THROOM — Supporting POSIX Multithreaded Binaries on a Cluster

Henrik Löf, Zoran Radović and Erik Hagersten

Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
henrik.lof@it.uu.se

Abstract. Today, most software distributed shared memory systems (SW-DSMs) lack industry standard programming interfaces which limit their applicability to a small set of shared-memory applications. In order to gain general acceptance, SW-DSMs should support the same look-and-feel of shared memory as hardware DSMs. This paper presents a runtime system concept that enables unmodified POSIX (Pthreads) binaries to run transparently on clustered hardware. The key idea is to extend the single process model of multi-threading to a multi-process model where threads are distributed to processes executing in remote nodes. The distributed threads execute in a global shared address space made coherent by a fine-grain SW-DSM layer. We also present THROOM, a proof-of-concept implementation that runs unmodified Pthread binaries on a virtual cluster modeled as standard UNIX processes. THROOM runs on top of the DSZOOM fine-grain SW-DSM system with limited OS support.

1 Introduction

Clusters built from high-volume compute nodes, such as workstations, PCs, and small symmetric multiprocessors (SMPs), provide powerful platforms for executing large-scale parallel applications. Software distributed shared memory (SW-DSM) systems can create the illusion of a single shared memory across the entire cluster using a software run-time layer, attached between the application and the hardware. In spite of several successful implementation efforts [1], [2], [3], [4], [5], SW-DSM systems are still not widely used today. In most cases, this is due to the relatively poor and unpredictable performance demonstrated by the SW-DSM implementations. However, some recent SW-DSM systems have shown that this performance gap can be narrowed by removing the asynchronous protocol overhead [5], [6], and demonstrate a performance overhead of only 30-40 percent in comparison to hardware DSMs (HW-DSM) [5]. One obstacle for SW-DSMs is the fact that they often require special constructs and/or impose special programming restrictions in order to operate properly. Some SW-DSM systems further alienate themselves from HW-DSMs by relying heavily on very weak memory models in order to hide some of the false sharing created by their page-based coherence strategies. This often leads to large performance variations when comparing the performance of the same applications run on HW-DSMs.

We believe that, SW-DSMs should support the same look-and-feel of shared memory as the HW-DSMs. This includes support for POSIX [7] threads running on some standard memory model and a performance footprint similar to that of HW-DSMs, i.e., the performance gap should remain approximately the same for most applications. Our goal is that binaries that run on HW-DSMs could be run on SW-DSMs, without modifications.

In contrast to a HW-DSM system, where the whole address space of all processes are kept coherent by hardware, most SW-DSMs only keep coherence for specified segments in the user-level part of the virtual address space. This segment, which we call G_MEM, is mapped shared across the DSM nodes using the interconnect hardware. Furthermore, the text (program code), data and stack segments of the UNIX process abstraction are private to the parent process and its children on each node of the cluster. This creates a SW-DSM programming model where special constructs are needed to separate shared data, which must be allocated in G_MEM, from private data, which is allocated in the data and stack segments of the UNIX process at program loading. This is often done by creating a separate heap space in G_MEM with an associated primitive for doing allocation. In a standard multi-threaded world, there exist only one process and one address space which is shared among all threads. There is no distinction between shared and private data. Consider the following example: An application allocates a shared global array for its threads to operate on. This is often done by a single thread in an initialization phase. In a typical SW-DSM system such as TreadMarks [3], a special `malloc()`-type call has to be implemented to allocate the memory for the shared array inside the G_MEM. Also, the pointer variable holding the address, which is allocated in the static data segment of the process, has to be propagated to all remote nodes. This is often done by introducing a special propagation primitive.

In this paper, we present THROOM which is a runtime system concept that creates the illusion of a single process shared memory abstraction on a cluster. In essence, we want to make the static data and heap segments globally accessible by threads executing in remote nodes without introducing special DSM constructs in the application code. In the light of the example above, the application should use a standard `malloc()` call and the pointer variable should be replicated automatically. The rest of this paper is organized as follows: First, the THROOM concept is presented. Second, we give a brief presentation of the SW-DSM used. We also specify the requirements of THROOM on the SW-DSM. Third, we present a proof-of-concept implementation of THROOM on a single system image cluster and finally we discuss the performance of this implementation as well as the steps needed to take THROOM to a real cluster.

2 The THROOM concept

In many implementations of SW-DSMs, the different nodes of the cluster all run some daemon process to maintain the G_MEM mappings and to deal with requests for coherency actions. In this paper we use the term *user node* to refer

to the cluster node in which the user executes the binary (the *user process*). All other nodes are called *remote nodes* and their daemon processes will be called *shadow processes*. In execution, THROOM consists of one process per node of the cluster.

The fundamental idea of THROOM is to distribute threads from the user process to shadow processes executing on remote nodes running different instances of a standard UNIX OS kernel. As discussed earlier, such systems exists, but they require non-standard programming models. To support a standard model such as POSIX, it is required that the whole address space of the user process can be accessed by all of the distributed threads. To accomplish this, we can simply place the text, data and stack segments inside a G_MEM-type segment made coherent by a SW-DSM. This will create the illusion of a large scale shared memory multiprocessor built out of standard software and hardware components.

3 DSZOOM - a Fine-Grained SW-DSM

Our prototype implementation is based on the sequentially consistent DSZOOM SW-DSM [5]. Each DSZOOM node can either be a uniprocessor, a SMP, or a CC-NUMA cluster. The node's hardware keeps coherence among its caches and its memory. The different cluster nodes run different kernel instances and do not share memory with each other in a hardware-coherent way. DSZOOM assumes a cluster interconnect with an inexpensive user-level mechanism to access memory in other nodes, similar to the remote `put/get` semantics found in the cluster version of the Scalable Coherent Interface (SCI), or the emerging InfiniBand standard that supports `RDMA READ/WRITE` as well as the atomic operations `CmpSwap` and `FetchAdd` [8].[1] Another example is the Sun Fire (TM) Link interconnect hardware [9].

While traditional page-based SW-DSMs rely on TLB traps to detect coherence "violations", fine-grained SW-DSMs like Shasta [10], Blizzard-S [11], Sirocco-S [12] and DSZOOM [5] insert the coherence checks in-line. In DSZOOM, this is done by replacing each load and store that may reference shared data of the binary with a code *snippet* (short sequence of machine code). In terms of THROOM, the only requirement on the SW-DSM system is that it uses binary instrumentation. THROOM will also inherit the memory consistency model of the SW-DSM system.

4 Implementing THROOM

This section discuss how we can implement the THROOM concept using standard software components and the DSZOOM SW-DSM.

[1] Atomic operations are needed to support a *blocking directory* protocol [5].

4.1 Achieving transparency

The most important aspect of THROOM is that it is totally transparent to the application code, no recompilation is allowed. To achieve this, we use a technique called *library interposition* or *library pre-loading* [13], which allow us to change the default behavior of a shared library call without recompiling the binary. Many operating systems implement the core system libraries such as `libc`, `libpthread` and `libm` as shared libraries. Using inter-positioning, we can catch a call, to any shared library and redirect it to our own implementations. In practice, this is done by redefining a symbol in a separate shared library to be pre-loaded at runtime. When an application calls the function represented by the symbol, the runtime linker searches its path for a match. Pre-loading simply means that we can insert an alternate implementation before the standard implementation in the search path of the linker. Pre-loading also allow us to reuse the native implementation. Original arguments can be modified in the interposer before the call to the native implementation is made.[2]

4.2 Distributing Threads

To distribute threads, the `pthread_create()` call is redefined in a pre-loaded library. The interposed implementation, first schedules the thread for execution in a remote shadow process. Second, the chosen shadow process is told to create a new thread, by calling the native `pthread_create()` from within the interposing library. The new distributed thread will start to execute in the shadow process, with arguments pointing to the software context of its original user process.

4.3 Creating a Global Shared Address Space

A minimal requirement for a distributed thread to execute correctly in a shadow process is that it must share the whole address space of the user process. To accomplish this, the `malloc()` call is redefined in a pre-loaded library to allocate memory from G_MEM instead of the original data segment of the user process. This will make all dynamically allocated data accessible from the shadow processes. Code and static data are made globally accessible by copying the segments containing code and static data from the user process to the G_MEM. The application code is then modified, using binary instrumentation, to access the G_MEM copy instead of the original segments. This will make the application execute entirely in the global shared memory segment. Hence, no special programming constructs are needed to propagate writes to static data. The whole process is also transparent in the sense that a user does not need access to the application source code, as binary instrumentation modifies the binary itself.

All references to the G_MEM must also be made coherent as the hardware only support remote reads and writes. This is taken care of by a fine-grain SW-DSM. If the SW-DSM use binary instrumentation to insert snippets for access

[2] To our knowledge, Linux, Solaris, HP-UX, IRIX and Tru64 all support library pre-loading.

control, we can simply add instructions needed for the static data access diversion to these snippets. In all cases, a maximum of four instructions were added to the existing snippets of DSZOOM. To lower the overheads associated with binary instrumentation, the present implementation does not instrument accesses to the stack. Hence stacks are considered thread private. Although this is not in full compliance with the POSIX model of multi-threading, it is sufficient to support a large set of pthread applications.

4.4 Cluster-Enabled Library Calls

Most applications use system calls and/or calls to standard shared libraries such as `libc`. If the arguments refer to static data, the accesses must be modified to use the G_MEM in order for memory operations to be coherent across the cluster. This can be done in at least two ways. We either instrument all library code or we overload the library calls to copy any changes from the user process original data segments to the G_MEM copies at each library call. Remember that un-instrumented code referencing static data of the application will operate in the original data segments of the user process. Hence, copying is needed to make any modifications visible to other nodes.

Instrumenting all library code is in principle, the best way to cluster-enable library calls. However, our instrumentation tool, EEL [14], was not able to instrument all of the libraries. Instead, we had to use the library interposition method for our prototype implementation. An obvious disadvantage of this method is that we have to redefine a large amount of library calls, especially if we want complete POSIX support. Another disadvantage is the runtime overhead associated with data copying, especially for I/O operations. A better solution would be to generate the coherence actions on the original arguments before the call is made in the application binary, see Scales et. al. [1]. This requires a very sophisticated instrumentation tool, which is outside the scope of this work.

5 Implementation Details

We have implemented the THROOM system on a 2-node Sun WildFire prototype SMP cluster [15], [16]. The cluster is running a single-system image version of Solaris 2.6 and the hardware is configured as a standard CC-NUMA architecture. Although, this system already supports a global shared address space, we can still use it to emulate a future THROOM architecture.

The runtime system is implemented as a shared library. A user simply sets the `LD_PRELOAD` environment variable to the path of the THROOM runtime library, and then executes the instrumented binary. As the system is a single system image we can use standard Inter Process Communication (IPC) primitives to emulate a real distributed cluster. The DSZOOM address space is set up during initialization using the `.init` section. This makes the whole initialization transparent. Control is then given to the application. The user process issues

a `fork(2)` call to create a shadow process, which will inherit its parents mappings by the copy-on-write semantics of Solaris. The two processes are bound to the two nodes using the WildFire first-touch memory initialization and the `pset_bind()` call. The home process then reads its own `/proc` file system to locate the `.text`, `.data`, and `.bss` segments and copies them to the G_MEM.

The shadow process waits on a process shared POSIX conditional variable to create remote threads for execution in the G_MEM. Parameters are passed through a shared memory mapping separated from the G_MEM. Since the remote thread is created in another process, thread IDs are no longer unique. To fix this, the remote node ID is copied into the most significant eight bits of the thread type, which in the Solaris 2.6 implementation is an unsigned integer. Similar techniques are used for other pthread calls. Also, the synchronization primitives of the application were overloaded using pre-loading to pre-prepared `PROCESS_SHARED` POSIX primitives to allow for multi-process synchronization. More details on the implementation are available in Löf et al. [17].

6 Performance Study

First a set a test pthread programs were run to verify the correctness of the implementation. To produce a set of pthread programs to be used as a comparison to DSZOOM, ten SPLASH-2 applications [18] were compiled using the GCC v2.95.2 compiler without optimization (-O0)[3] and a standard Pthread PARMACS macro implementation (c.m4.pthreads.condvar_barrier) was employed. No modifications was made to the PARMACS run-time system or the applications. To exclude the initialization time for the THROOM runtime system, timings are started at the beginning of the parallel phase. All timings have been performed on the 2-node Sun WildFire [15] configured as a traditional CC-NUMA architecture. Each node has 16 UltraSPARCII processors running at 250 MHz. The access time to node-local memory is about 330ns. Remote memory is accessed in about 1800ns. In Table 1, we see that more instructions are replaced in the case of THROOM since *all* references to static data have to be instrumented. This large difference in replacement ratio compared to DSZOOM is explained by the fact that DSZOOM can exploit the PARMACS programming model and use *program slicing* to remove accesses to static data that are not shared. Figure 1 and 2 shows execution times in seconds for 8- and 16-processor runs for the following THROOM configurations:

THROOM_RR THROOM runtime system using library pre-loading. Round-robin scheduling of threads between the two nodes. All references to static data are instrumented.

DSZOOM Used as reference. Aggressive slicing and snippet optimizations. Optimized for a two-node fork-exec native PARMACS environment, see [5].

[3] The code is compiled without optimization to eliminate any delay slots, which EEL cannot handle correctly.

Program	Problem size, Iterations	Replaced Loads (%)	Replaced Stores (%)
FFT	1 048 576 points (48.1 MByte)	44.6(19.0)	32.8(16.5)
LU-C	1024x1024, block 16 (8.0 MByte)	48.3(15.5)	23.0(9.4)
LU-NC	1024x1024, block 16 (8.0 MByte)	49.2(16.7)	27.7(11.1)
RADIX	4 194 304 items (36.5 MByte)	54.4(15.6)	31.4(11.6)
Barnes	16 384 bodies (8.1 MByte)	56.6(23.8)	55.4(31.1)
Ocean-C	514x514 (57.5 MByte)	50.6(27.0)	31.2(23.9)
Ocean-NC	258x258 (22.9 MByte)	51.0(11.6)	39.0(28.0)
Radiosity	room (29.4 MByte)	41.1(26.3)	35.1(27.1)
Water-NSQ	2197 mol, 2 steps (2.0 Mbyte)	50.4(13.4)	38.0(16.2)
Water-SQ	2197 mol, 2 steps (1.5 Mbyte)	48.5(15.7)	32.5(13.9)

Table 1. Problem sizes and replacement ratios for the 10 SPLASH-2 applications studied. Instrumented loads and stores are showed as a percentage of the total amount of load or store instructions. The number in parenthesis shows the replacement ratio for the DSZOOM SW-DSM without THROOM.

CC-NUMA Uses the same runtime system as DSZOOM but without any instrumentations. Coherence is kept by the WildFire hardware [5]

A study of Figures 1 and 2 reveals that the present implementation is slower than a state-of-the-art SW-DSM such as DSZOOM. The average runtime overhead compared to DSZOOM for THROOM_RR is 65% on 8 CPUs and 78% on 16 CPUs. In order to put these numbers into the context of total SW overhead compared to a HW-DSM, the average slowdown comparing the CC-NUMA and the DSZOOM cases is only 26%. The most significant contribution to the high overhead when comparing DSZOOM to THROOM is the increased number of instrumentations needed to support the POSIX thread model. Another source of overhead is the inefficient implementation of locks and barriers. This can be observed by comparing the performance of Barnes, Ocean-C, Ocean-NC and Radiosity from Figures 1 and 2. The performance of these four applications drops when increasing the number of threads as they spend a significant amount of time executing in synchronization primitives. The DSZOOM runtime system uses its own implementations of spin-locks and barriers which are more scalable.

7 Related work

To our knowledge, no SW-DSM system has yet been built that enables transparent execution of an unmodified POSIX binary. The Shasta system [1], [2] come closest to our work and this system has showed that it is possible to run an Oracle database system on a cluster using a fine-grain SW-DSM. Shasta has solved the OS functionality issues in a similar way as is done in THROOM although they support a larger set of system calls and process distribution. THROOM differs from Shasta in that it supports sharing of static data. THROOM also supports thread distribution. Shasta motivates the lack of multi-threading sup-

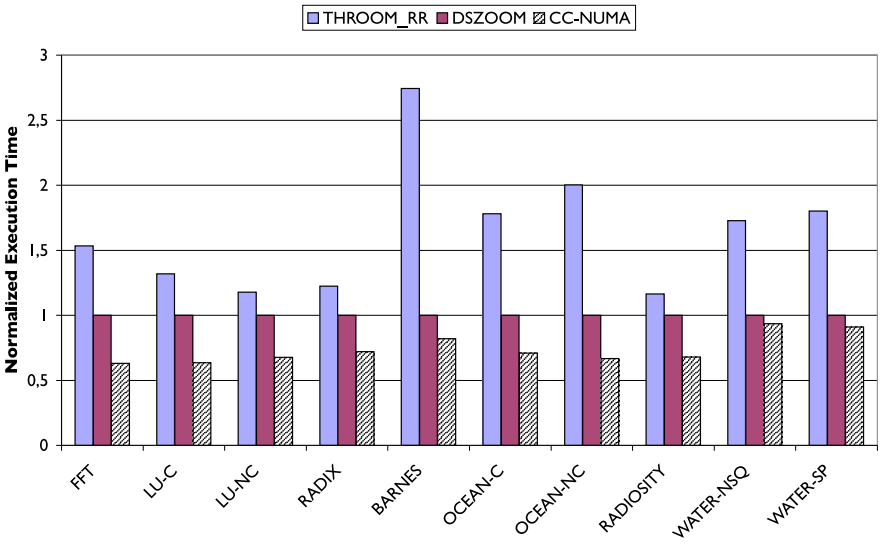

Fig. 1. Runtime performance of the THROOM runtime system. Two nodes with 4 CPUs each.

port by claiming that the overhead associated with access checks lead to lower performance [1].

Another system announced recently is the CableS system [19] built on the GeNIMA page-based DSM [6]. This system support a large set of system calls, but they have not been able to achieve binary transparency. Some source code modifications must be made and the code must be recompiled for the system to operate. Another work related to THROOM is the OpenMP interface to the TreadMarks page-based DSM [20] [3], where a compiler front-end translates the OpenMP pragmas into TreadMark fork-join style primitives. The DSM-Threads system [21] provide a page-based DSM interface similar to the Pthreads standard without binary transparency.

8 Conclusions

We have showed that it is *possible* to extend a single process address space to a multi-process model. Even though the current THROOM implementation relies on some of the WildFire's single system image properties, we are convinced that the THROOM concept can be implemented on a real cluster. In a pure distributed setting, additional issues need to be addressed. One way of initializing the system could be to use a standard MPI runtime system for process creation and handshaking. The address space mappings must also be set up using the RDMA features of the interconnect hardware. Also, synchronization needs to be handled more efficiently (see Radović et. al. [22]), and we need to create more

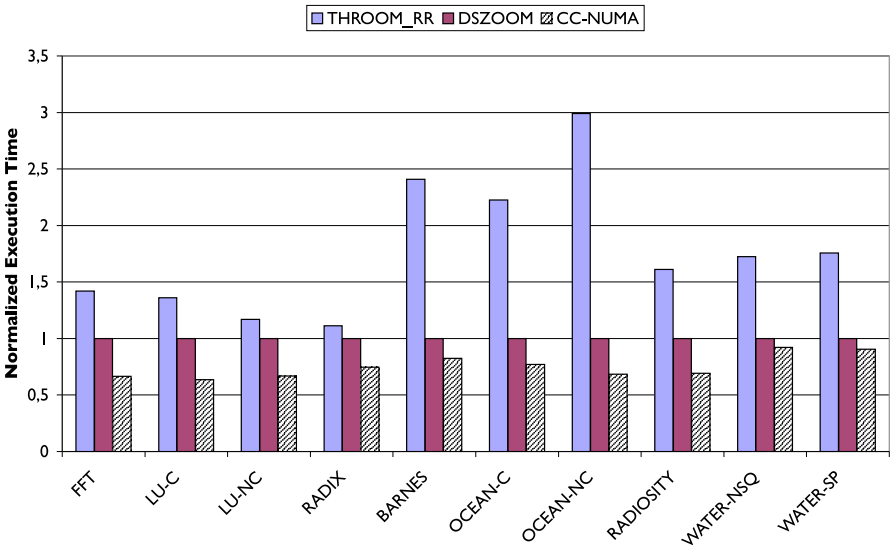

Fig. 2. Runtime performance of the THROOM runtime system. Two nodes with 8 CPUs each.

complete and more efficient support for I/O and other library calls. For complete POSIX compliance, we also need to address the problem of threads sharing data on the stack.

References

1. Scales, D.J., Gharachorloo, K.: Towards Transparent and Efficient Software Distributed Shared Memory. In: Proceedings of the 16th ACM Symposium on Operating System Principles, Saint-Malo, France. (1997)
2. Dwarkadas, S., Gharachorloo, K., Kontothanassis, L., Scales, D.J., Scott, M.L., Stets, R.: Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In: Proceedings of the 5th International Symposium on High-Performance Computer Architecture. (1999) 260–269
3. Keleher, P., Cox, A.L., Dwarkadas, S., Zwaenepoel, W.: TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In: Proceedings of the Winter 1994 USENIX Conference. (1994) 115–131
4. Stets, R., Dwarkadas, S., Hardavellas, N., Hunt, G., Kontothanassis, L., Parthasarathy, S., Scott, M.: Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In: Proceedings of the 16th ACM Symposium on Operating System Principle. (1997)
5. Radović, Z., Hagersten, E.: Removing the Overhead from Software-Based Shared Memory. In: Proceedings of Supercomputing 2001. (2001)
6. Bilas, A., Liao, C., Singh, J.P.: Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In: Proceedings

of the 26th Annual International Symposium on Computer Architecture (ISCA'99). (1999)
7. IEEE Std 1003.1-1996, ISO/IEC 9945-1: Portable Operating System Interface (POSIX)–Part1: System Application Programming Interface (API) [C Language]. (1996)
8. InfiniBand(SM) Trade Association: InfiniBand Architecture Specification, Release 1.0. (2000) Available from: `http://www.infinibandta.org`.
9. Sistare, S., Jackson, C.J.: Ultra-High Performance Communication with MPI and the Sun Fire(TM) Link Interconnect. In: Proceedings of the IEEE/ACM SC2002 Conference. (2002)
10. Scales, D.J., Gharachorloo, K., Thekkath, C.A.: Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII). (1996) 174–185
11. Schoinas, I., Falsafi, B., Lebeck, A.R., Reinhardt, S.K., Larus, J.R., Wood, D.A.: Fine-grain Access Control for Distributed Shared Memory. In: Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI). (1994) 297–306
12. Schoinas, I., Falsafi, B., Hill, M., Larus, J.R., Wood, D.A.: Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In: Proceedings of the 6th International Conference on Parallel Architectures and Compilation Techniques. (1998)
13. Thain, D., Livny, M.: Multiple Bypass, Interposition Agents for Distributed Computing. In: Cluster Computing. (2001)
14. Larus, J.R., Schnarr, E.: EEL: Machine-Independent Executable Editing. In: Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation. (1995) 291–300
15. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantative Approach 3:rd edition. Morgan Kaufman (2003)
16. Hagersten, E., Koster, M.: WildFire: A Scalable Path for SMPs. In: Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture. (1999) 172–181
17. Löf, H., Radović, Z., Hagersten, E.: THROOM — Running POSIX Multithreaded Binaries on a Cluster. Technical Report 2003-026, Department of Information Technology, Uppsala University (2003)
18. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95). (1995) 24–36
19. Jamieson, P., Bilas, A.: CableS: Thread Control and Memory Mangement Extentions for Shared Virtual Memory Clusters. In: 8th International Symposium on High-Performance Computer Architeture, HPCA-8. (2002)
20. A. Scherer, H. Lu, T.G., Zwaenepoel, W.: Transparent Adaptive Parallelism on NOWs using OpenMP. In: Principles Practice of Parallel Programming. (1999)
21. Mueller, F.: Distributed Shared-Memory Threads:DSM-Threads. In: Proc. of the Workshop on Run-Time Systems for Parallel Programming. (1997)
22. Radović, Z., Hagersten, E.: Efficient Synchronization for Nonuniform Communication Architectures. In: Proceedings of Supercomputing 2002. (2002)