Distributed Shared-Memory Threads: *DSM-Threads*
– Description of Work in Progress –

Frank Mueller
Humboldt-Universität zu Berlin, Institut für Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de   *phone: (+49) (30) 20181-276*

Abstract

This paper is, to our knowledge, the first description of a system to support distributed threads on top of POSIX Threads (Pthreads) via distributed virtual shared memory (DSM). The aim of *DSM-Threads* is to provide an easy way for a programmer to migrate from a concurrent programming model with shared memory (Pthreads) to a distributed model with minimal changes of the application code. Thus, a programmer may continue to use the shared-memory algorithms *and* exploit to processing power of a distributed system without dealing with the more complex (and harder to understand) models of distributed algorithms. This paper discusses design goals, design decisions, and implementation choices of DSM-Threads. As the DSM runtime system is itself implemented as a multi-threaded system over Pthreads on each node and copes without compiler or operating system modifications, several problems arise and their solutions are discussed. Several data consistency models are supported to facilitate ports from Pthreads to DSM-Threads and address the requirements of advanced concepts for multiple writers. Heterogeneous systems are supported by handling different data representations at the communication level. Data and thread distribution are realized via a preprocessor based on trace data. The resulting system depends in large parts on POSIX 1003.1c (Threads) compliant operating systems and a set of Gnu tools, which enhances the portability of DSM-Threads.

1 Introduction

In recent years, distributed systems have enjoyed increasing popularity since they provide a cost-effective means to exploit the existing processing power of networks of workstations and a true alternative to shared-memory multi-processors (SMPs). In addition, SMPs do not scale well beyond a relatively small number of processors (typically no more than 40 CPUs) since the system bus becomes a bottleneck. Depending on the communication medium and topology, distributed systems have the potential to scale to a much larger number of processors since the performance may only degrade logarithmically with regard to the number of *contending* nodes for several synchronization algorithms. However, there are several problems associated with the transition from shared-memory systems to distributed systems, which often limit the performance gains.

First, the communication medium between processing nodes (workstations) often poses a bottleneck that is the single largest cause of limitation in terms of performance. New communication media and protocols are promising an increase in throughput by at least an order of a magnitude (*e.g.*, FDDI, FastEther, ATM), which should make a network of workstations more attractive for distributed computing. However, the overhead of communication will always be much larger than the overhead of local computation (*e.g.* local memory access). As a result, distributed applications must restrict communication to the absolute minimum to be competitive.

Another problem is rooted in the absence of a global state within a distributed system. Thus, conventional techniques for concurrent programming cannot be applied anymore. Instead of reusing concurrent algorithms, distributed algorithms have to be developed based on explicit communication (message passing). Simple solutions often employ a centralized approach with a single server that provides a global state to its clients, but the server may create a bottleneck again. Decentralized approaches may have to ensure the causality of the system (*e.g.* via timestamps), incurring additional communication overhead. As a result, distributed algorithms are often hard to understand.

An alternative solution is provided by *distributed virtual shared memory (DSM)* [15, 16]. DSM provides the logical view of a portion of an address space, which is shared between physically distributed processing nodes, *i.e.* a global state. Address references can be distinguished between local memory accesses and DSM accesses, *i.e.* an architecture with non-uniform memory access (NUMA) is created. In addition, the well-understood paradigm of concurrent programming can be used without any changes since the communication between nodes is transparent to the programmer.

In the following, we describe a system to support distributed threads on top of POSIX 1003.1c (Pthreads [18]) via DSM. The aim of *DSM-Threads* is to provide an easy way for a programmer to migrate from a concurrent programming model with shared memory (Pthreads) to a distributed model with minimal changes of the application code. Thus, a programmer may continue to use the shared-memory algorithms *and* exploit to processing power of a distributed system without dealing with the more complex (and harder to understand) models of distributed algorithms.

The paper is structured as follows. Section 2 discusses design goals, gives an overview of the overall design and identifies design issues. Section 3 explains how DSM data can be distinguished from other data. Section 4 presents the communication layer and its support for heterogeneous distributed systems. Section 5 summarizes the data consistency models supported by the system. Section 6 discusses different synchronization methods and their realization in a distributed environment. Sections 7 and 8 highlight data and thread distribution, respectively. Section 9 summarizes system dependencies. Section 10 discusses open problems and future work. Section 11 compares this work with related work. Finally, section 12 provides the conclusions for this paper.

2 Overall Design

The design of the system was driven by a number of design goals, which limited the options for design decisions but, at the same time, provided the base for a (hopefully) wide acceptance of the system. The major design goals are summarized as follows:

- portability;
- interface close to Pthreads (syntactically and semantically);
- minimal modifications to source programs when migrating from Pthreads to DSM-Threads (toward "backwards compatibility");
- options for additional interfaces to fine-tune applications (for expert users);
- least possible communication overhead (since communication may be slow);
- heterogeneous support;
- independent subsystems (e.g. communication, simple interface, advanced interface);
- extensible interface (for future enhancements);
- decentralized approaches (to avoid bottlenecks).

The concrete design of a DSM environment involves a multitude of issues, most of which are dependent on one another to some extent. Before discussing each of these issues, we shall give an overview of the major design issues that have to be addressed:

- DSM data identification;
- communication;
- data consistency and access protocols;
- distributed synchronization methods;
- data distribution;
- thread distribution.

Other issues, such as tracing or heterogeneous system support, shall be discussed in the context of one of the above items. Before we address each of these issues in more depth in this paper, the overall design of the system shall be explained.

On each node that is utilized for distributed thread execution in the system, a process is created that contains DSM-system threads, a Pthreads systems (currently FSU Pthreads [13]) and user threads (see Figure 1). The DSM runtime system consists of a communication server, which is a separate thread of control with the highest priority of all threads within this process. The task of the communication server is to open a communication channel and wait for incoming messages. When a message is received, the protocol header is decoded to determine the message type. Depending on the type, a message may be acted upon by the communication server if little work has to be done (*e.g.*, for forwarding page requests), or the processing of the message may be delegated to a worker thread.

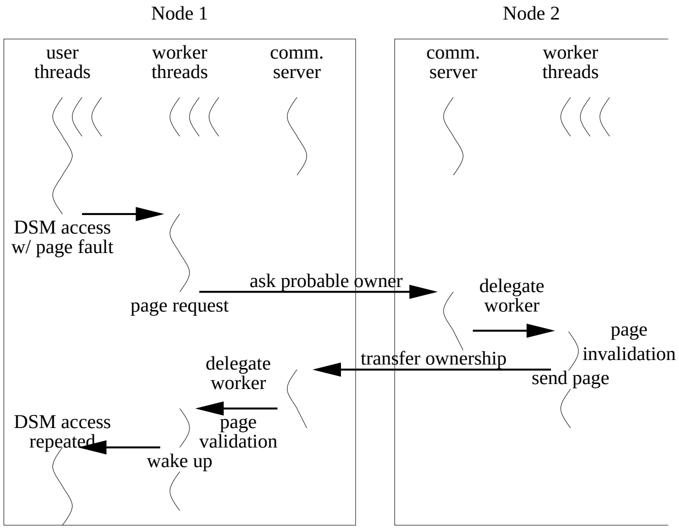

Figure 1: Operational Model for Page Fault Handling

Worker threads can also be activated via the DSM runtime system to function as user threads (`dsm_thread_create`) or when a page fault occurs, *i.e.* when a DSM datum is referenced with insufficient access rights on the node. The page fault handler then activates a worker thread to perform the task of obtaining the proper access rights to the page, which is an asynchronous signal-unsafe operation [18] that cannot be safely executed within the page fault handler. The worker thread may then send a request to the communication server of another node (the probable owner). Upon response, the communication server of the current node delegates a worker to validate the page and change the access rights. The worker then reactivates the user thread.

The responsiveness of a node is furthermore ensured by restricting the allocation of worker threads to task. Consider n page faults that occupy all n worker threads, such that the communication server cannot delegate page requests to any worker anymore. If all nodes were in such a state, the system would deadlock. Thus, one has to require that the pool of worker threads may at no time be engaged in handling tasks of only one type. We actually go one step further: First, we require that the size of the worker pool be at least the number of delegated message types (*i.e.*, task types assigned to workers). Second, we restrict the allocation of worker threads such that a worker may only be allocated a task if no worker is engaged in handling this message type yet or if the number of idle workers exceeds the number of currently unhandled message types.

3 DSM data identification

A runtime-supported DSM system implemented in software without modifications of the compiler or operating system has limited options to identify the DSM data area and control access to it. Modern operating systems generally provide an interface to control access to the virtual memory mapping, in particular to modify the access rights of memory pages owned by a process. With this facility, a page-based DSM system can be implemented.

One of our design goals was to minimize the modifications to existing source programs. Thus, the declaration of DSM data should be as simple as identifying the global data that is accessed by different DSM threads. However, a page-based DSM system requires that the DSM data be declared in a consecutive subsection of the address space. We therefore require the user to declare all DSM data (and only DSM data) in an area between two macros (see Figure 2). Besides bounding the DSM data area, the macros ensure that DSM addresses reside on different pages of the virtual memory system than non-DSM data. Additional macros support various data consistency models within the DSM declaration area. We require that pointers to non-DSM data not be used within DSM data structures. Furthermore, DSM threads may only take constant data or DSM data as arguments. This ensures that only values of non-DSM data but not pointers to DSM data shall be made available to other nodes.

```
#include <dsm.h>

DSM_DATA_BEGIN
int z = 1;     /* DSM data, shared by all nodes       */
float f = 1.0;
DSM_DATA_END

int y = 2;     /* global data, separate for each node */

void new_thread(void *arg)
{
...
}

void main(int argc, char **argv)
{
  dsm_attr_t attr; /* local data */
  dsm_thread_t dsm_thread;
...
  dsm_init(argc, argv);
...
  dsm_attr_init(&attr);
  dsm_attr_setnode(&attr, node); /* node DSM-thread runs on */
  dsm_thread_create(&dsm_thread, &attr, new_thread, arg);
...
  dsm_exit(NULL);
}
```

Figure 2: Sample Program

We are considering to restrict our system to operating systems compliant with the ELF object format. This would lift a number of constraints and ease the transition of existing software to our system. Generally, initialized global data is assigned to consecutive addresses if declared consecutively. Unfortunately, this does not hold for uninitialized global data. However, the ELF standard allows the declaration of user-defined data sections. A new section for DSM data would ensure that initialized as well as uninitialized DSM data be mapped to consecutive addresses, regardless of its origin of declaration. Thus, DSM data declarations may be mixed with non-DSM data declarations in the source code and may even be declared in different modules.

4 Communication and Heterogeneous Data Representations

Our system does not rely on any particular communication protocol. It assumes a message-passing paradigm. It fully abstracts from the communication medium but requires knowledge about the quality of communication, such as reliability (loss of data) and ordering (or absence of ordering). A current version implements reliable communication via sockets that guarantee that messages are received in the order that they are sent without a loss.

The communication facilities support heterogeneous environments by data encoding. We take advantage of the consecutive DSM data space and extract type information about each DSM datum prior to program execution. Typically, two nodes engage in a bidirectional communication, for example when node A requests data (a page) that node B owns and node B responds with this data to A. On a request, information about the data representation is embedded in the the header of the communication protocol. The receiving node compares this information with its data representation. If the representation matches, raw data is sent in the response including another header to indicate once more that the same data representation is used. If, on the other hand, the representation differs, each DSM datum to be sent is encoded in the external data representation format (XDR [17]) and, conversely, decoded at the receiver. This approach ensures portability while supporting heterogeneous environments and avoids the encoding overhead when possible.

In a heterogeneous environment, the layout of DSM data also has to be considered. It is imperative that a DSM datum reside on the same DSM page for each node of the distributed system. This ensures that an access fault for a datum refers to the same DSM page number within the entire system. In other words, while the absolute addresses may differ from node to node, the relative location *w.r.t. DSM pages* has to match. For example, a DSM datum resides on page n on each node but may have completely different absolute addresses, or even different offsets from the beginning of the page. This requirement has implications on the data representation within a heterogeneous environment. Since DSM data comprises a consecutive data space, the data types and alignments either have to match for different representations or data has to be placed explicitly on the next page when necessary. The former approach may require modifications to the compiler. The latter solution can be realized by compiler modifications or by preprocessing. Since we wanted to avoid compiler modifications, we chose the latter approach, as described in the following.

Each architecture in the distributed system is probed for its representation of basic data types and alignment rules by a generic test program. The results are merged to determine the greatest common size and alignment for each type. The DSM data declarations are then reprocessed and the worst-case consumption of data space is recorded. When these specifications indicate that a datum is placed on a new system page for at least *one* architecture, a page feed is generated for *all* architectures. This page feed may come prior to filling the entire page on other architectures and may thus result in a modest waste of space but on the "worst-case" architecture a page feed will be required.

5 Data Consistency and Access Protocols

Keeping DSM data consistent can be regarded as a trade-off between user-friendliness and efficiency. The user would prefer to keep a globally consistent view of data changes at all times; after all, the DSM data is said to be *shared* from the user's view. However, the communication overhead in distributing a change of DSM data to other nodes imposes a limit on the frequency of synchronization to ensure efficiency, responsiveness, and user acceptance of the system.

We decided to provide the user with a number of consistency models:

- Single Reader, Single Writer (SRSW): optional interface for the user when no distinction between read and write locks exists;
- Multiple Reader, Single Writer (MRSW): default for the user, implemented via an improved dynamic distributed manager algorithm supporting page migration with page invalidation [8];
- Multiple Reader, Multiple Writer (MRMW): advanced user-models, including Entry Consistency (EC) [4], open for future extensions.

The operational model of handling page faults is depicted in Figure 1 to illustrate the internal architecture of DSM-Threads. This simple case assumes that the probable owner and the actual owner of a page happen to be the same. Otherwise, page requests would be forwarded along the chain of probable owners by the communication server to avoid the overhead of context switches to worker threads. We omit further details of the decentralized dynamic manager algorithm and refer to the literature [8]. The protocols can be changed dynamically for a page. All protocols require that the user access the data in mutual exclusion; otherwise the result is undefined. The interfaces for mutually exclusive access are discussed in the following section.

6 Distributed Synchronization Methods

The following basic synchronization methods are supported: mutual exclusion, conditional synchronization, and barriers. The former two concepts should already be known to Pthreads users. The latter is designed for programmers that potentially want to modify existing programs to improve the efficiency. The most relevant synchronization interfaces are as follows:

- `dsm_mutex_lock`, `dsm_mutex_unlock`, `dsm_mutex_trylock`
- `dsm_mutex_read_lock`, `dsm_mutex_write_lock`, `dsm_mutex_read_trylock`, `dsm_mutex_write_-trylock`
- `dsm_cond_wait`, `dsm_cond_timedwait`, `dsm_cond_signal`, `dsm_cond_broadcast`
- `dsm_barrier`

The distinction between read locks and write locks does not exist for Pthreads. Thus, we provide an analog interface `dsm_mutex_lock`, which can be implemented directly as `dsm_mutex_write_lock`. However, the distinction between read locks and write locks should be easily integrated into existing programs. Only then can the user take fully advantage of the MRSW model. Multiple writer protocols optionally allow the specification of attributes during lock initialization to associate locks with DSM data (at a finer granularity than pages). The lock and unlock operations represent the acquire and release operations, respectively, commonly used for MRMW protocols. Notice that multiple writer protocols may circumvent page faults altogether if associated with DSM data since a test for proper access rights is made when a mutex is locked. If the access rights to the data associated with the lock are insufficient, the lock operation blocks while the data is requested using the probable owner chain. In other words, data requests for locks can be piggy-backed with lock requests for multiple writer protocols. For single writer protocols, we cannot employ this approach of avoiding page faults since we aim at backwards compatibility with Pthreads. In Pthreads, locks are not explicitly associated with data structures via the interface (lock initialization).

The implementation of mutual exclusion by locks uses a token-passing approach, where a distributed queue of threads contending for the lock is kept by the requesting nodes [14]. Notice that we distinguish the owner of a token (a node) and the owner of a lock (a thread). When an attempt is made to acquire a distributed write lock, a local lock table is consulted. If the current node holds the token, locking the mutex is performed via the Pthreads interface on a local mutex in the lock table associated with the distributed lock. If, however, the current node does not hold the token, the request is forwarded to the probable owner of the token recorded in the lock table. This is an extension of the migration algorithm used for DSM pages. Read lock attempts are also served in analogy to the page handling with the exception that a "lock invalidation" requires that the read lock of the current node and all nodes in the copy set be released before the invalidation is acknowledged. We are currently investigating an extension of this algorithm to handle different queuing policies (strict priority FIFO, priority inheritance / ceiling).

Distributed conditional synchronization is realized in much the same way as token locks. If a thread is suspended on a conditioned wait, signals are sent to the owner of the condition variable. This owner is defined as the holder of the token corresponding to the lock used for the conditional synchronization. Notice that only one mutex lock may be used in conjunction with conditional synchronization at a time. Thus, the definition provides a unique owner.

Barriers could be implemented on top of distributed locks and conditional synchronization. However, this would require that a node at a barrier contend with all other barrier participants within DSM memory since a counter needed to be modified by each node. The corresponding overhead of communicating DSM memory content may be avoided by using a tree barrier. First, threads form a group by calling a group registration interface. During the registration process, a bidirectional, balanced communication tree is built for future barrier calls. The first thread to register for a group is the root (logged in DSM data) while consecutive threads ask the root for inclusion in the group. The 2nd and 3rd threads become the right and left children of the root, while any following registrations are forwarded to the right or left child for even and odd registration numbers. Thus, registrations are either forwarded if two children exist or they become a child of the current node.

When a thread reaches a barrier, it first awaits the barrier message of its children, then sends a barrier message to its parent and finally awaits the parent's acknowledgment to its children. Threads can be removed from a group by informing the parent and the children of their new neighbors. Deletion will result in non-balanced trees but later insertions may regain the balance. This approach seemed preferable to the overhead of rebalancing the entire tree at each deletion. The deletion of the root furthermore requires logging the new root in DSM memory.

7 Data Distribution

The performance of a DSM system is often limited due to frequent page faults. For example, two nodes access distinct data on the same DSM page in an interleaved access pattern causing the content of the page to be communicated to the other node. This "false sharing" of DSM data can be addressed by distributing the distinct data onto different DSM pages. We collect trace data during execution time of an application recording page faults. After one or more runs, a user may opt to recompile the system using the data redistribution support. This facility automatically reorders DSM data locations based on access patterns. Data references are related to a group of threads that access this data within short consecutive intervals. This ensures that threads contending for data are placed within a group while other threads accessing the data in a later phase of the application do not belong to such a group. For each thread group, the corresponding DSM data is placed on distinct pages during recompilation. Currently, a preprocessor is designed to achieve reordering on the source level in a portable fashion. We are considering to handle data redistribution at the binary level in a portable way in the future but only for ELF formats.

Our design goals to handle C programs with global DSM data much like Pthreads programs limits the approaches for data distribution in the sense that references to DSM data cannot be changed by the runtime system. Thus, data redistribution has to result in a second compilation or linking pass. In contrast, object-oriented DSM systems or DSM systems with compiler modifications may take advantage of their multi-level indirection scheme for accessing data. Changes in the dispatching mechanism (changing 2nd level indirections) suffice to redistribute data.

8 Thread Distribution

A distributed thread can be created by calling a primitive `dsm_thread_create` (see Figure 2), which resembles the interface for creating POSIX threads but lets the programmer optionally specify a destination node via attributes. If no destination node is specified, the DSM system will select such a node using a history of load information and CPU throughput. For repeated executions of an application, trace data and thread group information may be used to distribute threads upon creation. In particular, a small number of threads of a group may be placed on the same node if very frequent page faults occurred for a distributed configuration, *i.e.* if the overhead of page faults for a distributed execution is likely to exceed the overhead of executing multiple threads on one node. Using these methods, repeated traces under different configurations can be used to improve the overall performance gradually by finding better thread distributions.

9 System Dependencies

Portability is a prime goal of the designed system. However, there are a number system dependencies caused by the selection of implementation choices to realize our design. First, we assume Pthreads, an interface to change page protections (`mprotect`), and signal delivery upon violation of page protections (`SIGSEGV`). These assumptions are consistent with POSIX 1003.1c [18]. A more subtle dependency is the knowledge of the fault address during page fault handling. While all systems we investigated provided such a facility, it is not part of any standard we know. Thus, fault addresses present a non-portable feature of our system, which have to be adjusted for new target systems. We also expect Pthreads synchronization primitives to be asynchronous signal-safe, *i.e.* routines for mutual exclusion and conditional synchronization are called within the page fault handler.

The alignment of DSM data on page boundaries also presents a portability challenge. Currently, we use assembler directives for page alignment (ELF object format) under the Gnu C compiler or Gnu linker options (AOUT object format) . We assume that consecutive addresses are assigned to initialized global data in the order in which they are declared. Uninitialized global data does not necessarily follow this ordering and is currently excluded from DSM data or requires preprocessing. We are considering to restrict ourselves to the ELF object format, which would enable us to lift this restriction.

Heterogeneous support requires the use of the external data representation standard (XDR [17]), which is widely used for remote procedure calls. We also extract DSM data from executables using the Gnu binary utilities and the DWARF debugging format. In addition, the Gnu binary file descriptor support (BFD) is used to provide a bijective mapping between function addresses and function names during thread creation.

In summary, our system dependencies concentrate around Gnu software, which has been widely ported. As a result, the DSM runtime system can be ported to a large number of architectures and operating systems. Currently, the lack of support for POSIX threads often presents more of a problem since the POSIX 1003.1c standard was only passed in 1995 but we expect future releases of operating systems to support the standard.

10 Open Problems and Future Work

Page validation in a multi-threaded environment poses the following problem. When a page is communicated to a node and placed in a buffer, this buffer needs to be copied into the page on the current node. The copy operation assumes that the page be write-accessible on this node. However, on a shared-memory multi-processor user threads may execute during this page update, even if they have a lower priority than the thread performing the page update. This creates a race condition in reference to the page update. The user threads may successfully access the *old* content of the page (write access enabled) before the updating thread writes the new content onto the page.

The source of the problem is rooted in the lack of an atomic "page update and change access rights" facility. Conventionally, such an interface may not be necessary but in a multi-threaded environment it becomes essential. However, there is an alternative.

The POSIX interface to virtual memory mapping, `mmap`, could be extended to allow an *existing* page to be *remapped* at a different virtual memory location. This would allow the temporary remapping of a page into a reserved area (not known to user threads), followed by enabling write access and the actual page update, which may be non-atomic. Only then would the page be mapped back to its original location in virtual memory. Thus, a race condition does not exist any longer since user threads cannot access the page prior to its remapping. We are currently experimenting with such an interface on the Linux operating system. Ongoing work also includes DSM heap allocation, priority handling, and an evaluation of a complete implementation. Furthermore, the system has been used as an extension to the Ada95 runtime system in order to support so-called passive partitions in a distributed environment via DSM [12].

11 Related Work

DSM systems have been an increasingly popular area of research. Several overviews compare different systems [15, 9, 16]. Software implementations of DSM, such as discussed in our paper, can be distinguished by their reliance on modifications of the operating system, source language, compiler, and runtime system. For a fair comparison with this work, we focus strictly on runtime-system modifications. These systems handle data sharing at the page level using page protection mechanisms as described in our work. The decentralized dynamic manager algorithm used in our system for the MRSW protocol is due to Li and Hudak [8]. An extension of this algorithm is used to implement token-based mutual exclusion. Notice that Li and Hudak's algorithm reduced to writers only is equivalent to the token-based algorithm by Naimi *at. al.* [14].

TreadMarks is a page-based system supporting different consistency protocols and multiple writers, most notably a lazy release consistency model, where only the data associated with a lock is updated upon locking (acquire) [2]. Midway also supports multiple consistency protocols, which can be changed dynamically [4]. Among these protocols is the model of entry consistency that extends lazy release consistency to distinguish read and write locks. This consistency model is adopted for the multiple writer protocol in our work.

Filaments employs a set of lightweight worker threads to act upon remote processing requests quickly, using adaptive schemes to dynamically reconfigure data placement [10]. We acknowledge the importance of data and thread distribution but use tracing and preprocessing techniques to address them statically since dynamic reconfiguration cannot be achieved without compiler modifications. This restriction is due to our backwards compatibility with Pthreads. Other DSM systems handle data at the level of objects (*e.g.* Orca [3]), tuples (*e.g.* Linda [1]), or at a variable sizes (*e.g.* Munin [5]), which also facilitates dynamic data placement via indirection.

Currently, distributed thread systems lack implicit support for global data. Chant uses a message-passing paradigm with a polling interface to the user, it supports remote service requests (RSRs) and remote thread operations, (*e.g.* joining threads) [7]. Nexus also uses RSRs and adds dynamically created global pointers (GPs) to reference remote data [6]. An implicit notion of DSM data is not supported but they can be built on top of Nexus or Chant. Our system cannot use non-threaded RSRs executing in interrupt handlers since DSM services require the execution of asynchronous signal-unsafe operations for page protection (due to POSIX [18]), thereby requiring the approach of workers and a communication server, which is similar to threaded RSRs. Ariadne implements migratable threads for shared-memory and distributed system [11]. On a distributed system, the access of a shared object causes the thread to migrate to the *static* location of the object. Thus, an implicit notion of global data is supported but the overhead of thread migration may cause overhead repeatedly. Our approach of a DSM system should provide *dynamic* adaptivity in terms of data migration.

12 Conclusion

We have presented design and implementation issues of DSM-Threads, a runtime system for distributed execution with distributed virtual shared memory. We motivated the design goals and stressed the ease of migration for a programmer from concurrent programming with POSIX threads on uniprocessors and shared-memory multi-processors to DSM-Threads on a distributed system. By providing a similar interface in syntax and semantics, an application should be transfered from Pthreads to DSM-Threads with ease. Extensions to the DSM-Threads interface provide the means to fine-tune an application, if necessary.

We discussed our choice of various data consistency protocols on a page-based level for multi-reader / single-writer protocols and a protocol for multiple writers via lock association. We detailed implementation choices, such as the selection of decentralized algorithms to support page-based consistency and distributed synchronization schemes (locks, condition variables, and barriers). A framework for data and thread distribution was presented to reduce "false sharing" while improving the overall throughput. Several restrictions due to heterogeneous support and backwards compatibility to Pthreads are emphasized to sketch the limitations of our approach. In addition, problems are identified that are inflicted by the presence of multi-threading and the restriction to avoid modifications to the operating system and compiler for reasons of portability. This paper is, to our knowledge, the first description of a system to support distributed threads on top of POSIX Threads via distributed virtual shared memory.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8), May 1986.

[2] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, February 1996.

[3] H. Bal and A. Tannenbaum. Distributed programming with shared data. In *International Conference on Computer Languages*, pages 82–91, 1988.

[4] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *COMPCON Conference Proceedings*, pages 528–537, 1993.

[5] J. Carter, J. Bennet, and W. Zwaenepoel. Implementation and performance of munin. In *Symposium on Operating Systems Principles*, pages 152–164, 1991.

[6] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[7] M. Haines, D. Cronk, and P. Mehrotra. On the design of chant: A talking threads package. In *Supercomputing*, pages 350–359, November 1994.

[8] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions of Computer Systems*, 7(4):321–359, November 1989.

[9] V. Lo. Operating systems enhancements for distributed shared memory. *Advances in Computers*, 39:191–237, 1994.

[10] D. K. Lowenthal, V. W. Freeh, and G. R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, August 1996.

[11] E. Mascarenhas and V. Rego. Ariadne: Architecture of a portable threads system supporting threads migration. *Software Practice & Experience*, 26(3):327–356, March 1996.

[12] F. Mueller. Supporting Ada95 passive partitions in a distributed environment. In *Ada Europe*, June 1997. (accepted).

[13] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[14] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *JPDC: Journal of Parallel and Distributed Computing*, 34(1):1–13, April 1996.

[15] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[16] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, pages 63–79, Summer 1996.

[17] Sun Microsystems, Inc., Mountain View, California. *External Data Representation Protocol Specification*, February 1986. Part Number 800-1324-03.

[18] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].