to grant access to shared data to other processes.

Based on this division, Release Consistency is defined using the following three consistency conditions [116]:

1. Before a read or write access is allowed to perform with respect to any other processor, all previous *Acquire* accesses must be performed.

2. Before a *Release* is allowed to perform with respect to any other process, all previous read and write accesses must be performed.

3. *Acquire* and *Release* accesses are sequentially consistent with respect to one another.

Informally, the two operations *Acquire* and *Release* represent routines controlling the visibility of data on remote nodes defining a window of safe access to shared data. They are therefore enhanced with synchronization semantics which are used to control the task structure since only this combination enables the control of both data and tasks as required from an application point of view.

With regards to locks, typically an *Acquire* is combined with a lock operation, while a *Release* is combined with a corresponding unlock operation. This ensures that the data accessed during the critical section always represents the current global state (due to the *Acquire*) and that any modification is made available after the critical section is left (due to the *Release*).

Implementation

Based on this informal description, it is clearly visible that the *Acquire* operation corresponds to the read invalidation described above, as in both cases the data on the local nodes needs to be updated, i.e. old data needs to be invalidated. On the other side, the *Release* operation can be implemented with a write flush since in both operations the locally modified data is pushed across the network and therefore made available.

More precisely, the implementation is as follows:

- *Aquire*
    - Perform lock operation
    - Perform read invalidation

- *Release*
    - Perform write flush
    - Perform unlock operation

This implementation approach satisfies all three conditions stated above and therefore provides a full Release Consistency implementation.

Differences to software implementations

Software implementations distinguish further between *Eager* and *Lazy Release Consistency* (*ERC* and *LRC* respectively) [116]. This difference influences when memory updates are sent to remote notes — ERC implementations send the information to any other node at the time of the *Release* operation, while in LRC implementations the processes receive and consume them at the time of the *Acquire* call.

This difference in implementation also has a subtle influence on the memory consistency model realized by it. While in ERC, all information released by previous *Release* operations is available in the whole system before the next *Acquire* on any node, LRC implementations only guarantee the availability of the information on the node performing the *Aquire*.

The HAMSTER–based realization of RC for NUMA architectures presented above can not be cleanly classified as either ERC or LRC, but rather represents an intermediate solution. Like in ERC, *Release* operations push their memory update to any other node and therefore make them available. However, their contents may not be visible immediately since stale values might still be stored in the processor caches. Even though these are not invalidated before the next *Acquire* operation, sporadic cache evictions caused by cache line replacements can lead to an earlier deletion of stale values and hence to the visibility of the new information to the processor.

6.3.4 Implementing Scope Consistency

The *Release Consistency* model described above creates an implicit relation between *Release* and the following *Acquire* since any data written before the *Release* must be made visible after the following *Acquire*. This relation can be loosened by introducing an explicit relationship between groups of *Acquire* and *Release* operations and restrict the visibility of data after an *Acquire* to the data written before a *Release* within the same group. Hence, it implicitly decouples the memory updates caused by the *Acquire* and *Release* operations in the different groups and thereby also the memory consistency constraints in the memory regions which are updated between the respective operations. This creates so called Consistency Scopes, as introduced by [97], without having to specify an explicit association of data and consistency operations.

Consistency conditions

As this concept of Scope Consistency is just a straightforward extension or generalization of the Release Consistency already discussed above, its consistency conditions are also very closely related. The key difference is the introduction of consistency scopes which can be opened and closed by applications. Any read or write operation is then performed with respect to any open scope and write operations are assumed to be completed with respect to a scope on closing this scope. Using these concepts, the consistency conditions of Scope Consistency can be defined as in [97] (with a session being the interval during which a consistency scope is open at a given process).

1. Before a new session of a consistency scope is allowed to open at a process P, any write previously performed with respect to that consistency scope must be performed with respect to P.

2. A memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened.

Informally, this definition leads to the memory behavior that memory accesses made during a scope are only guaranteed to be visible after the scope has been closed and also only within the same scope opened by another process. Any other access is not guaranteed to be visible. The first rule is responsible for ensuring that updates are properly propagated after a close and the second rules guarantees that these updates have been completed in time.

The important consistency enforcing mechanisms in these concepts are the opening and closing of scopes as these represent the points during a program's execution in which the visibility of data changes. Contrasted to Release Consistency, opening a scope is the same as an *Acquire* (only restricted to the scope) and closing a scope is the same as a *Release* (again restricted to the scope).

Like Release Consistency, this concept has also to be used in conjunction with synchronization mechanisms in order to allow applications a useful assumption about the underlying memory behavior. In order to exploit the scope concept to a maximum, each lock is normally associated with its own scope as it is assumed that each lock is responsible for the management of a distinct part of the overall data set. During the lock operation the respective scope is opened, i.e. the scoped *Acquire* is performed, and during the unlock operation the scope is closed again, i.e. the scoped *Release* is performed.

Using this inherent connection, the usability of Scope Consistency is drastically increased since an explicit scope management would significantly increase the coding complexity of applications. Experiments [97] have actually shown that most codes written for Release Consistency can directly be used with Scope Consistency without any code modification, but at improved performance.

Implementation

A HAMSTER–based implementation of Scope Consistency can be achieved in a straightforward manner using the scoped versions of the consistency enforcing mechanisms introduced in Chapter 5.2.4. These have been derived from the general mechanisms which are used to implement Release Consistency, in the same manner as Scope Consistency has been derived from Release Consistency. Both exploit the implicit dependency between *Acquire* and *Release* operations and optimize by restricting this dependency to explicit groups, the so–called scopes.

An implementation of Scope Consistency can therefore be achieved based on the RC implementation discussed above by replacing the unconditional write flushes and read invalidations with their scoped counterparts. In addition, a new scope is implicitly allocated and associated with every lock requested by an application. This results in the intended consistency model, fully compliant with the constraints and their informal interpretation mentioned above.

Differences to software implementations

Due to the close relation between the implementations of the two consistency models, the differences to the software implementation of RC also apply to Scope Consistency. In addition, another difference is introduced by the fact that any read invalidation or write flush always has a global impact on the overall system whereas pure software implementations can control which data the consistency operation can be applied to. This leads to earlier transmissions of data than necessary, however without changing the guarantees of the consistency model. Therefore, this slight difference does not have an impact on applications using this model.

A second difference between a software and a NUMA–based implementation can also be seen in how the transition of *Release Consistency* to *Scope Consistency* affects the performance of applications. While software implementations use the scope information to reduce the memory traffic and to defer the communication, NUMA–based systems use scopes to reduce the number of necessary invalidations resulting in fewer cache misses during the execution. This difference, however, is transparent for the user.

6.3.5 Experimental Evaluation

An evaluation of these consistency model implementations is rather difficult. For one, no new models are introduced, but rather given concepts, whose efficiency has been extensively proven before [116, 62, 256, 97], are discussed in a new environment. In addition, any evaluation based on application performance gives more an indication of the performance of the underlying DSM system than of the consistency model. Such a comparison, besides being unable to provide the evaluation intended, is also very likely to be unfair since the software DSM systems available for this work use TCP/IP–based communication with Fast Ethernet, while the communication in the NUMA–based SCI system is performed directly in hardware without protocol overhead.

In order to still give a first impression with regard to performance, a few low–level benchmarks are used to help contrast a few key characteristics of the NUMA–based system using relaxed consistency with their software DSM counterparts. These benchmarks are synthetic and mimic a few extreme memory access patterns.

In addition, two larger benchmarks from the SPLASH–2 suite [233] are used to identify the difference between *Scope* and *Release Consistency* using the HAMSTER approach. These experiments show how and to which degree using Scope Consistency affects the execution of these applications and allows a first assessment of a performance improvement.

Together, these two sets of experiments give a first insight into the characteristics of NUMA–based relaxed consistency models and how they behave in contrast to traditional approaches using software protocols.

Operation	SW–DSM	NUMA–DSM
Acquire	2.35 μs	471.20 μs
Release	2.45 μs	25.05 μs
Acquire (short write / 4 bytes)	1885 μs	3175 μs
Release (short write / 4 bytes)	25 μs	25 μs
Acquire (long write / 64 Kbytes)	112535 μs	44030 μs
Release (long write / 64 Kbytes)	65 μs	30 μs

Table 6.1 Performance of *Acquire* and *Release* operations.

Low–level mechanisms and synthetic benchmarks

This set of experiments concentrates on the low–level performance of the consistency mechanisms for NUMA architectures contrasted to the performance of the equivalent mechanisms found in the software DSM systems (in this case the TreadMarks system [5]).

As a first approach, the performance of the actual *Acquire* and *Release* operations are measured. The results can be seen for both the software DSM and the NUMA–based version in the upper rows of Table 6.1. They seem to suggest that the software DSM counterparts are significantly faster than the NUMA–DSM mechanisms. An explanation for these numbers can be found in the different implementations: software DSM systems maintain an implicit access control to the globally shared data in order to be able to detect when data needs to be propagated to or fetched from a remote memory location. A respective system therefore can easily detect that this benchmark does not perform any global data access (as only the execution time for the routine itself is measured) and hence the operations do not perform any communication. In the NUMA-DSM system, on the other hand, the system does not have such information, requiring it to always flush or invalidate the respective buffers, thereby resulting in the times shown.

This picture changes when communication is introduced. The rest of Table 6.1 shows the time needed to perform lock / write / unlock cycles on 6 nodes. In this scenario, the software DSM system also shows significant execution times since it has to perform the data transport during the *Acquire* operations. With larger amounts of data, this also leads to a larger overhead resulting in significantly longer execution times than the NUMA–DSM counterparts, which only need to flush and invalidate the respective buffers.

This behavior is further detailed in Figure 6.1, which shows the distribution of execution times for the two operations as well as the actual loop times for various amounts of data written during the loop. Again these numbers demonstrate the radically different implementations in these two system: the distribution for the software DSM system is rather independent of the write granularity since the amount of work required for the two major phases, the actual loop body and the *Acquire*, linearly depends on the amount of data written. The execution time for the loop body is mainly caused by the page faults used by the DSM system to keep track of the memory state while the time for the *Acquire* is caused by the actual communication. In the NUMA–DSM case, on the other side, the global data

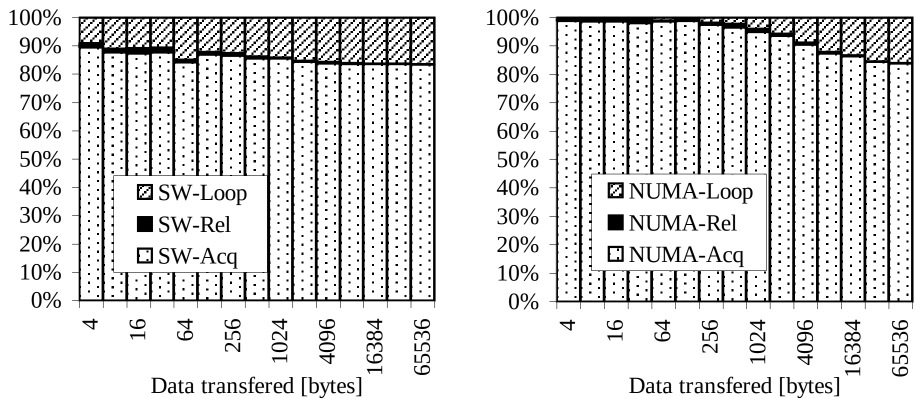


Figure 6.1 Time distribution during *Acquire/Write/Release* cycle (SW–DSM left, NUMA–DSM right).

access during the loop body is executed fully in hardware, keeping the amount of time required for it extremely low. Due to this, the vast majority of the time spent is during the *Acquire*. With rising amount of data to be transfered, the interconnection network and the local I/O busses become a bottleneck, leading to an increased loop execution time, which can be seen in the right graph.

It should be noted that in both cases, the execution time spent for the *Release* operation is insignificant, however, due to different reasons. The SW–DSM system simply performs some bookkeeping, a purely local operation without much computational overhead, while the NUMA–DSM system performs a write flush which only stalls until all data has been transmitted, again a rather short time due to the low latencies provided by SCI.

The performance experiments above, however, have been skewed by the fact that all processors actually perform a full lock operation. This generates lock contention resulting in the fact that a majority of the time spent for the *Acquire* operation actually comes from waiting for a lock. To avoid this influence, the following experiments use a barrier based approach, in which all processors (again all 6 nodes) access the global memory during an iteration and the memory consistency is enforced through both a *Release* before and an *Acquire* after the barrier. These two operations, however can only be measured together and combined with the barrier wait time since the TreadMarks system does not offer these operations separately. However, as the barrier wait time is expected to be low (all processors execute the same amount of code) and the execution time of the *Release* operation is very short (as seen above), the time measured still gives a good impression about the *Acquire* time.

The graphs in Figure 6.2 show the results of these experiments based on the full execution time of one iteration, i.e. loop body and barrier time, for a range of different data sizes. Three different memory access patterns have been tested: read–only, write–only, and read–write in a consumer–producer like manner.

When comparing the read–write curve with the other two, the data shows that the per-

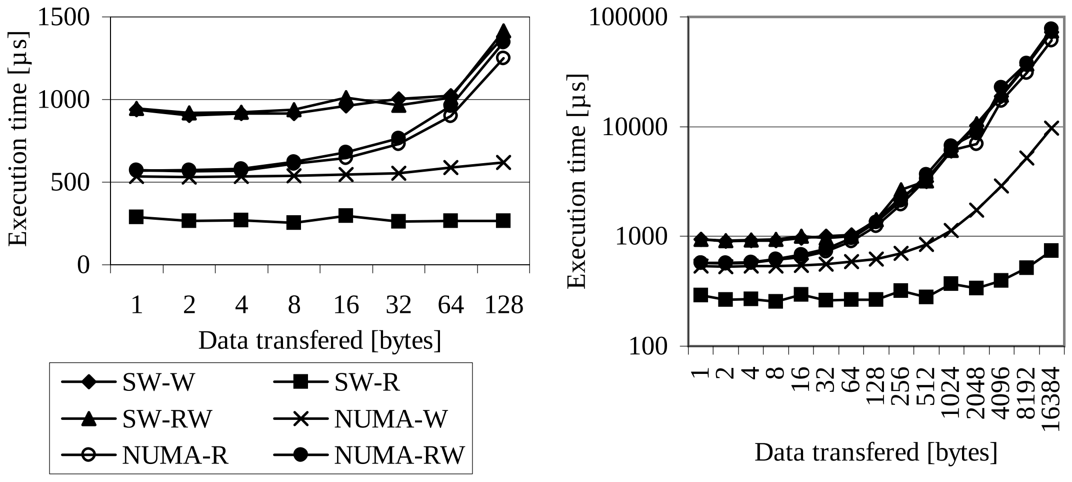


Figure 6.2 Performance under *Read*, *Write*, and *Read/Write* traffic.

formance of a software DSM system is bound by the behavior caused by writes since those actually cause communication in such a system. In the read–only scenario, on the other side, virtually all reads can be performed from replicated copies in local memory and the performance is only slightly affected by the bookkeeping overhead of the DSM system. In NUMA–based DSM system, the scenario is radically different: here the performance is bound by the read performance because these are the operations which trigger the actual communication in these scenarios, whereas writes can be pipelined by the local network adapter and therefore do not lead to a drastic overhead.

This different communication behavior also leads to a different distribution between the time spent during the loop body compared to the time spent during the barrier. This is shown in Figure 6.3 for the most significant case, the read–write scenario. The software DSM system spends almost an equal amount of time during each of these two parts and the time rises roughly linearly with the amount of data transfered. This comes from the fact that these systems require the bookkeeping during the loop and the actual communication during the *Acquire*. NUMA–DSM systems, on the other hand, spend most of their time during the loop body, which can mostly be related to the latencies caused by remote reads, while the time needed for the barrier stays almost constant, because the same operations, i.e. the flush and the invalidations, are always performed.

At the end, it should be noted that all of these experiments just show the raw performance of the system under the synthetic conditions of the memory transfer benchmarks. While this allows a clean insight into the different characteristics, these benchmarks do not allow an assessment of the different system performances. Many of the beneficial properties present in most application codes, like spatial and temporal data reuse or a significant amount of local operations without global data access, are not present in these benchmarks.

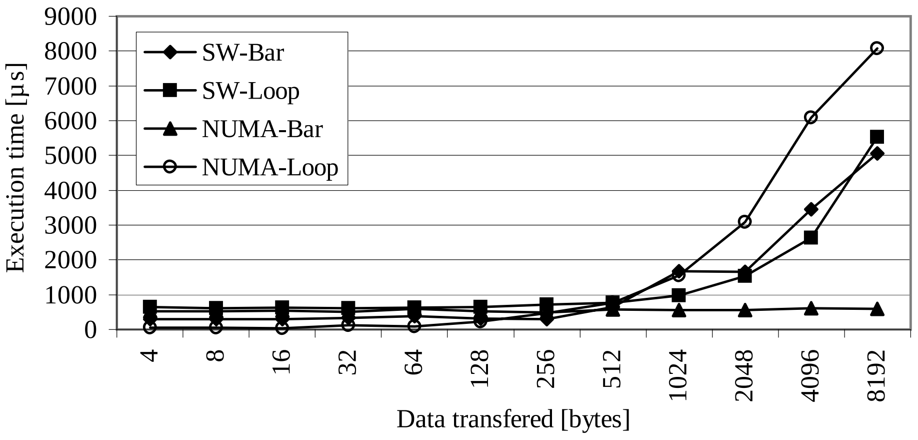


Figure 6.3 Distribution of work in the *Read/Write* scenario.

Release vs. Scope Consistency on NUMA systems

In a second set of experiments, the differences between *Release* and *Scope Consistency* for NUMA–based architectures is investigated. For this purpose, two applications from the SPLASH–2 suite [233], namely the *RADIX* sorting kernel and the *WATER N-Squared* molecular dynamics code, are used in combination with a HAMSTER port of the ANL macros. These two codes both use a rather large number of locks and therefore provide a suitable basis for an analysis of these two consistency models.

Table 6.2 summarizes the results of this set of experiments. It includes data for both codes gathered during two runs each: one using *Release* and one using *Scope Consistency*. In both cases the number of lock, barrier, and consistency operations are presented for two areas: the total code (including any operation needed by the NUMA–DSM framework for its own setup and initialization) and the computational core.

In both codes, the use of *Scope Consistency* enables a significant reduction of the number of *Acquire* operations during the execution, mainly during the core phase. Up to 68% percents of all *Acquires* could be avoided and with them the related read invalidation operations. In both codes, this also leads to an increase in the overall performance.

6.3.6 Applicability

It should be noted that these results show a general direction of the characteristics that can be expected when using NUMA–based consistency models. The concrete performance data depends strongly on the properties of the underlying hardware. These can vary greatly due to different network speeds and latencies as well as due to different optimizations like streaming, buffering, prefetching, or memory coherency guarantees. The general trend shown by the experiments above, however, will be the same across all platforms since they depend on the direct exploitation of the NUMA principle. These results should therefore be applicable to in general any NUMA–based DSM system.

	RADIX		WATER (N-Squared)	
	Release C.	Scope C.	Release C.	Scope C.
Data set size	262144 keys		1331 molecules	
Locks (total)	158	158	663	663
Locks (core)	12	12	89	89
Barriers (total)	330	330	1095	1094
Barriers (core)	24	24	24	24
Acquire (total)	621	615	2295	2183
Acquire (core)	24	18	113	36
Release (total)	621	621	2295	2295
Release (core)	24	24	113	113
Ops saved (total)	6 / 0.97 %		112 / 4.88 %	
Ops saved (core)	6 / 25 %		77 / 68.14 %	
Execution time	4165 ms	3990 ms	8276 ms	8154 ms
Improvement	4.4 %		1.5 %	

Table 6.2 Properties of two applications running with both *Release* and *Scope Consistency* (on 4 nodes).

6.4 Transparent Multithreading on top of HAMSTER

The most commonly used shared memory programming models are thread libraries. They exist as part of the native API of most modern operating systems and are used to both hide I/O latency and efficiently exploit symmetric multiprocessor (SMP) systems. Especially the use for the second purpose makes these programming model an interesting target for this work. By implementing them within the HAMSTER framework in a transparent fashion, multithreading can be extended from SMP systems to clusters exploiting their increased scalability.

Currently two different thread APIs have been implemented on top of HAMSTER: the POSIX thread API for Linux [219] and the Win32 thread API for Windows NT/2000™ [154]. Both will be explained in more detail below.

6.4.1 State of the Art

Any of the common general purpose operating systems comes with a native thread API. This includes the two operating systems supported by HAMSTER, namely Windows NT/2000™ with its Win32 thread API [154] and Linux, which since the kernel version 2.0 includes POSIX compliant threads [219]. These APIs, however, differ in the concrete semantics making it difficult for porting codes across platforms.

To compensate for this problem, several projects have been initiated to form a portable thread layer across various platforms. Examples for this kind of system are ACE [191] developed at the Washington University, St. Louis (MO) and Thread.h++ from Precision Software [248]. They export a new API with an identical semantics across all supportive platforms and therefore allow a quick and easy porting of applications between platforms. However, the semantics of this new API differs from any of the platform specific APIs in order to accommodate the subtle differences, again making the initial port of existing applications a complex task.

In addition to these thread APIs, which present potential target programming models for the HAMSTER framework, projects targeting distributed implementations of thread APIs on top of global memory abstractions are also related to the work discussed in this chapter[2]. Two examples of this, using opposite approaches, are the DSM–Threads [164] and the RThreads [46] projects. Both aim at providing a POSIX threads like environment for distributed computing, but while the DSM–Threads are layered on top of the global virtual memory abstraction created by a conventional SW–DSM system (see Chapter 4.1) implemented as a pure user–level library, the RThreads approach utilizes a special pre–processor to identify and implement remote memory accesses. In addition, the RThreads require some simple user interaction in form of annotations in the intermediate code to complete the global memory abstraction. While this increases the complexity for the programmer, it also allows for easy optimizations directly adjusted to the runtime behavior of the target application.

6.4.2 Global Thread Identification

One of the main issues when providing a global thread abstraction is the creation of a global name or ID space for the threads and their associated resources. Any identifier has to be globally unique and must allow the determination of both the node on which the thread or the resource resides and a reference to the node local resources managed by the operating system.

In both HAMSTER–based distributed thread systems currently in existence this issue is resolved in the same way. In both systems, Linux and Windows NT/2000™ , threads identifiers are 32 bit values handed out by the operating system on their creation. However, never all of the 32 bits are used allowing the HAMSTER system to convert the upper 8 bits into a storage for the node identifier the thread resides on (see Figure 6.4). This approach has the significant advantage that the data structure representing a thread in the respective original operating system API can be kept, hence avoiding any impact on the actual API.

In order to maintain transparency, the new HAMSTER–based versions of the APIs only use the extended globalized versions of the thread identifier for any interaction with applications running on top of them. Any conversion into native operating system identifiers is done implicitly without any need for a user intervention.

6.4.3 Forwarding Requests to Potentially Remote Threads

The HAMSTER framework with its core, the SCI Virtual Memory, provides all necessary means and mechanisms to implement a cluster-wide distributed thread library. Most of the

[2]Distributed threads without a global memory abstraction are more related to the task management module and are therefore briefly discussed in Chapter 5.4.1.

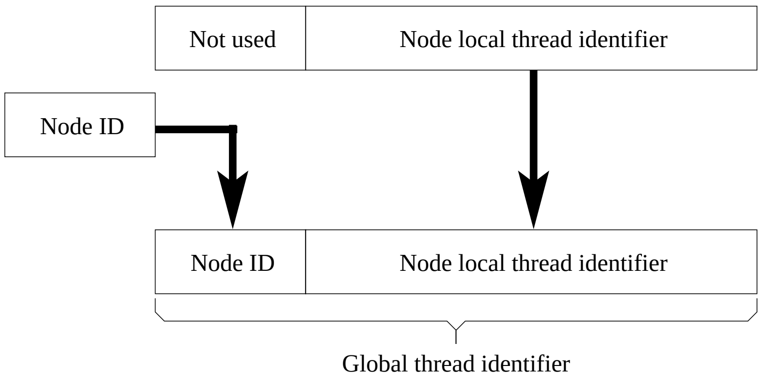


Figure 6.4 Adding global information to the thread identifier of both Linux and Windows NT/2000 (TM).

required functionality, like a global virtual address space, cross–cluster synchronization operations, and cluster management mechanisms are already present and simply have to be specialized in order to fulfill the semantic requirements of a distributed thread library.

The main component that still needs to be implemented is a forwarding mechanism that allows the transparent management of potentially remote threads. Any routine that has the capability of modifying the state of another thread is intercepted, the location of the specified thread is determined based on the global identifier introduced above, and the request is handed on to the node on which the thread is currently executing. Routines that only affect the own thread, in contrast, are directly executed on the same node resulting in virtually no overhead.

The only exception from these guidelines is represented by the routines responsible for spawning of a new thread within the current process (*CreateThread* for the Win32 thread API and *posix_create* for the POSIX counterpart). Here the target node on which the new thread will be created has to be determined by the routine itself. Before returning to the user, the respective thread creation routine converts the local thread identifier returned by the operating system call into a global thread identifier. This new identifier is then returned to be used for any future reference to the newly created thread.

The location for a new thread is currently decided using a standard round-robin scheme based on the global activity counter provided by the task module. For the future, a more flexible and load-adaptive thread placement scheme is envisioned. This will help boost the performance of applications with inherent large scale parallelism and in performance heterogenous clusters or non–dedicated environments.

6.4.4 Memory Consistency Model

Even though any thread library is by default intended and normally implemented for CC–NUMA or UMA systems, they normally do not require a fully coherent memory subsystem. Often the respective standards specifically include descriptions of the minimal requirements, which lead to models similar to *Weak Consistency* (see Chapter 4.1.2). Examples for this are the POSIX threads [219] or the Java™ threads [104].

They all specify a set of routines from their API, which upon completing their respective tasks need to provide a consistent memory state on the node they were called from. These routines, also called consistency points due to their effect on the local memories, are mostly routines which provide some kind of global synchronization, like mutex or thread creation routines. To implement this consistency model, the consistency point routines are enhanced by both read invalidations and write flushes, resulting in the intended memory consistency at the respective points of execution.

It should, however, be pointed out that multithreaded applications written for SMPs or similar tightly coupled architectures might not be implemented according to the specified memory models. Often they rely on the stronger hardware enforced consistency for correct execution. When porting such non–compliant applications to a thread library with a weaker, but still standard compliant, memory model, problems will occur in the form of races, stale memory regions, and missing update propagations. This can only be fixed by introducing additional synchronization points into the application transforming it into a standard compliant code.

One example for such a problem is for example present in some of the SPLASH–2 benchmark codes [233]. These codes were originally intended for a CC–NUMA architecture, namely the FLASH architecture developed at Stanford University [128], but are now used in various versions as a general shared memory benchmark suite. Within these codes most accesses to shared data are cleanly protected by mutex or other mechanisms with implicit synchronization points. However, at some rare locations, a few of the applications poll on a global variable for certain system wide events. This polling is done without any synchronization construct to avoid the high overhead of such operations. On a cache coherent architecture with sequential consistency this approach is feasible and probably the most efficient ones available. The polling variable will be contained in the first level cache during the poll loop, hence even eliminating any effect on system wide bus resources. On a write to this synchronization variable by any processor, the cache coherency controller invalidates the copy in the cache that is being polled on, triggering a reload at the next access and hence delivering the new value without the need for any further synchronization. This means the actual write itself is used as the synchronization point.

In a relaxed consistency model, however, this write does not trigger a remote invalidation of the synchronization variable and therefore does not carry the implicit synchronization as in the case of sequential consistency. The polling processor does not see the new value causing it to continue polling indefinitely. This can only be changed with additional synchronization constructs being introduced, e.g. by a mutex around any access to the synchronization variable. This, however, has to be done with great care, as the cost of very fine grain synchronization is very high and may cause, depending on the application and its behavior, high overheads.

In order to avoid these overheads, several options exist within the HAMSTER framework. These, however, are normally not anymore compliant with the respective thread API and lead in most cases to new extensions to the programming model. They therefore sacrifice full transparency for optimal performance and hence should only be applied as a last step of incremental and optional tuning.

One option is to directly add the respective flush and invalidate routines from the consistency module at the locations in the code at which either new values need to be propagated or stale values could affect the execution. This would avoid the actual synchronization operation and its global impact, while still guaranteeing correct program behavior. The result could be seen as a new, more efficient application–specific consistency model, adapted to the special synchronization constructs of the application.

While this option does not involve any major code changes and can be realized by just adding a few calls to the code, a second option, the allocation of special synchronization variables in a memory region with higher coherency guarantees, is most likely to involve more changes. Here the variables have to be specifically allocated and made available to the application in an ordered fashion. On the other hand, this option is likely to improve the overall performance, as no additional consistency enforcing mechanism needs to be applied when accessing the synchronization variable. This reduces the overall impact of consistency mechanisms as fewer system wide flushes or invalidations are performed.

6.4.5 Extended Synchronization

Besides standard locks used for the implementation of mutexes and critical regions, an additional very typical part of the synchronization repertoire of most thread APIs are notification mechanisms. These allow a thread to block and wait for an event to be triggered by another thread. Despite this common functionality, the actual semantics varies greatly between the various thread APIs making it difficult for HAMSTER to provide common services to support this in a platform independent way. The implementation of these notification mechanisms is therefore currently not part of the HAMSTER base functionality (or more specific the synchronization module). They are rather realized by the respective implementations of the transparent thread programming model and are based on the existing notification services of the OS. This maintains their original OS specific semantics.

Notification services are provided in the POSIX standard [219] in the form of condition variables. Any thread can use such variables to block awaiting the condition to be set by any remote thread. This setting of the condition is hereby done using a signal like mechanism, as is available in any modern UNIX operating system. In the Win32 API [156], a similar functionality is available in the form of so–called *EVENTS*. Also here threads can block until an event is signaled by another thread and any thread can set or reset events.

The main difference between the two models lies in the fact that Win32 events are persistent, i.e. an event can maintain a set status even if no thread is currently waiting, whereas POSIX conditional variables are not persistent leading to a signal being lost when no thread is waiting on it at the time it is sent. In addition, the Win32 event mechanism is richer as any system resource, including processes, threads, and files, are abstracted events and can be blocked on awaiting some kind of event associated with this resource (e.g. the termination of a process). In addition, the Win32 API allows waiting for multiple resources at the same time in an atomic fashion.

In order to deal with these severe semantic differences, the distributed thread APIs currently existing on top of HAMSTER implement notification using the forwarding mechanism discussed above. Each notification variable, independent of whether it is a POSIX condition variable or a Win32 event, is realized on one of the nodes in a home–based fashion using the original OS API. Any operation on this notification variable is then sent to this home node of the variable and executed there. Even though this implementation scheme causes additional communication overhead, it guarantees the identical semantics to the original native API and hence a correct program execution.

6.4.6 Available APIs and Limitations

As mentioned above, the HAMSTER framework currently includes distributed thread APIs for both Windows NT/2000™ Win32 [154] and Linux/POSIX threads [197]. This selection was made as these two models are predominantly used for existing applications on top of these target systems. Other APIs, however, like the portable thread layers mentioned above, could be implemented in a very similar fashion and are expected to exhibit similar characteristics in both performance and functionality.

Work on the distributed thread APIs was initially done within the SISCI project [207, 50] already mentioned in Chapter 2.3. Its goal was to create a complete software infrastructure for SCI–based cluster system consisting of both message passing and shared memory APIs, the latter part being covered by these thread APIs on top of the HAMSTER system [192, 194].

The transparency of these two thread APIs is, however, not complete thereby leaving some differences between the semantics of the native and the HAMSTER based API. One source for these differences comes from the fact that in addition to these purely thread–based services, the thread APIs in both operating systems also offer several routines for the efficient cooperation between several multithreaded processes. These routines are not included in this work as the SCI-VM solely deals with providing a global memory abstraction of a single process. This also restricts any programming model on top of it to be single process oriented. This restriction, however, is of rather low importance for the execution of typical multithreaded, computationally intensive applications and therefore does not impose a major restriction to the overall approach.

A second difference can be found in how the distributed threads treat I/O. As the global process abstraction created by the SCI-VM and the cluster control module does not include full I/O transparency, this limitation is also present in the distributed thread APIs. Any I/O resource allocated by any thread is therefore only visible in the team hosting the particular thread and can not safely be shared between other threads. This restricts the I/O patterns used by multithreaded applications on top of HAMSTER mainly to models with one thread being solely responsible for I/O (typically the initial thread), while all others function purely as computational threads without any I/O. This model, however, is very common among multithreaded applications in the area of computationally intensive codes targeted by this work.

6.4.7 Performance Evaluation

In order to validate the approach of distributed multithreading as it has been described above and to highlight some special performance details resulting from the integration of inter–node shared memory with local multithreading, a few experiments have been conducted using the distributed Win32 thread API. All of them were executed on the same platform as the other experiments so far, although with the older PCI-SCI adapter card D310 from Dolphin ICS.

Two different numerical kernels were used: a Successive OverRelaxation (SOR) and an LU decomposition. Both codes are based on SMP oriented counterparts from various shared memory benchmark suites and were only slightly modified and adapted to the Win32 thread API. In addition, as the data locality exhibited by the code has a major influence on the performance, the data placement hints present in the original codes were taken into account. Due to the flexibility of the utilized HAMSTER framework, this optimization did not require major code changes, but merely some minor annotation at the memory allocation points in the code. The relaxed consistency model was applied only through the implicit mechanisms described above without requiring any code modifications.

The first code that has been examined is the Successive OverRelaxation (SOR) — a numerical kernel that is used to iteratively solve partial differential equations. This is a quite simple test code capable of showing the basic performance of the overall system. The speedup of the application is shown in Figure 6.5 for the two different matrix sizes 1024x1024 and 2048x2048. The results are split into two different lines, one in which each node was used as a single node with one thread each, and one in which the SMP capabilities are used by spawning two threads per node. Both curves show good scaling behavior, with the better scaling properties for the larger data set.

Interesting is that the speedup is higher for the single processor testcase than for the SMP experiments. Especially worth noting is that the execution of two threads on two separate nodes on top of the SCI-VM exceeds the performance of two threads running on top of the native thread API on a single node. This behavior comes from the fact that in the latter case both processors access the same processor bus and therefore cause more contention than in the first case, where each processor has its own dedicated processor bus.

This observation can also be seen in more detail in the work decomposition of the executed code shown in Figure 6.6. This graph breaks the complete execution time of the code down into the significant subparts: the actual work (assumed to be the same in all cases), the shared memory access overhead caused by the HW–DSM and local processor bus contention, the overhead caused by having to enforce consistency in the relaxed consistency model (measured by the difference to a second, incoherent run), and the barrier wait time. In the case of two threads per node (left columns) the access overhead is significantly larger due to bus contention. This also causes the consistency overhead to be higher since also this operation triggers memory updates. In the single processor case, the access overhead is much smaller and the time needed to enforce consistency is almost negligible.

The second code that was examined is the LU decomposition, one of the fundamental algorithms used in computational numerics. It is used to factorize a dense matrix into

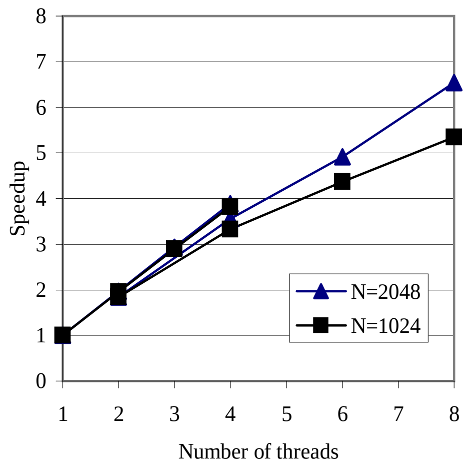


Figure 6.5 Speedup of the SOR code with up to 8 threads (4 nodes/8 CPUs).

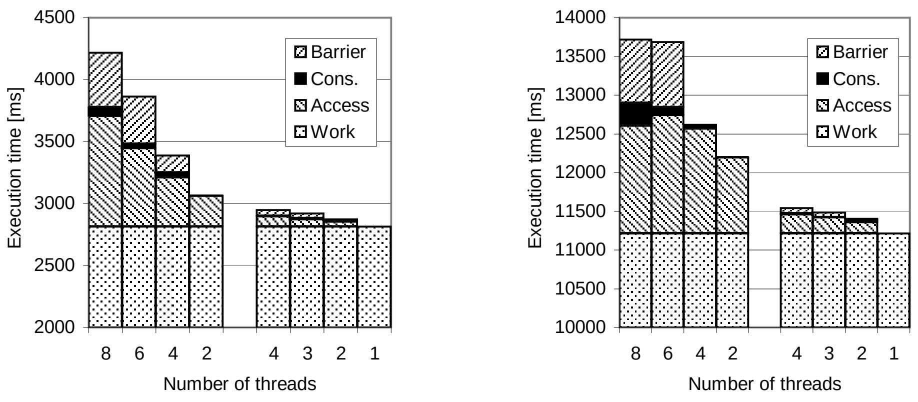


Figure 6.6 Work distribution of the SOR code with up to 8 threads (4 nodes/8 CPUs) — 1024x1024 matrix (left) and 2048x2048 matrix (right).

two diagonal matrices. This method can be applied to solve arbitrary systems of linear equations. The code, which is much more complex than the SOR code, was also executed on up to four nodes with two processor each and yielded similar results as the SOR code, which are shown in Figures 6.7 and 6.8. In contrast to the SOR code, however, it was impossible to assess the consistency overhead, as an execution without consistency enforcing mechanisms distorted the execution behavior too much to achieve reasonable results.

Due to the larger complexity of the algorithm and the longer execution time, the results are less distinct than in the SOR case. However, also here the execution with two threads per SMP node induces higher access cost and therefore higher overhead. The main performance problem in the LU code, however, lies in the application's scaling properties, as it can be seen for the small data set on more than 6 processors. From this point on, the barrier time increases significantly resulting in a lower parallel efficiency, as can be seen in the

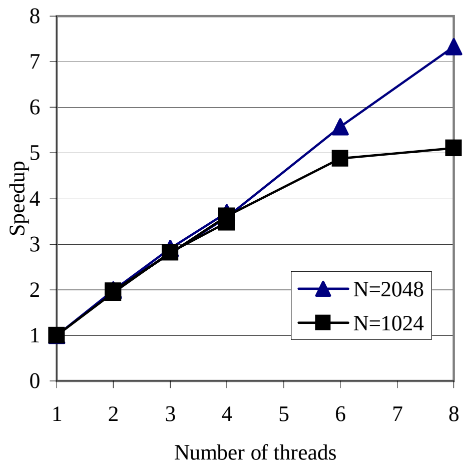


Figure 6.7 Speedup of the LU code with up to 8 threads (4 nodes/8 CPUs).

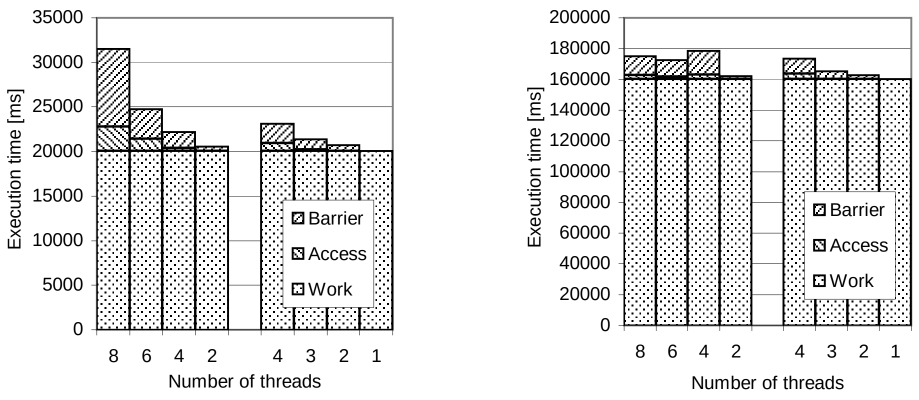


Figure 6.8 Work distribution of the LU code with up to 8 threads (4 nodes/8 CPUs) — 1024x1024 matrix (left) and 2048x2048 matrix (right).

speedup curve. It can be expected that the same can be observed for the larger data set with more than 8 nodes. This, however, is a limitation of the application rather than one of the execution environment.

In summary, the experiments show that the distributed thread APIs implemented within the HAMSTER framework allow the efficient execution of multithreaded applications on cluster architectures. They extend their scalability beyond the standard dual SMP configurations found in the PC world by taking advantage of both the intra-node SMP capabilities and the cost effective scaling of commodity clusters without having to sacrifice the easy–to–use programming model.

6.5 Support for Explicit Shared Memory Models

Any programming model discussed so far in this work adheres to the strict definition of a shared memory programming model set forth in Chapter 2.1.1. They all are based on a single identical view on the virtual memory transparently provided by the underlying system. In the case of an implementation within the HAMSTER framework, this is taken care of by the SCI Virtual Memory or SCI-VM (see Chapter 4).

In addition to those models, however, a few other programming models exist that, while utilizing principles from the shared memory world, do not exhibit the characteristics of a transparent access to the global address space. Rather users have to explictly specify such accesses to remote data within the global shared memory. These models, summarized under the name "explicit shared memory programming", can also be implemented within the HAMSTER framework. However, they require a different implementation strategy than the systems discussed so far.

6.5.1 The Cray T3D/E Put/Get Programming Interface

One example for such an explicit programming model is the *shmem* library [98] introduced by Cray and mainly intended for their T3D/E™ MPPs [235]. This API is by now also available on most SGI and Cray MPPs and has therefore found its user community. It implements a put/get programming model which allows programmers to access the virtual address range of any node using explicit read (*Get*) or write (*Put*) operations. In addition, the API provides a comprehensive set of synchronization and collective operations completing the overall programming model.

The programming interface

As already mentioned above, the *shmem* programming model does not rely on a single global virtual address space, but rather implements independent address spaces in each process on each node. These can then be accessed using the special one–sided communication routines *Put* and *Get* (see also Table 6.3) which allow the direct transfer of data to and from a remote memory location without interrupting the execution on the target node. Using these routines, both static application data as well as dynamic heap data can be accessed and modified.

In addition to these transfer routines, the shmem programming model also contains several routines for the efficient support of collective operations, which are also listed in Table 6.3. This includes the functionality to broadcast data to all nodes, to collect and concatenate data from all nodes, and to perform a data aggregation using a given operation across a given set of nodes. These operations contain all typical commutative and associative routines like sum, product, computation of maximum and minimum as well as the logical operations and, or, and xor.

Both, the collective operations and the transfer routines are available for a broad range of different types and different transfer lengths. This broadens the API and provides additional safety for the programmer since it enables explicit type checks.

Name	Description
Put	write data into remote memory
Get	read data from remote memory
Broadcast	broadcast data to all nodes
Collect	combine data from all nodes
OP_to_all	perform operation OP across all nodes OP = sum,prod,and,or,xor,max,min
Barrier	perform a barrier over a set of threads
Set_lock	acquire a global lock
Clear_lock	release a global lock
Num_pes	query the number of participating nodes
My_pe	query the rank of the local node

Table 6.3 Main routines provided by the *Shmem* programming model (names simplified)

The API is completed with the typical lock and barrier routines known from other shared memory programming models as well as with routines to query the number of nodes participating and the rank of the local node.

Architectural support on the T3D/E™

The T3D/E™ systems [235], for which this library was mainly intended, are very much suited for such a programming approach since their architecture matches this model very closely and can hence directly exploit its advantages. Its NUMA architecture allows the direct access to the address space on any remote node and can thereby directly perform any *Put* or *Get* operation without influencing the remote node. In addition, the T3D/E™ systems contain special hardware mechanisms for an efficient implementation of global operations and therefore allow a high–performance implementation of synchronization and collective operations.

This beneficial match between architecture and programming model has made the put/get model an important programming model for these architectures. Next to message passing libraries it has found its place in the overall software infrastructure for NUMA–like MPPs.

6.5.2 Implementing Put/Get in HAMSTER

This success motivates the investigation on how to implement this explicit shared memory programming model within the HAMSTER framework. It opens SCI–based clusters to a new class of applications and allows them to be either used as low–cost alternatives to large–scale MPPs or as efficient front–end systems for MPPs, which are available for application development and debugging.

Most of the functionality of the *shmem* interface, like the synchronization constructs or the collective operations, can easily be implemented using the standard mechanisms already introduced. In contrast to the other programming models, however, this implementation also requires the ability to access memory locations in the virtual address space of other nodes. This can easily be achieved for newly allocated data since it can be allocated in a globally visible address space and thereby be made available to any node. Dealing with the static application data, however, is more complex, as it is by default controlled by the local operating system instances rather than by the SCI-VM system.

For this purpose, the memory management module offers the explicit distribution of static data. After applying this type of distribution, the static data of every node is made globally visible through appropriate additional mappings. Every node can now directly access this space on any other node and hence perform the required *Put* and *Get* operations.

In contrast to the models discussed so far, which rely on a fully transparent data distribution, put/get models contain implicit descriptions for an optimal distribution. This can be drawn directly from the semantics of the *Put* or *Get* operations, which imply a direction of transfer. *Put* operations write to remote locations while *Get* operations read remote data. In both cases, the referenced data has been allocated and is under the control of the respective remote node. Hence, any data, whether part of the static application data or newly allocated, should be physically placed on the particular node containing or allocating the data. This leads to a consistent mapping of the logical location of data seen from the programmer's point of view and the physical data layout and thereby to a predictable application performance. In addition, this programming model, as in general any explicit shared memory programming model, does not rely on transparent caching since any data transfer is handled explictly. This allows the use of uncached memory resulting in a higher write performance due to improved write combining (see also Chapter 4.7).

Both this data placement and memory type selection can easily be achieved with the data distribution and coherency annotations provided by the HAMSTER memory management module. Together with the already discussed explicit distribution for static application data, the HAMSTER system therefore provides a suitable implementation platform even for explicit shared memory programming models and thereby proves the wide applicability of this approach.

6.5.3 Performance Aspects

Using the implementation guidelines described above, a subset of the Cray *shmem* library has been implemented on top of HAMSTER. It provides the main routines for *Put* and *Get*, as well as most of the synchronization routines and collective operations and therefore satisfies the requirements of many applications. For this section, the put/get model in its current state has been evaluated using a few small benchmarks showing the performance of the put/get and some collective operations.

Figure 6.9 shows the raw bandwidth that has been achieved using the *Put* and *Get* routines. Both exhibit a performance close to the raw performance measured on top of the SCI-VM (see Chapter 4.7). The performance of the *Get* routine is directly limited by the SCI read performance and does not exceed 1 MB/s. The *Put* bandwidth, however, profits from the high SCI throughput. Due to the explicit nature of the communication, the

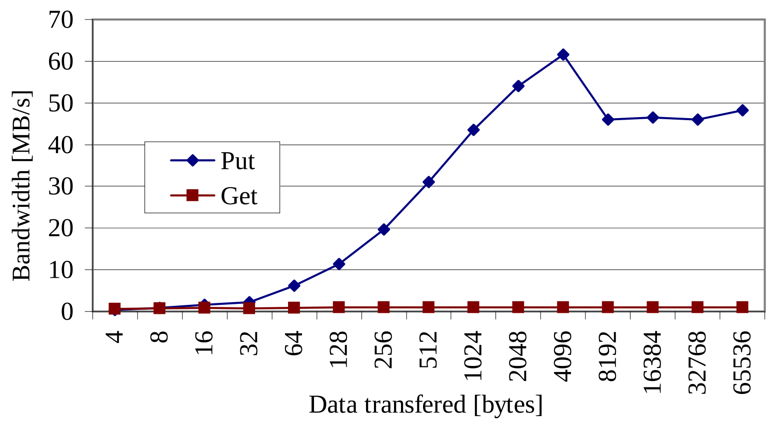


Figure 6.9 Put/Get bandwidth.

routine is also capable of exploiting the write combining optimization using explicit write cache flushes mentioned in Chapter 4.7 and thereby exceeds the default SCI-VM write performance. Compared to the maximal peak bandwidth (around 85 MB/s), however, the *Put* operations infers an overhead due to necessary address translations and a full store barrier after the communication ensuring the data has fully arrived on the receiver. It could be expected that with larger data transfer sizes this overhead will be amortized. However, with transfers longer than a page (4096 bytes) the same write combining anomaly as in the raw experiments can be observed, limiting the performance on longer transfer to close to 50 MB/s.

The second set of experiments examines the performance of the broadcast mechanism. It is implemented as a series of consecutive transfers to the target nodes using several *Put* operations. This basic implementation scheme is clearly visible in the performance, as it is shown in Figure 6.10 for both large and small transfer sizes. The time needed for a broadcast rises both with increasing data transfer sizes and with the rising number of target nodes. The only exception can be seen for a transfer of 32 bytes, as the SCI protocol [92] does not contain 32 byte packets and the transfer therefore needs to be split into two 16 byte packets.

Besides broadcasts the *shmem* library also offers further, more sophisticated global operations, most importantly collective operations across sets of processors. These have been evaluated using a collective reduction across various numbers of nodes and with a different number of concurrent global reductions ranging from 1 to 64. The results for this experiment are shown in Figure 6.11. As expected, the execution time for the global reduction is nearly linear to both number of nodes and the number of reduction operations and ranges from nearly zero execution time for a local operation on one node to around 1350 μs for 64 reductions with data from 6 nodes.

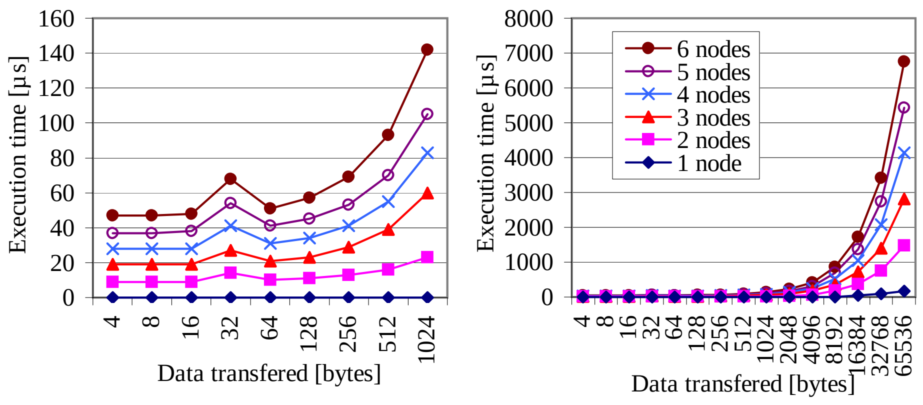


Figure 6.10 Broadcast performance with varying numbers of target nodes and data transfer sizes (small transfer sizes left, large transfer sizes right).

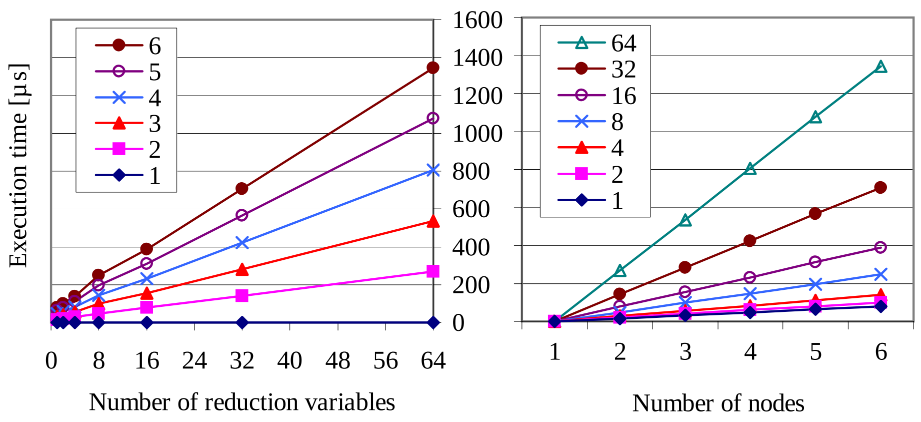


Figure 6.11 Performance of collective addition across various number of nodes (left) and using various numbers of reduction variables (right).

Programming Model	#Lines	#API calls	Lines/call	Platform
SPMD model	502	23	21.8	NT & Linux
SMP/SPMD model	581	25	23.2	NT & Linux
ANL macros	146	20	7.3	NT & Linux
TreadMarks API	326	13	25.1	NT & Linux
HLRC API[3]	137	25	5.5	NT & Linux
JiaJia API (subset)[3]	43	7	6.1	Linux
Dist. POSIX threads	725	51	14.2	Linux
WIN32 threads	988	42	23.5	NT
Cray put/get (shmem) API	505	29	17.4	Linux

Table 6.4 Implementation complexity of programming models using HAMSTER.

6.6 Implementation Complexity Analysis

As the HAMSTER system was designed with the intention to support a large set of different programming models on top of a single architecture, the system was built in a way that allows new models to be implemented with low effort. For this purpose the individual management modules offer a wide range of services, each associated with a large parameter space. Often the functionality required by the target programming model can therefore be achieved by specializing a similar service of HAMSTER. Normally only a few specialized API calls have to be implemented separately, as has been demonstrated in the implementation discussions in this chapter.

An indication of the complexity of the programming models implemented using the HAMSTER framework is given in Table 6.4. This list also contains several models, which have been implemented within HAMSTER, but have not discussed above. This includes an extended SPMD model capable of running with multiple threads on SMP nodes (SMP/SPMD), a direct implementation of the ANL macros used in the SPLASH–2 benchmark suite [233], and the emulation of APIs from existing software DSM systems like JiaJia [83] and HLRC [179].

The information presented in the table shows the lines of code necessary to implement each programming model in relation to the complexity of the programming model expressed through the number of available API calls. While such a measure is by nature highly inaccurate, it anyway gives a first impression of the amount of work required for such an implementation. The table lists most of the programming models currently available on top of the HAMSTER framework. These programming models differ greatly in terms of execution model, memory management, transparency, and consistency model.

Despite the clear functional differences, Table 6.4 shows that neither of the programming models required extreme effort to be implemented on top of HAMSTER; on average not more than 25 lines of code had to be written per API call for most of the programming models. The most complex example for such a special routine is the forwarding mechanisms present in both thread APIs available on top of HAMSTER (see also Chapter 6.4.3). This leads to the rather large number of lines necessary to implement these two models, which, however, is compensated by the large number of routines necessary in these models, again resulting in a quite low number of lines per API call.

Keeping the necessary effort to create a HAMSTER–based shared memory model at such a low level, opens the possibility to easily add a large variety of models to the HAMSTER framework. Only the specifics of the individual model have to be implemented or deduced from the general mechanisms of HAMSTER through specialization; the actual core with all its complexity of typical DSM systems stays the same for any programming model. This approach therefore has the potential to unify the large number of existing shared memory models on top of a single system with a single core.

[3]Based on the SPMD programming model.

6.7 Visions for Further HAMSTER–based Models

So far the discussion of programming models has concentrated on implementing existing models on top of the HAMSTER framework. It is, however, also possible to utilize HAMSTER for the implementation of new programming models or as a runtime system for higher–level programming abstractions or languages. This will be discussed in more detail within this section.

6.7.1 Application or Domain Specific Programming Models

The complexity analysis in the previous chapter shows that the HAMSTER approach allows the implementation of new programming models with very little complexity. This does not only enable the creation of a large set of existing programming models, but provides the opportunity to establish new domain or even application specific models at only marginally extra cost. This opens the possibility for reducing the implementation complexity of the application itself, by providing special primitives already within the programming model, to implement low–level optimizations at system level, and to adjust the environment to special requirements from the ground up. This has the potential to lead to both a higher efficiency with respect to performance and coding complexity at application level.

The easiest way to reach such an optimized programming model is to start at a given programming model in principle suited for the intended target application or domain and extend it. In the simplest case, these extension can export some of the underlying HAMSTER facilities to the application. Especially promising with respect to potential impact on performance are hereby services from the area of the memory allocation and distribution policies as well as the low–level and direct consistency control. The first group gives applications control over the memory distribution in the system allowing for an adaptation to its implicit data locality properties and memory access patterns, while the second group enables the creation of application specific consistency models with weaker consistency guarantees as the general ones discussed above. This can help to reduce unnecessary calls to consistency enforcing invalidations or flushes eliminating their global impact on the overall application.

One example for such a custom programming model has already been discussed above in the context of the flag synchronization in multithreaded programs. There, additional functionality added to the existing API, either in the form of explicit consistency mechanisms or through designated, uncached flag variables, have enabled a domain specific optimization.

This idea can be taken a step further by not only providing a few extra shared memory services for performance reasons, but also by including task and memory constructs from an application or typical representatives of application domains. This can go as far as directly providing a complete task management framework combined with implicit data management. This would lead to an evolution from a programming model towards a full runtime system with complete control over the application, but implemented on top of HAMSTER and using its shared memory as its prime information sharing platform without necessarily exporting it directly to the user.

As of now, no such domain specific model has been developed on top of the HAMSTER environment, as the work so far has concentrated on providing existing programming models for better code portability and therefore forming a corner stone in the overall software infrastructure for SCI–based clusters. In addition, the evaluation of the HAMSTER system so far required to a large degree the use of special kernels and benchmarks from given suites in order to expose performance details and to stay comparable with other work.

Only few actual real–world applications have been used within the HAMSTER project so far. Chapter 7 will list two of them and discuss their performance. Both are from the field of medical imaging, a very promising field for the utilization of high–performance computing. These two applications, as well as a few more from the same field that have not been investigated within HAMSTER yet, exhibit similar properties with respect to I/O requirements, task structure, and data management and therefore have a good potential to profit from such a domain specific programming model or system in the future.

6.7.2 HAMSTER as Runtime Backend

Another area of potential target programming models are shared memory programming languages with implicit memory and task management. Those systems normally rely on their own compiler, which could be adapted to either produce code for a programming model available on top of HAMSTER or to directly generate HAMSTER code. This would result in using the HAMSTER system as a runtime system for the particular programming system.

Currently two of such shared memory languages have found a significant user community, OpenMP [172] and High Performance Fortran (HPF) [236], even though many more exist especially as research systems. Both OpenMP and HPF are based on a global process and memory abstraction and provide explicit pragmas to the user which can be used to specify sharing and concurrency patterns. However, the fundamental concept of parallelism varies greatly between the two — while HPF is geared towards data parallelism and emphasizes on data distribution primitives along with a simple "owner computes" rule, OpenMP supports task parallelism based on the master/slave principle. Both approaches, though, can be implemented on top of HAMSTER due to the flexibility in the task and memory management modules, opening the possibility for easy ports of both OpenMP and HPF systems to NUMA enabled architectures, including SCI–based clusters.

6.8 Summary

This chapter has shown that the HAMSTER system is capable of supporting a large variety of different shared memory programming models with different goals, target application areas, and characteristics. The basis for any implementation is formed by the HAMSTER interface, which combines all services exported by the HAMSTER management modules with additional support mechanisms, such as portable timing.

The available programming models range from distributed thread libraries allowing true multithreading on cluster platforms all the way to explicit shared memory programming models in the form of a put/get library. It also includes APIs from many known SW–DSM systems providing direct code portability from any of these to HAMSTER–based platforms. This chapter has also shown that, due to the wide variety and flexibility of the services offered by the HAMSTER interface, each programming model can be implemented without much effort or complexity. This enables the quick and efficient creation of many different programming models on top of a single platform and thereby fulfills one of the main goals behind the overall system, namely the elimination of the major problems associated with the large number of existing shared memory programming models.

In addition, all programming models built on top of HAMSTER, despite their functional differences, directly rely on the same efficient DSM core, the SCI-VM. Any kind of shared memory access is directly handled by this core without the need for any further intervention by the programming model layer. This guarantees that all programming models exhibit equal behavior with regard to memory accesses and therefore automatically provides, in addition to easy code portability to HAMSTER, also performance portability between HAMSTER models. This alleviates the programmer of programming models from the burden of extensive tuning of the management code, as it is typically necessary with SW–DSM system or protocol implementations.

Chapter 7

Application Performance Evaluation

The evaluation of a complex system like HAMSTER based only on benchmark suites alone, however, is not enough. It should also include experiments with a few large scale, real–world codes with a concrete application background. For this purpose, the following sections present two applications from the field of nuclear medical imaging, more precisely the Positron Emission Tomography (PET). In this field several computationally challenging tasks exist which demand parallel processing in order to be solved in a suitable time. Two of these, the iterative reconstruction of images and their spectral analysis, are introduced and evaluated below.

Both applications have been developed and are provided by the "Nuklearmedizinische Klinik and Poliklinik"[1] at the "Klinikum Rechts der Isar", the university hospital of the "Technische Universität München". Within the context of a long–term cooperation with this institution, these applications have been used in various projects [210, 149, 113], including the one presented here.

7.1 The Principle of Positron Emission Tomography

Positron Emission Tomography is a relatively young technique in nuclear medical imaging and is used in addition to the traditional techniques like Computer Tomography (CT) or Magnetic Resonance Imaging (MRI) [77]. In contrast to those methods though, PET imaging enables images of the active processes inside a patient's body by using tracer techniques while the conventional methods only deliver information mainly about the anatomy. PET imaging therefore provides valuable functional information about a patient's metabolism, perfusion, or receptor density in tissue. The information acquired can be used to detect a large variety of diseases, including dementia, cancer, and problems in the myocardial flow.

The basic principle of PET is based on positron emitting substances, the so–called tracers, which are injected into a patients body and then monitored from the outside using a PET scanner. These tracers are typically derived from substances which play an important role in the human metabolism. The original substances are then marked or radiolabeled with positron emitting isotopes. Nevertheless, the tracer substances have the same chemical properties as their unlabeled counterparts and therefore fully participate in the patient's metabolism. Areas of high activity regarding a certain substance will therefore lead to a

[1]Roughly translated as "clinic for nuclear medicine"

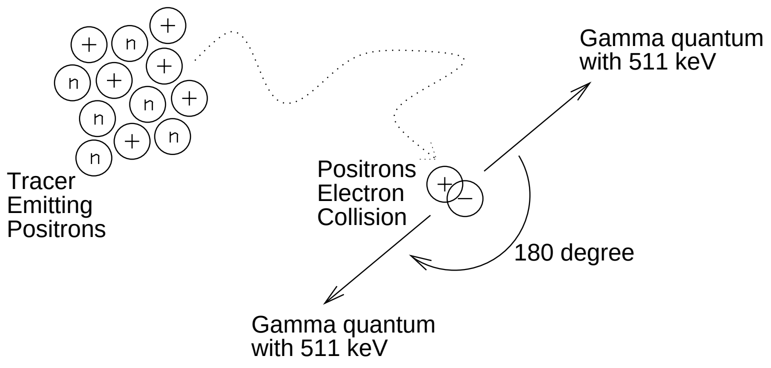


Figure 7.1 Positron / Electron annihilation (from [166]).

concentration of the corresponding tracer, which can then be detected. Due to this, PET enables the visualization of the human metabolism and allows the detection of deficiencies or malfunctions, which can indicate serious diseases.

For the detection of the tracer within the body, the property of the tracer to emit positrons is used. These positrons, which have a rather high interaction probability, will collide with free electrons nearby and the two charges will annihilate themselves leading to the creation of two gamma quanta (see also Figure 7.1). These quanta now have the property that they radiate away from the point of the positron / electron collision in exactly opposite directions, i.e. at an angle of 180 degrees. These quanta can then be detected by a surrounding PET scanner.

The scheme of such a PET scanner is shown in Figure 7.2. Its main component is a detector ring, which is capable of recognizing these gamma rays. The scanner hardware then filters the incoming events for timely coincidences since those have a high likelihood to belong to the same positron / electron annihilation and therefore indicate the presence of the original tracer substance at this point in the patient's body. The exact location, however, can not be known since only the coincidence line, the line between the two detectors which have recognized two coincident events, can be observed.

The information of all coincidence lines acquired during one scan is stored in a number of two dimensional image planes (orthogonal to the patient's body axis). These image planes, however, do not represent geometric coordinates, but are based on the angle of coincidence lines and their distance from the center of the scanner. The results are so–called Sinogram planes; an example for such a plane is shown 7.3.

The original data represents integral values of the activity distribution and needs to be converted into volume information representing the underlying real distribution. This is done in the so–called image reconstruction step. Furthermore time sequences monitoring the change in activity distribution can be analyzed by e.g. Spectral Analysis resulting in functional parameter characteristics for the metabolic status of the tissue.

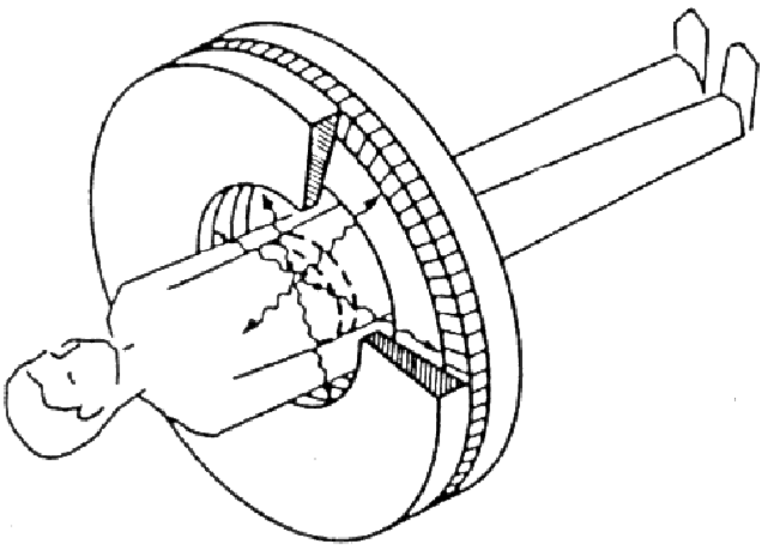


Figure 7.2 Schematic structure of a PET scanner (from [166]).

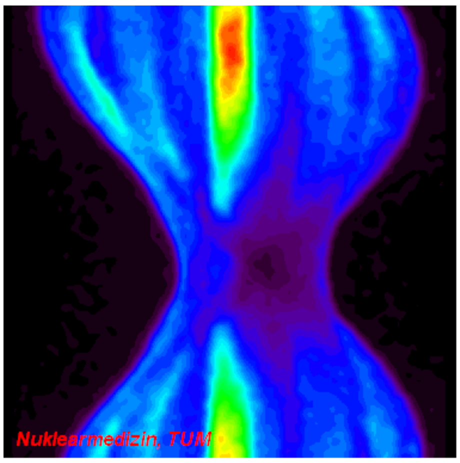


Figure 7.3 Typical sinogram as delivered by the PET scanner before reconstruction.

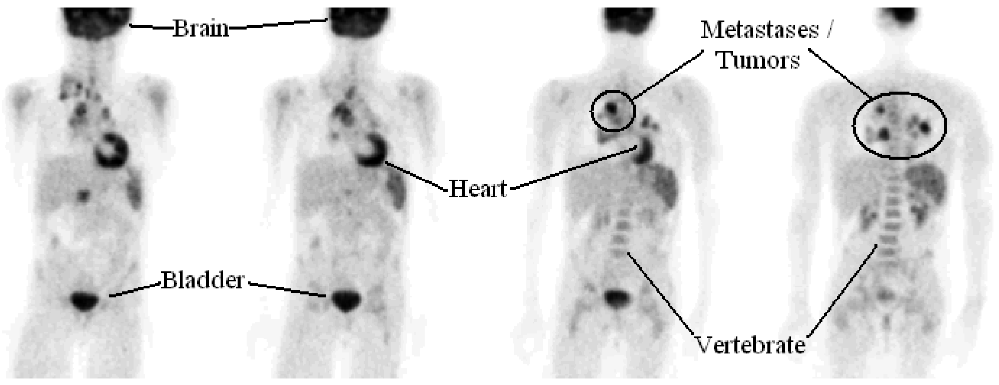


Figure 7.4 Reconstructed and post–processed images of a human body, tracer F18 labeled glucose analog (FDG) (coronal slices, Nuklearmedizin TUM).

7.2 Reconstruction of PET Images

As described above, the first step in the post–processing of medical PET data is its reconstruction from projection space (sinograms) to image space (voxels). The starting point for this process are the sinogram data planes acquired during the PET scan. These are transformed during the reconstruction process into a three dimensional voxel set. The result can then be visualized by creating various two dimensional cuts through the volume, as shown in Figure 7.4 for a whole body data set, or by using sophisticated volume rendering algorithms [57, 220, 71]. In both cases the resulting images enable a diagnosis based on the PET image taken from the patient in treatment, e.g. defining the extent of a tumor or the number of metastases.

7.2.1 Iterative PET Reconstruction

The traditional way to accomplish the reconstruction of PET images is the so–called Filtered Back Projection (FBP). This method simply projects the acquired sinogram data into the two dimensional space in a straightforward and direct manner. While this approach can be implemented without much complexity and has therefore been used in most commercial systems around the world, it has the distinct disadvantage of a low image quality with many artefacts, especially in areas with very high tracer concentrations. This can be seen on the left side of Figure 7.5. In recent years, research has therefore focused on new algorithms using an iterative approach [132, 54], which produce images of much higher quality.

In these methods, the reconstruction starts with a first approximation of the final image. Often the result of a FBP is used for this purpose. From this image, the corresponding scanner output (in the projection space) is computed using a forward projection. The resulting sinogram data is compared with the data acquired from the actual PET scan and the image approximation is adapted accordingly. This process is repeated until the projected sinogram data has become sufficiently close to the real data at which time the image approximation is accepted as the final reconstructed image.

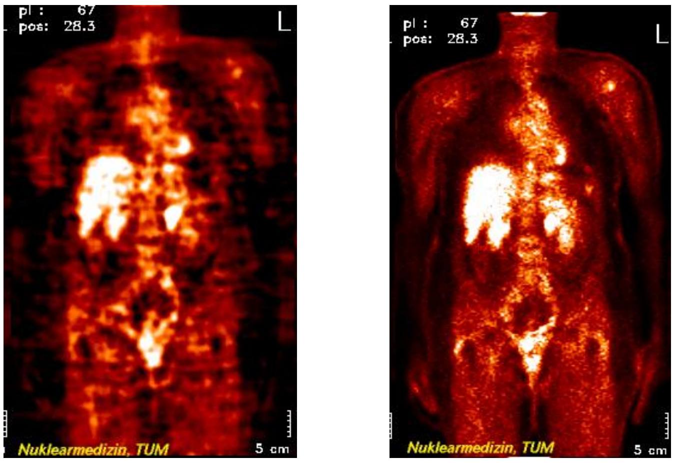


Figure 7.5 Comparing the image quality of FBP (left) and an iterative reconstruction (right), both with tracer F18 labeled glucose analog (FDG).

These iterative approaches provide a significantly better image quality as can be seen on the right side of Figure 7.5. Compared to the outcome of the FBP, this image contains less artefacts and noise and provides new anatomical details not visible before. This improved quality comes from the fact that during the forward projection process additional information about the scanner, like scanner geometry and background data, can be taken into account. In addition, this method allows to take the discretization of the acquired data into account [166].

The drawback of these approaches, however, is their high computational demand stemming from the iterative concept. This renders them unsuitable for current stand–alone workstations; only parallel processing is able to provide the necessary performance required for these iterative image reconstruction algorithm to succeed.

7.2.2 Parallel Implementation

The parallelization of this application used in this work is based on the reconstruction application developed at the Klinikum Rechts der Isar [166, 211]. It uses a special reconstruction library called ASPIRE [54] which has been designed and implemented at the University of Michigan and supports several reconstruction algorithms with different forward projection and image approximation methods.

This library performs the reconstruction of the PET image independently for each image plane, i.e. each sinogram plane is transformed into an image plane in the final image and this process is executed for each input image plane. This property can be used for a straightforward parallelization of this application using an SPMD style. First a master thread reads the complete input data set into a piece of global memory. Each thread is then assigned an equal number of planes to reconstruct. For each of these planes, the corresponding thread reads the input data from the global memory, performs the reconstruction

Description	Whole body
Size	130 Mbyte
Planes	282
Scan Res.	256x192
Image Res.	128x128
Scans	6

Table 7.1 Parameters for the data set used in the experiments.

using the ASPIRE library, and then stores the resulting image plane again in global memory. After all threads have completed their tasks, the final image is read from the global memory by the master thread and stored to disk.

The implementation of this concept is done using the SPMD programming model implemented on top of HAMSTER discussed in Chapter 6.2 and Appendix B. In addition, one slight optimization has been performed regarding the distribution of the input data set in the global memory. It is distributed in a way that the data for an image plane is mostly colocated with the thread to which the plane is assigned . As a result, during the I/O phase the raw data will directly be transfered to the node responsible for its further processing using the more efficient remote writes, instead of having to rely on read operations during the reconstruction phase. This optimization is performed using only the locality annotations of the memory management module by applying a block cyclic data distribution with block sizes equal to the size of an input plane; no further code changes were necessary.

7.2.3 Performance Discussion

For the evaluation of this application on top of HAMSTER, an example data set containing the scanned information of a whole human body is used. It was acquired during several consecutive scans of the different body sections, which are then merged into one full image. Its complete size and resolution shown in Table 7.1 and some resulting image slices from a reconstructed volume are shown in Figure 7.6.

The code has been executed on up to 6 nodes with up to 2 threads each. Figure 7.7 shows the results of this experiment in terms of speedup compared to a sequential execution without HAMSTER. It shows that this code exhibits good scaling properties with a speedup of more than 5 on 6 nodes with 1 thread each and of more than 8 on 6 nodes with 2 threads each. In addition, the curves are fairly linear and do only show a slight degradation with higher numbers of nodes. Due to this, it can be expected that this application will also scale beyond the 12 CPUs available in the SMiLE system. It can also be observed that the performance on configurations with one thread per node is slightly better than on those with two threads (assuming equal number of total threads). This is due to the fact that the reconstruction algorithm imposes high demands on the system bus leading to a contention with two compute threads present on one node.

In any case, a significant part of the total overhead is caused by the file I/O required to

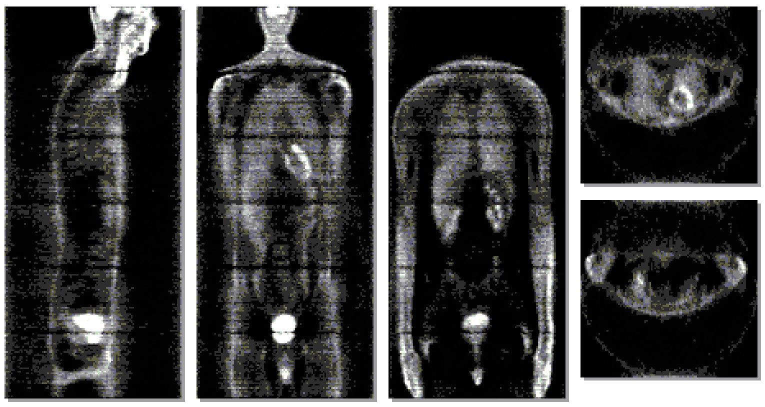


Figure 7.6 Reconstructed images — data set whole body (before proper alignment of neighboring scans and without attenuation correction; the borders between the 6 individual scans are still visible).

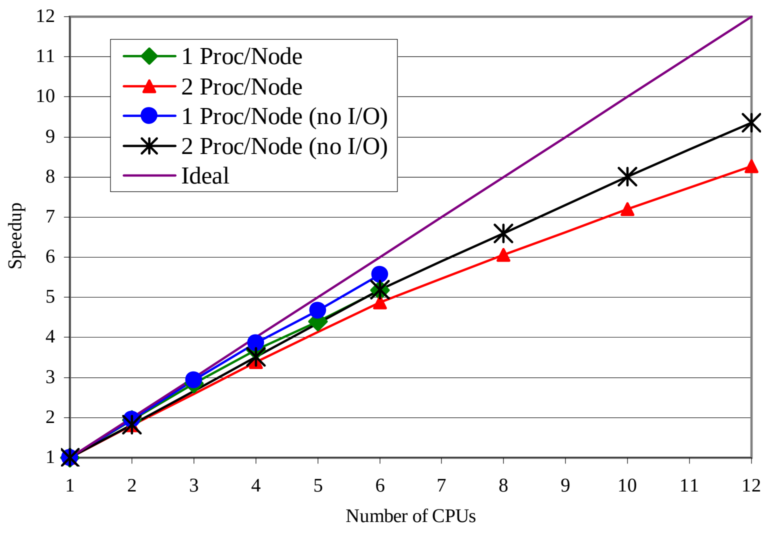


Figure 7.7 Speedup of the PET image reconstruction on up to 12 CPUs.

deal with the rather large input and output data sets. This phase is currently implemented in a sequential fashion and hence limits the possible overall speedup [4]. This deficiency, however, is neither induced by the shared memory programming paradigm nor the HAMSTER system. Therefore, Figure 7.7 also includes a speedup curve without the impact of this phase, which allows the separate assessment of the performance within the actual computational phases. It shows that the actual computation exhibits a speedup of close to 10 and therefore excellent scaling capabilities.

For a further in–depth discussion of these results, Figure 7.8 shows the time spent during the execution of the application split into the four major phases reconstruction, weights

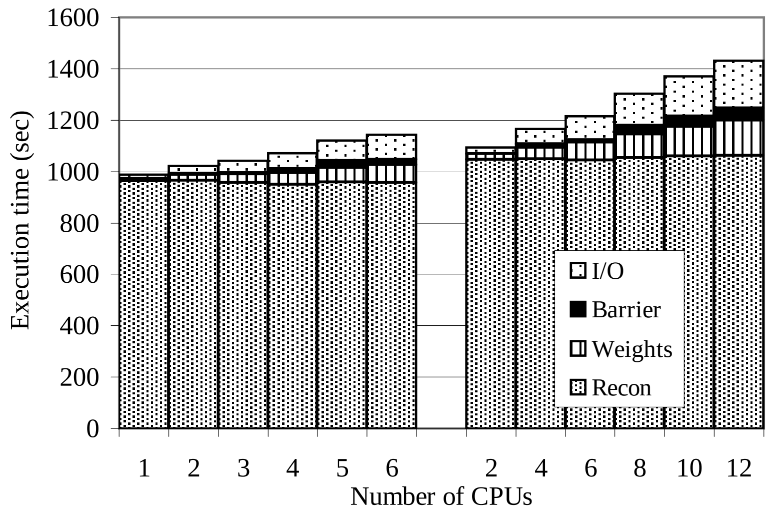


Figure 7.8 Aggregated execution times for the PET image reconstruction (left: 1 thread/node, right: 2 threads/node).

(a necessary preprocessing step), final barrier to ensure global completion, and file I/O. All times shown are aggregated across all threads and hence present the complete amount of work performed during the execution. This data shows that the actual reconstruction phase is fully scalable across all numbers of nodes, as can be expected due to the independent computation of the individual planes. In the case of just one thread per node (left side of the figure), the reconstruction times are even slightly improved due to the increased memory bandwidth and cache size available in the overall system. In a two thread scenario, on the other hand, the reconstruction time is burdened with a constant overhead caused by the memory contention within the SMP nodes. In addition, the computational load is rather balanced across all nodes resulting in low barrier times.

The actual overheads associated with the code are introduced by the weights and the I/O phase. In its current version, both are purely sequential and hence the total aggregated work increases linearly with the number of threads. While this impact can not be avoided with regard to the weights phase, as this is an integral part of the overall application, the I/O phase could be further optimized, e.g. by using parallel I/O.

7.3 Spectral Analysis of PET Images

Once this first step of image reconstruction is completed, further analysis steps can be taken in order to enable a more detailed measurement of physical parameters. One of them is the spectral analysis of a series of images over time [42, 224]. Again, this is a computationally quite challenging task requiring the use of parallel processing.

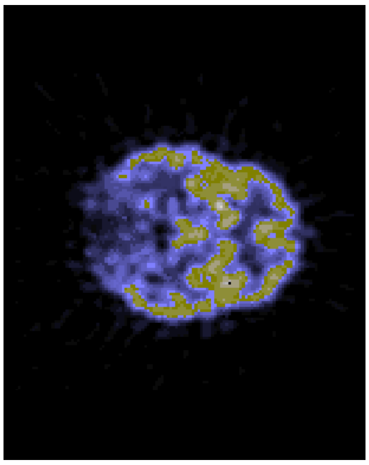


Figure 7.9 Sample slice of a human head, representing the impulse response function at 60 min after injection, tracer: dopamin receptor legand C–11 diprenorphin.

7.3.1 Application Description

Spectral Analysis is a new and innovative analysis procedure for PET images first introduced by [42]. It can be used to compute a transitional function, the so–called Impulse Response Function (IRF), which allows the evaluation of a tracer's concentration within the patient's body over time (e.g. for up to 60 minutes, depending on the tracer). It uses a series of PET scans taken from the same body part at different times as input together with a separately measured curve describing the concentration of the tracer in the patients bloodstream and uses this information to compute the IRF.

Of special importance for a further medical diagnosis based on the acquired information are the values of the IRF at 1 minute and at 60 minutes as well as an approximation of the integral of the curve with respect to time. These three values are extracted from the IRF at the end of the computation for each image point and stored as the final result.

One of the strongholds of this approach is that the IRF is computed for each pixel independently and hence enables a fine granularity and a high resolution for the resulting data. Consequently, the computation is performed independently on each series of pixels at a particular location in the 3D volume over time and also the three resulting IRF values are again stored as new 3D volumes. Like above, the resulting volumes can be viewed using volume rendering techniques. An example for this can be seen in Figure 7.9.

7.3.2 Parallel Implementation

Like the PET reconstruction, this code can also be implemented using a data parallel approach. However, the finer data granularity required by the algorithm enables the splitting of individual planes, rather than distributing the whole plane as in the PET reconstruction code. Like above, a master thread is responsible to read the input data for each plane and to store it in global memory. This data consists of the individually reconstructed image planes from all scans acquired during the study. Each node then works on its own part of each image plane and reports its contribution in a piece of global memory. This leads to a finer granularity of the parallelization and hence to a potentially more balanced parallel execution with respect to system load.

The code used for the following experiments is based on an algorithm developed at MRC Cyclotron Unit of the Hammersmith Hospital, London and has already been implemented at the Klinikum Rechts der Isar as a shared memory application [224] using the SW–DSM system TreadMarks [5]. The port to HAMSTER is therefore easily accomplished by using the TreadMarks API implemented on top of HAMSTER. The new code just has to be relinked with this implementation of the TreadMarks API in order to reach a final executable [199].

Furthermore, in order to maintain the transparency of the underlying TreadMarks API, no optimizations like locality annotations or custom consistency models have been applied. The experiments conducted using this code are therefore a good indication of the ability to easily move an existing application code to the HAMSTER platform by providing the corresponding programming model.

7.3.3 Performance Discussion

For all measurements a single real–world data set has been used. Its data was acquired by a sequence of PET scans of a human head over a time period of 25 samples. Each sample consists of 47 individual slices of 128x128 pixels. The total raw size of the input data is roughly 38.5 Mbyte. After the computation, three 3D images are written back to disk consisting again of 47 individual slice images at a resolution of 128x128 pixels. The total output volume per written image is roughly 1.5 Mbyte.

The code has been evaluated using the same scenarios as with the PET reconstruction described above. Figure 7.10 shows the resulting speedups over the sequential execution without HAMSTER. As above, the graph includes data for both the full application and the computational phases only without the impact of the sequential file I/O. The data shows good scaling properties with a speedup of more than 8.5 on 12 CPUs for the complete application. In contrast to the situation above, the performance with one thread per node is slightly lower than with two threads per node (again assuming the same total number of threads). This stems from the fact that the Spectral Analysis has only limited memory bandwidth requirements and hence does not saturate the system bus, even with two threads per node. Due to its fine grain access granularity, it rather benefits from the low read latencies of intra–node shared memory which leads to the observed performance benefits.

This is also visible from Figure 7.11, which shows the aggregated execution times across all participating threads split into the main program phases: the actual work performed during the spectral analysis, which is assumed to be equal across all configurations, the overhead induced by using NUMA-DSM, computed as the difference between the measured execution time and the actual work, the file I/O phase, and the waiting time during barriers. The graph shows that the overhead times, which are caused by both implicit NUMA communication and consistency enforcing mechanisms and hence introduced

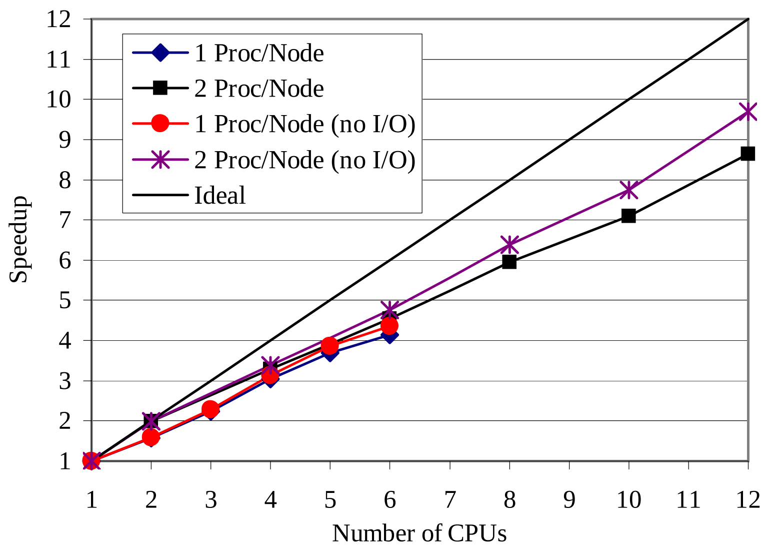


Figure 7.10 Speedup of the Spectral Analysis on up to 12 CPUs.

through the HAMSTER framework, are generally slightly lower in the cases with two threads per node.

In addition, the figure shows that the overhead values vary across the different numbers of nodes which can be attributed to the different memory layouts created by the transparent memory management of the SCI-VM. This can inherently influence the overall data locality and thereby also the overall performance.

The two remaining phases, the I/O and the barrier, behave as expected. The first phase, the aggregated barrier time, is rather low across all configurations due to only little load imbalances. With raising numbers of nodes, however, this impact increases slightly. The other phase, the file I/O phase, is executed during a sequential piece of code and its execution time increases linearly over the number of utilized threads.

7.4 Summary

This chapter has shown the applicability of the HAMSTER framework in a real–world scenario. Two large scale applications from the area of nuclear medical imaging, the reconstruction of PET images and their spectral analysis, have successfully been ported to HAMSTER and have exhibited good performance and scalability properties. The performance is mainly hindered by external circumstance, most dominant I/O. Both applications work on large amounts of data resulting in a significant amount of time spent during this particular phase.

The parallelization processes of these applications have been fundamentally different and show the two main usage scenarios of HAMSTER. The PET image reconstruction software has been parallelized from scratch enabling the utilization of a new and adapted API, in this case the SPMD programming model. This has also allowed the use of special

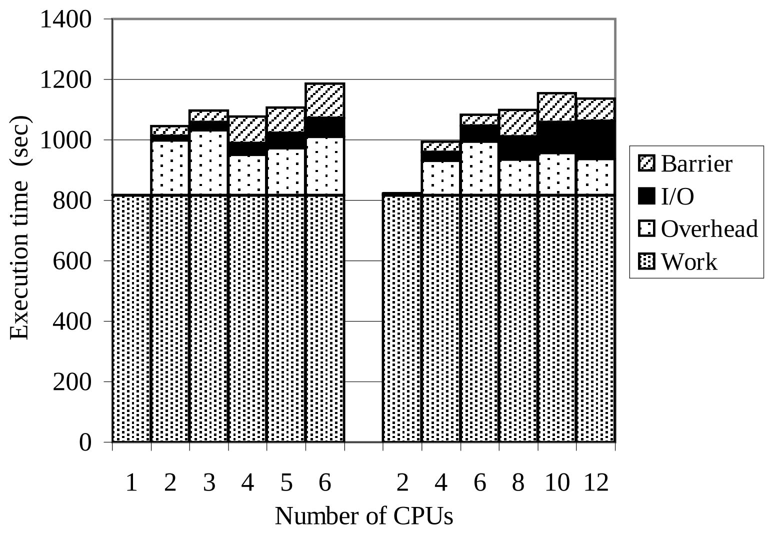


Figure 7.11 Aggregated execution times for the Spectral Analysis (left: 1 thread/node, right: 2 threads/node).

optimization mechanisms provided by HAMSTER which are generally not exposed by many existing programming models. The spectral analysis application, on the other side, has already existed as a shared memory application using the TreadMarks [5] system. Its port to HAMSTER was therefore facilitated by the existence of this particular programming model on top of HAMSTER and was accomplished by simply recompiling and linking with the HAMSTER–based API. This shows that the HAMSTER system is capable of supporting both new and existing APIs on top of a single framework and hence eases both the parallelization of new applications and the porting of existing shared memory codes. This was one of the major design goals behind the HAMSTER framework.

These promising results encourage the work on further large scale applications. Potential targets will be further applications from the area of nuclear medical imaging like image realignment procedures or statistical image evaluation algorithms as well as applications from the area of computer graphics, more specifically volume rendering [228]. The latter application domain poses many problems suitable for a shared memory parallelization, as large data sets are randomly accessed by the different threads. They can also be combined with the already discussed medical imaging applications as an additional post–processing step enabling a high–quality visual presentation of the processed data.

Chapter 8

Conclusions and Outlook

The widespread use of shared memory programming for High Performance Computing (HPC) is currently hindered by two main factors: the limited scalability of architectures with hardware support for shared memory and the abundance of existing programming models. The former issue has sparked research in the area of DSM systems. These deploy complex software mechanisms to create a global memory abstraction in more scalable NORMA environments. However, they are often inherently connected with performance problems and, in addition, do not solve the second issue. In contrast, the large amount of work done in the DSM area has led to the development of a significant number of independent systems, each with its own API, thereby further worsening the situation.

In order to fully solve these problems, a comprehensive framework for shared memory programming on top of loosely coupled systems is required. In addition, architectures with a limited form of hardware support for shared memory, which does not destroy the scalability or cost advantages, should be the target for such a framework. This support can then be used to avoid the typical performance problems of pure software implementations.

A framework fulfilling these criteria, called HAMSTER (Hybrid-dsm based Adaptive and Modular Shared memory archiTEctuRe), has been designed, implemented, and evaluated within this work. It benefits from the advantages of loosely coupled non–CC–NUMA architectures, i.e. scalability and cost effectiveness, and exploits their hardware support to enable the low–complex implementation and efficient use of almost any shared memory programming model. The complete system is currently targeted towards SCI–based clusters, but its principal concepts are applicable to any other NUMA architecture. It thereby provides a general approach capable of filling this gap in the overall software infrastructure for loosely–coupled NUMA–based systems.

The core component of the HAMSTER framework is an efficient Hybrid–DSM system which is capable of closing the semantic gap between the global physical memory provided by the underlying architecture and the global virtual memory required for shared memory programming. It relies directly on the given hardware support for any communication and combines this with a software component responsible for creating the appropriate memory mappings as part of a cluster–wide memory management scheme. The finer granularity of the underlying hardware memory accesses thereby helps to avoid typical problems seen in SW–DSM systems like false sharing or complex software implementations of differential update protocols. It therefore allows the full exploitation of the hardware capabilities present in the underlying architecture.

Based on this Hybrid–DSM system, the HAMSTER framework defines and implements several independent and orthogonal management modules. This includes separate modules for memory, consistency, synchronization, and task management as well as for the control of the cluster and the global process abstraction. Each of these modules offers typical services required by implementations of shared memory programming models. Combined they form the HAMSTER interface which can then be used to implement the intended shared memory programming models without much effort.

This capability has been proven through the implementation of a number of selected shared memory programming models on top of the HAMSTER framework. These models range from transparently distributed thread models all the way to explicit put/get libraries and also include various APIs from existing SW–DSM systems with different relaxed consistency models. It therefore covers the complete spectrum of shared memory programming models and underlines the broad applicability of this approach.

The presented concepts have been evaluated using a large number of different benchmarks and kernels exhibiting the performance details of the individual components. In addition, HAMSTER was used as the basis for the implementation or port of two real–world applications from the area of nuclear medical imaging, more precisely the reconstruction of PET images and their spectral analysis. These experiments cover both the porting of an already existing shared memory application using a given DSM API and the parallelization of an application from scratch using a new, customized API. In both cases, the system provided an efficient platform resulting in a very scalable execution.

With the completion of the prototypical implementation and extensive evaluation presented in this thesis, the work on the HAMSTER system will not be abandoned. It has sparked many new ideas and offers the potential for further research. Two of the potential future directions are briefly presented within this chapter, along with a discussion of the overall system's applicability to current hardware implementations. The chapter concludes with a few final remarks on the future potentials of HAMSTER.

8.1 Applicability

The HAMSTER framework in its current state is a fully working prototype system. All mechanisms are implemented and have been demonstrated in the various experiments throughout this work. However, the description of the HAMSTER system has also shown that the current system still has to cope with some shortcomings, especially with regard to the SCI-VM, as described in Chapter 4.8.

Because of these open issues, the HAMSTER system is at the moment not yet in a state to serve as the basis of production–type environments. For this, both the operating system integration and the transparency of the underlying hardware need to be improved. Only then can a stable and reliable environment required for real–world application scenarios be achieved.

Despite these problems, however, the HAMSTER framework represents a general and open multimodel shared memory programming environment for NUMA systems. Its concepts are applicable far beyond its current implementation on top of SCI and can in principle be ported to any NUMA–type architecture without major changes. It therefore has the potential to generally enable shared memory programming on the rising class of NUMA–based systems.

8.2 Future Directions

Besides the removal of the implementation challenges mentioned earlier and the transformation of the HAMSTER framework into a safe and reliable production environment both on SCI and on other NUMA architectures, the system offers several opportunities for further basic research. The two most promising ones among them are briefly discussed below.

8.2.1 Towards a Cluster–enabled Operating System

Shared memory programming, as defined for this work, is associated with a single global address space and therefore also requires a global process abstraction. Within HAMSTER, this abstraction is created by the cluster control module which has the capabilities of merging the individual teams on each node to a global entity. This forms a suitable environment for the implementation of shared memory programming models, as it is the goal of this work.

The global process abstraction, however, is currently not complete. It covers only the creation of a single global address space enabling the transparent placement and location independent execution of threads within this global abstraction. Other operating system services, which are normally also transparently available throughout a process, are not covered and need to be explicitly considered when porting codes to HAMSTER–based environments. The most severe among them is the missing transparency of I/O. Due to the distinct operating system instances, this is still handled node–local.

In order to overcome this deficiency, a transparent, global I/O framework needs to be developed and coupled with HAMSTER. Such a framework needs to be capable of transparently intercepting I/O operations that are intended for other nodes and forward their execution to the appropriate target. This greatly reduces the semantic gap between the individual operating system instances on the different nodes and in fact represents a first step towards a cluster–aware, global operating system. This turns clusters, normally loosely coupled with regard to both hardware and software, into single machines with a single operating system instance thereby providing users with an easy–to–comprehend single system image. Nevertheless, the beneficial properties with regard to scaling and cost efficiency of the underlying architectures are preserved, making such systems a potentially good trade–off between scalable hardware and easy–to–use software.

8.2.2 A Tool Environment for On–line Monitoring

The various experiments throughout this work have shown that some applications require incremental performance tuning in order to show a significant speedup. Specifically the locality of memory accesses plays an important role in the overall performance. Therefore, the memory management, as it has been described in Chapter 5.1, includes mechanisms to specify the memory distribution at allocation time using locality annotations. While this approach has proven to be useful in optimizing applications, it is burdened with two inherent disadvantages: for one, the user has to add the annotations into the codes by hand (often destroying the full transparency of the underlying programming model) and secondly the annotations are static and do not allow a dynamic adaptation to irregular memory access patterns. To avoid these problems, it would be desirable to manage data locality dynamically through an appropriate adaptive runtime system.

Work in this direction has already been started within the SMiLE project with the goal of developing a comprehensive on–line monitoring infrastructure for shared memory programming environments. On the hardware side, a special monitor capable of observing the memory traffic on the SCI network is under development [78]. The data acquired through this device will then be fed into a monitoring software stack that is designed to fit to the HAMSTER structure [111] and is built within OMIS [144], a general–purpose infrastructure for interoperable on-line monitoring and tool support. From there, data is made available to sophisticated higher–level tools, like a visualizer or a page and thread migration mechanism [217, 110].

8.3 Final Remarks

This work has provided a comprehensive overview on how to use shared memory programming in NUMA–based architectures emphasizing the efficient exploitation of the hardware capabilities present in these architectures. With the introduction of a novel Hybrid–DSM system and with its proposal for a general, open, and flexible shared memory framework capable of supporting almost any arbitrary shared memory programming model, it has opened a new field of research and has sparked new ideas. The list above briefly discusses some of the major ones among them. Follow–up work in several directions has already begun and will be continued in the future within the context of the SMiLE project.

However, the success beyond pure research and the applicability of the overall system is currently severely limited by the missing hardware transparency in the underlying architecture that can not be compensated in software. This restricts the system to a prototypical study and also prevents any use of the overall system for a production or commercial use limiting the impact of this work.

Nevertheless, the results achieved during this work are very promising and show both the principal feasibility of the concepts presented and the good prospects for their future use. In order to fully exploit them, however, the system needs to be moved to a more stable and reliable hardware platform in order to evolve into a mature state. This can mean both the use of future developments in the area of PC–based SCI clusters leading to error free hardware implementations or the port of the system presented here to other NUMA machines not suffering from these problems. Under these circumstances, the HAMSTER framework has the chance to fully exploit its potentials and to significantly promote shared memory programming, even for loosely coupled architectures.

Appendix A

The HAMSTER Execution Environment

The following section provides a brief overview on how to install and use the HAMSTER framework. The description is based on a binary distribution package containing both the binary version of the HAMSTER framework, the required tools, and the SPMD programming model in source. This package is available from the SMiLE software repository[1].

A.1 System Requirements

The current distribution is only available for Linux clusters. The individual cluster nodes have to be standard PCs (single or dual processors) and have to be equipped with Intel Xeon–II™ CPUs or compatible. The latter one is required in order to provide the necessary capabilities for a cache configuration at page granularity. All nodes have to be connected with both Ethernet and SCI. For the latter one, currently only the D320 adapter card (Dolphin firmware) is tested with the current distribution. The SCI system has to be configured as either a single ring or via a switch. Torus topologies are not supported.

On the software side, the HAMSTER distribution requires Linux with a 2.2 kernel (only 2.2.5 is tested, but others might work). It has been developed and tested with a SuSE 6.4 installation, but again others might work. In any case, all nodes must be installed with exactly the same configuration with respect to installed libraries. In addition, it is beneficial to provide a cluster–wide shared file system as a basis for the execution of HAMSTER codes.

Before proceeding with the installation of HAMSTER, the SCI network, the low–level drivers, and the SISCI API have to be installed, configured, and tested as described by Dolphin. In addition the *binutils* package needs to be installed in source and compiled.

A.2 HAMSTER Directory Structure

In order to install HAMSTER, the package needs to be extracted using tar. This creates a new directory containing the following subdirectories:

[1]http://smile.in.tum.de/software/

- *bin*
  This directory contains the necessary tools required to run HAMSTER. This includes the patch tools needed for the identification of the static application data.

- *include*
  This directory contains all include files needed for both the creation of programming models and for the compilation of applications making direct use of HAMSTER. This includes any application using the extended timing facilities of HAMSTER.

- *kernel*
  The HAMSTER system is based on a few kernel extensions. These are contained in this directory together with the required driver for the new memory management (*scivm.o*).

- *lib*
  In this directory the actual HAMSTER binaries are stored which need to be linked to the final applications.

- *make*
  This directory contains a set of makefiles and configuration files easing the compilation process of both programming models and applications.

- *model*
  Here all programming models developed for HAMSTER should be stored. In the original directory structure, this directory contains a subdirectory for the SPMD model and its example files.

- *sisci*
  The HAMSTER system requires some extension in the Dolphin drivers. Therefore, this directory contains a new set of drivers and a new API with these extensions integrated.

A.3 Installing the HAMSTER Environment

Due to the tight integration with the underlying operating system, the HAMSTER framework requires an extension to the Linux kernel. An appropriate patch together with instruction on how to apply it is included in the *kernel* subdirectory.

As a next step, the SCI driver structure needs to be replaced in order to include the necessary SCI driver extensions required for the direct ATT management and interrupt support. For this purpose, the installed SCI drivers and the SISCI API need to be replaced with the appropriate files in the *sisci* subdirectory.

Besides the new Dolphin drivers, also a special memory management driver allowing the system to implement direct page maps needs to be installed. The appropriate binary is included in the *kernel* directory along with a script which creates the necessary device files in */dev/scivm*.

The installation of the HAMSTER system is completed by adjusting the path informations in make.base contained in the make directory. More information on this is included in this file itself.

A.4 Linking Against HAMSTER

Any application running on top of a HAMSTER–based programming model needs to link against the libraries in the lib directory. In addition, both the SISCI API library and the *binutils* library needs to be included in the list of libraries. Any link should by done statically in order to ensure a consistent data layout across all nodes.

An example on how a programming model is compiled and how a sample application is compiled and linked is included in the distribution based on the SPMD programming model. It can be found in the *model/spmd* directory. There, the appropriate makefiles can be found which can be directly used to compile both the SPMD programming model and the sample application. For this purpose, only make needs to be started without any further arguments. These makefiles can also be used as the basis for further developments.

A.5 Running Applications

Before starting a HAMSTER–based application, first the cluster configuration needs to be specified. For this purpose, the user has to create a *.hamster* file in their home directory. This ASCII file should contain a line for each node to be used for the application. This text line has to contain the DNS name of the node, the SCI node ID, and the scale of the node (number of threads suitable for this node), each separated by a space. A sample configuration file can be found in the HAMSTER main directory.

After this step, the application is ready to run. For this, it has to be started on each node contained within the configuration file separately. On nodes listed twice within the configuration file, two copies of the application need to be started. The first node mentioned in the configuration has the role of a master during the initialization and therefore should be started last. As soon as all copies of the application have been started, the master initiates the initial communication between the individual application instances and merges them to a single global process abstraction.

Appendix B

SPMD: A Sample Programming Model on Top of HAMSTER

One of the programming models supported by the HAMSTER system is the SPMD (Single Program Multiple Data) model introduced in Chapter 6.2. As this is one of the leanest and simplest, it is used here as an example of a HAMSTER programming model. It is also included in the binary distribution of the HAMSTER system available from the SMiLE software repository[1]. The following chapter will introduce the specification of this programming model and provide a short guide on how to use it.

B.1 API Description

In the following, the specification of the SPMD API available to the programmer is briefly described.

Initialization and error control

The programming model uses an automatic initialization that is triggered during the first call to any routine. At this point any configuration of the individual HAMSTER modules used by the SPMD programming model is executed implicitly. Therefore, the user is not required to explicitly ensure proper initialization.

In order to keep this programming model simple and suited for small experiments with the HAMSTER system, it also contains a very simple error management. None of the routines report errors, but will rather lead to an abortion of the program. As any potential error that could be triggered by this programming model is either a fatal error caused by problems within the HAMSTER system, which would leave the application in a useless state, or by programming/parameters errors, this simplification in error management does not significantly restrict the usability of the programming model.

[1]http://smile.in.tum.de/software/

Global process abstraction control

spmd_getNodeNum

Call: unsigned int **spmd_getNodeNum**()

This routine returns the rank of the currently running thread within the current global process abstraction. This number can then be used to control the work or task distribution or restrict certain tasks, like I/O or application initialization, to only specific threads.

spmd_getNodeCount

Call: unsigned int **spmd_getNodeCount**()

This routine returns the total number of the currently running threads within the current global process abstraction. Together with the rank of the local thread, this number can then be used to control the work or task distribution.

Barrier synchronization

spmd_allocBarrier

Call: int **spmd_allocBarrier**()

This routine can be used to allocate a barrier. It has to be called by all threads in the system concurrently. The identifier for the newly allocated barrier is returned as an integer and can be used in subsequent *spmd_barrier* calls.

spmd_barrier

Call: void **spmd_barrier**(int *barrierid*)

This routine triggers a full barrier of all participating threads within the current global process abstraction. This means that every thread has to call this routine with the same argument before any thread is allowed to return from this routine and therefore to continue execution. This routine has to be passed an argument containing a valid barrier identifier previously allocated using *spmd_allocBarrier*. It has to be noted that a barrier includes both an *acquire* and a *release* operation (see Chapter 6.3.3), i.e. the enforcement of full local consistency. As a barrier call in this programming model involves all threads, it therefore also guarantees a global fully consistent memory of the complete global process abstraction.

Lock synchronization

spmd_allocLock

Call: int **spmd_allocLock**()

This routine can be used to allocate a lock. It has to be called by all threads in the system concurrently. The identifier for the newly allocated barrier is returned as an integer and can be used in subsequent *spmd_lock* and *spmd_unlock* calls.

spmd_lock

Call: void **spmd_lock**(int *lockid*)

This routine executes a lock operation. In case the lock is already taken by another thread, the call does not return until the other thread releases the lock and the current thread becomes the new owner. The routine has to be passed an argument containing a valid lock identifier previously allocated using spmd_allocLock. It has to be noted that a lock includes an *Acquire* operation (see Chapter 6.3.3), which when used together with the corresponding *Release* operation at unlock time creates a *Release Consistency* (RC) [116] model providing a consistent data access under the mutual exclusion of the used lock (see also Chapter 6.3.3).

spmd_unlock

Call: void **spmd_unlock**(int *lockid*)

This routine unlocks a lock held by the local thread making it available again to other, potentially already waiting threads. The routine has to be passed an argument containing a valid lock identifier previously allocated using spmd_allocLock. It has to be noted that an unlock includes a *Release* operation (see Chapter 5.2), which when used together with the corresponding *Acquire* operation at lock time creates a *Release Consistency* (RC) [116] model providing a consistent data access under the mutual exclusion of the used lock (see also Chapter 6.3.3).

Additional synchronization

spmd_sync

Call: void **spmd_sync**()

This routine enforces the consistency of the complete local memory by implicitly calling both an *Acquire* and a *Release* as discussed in Chapter 5.2. Note that this routine only affects the local memory and does not have a global effect. Note further that the SPMD programming model discussed here includes an implicit consistency management included in the barrier and lock routines. Therefore, the use of this routine should normally not be necessary.

Memory management

spmd_alloc

Call: void* **spmd_alloc**(int *size*)

The *spmd_alloc* routine can be used to allocate new global virtual memory for the application. The allocation is process is conducted using the default settings for memory coherency and distribution. This results in a piece of memory with a maximum of memory optimizations enabled distributed across all nodes at finest possible granularity. It has to be called by all threads in the system concurrently and returns the same virtual address of the newly allocated memory to all nodes.

spmd_allocOpt

Call: void* **spmd_allocOpt**(int *size*, localitySpec_t **locSpec*)

Like the routine above, also this routine allows the allocation of global memory. In contrast to above, however, the user has the option to specify parameters for guiding the allocation process. These parameters allow to influence the memory distribution as described in Chapter 5.1. They can be set through the *localitySpec_t* structure, which is passed to the routine as the second parameter. More information about the individual fields within the structure and on how to use them can be found in the SPMD header files.

spmd_allocCoh

Call: void* **spmd_allocCoh**(int *size*, coherencySpec_t **cohSpec*)

Also this routine allows the allocation of global memory, however, with the option to specify a memory coherency type for the newly allocated memory (using the *coherencySpec_t* structure). More information about the individual fields within the structure and on how to use them can be found in the SPMD header files.

spmd_allocOptCoh

Call: void* **spmd_allocOptCoh**(int *size*, localitySpec_t **locSpec*,coherencySpec_t **cohSpec*)

This routine combines the memory allocation functionality of the two previous routines and allows both locality and coherency specifications.

spmd_getGlobalMem

Call: unsigned int* **spmd_getGlobalMem**()

The last routine of the SPMD programming model allows the allocation of global memory from within the SCI-VM configuration space. This memory is always set to the highest possible memory coherence type and can serve for simple configuration and communication purposes. Strictly seen, the existence of this routine is not required, as the *spmd_allocOpt* also allows the allocation of memory with equal coherence type. Using this routine, however, no new memory needs to be allocated, as resources already in use within the SCI-VM are reused.

B.2 A Simple Example Code for the SPMD Programming Model

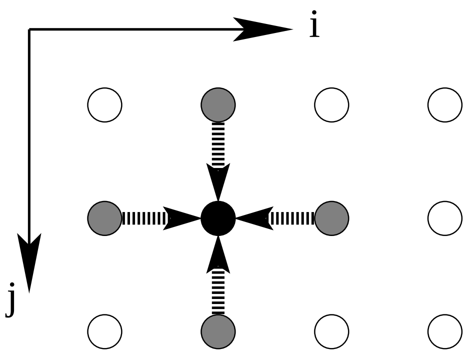
To help understand the programming model, this Section provides a small example parallel program using the SPMD programming model, a simple Successive OverRelaxation (SOR). This code has already been used in Chapter 5.1.3 and its performance issues have been discussed there. This source code is also included in the binary distribution of the HAMSTER software available in the SMiLE software repository at LRR–TUM.

A simple SOR code

A Successive OverRelaxation (SOR) is a common method to solve Partial Differential Equations (PDE) described by a dense matrix with boundary values. The SOR algorithm mainly consists of a loop executed a given number of iterations. Within this loop, each point of the dense matrix is updated once using a certain, PDE and problem specific update pattern or stencil defining how each node's new value has to be computed from the values of its neighboring points. For the code used here, a very simple stencil is used that just updates each point by the average value of all four direct neighbors. A graphical representation of this stencil can be seen in Figure B.1 along with its mathematical definition.

The parallelization of this SOR approach is rather straightforward and can be done using simple domain decomposition. This process is visualized in Figure B.2 from left to right based on the assumption of a parallelization for two nodes. The left most graphic shows the dense matrix on which the SOR operates. Then the necessary boundary is added around the actual data matrix. This matrix can then be split into equal parts according to the number of nodes, in this example two. Also the newly created subparts are again considered with their own private boundary allowing each subpart to be computed separately during one iteration by the same SOR algorithm. Between iterations, a barrier has to be introduced in order to keep all participating threads within the same iteration.

The right most graphic in the figure shows the overlap of data between the two independent partitions. This is exactly the data which is used by both threads leading to communication between the partitions.



$$U(i,j) = \frac{U(i-1,j)+U(i+1,j)+U(i,j-1)+U(i,j+1)}{4.0}$$

Figure B.1 Update pattern for each point in the matrix.

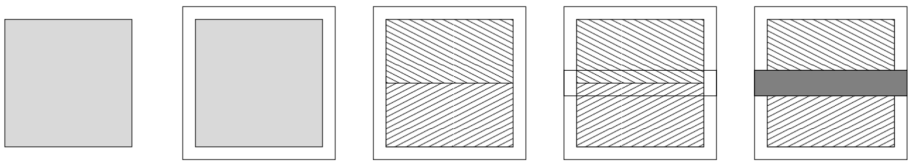


Figure B.2 Data distribution and sharing pattern for the SOR code — from left to right: initial dense matrix, boundary, splitting the matrix, local boundary, area with implicit communication.

It should be noted that this kind of parallelization does not result in a parallel program with exactly identical behavior as in the sequential case. In the parallel case the order in which matrix points are updated does not match the order of the sequential code. This also leads to different matrix values, as they are based on the values of the neighbors of which some might have been updated under one scheme, but not under the other. Due to the numerical properties of the SOR algorithm, however, this does not influence the final result after a convergence has been reached and is therefore safe to use.

Source code

The following section lists the source code implementing the SOR kernel described above using the SPMD programming model.

```
///////////////////////////////////////////////////////////////////////////////
//
//  EXAMPLE: small SOR example for SPMD programming model
//
//  September 2000, Version 1.0, (c) Martin Schulz, LRR-TUM
//
//  Usage:              Master: example <matrix size> <iterations>
//                      Slave:  example <matrix size> <iterations>
```

```
//
//  Linux version
//
////////////////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>

#include <spmd.h>

////////////////////////////////////////////////////////////////////////////////
// timing routines
double time_temp;
#define STARTTIME time_temp=getTimer()
#define ENDTIME (getTimer()-time_temp)
double getTimer() {
  struct timeval tv;
  struct timezone tz;
  gettimeofday(&tv, &tz);
  return ((double) tv.tv_usec)+(((double) tv.tv_sec)*1000000.0);
}

////////////////////////////////////////////////////////////////////////////////
// Matrix access macro
#define u(x,y) matrix[(x)*(matrix_n+2)+(y)]

////////////////////////////////////////////////////////////////////////////////
// Main routine
int main(int argc, char **argv) {
  float           *matrix;
  double          timer;
  localitySpec_t  locOpt;
  int             matrix_n, iterations, alloc_size;
  int             number, count, i, j, k, start, end;
  int             global_barrier;

  // Print header
  printf("\n\nSPMD example, v1.0\n\n");
  printf("(c) Martin Schulz, September 2000\n\n");
  if (argc!=3)
    {
      printf("Usage: example <matrix size> <iterations>\n");
      exit(1);
    }
  matrix_n=atoi(argv[1]);
  iterations=atoi(argv[2]);

  alloc_size=(matrix_n+2)*(matrix_n+2)*sizeof(float);
  printf("Allocation size in bytes:   %i\n",alloc_size);
  printf("Iterations:                 %i\n",iterations);
  printf("Matrix size:                %i\n\n",matrix_n);

  // Form here on, the SCI-VM is ready to use, but no global memory
  // has yet been allocated
  count =spmd_getNodeCount();
  number=spmd_getNodeNum();

  // compute my work
  start = (matrix_n / count) * number + 1;
  end   = (matrix_n / count) * (number+1) + 1;
  if (number==0)
    start=1;
  if (number==count-1)
    end=matrix_n + 1;
  printf("\nID %d, Working on rows %d - %d\n\n", number, start, end);

  // allocate barrier
  global_barrier=spmd_allocBarrier();

  // allocate global segment
  // locOpt.mode=SCIVM_BLOCKDIST;
  // locOpt.node=0;
  // matrix=(float*) spmd_allocOpt(alloc_size,&locOpt);
  matrix=(float*) spmd_allocOpt(alloc_size,NULL);
  spmd_barrier(global_barrier);

  // start initialization
  if (number==0) {
      for (j=0; j<matrix_n+2; j++)
        u(0,j)=1.0;
    }
  for (i=start; i<end; i++) {
      u(i,0)=1.0;
      for (j=1; j< matrix_n+1; j++)
        u(i,j)=0.0;
      u(i,matrix_n+1)=-1.0;
    }
  if (number==count-1) {
      for (j=0; j<matrix_n+2; j++)
        u(matrix_n+1,j)=-1.0;
    }

  // start timing
  spmd_barrier(global_barrier);
  STARTTIME;

  // Do SOR
  for (k=1; k<iterations; k++) {
      // do one SOR step
      for (i=start; i<end; i++)
        for (j=1; j<matrix_n+1; j++) {
            // do SOR for one point
            u(i,j) = (u(i+1,j)+u(i-1,j)+u(i,j+1)+u(i,j-1)) / 4.0;
          }
      spmd_barrier(global_barrier);
    }

  // end timing
  timer=ENDTIME;
  printf("Time elapsed %f ms\n\n",timer/1000.0);

  // print result diagonal
  if (number==0) {
      for (i=0; i<matrix_n+2; i++) {
          printf("u(%4d,%4d) = %f\n",i,i,u(i,i));
        }
    }
  printf("\n");

  // final barrier
  spmd_barrier(global_barrier);

  // That’s it
  spmd_stop();
}
```

Running the code

After the compilation of the example code and after linking it against the HAMSTER modules, the code is ready for execution. Appendix A describes the necessary procedure for this.

The code takes two arguments: first the size of the matrix (an integer number stating the number of elements in each direction) and the number of iterations, i.e. the number of SOR steps, to be executed. Note that only with an iteration number greater or equal than half of the matrix size the code will produce a fully computed dense result matrix. For initial tests, however, smaller iteration numbers might/should be used.

Sequential counterpart

For comparison, the binary distribution also includes a sequential counterpart of the code discussed above. It can be used to understand the parallelization process using the SPMD programming model and also for simple benchmarking and speedup evaluation.

Abbreviations

A

AM	Active Messages
ANL	Argonne National Laboratory
ANSI	American National Standards Institute
APC	Asynchronous Procedure Call
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ATT	Address Translation Table

C

CC–NUMA	Cache–Coherent Non–Uniform Memory Access
CERN	Centré Européenne pour la Recherche Nucléaire
CoP	Cluster of PCs
CORBA	COmmon Request Broker Architecture
COTS	Commodity–Of–The–Shelf
CPU	Central Processing Unit
CRL	C Region Library
CT	Computer Tomography
CVM	Coherent Virtual Machine

D

DEC	Digital Equipment Corporation
DLL	Dynamic Link Library
DMA	Direct Memory Access
DNS	Domain Name Service
DSM	Distributed Shared Memory
DUnX	Duke University nX2

E

EC	Entry Consistency
ECC	Error Correction Code
ERC	Eager Release Consistency

F

FBP	Filtered Back Projection
FIFO	First–In First–Out
FLASH	Flexible Architecture for SHared memory

G

GAC	Global Activity Counter

H

HAMSTER	Hybrid dsm–based Adaptive and Modular Shared memory archiTEctuRe
HP	Hewlett Packard
HPC	High Performance Computing
HPF	High Performance Fortran
HW–DSM	HardWare Distributed Shared Memory

I

IBM	International Business Machines
IEEE	Institute of Electrical and Electronics Engineers
ICS	InterConnect Solutions
ISS	Interconnect Systems Solutions
IRF	Impulse Response Function
IRM	Interconnect Resource Manager
IVY	Integrated shared Virtual memory at Yale

J

JDK	Java^{TM} Development Kit

L

LAN	Local Area Network
LC	Link Controller
LGF	Last Global Flush
LLI	Last Local Invalidation
LRC	Lazy Release Consistency

M

MPI	Message Passing Interface
MPP	Massively Parallel Processor
MRI	Magnetic Resonance Imaging
MTRR	Memory Type Range Register
MuSE	Multithreaded Scheduling Environment

N

NIC	Network Interface Card
NORMA	NO Remote Memory Access
NoW	Network of Workstations
NUMA	Non–Uniform Memory Access

O

OS	Operating System

P

PC	Processor Consistency *or* Personal Computer
PCI	Peripheral Component Interconnect
PDE	Partial Differential Equation
PET	Positron Emission Tomography
PoP	Pile of PCs
POSIX	Portable Operating System Interface for uniX
PSB	PCI–SCI Bridge chip
PVM	Parallel Virtual Machine
PVP	Parallel Vector Processor

R

RAL	Rutherford Appleton Laboratories
RAM	Random Access Memory
RC	Release Consistency
RPC	Remote Procedure Call

S

S2MP	Scalable Symmetric MultiProcessing
SAN	System Area Network
SAR	Synthetic Aperture Radar
SC	Sequential Consistency
SCI	Scalable Coherent Interface
SCI-VM	SCI Virtual Memory
ScC	Scope Consistency
SINTEF	Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology
SISCI	Standard software Infrastructure for SCI–based systems
SMiLE	Shared Memory in a LAN–like Environment
SMP	Symmetric MultiProcessor
SP	Service Pack
SPLASH	Stanford Parallel Applications for Shared Memory
SPMD	Single Program Multiple Data
SSI	Single System Image
SSP	Scali Software Platform
SOR	Successive Over Relaxation
SW–DSM	SoftWare Distributed Shared Memory

T

TSO	Total Store Order

U

UMA	Uniform Memory Access

V

VPM	Virtual Parallel Machine

W

WAN	Wide Area Network
WC	Weak Consistency

Bibliography

[1] G. Acher, W. Karl, and M. Leberecht. PCI-SCI-Protocol Translations: Applying Microprogrammable Concepts to FPGA. In R.W. Hartenstein and A. Keevallik, editors, *8th International Workshop on Field Programmable Logic and Applications, FPL'98*, volume 1482 of *Lecture Notes in Computer Science*, pages 99–108, Tallinn, Estonia, August 1998. Springer-Verlag.

[2] G. Acher, W. Karl, and M. Leberecht. *The TUM PCI/SCI Adapter*, chapter 4, pages 89–101. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[3] S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. Rice University ECE Technical Report 9512 and Western Research Laboratory Research Report 95/7, Department of Electrical and Computer Engineering, Rice University and Western Research Laboratory, DEC, September 1995.

[4] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[5] 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*, 29(2):18–28, February 1995.

[6] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *In Proceedings of the International Conference on High Performance Computer Architecture (HPCA)*, 1997.

[7] G. Antoniu, L. Bouge, and R. Namyst. An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System. In J. Rolim et al., editor, *Parallel and Distributed Processing, Proceedings of IPDPS workshops including RTSPP*, volume 1586 of *LNCS*, pages 496–510. Springer Verlag, Berlin, April 1999.

[8] G. Appollonia, J. Méhaut, R. Namyst, and Y. Denneulin. SCI and distributed multithreading: the PM2 approach. In H. Hellwagner and A. Reinefeld, editors,

Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98. Cheshire Henbury, September 1998. ISBN: 1-901864-02-02.

[9] Dolphin Interconnect Solutions AS. *Link Controller 3 (TM) Specification*. Olaf Helsets vei 6, P.O. Box 70, Bogerud, N-0621 Oslo, Norway, September 2000. Preliminary version 0.84, Available under NDA from Dolphin ICS.

[10] O. Aumage, L. Bougé, A. Denis, J. Méhaut, G. Mercier, R. Namyst, and L. Prylli. Madeleine II: a Portable and Efficient Communication Library for High–Performance Cluster Computing. In *In Proceedings of the IEEE Conference on Cluster Computing, Cluster 2000*, pages 78–87. IEEE Computer Society, December 2000.

[11] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):1,42,349, March 1992.

[12] A. Barak, O. La'adan, and A. Shiloh. Scalable Cluster Computing with MOSIX for LINUX. In *In the Proceedings of Linux Expo, Raleigh, NC*, pages 95–100, May 1999.

[13] A. Belias, A. Bogaerts, D. Botterill, J. Dawson, E. Denes, F. Giacomini, R. Hauser, C. Hortnagel, R. Hughes-Jones, S. Kolya, D. Mercer, R. Middleton, J. Schlereth, P. Werner, and D. Wickens. *SCI Prototyping for the Second Level Trigger System of the ATLAS Experiment*, chapter 23, pages 397–414. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[14] A. Belias, A. Bogaerts, D. Botterill, F. Giacomini, R. Hauser, R. Middleton, P. Werner, and F. Wickens. ATLAS Level–2 Trigger SCI Demonstrator Evaluation Report. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe 2000, The 3rd international conference on SCI–based technology and research*, pages 101–109. SINTEF Electronics and Cybernetics, August 2000. ISBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[15] A. Belias, A. Bogaerts, D. Botterill, F. Giacomini, R. Middleton, F. Wickens, and P. Werner. Evaluation of a 16–port SCI Switch. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe 2000, The 3rd international conference on SCI–based technology and research*, pages 101–109. SINTEF Electronics and Cybernetics, August 2000. ISBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[16] A. Belias, L. Iftode, and J. Singh. Shared Virtual Memory across SMP Nodes Using Automatic Update: Protocols and Performance. In *Proceedings of the 6th workshop on Scalable Shared–Memory Multiprocessors*, October 1996. Also as Princeton Technical Report TR-517-96.

[17] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type–Specific Memory Coherence. In *In Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, 1990.

[18] A. Blumrich, R. Alpert, Y. Chen, D. Clark, S. Damianakis, C. Dubnicki, E. Felten, L. Iftode, K. Li, M. Martonosi, and R. Shillner. Design Choices in the SHRIMP System: An Emprical Study. In *In Proceedings of the International Symposium of Computer Architecture (ISCA)*, 1998.

[19] M. Blumrich, R. Alpert, Y. Chen, D. Clark, S. Damianakis, C. Dubnicki, E. Felten, L. Iftode, K. Li, M. Martonosi, and R. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, May 1998.

[20] 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.

[21] B. Breshad and M. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, September 1991.

[22] B. Breshad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *In the Proceedings of COMPCON*, 1993.

[23] H. Bugge. Affordable Scalability using Multicubes. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 25–28. Cheshire Henbury, September 1998. ISBN: 1-901864-02-02.

[24] H. Bugge and P. Husoy. Efficient SAR processing on the Scali System. In *In the proceedings of the 2nd International Workshop on Embedded HPC Systems and Applications (held in conjunction with IPPS)*, April 1997.

[25] R. Butenuth and E. Rehling. Armenius: Software für Linux-basierte SCI-Cluster. In W. Rehm and T. Ungerer, editors, *Tagungsband zum 2. Workshop Cluster Computing*, number CSR-99-02 in Chemnitzer Informatik–Berichte, pages 15–24, March 1999.

[26] R. Buthenuth and H. Heiss. Shared Memory Programming on PC-based SCI Clusters. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 143–148, September 1998. ISBN: 1-901864-02-02.

[27] G. Cabillic, G. Muller, and I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. In *In the Proceedings of the 14th Symposium on Reliable Distributed Systems*, 1995.

[28] G. Cabillic and I. Puaut. Stardust: an environment for parallel programming on networks of heterogeneous workstations.

[29] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *In Proceedings of 13th Symposium on Operating System Principles (SOSP),* pages 152–164, October 1991.

[30] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency–Related Communication in Distributed Shared Memory Systems. *ACM Transactions on Computer Systems*, 1995.

[31] R. Chandra, A. Gupta, and J. Hennessy. Cool: An object–based language for parallel programming. *Computer*, 27(8):13–26, August 1994.

[32] R. Chandra, A. Gupta, and J. Hennessy. *COOL*, chapter 6, pages 215–255. In Wilson and Lu [231], 1996.

[33] K. Chandy and S.Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett Publishers, Boston and London, 1992.

[34] A. Charlesworth. STARFIRE: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, February 1998.

[35] Convex Computer Coorperation, Richardson, Texas, USA. *Convex Exemplar Architecture*, 2nd edition, November 1994.

[36] Compaq Computer Corp., Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0,* December 1997. Available with NDA via www.viarch.org.

[37] Data General Corporation. Data General's NUMALiiNE Technology: The Foundation for the AV 20000 Server. white paper, 1998. http://www.dg.com/about/html/numaliine_technology_av20000_f.html.

[38] Intel Corporation. *MultiProcessor Specification, Version 1.4*, May 1997. Available from Intel's developer website.

[39] D. Cortesi and J. Fier. Origin2000 and Onyx2 Performance Tuning and Optimization Guide. Technical Report 007-3430-002, Silicon Graphics, Inc., 1998. Available at http://techpubs.sgi.com/.

[40] B. Costinescu and A. Lioy. The HPPC–SEA DVSM library. Technical report, Politehnica University of Bucharest and Politecnico di Torino, November 1998. This report was written for the HPPC–SEA project.

[41] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINIUM. In *In Proceedings of the 12th Symposium on Operating Systems Principles (SOSP)*, pages 32–44, December 1989.

[42] V.J. Cunningham and T. Jones. Spectral analysis of dynamic PET studies. *Journal of Cerebral Blood Flow and Metabolism*, 13(1):15–23, January 1993.

[43] W. Dijkstra. Solution to a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, September 1965.

[44] Dolphin Interconnect Solutions, AS. *PCI–SCI Bridge Functional Specification*, November 1996.

[45] M. Dormanns. *Shared Memory Parallelization of the GROMOS96 Molecular Dynamics Code*, chapter 22, pages 383–396. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[46] B. Dreier, M. Zahn, and T. Ungerer. Parallele und verteilte Programmierung mittels Pthreads und Rthreads. In W. Rehm, editor, *Tagungsband zum 1. Workshop Cluster Computing*, number CSR-97-05 in Chemnitzer Informatik–Berichte, pages 63–85, November 1997.

[47] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *In Proceedings of the 13th International Symposium on Computer Architecture (ISCA)*, pages 434–442, 1986.

[48] M. Dubois and S. Thakkar, editors. *Scalable Shared–Memory Multiprocessors*. Kluwer Academic Publishers, Boston, MA, 1992.

[49] M. Eberl. Realisierung einer DSM–Implementation von Active Messages am Beispiel eines SCI-gekoppelten Sun Workstation Clusters. Diplomarbeit, Technische Universität München, 1996.

[50] M. Eberl, H. Hellwagner, B. Herland, and M. Schulz. SISCI — Implementing a Standard Software Infrastructure on an SCI Cluster. In W. Rehm, editor, *Tagungsband zum 1. Workshop Cluster Computing*, number CSR-97-05 in Chemnitzer Informatik–Berichte, pages 49–61, November 1997.

[51] M. Eberl, H. Hellwagner, W. Karl, M. Leberecht, and J. Weidendorfer. Fast Communication Libraries on an SCI Cluster. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 165–175. Cheshire Henbury, September 1998. ISBN: 1-901864-02-02.

[52] M. Eberl, W. Karl, C. Trinitis, and A. Blaszczyk. Parallel Computing on PC Clusters — An Alternative to Supercomputers for Industrial Applications. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain*, volume 1697 of *LNCS*, pages 493–498. Springer Verlag, Berlin, September 1999.

[53] W. Bolosky et al. NUMA Policies and Their Relation to Memory Architecture. In *In Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 1991.

[54] J. Fessler. Aspire 3.0 user's guide: A sparse iterative reconstruction library. Technical Report TR–95–293, Communications & Signal Processing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan Ann Arbor, Michigan 48109-2122, November 2000. Revised version.

[55] M. Fischer and A. Reinefeld. A PVM Implementation for Hetereogeneous SCI Clusters. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 159–164. Cheshire Henbury, September 1998. ISBN: 1-901864-02-02.

[56] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 40:35–48, 1996.

[57] H. Fuchs, M. Levoy, and S. Pizer. Interactive Visualization of 3D Medical Data. *Computer*, 22(3):45–51, August 1989.

[58] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Technical report, January 1994.

[59] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximinzing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *In Proceedings of Operating System Design and Implementation (OSDI)*, 1999.

[60] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Press, 1994.

[61] 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.

[62] K. Gharachorloo. *Memory Consistency Models for Shared–Memory Multiprocessors*. PhD thesis, Stanford University, December 1995. Also published as Stanford University Technical Report CSL-TR-95-685 and WRL research report 95/9.

[63] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared–Memory Multiprocessors. In *In Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, June 1990.

[64] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B. Johnson, H. Kohmann, R. Nordstrøm, and P. Werner. *Low–level SCI software functional specification*. Dolphin ICS and CERN, version 2.1.1 edition, March 1999. Also deliverable D.1.1.1, ESPRIT project 23174 / SISCI, Available from http://www.dolphinics.no/.

[65] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2):12–18, February 1996.

[66] V. Gonzales, E. Sanchis, and G. Torralba. Multinode Performance Evaluation: experience using the Dolphin 4–port switch. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 75–81. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[67] J. Goodman, M. Vernon, and P. Woest. Efficient Synchronization Primitives for Large–Scale Cache–Coherent Multiprocessors. In *In Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–73, 1989.

[68] L. Grabowsky, T. Radke, and W. Rehm. Cluster-MPI – An optimized MPI subset for configurable cluster systems: SCI connected SMPs as a test case. In W. Rehm, editor, *Tagungsband zum 1. Workshop Cluster Computing*, number CSR-97-05 in Chemnitzer Informatik–Berichte, pages 125–131, November 1997.

[69] R. Grass. *Siemens hpcLine — Intel (TM)-basierte hoch–skalierbare Server für technisch–wissenschaftliche Anwendungen*. Siemens AG, Information and Communication Products, High Performance Computing. Available from http://www.hpcline.de/.

[70] R. Grindley, T. Abdelrahman, S. Brown, S. Caranci, D. DeVries, B. Gamsa, A. Grbic, M. Gusat, R. Ho, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, P. McHardy, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic. The NUMAchine Multiprocessor. In *In Proceedings of the International Conferenc on Parallel Processing (ICPP)*, August 2000.

[71] A. Gueziec and R. Hummel. The wrapper algorithm: Surface extraction and simplification. In *Proceedings of the IEEE Workshop on Biomedical Image Analysis*, pages 204–213, 1994.

[72] J. Hansen, P. Koch, and E. Jul. A Stream Protocol Implementation for an SCI–based Cluster of Workstations. In *In the Proceedings of the Workshop on Cluster–based Computing (WCBC) (held in conjunction with ICS)*, pages 16–20, June 1999.

[73] M. Heines, D. Cronk, and P. Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proceedings of Supercomputing 1994*, pages 350–359, November 1994.

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

[75] H. Hellwagner and A. Reinefeld, editors. *SCI: Scalable Coherent Interface. Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *LNCS State-of-the-Art Survey*. Springer Verlag, October 1999. ISBN 3-540-66696-6.

[76] H. Hellwagner and J. Weidendorfer. *SCI Sockets Library*, chapter 11, pages 209–229. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[77] H.-J. Hermann. *Nuklearmedizin*. Urban und Schwarzenberg, 1998.

[78] R. Hockauf, J. Jeitner, W. Karl, R. Lindhof, M. Schulz, V. Gonzales, E. Sanquis, and G. Torralba. Design and Implementation Aspects for the SMiLE Hardware Monitor. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe 2000, The 3rd international conference on SCI–based technology and research*, pages 47–55. SINTEF Electronics and Cybernetics, August 2000. ISBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[79] R. Hockauf, W. Karl, M. Leberecht, M. Oberhuber, and M. Wagner. Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-Based Monitoring Approach for DSM Systems. In David Pritchard and Jeff Reeve, editors, *Euro-Par'98 Parallel Processing, 4th International Euro-Par Conference, Southampton, UK, September 1-4, 1998 Proceedings*, volume 1470 of *Lecture Notes in Computer Science*, pages 206–215, Berlin, September 1998. Springer Verlag.

[80] G. Horn. *Scalability of SCI Ringlets*, chapter 7, pages 151–166. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[81] G. Horn and W. Karl, editors. *Proceedings of SCI-Europe 2000, The 3rd international conference on SCI–based technology and research*. SINTEF Electronics and Cybernetics, August 2000. ISBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[82] R. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, 15(1):37–45, February 1995.

[83] W. Hu, W. Shi, and Z. Tang. JiaJia: An SVM System based on a New Cache Coherence Protocol. In *In the Proceedings of High Performance Computing and Networking (HPCN-Europe)*, volume 1593 of *LNCS*, pages 463–472, April 1999.

[84] L. Huse, K. Omang, and H. Bugge. ScaFun – A Fundament for Process Communication. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 13–20. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[85] L. Huse, K. Omang, H. Bugge, H. Ry, A. Haugsdal, and E. Rustad. *ScaMPI — Design and Implementation*, chapter 14, pages 249–261. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[86] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., 1993.

[87] 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.

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

[89] IBM. The IBM NUMA–Q enterprise server architecture, Solving issues of latency and scalability in multiprocessor systems. Technical report, March 2001. http://www.sequent.com/whitepapers/numa_arch.html.

[90] IEEE Computer Society. *IEEE Std 896–1987: IEEE Standard backplane bus specification for multiprocessor architectures: Futurebus*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, July 1988.

[91] IEEE Computer Society. *IEEE Std 1496–1993: IEEE Standard for a Chip and Module Interconnect Bus: SBus*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, September 1993.

[92] 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.

[93] IEEE Computer Society. *IEEE Std 896–1994: Information technology — microprocessor systems — Futurebus+ — Logical protocol specification*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, April 1994.

[94] IEEE Computer Society. *IEEE Std 1394–1995: IEEE Standard for a high performance serial bus*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, August 1996.

[95] IEEE Computer Society. *IEEE Unapproved Draft 1386 D2.2: Draft Standard for a Common Mezzanine Card Family: CMC.* The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, April 2000.

[96] L. Iftode and J. Singh. Shared Virtual Memory: Progress and Challenges. *Proceedings of IEEE*, 87(3), March 1999.

[97] L. Iftode, J. Singh, and L. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Theory of Computer Systems*, 31:451–473, 1998.

[98] Cray Inc. *SHMEM, in CRAY T3E C and C++ Optimization Guide*, chapter 3. SG–2178 3.0.1, Available at http://www.cray.com/.

[99] Intel Corporation. *Intel Architecture Software Developer's Manual for the PentiumII*, volume 1–3. published on Intel's developer website, 1998.

[100] 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*, page 148, August 1997.

[101] B. Johnsen. Write back caching of SCI memory. personal communication with Dolphin ICS, March 1998.

[102] K. Johnson, M. Kaashoek, and D. Wallach. CRL: High performance all-software distributed shared memory. In *In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[103] T. Johnson. A Performance Comparison of Fast Distributed Synchronization Algorithms. Technical Report TR98002, Department of CIS, University of Florida, Gainsville, FL 32611-2024, 1998.

[104] B. Joy, G. Steele, J. Gosling, and G. Bracha. *Threads and Locks, in The Java Language Specification, Second Edition (The Java Series)*, chapter 17. Addison-Wesley, 2nd edition, June 2000. ISBN: 0201310082, Online version available at http://java.sun.com/.

[105] R. LaRowe Jr. and C. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[106] W. Karl and G. Horn, editors. *Conference Proceedings of SCI–Europe 1999*. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[107] W. Karl, M. Leberecht, and M. Oberhuber. *SCI Monitoring Hardware and Software: Supporting Performance Evaluation and Debugging*, chapter 24, pages 417–432. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[108] W. Karl, M. Leberecht, and M. Schulz. Optimizing Data Locality for SCI–based PC–Clusters with the SMiLE Monitoring Approach. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 169–176, October 1999.

[109] W. Karl, M. Leberecht, and M. Schulz. Supporting Shared Memory and Message Passing on Clusters of PCs with a SMiLE. In A. Sivasubramaniam and M. Lauria, editors, *Proceedings of Workshop on Communication and Architectural Support for Network based Parallel Computing (CANPC) (held in conjunction with HPCA)*, volume 1602 of *Lecture Notes in Computer Science (LNCS)*, pages 196–210, Berlin, 1999. Springer Verlag.

[110] W. Karl, M. Schulz, and J. Tao. Using the SMiLE Monitoring Infrastructure to Detect and Lower the Inefficiency of Parallel Applications. In *In the Proceedings of High Performance Computing and Networking (HPCN-Europe)*, volume 1823 of *Lecture Notes in Computer Science*, pages 270–279. Springer Verlag, Berlin, May 2000.

[111] W. Karl, M. Schulz, and J. Trinitis. Multilayer Online-Monitoring for Hybrid DSM systems on top of PC clusters with a SMiLE. In *Proceedings of 11th Int. Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1786 of *LNCS*, pages 294–308. Spring Verlag, Berlin, March 2000.

[112] W. Karl, M. Schulz, M. Völk, and S. Ziegler. NEPHEW: Applying a Toolset for the Efficient Deployment of a Medical Image Application on SCI–based clusters. In A. Bode, T. Ludwig, W.Karl, and R. Wismüller, editors, *Euro-Par 2000 — Parallel Processing*, volume 1900 of *Lecture Notes of Computer Science (LNCS)*, pages 851–860. Springer Verlag, Berlin, September 2000.

[113] W. Karl, M. Schulz, and M. Völk S. Ziegler. Meeting the Computational Demands of Nuclear Medical Imaging using Commodity Clusters. In *Proceedings of the Internationa Conference on Computational Science (ICCS)*, May 2001. to appear.

[114] S. Karlsson and M. Brorsson. An Infrastructure for Portable and Efficient Software DSM. In L. Iftode and P. Keleher, editors, *Proceedings of the First International Workshop on Software Distributed Shared Memory (WSDSM)*, June 1999. Available from http://www.cs.umd.edu/˜keleher/wsdsm99/.

[115] S. Kaxiras. Kiloprocessor Extensions to SCI. In *In the proceedings of the International Parallal Processing Symposium (IPPS)*, April 1996.

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

[117] P. Keleher. *CVM: The Coherent Virtual Machine*. University of Maryland, cvm version 1.0 edition, November 1996. http://www.cd.umd.edu/projects/cvm/.

[118] P. Keleher. Symmetry and Performance in Consistency Protocols. In *In Proceedings of the International Conference in Supercomputing (ICS)*, pages 43–50, June 1999.

[119] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. An Evaluation of Software–Based Release Consistent Protocols. *Journal of Parallel and Distributed Processing*, 29, October 1995.

[120] R. Kessler and J. Schwarzmeier. CRAY-T3D: A New Dimension for Cray Research. In *Digest of Papers: CompCon Spring '93*, pages 176–182, San Francisco, CA, February 1993. IEEE, IEEE Computer Society Press, Los Alamitos, CA.

[121] Y. Khalidi, J. Bernabeu, V. Matena, K. Shirriff, and M. Thadani. Solaris MC: A Multi–Computer OS. Technical Report SMLI TR-95-48, Sun Microsystems Laboratories, 2550, Garcia Avenue, Mountain View, CA 94043, November 1995.

[122] P. Koch, E. Cecchet, and X. de Pina. Global Management of Coherent Shared Memory on an SCI Cluster. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 51–57. Cheshire Henbury, September 1998. ISBN: 1-901864-02-02.

[123] P. Koch, J. Hansen, E. Cecchet, and X. Ronsset de Pina. SciOS: An SCI-based Software Distributed Shared Memory. In *Proceedings of the First International Workshop on Software Distributed Shared Memory (WSDSM)*, June 1999. Available at http://www.cs.umd.edu/˜keleher/wsdsm99/.

[124] R. Koeninger, M. Furtney, and M. Walker. A Shared Memory MPP from Cray Research. *Digital Technical Journal*, 6(2), 1994. http://www.digital.com/info/DTJE01/DTJE01SC.TXT.

[125] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Maira, S. Dwarkadas, and M. Scott. VM–Based Shared Memory on Low–Latency, Remote–Memory–Access Networks. Technical Report Technical Report No. 643, Department of Computer Science, University of Rochester and DEC Cambridge Research Lab, November 1996.

[126] L. Kontothanassis and M. Scott. High Performance Software Coherence for Current and Future Architectures. *Journal for Parallel and Distributed Computing*, 1995.

[127] C. Kurmann and T. Stricker. *A Comparison of Three Gigabit Technologies: SCI, Myrinet and SGI/Cray T3D*, chapter 2, pages 39–68. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[128] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *In the Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 302–313, April 1994.

[129] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):241–248, 1979.

[130] L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[131] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *In Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, pages 241–251, June 1997.

[132] R. Leahy and C. Byrne. Recent developments in iterative image reconstruction for PET and SPECT. *IEEE Transactions on Nuclear Sciences*, 19:257–260, 2000.

[133] M. Leberecht. *An Efficient Runtime System Combining Dataflow, Multithreading, and Distributed Shared Memory*. PhD thesis, Technische Universität München, April 1999. Published as Volume 14 of the Research Series, LRR–TUM (Arndt Bode, Editor), Shaker–Verlag, Aachen.

[134] M. Leberecht. *The MuSE Runtime System for SCI Clusters: A Flexible Combination of On–Stack Execution and Work Stealing*, chapter 20, pages 349–364. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[135] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986. Available as TR492.

[136] K. Li. IVY: A shared virtual memory system for parallel computing. In *In Proceedings of the International Conferenc on Parallel Processing (ICPP)*, volume 2, pages 94–101, 1989.

[137] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory. *Transactions on Computer Systems*, 7(4):321–359, November 1989.

[138] M. Liaaen and H. Kohmann. *Dolphin SCI Adapter Cards*, chapter 3, pages 71–87. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[139] C. Lin and L. Snyder. ZPL: An Array Sublanguage. In *In Proceedings of the 6th Sixth International Workshop on Languages and Compilers for Parallel Computing*, pages 96–114, 1993.

[140] V. Lindenstruth, A. Bogaerts, H. Bugge, M. Davis, D. Gustavson, J. Heidbrink, H. Hellwagner, B. Herland, D. James, S. Klein, Q. Li, M. Mahalingam, J. Merkey, W. Mueller, T. Nygaard, H. Richter, D. Roehrich, T. Sheikh, P. Werner, J. Whitfield, and R. Wipfel. SCI Physical Layer API. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 137–141. Cheshire Henbury, September 1998. ISBN: 1-901864-02-02.

[141] V. Lindenstruth and D. Gustavson. *SCI Physical Layer API*, chapter 10, pages 191–208. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[142] P. Gustad K. Loechsen and O. Toerudbakken. High performance switching and congestion avoidance in SCI. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe 2000, The 3rd international conference on SCI–based technology and research*, pages 119–126. SINTEF Electronics and Cybernetics, August 2000. ISBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[143] H. Lu, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proceedings of Supercomputing '95*, December 1995.

[144] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.

[145] M. Schulz. Efficient deployment of shared memory models on clusters of PCs using the SMiLEing HAMSTER approach. In A. Goscinski, H. Ip, W. Jia, and W. Zhou, editors, *Proceedings of the 4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 2–14. World Scientific Publishing, December 2000.

[146] M. Schulz and H. Hellwagner. Extending NT Virtual Memory by SCI–based Hardware DSM. In *Proceedings of the USENIX Windows NT Symposium*, page 169, August 1998.

[147] M. Manzke and B. Coghlan. Non-Intrusive Deep Tracing of SCI Interconnect Traffic. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 53–58. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[148] R. Marejka. Multi-threaded Programming. Technical report, Solaris 2 Migration Support Centre, Sun Microsystems Inc., March 1996. Revision B.2, Available from http://www.sun.com/.

[149] M. May. Vergleich von PVM und CORBA bei der verteilten Berechnung medizinischer Bilddaten. Diplomarbeit, Technische Universität München, 2000.

[150] J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared–Memory Multiprocessors. *ACM Transactions on Computer Systems*, February 1991.

[151] D. Mentre and T. Priol. NOA: A Shared Virtual Memory over a SCI cluster. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 43–50. Cheshire Henbury, September 1998. ISBN: 1-901864-02-02.

[152] Message Passing Interface Forum (MPIF). MPI: A Message-Passing Interface Standard. Technical Report, University of Tennessee, Knoxville, June 1995. http://www.mpi-forum.org/.

[153] M. Michael and M. Scott. Scalability of Atomic Primitives on Distributed Shared Memory Multiprocessors. In *In Proceedings of the International Conference on High Performance Computer Architecture (HPCA)*, 1995.

[154] Microsoft Cooperation. *Microsoft Platform Software Development Kit*, chapter About Processes and Threads. Microsoft, 1997. available with Microsoft's SDK.

[155] Microsoft Cooperation. *Microsoft Platform Software Development Kit*, chapter Synchronization. Microsoft, 1997. available with Microsoft's SDK.

[156] Microsoft Cooperation. *Microsoft Platform Software Development Kit*. Microsoft, 1997. available with Microsoft's SDK.

[157] Sun Microsystems. The Sun Enterprise Cluster Architecture, Technical White Paper. Technical report, 1997. Available at http://www.sun.com/software/cluster/wp-arch/wp.pdf.

[158] SUN Microsystems. *JavaSpaces (TM) Service Specification, Version 1.1*, October 2000. Available from http://java.sun.com/.

[159] Inc. MIL 3. *OPNET Modeler manuals*. 3400 International Drive NW, Washington DC, 20008, USA, 1989–1997.

[160] L. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *In Proceedings of the 4thInternational Symposium on High Performance Computer Architecture (HPCA)*, February 1998.

[161] D. Moosberger. Memory Consistency Models. Technical Report TR 93/11, Department of Computer Science, University of Arizona, Tuscon, AZ 85721, 1993.

[162] C. Morin and I. Puaut. A Survey of Recoverable Distributed Shared Memory Systems. Technical Report Publication Interne 975, IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cédex, France, December 1995.

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

[164] 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 (held in conjunction with IPPS)*, pages 31–40, April 1997.

[165] F. Müller. Decentralized Synchronization for Multi–threaded DSMs. In L. Iftode and P. Keleher, editors, *Proceedings of the Second International Workshop on Software Distributed Shared Memory (WSDSM)*, May 2000. Available at http://www.cs.rutgers.edu/˜wsdsm00/.

[166] F. Munz. Parallele Rekonstruktion von Volumendaten. Diplomarbeit, Technische Universität München, 1995.

[167] J. Nieplocha and R. Harrison. Shared Memory NUMA Programming on I–Way. In *In Proceedings of the 5th International Symposium on High Performance Distributed Computing (HPDC)*, 1995.

[168] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A Non–Uniform–Memory–Access Programming Model For High–Performance Computers. *The Journal of Supercomputing*, 10:169–189XS, 1996.

[169] B. Nitzberg and V. LO. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, pages 52–59, August 1991.

[170] K. Omang. Performance Results from SALMON, A Multiprocessor Environment based on Workstations connected by SCI. Technical Report Research Report 208, University of Oslo, Department of Informatics, November 1995. ISBN: 82–7368–120–3.

[171] K. Omang. Synchronization Support in I/O Adapter Based SCI Clusters. In *Proceedings of Workshop on Communication and Architectural Support for Network based Parallel Computing (CANPC)*, volume 1199 of *Lecture Notes in Computer Science (LNCS)*. Springer, Berlin, February 1997.

[172] OpenMP Architecture Review Board. *OpenMP C and C++ Application, Program Interface*, Version 1.0, Document Number 004–2229–01 edition, October 1998. Available from http://www.openmp.org/.

[173] J. Ousterhout, A. Cherenson, F. Douglis, M . Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[174] 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.

[175] D. Patterson and J. Hennessy. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2 edition, 1996.

[176] K. Petersen and K. Li. Multiprocessor Cache Coherence Based on Virtual Memory Support. *Journal of Parallel and Distributed Computing, Special Issue Scalable Shared Memory Multiprocessors*, 29, October 1995.

[177] S. Petri, G. Lustig, C. Grewe, R. Hagenau, W. Obelöer, and M. Boosten. Performance Comparison of Different High–Speed Networks with a Uniform Efficient Programming Interface. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 83–90. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[178] T. Priol, C. René, and G. Alléon. *Programming SCI Clusters Using Parallel CORBA*, chapter 19, pages 333–348. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[179] M. Rangarajan, S. Divakaran, T. Nguyen, and L. Iftode. Multi–threaded Home–based LRC Distributed Shared Memory. In *In the Proceedings of the 8th Workshop on Scalable Shared Memory Multiprocessors (held in conjunction with ISCA)*, May 1999.

[180] E. Rehling. Multithreading for SCI Clusters: Yasmin and the Sthreads Library. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 27–33. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[181] A. Reinefeld and H. Hellwagner, editors. *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*. Cheshire Henbury, September 1998. ISBN: 1-901864-02-02.

[182] G. Ronneberg and O. Lynse. An OPNET–based Simulation Model of SCI–Nodes. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 131–137. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[183] G. Ronneberg, O. Lynse, and G. Horn. Evaluation and Suggested Improvements for the SCI Flow Control. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 101–112. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[184] A. Rubini. *Linux Device Drivers*. O'Reilly & Associates, Inc., 1 edition, February 1998. ISBN: 1-56592-292-1.

[185] S. Ryan. The design and implementation of a portable driver for shared memory cluster adapters. Technical Report Research report 255, University of Oslo, Department of Informatics, December 1997. ISBN 82–7368–177-7.

[186] S. J. Ryan and H. Bryhni. Eliminating the Protocol Stack for Socket Based Communication in Shared Memory Interconnects. In *Proc. Int'l. Workshop on Personal Computer based Networks of Workstations (held in conjunction IPPS'98)*, Orlando, Florida, USA, April 1998.

[187] H. Sandhu. Integrating Applications with Cache and Memory Management on a Shared Memory Multiprocessor. In *Proceedings of CASCON*, 1992.

[188] H. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, May 1993.

[189] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A Low Overhead, Software–Only Approach for Supporting Fine–Grain Shared Memory. Technical Report WRL Research Report 96/2, Digital Western Research Laboratory, 250 University Avenue, Palo Alto, California 94301, USA, November 1996.

[190] Scali Computer AS. *Scali Software Platform — SSP*. Olaf Helsets vei 6, Postboks 70, Bogerud, N-0621 Oslo, Norway, June 2000. Available at http://www.scali.com/.

[191] D. Schmidt. An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit. Technical Report WUCS-95-31, Department of Computer Science, Washington University, St. Louis, MO, 1995.

[192] M. Schulz. Report on Pthreads design and test suite on SCI/PC/NT cluster. Technical report, Technische Universität München, May 1998. Deliverable D 4.1.1, ESPRIT Project 23174 — SISCI.

[193] M. Schulz. SCI-VM: A flexible base for transparent shared memory programming models on clusters of PCs Martin Schulz. In J. Rolim, F. Müller, and et. al., editors, *Parallel and Distributed Computing / Proceedings of HIPS '99*, volume 1586 of *Lecture Notes in Computer Science*, pages 19–33, Berlin, 1999. Springer Verlag.

[194] M. Schulz. SISCI Pthreads implementation report. Technical report, Technische Universität München, November 1999. Deliverable D 4.1.4, ESPRIT Project 23174 — SISCI.

[195] M. Schulz. *True shared memory programming on SCI-based clusters*, chapter 17, pages 291–311. Volume 1734 of Hellwagner and Reinefeld [75], October 1999. ISBN 3-540-66696-6.

[196] M. Schulz. Efficient Coherency and Synchronization Management in SCI based DSM systems. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe 2000, The 3rd international conference on SCI–based technology and research*, pages 31–36. SINTEF Electronics and Cybernetics, August 2000. ISBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[197] M. Schulz. Multithreaded Programming of PC clusters. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT) 2000, Philadelphia, PA, USA*, pages 271–278. IEEE, October 2000.

[198] M. Schulz and W. Karl. Hybrid-DSM: An Efficient Alternative to Pure Software DSM Systems on NUMA Architectures. In L. Iftode and P. Keleher, editors, *Proceedings of the Second International Workshop on Software Distributed Shared Memory (WSDSM)*, May 2000. Available at http://www.cs.rutgers.edu/˜wsdsm00/.

[199] M. Schulz, M. Völk, W. Karl, F. Munz, and S. Ziegler. Running a spectral analysis code on top of SCI shared memory using the TreadMarks API. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 35–43. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[200] M. Schulz, M. Völk, W. Karl, and S. Ziegler. Effiziente iterative PET-Bild Rekonstruktion auf einem Cluster von PCs. *Journal of Radiationoncology. Biology. Physics – Abstraktband des gemeinsamen Jahreskongresses der DEGRO, ÖGRO, DGMP*, 176(1), October 2000.

[201] M. Scott. Is s–dsm dead? Keynote talk given at 2nd workshop for Software Distributed Shared Memory (WSDSM), Santa Fe, NM, USA, May 2000. Slides available at http://www.cs.rochester.edu/u/scott/interweave/WSDSM.pdf.

[202] S. Scott. *A Cache Coherence Mechanism for Scalable Shared–Memory Multiprocessors*, chapter 18. In Suzuki [214], 1992.

[203] S. Scott, J. Goodman, and K. Vernon. Performance of the SCI Ring. In *In the proceedings of the 19th International Symposium of Computer Architecture (ISCA)*, May 1992.

[204] D. Siguenza, F. Mora, and A. Sebastiá. SCI Network Simulations with VHDL. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 125–129. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[205] P. Sindhu, J. Frailong, and M. Ceklov. *Formal Specification of Memory Modules*. In [48].

[206] J. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared–memory. *Computer Architecture News*, 22(1):5–44, 1992.

[207] SISCI Consortium. Standard Software Infrastructures for SCI-based Parallel Systems (SISCI). http://www.parallab.uib.no/projects/sisci/, August 1997. Support by the EU under EP 23174.

[208] B. Skaali, B. Nossum, I. Birkeli, and D. Wormald. SCIview — SCI test, Verification, and Monitoring Instrumentation. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 47–52. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[209] E. Speight and J. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Workshop*, pages 95–106, August 1997.

[210] A. Stamatakis. Evaluation of Interoperable Tool Deployment for the Late Development Phases of Distributed Object–Oriented Programs. Diplomarbeit, Technische Universität München, February 2001.

[211] T. Stephan. Erweiterung eines parallelen Rekonstruktionsprogramms von PET–Volumendaten um Komponenten zur Stapelverarbeitung und Lastverwaltung. Diplomarbeit, Technische Universität München, May 1997.

[212] R. Stets, D. Chen, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, G. Magklis, S. Parthasarathy, U. Rencuzogullari, and M. Scott. The Implementation of Cahsmere. Technical report, Department of Computer Science, University of Rochester, Rochester, NY 14627–0226.

[213] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *In Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[214] N. Suzuki, editor. *Shared Memory Multiprocessing*. MIT Press, 1992.

[215] M. Swanson, L. Stoller, and J. Carter. Making Distributed Shared Memory Simple, Yet Efficient. In *Proceedings of the Workshop on High–Level Programming Models and Supportive Environments (HIPS) (held in conjunction with IPPS)*. IEEE, April 1998.

[216] A. Tanenbaum. *Moderne Betriebssysteme*. Carl Hanser Verlag & Prentice Hall, 1994.

[217] J. Tao, W. Karl, and M. Schulz. Understanding the Behavior of Shared Memory Applications Using the SMiLE Monitoring Framework. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe 2000, The 3rd international conference on SCI–based technology and research*, pages 57–62. SINTEF Electronics and Cybernetics, August 2000. ISBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[218] H. Taskin, R. Buthenuth, and H. Heiss. SCI for TCP/IP with Linux. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 155–157. Cheshire Henbury, September 1998. ISBN: 1-901864-02-02.

[219] 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.

[220] U. Tiede, K. Hoehne, M. Bomans, A. Pommert, M. Riemer, and G. Wiebecke. Investigation of medical 3d–rendering algorithm. *IEEE Computer Graphics & Applications*, pages 41–53, March 1990.

[221] M. Trams and W. Rehm. A new generic and reconfigurable PCI–SCI bridge. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 113–120. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[222] C. Trinitis, M. Eberl, and W. Karl. Numerical Calculation of Electromagnetic Problems on an SCI Based PC-Cluster. In *2000 International Conference on Parallel Computing in Electrical Engineering (PAREL EC 2000)*, pages 166–170, Washington DC, USA, 2000. IEEE Computer Society.

[223] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC–NUMA Compute Servers. In *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[224] M. Völk. Parallelisierung eines Spektralanalysealgorithmus mittels TreadMarks. Diplomarbeit, Technische Universität München, February 1999.

[225] 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 (ISCA)*, May 1992.

[226] C. Wagner and F. Müller. Token–Based Read/Write–Locks for Distributed Mutual Exclusion. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro–Par 2000, Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science (LNCS)*, pages 1185–1195. Springer Verlag, Berlin, September 2000.

[227] A. Walsch, V. Lindenstruth, and M. Schulz. A 1 MHz Transaction Processor Farm For High Energy Physics. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe 2000, The 3rd international conference on SCI–based technology and research*, pages 89–99. SINTEF Electronics and Cybernetics, August 2000. ISBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[228] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques*. Addison Wesley, 1992.

[229] J. Weidendorfer. Entwurf und Implementierung einer Socket-Bibliothek für ein SCI-Netzwerk. Diplomarbeit, Technische Universität München, 1997. Available at http://wwwbode.in.tum.de/˜weidendo/.

[230] G. Wilson. Linda–Like Systems and Their Implementation. Technical Report 91–13, Edinburgh Parallel Computing Centre, June 1991.

[231] G. Wilson and P. Lu, editors. *Parallel Programming using C++*. Scientific and Engineering Computation Series. Massachusetts Institute of Technology, 1996.

[232] P. Woest and J. Goodman. *An Analysis of Shared–Memory Synchronization Mechanisms*, chapter 17. In Suzuki [214], 1992.

[233] 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 (ISCA)*, pages 24–36, June 1995.

[234] J. Worringen and T. Bemmerl. MPICH for SCI–Connected Clusters. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*, pages 3–11. SINTEF Electronics and Cybernetics, September 1999. ISBN: 82-14-00014-9, Also available at http://wwwbode.in.tum.de/events/.

[235] WWW:. CRAY T3E Series
. http://www.cray.com/products/systems/crayt3e/, November 1998.

[236] WWW:. HPCC - HPF (High Performance Fortran)
. http://hpcc.soton.ac.uk/RandD/hpf/hpf.html, December 1999.

[237] WWW:. Welcome to Giganet
. http://www.giganet.com/, May 1999.

[238] WWW:. ALINKA High Performance Linux Clustering
. http://www.alinka.com/araisin.php3, December 2000.

[239] WWW:. Dolphin Interconnect — Home Page
. http://www.dolphinics.no/, December 2000.

[240] WWW:. Dolphin Interconnect — Research Project Partners
. http://208.179.47.35/resproject.html, December 2000.

[241] WWW:. Interconnect Systems Solutions, Home
. http://www.iss-us.com/, July 2000.

[242] WWW:. NEPHEW Homepage
. http://www.arttic.com/projects/NEPHEW/, January 2000.

[243] WWW:. Peakware — Matra Systems & Information
. http://www.matra-msi.com/ang/savoir_infor_peakware_d.htm, January 2000.

[244] WWW:. RM 600
. http://www.fujitsu-siemens.com/servers/rm/rm_us/rm600e.htm, July 2000.

[245] WWW:. SMiLE: SCI-WG
. http://wwwbode.in.tum.de/Par/arch/smile/sciwg/, January 2000.

[246] WWW:. DIAMANT: (EC IST 1999 12078) — Digital Film Manipulation System
. http://diamant.joanneum.ac.at/, February 2001.

[247] WWW:. Innovative scientific, technological, engineering soultions (Home page of AEA Technology
. http://www.aeat.com/, February 2001.

[248] WWW:. Precision Software GmbH — Threads.h++
. http://www.precisions.de/de/tp/rogue/cpp/threads/, January 2001.

[249] WWW:. Scali Home Page: Scalable Linux Systems — Affordable Supercomputing — Cluster Technology
. http://www.scali.com/, February 2001.

[250] WWW:. SCILAB Technology AS
. http://www.scilabtech.com/, January 2001.

[251] WWW:. Sun Servers (Hardware, SUN Enterprise (TM) servers) . http://www.sun.com/servers, February 2001.

[252] WWW:. Welcome to CFX — The Fluid Approach to CFD Business Solutions . http://www.software.aeat.com/cfx/default.asp, February 2001.

[253] WWW:. Welcome to PCI SIG . http://www.pcisig.com/, February 2001.

[254] M. Zekauskas, W. Sawdon, and B. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[255] S. Zhou, M. Stumm, K. Li, and D Wortmann. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.

[256] Y. Zhou, L. Iftode, J. Singh, K. Li, B. Toonen, I. Schoinas, M. Hill, and D. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *In Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, 1997.

Index