Decentralized Synchronization for Multi-Threaded DSMs

Frank Mueller

Humboldt University Berlin, Institut f. Informatik, 10099 Berlin, Germany
mueller@informatik.hu-berlin.de, phone: (+49) (30) 2093-3011, fax: -3010

Abstract

Synchronization protocols provide the means to arbitrate data accesses to shared variables in a DSM system. This work discusses the design and implementation of basic synchronization primitives for DSMs. It contributes a detailed study on the impact of per-node multi-threading on synchronization. Furthermore, the need for decentralized approaches for scalability is emphasized. Barriers are realized through a dynamically generated tree for a multicast on a barrier release. Communication parameters of different nodes can be taken into account when building this tree. Mutual exclusion is supported by serving requests in FIFO order of their arrival over a token-passing protocol. Conditional synchronization also adheres to FIFO ordering when queuing and signalling wait requests. Furthermore, conditional synchronization follows the token-passing approach for waits and signals but utilizes a static tree for broadcasts. The paper motivates the design decisions resulting in the described protocols and indicates potential problems and solutions for data or message races.

1. Introduction

In the past, DSMs used to support a single thread of execution per node. However, performance studies of multi-threading for multiprocessor indicated a promising benefit from multi-threading. These studies were subject to modeling through simulations and included results on actual multiprocessors, *e.g.*, by Bianchini and Lim [1].

Muller *et al.* investigated different partition schemes for grouping threads on the same or on different processors [8]. Their results indicate that thread placement is application dependent, which underlines the need to support both per-node multi-threading with local synchronization as well as distributed synchronization primitives.

Thitikamol *et al.* study the impact of per-node multi-threading for DSMs on a number of parallel benchmarks [13]. Their results show a potential for speed-ups for all benchmarks, in two cases of over 20%, when multi-threading is added to a DSM.

Other multi-threaded DSMs (MT-DSMs) have been designed to exploit this potential of threads to hide communication latencies, *e.g.*, Millipede [4], Brazos [10] and DSM-Threads [5] support multiple threads per node, just to name a few. Speight *et al.* also discuss the impact of multicasts supported as a communication mechanism in contrast to point-to-point messages [11].

However, most DSMs use centralized protocols to implement synchronization primitives, often citing better performance due to a relatively small number of nodes in the system.

This paper describes the design of decentralized synchronization methods for multi-threaded DSMs. In the following, protocols for barriers, locks and conditional waits are presented, emphasizing the impact of per-node multi-threading within a DSM. The assumptions for the design include the following aspects:

- The design of synchronization protocols addresses the problem in a distributed system with communication through underlying message passing.
- Communication is point-to-point only between any two nodes.
- Communication is reliable but out-of-order message delivery is tolerated.
- Protocols should be decentralized (rather than client-server oriented) to provide better scalability.
- Per-node multi-threading is supported through a POSIX threads (Pthreads) style.

These assumptions provide a framework for synchronization in the presence of multi-threading that should be general enough to be widely applicable, *e.g.*, for in-order message passing (without changes) or for unreliable message passing (with minor changes). Client-server approaches should use different protocols, *e.g.*, to arbitrate mutual exclusive access through permission from a single server. However, such approaches may be subject to a bottleneck implying potentially

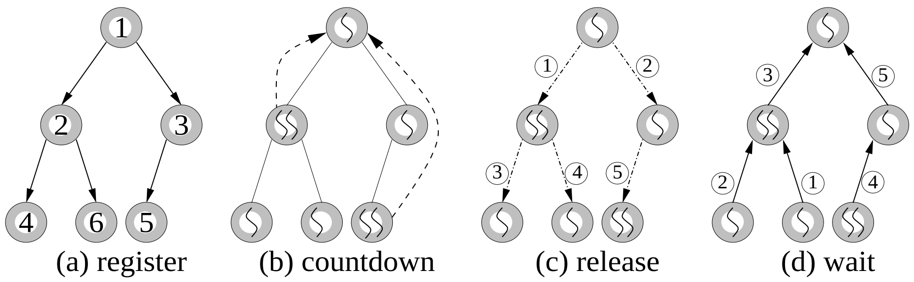

Figure 1: Phases of Barrier Synchronization

poor scalability. Hence, this work describes *decentralized* approaches to communication based on point-to-point message passing while trying to minimize the message overhead and the latency of message distribution.

2. Barriers

Barriers serve as a common synchronization point for all participants of this synchronization in order to ensure data consistency with regard to results calculated prior to the barrier. In a DSM, barrier protocols are often used to piggyback data with barrier messages to realize consistency. In the following, a description of the barrier design is given that allows such piggybacking.

Barriers may be designed as "flat" barriers or tree barriers for n participating nodes. A tree barrier designates a root node and orders other participating nodes in a tree structure relative to this root. A flat barrier is a special case of a tree barrier of height 2, *i.e.*, a root and its $n-1$ children. A balanced binary tree barrier has 2 children for non-leaf nodes with an overall height of $log(n)$.

A binary tree barrier has the advantage over a flat barrier that messages can be distributed in parallel along the different branches. For this reason, a binary tree design will be discussed in the following. At the same time, the design can be easily extended to support $k-ary$ trees or a LogP-like tree with the same protocol [3].

A barrier in a MT-DSM represents a multicast synchronization structure in the sense that the n participants may be a subset of the total number of nodes in the system and there may be more than one thread per node participating in a barrier. In order to minimize the message overhead, messages should only be exchanged once for a pair of neighboring nodes in the tree, regardless of the number of participating threads within each node.

The design of the barrier includes two different phases, a registration phase and a repetition phase. The registration phase involves the process of building a tree structure but also includes the first barrier round. The repetition phase supports consecutive rounds on the same barrier over the already built tree. The registration phase consists of the following steps:

1. The node of the first thread to arrive at the barrier becomes the designated root of the tree and anticipates $n-1$ pending participants.
2. If a thread of a new node arrives at the barrier, it tries to "register" with the root. The root then decreases the pending participants. Any node in the $k-ary$ tree receiving a "registration" message will forward such a request alternating between its k children, if existent. If there are less than k children, then another child node is added for the current node, *i.e.*, up to two children for our binary tree.
3. If a thread of an already registered node in the tree arrives at the barrier, it sends a "countdown" message directly to the root to decrease its count of pending participants.
4. Upon zero pending participants, the root sends a "release" message to its children. Any receiving node will transitively forward this message to its own children, mark the sender as its father in the tree and wake up all local threads participating in the barrier.

Figure 1(a) depicts the registration phase for a sequence of nodes 1 to 6 arriving at the barrier in numerical order. Figure 1(b) illustrates that further threads arriving at the barrier in nodes 2 and 5 simply send a countdown message (dashed arcs) to the root. Figure 1(c) depicts the order in which release messages are sent after the root has been noticed of all arrivals.

In step 3, the countdown message is sent directly to the root for two reasons. First, the message is not sent to the father in the tree since the father may not be stored yet. Second, the number of local threads per node is not known a priori implying that a node cannot accumulate local countdowns before relaying them elsewhere. During the next phase, such an accumulation is realized.

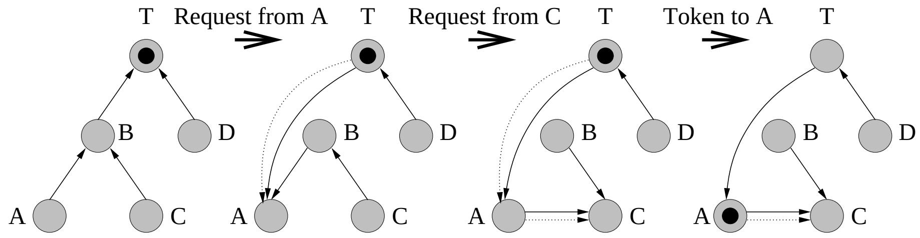

Figure 2: Single-Threaded Mutual Exclusion

The repetition phase for consecutive barriers utilizes the existing tree as follows:

1. If a thread arrives at a barrier or if a node receives a "wait" message, it decreases the number of locally pending participants, which includes both local threads and its k children. If this number reaches zero, a "wait" message is sent to its father for non-root nodes. For the root node, a "release" is sent to all children in this case.
2. If a node receives a "release" message, it will transitively forward this message to its own children and wake up all local threads participating in the barrier.

Figure 1(d) depicts the first step where wait messages on a higher level in the tree can only be sent after messages from lower levels have been received, indicated by a sample message order.

The dynamic registration phase allows an efficient distribution of messages once the repetition phase starts. It enforces a communication structure of a balanced tree. Notice also that it favors fast nodes since they are likely to be placed at a higher level within the tree and can thereby speed up the message distribution. This affects networks or clusters of non-uniform computing nodes with regard to their speed, *i.e.*, this approach may prove useful where simple LogP fails due to differing parameters depending on a node's architecture. The design could be generalized by increasing the arity of nodes at higher levels since these nodes are likely to be faster and may handle more messages within the same time frame than nodes on lower levels in the tree. The design allows such non-uniform tree structures. The optimal arity is subject to an extended LogP model taking node-specific parameters into account but the details are beyond the scope of this paper.

The overall design provides an efficient solution to barrier synchronization in MT-DSMs for a constant number of participants. The message overhead is $O(n)$ for n participants with a message delivery overhead of $O(log(n))$ since messages are delivered in parallel assuming sufficient bandwidth and proper tree construction with respect to latencies. If different sets of nodes were to participate in barriers, separate barrier objects had to be used per set, *i.e.*, by going through a registration and repetition phase for each set.

3. Mutual Exclusion

Mutual exclusion provides the means to arbitrate the access to shared resources, *i.e.*, mostly DSM data in our case. In an MT-DSM, distributed mutual exclusion between nodes and between threads on a node must be realized. In the following design, distributed mutual exclusion is based on a token-passing dynamic approach while mutual exclusion on a node uses POSIX-style mutex locks and condition variables [12]. Due to message latencies in a distributed system, locking attempts result in suspension of the requesting thread whose execution resumes only when mutual exclusion can be granted. Furthermore, FIFO ordering of requests shall be realized to ensure fairness.

3.1. Base Protocol

The base protocol for mutual exclusion adopted in this work is derived from Naimi *et al.* [9]. In this decentralized protocol, nodes form a logical tree pointing via *probable owners* towards the root. Consider Figure 2. The root T holds the token for mutual exclusion. A request for mutual exclusion travels along the probable owners (solid arcs) to the token holder. In the example, the request by A is sent to B. Node B forwards the request along the chain of probable owners to T and sets its probable owner to A. When a request arrives at the current token holder, the probable owner and a *next* pointer are set to the requester, *i.e.* T sets the *next* pointer (dotted arc) and the probable owner to A. The next request from C is first sent to B. Node B forwards the request to A following the probable owners and sets its probable owner to C. Node A sets its *next* pointer and probable owner to C. When the token holder returns from its critical section, the token is sent to the node that *next* points to. In the example, T passes the token to A and deletes the *next* pointer.

Consider the process of request forwarding again. When a request passes intermediate nodes, the probable owner is set to the requesting node. This causes

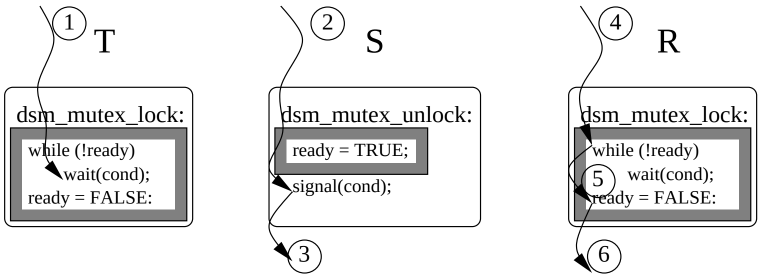

Figure 3: MT Data Race on Lock Contention

a transformation of the tree to a forest. The traversed nodes form a tree rooted in the requesting node, which will be the token holder of the future. Nodes that have not been traversed are still part of the original tree rooted in the current token holder. For example, when B forwarded A's request to T, B set the probable owner to A. At this point, one logical tree $C \to B \to A$ is rooted in A while another tree $D \to T$ is still rooted in T. Once the request reaches T, the separate trees are merged again (2nd tree of Figure 2).

Should k requests be in transit, then there may be up to $k+1$ separate trees in the forest. Once all requests have arrived at current or future token holders, the forest collapses to a single tree again rooted in the last requester. Hence, the probable owner chain is subject to dynamic path compression as part of the protocol, thereby forming the aforementioned tree. The *next* pointers form a distributed queue of pending requests from the current token holder to the last requester. A new request would simply be appended at the end as described above.

This protocol has an average message overhead of $O(log(n))$ since requests are propagated through a tree. One additional message is needed to send the token.

3.2. Multi-Threading Extensions

The base model assumes that there is only one requester per node at a time and requests are ordered FIFO. In a multi-threaded environment, multiple requests may be issued from a node. This implies the need for additional data structures and a potential for data races, as discussed in the following.

First, the protocol is extended to support a list of *next* pointers per node, where each pointer represents a request of a different thread. This ensures that the length of the distributed queue is no longer bounded by the number of nodes but instead by the number of threads over all nodes. Requests still follow the chain of probable owners to its tail (*i.e.*, the node whose probable owner points to itself), where the request is appended. Notice that if a request is issued on a node at the tail of such a non-empty queue, then the new request is simply appended to the local list, which may even include multiple consecutive requests originating from local threads. In such a case, an unlock causes the token to be passed from one local thread to the next one obeying the FIFO policy. In general, the *next* pointer still provides a distributed queue structure that may now be cyclic with regard to the set of nodes.

Second, POSIX-style mutex locks and condition variables utilized on the local node to await the token may cause data races between multiple threads on this node. Consider the situation depicted in Figure 3. Assume that all thread share a node and flags are shared between threads, expect for the "race" flag, which is local (on the stack). Shaded boxes depict a common monitor (POSIX lock). Thread T is suspended in a conditional wait since the token is not present yet. When thread S issues an unlock and the token arrives at T's node (incidentally the same node as S and R), a corresponding ready flag is set and the condition is signalled. Before thread T resumes execution, another thread R issues a lock request and utilizes the available token racing by thread T. This clearly violates the FIFO policy of queuing.

The problem can be resolved (see Figure 4) by introducing a counter of waiters, which is incremented before entering the while loop in T and decremented after the loop, respectively. The data race in thread R can then be prevented by introducing an on-stack flag "race", which is set if "ready" is true and number of waiters is non-zero before the increment. The loop iterates while the ready flag or the race flag is true, and the race flag is cleared inside the loop. In effect, the race flag forces locally competing threads into an initial wait, even if the token condition has been signalled.

The resulting protocol still has a message complexity of $O(log(n))$ since no additional messages are introduced and path compaction still occurs within n nodes as before. This becomes clear since probable owners only relate to nodes, not threads, even though the message queue may be cyclic (and longer than n).

The counter approach is a general method to make

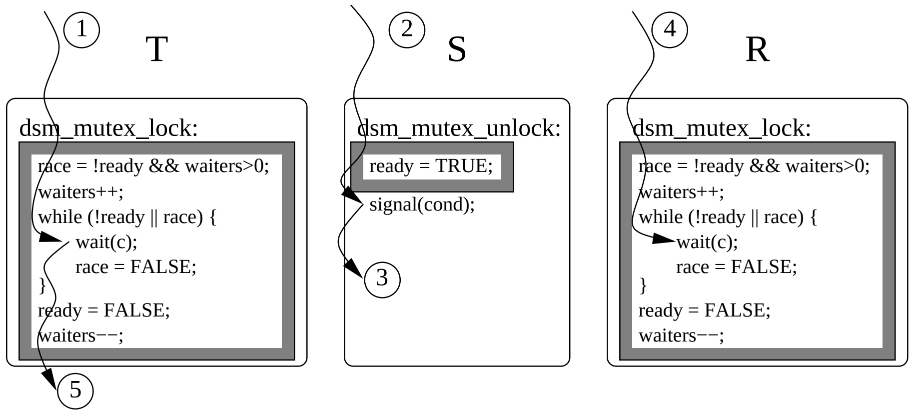

Figure 4: Race prevention by Counting Waiters

conditional waits fair, *i.e.*, the guarantee FIFO queuing, even in the presence of contention. It relies on the assumption that the wait operation is not subject to spurious wakeups itself. For Pthreads, spurious wakeups may exists for pthread_cond_cond and it depends on the actual implementation if they actually may occur. The problem can be described as follows. A spurious wakeup, *e.g.*, due to an interrupt, may cause a thread in the middle of a queue to wake up, re-check the condition and suspend. This results in re-queuing at the end of the FIFO queue, *i.e.*, the thread lost its earlier turn due to the spurious wakeup. A more general approach to providing fair and true FIFO waits of Pthreads requires explicit queuing of thread identifiers. The value of the thread identified can then be used in the condition to check for wakeups. We decided against this method due to the additional overhead of bookkeeping given the fact the spurious wakeups do not occur frequently, if at all.

4. Conditional Waits

Conditional waiting in a DSM utilizes the changing values of DSM variables tested in a loop condition in analogy to POSIX. The previous section already introduced mutex locks for MT-DSM systems. The remaining description is dedicated to the design of conditional waits in a distributed environment with multithreading per node. For this purpose, an extended version of the mutex protocol of the prior section is adopted and applied to conditional waiting.

The mutex protocol used a probable owner list, a distributed next queue and a token. For conditional waits, the list and the queue ensure the handling of requests and their service, as before. The token, however, is not passed unconditionally to the next requester upon unlocks. Instead, the token represents the head of the distributed next queue, *i.e.*, the waiter that should receive the first conditional signal issued.

The head of the distributed next queue should also be managed by a decentralized approach, namely, a *current owner* chain. This chain is similar to the probable owners in that it always points towards to head of the next queue. The following rules serve this purpose:

1. The current owner is set to self when a node receives the token.
2. The current owner it is set to the destination if a node forwards the token to a destination node.

In addition, conditional waiting is also subject to the FIFO policy of queuing. This has the same implications as with mutex locks, namely, a data race has to be prevented by a counter of waiters guarding the loop entry of the POSIX-style conditional wait.

Registration of conditional waits has an average message overhead of $O(log(n))$ since requests are propagated through a tree via the probable owner chain.

4.1. Conditional Signalling

Conditional signalling can then be realized by submitting a signal message to the head of the next queue indicated by the current owner chain. Upon reception of this message, the suspended thread is reactivated and the token may be passed one element down the next queue.

Caution has to exercised to prevent a possible message race, depicted in Figure 5. Node T holds the token but there is no wait pending on condition variable c, indicated by zero waiters. Node R obtains a mutex m, evaluates the global flag as false and suspends on c implicitly releasing m. This includes a wait request from R to T. Meanwhile, node S obtains m, clears the flag, releases m and signals c, which results in a message to T as well. If this signal from S arrives at node T *before* the queuing request from R, then the signal is absorbed while request R may pend indefinitely, possible causing a deadlock.

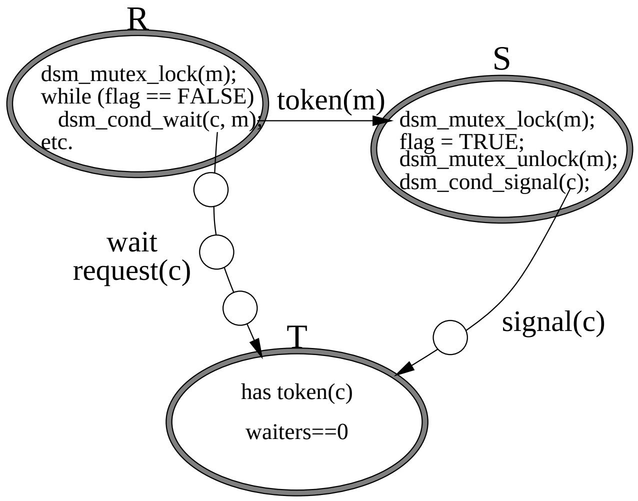

Figure 5: Message Race on Conditional Signal

At first, it may seem that this scenario can only occur with out-of-order message delivery but it turns out that even in-order message delivery faces this problem. Consider the wait request from R to T. If the probable owner chain leading to T includes more intermediate nodes than that from S to T, the wait message may take longer than the signal message, for example.

The cause of this race is rooted in the semantics of waits on a condition variable associated with a mutex in conjunction with the corresponding communication of messages and their partial ordering. The semantics requires an "atomic" release of the mutex and queuing of the conditional wait [12]. While this can be achieved on a true shared memory architecture, a distributed system cannot guarantee this. The problem may be solved in a different manner:

1. Signals received at an empty queue head are accumulated.
2. When a request arrives at the head, it is immediately signaled and the signal counter is decremented.
3. If the condition token is passed from one node to another (in the event of signalling), pending signals are piggybacked.

This solution obviously prevents deadlocks in the above scenario. Notice that it may result in spurious wakeups. If a signal is accumulated, the flag is set by another thread and a wait request is issued next, then the signal may cause an immediate wakeup. Upon resumption, the flag is still set and the thread issues another wait request in the loop, *i.e.*, a spurious wakeup has occurred. These spurious wakeups do not cause any harm. In fact, they are explicitly covered by the wait protocol since the conditional has to be re-checked after a wakeup according the the semantics for conditional waits.

Conditional signalling has the same average message overhead as conditional waiting of $O(log(n))$ since requests are propagated through a tree. Instead of using the probable owner chain, the current owner chain defines the propagation path. The current owner chain is also subject to the same dynamic path compression rules as the probable owner chain.

4.2. Conditional Broadcasts

A conditional broadcast is, in effect, a multicast to all conditional waiters. A seemingly obvious approach for a multicast protocol would be to signal all waiters in the distributed next queue. However, the message race problems that conditional signals were subjected to are also relevant to conditional broadcasts.

Again, there exists a potential message race between the broadcast signal and the wait request. If the signal races ahead of the wait to the head of the distributed next queue, then the wait will not be affected by the broadcast. However, the value of the global flag indicated a need for waiting. In effect, the waiter may be blocked indefinitely, *i.e.*, possibly deadlocked, due to a semantically incorrect synchronization sequence subject to the design of the protocol.

The approach of signal accumulation was a solution to this problem for conditional signals. For broadcasts (multicasts) on the condition variable, however, this approach does not fix the problem. There is no global knowledge of the number of signals that would need to be accumulated in order to ensure that all waiters receive a signal. The length of the distributed next queue does not cover all waiters since wait requests may still be in transit, as outlined before. An ad hoc approach would be to simply use the maximum number of threads over all nodes but this may potentially result in a large number of spurious wakeups, not just one as in the case of a conditional signal. Furthermore, in a dynamic DSM, new threads may be created on the fly so that the global number of threads may not even be available at any time.

The solution taken here relies on a separate data structure, a static tree of all nodes in the DSM, to issue the broadcast request. The construction of the tree is subject to the thread creation process and remains in place until program termination. A broadcast is issued using the following steps:

1. Send a conditional broadcast message to the root of the static tree of all nodes.
2. Upon reception of the broadcast, relay the message to all children, resume the execution of threads waiting locally on the condition variable and delete all local next pointers for this condition variable.

This approach realizes a traditional broadcast but it does not change the dynamic paths used for conditional waits or signals, *i.e.*, the probable and current owner chains. Notice that this approach results in a true broadcast whereas a multicast would have sufficed. It appears that the utilization of dynamic data structures as described above does not provide an easy solution to the multicast problem. On the other hand, a static broadcast tree has the advantage of a logarithmic message relay for n nodes while other approaches would need to iterate the distributed next queue, potentially resulting in a linear message relay for a large number of waiters. In summary, the message overhead for the tree broadcast is $O(n)$ but the message delivery overhead is $O(log(n))$ since messages are delivered in parallel assuming sufficient bandwidth.

5. Implementation

The synchronization mechanisms have been implemented in DSM-Threads, a DSM system with asynchronous communication and a POSIX-style interface as indicated by the examples in the previous section. The synchronization mechanisms have proved to be robust in functionality tests of up to 30 threads per node over 8 nodes. These tests included stress tests with high contention for conditional waits and signals, locks and barriers in the DSM system. A number of Splash-2 benchmarks have also been ported and show correct functional behavior [14]. These tests were performed on a PC Cluster under Linux using DSM-Threads with FSU Pthreads for per-node multi-threading [7]. We also utilized a Fast-Ethernet switch and a Myrinet switch. The former works under TCP and a UPD extension that prevents message losses, the latter utilizes Madeleine [2] over BIP. We are currently evaluating the performance of the benchmarks with respect to the involved synchronization overhead and its implications for scalability.

Thitikamol *et al.* indicated problems when algorithms originally used "global" data that was assumed to be distinct between computation nodes. When multi-threading is used, threads on the same node suddenly share this data, which may cause problems. In our work, we did not encounter problems with per-processor global data for two reasons. First, the design of DSM-Threads included per-node multi-threading from the beginning. Hence, any algorithms implementing DSM protocols already deal with situations where data is shared between local threads. The synchronization protocols discussed in this paper are one example, extensions to consistency protocols for multi-threading are another example [6]. Second, DSM-Threads adheres to an API similar to Pthreads. This implies that applications for DSM-Threads should follow the shared-memory programming paradigm. For example, Splash-2 benchmarks [14] follow this paradigm in that a global variable simply becomes a DSM-global variable. Any Pthreads applications are also suitable for distributed execution under DSM-Threads with only minor modifications due to name changes in the API.

6. Conclusions

This paper describes the design and implementation of basic synchronization primitives in a DSM emphasizing the impact of per-node multi-threading on one side and the need for decentralized approaches for scalability on the other side. Barriers are realized through a dynamically generated tree for a multicast on a barrier release. Communication parameters of different nodes can be taken into account when building this tree. Mutual exclusion is supported by serving requests in FIFO order of their arrival over a token-passing protocol. Conditional synchronization also adheres to FIFO ordering when queuing and signalling wait requests. Furthermore, conditional synchronization follows the token-passing approach for waits and signals but utilizes a static tree for broadcasts. The paper motivates the design decisions resulting in the described protocols and indicates potential problems and solutions for data or message races.

Acknowledgements

Thomas Röblitz tracked down several problems during the implementation stages.

References

[1] Ricardo Bianchini and Beng-Hong Lim. Evaluating the performance of multithreading and prefetching in multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):83–97, August 1996.

[2] Luc Bougé, Jean-Fran cois Méhaut, and Raymond Namyst. Madeleine: an efficient and portable communication interface for multithreaded environments. In *Parallel Architectures and Compilation Techniques*, pages 240–247, October 1998.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[4] A. Itzkovitz, A. Schuster, and L. Shalev. Supporting multiple parallel programming paradigms on top of the millipede virual machine. In *Workshop on High-Level Programming Models and Supportive Environments*, pages 25–34, April 1997.

[5] F. Mueller. Distributed shared-memory threads: DSM-threads. In *Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, April 1997.

[6] F. Mueller. On the design and implementation of DSM-threads. In *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).

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

[8] H. L. Muller, P. W. A. Stallard, and D. H. D. Warren. Multitasking and Multithreading on a Multiprocessor with Virtual Shared Memory. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, pages 212–221, February 1996.

[9] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, April 1996.

[10] W. E. Speight and J. K. Bennett. Brazos: A third generation DSM system. In *Proc. of the USENIX Windows NT Workshop*, August 1997.

[11] W. E. Speight and J. K. Bennett. Using multicast and multithreading to reduce communication in software DSM systems. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pages 312–322, February 1998.

[12] 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].

[13] K. Thitikamol and P. Keleher. Per-node multithreading and remote latency. *IEEE Transactions on Computers*, 47(4):414–426, April 1998.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.