Visualizing the Memory Access Behavior of Shared Memory Applications on NUMA Architectures

Jie Tao*, Wolfgang Karl, and Martin Schulz

LRR-TUM, Institut für Informatik,
Technische Universität München, 80290 München, Germany
E-mail: {tao,karlw,schulzm}@in.tum.de

Abstract. Data locality is one of the most important issues affecting the performance of shared memory applications on NUMA architectures. A possibility to improve data locality is the specification of a correct data layout within the source code. This kind of optimization, however, requires in depth knowledge about the run-time memory access behavior of programs. In order to acquire this knowledge without causing a probe overhead, as it would be caused by software instrumentation approaches, it is necessary to adopt a hardware performance monitor that can provide detailed information about memory transactions. As the monitored information is usually very low-level and not user-readable, a visualization tool is necessary as well. This paper presents such a visualization tool displaying the monitored data in a user understandable way thereby showing the memory access behavior of shared memory applications. In addition, it projects the physical addresses in the memory transactions back to the data structures within the source code. This increases a programmer's ability to effectively understand, develop, and optimize programs.

1 Introduction

Clusters with NUMA characteristics are becoming increasingly important and are regarded as adequate architectures for High Performance Computing. Such clusters based on PCs have been built for our SMiLE project (Shared Memory in a Lan–like Environment) [8]. In order to achieve the necessary NUMA global memory support, the Scalable Coherent Interface (SCI) [1,5], an IEEE standardized System Area Network, is taken as the interconnection technology. With a latency of less than 2 μs and a bandwidth of more than 80 MB/s for process to process communication, SCI offers state-of-the-art performance for clusters. Shared memory programming is enabled on the SMiLE clusters using a hybrid DSM system, the SCI Virtual Memory (SCI-VM) [9,14]. In this system, all communication is directly handled by the SCI hardware through appropriate remote memory mappings, while the complete memory management and the required global process abstraction across the individual operating system instances is handled by a software component. This creates a global virtual memory allowing a direct execution of pure shared memory applications on SCI-based clusters of PCs.

* Jie Tao is a staff member of Jilin University in China and is currently pursuing her Ph.D. at Technische Universität München in Germany.

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2074, pp. 861–870, 2001.

Many shared memory applications, however, initially do not run efficiently on top of such clusters. The performance of NUMA systems depends on an efficient exploitation of memory access locality since remote memory accesses are still an order of magnitude slower than local memory accesses, despite the good latency offered by current interconnection technologies. Unfortunately, many shared memory applications without optimization exhibit a poor data locality when running on NUMA architectures and therefore do not achieve a good performance. It is necessary to develop performance tools which can be used to enable and exploit data locality. One example of such tools is a visualizer which allows the user to understand an application's memory access behavior, to detect communication bottlenecks, and to further optimize the application with the goal of better data locality.

Such tools, however, require detailed information about the low-level memory transactions in the running program. The only way to facilitate this without causing a high probe overhead is to perform the data acquisition using a hardware monitor capable of observing all memory transactions performed across the interconnection fabric. Such a monitoring device has been developed within the SMiLE project.

The hardware monitor traces memory references performed by the running program and gives the user a complete overview of the program's memory access behavior in the form of access histograms. Its physical implementation is based on SCI-based clusters; the general principles, however, could be applied to any NUMA architecture in the same way, therefore providing a general approach for the optimization of shared memory applications on top of such systems.

The information from the hardware monitor, however, is purely based on physical addresses, very detailed and low-level, and is therefore not directly suitable for the evaluation of applications. Additional tools are necessary in order to transform the user-unreadable monitored data in a more understandable and easy-to-use form. One example for such a tool, a visualizer, is presented in this paper. It offers both a global overview of the data transfer among processors and a description of given data structure or parts of arrays. This information can be used to easily identify access hot spots and to understand the complete memory access behavior of shared memory applications. Based on this information, the application can then be optimized using an application specific data layout potentially resulting in significant performance improvements.

The remainder of this paper is organized as follows. Section 2 introduces a tool environment for efficient shared memory programming. Section 3 describes the visualizer. In Section 4 a few related work are compared. The paper is then rounded up with a short summary and some future directions in Section 5.

2 Framework for On-line Monitoring

As many shared applications on NUMA systems do initially not achieve a good parallel performance primarily due to the poor data locality, a tool environment has been designed to enable extensive tuning. As shown in Figure 1, this environment includes three components: Data acquisition, an on-line monitoring interface called OMIS/OCM, and a set of tools.

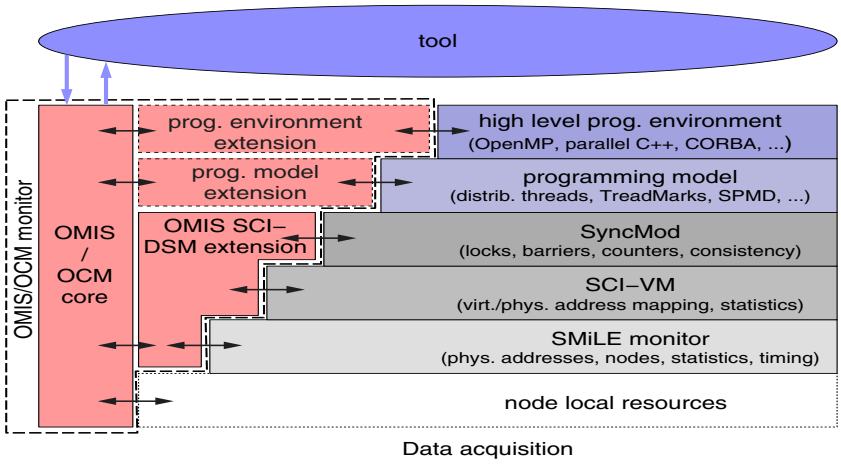

Fig. 1. Infrastructure of the on-line monitoring for efficient shared memory programming

2.1 Data Acquisition

The data acquisition component is used to collect the data required by tools. It comprises a flexible multilayer infrastructure, including both system facilities and additional software packages, which prepares and delivers data to tools for performance analysis and program tuning. The system facilities are node local resources on the PCs and the hardware monitor. The node local resources include performance counters in the processors as well as the complete process and thread management services of the local operating systems. The hardware monitor [6,7] provides the detailed information about the low-level transactions performed on the SCI network. It snoops a synchronous local bus on the network interface, over which all SCI packets from and to the nodes are transfered. The information carried by the packets is extracted and triggered by an event filter. Events of interest are counted. The information gathered by the hardware monitor is spilled to the corresponding buffer in the main memory via the PCI local bus.

On top of the hardware monitor, the SCI-VM handles further parts of the monitored data, e.g. the physical addresses, which are not suitable for a direct use of tools. As the current implementation of the hardware monitor uses physical addresses for identifying the access destination of a memory transaction event, the SCI-VM provides a virtual-to-physical address mapping list enabling tools to define events using physical addresses.

As the hardware monitor lacks the ability to monitor synchronization primitives, which generally have a critical performance impact on shared memory applications, the SyncMod module [15] is responsible for any synchronization within shared memory applications including locks and barriers. Information delivered by the SyncMod module ranges from simple lock and unlock counters to more sophisticated information like mean and peak lock times and peak barrier wait times. This information will allow the detection of bottlenecks in applications and forms the base for an efficient application optimization.

On top of the SCI-VM and the SyncMod, shared memory programming models can be implemented depending on the user's demands and preferences. Such programming models can be standard APIs ranging from distributed thread models to DSM APIs like the one of TreadMarks [2], but also form the basis for the implementation of more complex programming environments like CORBA [16] and OpenMP [12].

2.2 OMIS/OCM Interface

In order to enable tools to access the information gathered by the data acquisition component, a system and tool independent framework, the OMIS/OCM, is adopted. OMIS (On-line Monitoring Interface Specification) [3] is the specification of an interface between a programmable on-line monitoring system for distributed computing and the tools that sit on top of it. OCM [19] is an OMIS Compliant Monitoring system implemented for a specific target platform. The OMIS/OCM system has two interfaces: one for the interaction with different tools and the other for interaction with the program and all run-time system layers. It comprises core of services that are fully independent of the underlying platform and the programming environment. It can be augmented by optional extensions to cover platform or programming environment specific services for a special tool. As we apply OMIS/OCM in our research work to implement a non-conflicting access from tools to the data acquisition component, the OMIS/OCM core has to be extended with various extensions allowing the definition and implementation of new services related to all of the components within the data acquisition multilayers. Using these services, information can be transfered from nodes, processes, and threads to the tools.

2.3 Tools

The information offered by the data acquisition component is still low-level and difficult to understand; tools are therefore required to manipulate the execution of programs and ease the analyzation and optimization of programs. Currently two tools are envisioned. A visualizer is intended to exhibit the monitored data in an easy-to-use way allowing programmers to understand the dynamic behavior of applications. As the SCI-VM offers all necessary capabilities to specify the data layout in the source code, programmers are enabled to optimize programs with a better data layout and further to improve data locality. Another tool, an adaptive run-time system, is intended to transparently modify the data layout at run-time. It detects communication bottlenecks and determines a proper location for inappropriately distributed data using the monitored information and also the information available from the SCI-VM, the programming models, and even the users. Incorrectly located data will be migrated during the execution of a program, resulting in a better data locality.

3 Visualizing the Run-Time Memory Access Behavior

Data locality is the most important performance issue of shared memory programming on NUMA architectures and a visualization tool provides a direct way to understand

the memory access behavior. Therefore, a first endeavor in the implementation of an appropriate tool environment focuses on the visualizer. Based on the monitored data, this toolset generates diagrams, tables, and other understandable representations to illustrate the low-level memory transactions. It aims at providing programmers with a valuable and easy-to-use tool for analyzing the run-time data layout of applications.

The visualizer is implemented using Java in combination with the Swing library to take full advantages of the advanced graphic capabilities of Swing minimizing the implementation efforts. It deals with the monitored data and generates diagrams, tables, curves, etc. The monitored data, however, is not directly applied to the visualizer since it is fine-grained and scattered. Some functions are provided to acquire corse-grained information from this original data. These functions are compiled to a shared library and connected to the visualizer using the Java Native Interface.

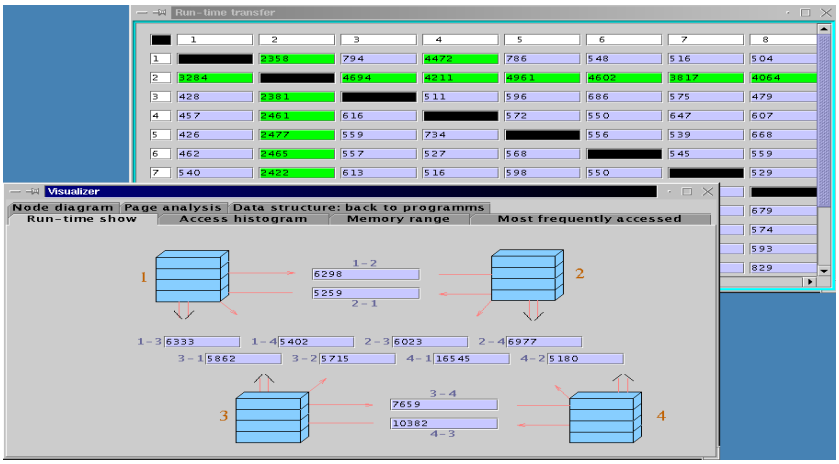

Fig. 2. The Run-time show window

The visualizer provides several different views of the acquired monitored data in order to show the various aspects of an application's memory access behavior. It shows the low-level data transfer among processors, the detailed access numbers to pages and memory regions, communication hot spots, the access behavior within a single page, and the projection back to the data structure within the source codes. Figure 2, 3, and 4 illustrate the common windows showing the monitored data acquired during the simulated execution of the RADIX program from the SPLASH2-Benchmark suite, with a working set size of 256KB running on a 4-nodes SCI-cluster. As the implementation of the hardware monitor is still on going, a simulation system [17] has been developed that simulates the execution of shared memory programs on hardware monitor equipped SCI clusters. The monitored data generated by the simulation system is exactly the one that will be provided by the future hardware monitor. We have applied the simulation system to the visualizer to enable first experiments.

3.1 Run-Time Show and Detailed Access Numbers

The windows shown in Figure 2 and 3 provide some detailed information about the individual memory transactions. The "Run-time show" window, as illustrated in Figure 2, is used to reflect the actual transfer between nodes. Two different diagrams are available: a graph version for dealing with clusters with up to 4 PCs and a table version for larger configurations. The four components in the graph represent four memories located on the different nodes and the arrows between them represent the flow of data among nodes. Rectangles between components represent counters, which are used to store the number of data transfers between two nodes and are dynamically modified corresponding to the run-time data transfer. One counter is incremented every time a data transaction is performed between the two memories. For larger systems, a counter matrix is used to exhibit the node-to-node data transfer. Communication hot spots are identified using different foreground colors. A counter is highlighted when its value exceeds an user defined threshold and marked with another bright color if its value exceeds a second threshold.

The exact numbers of memory accesses to pages and regions are illustrated in the "Access histogram" table and the "Memory region" window. The "Access histogram" reflects the memory accesses to the whole global virtual memory, while the "Memory region" shows only a region of it. Each row of the "Access histogram" table displays the virtual page number , the node number, the access source, the number of accesses to that page, and the ratio of accesses from this source to the total accesses from all nodes to that page. The information shown in the "Memory region" window includes primarily the number of accesses to a user specified memory region on a node. Excessive remote accesses are highlighted in the "Access histogram" table and the exact sorting of these accesses is listed in the "Most frequently accessed" window.

3.2 Node Diagrams

Further details about memory accesses to pages can be found in the "node diagram" window shown in Figure 3. This graph is used to exhibit the various aspects of memory transactions with page granularity. It uses colored columns to show the relative frequency of accesses performed by all nodes to a single page. One color stands for one node. Inappropriate data allocation can be therefore easily detected via the different height of the columns. Page 515 (virtual number), for example, is located on node 1 but accessed only by node 2. It is therefore incorrectly allocated and should be placed on node 2. This allocation can be specified by users within the source code using annotations provided by the SCI Virtual Memory. In order to support this work, a small subwindow is combined to the "Access histogram". The corresponding data structure of a selected page will be shown in this small upper window when a mouse button is pressed within the graphic area. The node diagrams therefore direct users to place data on the node that mostly requires it. Each diagram shows only a certain number of pages depending on the number of processors. A page list allows users to change the beginning of a diagram, enabling thereby an overall view of the data layout.

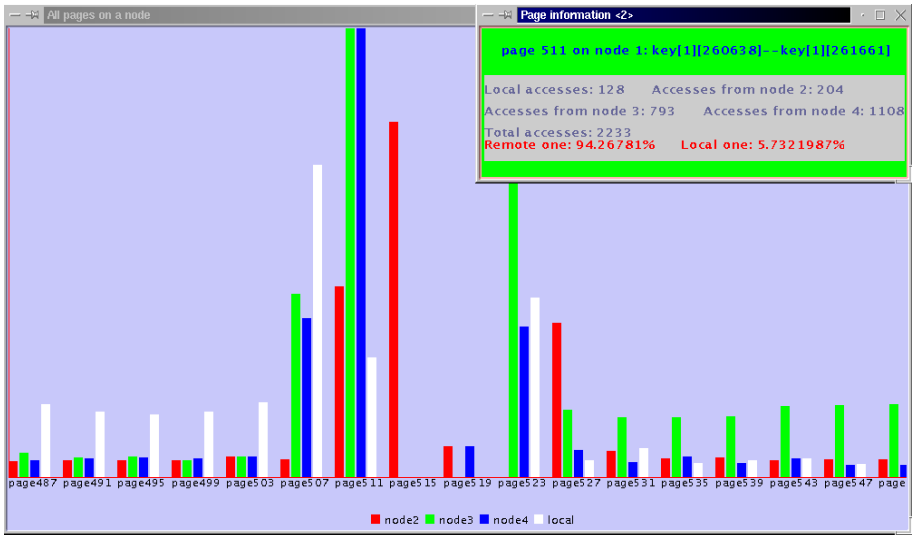

Fig. 3. An access diagram of node 1

3.3 Page Analysis

The description above has shown that node diagrams provide information for a better allocation of individual pages which are frequently accessed by only one node. The allocation of pages accessed by more processors, however, is more difficult and requires a further analysis of the accesses within them. The "Section" diagram, "Read/write" curve, and "Phase" diagram, as shown together in Figure 4, provide the support for users to make a reasonable decision about this issue. The "Section" diagram exhibits the access behavior of different sections within a single page, the "Read/write" curve offers the comparison between the frequency of reads and writes to different sections, and the "Phase" diagram provides the per-phase statistics of memory accesses to a page. The section size can be specified by the user and phases are marked by global barriers within the applications. Corresponding data structures of selected sections can be displayed in the same way as done in the "Node digram".

3.4 Projecting Back to the Source Code

The visualized memory access behavior with various granularities is not sufficient for the analyzation and optimization of data locality, as programmers do not use the virtual addresses explicitely in the source codes. Page numbers and memory addresses therefore have to be projected onto the corresponding data structures within the source codes of running applications. For this purpose, the visualizer offers a "Data structure" window to reflect this mapping. Instead of displaying only one page or section of interest as it is the case in the "Node diagram" and the "Page analysis" windows, this window shows all the shared variables occurring in a source code. In combination with the "Access histogram", the "Data structure" window provides users with a global overview of accesses to the

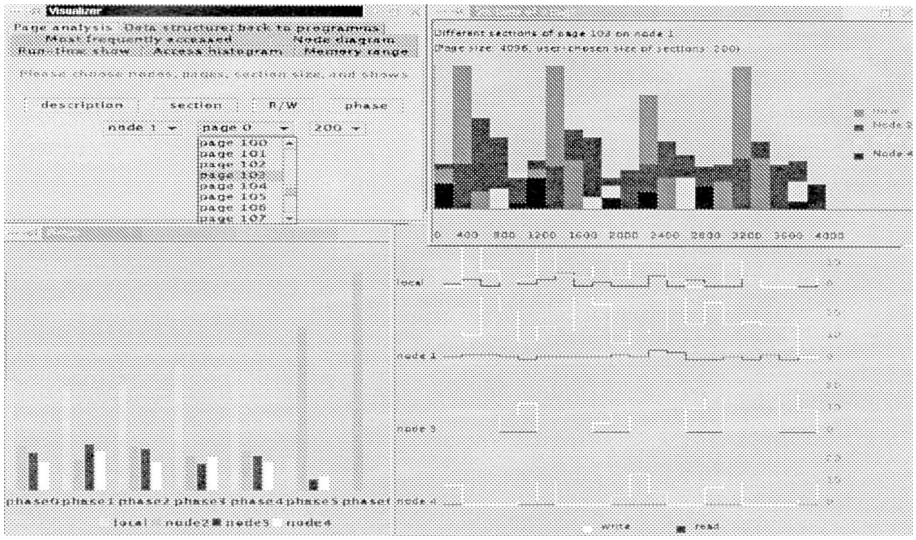

Fig. 4. The detailed access character of a page

complete working set of the application. The relationship between the virtual addresses and the variables, which forms the base of the projection, is currently accomplished using a set of macros available in the programming models. The SCI-VM delivers the information required for the transformation of physical to virtual addresses. As the programming models provide corresponding macros, this work will not cause much inconvenience for the users. On the other hand, many troubles and inaccuracies related to modifying compilers or extracting symbol information from binaries are avoided.

4 Related Work

Few research projects have focused on providing hardware-supported monitoring for network traffic in clusters. The performance monitor Surfboard for the Princeton SHRIMP multicomputer [10] is one example. It is designed to measure all communications and respond to user commands. The monitored data is written to a DRAM as needed. Unlike our research work the SHRIMP project uses Surfboard primarily to measure the packet latency, packet size, and receive time. Another trace instrument [11] has been designed at Trinity College Dublin for SCI-based systems. Full traces are transfered into a relational database. Users are allowed to access this database only after the monitoring.This instrument is therefore intended for offline analysis, debugging and validation.

Visualization tools, however, have widely been investigated. VISTOP [18] is a state based visualizer for PVM applications. PROVE [4] is a performance visualization tool designed for an integrated parallel program development environment. Part of the work introduced in [13] is the visualization of the measured performance data obtained through

the instrumentation of source code with calls to software probes. Unlike these performance visualization systems focusing on presenting process states and process communication, the visualization tool described in this paper is more oriented to visualizing the interconnect traffic. Based on a powerful performance monitor it provides programmers with a valuable support in optimizing the applications towards a better data locality at run-time.

5 Conclusion and Future Work

Data locality is one of the most important performance issues on NUMA architectures. It has been addressed intensively in recent years. Optimizing programs with the goal of a better data layout can significantly improve data locality. This requires the user, however, to understand a program's memory access behavior at run-time. This is a complex and difficult task since any communication on such architectures is performed implicitly and handled completely by the NUMA hardware. A scheme to ease this work is to monitor the interconnection traffic and then to visualize the memory access behavior, forming the base for a further locality optimization by programmers.

Such a hardware performance monitor and a visualizer have been developed for NUMA-characterized SCI-based clusters. This paper presents primarily the visualizer which shows the fine-grained monitored information about the memory accesses in a higher-level and more readable way. In addition, it maps the monitored data back to the source code enabling an exact analysis of a program's data layout. Using this visualized information, programmers are able to detect the communication bottlenecks and further optimize programs with a better data layout potentially resulting in significant performance improvements.

As some windows of the current visualizer are only considered for small clusters, the next step in this research work is to design more flexible data representations allowing a comprehensive show of monitored data even for large clusters. In addition, different kinds of applications will be analyzed and optimized using the visualized information. It can be expected that these applications will benefit from this optimization significantly.

References

1. IEEE Standard for the Scalable Coherent Interface(SCI). IEEE Std 1596-1992,1993, IEEE 345 East 47th Street, New York, NY 10017-2394, USA.
2. C. Amza, A. Cox, S. Dwarkadas, and etc. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1995.
3. M. Bubak, W. Funika, R. Gembarowski, and R. Wismüller. OMIS-compliant monitoring system for MPI applications. In *Proc. 3rd International Conference on Parallel Processing and Applied Mathematics - PPAM'99*, pages 378–386, Kazimierz Dolny, Poland, September 1999.
4. P. Cacsuk. Performance visualization in the GRADE parallel programming environment. In *Proceedings of the 4th international conference on High Performance Computing in Asia-Pacific Region*, pages 446–450, Peking, China, 14-17, May 2000.

5. Hermann Hellwagner and Alexander Reinefeld, editors. *SCI: Scalable Coherent Interface: Architecture and Software for High-Performance Computer Clusters*, volume 1734 of Lecture Notes in Computer Science. Springer-Verlag, 1999.
6. 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 *Proceedings of Euro-Par'98 Parallel Processing / 4th International Euro-Par Conference Southampton*, volume 1470 of Lecture Notes in Computer Science, pages 206–215, UK, September 1998.
7. Wolfgang Karl, Markus Leberecht, and Michael Oberhuber. SCI monitoring hardware and software: supporting performance evaluation and debugging. In Hermann Hellwagner and Alexander Reinefeld, editors, *SCI: Scalable Coherent Interface: Architecture and Software for High-Performance Computer Clusters*, volume 1734 of Lecture Notes in Computer Science, chapter 24. Springer-Verlag, 1999.
8. Wolfgang Karl, Markus Leberecht, and Martin Schulz. Supporting shared memory and message passing on clusters of PCs with a SMiLE. In *Proceedings of the third International Workshop, CANPC'99*, volume 1602 of Lecture Notes in Computer Science, Orlando, Florida, USA(together with HPCA-5), January 1999. Springer Verlag, Heidelberg.
9. Wolfgang Karl and Martin Schulz. Hybrid-DSM: An efficient alternative to pure software DSM systems on NUMA architectures. In *Proceedings of the 2nd International Workshop on Software DSM (held together with ICS 2000)*, May 2000.
10. Scott C. Karlin, Douglas W. Clark, and Margaret Martonosi. SurfBoard–A hardware performance monitor for SHRIMP. Technical Report TR-596-99, Princeton University, March 1999.
11. Michael Manzke and Brian Coghlan. Non-intrusive deep tracing of SCI interconnect traffic. In *Conference Proceedings of SCI Europe'99*, pages 53–58, Toulouse, France, September 1999.
12. 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/.
13. Luiz De Rose and Mario Pantano. An approach to immersive performance visualization of parallel and wide-area distributed applications. In *Proceedings of the International Symosium on High Performance Distributed Computing*, Redondo Beach, CA, August 1999.
14. Martin Schulz. True shared memory programming on SCI-based clusters. In Hermann Hellwagner and Alexander Reinefeld, editors, *SCI: Scalable Coherent Interface: Architecture and Software for High-Performance Computer Clusters*, volume 1734 of Lecture Notes in Computer Science, chapter 17. Springer-Verlag, 1999.
15. Martin Schulz. Efficient coherency and synchronization management in SCI based DSM systems. In *Proceedings of the SCI-Europe, held in conjunction with Euro-Par 2000*, pages 31–36, Munich, Germany, August 2000.
16. J. Siegel. *CORBA - Fundamentals and Programming*. John Wiley & Sons, 1996.
17. Jie Tao, Wolfgang Karl, and Martin Schulz. Understanding the behavior of shared memory applications using the SMiLE monitoring framework. In *Proceedings of the SCI-Europe, held in conjunction with Euro-Par 2000*, pages 57–62, Munich, Germany, August 2000.
18. R. Wismüller. State based visualization of PVM applications. In *Parallel Virtual Machine –EuroPVM'96*, volume 1156 of Lecture Notes in Computer Science, pages 91–99, Munich, Germany, October 1996.
19. R. Wismüller. Interoperability support in the distributed monitoring system OCM. In *Proc. 3rd International Conference on Parallel Processing and Applied Mathematics - PPAM'99*, pages 77–91, Kazimierz Dolny, Poland, September 1999.