Computing on a Cluster of PCs: Project Overview and Early Experiences

Sven M. Paas, Marcus Dormanns, Thomas Bemmerl, Karsten Scholtyssik, Stefan Lankes
Lehrstuhl für Betriebssysteme, RWTH Aachen
Kopernikusstr. 16, D-52056 Aachen, Germany
Email: contact@lfbs.rwth-aachen.de
WWW:http://www.lfbs.rwth-aachen.de

Abstract. *This paper summarizes some projects that deal with parallel computing on clusters of PCs. Although quite different approaches have been chosen, they have in common that they aim at shared data parallelization on the cluster platform. One approach is based on a SCI (Scalable Coherent Interface) network that provides transparent remote memory access. A programming interface as well as first application parallelization results are summarized. The second approach relies on SVMlib, a new all-software Shared Virtual Memory subsystem (SVM) for Windows NT. In this paper, we discuss design issues, implementation details and first performance measurements of SVMlib.*

1 Introduction

As the performance of commodity off-the-shelf PCs approaches that of workstations, they become increasingly attractive for performance-critical applications, both from the technical computing area as well as from the commercial sector. The growing number of Beowulf-like clusters and their successful application emphasizes this [2,21]. Recently introduced high-performance interconnects, like Myrinet [5], the IEEE-standardized SCI [13] (Scalable Coherent Interface), etc. allow to build clusters of compute nodes with a balanced computation/communication performance ratio, further increasing their suitability for parallel processing.

We build such a PC cluster at our lab. It is interconnected with a general-purpose Fast Ethernet network to handle TCP/IP communication, e.g. for message-passing and the cluster-wide file system. Furthermore, it is equipped with a SCI-interconnect from the norwegian company Dolphin [7]. SCI implements entirely transparent access to memory segments also at remote nodes. This special purpose network is intended for parallel processing. The cluster is operating under Windows NT.

This paper provides an overview of some projects that aim at exploiting this cluster platform for parallel/distributed processing and reports about the experiences we gathered during the work. In distinction to other cluster computing projects, we are mostly interested in shared data parallelization. This paper starts with introducing our cluster platform in section 2. Section 3 describes some work that aims in exploiting SCI's hardware-based shared memory capabilities for parallelization. In section 4, a software-based shared virtual memory system (SVM), named SVMlib, is described.

2 The Cluster Platform

The cluster at our lab comprises out of 4 Intel Pentium singleprocessors and 2 Intel PentiumPro dualprocessors (soon to be expanded by another four). Besides a 100 MBit/s Fast Ethernet LAN, it is equipped with PCI-SCI adapters from the norwegian company Dolphin [7]. These, in conjunction with the device drivers allow to allocate segments of memory on each compute node that can be mapped into the virtual address space of local processes as well as of processes on remote compute nodes. A process on the compute node that owns such a segment has access to it in the ordinary way. Accesses of processes on remote nodes (called remote accesses) are fully transparently operated via the SCI network. However, this comes not for nothing: a remote memory access is on the order of one magnitude more expensive than a local one. A remote write access shows a latency of about 2.5 µs, a read access about 4.5 µs. When writing or reading larger chunks of memory, several stream buffers of the SCI adapter card that can handle multiple outstanding transactions and some prefetching, can be exploited. This leads to a peak bandwidth of about 28 MByte/s for writing and about 8 MByte/s for reading. Due to this performance characteristic, such architectures are classified as NUMA (Non-Uniform Memory Access) shared memory platforms.

A serious issue that influences application parallelization is that although SCI defines a cache coherency layer, the PCI-SCI adapters do not implement it. It is not possible due to the fact that the PCI-bus as an I/O bus does not participate in the local cache coherency mechanisms of the compute nodes.

Assembling such a cluster provides the freedom of choosing between a lot of singleprocessor machines or some fewer dualprocessor- or even fewer quadprocessor-SMPs (Symmetric Multi-Processors). Looking at the price, dualprocessors possess the lowest price per processor. But considering the ratio of local to remote memory for a fixed total number of processors in such a cluster, quadprocessor-SMPs might be attractive. However, scalability of a multiprocessor itself is an issue due to limited bandwidth of each local memory system. Table 1 shows the accumulated memory bandwidths for a different number of active processors in a quadprocessor PentiumPro SMP. They were measured by performing operations on large vectors according to the STREAM benchmark [17].

number of processors	performed vector operation			
	copy	scale	add	triad
1	170.3	172.6	201.0	187.7
2	177.0	178.3	204.8	198.8
3	173.4	170.8	198.2	196.6
4	176.6	176.0	203.2	202.6

Table 1: Accumulated STREAM memory bandwidth (in MByte/s) for a different number of processors in a quadprocessor Intel PentiumPro SMP system.

These results show clearly that the memory system of off-the-shelf PC multiprocessors saturates for memory intensive code (as many from the technical/scientific computing sector are)

already for a single processor. Putting it all together, assembling such a cluster out of dualprocessors seems to be a well balanced compromise.

3 Parallel Programming on a SCI-Cluster

The SCI-related part of the cluster computing effort aims at shared data parallelization, exploiting the capabilities of the SCI network to establish global segments of shared memory. This differs to other approaches that employ SCI just as a fast message-passing interconnect to run message-passing codes on such a cluster [11,12].

3.1 The Shared Memory Interface: A Programming Interface for NUMA Clusters

We have developed a programming interface, SMI (Shared Memory Interface), for shared data parallelization on a SCI-Cluster. SMI is implemented as a library, to be used as a high-level parallelization extension to common programming languages like C, C++ and Fortran. It exists a version for Solaris, Linux and Windows NT. A SMI application is executed by a couple of concurrent processes (e.g. standard UNIX process), initially all with a private virtual address space. By calling the respective SMI functions, processes can install segments of globally shared memory. The programmer can use a SPMD programming model (Single Program Multiple Data), in which all processes execute the same program, or a MPMD programming model (Multiple Program Multiple Data). However, some restrictions apply to MPMD programming model because certain SMI functions have to be called collectively.

The SMI library includes currently about 30 functions, which can be divided into categories:

- Initializing and execution environment
- Shared regions
- Dynamic allocation of globally shared memory
- Synchronization
- Loop-splitting and -scheduling support

More details than in this paper can be found in [8].

3.1.1 Initializing and Execution Environment

The function `SMI_Init(int* argc, char** argv)` initializes the SMI library. This function must be called collectively from all processes. After the execution of this function the following information is accessible:

- the total number of concurrent processes
- the individual rank number of each process
- the total number of compute nodes on that processes of a SMI application are executed on
- the individual rank of the compute node

3.1.2 Shared Regions

A shared region with total size `tsz` can be established among the processes with a collective call to the function:

```
error_t SMI_Create_shreg(int tsz, int dist_policy,
                         int* dist_param, int* id, void** adr)
```

The parameter `dist_policy` determines how a region is composed of shared segments and on which home nodes these are located. The `UNDIVIDED` policy allocates the regions as a single segment that is located on the compute node on that process `dist_param` is executed on. If `BLOCKED` is specified, the region is composed of as many segments as compute nodes are involved, physically located one per compute node. The size of each segment is chosen proportional to the number of SMI processes on the corresponding node. Another policy is `CUSTOMIZED` that allows to arbitrary specify how a region is composed out of segments and their assignment to nodes.

3.1.3 Synchronization

SMI offers exclusive locks and barriers as synchronization primitives. Both are implemented by pure user-level software algorithms with active waiting (also called spinning). Definitely, active waiting is a wastefulness of processor time, but there are some reasons to do so:

- Current applications that are parallelized based on SMI come from the technical/scientific computing sector. For this application area, it is typical to install as many processes/threads as processors are under disposal. An essential precondition for good performance is that there is just minimum idle time due to synchronization. But in such circumstances, active waiting is no problem, because it is rare and does not block a processor that could otherwise perform more valuable tasks.
- Blocking synchronization primitives are operating system functions. All such functions come along with a significant overhead. User-level synchronization primitives avoid this overhead.
- Last but not least, implementing blocking (kernel-level) synchronization primitives requires to be able to trigger interrupts at a remote compute node. The SCI standard defines such transactions, but the currently available hardware/driver does not provide this functionality. Therefore, it is currently not possible to implement blocking synchronization primitives.

Besides pure synchronization, mutexes and barriers in SMI have to fulfill another very important duty: they have to provide the user with a manageable memory model. All kind of read and write buffers, as they are used within the processor cache and the SCI adapters, raise data consistency problems. SMI ensures coherency at least at synchronization points. This is done by invalidation of the SCI-adapter read buffers at lock-, successful trylock- and barrier-operations at the processor that performs one of these operations. At unlock- and barrier-operations, the invoking processors' write buffers and those on the SCI adapter are flushed to memory. The resulting memory model is release consistency.

3.1.4 Dynamic Memory Allocation

A shared memory region can either be used as a flat piece of memory, e.g. to store a single array, or for dynamic memory allocation by all processes. This allows to cooperatively and dynamically assemble shared data structures. For a region that shall be used in this way, a

memory manager has to be installed. This can be done for a region with the identifier `id` by calling the function `SMI_Init_shregMMU(int id)`. This function initializes all data structures that keep track of the already allocated and the still free portions of memory and creates all necessary locks to guarantee atomicity of a memory allocation request in this parallel execution environment. The necessary data-structures are allocated within the region itself. The accessibility of these structures for all processes is necessary to allow all process to allocate memory therein. Memory can thereafter be allocated/freed by all processes calling the functions

```
SMI_{I,C}malloc(int size, int region_id, void** address)
```

and

```
SMI_{I,C}free(void* address)
```

While an `I`-type function is called from a separate process, returning the address of the allocated piece of memory just to the calling process, the `C`-type function is a collective function that must be called by all processes and therefore states a global synchronization point. The `SMI_Cmalloc` function also allocates just a single piece of memory, but afterwards, all processes are aware of it's address and can directly access it.

3.2 Application Parallelization

Employing SMI, shared data parallelization of a quite diverse set of application codes is in progress. Some first experiences have already been obtained in parallelizing a matrix-vector multiplication kernel for large sparse irregular matrices [9]. Two other just starting projects deal with the parallelization of a room-acoustic simulation code and a performance-critical module of an airline flight schedule code.

The parallelization effort that has advanced most far till this point in time concerns the molecular dynamics simulation code GROMOS 96 [20]. In a time-step fashion, trajectories of atoms of a molecular ensemble (e.g. a protein in a solvent) are computed. The dynamic is determined by the forces, acting between pairs of atoms. The evaluation of all the forces, effecting each atom, that cannot be neglected in magnitude, i.e. all those between atoms that are geometrically adjacent within a specific cut-off radius, is a very time consuming algorithm. It is based on a sparse irregular grid, the so-called pair-list of atoms, that has to be processed. This module, together with the module that generates the pair-list have been parallelized. Table 2 shows the first results for a minimum cluster configuration.

4 SVMlib

4.1 Overview

SVMlib (*Shared Virtual Memory Library*) is an all-software, page based, user level shared virtual memory [3] subsystem for clusters of Windows NT workstations. The library has been designed to benefit from several Windows NT features like preemptive multithreading and support for SMP machines. Unlike most software DSM systems, SVMlib itself is truly multithreaded. It also allows to create several preemptive user threads to speed up the computation on SMP nodes in the cluster. Currently the library uses TCP/IP sockets for communication pur-

		Thrombin N=3,078	Thrombin in water N=19,359	Water (medium) N=5,184	Water (large) N=41,472
original code (sequentiel)		48.4	101.9	189.3	305.0
parallel code	1 proc.	44.0	95.1	170.5	259.2
	2 procs. on 1 SMP node	31.2	54.4	101.1	154.2
	2 procs on 2 SCI-connected nodes	37.0	64.2	122.5	165.1

Table 2: Run times (in seconds) for different benchmark problem (N states the total number of atoms of the molecular ensemble) of the original and the parallelized GROMOS 96 code on the PentiumPro cluster.

poses but it will also support efficient message passing using Dolphins implementation of SCI.

SVMlib provides a C/C++ API that allows the user to create and destroy regions of virtual shared memory that can be accessed fully transparently. Also different synchronization primitives like barriers and mutexes are part of the API. To keep track of accesses to the shared regions, SVMlib handles page faults within the regions via structured exception handling provided by the C++ run time system of Windows NT.

At the current stage, two different memory consistency models are supported by three different consistency protocols. The first consistency model offers the widely used though fairly inefficient *sequential consistency* [16] model. This model is supported by single writer as well as multiple writer protocols. Secondly, the distributed lock based *scope consistency* [14] is implemented.

Our main goal in this project is to examine the impact of efficient distributed synchronization protocols to the performance of a SVM system.

4.2 Memory Consistency Models

Uniprocessor systems present a simple model of the memory to the programmer. A read operation always returns the „last“ value written to a given memory location. Likewise a write operation changes the value at the given location. „Subsequent“ reads at this location will return the „last“ written value until the „next“ write operation occurs.

When multiple processors are involved, the memory model becomes more complex because the definitions of „last value written“, „subsequent read“ and „next write“ become unclear. Hence, several consistency models have been proposed [5,6,10,14,15] that place specific requirements on the order that shared memory accesses from one processor are observed by other processors. In other words, memory consistency models define which orderings are legal

when accessing a common set of locations.

The *sequential consistency* model [16] requires the execution of a parallel program to appear as some interleaving of the execution of the parallel process on a sequential machine. Software implementations of this consistency model are straightforward but fairly inefficient and provide a bad scalability. The problem is an effect called *false sharing*. False sharing occurs when different processors try to access different memory locations that are located in the same memory page. Because the SVM system maintains coherence on page granularity this kind of access pattern causes the page to be moved from one processor to the other even if the accessed data is logically independent. Obviously this can cause a great performance leak.

To avoid these undesired effects so called *weak consistency* models have been developed. The basic idea behind these models is that distributed synchronization is needed to make parallel programs deterministic. Race conditions in parallel programs result in an unpredictable behavior even if sequential consistency is used. So weak consistency models combine synchronization events with memory updates.

One of these weak models we use within SVMlib is *scope consistency* [14]. In scope consistency, each lock forms a synchronization scope of its own and all changes made within the scope of a lock are associated with this lock. So when a process acquires a lock (or opens a scope) it will be informed which memory locations have been changed by other processes in earlier incarnations of this scope. It will *not* be informed about changes made outside this scope.

4.3 Design of SVMlib

When designing a SVM system, several design choices have to be made. Our primary goal when we started this project was to develop a highly flexible and extendable research instrument. We therefore decided to build SVMlib as a set of independent modules where each can be exchanged without influencing the other modules.

Another important choice was the platform to build SVMlib on. As Windows NT is a modern operating system with some interesting features like true preemptive kernel threads, SMP support and a rich API we decided to use workstations running Windows NT as primary platform. However, as UNIX workstations also play an important role, a compatibility layer (*nt2unix*) implementing a subset of the Win32 API was developed for Sun Solaris(tm). Figure 4.1 shows the overall design of SVMlib. On the top level four modules are used.

The first is the *memory manager* that handles the creation and destruction of shared memory regions, catches page faults and implements the memory dependent part of the user interface. The memory manager manages a set of regions where each region can use a different consistency model and coherence protocol.

The second part is the *lock manager* that provides an interface that allows to create and destroy primitives for distributed process synchronization - mutexes as well as global barriers and semaphores.

For internode communication purposes the *communicator* is used. The user will never directly use this module. It is for internal purposes only. The communicator provides a simple interface

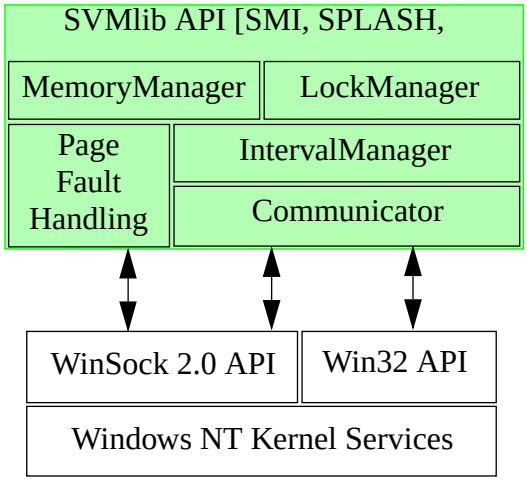

Figure 4.1: SVMlib components.

containing a barrier, a broadcast algorithm and the possibility to send messages to each other node. This module has been designed to be active itself. To take advantage of the SMP support of Windows NT the communicator uses threads to handle incoming messages.

The last main module is the *interval manager* that allows to implement weak consistency models like lazy release consistency [15] or the currently used scope consistency [14]. The user will never have to access this module directly. It is used as a bridge between the memory and the lock manager when weak consistency models are used. This is needed because both locks and memory pages handle a part of the weak consistency model.

SVMlib provides several API personalities to the application programmer. First of all, a native C and C++ API is provided. For compatibility to other SVM systems and existing shared memory implementations, other interfaces to shared memory programming are supported. Currently, these interfaces include the *Shared Memory Interface* (SMI) [8], the macro interface of *Stanford Parallel Applications for Shared Memory* (SPLASH) [22] and the *Coherent Virtual Machine* (CVM) [19]. Other interfaces are planned to be supported in the future.

On the operating system side, we use a hybrid approach to support the native Windows NT environment as primary platform and the UNIX world (i.e. Sun Solaris SPARC/x86) as secondary platform. As mentioned above, to achieve this goal, we developed a small emulation layer called *nt2unix* to support all Win32 API calls we use under NT on the UNIX system.

4.4 Implementation Details

SVMlib is implemented as a static or dynamic library. It is fully implemented in C++ and makes heavy usage of the polymorphism provided by this language. That results in a good extensibility. It is obvious that the usage of virtual methods affects the performance. But we think that the advantages predominate the disadvantages.

4.4.1 Memory management and page fault handling

As described in the previous paragraph, the memory manager is responsible for the memory

management. It handles a list of memory regions represented by C++ classes. Regions are identified by unique identifiers chosen by the user. This approach makes it possible to create a region only in a subset of the processes involved in the computation.

To keep track of the memory accesses to the shared virtual memory, SVMlib uses the memory protection facilities of Windows NT on a per page basis. Internally, each virtual shared page is represented by a class derived from a generic class called `CPage`. These classes contain code to handle the consistency protocol. This allows to add new protocols just by deriving a new class from `CPage`. Neither the memory manager nor the regions have to be changed to add a new protocols.

Page faults are handled using a unique Windows NT mechanism called structured exception handling (SEH). To catch a page fault, the library implements a `main()` function that just calls the user main function `SVMmain`. The call of `SVMmain` is protected by a `__try-__except()` block. The code fragment looks like this:

```
__try {
        ret = SVMmain(argc, argv);
} __except(MemoryExceptionFilter(GetExceptionInformation())){};
```

So whenever an exception occurs the function `MemoryExceptionFilter()` is called. This routine just checks if it was a memory exception and calls a method of the memory manager that searches the page the exception occurred in and calls its handler. When the call returns, the page is accessible and the program can continue:

```
// This function actually handles memory exceptions.
LONG MemoryExceptionFilter(_EXCEPTION_POINTERS *ExceptionInfo) {
  // get detailed exception information
  EXCEPTION_RECORD *ExRec=ExceptionInfo->ExceptionRecord;
  // Is it really a memory access violation ?
  if(ExRec->ExceptionCode!=EXCEPTION_ACCESS_VIOLATION) {
    // No -> try next handler for this exception.
    return EXCEPTION_CONTINUE_SEARCH;
  }
  // Yes -> it is really a page fault.
  DBG((ExRec->ExceptionInformation[0]?"Write":"Read")<<
      "-violation at"<<(void*)ExRec->ExceptionInformation[1]);
  // Is it a page fault in one of our regions ?
  if(MManager.HandlePageFault(ExRec->ExceptionInformation[0],
          (LPVOID)ExRec->ExceptionInformation[1]))
    // Yes -> continue execution with modified access rights.
    return EXCEPTION_CONTINUE_EXECUTION;
  else {
    // No -> a page fault somewhere else.
    SVM_ERROR((ExRec->ExceptionInformation[0]?"Write":"Read")<<
      "-violation at "<<(void*)ExRec->ExceptionInformation[1]);
    return EXCEPTION_CONTINUE_SEARCH;
  }
}
```

4.4.2 Synchronization management

The second main part of the library is the synchronization management. Each synchronization primitive consists of two parts; the user interface and the message handlers. The interface is used by the user to acquire a lock, the handlers are used by the lock manager to handle incoming messages.

Currently two different mutexes, a new distributed reader/writer lock algorithm (see [18] for details) and a global barrier are implemented. Because the main goal of our project is the examination of different synchronization primitives this part of the library will grow heavily in the future.

4.4.3 Communication

The communication subsystem provides basic message passing facilities. As all other parts of the library it is implemented in a C++ class. To make it easy to change the communication platform an abstract base class is used. Currently Berkeley TCP/IP sockets from the WinSock32 library of Windows NT are used. In the future we will use the SCI interconnect installed in our PC cluster to exchange messages. For efficiency reasons the communicator uses multiple threads to handle incoming messages. Each message has a special header containing the size of the message and an identifier describing the recipient of the message, e.g. the memory manager or the lock manager. The communicator dispatches the message to the appropriate part of the library where the message is handled. The use of multiple threads makes it possible to take full advantage of SMP systems. If multiple processors are available there is no need to interrupt the user thread when a message arrives. This can help to hide communication latency.

4.5 First performance measurements

As the implementation of SVMlib still is under development (especially for the weaker consistency models), we can only give some early metrics used to characterize the performance of SVMlib:

- *Page Fault Detection Time.* This value includes the mean time from the occurrence of a processor page fault on a protected page to the entrance of the handling routine. That is, this time includes all operating system overhead to deliver a page fault exception to user code. Note that there seems to be no difference between the NT Server and NT Workstation version with respect to exception handling. We compared these values with user level page fault detection under Solaris 2.5.1 for Intel and SPARC, respectively. Under UNIX, the memory exception handling mechanism of Windows NT is emulated by catching the `SIG-`

	SuperSPARC, 50 MHz	**Pentium, 133 MHz**	**Pentium Pro, 200 MHz**
Windows NT 4.0 Server / Workstation	-	28 µs	19 µs
Solaris 2.5.1	135 µs	92 µs	48 µs

SEGV signal.

- *Page Fault Time*. This value includes the mean time to handle one page fault. This time excludes the page fault detection time mentioned above. It includes the overhead due to the coherence protocol and communication subsystem. In the current implementation, the times measured are mainly influenced by the high TCP/IP latency. The measurements were made using the FFT application of the set of CVM examples [19]. This application implements a Fast Fourier Transformation on a 64 x 64 x 16 array. The coherence protocol used

#Nodes	Read / Write / Average Fault Time [ms] (CVM on Solaris)	Read / Write / Average Fault Time [ms] (SVMlib on Solaris)	Read / Write / Average Fault Time [ms] (SVMlib on Win32)
2	11.3 / 0.8 / 4.4	4.5 / 1.3 / 2.2	3.4 / 1.1 / 1.8
3	12.0 / 0.8 / 5.8	4.6 / 1.8 / 2.7	3.4 / 1.4 / 2.3
4	16.7 / 0.9 / 7.1	4.9 / 1.8 / 3.1	4.0 / 1.5 / 2.4

  is a multiple reader / single writer protocol implementing sequential consistency. We compared three configurations running FFT: (1) *CVM on Solaris*: the CVM [19] system running on Solaris 2.5.1, Sun SS-20, Ethernet; (2) *SVMlib on Solaris*: the Solaris version of SVMlib, running on the same platform as (1); (3) *SVMlib on Win32*: the Win32 version of SVMlib, running on Windows NT 4.0, Intel Pentium-133, FastEthernet. Naturally, the Win32 time values mainly reflect the improved network performance of FastEthernet.

5 Summary and Conclusion

Exploiting the capabilities of an SCI-interconnected PC cluster for shared data parallelization turned out to be difficult on the one hand as well as promising on the other hand. It is difficult, because several peculiarities of the hardware and device drivers became obvious: read and write buffers on the SCI adapter as well as those of the processors make it difficult to provide a manageable memory model to the user and to correctly implement synchronization mechanisms. Allocation of large shared regions is not allowed due to limitations of the address translation of the SCI adapters. Such regions have to be constructed out of several small one. Furthermore, mapping of shared regions to the same address in each process made modifications in the Windows NT SCI device driver necessary. But this is essential to be able to exchange pointers. The SMI programming interface evolved in a way that all those problems are hidden from the user but provides him with a comfortable basis for application parallelization. The pleasant experience is that first applications have successfully been parallelized. Although there has to be gained a lot more of experience with application parallelization, the off-the-shelf cluster platform, although not being an entire cache coherent DSM system, together with a suitable programming interface turned out to be a well suited platform also for shared data parallelization.

In a second project overview, we introduced SVMlib, a library introducing the distributed shared memory programming abstraction to clustered Windows NT workstations by a software-only approach. It is one of the first published SVM systems running on Windows NT.

First performance results running one of the SPLASH applications with sequential consistency were presented.

The project is still in its start-up phase. As SVMlib is currently being implemented, several extensions to the system are being developed:

- *Parallel Shared Virtual Memory Allocation.* While SVMlib supports creating and destroying regions of consecutive shared virtual memory pages, an extension for allocation of shared memory *within* the regions is developed. This feature is especially useful to emulate virtually shared data heaps;
- *Performance Evaluation and Visualization*. To help the programmer to detect false sharing and synchronization bottlenecks, a tool instrumenting the synchronization behaviour of SVMlib applications is developed;
- *SCI/SVMlib Integration*. The obvious step to avoid the high communication latency of TCP/IP is to employ the SCI network of our cluster as low latency network. Another direction we are following is to use SCI and SVMlib to build a *hybrid* DSM system, where coherence protocols are implemented by cache line invalidations, not by explicit messaging.
- *Cross-platform SVMlib.* In the current stage, SVMlib runs on Windows NT as primary and Solaris as secondary platform. We are trying to use both versions concurrently, e.g. to use SVMlib to network NT and UNIX workstations to a virtually shared machine.

References

[1] Ahamad, M., Bazzi, R. A., John, R., Kohli, P., and Neiger, G. *The Power of Processor Consistency (Extended Abstract)*. In Proc. of the 5th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA‘93), pages 251-260, June 1993.

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

[3] Berrendorf, R.; Gerndt, M.; Mairandres, M.; Zeisset, S.: *A Programming Environment for Shared Virtual Memory on the Intel Paragon Supercomputer*, ISUG Conference, Albuquerque, 1995

[4] Bershad, B. N.; Zekauskas, M. J.: *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, Sept. 1991.

[5] Boden, N. J.; Cohen, D.; Felderman, R. E.; Kulawik, A. E.; Seitz, C. L.; Seizovic, J. N.; Su, W.-K.: *Myrinet - A Gigabit-per-Second Local-Area Network.* IEEE Micro, Febr. 1995.

[6] Carter, J. B.: *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency.* PhD thesis, Department of Computer Science, Rice University, September 1993.

[7] Dolphin Interconnect Solutions: *PCI-SCI Cluster Adapter Specification.* Jan. 1996.

[8] Dormanns, M.; Sprangers, W.; Ertl, H.; Bemmerl, T.: *A Programming Interface for NUMA Shared-Memory Clusters.* Proc. High Perf. Comp. and Networking (HPCN), pp. 698-707, LNCS 1225, Springer, 1997.

[9] Dormanns, M.; Sprangers, W.; Ertl, H.; Bemmerl, T.: *Performance Potential of an SCI Workstation Cluster for Grid-Based Scientific Codes.* Proc. High Perf. Computing (HPC), pp. 226-231,

1997.

[10] Fu, S. S. and Tzeng, N.-F.: *Aggressive Release Consistency for Software Distributed Shared Memory.* In Proc. of the 17th Int. Conf. on Distributed Computing Systems (ICDCS‘97), May 1997.

[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] Hellwagner, H.; Karl, W.; Leberecht, M.: *Enabling a PC Cluster for High Performance Computing.* Speedup Journal, Vol. 11, No. 1, 1997.

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

[14] Iftode, L.; Singh, J. P.; Li, K.: *Scope Consistency: A Bridge between Release Consistency and Entry Consistency.* In Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA‘96), June 1996

[15] Keleher, P.; Cox, A. L.; Zwaenepoel, W.: *Lazy Release Consistency for Software Distributed Shared Memory.* In Proc. of the 19th Annual, Int. Symp. on Computer Architecture (ISCA‘92), pp 13-21, May 1992

[16] Lamport, L.: *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Transactions on Computers, C-28(9), pp. 690-691, September 1979

[17] McCalpin, J. D.: *A Survey of Memory Bandwidth in Current High Performance Computers.* IEEE TCCA Newsletter, Dec. 1995.

[18] Paas, S. M.; Scholtyssik, K.: *Efficient Distributed Synchronization within an all-software DSM system for clustered PCs.* 1st Workshop Cluster-Computing, TU Chemnitz-Zwickau, November 6-7, 1997

[19] Thitikamol, K.; Keleher, P.: *Multi-Threading and Remote Latency in Software DSMs*. In: 17th International Conference on Distributed Computing Systems, May 1997

[20] van Gunsteren, W. F. et. al.: *Biomolecular Simulation: The GROMOS 96 Manual and User Guide.* vdf Hochschulverlag AG an der ETH Zürich and BIOMOS b.v., Zürich, Groningen, 1996.

[21] Warren, M. S.; Becker, D. J.; Goda, M. P.; Salmon, J. K.; Sterling, T.: *Parallel Supercomputing with Commodity Components.* Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp. 1372-81, 1997.

[22] Woo, S. C.; Moriyoshi Ohara, M.; Torrie, E.; Singh, J. P., and Gupta, A.: *The SPLASH-2 Programs: Characterization and Methodological Considerations.* In Proc. of the 22nd International Symposium on Computer Architecture, pp. 24-36, Santa Margherita Ligure, Italy, June 1995