Optimizing Software Shared Memory Communication with InfiniBand

Hubert Eichner, Carsten Trinitis and Tobias Klug

Lehrstuhl fuer Rechnertechnik und Rechnerorganisation
Technische Universität München
Boltzmannstr. 3, 85748 Garching bei München, Germany
Email: {eichnerh,trinitic,klug}@cs.tum.edu

Abstract: Implementations of algorithms based on shared memory are widely used in the area of High Performance Computing. The concept of cluster computing, however, does not imply the availability of a globally shared memory across nodes in that cluster. Therefore, several concepts for an emulation of a globally shared memory, called Software Distributed Shared Memory (SDSM), exist. However, these concepts did not achieve widespread acceptance because a significant part of the overall execution time is spent on the emulation of the shared memory region. With the introduction of highspeed interconnect-technologies, e.g. Myrinet\textsuperscript{1} or InfiniBand\textsuperscript{2}, higher bandwidth and lower latencies have become available. However, special inter-process communication methods like direct access to other nodes’ memory are especially useful for the further development of existing SDSM concepts. Using this techniques, existing implementations can be greatly simplified and optimized. This paper discusses some common problems of existing SDSM-protocols and then presents solutions for these issues by taking advantage of InfiniBand specific communication patterns.

1 Introduction

Algorithms for compute-intensive tasks based on shared memory are widely used in the area of High-Performance-Computing (HPC).

On modern operating systems, the address spaces of single processes are separated; in order to access the memory of another process, shared memory techniques are used. Especially for NUMA (Non-Uniform-Memory-Access)-architectures, e.g. SGI’s Altix-Technology\textsuperscript{3}, or with SMP (Symmetric Multiprocessing)-architectures, parallel computation is distributed among multiple processes sharing a common part of main memory.

With clusters, i. e. the grouping of multiple, interconnected computers, this technique of computation is normally not supported by the hardware.

\textsuperscript{1}http://www.myri.com/myrinet
\textsuperscript{2}http://www.infini.nibanda.org
\textsuperscript{3}http://www.sgi.com/altix
Because of the wide adoption of clusters and high-speed interconnects, e.g. Myrinet or InfiniBand, it is desirable to use shared memory programming with multiple, interconnected computers as well. Some projects try to emulate a globally shared memory region by enhancing existing systems with special hardware like e.g. the SCI-technology\cite{Sta90}. On the other hand, many projects tried to implement pure software-based solutions, normally referred to as Software DSM (SDSM) protocols. As opposed to NUMA- or SMP-architectures, consistency is not guaranteed by cache- or memory-controllers, but by a software-based consistency-protocol.

This paper presents an implementation of an SDSM-protocol based on the AMD64-architecture\footnote{http://www.amd.com/amd64} and the InfiniBand technology. First, the single components of such a system are discussed. Finally, InfiniBand-based optimizations and benchmark results are presented.

## 2 Overview of HLRC

### 2.1 HLRC Basics

The HLRC-protocol [IIt98] describes an architecture for SDSM-systems which provides a consistency model called \textit{Home Based Lazy Release Consistency}. It is based on existing protocols like \textit{Eager Release Consistency} [CBZ91] and \textit{Lazy Release Consistency} [KChZ92], [Kel94] and was first presented in [IIt98]. This section gives a short overview of HLRC to help understand the optimizations presented in section 3.

Each process participating in the computation is presented a virtually contiguous region of memory for shared memory operations. Access to this region is controlled by the underlying protocol (i.e. HLRC), thus allowing the protocol to virtualize the shared memory region by tracing accesses and communicating with other nodes. Technical details about HLRC implementations are described in [Ose04].

Two or more accesses conflict (\textit{conflicting accesses}) when they reference the same memory location and at least one of them is a write access. Similar to existing shared memory programming paradigms like e.g. OpenMP\footnote{http://www.openmp.org}, conflicting accesses can be avoided by using mutual exclusion algorithms and global synchronization functions. This functionality is provided by so-called \textit{locks} and \textit{barriers}, respectively.

The shared memory region is split up into page frames of 4KB size each on every node. In common shared memory systems, page frames with the same index relative to the shared memory region’s base address in different processes relate to the same underlying physical page. This characteristic could be simulated in an SDSM system by first assigning each page an owner, a so-called \textit{home node}. Pages that have a home node different from the current node are called \textit{remote pages}; the reference version of a page at its home node is called \textit{original page}. Sequential consistency could be achieved by avoiding concurrent accesses to the same page by different nodes. Thus, one page could be only written to or
read from by one node at a certain point in time.

Obviously, this approach would result in great performance degradation because of high network bandwidth usage and high latencies for most memory accesses. In addition, trac- ing accesses to pages by protecting them can be done on a per-page-granularity only. Therefore, HLRC describes a method to allow nodes caching of pages after fetching them from their home nodes. A page can be cached for a certain amount of time, called an *interval*. The current (node specific) interval ends, and a new interval starts when either a lock is released or when a barrier is performed. When an interval ends,

1. other nodes involved are notified about pages changed by sending them *invalidation messages*. Involved nodes are either the remote node acquiring the released lock or, in case of a barrier operation, all other nodes.

2. changes to cached pages are sent to their respective home node so they will be incorporated into the original page.

It can be shown that a *properly labeled* program based on HLRC produces the same result as it would on a memory model where all accesses are executed in sequential order (e.g. an SMP system). A program is properly labeled if all conflicting accesses are separated by an *acquire-release-chain*. An Acquire-release-chain is a block of instructions accessing memory that is enclosed by a beginning acquire of a certain lock and an ending release of that lock.

Further on, each node is identified by a unique number (*ID*), beginning with 0.

### 2.2 Current Implementations

Existing HLRC implementations depend on the existence of two threads: A *computation thread* submits outgoing requests to other nodes and performs the actual computation. A *request handler thread* processes asynchronous requests from other nodes. Requests to other nodes can either be requests for a page, requests for a lock, or requests for applying remote changes to a local page. Standard interconnect technologies like Gigabit Ethernet [IEE02] do not natively support operations for directly accessing remote memory locations. Thus, action is required by each node to receive requests and answer them according to a client-server-model. Each node is a client and a server at the same time, performing computations on the one side and serving requests received from other nodes on the other side.

Running an extra thread for incoming requests adds significant overhead to the protocol:

- Outgoing requests to other nodes stall the execution of the computation thread until the request is being served by the remote node.

- Incoming requests are serialized by the request handler thread. This results in high latencies for the requesting nodes when many requests arrive at the same time.
On uniprocessor systems, a significant amount of time can pass until the request handler thread is invoked by the operating system’s scheduler. Additionally, invocation of the request handler thread directly reduces the time available for the computation thread.

InfiniBand provides several operations for accessing remote memory locations that are used to eliminate the need for the request handler thread. The next section will give a short introduction to InfiniBand and these operations.

3 InfiniBand

InfiniBand is a standardized interconnect technology specified by the InfiniBand Trade Association\(^6\) for interconnecting processor nodes and I/O nodes. Besides specifying high bandwidths in the range of 250 MBytes/s to 3 GBytes/s, it provides very low latencies for intra-node connections. This is achieved by completely eliminating the need for a protocol stack in software. The InfiniBand equivalent of a network card, a Host Channel Adapter (HCA), is used for sending and receiving data directly from user level programs using DMA, avoiding entering the kernel context or using the host’s CPU. The kernel level driver is used in the initialization phase of InfiniBand connections only. This results in very low latencies around 5\(\mu\)s [Reg05]. Tests based on Ethernet in contrast have proven latencies starting at 40\(\mu\)s [Nat05].

The InfiniBand specification describes several modes for transmitting data. This section will outline these modes that can be used to eliminate the request handler thread used in existing DSM implementations.

- **RDMA-Read Operation**
  The RDMA-Read (Remote Direct Memory Access Read) operation allows a host to receive memory regions on other hosts without invocation of the remote or the local CPU. The requester specifies a remote virtual address and a translation table index called \(R\_KEY\) used by the remote HCA to translate the virtual address to a physical address. Additionally, a local virtual address and the size of the data to be fetched are specified before submission of the RDMA-Read request. After receiving the request, the remote HCA translates the requested virtual address into a physical address, fetches the data via DMA and sends it to the requesting node. The requester’s HCA then copies the data to main memory, again using DMA. The processor on the remote host is not invoked.

- **RDMA-Write Operation**
  The RDMA-Write operation can be used to write data from local main memory into another node’s main memory. The technical details are the same as for RDMA-Read.

\(^6\)http://www.infinibandta.org
• Atomic-Compare-&-Swap
The Atomic-Compare-&-Swap operation is used to compare the value of a given remote memory location with a user-specified value. If these values are equal, the remote memory location’s value is overwritten with a new, given value. This operation is atomic in the sense that no other InfiniBand specific messages may access the given memory location while the Compare-&-Swap operation is performed.

4 InfiniBand optimizations for HLRC

As outlined in section 2.1, the request handler thread is responsible for handling three types of incoming messages:

• Requests for a page (page-fetch)
• Requests to apply changes a sender made on a page owned by the receiving node
• Requests for a given lock

While there are more possible InfiniBand-specific optimizations than just eliminating the request handler thread, our current work focuses on implementing replacements for the three aspects noted above and measuring how these changes influence protocol performance. This section will discuss in detail how the request handler thread can be eliminated by using the facilities provided by InfiniBand as explained in section 2.2.

Similar changes to HLRC were proposed by Ranjit Noronha and Dhabaleswar K. Panda in [Ran04]. Our implementation, however, is based on previous work which has been carried out in [Ose04] and [OTTM04], dealing with an initial porting of a VIA-based HLRC-implementation to InfiniBand. This implementation was then extended to AMD’s 64-bit-architecture AMD64 and is currently being optimized to make specific use of InfiniBand features as described in the following paragraphs.

4.1 Requests for pages

The most straightforward optimization is applied to page requests by using RDMA-Read operations to fetch pages from remote nodes. However, one special case has to be taken into account. In HLRC, every page has a home node. This is the node referenced by other nodes when accessing and committing changes to this page. There are some general rules concerning home nodes in the implementation with a request handler thread:

1. The first node accessing a page becomes the home node. To determine the ID of the home node, every access to a page with an unknown home node results in a page fetch from a default home node indicated by a simple modulo operation (default_home=page_num%num_nodes).
2. The first node requesting the page from this default home node becomes the actual home node of that page.

3. Further requests for that page arriving at the default home node are then forwarded to the actual owner. The requesting node is notified about the ID of the actual home node when receiving the requested page.

Without a request handler thread, requests for pages cannot be forwarded; thus, a different approach was chosen to determine the home node of a given page: every node uses an array of home node IDs for all pages, `page_home[NUM_PAGES]`, which is accessible from remote nodes using InfiniBand specific functions. Initially, the home nodes of all pages are unknown and initialized to a value of HOME_UNKNOWN.

When first accessing a page, the value `page_home[requested_page]` at the default home node is read, compared with HOME_UNKNOWN and, if equal, replaced by the ID of the requesting node. This is done using InfiniBand’s Atomic-Compare-&-Swap operation. The requesting node can then determine whether the operation was successful by looking at the value read. In case the value is HOME_UNKNOWN, the requester knows that it is now the home node of that page. If the value read is different from HOME_UNKNOWN, it presents the actual home node for the page, resulting in a page fetch operation from that node by issuing an RDMA-Read operation. Obviously, this value is then being stored in the requester’s own `page_home` array, thus future page fetch operations are directly performed on the actual home node without contacting the default home node again. Even the default home node of that page uses the InfiniBand Atomic-Compare-&-Swap operation in order to ensure multiple concurrent Compare-&-Swap accesses to that memory location are executed both in order and atomically.

![Figure 1: Determining the home node of a page](image)

Figure 1 illustrates the two cases that can happen when a node tries to fetch a page from its default home node. When the page’s home is still unknown, it is set to the requester’s ID. When the home is already known, the requester stores this information and subsequently fetches the page from it’s actual home (not shown).
4.2 Applying remote changes to a local page

An important characteristic of HLRC is the *multiple-writer-scheme* - multiple nodes may concurrently write to the same shared page. Actually, different nodes performing write accesses to the same shared page are accessing a local copy of that page previously fetched from the home node. When an interval ends due to a lock acquire or a barrier operation, a node computes all changes it made to remote pages. This is done by comparing the local copy of each changed page with a copy of that page which was created when the page was initially fetched. In the implementation with two threads, these changes are then sent to the home node of that page where they are incorporated into the original page by the request handler thread.

Using Infi niBand’s RDMA-Write operation, these changes can be directly written to the original page at the home node, without invocation of the remote node.

4.3 Requests for locks

Locks allow the programmer to prevent concurrent accesses to the same memory region. A lock can be held by one node at a time only. When a node requests a lock, its execution stalls until the lock is granted. In implementations with a request handler thread, every lock is associated with a specific *default node* that is first contacted when a node attempts to acquire a lock. This default node is always aware of the last node that tried to acquire the lock. This last acquirer is also known to be the currently last node releasing the lock, thus, new incoming requests for locks will be forwarded to that node, with the requester becoming the new last acquiring node. Thus, a *lock-sequence* of nodes trying to acquire a lock is implicitly constructed by the default home node, thereby preventing concurrent execution of commands embraced by an acquire-release-chain for a specific lock.

The last step for the elimination of the request handler thread is to use Infi niBand operations to implement a locking algorithm. Similar to the first access to a shared page, the problem is that the request handler thread forwards requests for a specific lock. Each lock is again associated with a default node determined by a simple modulo calculation. The approach we chose is to place a per-lock data structure on the default node of a lock. This memory region is used to coordinate accesses to the lock.

In consists of two variables accessed by nodes trying to acquire a lock 1:

1. `last_owner[1]`
   This variable contains the last node acquiring lock 1

2. `lock_access[1]`
   This variable is used for locking access to `last_owner[1]` by setting it to either `LOCKED` or `UNLOCKED` using Atomic-Compare-&-Swap.

By accessing and modifying this data structure with a combination of RDMA-Read, RDMA-Write and Atomic-Compare-&-Swap operations, an acquiring node can determine which
node it has to contact for the lock (the last node in the lock-sequence) and place itself at the end of the lock-sequence by putting its own ID into the structure. Then, the acquiring node contacts the formerly last node of the lock-sequence and checks if it has to wait until the lock will be released or if the lock is already available. If the lock is not yet available, it places its ID in a data structure on the remote node and awaits notification that will be sent as soon as the lock is released by the locking node. This check can be performed using the Atomic-Compare-&-Swap operation again. In both cases, the requesting node finally receives invalidation messages by issuing an RDMA-Read operation and continues its operation.

4.4 Further optimizations

Apart from eliminating the request handler thread, additional optimizations have been implemented: Invalidation messages, timestamps and barrier operations are all performed using InfiniBand operations, thus minimizing the amount of time the processor spends for protocol processing. Furthermore, pipelining of messages is used where possible.

Further research will focus on the optimization of the current barrier implementation (see [Hoe05]) and the mprotect system call by only invalidating single TLB (Translation Lookaside Buffer) entries instead of flushing the whole TLB as implemented in the current Linux kernel. These topics are subject part of current research.

5 Results

The process of extending the existing implementation to the AMD64 architecture has been successfully finished. The same applies to the elimination of the request handler thread with further optimizations currently being performed. Preliminary tests have shown that the algorithms used for fetching pages, applying changes to remote pages and the new locking algorithm work correctly. The following measurements were performed on a 36-node cluster of Quad-2.4GHz-Opteron systems connected via 4x-InfiniBand. They illustrate the differences between the implementations described in ViSMI (Virtual Shared Memory for InfiniBand clusters) and HoIBe (HLRC over InfiniBand extended). ViSMI refers to the implementation presented in [Ose04]. It uses InfiniBand for sending and receiving messages, but still uses a request handler thread. HoIBe is the current implementation without a request handler thread. Thus, the measurements illustrate only the speedups achieved by eliminating the request handler thread.

First measurements yield good performance regarding protocol overhead in comparison with previous implementations. The most significant speedup can be observed for page fetch operations.

The current implementation makes use of one computation thread per node only. All tests were performed on multiprocessor systems, thus both threads of ViSMI were scheduled on different CPUs. Using uniprocessor systems or forcing both threads to run on the
same CPU would lead to much higher execution times of ViSMI because of latencies between the arrival of an InfiniBand message and the operating system scheduling the communication thread. Based on current Linux timer interrupt intervals of about 4ms and depending on how many requests are received by the communication thread, execution time would increase significantly. We chose not to force both threads to run on the same CPU for two reasons:

1. Most state-of-the-art cluster systems are comprised of multiprocessor nodes.

2. This problem is specific to RDMA interconnects like InfiniBand where the operating system and the CPU are bypassed on packet arrival because of the interconnect hardware making use of a DMA operation to copy the received data to main memory. Other interconnect technologies like e.g. Ethernet generate an interrupt on packet arrival, thus allowing the operating system to switch to threads waiting for incoming data.

We did, however, run tests with both threads of ViSMI being scheduled on the same CPU that showed strongly varying execution time degradations in the area of 10% up to 70%. Still, having an HLRC implementation without a communication thread makes it much more easier to port HLRC to multiprocessor systems, i.e. each CPU would be assigned one computation thread.

Figure 2 illustrates measurements based on a Fast-Fourier-Transformation of $2^{22}$ complex values for both the overall execution time and the time spent for protocol activity.

Overhead time is measured as the sum of the times spent for protocol activity like fetching pages, barrier operations, sending diffs, acquiring or releasing locks and, in the case of ViSMI, the time spent in the communication thread. The protocol overhead does not include the time spent in the hardware and the operating system for reacting to page faults;
therefore, differences between overhead are not necessarily as high as differences between execution times.

In order to better understand the impact of our changes to execution time and to check proper working of the protocol, a test application was written, benchmarking several common execution scenarios:

In the left illustration in figure 3 the current node becomes the home node for a page whose default home node is not the current node. In the right illustration, changes applied to a cached remote page are sent to its home node.

First, it is far more likely to become the home node of pages with a different default node than the current node with an increasing number of nodes. Second, the performance improvements for becoming the home node of remote pages are clearly greater than the performance degradation for becoming the home node of local pages.

![Graphs showing Remote Page Assignment and Sending Continuous Changes](image)

Figure 3: Remote Page Assignment / Sending continuous changes

The process of releasing and acquiring locks was not benchmarked because it turned out that it played virtually no role in the protocol overhead of all tested applications (less than 0.1% of the overall execution time).

Figure 3 also illustrates how HoIBe and ViSMI perform when sending changes to remote nodes. When sending changes made to a cached remote page that are scattered across that page, ViSMI performs better than HoIBe. ViSMI sends diffs in a pipelined manner; that is, it sends diffs to other nodes but doesn’t wait for them to be applied to the original page; in order to maintain coherence, the communication thread of the home node of that page will not answer requests for the page before all relevant diffs have arrived. This is done using a timestamp mechanism for diffs. For HoIBe, we did not implement that timestamp mechanism because it would result in two messages for fetching a page instead of one which would greatly increase latency on page fetches; instead, HoIBe sends all diffs and waits for them being applied by the means of a completion mechanism called Completion Queue Element. However, strongly scattered changes to a remote page are expected to
appear very rarely in practice.

It must finally be noted that the overall execution time greatly depends on the memory access pattern of a given application. While most applications perform better, some applications tested do not.

Future work will concentrate on researching the impact of sending diffs in a pipelined manner using timestamps and porting the current implementation to support multiprocessor systems by running computation threads on each available CPU.

References


