OptiSCI: a Visual Environment to Optimize the Placement of Shared Memory Segments on a SCI Cluster*

Rafael B. Ávila[1], César A. F. De Rose[2], Tiago Ferreto[2], Marcos E. Barreto[1], Philippe O. A. Navaux[1], Hans-Ulrich Heiß[4], Roberto A. Hexsel[3]

[1] Institute of Informatics
Federal University of Rio Grande do Sul
PO Box 15064 — 90501-970 Porto Alegre
Phone: +55 51 316-6165 Fax: +55 51 319-1576
{avila,barreto,navaux}@inf.ufrgs.br
Brazil

[2] Catholic University of Rio Grande do Sul
Brazil

[3] Federal University of Paran´a
Brazil

[4] University of Paderborn
Germany

Abstract—

The Scalable Coherent Interface (SCI) is an IEEE interconnection standard which is becoming widely used for the construction of parallel clusters of workstations. SCI provides a hardware-supported common address space shared by the computing nodes, enabling the use of shared-memory as a distributed communication mechanism. Due to the strong NUMA characteristic of SCI-based clusters, the placement of shared segments among the nodes has to be carefully planned, in order to minimize the costs of network communication latencies. In this paper we present a visual tool, OptiSCI, to help in the task of placing shared memory segments onto a SCI cluster with respect to such communication costs. OptiSCI is composed of a graphical modelling tool and a hardware simulator which makes use of a detailed cost model of SCI clusters in order to produce reliable results. At the end the tool is validated against a real implementation of a parallel application on a Linux SCI cluster.

Keywords—**SCI, distributed shared memory, NUMA, placement strategy.**

I. Introduction

The widely-spread practice of cluster computing [9, 3] has stimulated the development and implementation of many high-performance communication technologies such as Myrinet [1] and SCI (Scalable Coherent Interface) [8] for standard personal computers. Though established in 1992, the SCI standard has only recently been implemented in practice as PCI and SBus interface cards [4] by the Norwegian company Dolphin Interconnects. As a result, SCI has been gaining attention, in the last few years, as an interconnect technology for the establishment of parallel clusters [5].

The main differential characteristic of SCI is the ability to transform a distributed-memory cluster in a shared-memory machine. SCI provides a hardware-supported common address space which is shared by the nodes of a cluster, which allows communication between nodes to be performed by regular memory-access instructions, consequently transparent to the programmer.

Such a distributed shared-memory machine is referred to as a NUMA (Non-Uniform Memory Access) [7] machine, where accesses to remote memory are considerably more expensive than local accesses. For this reason, applications based on shared-memory cannot simply be ported unmodified from SMP architectures to SCI-based clusters, what usually results in very poor performance [2]. In other words, shared-memory applications for SCI-based clusters must be carefully planned, in terms of placement of shared memory segments, in order to minimize the cost of remote memory access.

With the goal of minimizing the effort in this task, we have developed *OptiSCI*, a visual tool designed for modelling and evaluating the performance of DSM applications when run on clusters connected by SCI. The idea is that OptiSCI be used to obtain, by simulation, an overall placement strategy to guide the actual implementation of the application. The tool is the result of an international cooperation project between the University of Paderborn, in Germany, and the Brazilian universities UFRGS, PUCRS and UFPR, within the CAPES/DAAD cooperation programme.

The paper is organised as follows: Section II presents

*Work supported by the PROBRAL Project No. 065/98, CAPES/DAAD International Cooperation Programme

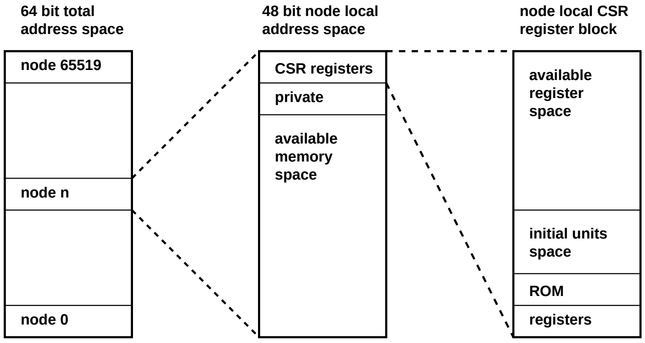


Fig. 1. The SCI address map.

some background information on SCI; in Section III we present the OptiSCI tool and describe its functionalities; Section IV brings the validation of OptiSCI with the implementation of a distributed scheduling application; finally, Section V brings the authors' conclusions and future directions.

II. Background

The *Scalable Coherent Interface* (SCI) is an IEEE standard that provides computer-bus-like services to a set of nodes via fast unidirectional links connected in a ring. SCI uses a point-to-point interface between the network nodes, which allows several topologies like rings, meshes, multistage networks and crossbars to be chosen. The ring topology, however, is especially suitable since it is very simple and inexpensive to realize. The SCI standard specifies the supported interface to run at 500MHz over 16 parallel signals yielding a raw point-to-point throughput of 1GB/s.

The standard implements a 64-bit address space which is divided on 16-bit unique node IDs with each node having a 48-bit local address space, as shown in Fig. 1. The uppermost 16 node addresses are reserved for special purposes, thus leaving 65520 possible nodes in a SCI system.

A SCI cluster therefore does not only provide the facilities for messsage passing communication, but also enables parallel programs to use shared memory segments. Unfortunately, the PCI-based implementation has, unlike the original IEEE standard, a major drawback: the idea of the standard is to have a cache coherent system which spreads over the whole cluster onto many nodes. Every node has its private memory and, in addition to the built-in Level 1 and Level 2 caches, a SCI cache for caching remote memory. The caching of remote memory is, however, not possible for PCI-based systems, since transactions on the main bus of a local node are not visible on the PCI bus (Fig. 2). Thus, a PCI-based card like the Dolphin SCI card used in this paper cannot take part in the coherence protocol on the main bus. Caching of local memory exported by an SCI card is nevertheless possible when accesses from a PCI card into the main memory go through the same bus as CPU-memory traffic, which is the case of standard PCI-based PCs.

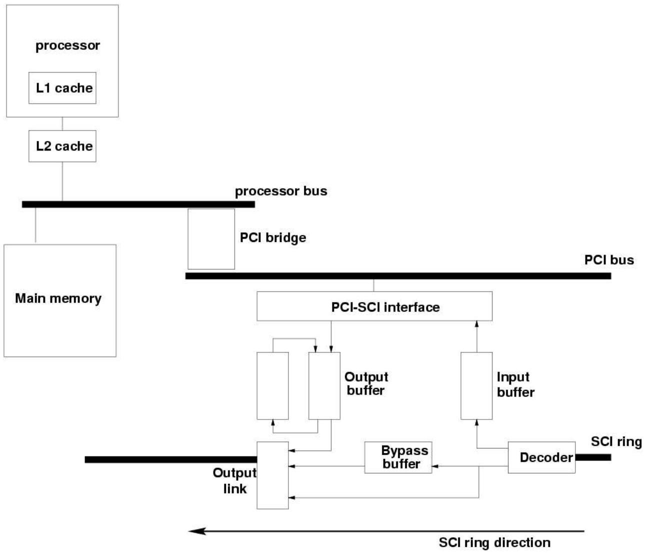


Fig. 2. A PCI-based SCI node.

III. The Visual Optimization Environment

Our proposal to minimize the effort in correctly placing shared segments consists of a visual environment, called OptiSCI, where DSM applications can be modelled and evaluated in relation to performance. The idea is, however, not to construct each detail of the application, but rather to build a general model of its behaviour by giving an initial segment distribution and describing remote accesses to them. After evaluating this first configuration, the user can easily move segments around and reevaluate the application as many times as desired to obtain an optimal placement strategy, which shall guide the final implementation.

The environment is divided in two graphical tools: DAMIT (DSM Application Modelling Tool), with which the user interacts directly during the modelling step, and SCIPos (SCI Placement Optimization System), which simulates the hardware and evaluates the application. These tools are presented next.

A. *The* DAMIT *Application*

This tool, implemented in Java for its powerful GUI development support, provides a graphical interface for constructing DSM applications. It has been implemented on the Brazilian side of the cooperation by the groups at UFRGS and PUCRS.

Figure 3 shows a screen-shot of the tool. As mentioned before, DAMIT is used by the programmer to model and later fine-tune the application.

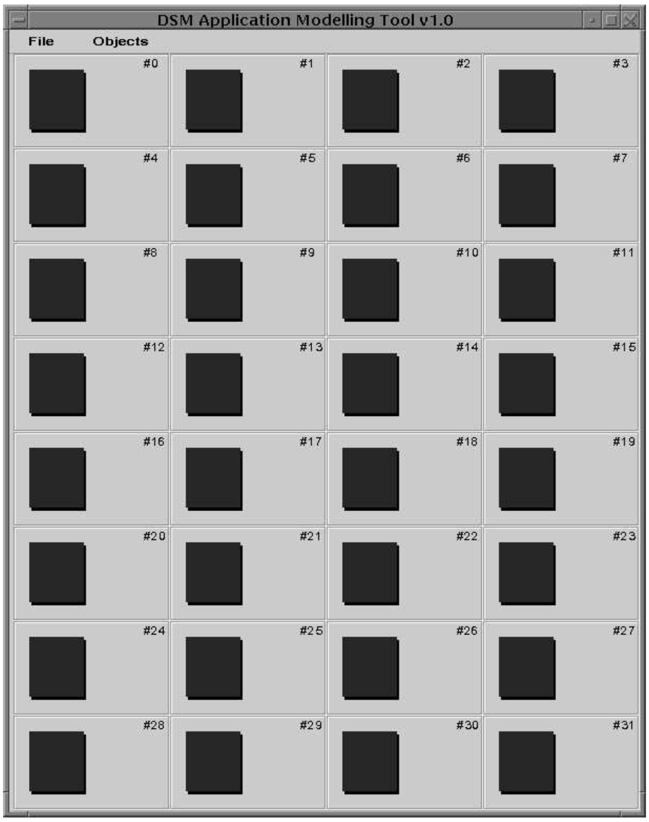
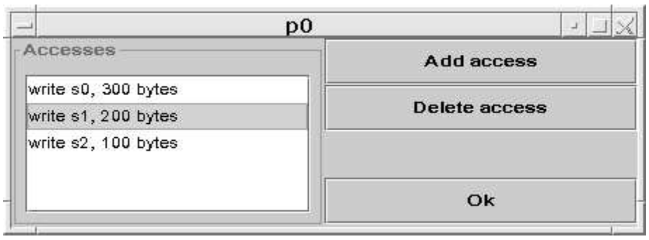


Fig. 3. Screen-shot of the DAMIT application.



write s0, 300 bytes
write s1, 200 bytes
write s2, 100 bytes

Fig. 4. Definition of accesses to shared segments.

Fig. 5. The SCIPOS application.

The main screen shows the available computational nodes in a grid layout, where each node holds one single process, represented by a dark box. Shared segments can be created in the available space within a node, and can later be moved from one node to another. The behaviour of the application is modelled by double-clicking a process and adding accesses to shared segments, as illustrated in Fig. 4. These accesses are represented by lines drawn from a process to a node (see Fig. 9). Different colors are used to separate read accesses (more expensive) from write accesses.

The DAMIT tool generates as output a data file containing incidence matrices which represent the communications involved in the application. This file is then used as input for the second tool.

The moving of shared segments from node to node is the key point of the DAMIT tool. The idea is that many different configurations can be tested in a short period of time, so that the final implementation can use the placement strategy obtained from this step. Section IV shall present a concrete example in the use of the tool.

B. *The* SCIPOS *Program*

The SCIPOS program, developed jointly by the groups at UFPR and Uni-Paderborn, is written in ANSI C and simulates the actual communication process. After importing the matrices generated by DAMIT, the program calculates the costs involved in the remote accesses between nodes and displays a graphical screen (Fig. 5), similar to that of DAMIT, showing the nodes and the costs generated by each one in terms of distributed shared memory and message passing[1]. Both the total and the individual costs for remote reading and writing are displayed (Fig. 6). Based on these information, the user is able to estimate the performance of the application and decide on additional reconfiguration.

[1] SCIPOS has been designed to support this kind of communication as well.

Fig. 6. Communication costs as informed by SCIPOS.

B.1 Hardware Model

In order to present results as reliable as possible, the estimation of communication costs in this part of the environment has been given detailed attention, being based on a further development of the measurements done by Hexsel [6] on SCI-based architectures. This section discusses the hardware modelling used in the environment.

Measurements represent communication costs. In some cases, the original values have been combined or used to construct average timings in order to simplify the model and increase the performance of the implementation. Hardware contention is not considered in the cost model. All values depend on the $size$ of data (in bytes) that is being processed and on the clock frequencies of the individual hardware components:

- $MCycle$: one cycle of the memory bus (100MHz)
- $PCycle$: one cycle of the PCI bus (33MHz)
- $BCycle$: one cycle of the PCI-SCI bridge (internal clock frequency: 1000MHz)
- $SCycle$: one cycle of the SCI ring (500MHz)

From these basic values, the other data are constructed as follows:

- $T_{memory\_access} = 14 * MCycle * \frac{size}{8}$
  Accessing the local memory
- $T_{crossing\_PCI} = 3 * PCycle * \frac{size}{4}$
  This time is needed to cross the PCI bus via processor-PCI bridge and reach the PCI-SCI bridge
- $T_{preparing\_SCI} = 12 * BCycle * \frac{size}{64}$
  Time needed for wrapping up the data
- $T_{insertion} = 200ns * \frac{size}{64}$
  Time for inserting the packet onto the SCI ring
- $T_{pass\_node} = 4 * SCycle + 25ns$
  The time used for passing through a node on the SCI ring including the propagation delay in 5m cable to the next node.
- $T_{changering}(size) = 12 * BCycle * \frac{size}{64}$
  Time needed for a SCI packet to change dimensions in the SCI torus
- $T_{receive} = 8 * BCycle * \frac{size}{64}$
  Removing the packet from the ring and unwrapping it in the target node.

Remote Writing A write access to a remote segment in the SCI-cluster consists of the following steps: the instruction to write certain data in a remote segment is given by the CPU, the required data is fetched from the local memory via the memory bus into the processor-SCI bridge, then onto the PCI bus. In the PCI-SCI bridge, the data is wrapped in SCI packets and sent via the linkbus onto the SCI ring. Routing follows the SCI specifications, i.e. the packets travel along the horizontal ring until the matching vertical ring is met; then they change direction and move on until the target node is reached. There the SCI card picks up the packages and transfers them via linkbus to the PCI-SCI bridge, where the original data format is restored. After passing PCI bus and processor-PCI bridge, the data is stored in the remote segment via the memory bus.

The nodes shall be referred to as $A = \alpha_{i,j}$ resp. $B = \alpha_{k,l}$, the summations are provided without possible *modulo* operations for better reading. So, in total, the time needed for a remote write can be formalized as follows:

$$\begin{aligned}
T_{remote\_write}(A,B) \quad = \quad & T_{memory\_access}(\alpha_{i,j}) \\
& + T_{crossing\_PCI}(\alpha_{i,j}) \\
& + T_{preparing\_SCI}(\alpha_{i,j}) \\
& + T_{insertion}(\alpha_{i,j}) \\
& + \sum_{m=i}^{k} T_{pass\_node}(\alpha_{m,l}) \\
& + T_{changering} \\
& + \sum_{m=j}^{l} T_{pass\_node}(\alpha_{i,m}) \\
& + T_{receive}(\alpha_{k,l}) \\
& + T_{crossing\_PCI}(\alpha_{k,l}) \\
& + T_{memory\_access}(\alpha_{k,l})
\end{aligned}$$

Remote Reading A little more effort is needed to *read* from a remote segment since there is no possibility of a *read-ahead* or buffering data requests. Actually, the read access on a shared memory segment can be compared to a kind of *send_data* instruction which follows a remote write from target to source node:

$$\begin{aligned}
T_{remote\_read}(A,B) \quad = \quad & T_{remote\_write}(A,B) \\
& + T_{crossing\_PCI}(\alpha_{k,l}) \\
& + T_{preparing\_SCI}(\alpha_{k,l}) \\
& + T_{insertion}(\alpha_{k,l}) \\
& + \sum_{m=j}^{l} T_{pass\_node}(\alpha_{i,m}) \\
& + T_{changering} \\
& + \sum_{m=i}^{k} T_{pass\_node}(\alpha_{m,l}) \\
& + T_{receive}(\alpha_{i,j}) \\
& + T_{crossing\_PCI}(\alpha_{i,j}) \\
& + T_{memory\_access}(\alpha_{i,j})
\end{aligned}$$

On the following lines, the number of processes to be mapped shall be denoted with k, the number of available segments with m. The number of nodes on which the processes and segments are to be placed is given by the letter n. Thus, the actual event of allocating the processes onto the nodes can be described as a function

$$\pi : \{1 \ldots k\} \to \{1 \ldots n\}$$

and also the placing of the segments is a function

$$\phi : \{1 \ldots m\} \to \{1 \ldots n\}$$

so that the time needed for the communication of process i can be formalized as

$$\begin{aligned} T_{sm}(i) &= \sum_{j=1}^{m} T_{remote\_read}(\pi(i), \phi(j)) \\ &\quad + \sum_{j=1}^{m} T_{remote\_write}(\pi(i), \phi(j)) \end{aligned}$$

Hence the overall cost function to be minimized is given as

$$\Phi = \sum_{i=1}^{k} T_{sm}(i)$$

This is the value presented by SCIPOS as the global shared memory cost, which must be observed by the programmer when fine-tuning the application.

IV. Tool Validation

In this section we will use the OptiSCI tool to model a distributed application and validate the simulated results with a distributed implementation of the same application in a SCI cluster.

As target machine we used the SCI cluster from the Research Center for High Performance Computing (CPAD–PUCRS/HP). The cluster is composed of 4 nodes connected by a 500MHz unidirectional SCI ring. Each node is a HP Vectra VE8 with a Pentium III running at 550MHz and 128MB main memory. The operating system used in these nodes is RedHat Linux R6.2, with kernel 2.2.14.

The application has been developed with Yasmin (*Yet Another Shared Memory Interface*) [10], a shared-memory API from the University of Paderborn. The base of Yasmin programming is the creation and sharing of memory segments, with synchronisation by means of distributed mutexes and conditional variables, among others.

The chosen example application is a distributed scheduler using three ready queues for tasks of different priorities, two master and two slave processes. It is a classical consumer/producer model with slaves writing to the queues according to the priority of the incoming tasks (producers) and masters reading from the queues to execute the tasks (consumers). The queues are processed in FIFO order, with slaves writing at the tail and masters reading from the head. Each master scheduler is associated to one processor and will read from the queues in a round robin fashion giving more attention to higher priority queues. In a round a master read three tasks from queue #0, two tasks from queue #1 and only one task from queue #2, resulting in more CPU time to higher priority queues. There is no explicit communication between processes and synchronization is implicit in the access to the shared memory segments (queues). Figure 7 presents a graphical representation of the distributed application (a) and a data access graph (b).

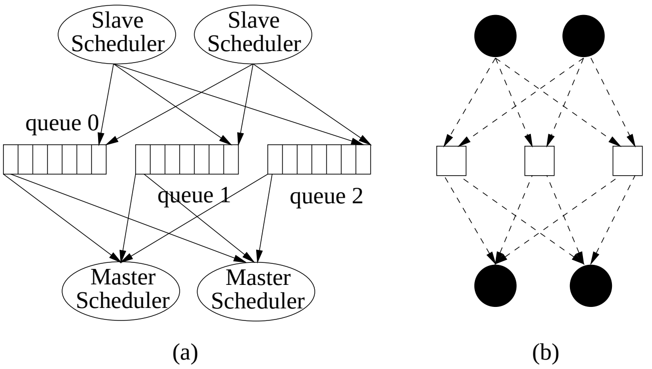


Fig. 7. Representation of the modelled application.

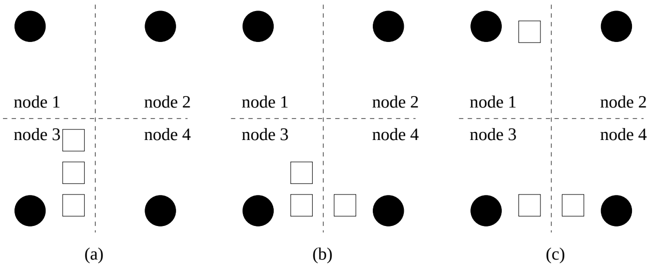


Fig. 8. Three possible mappings for the application.

The problem now is to suitably map the graph in Fig. 7b to the 4 nodes of the target machine. Figure 8 presents three possible solutions to this problem. Processes are represented by circles and the queues by squares. Grey circles represent the slaves and black circles the masters.

It is reasonable to map each process of the distributed application to one node of the target machine but the placement of the queues is less obvious. To help deciding each of the three placements should be used we apply the visual tools. The three mappings have been built and the corresponding costs calculated. Figure 9 shows a view of mapping *a* modelled with DAMIT. For clearness, only the accesses to

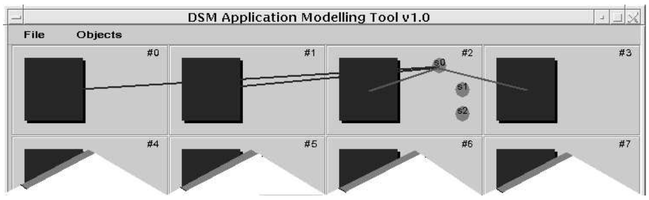


Fig. 9. Partial representation of mapping *a*.

TABLE I
EXECUTION TIMES AND SIMULATION RESULTS FOR THE 3 MAPPING STRATEGIES.

Mapping	time (s)	simulation
a	??	??
b	??	??
c	??	??

queue #0 are shown. Moreover, since the available cluster is composed of 4 nodes, only the upper ring of the modelled hardware is used. Due to the routing mechanism of SCI, the additional nodes simply do not get involved in the application, and the simulation results are not affected.

To validate the results obtained with the visual tool we implemented three versions of the distributed scheduler, one for each mapping of Fig. 8. Each slave scheduler has to handle 600 tasks, 300 with priority #0, 200 with priority #1 and 100 with priority #2. No arrival times are modelled for these tasks (the slave input buffer is full by program start). The program ends when the masters finish processing the 1200 tasks out of the three queues. The task representation to be inserted and removed in the queues is 2KB long.

Table I presents a comparison between the execution times and the simulated results for each mapping. It is possible to observe that the variations in values from one mapping to another are proportionally kept the same for both columns, which leads us to conclude that the tool is effective in simulating the SCI hardware.

V. CONCLUSIONS AND PERSPECTIVES

The design of shared-memory applications on SCI-based clusters is still a task that demands careful planning. The OptiSCI environment presented in this paper is an attempt to minimize the effort in this task. By means of a quick modelling/evaluation process, the programmer of SCI is able to obtain a *quasi*-optimal segment placement strategy to be followed at the time of the actual implementation.

We have shown an example of OptiSCI use in the modelling of a distributed scheduling application, validated against a real implementation of the same algorithm with the Yasmin API. The obtained results show the effectiveness of the tool, having presented performance variations which correspond to the measures done in practice, within a 10% error margin.

This version of OptiSCI does not find the best mapping solution, but only calculates the involved costs for the solution proposed by the user. This forces the user to be involved in the mapping process and helps him to better understand the cost model. For the future we consider the addition of mechanisms to suggest better mapping alternatives.

Being NUMA machines, SCI clusters represent a new challenge for parallel and distributed applications programmers. The resulting applications will run efficiently only if programmers master the mapping cost model. Our tool aims to help the better understanding of the involved costs in this procedure and is a step in this direction.

ACKNOWLEDGEMENTS

The authors would like to thank CAPES and DAAD for the financial support of this project. The work is also partially supported by studentship grants from CNPq and CAPES.

REFERENCES

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

[2] Roger Butenuth and Hans-Ulrich Heiss. Shared memory programming on PC-based SCI clusters. In Hermann Hellwagner and Alexander Reinefeld, editors, *Proc. of SCI-Europe'98*, Bordeaux, France, September 1998.

[3] Rajkumar Buyya, editor. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, Upper Saddle River, 1999.

[4] The Dolphin SCI interconnect. Available at http://www.dolphinics.no, 1996.

[5] Hermann Hellwagner and Alexander Reinefeld, editors. *SCI: Scalable Coherent Interface: Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *Lecture Notes in Computer Science*. Springer, Berlin, 1999.

[6] Roberto A. Hexsel. *A Quantitative Performance Evaluation of SCI Memory Hierarchies*. PhD thesis, University of Edinburgh, October 1994.

[7] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, Boston, 1998.

[8] IEEE. IEEE standard for scalable coherent interface (SCI). IEEE 1596-1992, 1992.

[9] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. *How to Build a Beowulf: a Guide to the Implementation and Application of PC Clusters*. MIT, Cambridge, 1999.

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