A Programming Interface for NUMA Shared-Memory Clusters

Marcus Dormanns, Walter Sprangers, Hubert Ertl, Thomas Bemmerl
Lehrstuhl für Betriebssysteme, RWTH Aachen
Kopernikusstr. 16, D-52056 Aachen, Germany
Email: `contact@lfbs.rwth-aachen.de`

Abstract. *We describe a programming interface for parallel computing on NUMA (Non-Uniform Memory Access) shared memory machines. Although the interest in this architecture is rapidly growing and more and more hardware manufacturers offer products of this type, there is still a lack in parallelization support. We developed SMI, the Shared Memory Interface, and implemented it as a library on an SCI-coupled cluster of workstations. It aims at providing sophisticated support to account for the NUMA performance characteristic and to allow a step-by-step parallelization. We show it's application to the parallelization of a sparse matrix computation.*

Keywords: *parallel programming interface, shared memory parallelization, NUMA multiprocessor*

1 Introduction

The recently emerging trend in parallel computer architecture is directed towards *scalable shared memory machines* [17]. I.e. machines with a globally shared address space across the entire machine but comprised out of distributed compute nodes. Each node is equipped with a local memory module and adapted to an interconnect, serving memory accesses to addresses residing in a non-local memory module. This architectural principle results in a NUMA (non-uniform memory access) performance characteristic, i.e. local memory accesses are served much faster than remote ones. Due to the at least partially separated memory access transactions among the compute nodes, these machines scale to a higher degree of parallelism than traditional bus-based symmetrical multiprocessors (SMPs) that suffer from central bottlenecks.
There are quite a lot of good reasons for this trend. From an application programmer's point-of-view, the global address space allows shared data parallelization, that is commonly preferred to message passing due to it's simplicity and the opportunity of a smoothly step-by-step parallelization considering the amount of code running in parallel. Considering performance, data exchange on a very fine-grain but low-latency basis (i.e. single cache-lines) permits comfortable parallelization also for algorithms with very irregular and maybe dynamically changing access patterns.
Besides several research prototypes [17,21], already available commercial machines of this type are the HP/Convex Exemplar series [7] and the SGI Origin. Data General [6] and Sequent [18] announced enterprise server clusters comprising of off-the-shelf Intel PentiumPro multiprocessor modules. Those from HP/Convex, Data General and Sequent are based on the IEEE-standardized Scalable Coherent Interface (SCI) [14]. SCI defines the physical link and logical protocol layer of a high-speed network (based on Gigabit and Gigabyte signalling technology). Additionally, SCI contains a memory coherence protocol layer, which is the most distinguishing feature, since it provides cache coherency of physically distributed memory across an entire cluster. DEC's Memory Channel is something similar but based on the different Reflective Memory

technology [13]. Beside these systems, a very interesting good value alternative is the clustering of workstations or PCs with SCI adapter cards that are already available for Sun's SBus and announced for the PCI bus from the Norwegian company Dolphin [8,9]. Common to all those platforms, in contrast to former parallel machines, is that they are targeted to the commercial market sector: e.g. transactions processing systems and decision support systems. This raises the hope that parallel machines of this type might spread and survive. Nonetheless, they also offer an exciting new perspective for low-cost parallel scientific computing (cf. [3]).

Not unusual in computer science, software support to facilitate the comfortable development of parallel programs and parallelization of already existing ones on these platforms hangs behind the hardware development. In this paper, we attend to this issue in that we present a programming interface, SMI (Shared Memory Interface), for parallel computing on NUMA shared memory machines, mainly but not exclusively targeted to the technical/scientific area. SMI is implemented as a library, to be used as a high-level parallelization extension to common programming languages like C and Fortran, like most of the message passing libraries for distributed memory machines. It is build on top of the common rudimentary capability of the underlying platforms to install segments of shared memory between processes. The main design goals are:

- *Ease-of-Parallelization*: Comfortable step-by-step parallelization, requiring only minor code changes without the need to tackle the whole code right from the beginning.
- *Performance*: Providing sophisticated support for the parallelization work, particularly accounting for the NUMA performance characteristic [1].
- *Portability*: The interface should not rely on any capability special to a single NUMA system.

In contrast to shared object libraries like e.g. *Global Arrays* [19], which also provide high-level support for shared data structures but that can only be accessed and processed with certain library functions, SMI directly allows entirely transparent shared data structures - just shared memory. SMI provides user-control over the physical location of shared data structures, which finally determines which process(es) has a local, fast access to it and which process(es) a remote, expensive one. Moreover, an address space region of a single data structure can be partitioned with each part mapped to a different memory module but still remaining continuously addressable. SMI allows to temporarily switch-off the very expensive but often unnecessary consistency protocol between shared data. After a phase, during that consistency is not necessary, a consistent globally shared view can be re-established with a couple of partially general-purpose and partially application-specific functions.

At this time, we have implemented SMI on a cluster of Sun multiprocessor workstations, coupled with plug-in SCI adapter cards from Dolphin [8,9,11,20].

The reminder of this paper is structured as follows: section 2 explains and provides arguments for the chosen operational model. The design of the programming interface to write parallel programs for such an environment is described in section 3. To demonstrate it's comfort and capacities, section 4 discusses the parallelization of an irregular sparse matrix vector multiplication as a small but typical example. Finally, in section 5 some conclusions are drawn and an outlook is given onto our future work.

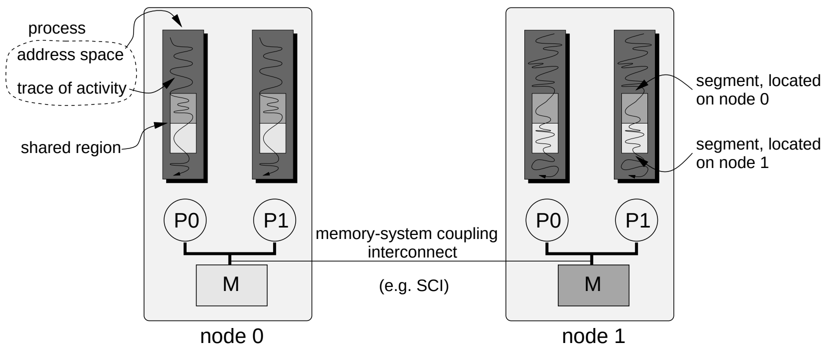
process
address space
trace of activity
shared region
segment, located on node 0
segment, located on node 1
P0
P1
M
memory-system coupling interconnect
(e.g. SCI)
node 0
node 1

Figure 1: Exemplary operational model, consisting of two multiprocessor nodes and two processes executing on each node with one common shared region. The shared region is partially located in the memory of node 0, partially in that of node 1.

2 The Operational Model

A SMI application is executed by a couple of concurrent processes (e.g. standard UNIX processes), initially each with a private virtual address space. By calling the respective SMI function, processes can install segments of shared memory among them to store certain data structures inside (see figure 1). This is a SPMD (Single Program Multiple Data) model, in that each process executes the same code (asynchronously if not explicitly synchronized) on different portions of shared data.
In contrast to a thread-based model, that is already available for several years on most small-scale SMPs with an entirely shared address space, our model possesses several advantages:

- Typically, a considerable amount of memory accesses regard data structures that are more or less entirely accessed by all processes, e.g. tables of physical properties of atoms in a molecular dynamics simulation. Even more, a lot of them are accessed read-only. Within a thread-model, just one instance of this data in the whole system exists. In a NUMA system, all but the processes on the home node of such data suffer from expensive remote memory accesses to this data. A replication has to be performed explicitly together with the code changes coming along. Not so in our framework where only some data structures are shared. All but these are replicated, i.e. accesses are local to them.
- In a thread-model no facility is provided to take care of the physical home node of shared data, that is performance critical as outlined above. Most times, a reasonable parallelization means that most of the data is processed by just one process, real data sharing is restricted to a small fraction. Therefore, a reasonable mapping is not only required but also possible.

Besides these performance motivated arguments, another one comes from portability consideration. Spreading threads of a single process among several compute nodes of a NUMA multiprocessor is only possible if they are managed collectively by a single system-wide operating system. This is not true for SCI-clustered workstations or PCs wherein an independent copy of the operating system runs on each (SMP-) node, making use of the shared memory only at user-level.

3 The Interface Design

The most fundamental of SMI is it's flexibility in allocating a piece of memory, shared among the processes, especially in the NUMA environment. This is treated in a first subsection, followed by a short description of all the essential but common features of a parallel programming library like process identification and synchronization facilities. We then describe the possibilities to switch between different modes of consistency in a separate subsection though it is the central issue to obtain high performance and to allow a step-by-step parallelization.

3.1 Building Blocks: Pieces of Shared Memory

Within SMI, an individually manageable piece of shared memory is called a *shared region*. The intention is, that a single data structure corresponds to a shared region (whether it is static or dynamic). This notation follows [23], who introduced a shared virtual memory (SVM) system that maintains cache coherency not on page- but on region-granularity, typically a single data structure. A shared region can be composed out of several consecutive addressable *shared segments*, each locatable on a different compute node, accounting for the NUMA architecture. A shared region can be allocated with the function:

```
SMI_Create_shreg(int size, int dist_policy, int* dist_value,
                 int* region_id, void** address)
```

`size` specifies the amount of memory allocated. The distribution policy, i.e. the detailed composition of shared segments and their physical home node, is specified with the `dist_policy` parameter in conjunction with `dist_param`. At this time, three different policies are implemented:

- `UNDIVIDED`: The whole region consists of a single segment that is entirely located on the compute node of the process given in `dist_param.`
- `BLOCKED`: The region is split evenly into as many segments as there are processes. Segment `i` is located in the memory module of the compute node of process `i`.
- `CUSTOMIZED`: A user-defined splitting can be specified. `dist_param` is an array of parameters: the first element specifies the number of constituting segments, the following specify each segment's size and physical location.

An identifier for the region and it's starting address in the virtual address space is returned. SMI ensures that the region is mapped to identical virtual addresses in all processes. This is important since it allows the mutual exchange of pointers between processes to locations inside such a region. A region can be used to store a static data structure, e.g. an array of some type. For this purpose, no further special support from SMI is required. Alternatively, it can be used for dynamic memory allocation. If this is desired, SMI initializes all necessary data structures of a memory manager for a specified shared region. To make use of it, SMI provides the following four functions:

```
SMI_Cmalloc(int size, int region_id, void** address)
SMI_Cfree(void* address)
```

and

```
SMI_Imalloc(int size, int region_id, void** address)
SMI_Ifree(void* address)
```

Using these function, each process can allocate/free a piece of memory inside a specified region. The `C`- and the `I`-functions differ in their scope. While `C` denotes that the function has to be called collectively by all processes to allocate a single common piece of memory, `I` means that it is just individually called from a single process. The result regarding the memory allocation is both times the same. But in the first case, all processes gain knowledge of the address of the allocated piece of memory while in the second case just the calling process does. If another process needs to now it, it has explicitly to be notified about it. The memory management data structures are also kept globally shared such that each process can request a piece of memory in each region. This mode of usage only makes sense in conjunction with the distribution policy `UNDIVIDED`. In all other cases, the performance-critical property of the physical home node of an allocated piece of memory is beyond the control of the user. A typical usage could be to allocate one `UNDIVIDED` shared region for dynamic memory allocation at each process and to store their identifiers in a special array, indexed by process number. With this, shared memory can simply be allocated on each's process node on demand.

3.2 The Normal: Process Identification, Synchronization, ...

The usage of SMP compute nodes as building blocks for a NUMA shared memory cluster is highly recommended though they state the most cost-effective way of coupling some few processors. Therefore, not all memory accesses to a shared region that are to a segment of a remote process are necessarily also to a remote compute node coming along with high latencies. To account for this in SMI, each process is not only assigned an unambiguous process rank, but also a machine rank. These can be requested by calling respective SMI-functions. Also, a process can inform itself about the machine rank of another process, e.g. to decide if both reside on the same machine. A function is provided to map any process rank to it's corresponding machine rank. To simplify the exploitation of several processes residing on the same machine, SMI guarantees that such processes posses consecutive process ranks. From this follows, that shared segments of consecutive processes of a `BLOCKED` shared region also possess consecutive addresses in the virtual address spaces.

Another essential facility of parallel programming libraries and especially for the shared data model is synchronization. SMI provides barrier- and mutex-synchronization. At this point in time, we experiment with several purely software-based algorithms, especially taking into account the hierarchical UMA/NUMA performance characteristic (cf. [25]).

3.3 The Advanced: Switching Between Different Modes of Consistency

The typically employed protocols to ensure cache consistency are known to influence overall performance of shared memory parallel applications considerably [12]. Accordingly, a lot of work has been done to weaken the rigid sequential consistency [16]. Results are general purpose cache consistency protocols that normally behave as the strict sequential consistency at least to the typical application programmer, like Total Store Order and Processor Consistency [1] but are not as restrictive and therefore result in a better application performance. Especially for software SVM systems, whose performance is very sensitive to overhead induced by cache coherency, even weaker protocols like Release Consistency have been developed. Though they are still

general purpose, some of them require the programmer's involvement, e.g. by inserting respective `acquire/release` synchronization primitives into the code, associated with certain data. The SVM system Munin [4] is an excellent example how a sophisticated choice of the cache coherence protocol in dependence of the data reference pattern can improve performance considerably. One step further goes the mixed software/hardware DSM system Tempest/Typhoon from the Univ. of Wisconsin-Madison [10]. They describe application-specific cache-consistency protocols to improve application performance that are no longer general purpose but especially developed and implemented for certain algorithms.
These findings influenced the design of SMI but the derived solution differs from Munin and Tempest/Typhoon considerably because of the following reasons:

- We want to rely solely on standard hardware, e.g. SCI connected compute nodes. This restricts the consistency protocol to that offered by the underlying hardware, e.g. to sequential consistency in the case of SCI. This prevents SMI from solutions as implemented e.g. in Munin.
- We highly aim at simplicity in parallel program development and parallelization of already existing sequential software. This prohibits to enforce an application programmer to develop and implement a proprietary application specific consistency protocol.

SMI relies on the observation, that it is not necessary to maintain the overhead-inducing data consistency of shared regions all the time, if only it can be re-established at certain points in time. Therefore, SMI provides facility to mutual de-couple the accesses of individual processes to a specific shared region. When entering this mode, each process starts with the same view onto the formerly consistent data. Thereafter, modifications only effect the local processor's view. To implement this, it is not sufficient to switch-off the system's cache-coherency protocol. This would eliminate the cache coherency overhead but would not allow a modification to become not effective outside a process' address space. But this is necessary to allow meaningful temporary inconsistency that can be efficiently recombined to a consistent view afterwards, as will be explained below. In SMI, this de-coupling is implemented by (temporary) mapping a local memory segment within each process to the address, where formerly the shared region was mapped to, and to replicate the data of the shared region into each local segment. This is necessary and results in the following advantages:

- All memory accesses are local ones that are not as expensive as remote ones. The replication can be seen as if local memory is exploited for caching all the data; no remote memory accesses at all occur any longer (cf. [5]).
- Performance degradation due to false sharing [24] does not exists anymore.
- Unnecessary cache consistency does not exist at all.

This mode of usage of a shared region is entered by calling:

```
SMI_Switch_to_replication(int region_id)
```

When switching back to the global sharing mode, consistency between the different processes has to be re-established. Three strategies have been implemented, some of them requiring that the data or the inconsistencies induced during replication obey certain restrictions:

- SINGLE_SOURCE: The data of a specified local copy is taken as the afterwards shared region for all processes.
- MERGE: Assuming that modifications of different processes do not overlap, the shared region is afterwards a merge of the different instances among the processes, reflecting all the modifications.
- COMBINE: The re-established consistent shared region is a computed combination of all processes' instances. At this point in time, the provided combination procedures assume that the data structure inside the shared region is an array of some data-type. For each element, the consistent view is computed by an element-by-element combination of all processes' instances with a commutative and associative function. Beside user-defined functions, accumulation, max and min (of integers, single- and double-precision floating-point numbers) are predefined.

The corresponding function call is simply:

```
SMI_Switch_to_sharing(int region_id, int comb_strategy,
                      int param)
```

Enforcing consistency without switching back to the sharing mode can also be done with:

```
SMI_Consistency(int region_id, int comb_strategy, int param)
```

4 Efficient Programming with SMI

This section discusses the parallelization of a small program that performs a multiplication of the transpose of a sparse matrix with a vector. It was chosen because it allows to discuss several aspects with a single example. As already mentioned, one of the major aims of SMI is to simplify code parallelization, e.g. to allow a step-by-step parallelization and a maximum of code-reuse. For the actual example, this was necessary because we wanted to employ the linear algebra library SPARSKIT [22] with as few changes as possible to speed-up certain time-critical code sections.
First time this applies when the matrix is read from disk. By the exploitation of shared memory, it is not necessary that each process first evaluates which portion it really needs, than searches for it in the (sequential) file and reads it. Because for the file access in a typically environment with a non-parallel file system and with the original data files of the original sequential code, there is no disadvantage to perform it with a single process. The data can be distributed with SMI's comfortable capabilities afterwards, simply by enforcing consistency with the SINGLE_SOURCE mode. This allows for a maximum code reuse at maximum performance under the given conditions:

```
SMI_Init(&argc, &argv);       /* Initialization of SMI     */
SMI_Proc_rank(&proc_rank);    /* inform about process rank */
...
/* allocate shared regions for a sparse matrix in          */
/* compressed row storage(CRS) format: pointer arrays      */
/* ja, ia and the element array a                          */
SMI_Create_shreg(nz*sizeof(double),BLOCKED,&dummy,&a_id,a);
SMI_Create_shreg(nz*sizeof(int),  BLOCKED,&dummy,&ja_id,ja);
SMI_Create_shreg((n+1)*sizeof(int),BLOCKED,&dummy,&ia_id,ia);
```

```
SMI_Switch_to_replication(a_id);
SMI_Switch_to_replication(ia_id);
SMI_Switch_to_replication(ja_id);
...
if (proc_rank==0) {
   /* read matrix from file, employing original code */
   readmt_(&nmax,&nzmax,&job2,&iounit,a,ja,ia,rhs,&nrhs,
           guesol,&nrow,&ncol,&nnz,title,key,type,&ierr);
   ...
}
/* distribute data by simply enforcing consistency */
SMI_Consistency(a_id,   SINGLE_SOURCE, 0);
SMI_Consistency(ia_id,  SINGLE_SOURCE, 0);
SMI_Consistency(ja_id,  SINGLE_SOURCE,0);

/* do other initialization on private data with org. code */
...
```

The most interesting part is the actual multiplication of the transpose of the matrix with a vector. For the two vectors involved in the operation, shared regions are installed as for the matrix:

```
SMI_Create_shreg(n*sizeof(double), BLOCKED, dummy, &x_id, x);
SMI_Create_shreg(z*sizeof(double), BLOCKED, dummy, &y_id, y);
```

The SPARSEKIT subroutine for the multiplication does not require a real change due to parallelization but only a minor adaptation of the parameter list, so that not only the upper loop bound but also the lower one can be passed as an argument.

```
        subroutine atmux (n1, n2, x, y, a, ja, ia)
        real*8 x(*), y(*), a(*)
        integer n1, n2, ia(*), ja(*)

        do 1 i=n1,n2
           y(i) = 0.0
 1      continue

        do 100 i = n1,n2
           do 99 k=ia(i), ia(i+1)-1
              y(ja(k)) = y(ja(k)) + x(i)*a(k)
 99        continue
 100    continue

        return
        end
```

The concurrent invocation of this subroutine to perform the matrix operation $y \leftarrow A^T x$ is simply done by:

```
SMI_Switch_to_replication(y_id);
atmux_(n1, n2, x, y, a, ja, ia);
SMI_Switch_to_sharing(y_id, COMBINE_ADD_DOUBLE, dummy);
```

where `n1...n2` simple states a local part of the whole index range `1...n` in dependence of the local processor rank. The clue comes from the replication and following combination of the solution vector y, in which several partial solutions are accumulated from all processes. Without replication, a considerable part of the accesses are expensive remote ones, that additionally have to be separated from each other with suitable synchronization primitives which are expensive too. Alternatively, the code would have to be modified to replicate the array y from hand and combine the copies afterwards. This is prevented by the general-purpose SMI functionality. Furthermore, the combination can exploit SCI's fast message passing facilities [OmP96] that deliver an order of magnitude more bandwidth than single memory accesses. Or, if each process just modifies a separate region of the vector densely but just a few of all the other elements, this can be exploited by an enhanced version of the combination procedure that only performs the necessary combination for these few elements.

5 Conclusions and Outlook

Taking into account that remote memory accesses are one to two orders of magnitude slower than local one, it should be obvious that usual SMP-style parallel programs are not suited for the highly interesting NUMA shared memory architectures. This implies that there is really a need for programming interfaces that provide a simple but at the same time sophisticated support on this architecture. SMI addresses this issue twofold: firstly, it's operational process model allows a step-by-step parallelization. It is possible to start with only a few shared data structures and the respective small fraction of parallelized code. All other data structures and the code working on them can remain pure sequential. Furthermore, even not all of the code that deals with the shared data structures has to be parallelized in the first step. By switching to replication before and back to sharing after such a sequential phase, such code sections can remain. Secondly, the capability to temporary de-couple consistency and combine the different views afterwards to a shared, consistent data structure again addresses the need to reduce remote accesses and to avoid unnecessary overhead due to the underlying hardware consistency protocol. Definitely, this subject needs further attention to identify additional common algorithmic skeletons that allow a temporary de-coupling of consistency but possess a way to re-establish a shared consistent view afterwards.
A first implementation of the SMI library with a C-binding is nearly complete. However, the recombination algorithms need further improvements regarding performance. In the future, we will work on the parallelization of codes employing SMI, e.g. the molecular dynamics simulation package GROMOS, what requires an additional Fortran-binding. Surely, our knowledge about requirements on such a parallelization interface will grow during this work. This will find it's expression in the evolution of SMI.

References

[1] Abandah, G. A.; Davidson, E. S.: *Characterizing Shared Memory and Communication Performance: A Case Study of the Convex SPP-1000.* Technical Report CSE-TR-277-96, Dept. of EECS, Univ. of Michigan, Ann Arbor, 1996.

[2] Adve, S. V.; Gharachorloo, K.: *Shared Memory Consistency Models: A Tutorial.* WRL Research Report 95/7, Digital Western Res. Labs, Palo Alto, California, 1995.

[3] Bemmerl, T.; Ries, B.: *Programming Tools for Distributed Multiprocessor Environments.* Int. J. of High Speed Comp., Vol. 5, No. 7, pp. 595-615, 1993.

[4] Carter, J. B.; Bennett, J. K., Zwaenepoel, W.: *Implementation and Performance of Munin.* Proc. 13th ACM Symp. on Operating Sys. Principles (SOSP), pp. 152-164, Oct. 1991.

[5] Chandra, R.; Gharachorloo, K.; Soundararajan, V.; Gupta, A.: *Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols.* Proc. 8th ACM Int. Conf. on Supercomputing, pp. 274-288, 1994.

[6] Clark, R.; Alnes, K.: *SCI Interconnect Chipset and Adapter: Building Large Scale Enterprise Servers with Pentium Pro SHV Nodes.* Proc. Hot Interconnects IV, 1996.

[7] Convex Computer Corp.: *Convex Exemplar Architecture.* 1994

[8] Dolphin Interconnect Solutions, AS: *SPARC SBus-SCI Cluster Adapter Card.* White Paper, June 1995.

[9] Dormanns, M.; Sprangers, W.; Ertl, H.; Bemmerl, T.: *Performance Potential of a SCI Workstation Cluster for Grid-Based Scientific Codes.* Proc. High Perf. Computing, 1997.

[10] Falfasi, B.; Lebeck, A. R.; Reinhardt, S. K.; Schoinas, I.; Hill, M. D.; Larus, J. R.; Rogers, A.; Wood, D. A.: *Application-Specific Protocols for User-Level Shared Memory.* Proc. Supercomputing, 1994.

[11] George, A.; Todd, R.; Phillips, W.; Miars, M.; Rosen, W.: *Parallel Processing Experiments on an SCI-based Workstation Cluster.* Proc. 5th Int. Workshop on SCI-based High-Perf. Low-Cost Computing, pp. 29-39, March 1996.

[12] Gharachorloo, K.; Gupta, A.; Hennessy, J.: *Performance Evaluation of memory Consistency Models for Shared-Memory Multiprocessors.* Proc. 4th Int. Conf. on Arch. Support for Prog. Languages and Operating Systems, pp. 245-257, 1991.

[13] Gillet, R. B.: *Memory Channel Network for PCI.* IEEE Micro, pp. 12-18, Feb. 1996.

[14] IEEE: *ANSI/IEEE Std. 1596-1992, Scalable Coherent Interface (SCI).* 1992.

[15] Iftode, L.; Singh, J.P.; Li, K.: *Irregular Applications under Software Shared Memory.* Technical Report TR-514-96, Dept. of Computer Science, Princeton Univ, 1996.

[16] Lamport, L.: *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs.* IEEE Trans. on Computers, C-28(9), pp. 241-248, Sept. 1979.

[17] Lenoski, D. E.; Weber, W.-D.: *Scalable Shared-Memory Multiprocessing.* Morgan Kaufmann Publishers, 1995.

[18] Lovett, T.; Clapp, R.: *STiNG: A CC-NUMA Computer System for the Commercial Marketplace.* Proc. 23rd Annual Int. Symp. on Comp. Architecture, 1996.

[19] Nieplocha, J.; Harrison, R. J.; Littlefield, R. J.: *GLOBAL Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers.* Proc. Supercomputing, 1994.

[20] Omang, K.; Parady, B.: *Performance of Low-Cost UltraSparc Multiprocessors connected by SCI.* Research Report No. 219, Univ. of Oslo, Dept. of Comp. Science, June 1996.

[21] Protic, J.; Tomasevic, M.; Milutinovic, V.: *Distributed Shared Memory: Concepts and Systems.* IEEE Par. & Distr. Technology, Vol. 4, No. 2, pp. 63-79, 1996.

[22] Saad, Y.: *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations.* Technical Report 90-20, Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffet Field, CA, 1990.

[23] Sandhu, H. S.; Gamsa, B.; Zhou, S.: *The Shared Region Approach to Software Cache Coherence on Multiprocessors*. Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, pp. 229-238, 1993.

[24] Torrellas, J.; Lam, M. S.; Hennessy, J. L.: *False Sharing and Spatial Locality in Multiprocessor Caches.* IEEE TOC, June 1994.

[25] Zhang, X.; Yan, Y.; Castaneda, R.: *Evaluating and Designing Software Mutual Exclusion Algorithms on Shared-Memory Multiprocessors.* IEEE Par. and Distrib. Tech., Vol. 4, No. 1, pp.25-42, 1996.