High-Performance Distributed RMA Locks

Bachelor Thesis
Patrick Schmid
September 30, 2015

Advisors: Prof. Torsten Hoefler, Maciej Besta
Department of Computer Science, ETH Zürich
Abstract

Large-scale data processing is becoming more and more important. One aspect of this trend is that the number and significance of distributed-memory machines used for such computations is growing. Such architectures offer various unique architecture and programming solutions such as Remote Direct Memory Access (RDMA) hardware, Remote Memory Access (RMA) programming schemes, and deep hierarchical memory systems. These emerging multi-core multi-chip machines require novel synchronization schemes to achieve high performance of parallel codes. In this work, we illustrate a design of high performance distributed locks based on RMA. First, we provide an MCS lock that utilizes the knowledge of the architecture of the underlying machine to reduce the number of expensive inter-node data transfers. Second, we illustrate a design of a topology-aware distributed Reader-Writer (RW) lock that trades fairness for higher throughput. The RW lock is based on a novel scheme with one global queue for writers that maximizes locality and reduces inter-node lock passing. The readers on the other hand do not use any queue but mark their presence in the system with a single logical counter that is physically distributed over multiple nodes. This design enables several thresholds that determine a tradeoff between lock fairness and higher concurrency between readers resulting in higher throughput. The evaluation of our schemes on the CSCS Piz Daint (Cray XC30) supercomputing machine illustrates performance advantages over other state-of-the-art locking protocols. Thus, our locks can be used to accelerate deeply parallel workloads in today’s and tomorrow’s large-scale processing.
Contents

1 Introduction 1

2 Background 3
   2.1 The Design of Supercomputers 3
   2.2 Traditional NUMA-oblivious Locks 5
      2.2.1 Reader-Writer Locks 5
      2.2.2 MCS Locks 5
   2.3 State-of-the-art NUMA-aware Locks 6
      2.3.1 Hierarchical Locks 6
      2.3.2 NUMA-aware Reader-Writer Locks 7
   2.4 RMA Programming 8
      2.4.1 Utilized RMA Functions 8

3 RMA-based MCS Lock 11
   3.1 Differences to Traditional MCS Lock 11
   3.2 Utilized Data Structures 11
   3.3 RMA-based MCS Lock Implementation 12
      3.3.1 MCS Lock Initialization 12
      3.3.2 MCS Lock Acquire 12
      3.3.3 MCS Lock Release 13
      3.3.4 MCS Lock Finalization 14

4 Distributed Topology-aware Reader-Writer Lock 15
   4.1 The Intuition behind the Design 15
   4.2 Key Data Structures 16
      4.2.1 Design of a Counter 16
      4.2.2 A Queue of Writers 17
      4.2.3 A Tree of Queues 18
## Contents

4.3 Distributed Reader-Writer Protocol ........................................ 19
   4.3.1 Writer Lock Acquisition Level 2 to N .......................... 19
   4.3.2 Writer Lock Release Level 2 to N ............................... 21
   4.3.3 Writer Lock Acquisition Level 1 ................................. 22
   4.3.4 Writer Lock Release Level 1 ..................................... 24
   4.3.5 Reader Lock Acquisition ......................................... 27
   4.3.6 Reader Lock Release ............................................. 28

5 Evaluation ................................................................. 31
   5.1 Experimental Setup .................................................... 31
       5.1.1 Implementation Details ....................................... 32
       5.1.2 Benchmarks .................................................... 32
   5.2 RMA-based MCS Lock ................................................ 33
   5.3 Distributed Reader Writer Lock ..................................... 35
       5.3.1 Counter Distance .............................................. 36
       5.3.2 Writer Thresholds ............................................. 37
       5.3.3 Reader Threshold .............................................. 40
       5.3.4 Reader/Writer Rate ........................................... 41
       5.3.5 Comparison to the State-of-the-Art ......................... 42

6 Related Work .............................................................. 47

7 Conclusion ................................................................. 49

A Appendix ................................................................. 51
   A.1 Counter Distance ..................................................... 51
   A.2 Writer Thresholds .................................................... 53
   A.3 Reader Threshold ..................................................... 56
   A.4 Reader/Writer Rate .................................................. 60
   A.5 Comparison to the State-of-the-Art ............................... 62

Bibliography ................................................................. 63
Chapter 1

Introduction

The scale of today’s data processing is growing steadily. For example, the size of Facebook’s social graph is many petabytes [35] and the largest graphs processed by the well-known HPC benchmark Graph500 [28] have \(2^{42}\) vertices. Efficient analyses of such amounts of data require large distributed-memory massively parallel machines that have deep Non-Uniform Memory Access (NUMA) hierarchies and span multiple compute nodes.

Developing codes for such machines requires efficient synchronization mechanisms. Fine locks are among the most popular schemes. On one hand, they require some level of expertise from a developer to avoid risks such as deadlocks [15]. However, they have intuitive semantics and, when properly used, they outperform other schemes such as atomic operations or transactions.

Designing efficient locks for machines with deep hierarchical memory systems is challenging. Consider four threads competing for the same lock. Assume that two of them run on one CPU and the remaining two execute on the other one. Now, in a naive lock design oblivious to the underlying memory hierarchy, the lock may be passed between different CPUs up to three times, degrading performance. Recent advances [5, 11] tackle this problem by ordering threads grabbing the lock so that the amount of expensive inter-CPU communication is reduced. In the above example, the first thread to grab the lock would pass it to the other thread from the same CPU. Ultimately, his would result in only one inter-CPU lock transfer. However, to the best of our knowledge, no previous scheme targets distributed memory machines that span multiple compute nodes. This adds another complexity dimension due to the lack of automatic data coherence across the nodes. In addition, inter-node data transfers are significantly more expensive than any intra-node communication, calling for even more aggressive communication-avoidance strategies. Finally, previous NUMA-aware designs cannot be straightforwardly used in distributed-memory codes.
An important aspect of developing locking mechanisms for large-scale data processing is to properly match the type of the utilized lock to the targeted workload. It turns out that the majority of accesses in various analytics workloads are reads (e.g., 99.8% of requests to the Facebook graph are read requests [35]). Here, simple locks would entail unnecessary synchronization overheads. Instead, the Reader-Writer (RW) lock [26] is a more proper match. It distinguishes between processes that modify data in the critical section between those that only perform reads to reduce the amount of synchronization for readers. Initial RW NUMA-aware lock designs have recently been introduced [4] but they do not address distributed-memory machines.

Another challenge in developing distributed-memory schemes is to properly utilize state-of-the-art hardware mechanisms that provide significant performance advantages. Remote Direct Memory Access (RDMA) [30] is one of such schemes; it removes the OS and the CPU from the inter-node communication path. RDMA is the basis of Remote Memory Access (RMA) [12] programming models. They offer a Partitioned Global Address Space (PGAS) abstraction to the programmer and enable low-overhead one-sided access to remote memories with put/get communication primitives. RMA principles are implemented by various high-performance languages and libraries: Unified Parallel C (UPC) [34], Fortran 2008 [20], MPI-3 [27], or SHMEM [3]. We will illustrate how to utilize RMA principles to design high performance locking mechanisms for distributed-memory machines.

In summary, our key contributions are as follows:

- We illustrate the design of an RMA-based MCS lock for distributed-memory machines. Our lock matches the interface of the MPI-3 RMA specification.
- We describe an RMA-based distributed Reader-Writer lock that is aware of the underlying topology and uses this knowledge for trading fairness for higher throughput.
- We improve the implementation of the utilized MPI-3 RMA library foMPI [12] by seamlessly incorporating the distributed MCS lock into its design. The new lock implementation outperforms the previous scheme by 350% on average. We also improve the implementation of an atomic operation Fetch-And-Replace provided by foMPI by utilizing hardware-supported Cray communication primitives.
- We evaluate our design and illustrate its advantages over the state-of-the-art in both throughout (a mean of 755% over foMPI) and latency (a mean of 685% over foMPI).
Chapter 2

Background

We start by presenting traditional (§ 2.2) and state-of-the-art (§ 2.3) locks that we will later use and extend in our design. We will then (§ 2.4) present RMA programming.

2.1 The Design of Supercomputers

The term supercomputer was created for a computer with a high computational power compared to a computer that an average consumer uses. Today's supercomputers consists of thousands of processors that are placed in close proximity to each other like for example in a computer cluster. Therefore, the processors can be arranged to different architectures in order to make them work together as efficiently as possible and reduce the time for communication to a minimum. One approach can be seen in Figure 2.1 where the cores are placed on compute nodes. A blade consists of several nodes and multiple blades are situated on a chassis. The chassis in turn are put into cabinets. Multiple cabinets can be linked together to compose a supercomputer. On each level of this hardware hierarchy, a shared memory can be placed that all processes, which work on this piece of hardware, can use. Therefore, we have a memory hierarchy that is build up on the underlying hardware architecture. But there is a downside of this design that programmers have to consider. Each core has its own cache with their local copies of data that resides in the memory hierarchy. As soon as a process modifies the local copy, the cache of every other core that holds a copy of this piece of data is not coherent anymore and needs to be updated.

Because of the physical separation of hardware processes, computers organize processes into clusters known as NUMA domains. Two NUMA domains are close to each other if they are on the same hardware and their processes work on the same hardware on the NUMA domain above as well. For example two nodes that are on the same blade. In these systems, mem-
memory access latencies depend on the level of the memory hierarchy where the data has its home location and where it resides at the moment. This location is determined by the level of the process that last wrote to it. Thus, the proximity of a process to a memory location is increased when the process itself or one on the same NUMA domain accesses the memory location. On the contrary, proximity is decreased as soon as a process of another NUMA domain accesses the data. Ideally, one or multiple processes of the same
2.2 Traditional NUMA-oblivious Locks

2.2.1 Reader-Writer Locks

There exist many types of locking mechanisms. They range from a simple spin-lock to queue-based designs like the MCS lock that will be discussed later. One of them presents a very interesting solution to the problem of synchronizing concurrent operations and accesses by handling reads differently than writes. Thus, it is called the Reader-Writer (RW) or shared-exclusive lock. RW locks have a wide field of application. They are used in operating system kernels, databases and high-end scientific computing applications. Hence, RW locks have been studied thoroughly [8, 18, 21, 26]. The reason for the separation between readers and writers is that readers do not change the data but writers do. As a consequence readers can concurrently access data with a so called shared lock. With this approach, the lock can have a much higher number of grabbing and releasing a lock per second (which is called the throughput of a lock) than other lock designs where only one process at a time can access the critical section. However, the writers have to be treated differently than the readers. The problem is that they cause concurrency issues like race conditions where multiple processes modify a memory location at the same time. A programmer cannot influence which process is first and therefore the outcome depends on a sequence and timing of uncontrollable events. Locking algorithms avoid such situations by allowing only one process to access the “critical” section. An RW lock constrains only the writers to have such an exclusive access. In the meantime, all other writers and readers must wait for the duration of the lock. That is why it is also called an exclusive lock.

2.2.2 MCS Locks

Another lock algorithm is the MCS lock designed by Mellor-Crummey and Scott [25] that ensures First In, First Out (FIFO) ordering. It is an established and well-known design [31, 32] and therefore we used it as a base for our lock scheme. It has a high speedup over previous locking algorithms that are based on spin-waiting [1]. The processes, which want to grab a spin-lock, simply wait in a loop and repeatedly check whether the lock is available. The problem of these locks is that each spin-waiting process has a local copy of the global field in their cache that shows whether the lock is available or not. As soon as the lock holder modifies the field to tell it has finished,
everybody else’s cache entry will be invalidated. This causes a big amount of coherence traffic on the interconnecting bus. On the other hand, the MCS lock algorithm basically builds a queue of processes that want to grab the lock and the process at the head holds the lock. Each process in the queue provides a local flag and a pointer that points to the next process in the queue. For enqueueing a process needs to check a global field that points to the current tail of the queue and change it so that it points to the process itself. After that the process modifies the pointer of the predecessor to let it know who its successor is. When the lock holder wants to release the lock, it notifies its successor by changing the successor’s local field. A picture of an example MCS queue lock is given in Figure 2.2. The big advantage of this design over spin-locks is that every process spin-waits on a local flag instead of a global one. Overall, it uses only \( O(p + n) \) of space for \( p \) processes and \( n \) locks. In addition, it still works even if the machine does not always guarantee coherent caches.

![Figure 2.2: An example of an MCS queue.](image)

2.3 State-of-the-art NUMA-aware Locks

2.3.1 Hierarchical Locks

There has already been done some research about locks that are NUMA-aware [5, 10, 23, 29]. Most of them just consider two levels of a NUMA system. As far as we know, there is currently only one approach that exploits the locality of a multi-level system [5]. Since it has an MCS queue lock on each level in the memory hierarchy, we call it \( hMCS \). By using a queue on each NUMA domain it allows processes on the same domain grabbing the lock consecutively even if processes on other domains are waiting. This is done because passing the lock to a successor on the same NUMA domain is less expensive than passing it to a process on another domain. Figure 2.3 shows an example for a system that has three NUMA hierarchy levels. A process starts to acquire the global lock by enqueueing itself in the lowest level queue. If there is a predecessor, it will wait until it gets notified by the predecessor like in the MCS algorithm. Otherwise, the process is the first in the queue of this level and thus already holds the lock on this level and proceeds by enqueueing itself in the next level. This is done as long as the
2.3. State-of-the-art NUMA-aware Locks

top most level lock has been obtained. After finishing this, the process holds
the global lock and can proceed to its critical section. The release is almost
the same as in the MCS lock. If there is a predecessor on the same level, the
lock will be passed to it. But lacking a successor the lock has to be released
to the parent level.

2.3.2 NUMA-aware Reader-Writer Locks

Most of the proposed solutions in section 2.2.1 rely on a centralized structure
to coordinate processes. However, for high performance computers with a
deep NUMA memory hierarchy, this is not practical because of two reasons.
First, supercomputers have a high number of processes and if all of them
have to access one centralized data structure, the scalability is impaired.
Second, a centralized data structure needs to be saved somewhere in the
system. But this means that a lot of processes have a high communication
latency to access this data. It would be much better to have the data structure
on the same NUMA domain. Lev et al. presented an approach that is
based on a decentralized data structure called Scalable Non-Zero Indicator
(SNZI) [22]. They solve the problem by using a tree where each reader
arrives at one leaf. This tree is built up on the NUMA nodes so that every
process arrives at SNZI leaves associated with their node. On the other side,
writers are kept NUMA-oblivious and this again causes scalability issues.
Calciu et al. tackle this problem [4]. They present a NUMA-aware RW
lock that is based on the lock cohorting technique [11]. The writers first
need to acquire the cohort lock. Then before they execute the critical section
they have to ensure that there is no concurrent reader executing or about to
execute their critical section. The readers use a so called read indicator that
shows per node whether a reader is active or not. But since it is based on
lock cohorting, it only exploits two levels of a NUMA system. As already
stated, this will not be enough for the future.
2. Background

2.4 RMA Programming

In the RMA programming model, processes communicate by directly accessing one another’s memories. Usually, RMA is built over OS-bypass RDMA hardware for highest performance. Thus, RMA puts (writes to remote memories) and gets (reads from remote memories) offer low latencies, significantly improving performance over message passing communication [12]. RDMA is provided in virtually all modern networks (e.g., IBM PERCS [2], IBM’s Cell on-chip network, InfiniBand [33], iWARPi [13], and RoCE [19]). Moreover, numerous libraries and languages based on RMA offer unique features for parallel programming. Examples include MPI-3 RMA, UPC, Titanium [16], Fortran 2008, X10 [7], or Chapel [6]. Thus, the number of codes built with RMA is growing steadily.

In RMA, each process explicitly exposes an area of its local memory as shared. In MPI RMA, this region is called a window. Each process has a private and a public window. The process itself can modify the private and the public window but all other remote processes can only make changes to the public window. To ensure the consistency of these windows, they need to be synchronized. Once shared, a window can be accessed with various language-specific operations.

Another important value, which is needed for RMA, is the rank of an MPI-process. It is used to identify each process in an MPI communication environment and therefore is unique for each process.

There are two basic types of RMA operations: communication actions (often called accesses; they transfer data between processes), and synchronization actions (synchronize processes and guarantee memory consistency).

2.4.1 Utilized RMA Functions

We now shortly explain the utilized RMA functions. We show here simplified versions of the functions because we omit the details for clarity. As stated before, there are two categories of RMA operations. First, we describe the communication actions:

- MPI_Get fetches a piece of data on a remote MPI-process at the specified target location. This is only possible if the data is in the memory window of the target process.
- The counterpart of MPI_Get is MPI_Put which is essentially a remote write. Therefore, it puts the given data to the specified location in the memory window of the target process.
- The function MPI_Accumulate executes a specified operation at the target location with a given operand. We utilize the operations
MPI_REPLACE that replaces the target location with the operand and MPI_SUM that adds the operand to the target location.

- An extension of MPI_Accumulate is the function MPI_Fetch_and_op that atomically fetches first a piece of data at the target and then accumulates the operand to the target location with the declared operation.

- MPI_Compare_and_swap first checks whether the target memory location is equal to a compare value and if yes, swaps it with a given data. Moreover, it returns the value of the target location before the swap.

Below are the definitions of the functions:

```c
/* rank_of_target: Rank of the target from which the data is fetched */
* offset: Offset of the memory location that is fetched at the target’s window */
int64_t MPI_Get(int rank_of_target, int offset);

/* source_data: Data that is written to the target’s window */
* rank_of_target: Rank of the target on which the data is put */
* offset: Offset of the memory location where the data is put at the target’s window */
void MPI_Put(int64_t source_data, int rank_of_target, int offset);

/* operand: Data that gets accumulated to the target’s window */
* rank_of_target: Rank of the target on which the accumulation takes place */
* offset: Offset of the memory location that is used for the accumulation at the target’s window */
* operation: Operation that is used for the accumulation */
void MPI_Accumulate(int64_t operand, int rank_of_target, int offset, MPI_Op operation);
```

/* operand: Data that gets accumulated to the target’s window */
* rank_of_target: Rank of the target on which the operation takes place */

2. Background

To synchronize processes and guarantee memory consistency we use one synchronization operation: MPI_Win_flush. MPI_Win_flush completes all outstanding RMA operations that have been started by the calling process to the target rank. This holds for the source and the target. The definition of this function is the following:

```c
void mpi_win_flush(int rank_of_target);
```

Listing 2.2: Definition of the synchronization action
Chapter 3

RMA-based MCS Lock

We start the discussion of the distributed-memory locking schemes with an MCS distributed lock based on MPI-3 RMA (RMA-MCS). First, we show the differences to the traditional MCS lock (§ 3.1). Afterwards, we depict the utilized data structures (§ 3.2). At last, we explain in detail the implementation of such a RMA-MCS (§ 3.3).

3.1 Differences to Traditional MCS Lock

There are two main differences to the traditional MCS lock implementations that we described in section 2.2.2. In the first step, we have to adapt the MCS queue design to a distributed environment that consists of multiple compute nodes. Second, we need to incorporate RMA functions responsible for memory synchronization. One has to control it explicitly as there is no automatic data coherence provided by RMA.

3.2 Utilized Data Structures

We now discuss the data structures that are used by the lock schemes. They are all included in a RMA window and thus each process exposes their local instances to all other involved processes. First, we use a pointer to the tail of the MCS queue, located on a selected node. The rank of this node is saved in the field \texttt{rankTail}. Furthermore, we use another pointer at each process to the next process in queue. Finally, there is a variable (also at each process) that indicates whether a given process has to spin wait. We call it the \texttt{blocked} value of a process.

Each pointer consists of two variables. The first one is the process \texttt{rank}: it identifies a specific process. The other one is an address within the given process’ address space. Such a hierarchical structure corresponds to the hierarchical nature of the distributed environment.
Each lock must be initialized at the beginning of a program, only then can it be acquired and released. The allocated memory must be freed before the program termination. In Listing 3.1 one can see the constants that are used in the code. We now describe these four functions in more detail.

Listing 3.1: Constants used for the MCS RMA lock

```c
1/* Offsets for the queueNode array */
2enum {NEXT = 0, BLOCKED = 1};
```

3.3 RMA-based MCS Lock Implementation

3.3.1 MCS Lock Initialization

First, a process prepares its part of the memory window with the related variables so that other processes can access it. This is not a part of the traditional MCS design and is required due to the utilization of RMA. After that, we initialize the above mentioned three variables that constitute the allocated memory region. The value for indicating whether the process is blocked can be set to false. The initialization is only called once and therefore has no impact on the performance of acquiring and releasing the lock.

3.3.2 MCS Lock Acquire

After the initialization has been finished, each process can attempt to acquire the lock. The first step of the acquisition is to atomically fetch on the designated node the rank of the last in queue and replace it with its own rank. Fortunately, there is the powerful operation MPI_Fetch_And_Op in MPI-3.0 that exactly does these two things in one go. If there is no predecessor, the lock is acquired successfully. Otherwise, the processor has to put his rank into the next field of his predecessor with the function MPI_Put. Afterwards, it has to wait until its blocked value is set to 1. Below is the code listed for acquiring a lock:

```c
1int64_t predecessor, blocked;
2queueNode[BLOCKED] = 1;
3queueNode[NEXT] = -1;
4
5/* We add ourselves to the end of the queue */
6predecessor = MPI_Fetch_and_op(rank, rankTail, TAIL);
7MPI_Win_flush(rankTail);
8/* We check if we have a predecessor */
9if(predecessor != -1)
10{
```
3.3. RMA-based MCS Lock Implementation

```c
/* We make ourselves visible to the predecessor */
MPI_Put(rank, predecessor, NEXT);
MPI_Win_flush(predecessor);
/* Now spin on our local value ‘‘blocked’’ until we are given the lock */
do {
    blocked = MPI_Get(rank, BLOCKED);
    MPI_Win_flush(rank);
} while (blocked == 1);
```

Listing 3.2: Acquiring an MCS RMA lock

### 3.3.3 MCS Lock Release

If there is already a successor waiting at the point of releasing the lock, the passing can be done with one line of code. The blocked value of the next in the queue must be changed to false to unblock him. This is achieved with the function MPI_Put.

But if there is no successor, it is a bit more complex. The releaser has to atomically look whether his rank is still saved as the tail of the queue and if yes, change the tail field to -1 and then it is done. The operation that exactly does this is the MPI.Compare_And_Swap.

If the CAS is not successful, there is another process in the queue but has not been able to tell its rank. Thus, the process has to wait until that happens and then can unblock his successor. For this is again the function MPI_Put applied. The pseudo-code for releasing a lock looks as follows:

```c
int64_t currentRank, successor;
if (queueNode[NEXT] == -1) {
    /* See if we are waiting for the next to notify us */
    currentRank = MPI_Compare_and_swap(-1, rank, rankTail, TAIL);
    /* We have to ensure that MPI_Compare_and_swap has finished */
    MPI_Win_flush(rankTail);
    /* We check whether we are the only one in the queue */
    if (rank == currentRank) {
```
3. RMA-based MCS Lock

```c
/* We are the only process in the list */
return;

/* We wait until the successor makes itself visible */
do {
  successor = MPI_Get(rank, NEXT);
  MPI_Win_flush(rank);
} while (successor == -1);

/* Now we can notify the successor */
MPI_Put(0, successor, BLOCKED);
MPI_Win_flush(successor);
```

Listing 3.3: Releasing an MCS RMA lock

### 3.3.4 MCS Lock Finalization

In the beginning, we have initialized a memory region for communication. Now we have to free it again explicitly. Like the initialization this function is called only once and should be called at the end as the acquiring and releasing of locks does not work anymore after it.
Chapter 4

Distributed Topology-aware Reader-Writer Lock

In this chapter, we present the design of a distributed topology-aware Reader-Writer lock (RMA-RW). We start by explaining the intuition behind the scheme (§ 4.1). Later, we discuss the utilized data structures (§ 4.2) and provide more details on how the lock is acquired and released (§ 4.3).

4.1 The Intuition behind the Design

We now depict an implementation of the RMA-RW lock. There are two core ideas behind it. Firstly, we want to have a global queue throughout the network. This is not trivial because high performance computers are organized as clusters known as NUMA domains as we already stated. We used the hierarchical MCS lock [5] as a starting point for our queue design. We already have given a short description of it in section 2.3.1. We adopted it so that each NUMA level has its own queue and in order to acquire a lock, a node has to obtain the lock in every level. In Figure 4.1, we provide an example for a two level system. The second idea is based on the RW synchronization primitive that we discussed in section 2.2.1 and 2.3.2. Thus, readers are allowed to simultaneously hold the lock whereas only writers can acquire a lock for exclusive access. Considering these two concepts, we decided to have only a queue for writers that is based on a hierarchy like the hMCS lock. It is not suitable to have readers in this queue because it would require more communication between the nodes. Because of this decision, we need to have a data structure that regulates the struggle between readers and writers. Principally, it shows in which mode the lock is – either READ or WRITE. It has however one more feature. It can count how many readers have arrived at the critical section and how many have departed from it.
4. Distributed Topology-aware Reader-Writer Lock

![Diagram of multi-level queues on a two-level system.]

Figure 4.1: An example of multi-level queues on a two-level system.

4.2 Key Data Structures

We use three key data structures in our implementation. One of them is needed to control the struggle between readers and writers. Ideally, it should be decentralized since we want that not all readers have to check the same memory location. Moreover, it should make the readers topology-aware. Another structure is in charge of keeping the fairness between the writers because they only can execute the critical section one after another. Lastly, we want that the writers are NUMA-aware, too. For that purpose we have the last data structure. It uses the data structure that keeps the fairness between the writers multiple times – one per NUMA domain. This should reduce the lock passing latency between writers. We now describe each one of them in more detail.

4.2.1 Design of a Counter

We need a mechanism that enables each process to know whether a writer or many readers are active. Calciu et al. [4] presented a solution which “splits” the logical counter into multiple physical counters, one on each NUMA node. We extend this approach with the difference that we can choose how many counters reside on each NUMA node. The idea is that one can adjust the number of counters so that it exploits the underlying memory hierarchy at its best. This can even lead to having a counter not on every NUMA node but also every second or third. In any case, a reader lock acquirer only has to check one of these counters. On the other hand, a writer lock acquirer needs to know if there is a reader somewhere in the system holding a lock. This means that a writer checks each counter for active readers and then sets it to WRITE if there is none. For that case, we expand the design of the counter into two different fields. The first one counts the readers that arrive at the critical section and the second one counts the readers that depart from the
4.2. Key Data Structures

critical section. One can find out how many readers are active by taking the
difference of these two counters. There are two reasons for this design. On
one hand, one can easily find out how many readers have acquired a lock
since the last mode change. On the other hand, it reduces the contention
between acquirers and releasers and they can independently increase their
counter. But for that they have to lie on different cache lines. While releasing
the lock, a writer needs to change the counters back to READ to make sure
that waiting readers can acquire the lock. There is therefore a trade-off
between how easily a reader can check a counter and how many counters a
writer has to access.

However, we still need to define the values representing WRITE and READ.
Since we have two counters that are both a 64-bit integer, we choose the
counter for the arriving readers to indicate the lock mode as well. We will
later define that readers can add one to the arrival counter before checking
whether the mode is READ thus it can happen that readers increase the
counter even when the mode is WRITE. That means that we need to have
a big enough gap between the value of WRITE and the maximum value
of a 64-bit integer. We choose it therefore to be one half of the maximum
value of a 64-bit integer. As a consequence, every number smaller than
the value of WRITE represents the lock mode READ. To ensure that the
variable does not overflow, we introduce a constant READER_THRESHOLD
that regulates after how many reader lock acquisitions the lock gets passed
to the writers. We reset the counters every time this threshold for readers is
reached or when no reader is active but writers are waiting instead. In the
latter option, the writers obtain the lock and the last writer resets the counter
before the lock gets passed back to the readers. Thus, the threshold is never
reached. We set the threshold to a number that fits into a 32-bit integer and
our counters are 64-bit integer each. As a consequence is no overflows are
possible for the counters as desired.

4.2.2 A Queue of Writers

For the writers we would like to have a data structure that guarantees FIFO
ordering as good as possible and still exploits the locality of memory in a
NUMA system. We think that a queue best fits as such a data structure be-
cause we can have multiple queues on different levels of a NUMA hierarchy.
Thus, we can guarantee local FIFO ordering and the passing of a lock on
the same NUMA domain lowers the communication costs. We have indeed
already defined such a queue data structure in the Section 3.1 and we can
adopt it in this scheme. However, two major changes should be made. First,
we do not need the lockTail variable on any process since we will provide it
in the data structure in the next section. Second, we expand the use of the
field status so that each process can pass the count of locks that have been ac-
4. Distributed Topology-aware Reader-Writer Lock

required in this queue to his successor. Therefore, we introduce a value \textit{WAIT} that is the maximum value of a 64-bit integer. Everything smaller than that number indicates that the process does not have to wait for the lock. We use this extension to introduce a threshold for each node. Every time the count gets bigger than the threshold, the lock of the queue one level above in the hierarchy is released as well. This is a mechanism that ensures that no NUMA domain constantly keeps the lock which would result in a starvation of the processes on all other NUMA domains. Furthermore, we need a value that shows that a lock mode change has happened. It is only applied on level 1 in the hierarchy and therefore it will be further explained in the Sections 4.3.3 and 4.3.4. We call it \textit{MODE\_CHANGE} and it is one smaller than the maximum value of a 64-bit integer.

Since we extend the data structure from the RMA-MCS lock, we name the construct of these two fields \textit{queueNode} as well and use an array to represent it.

4.2.3 A Tree of Queues

The second key data structure is used to adopt the queue previously described to a NUMA hierarchy. As pointed out before, we want to have a queue on each level of the NUMA architecture. To accomplish that, we need a representational tree of the hierarchy. It begins at level 0 that is one level more than the number of levels in the memory hierarchy. We need this additional level since we want to have a queue on level 1 as well. On the level \(N\), the processor are building a queue. Each node on an arbitrary level \(i\) between 0 and \(N\) has the following information:

- **parent**: This is a pointer to the hierarchical parent node on level \(i - 1\). This pointer is not used on level 0.

- **lockNode**: The \textit{queueNode} which can be used for the queue on level \(i\).

- **tail**: This is a pointer to the MPI-process that is last in the queue on level \(i + 1\).

- **threshold**: Controls after how many lock acquisitions on level \(i\) the lock gets released to the parent level.

The underlying data structure is an array. We call this data structure \textit{hierarchyNode} because it holds the information about the hierarchy. It is similar to the \textit{HNode} of the hMCS algorithm [5] but we adopted it so that we can use it with RMA. Since the tail for the queue of level \(i\) is saved on the \textit{hierarchyNode} on level \(i - 1\), we need to make sure that each process knows the rank of the process where the \textit{hierarchyNode} is residing for each level. We provide the rank as the input parameter \textit{rankHierarchyParent}.
4.3 Distributed Reader-Writer Protocol

In this section, we explain how the readers acquire and release a shared lock and the writers an exclusive lock. A writer always starts at the leaf of the tree (level N) both for acquiring and releasing. On an arbitrary level i between 2 and N, a process executes the corresponding function to acquire (§ 4.3.1) and release (§ 4.3.2) a lock. If it reaches level 1, the process executes different operations (§ 4.3.3, § 4.3.4) since a writer resolves the conflict between readers and writers on this level. Readers do not have a hierarchy and therefore always execute the same function for the acquisition (§ 4.3.5) and release (§ 4.3.6) of a lock. In Listing 4.1 we declare all constants that we use and in Listing 4.2 we provide the declarations of the three utilized data structures.

1 /* Constants used for passing information between the writers */
2 enum {COHORT_START=0, CHANGE_MODE = INT64_MAX -1,
3 ACQUIRE_PARENT = INT64_MAX - 2, WAIT = INT64_MAX};
4 /* Constants used for solving the conflict between readers and writers */
5 enum {WRITE = INT64_MAX / 2, START_READ = 0};
6 /* Offsets for the queueNode array */
7 enum {NEXT = 0, STATUS = 1};
8 /* Offsets for the hierarchyNode array */
9 enum {PARENT = 0, LOCK_NODE = 1, TAIL = 2, THRESHOLD = 3};
10 /* Offsets for the counter array */
11 enum {ARRIVE = 0, DEPART = 1};

Listing 4.1: Declaration of the used constants

1 int64_t queueNode[2];
2 int64_t hierarchyNode[4];
3 int64_t counter[2];

Listing 4.2: Declaration of queueNode and hierarchyNode

4.3.1 Writer Lock Acquisition Level 2 to N

In Listing 4.3 the pseudo-code for this function is given. On each arbitrary level i between 2 and N, a process enqueues himself by using the function MPI_Fetch_and_replace (line 6). After that we need to call the function MPI_Win_flush (line 8) in order to make sure that the fetch and op has finished. The first in the queue on level i holds the local MCS lock and proceeds to acquire the lock for the level i - 1.

If there is a predecessor on level i, the process makes itself visible to the predecessor by using the function MPI_Put (line 14) and then waits until the
lock gets passed. While waiting the process needs to synchronize its private copy of the window with the public so that it will notify any changes to it. After that, there are two possibilities on how to proceed: if the predecessor has hit the threshold for level \( i \) and released the lock to the parent level, the waiting process has to acquire it again or the acquisition is finished and the critical section can be entered.

If there is no predecessor on level \( i \), the process proceeds to acquire the lock on level \( i - 1 \).

```c
int64_t predecessor, currentStatus;
queueNode[NEXT] = -1; queueNode[STATUS] = WAIT;

/* We place ourselves at the tail of queue at this level */
predecessor = MPI_Fetch_and_op(rank,
    rankHierarchyParent, TAIL, MPI_REPLACE);
/* We have to ensure that MPI_Fetch_and_op has finished */
MPI_Win_flush(rankHierarchyParent);

/* We check if we have a predecessor */
if(predecessor != -1)
{
    /* We make ourselves visible to the predecessor */
    MPI_Put(rank, predecessor, NEXT);
    MPI_Win_flush(predecessor);
    /* We wait until predecessor passes lock */
    do
    {
        currentStatus = MPI_Get(rank, STATUS);
        MPI_Win_flush(rank);
    } while(currentStatus == WAIT)
    /* Check if predecessor released the lock to parent level */
    if(currentStatus != ACQUIRE_PARENT)
    {
        /* Acquired the global lock */
        return;
    }
}
/* We need to acquire parent level lock either because we are the first on this level or the predecessor has hit the threshold */
4.3. Distributed Reader-Writer Protocol

Pseudo-code for this function is provided in Listing 4.4. For the release of a writer lock on an arbitrary level $i$ between 2 and $N$, a process first has to find out whether there is a successor. If there is one and the threshold for level $i$ is not already reached, the lock can be passed to him by executing an MPI_Put (line 8).

In case that there is no successor or the threshold is reached, the writer releases the lock for this level and tries to inform his successor that it has to acquire the level $i-1$ lock.

If there is no known successor, we need to make sure that no one already enqueued himself in the queue on this level. Therefore, the process checks whether it is still the only one in the queue by using the MPI_Compare_and_swap (line 21) operation on the tail. If it indeed is the only one, the release is done. Otherwise, the process has to wait until the successor makes himself known. This is only for a short moment since a process only executes an MPI_Win_flush between enqueueing and making itself visible to its successor. Then the successor of the process gets the notice that it needs to acquire the parent level lock.

Listing 4.3: Acquiring an RMA-RW writer lock for level 2 to $N$

```c

Listing 4.3: Acquiring an RMA-RW writer lock for level 2 to $N$

```
4. Distributed Topology-aware Reader-Writer Lock

Listing 4.4: Releasing an RMA-RW writer lock for level 2 to N

4.3.3 Writer Lock Acquisition Level 1

An example code for this operation is given in Listing 4.7. At this level, a process either gets the lock from a predecessor or if there is no one, it tries to get the lock from the readers by changing the mode to WRITE. In the latter case, it finishes the acquisition successfully by executing the function `Check_And_change_counters` (line 26). This function checks first each counter for active readers (Listing 4.5). If there are none, it switches the value of each counter to WRITE (Listing 4.6). In the check and the change function, the field `worldSize` is the number of MPI-processes and the `COUNTER_DISTANCE` holds the value for the counter distance that one can specify as input.

After changing all counters, the process has again to wait until no readers are active anymore. There is twice the check for active readers because it could be that while the writer is changing the counters, a reader acquires
4.3. Distributed Reader-Writer Protocol

the lock from a counter that is not already modified to WRITE. Thus, the
writer has to defer until that reader has finished as well.

However, if there is a predecessor, the process has to wait until the lock gets
passed. Nevertheless, it can happen that the predecessor has handed the
lock over to the readers. Therefore, the mode has to be changed back to
WRITE before entering the critical section. The change is again issued by
the function Check_And_change_counters (line 33).

```c
int i;
int64_t arriveCount, departCount;

for(i = 0; i < worldSize; i += COUNTER_DISTANCE)
{
    /* We get the two counts of this counter */
    arriveCount = MPI_Get(i, ARRIVE);
    departCount = MPI_Get(i, DEPART);
    MPI_Win_flush(i);
    /* We check whether no reader is active when we
     * already changed the counters to WRITE */
    if(arriveCount - departCount - WRITE == 0)
    {
        break;
    }
}
```

Listing 4.5: Function that checks each counter for active readers

```c
int i;
for(i = 0; i < worldSize; i += COUNTER_DISTANCE)
{
    /* We add WRITE to the arrival counter so that the
     * readers get blocked */
    MPI_Accumulate(WRITE, i, ARRIVE, MPI_SUM);
    MPI_Win_flush(i);
}
```

Listing 4.6: Function that changes each counter to WRITE

```c
int64_t predecessor, currentStatus;
queueNode[NEXT] = -1; queueNode[STATUS] = WAIT;
```
4. Distributed Topology-aware Reader-Writer Lock

5 /* We put ourselves at end of queue */
6 predecessor = MPI_Fetch_and_op(rank,
    rankHierarchyParent, TAIL, MPI_REPLACE);
7 /* We have to ensure that MPI_Fetch_and_op has
     finished */
8 MPI_Win_flush(rankHierarchyParent);
9
10 /* We check if we have a predecessor */
11 if(predecessor != -1)
12 {
13    /* We make ourselves visible to the predecessor */
14    MPI_Put(rank, predecessor, NEXT);
15    MPI_Win_flush(predecessor);
16    /* We wait until predecessor notifies us */
17    do
18    {
19        currentStatus = MPI_Get(rank, STATUS);
20        MPI_Win_flush(rank);
21    } while (currentStatus == WAIT)
22    /* Check if predecessor passed lock to the readers
     */
23    if(currentStatus == MODE_CHANGE)
24    {
25        /* The predecessor passed lock to the readers
             thus try to get it back */
26        Check_And_change_counters(WRITE);
27        queueNode[STATUS] = COHORT_START;
28    }
29 }
30 else
31 {
32    /* We acquired the lock on level 1, now we have to
       change the counters to WRITE */
33    Check_And_change_counters(WRITE);
34    queueNode[STATUS] = COHORT_START;
35 }

Listing 4.7: Acquiring an RMA-RW writer lock for level 1

4.3.4 Writer Lock Release Level 1

Listing 4.9 shows the pseudo-code for this sequence. At the root, a process
has to check whether the threshold is reached for writers. If so, the lock will
be passed to the readers by resetting the counters (line 9). A code for the
function Reset_Counter that resets one counter is given in Listing 4.8. The
function \emph{Reset Counters} is calling \emph{Reset Counter} on each counter. The rank of the process where the counter is residing is given as the input parameter \emph{rankCounter}. After resetting, we change the status for the successor to \emph{MODE CHANGE} which tells him that a lock mode change has happened and we set a local flag to indicate that we have already reset the counters.

```c
int64_t arriveCount, departCount;
int64_t subtractArriveCount, subtractDepartCount;

/* We first get the current values of the counters */
arriveCount = MPI_Get(rankCounter, ARRIVE);
departCount = MPI_Get(rankCounter, DEPART);
MPI_Win_flush(rankCounter);

/* We prepare the values that will be subtracted from the counters */
subtractArriveCount = -departCount;
subtractDepartCount = -departCount;

/* We need to make sure that we reset WRITE if it was set */
if(arriveCount >= WRITE)
{
    subtractArriveCount -= WRITE;
}

/* Subtract the values from the current counters */
MPI_Accumulate(subtractArriveCount, rankCounter, ARRIVE, MPI_SUM);
MPI_Accumulate(subtractDepartCount, rankCounter, DEPART, MPI_SUM);
MPI_Win_flush(rankCounter);
```

Listing 4.8: Function that resets one counter

After that, the process looks for his successor. In case of having one, the process will pass the lock to the successor using MPI Put (line 40) and with that notifies him about a possible lock mode change.

If there is no known successor, the lock cannot be simply released. The process has to check the tail of the queue for new arrivals with MPI Compare and swap (line 28) like in the MCS lock release. Being sure whether it has a successor or not, it will either pass the lock or successfully return. In any case, the process releases the lock to the readers (line 20) if that has not already happened.

```c
int64_t nextStatus, successor, currentRank;
bool isReset = false;
```
4. Distributed Topology-aware Reader-Writer Lock

```c
/* Add one to the count of consecutive lock acquisitions on this level */
nextStatus = queueNode[STATUS] + 1;
if(nextStatus == hierarchyNode[THRESHOLD])
{
    /* We need to pass the lock to the readers */
    Reset_Counters();
    nextStatus = MODE_CHANGE;
    isReset = true;
}
/* We check if we already have a successor */
successor = queueNode[NEXT];
if(successor == -1)
{
    /* We have no known successor thus we pass the lock to the readers */
    if(!isReset)
    {
        Reset_Counters();
        nextStatus = MODE_CHANGE;
    }
    /* We check if somebody already enqueued themselves */
    currentRank = MPI_Compare_and_swap(-1, rank, rankHierarchyParent, TAIL);
    /* We have to ensure that MPI_Compare_and_swap has finished */
    MPI_Win_flush(rankHierarchyParent);
    /* We check whether we are the only one in the queue */
    if(rank == currentRank)
    {
        return;
    }
    /* We wait until the successor makes itself visible */
    do
    {
        successor = MPI_Get(rank, NEXT);
        MPI_Win_flush(rank);
    } while (successor == -1)
```
4.3. Distributed Reader-Writer Protocol

Listing 4.9: Releasing an RMA-RW writer lock for level 1

4.3.5 Reader Lock Acquisition

A code snippet for this function is provided in Listing 4.10. In this part of the implementation, we have to know which reader operates on which counter. Therefore, we have a field `counterRank` that tells the rank of the process where the counter is saved. The first if statement (line 7) is used to keep the readers from trying to grab the lock all the time in case that the reader threshold is reached or the lock mode is on WRITE.

After the statement, it increments the counter of the reader by applying MPI_Fetch_and_op and then checks whether the count is below the threshold: if yes, the mode of the counter is on READ and we can enter the critical section otherwise that means either that the lock mode is on WRITE or the threshold for the readers is reached. If the latter is the case, we need additionally to change the mode to WRITE but only if a writer is waiting.

To verify if a writer is waiting we need the rank of the root of the hierarchy which we save in the field `hierarchyRoot`. In both cases, we set the local barrier that sends the reader waiting until the counter is reset.

However, it can happen that several nodes try to acquire a lock when the threshold is reached. We have to make sure that only the first reader who reaches the threshold must reset the counter if no writer is waiting. The reset of the counter is executed by the function `ResetCounter` (line 34). All others decrement the counter again and go back to the start and try to acquire the lock again. After resetting the counter, the designated process goes back to the start as well.

```
int64_t currentStatus, currentTail;
bool barrier = false;

start:

/* We check if we already failed in acquiring the lock */
if(barrier)
{
    /* We wait until the counters are reset */
    do
```
4. DISTRIBUTED TOPOLOGY-AWARE READER-WRITER LOCK

```c
{  
currentStatus = MPI_Get(counterRank, ARRIVE);
MPI_Win_flush(counterRank);
} while (currentStatus > READER_THRESHOLD)

/* We add one to the arrival counter */
currentStatus = MPI_Fetch_and_op(1, counterRank, 
    ARRIVE, MPI_SUM);
/* We have to ensure that MPI_Fetch_and_op has finished */
MPI_Win_flush(counterRank);
/* We check if the reader threshold is reached */
if (currentStatus >= READER_THRESHOLD)
{
    /* We cannot acquire lock thus set local barrier */
    barrier = true;
    /* We check if we were the first that hit the 
     * threshold */
    if(currentStatus == READER_THRESHOLD)
    {
        /* We are responsible for passing the lock to the 
         * writers if there are any */
        currentTail = MPI_Get(hierarchyRoot, TAIL) ;
        MPI_Win_flush(hierarchyRoot);
        if(currentTail == -1)
        {
            /* There are no waiting writers thus we reset 
             * the counters and unblock us */
            Reset_Counter();
            barrier = false;
        }
    }
    /* We back off and try again */
    MPI_Accumulate(-1, counterRank, ARRIVE, MPI_SUM);
    MPI_Win_flush(counterRank);
    goto start;
}
```

Listing 4.10: Acquiring an RMA-RW reader lock

4.3.6 Reader Lock Release

For releasing a reader lock, the only thing to do is increment the departing reader counter. Since the threshold is already checked during acquisition,
we do not have to do anything else while releasing. This is one line of code and is shown in Listing 4.11.

```c
/* We add one to the departure counter */
MPI_Accumulate(1, counterRank, DEPART, MPI_SUM);
MPI_Win_flush(counterRank);
```

Listing 4.11: Releasing an RMA-RW reader lock
We begin this chapter by explaining the setup that we use for our experiments (§ 5.1). Afterwards, we show the results of the evaluation of the RMA-MCS (§ 5.2) and then of the RMA-RW lock (§ 5.3).

5.1 Experimental Setup

We use an implementation of a state-of-the-art RW lock of a MPI implementation (foMPI) [12] as a comparison. foMPI has two locking schemes that we utilize:

- A simple spin-lock (foMPI-Spin) that allows mutual exclusion
- A RW lock algorithm (foMPI-RW) that provides shared and exclusive access

We did not compare to other state-of-the-art NUMA-aware locks because they do not approach distributed locks and thus are orthogonal to our scheme. Their solutions however could be incorporated into our approach to make it even more scalable.

We first present the results of the MCS lock compared to the foMPI-Spin. We thereafter show how we set the parameters for an optimized version of the RMA-RW lock. At the end, we demonstrate how well the RMA-RW lock performs compared to the RMA-MCS and foMPI-RW lock.

All of our experiments were conducted on CSCS Piz Daint (Cray XC30) which consists of computing nodes. Each node has an 8-core Intel Xeon E5-2670 CPU with 32 GiB DDR3-1600 RAM, an NVIDIA Tesla K20X with 6 GiB GDDR5 RAM. It uses GNU’s Programming Environment version 5.2.40. Since the cores have the technology of hyper-threading, we can have up to 16 processes on one node. The interconnection is based on Cray’s Aries in a Dragonfly topology. The batch system (slurm 14.03.7) chooses the allocated...
nodes. All our implementations were written in C++ and compiled with the GNU 5.2.40 g++ compiler at optimization level -O3.

5.1.1 Implementation Details

For building up the tree for the hierarchy we need to know how the topology looks like. We use libtopodisc [14] for getting a communicator between MPI-processes on one computing node. With this communicator a process can find out on which node it resides. This is already sufficient for our needs since we either communicate within a node or between all MPI-processes. In addition, we initialize on each process windows that can be accessed by all processes that are associated to the same communicator handle.

5.1.2 Benchmarks

We performed five different benchmarks. For all of them, one can choose the percentage of writers respectively readers for the RMA-RW and foMPI-RW lock. Before each acquisition, a process generates a random number to decide whether it will act as a reader or a writer for this iteration. Therefore, there is sometimes an evolution from readers to writers and the other way round. After that a process executes some iterations of the corresponding benchmarks to warm-up and then loops 100’000 times per MPI-process with the different codes of the benchmarks. At the end of each benchmark, there is a communication phase where all processes send their results to a selected process which calculates the average of the results. We measure two different metrics: latency and throughput. The latency shows the duration of acquiring respectively releasing a lock in seconds which is important in real-time environments. The throughput is the aggregate count of lock acquisitions and releases per second. It shows how well the locks perform for large-scale data- and communication-intensive workloads. The time measurements are taken by a high-resolution timer [17]. Another metric that one can quantify is the fairness. A high fairness is achieved if the processes acquire the lock in the order of arriving. Thus, the fairness is decreased if a process acquires the lock even when a process is waiting that has arrived earlier. We do not capture fairness in our benchmarks but we will discuss about it in some parts of our evaluation.

The first one is the Latency Benchmark (LB). It measures the duration in seconds for acquiring a lock and releasing a lock. There is nothing else in or after the critical section except for the time measurement. After the loop, it calculates the average time for acquiring a lock, releasing a lock and the sum of them on this process before passing the values to the selected process.

The second benchmark is the Empty Critical Section Benchmark (ECSB). It shows how high the aggregate throughput is with no workload in the critical
5.2. RMA-based MCS Lock

and non-critical section. In the loop, there is only one lock acquire and one release, nothing else. The time is measured before and after the loop to calculate the lock throughput per second.

We call the next benchmark Single Operation Benchmark (SOB) since it measures the throughput where there is only one single operation in the critical section. Every process that holds an exclusive lock executes a write and the processes holding a shared lock execute a read. We use this benchmark because such short critical sections often occur in a realistic workload.

In the next benchmark we can vary the workload in the critical section and thus it is called (WCSB). This is done by fetching and adding a number to a global counter plus spinning afterwards for a random duration which simulates local calculations. This duration has an upper limit that can be specified as input. In our tests, we used 4\(\mu s\).

The last benchmark is the Wait After Release Benchmark (WARB). It shows how the different locks behave under varying contention. In the critical section, there is again a counter that the processes count up. But after releasing, the lock they wait for a random time before acquiring one again. The upper limit for waiting is specified as an input parameter and is set to 4\(\mu s\).

There is an extension for these benchmarks. We use it to optimize the thresholds for the writers. In order to do that, we ensure that these thresholds are reached by setting four processes per node as writers and the rest as readers. With this feature the percentage of writers is set to 25% on each node. We will refer to this extension by putting a “W-” before the benchmark names (W-LB for example).

5.2. RMA-based MCS Lock

First, we present the results of the comparison between the RMA-MCS lock and the foMPI-Spin lock. The outcome is as expected because foMPI-Spin is a spin-lock and thus will not scale as well as an MCS lock implementation for high contention. The Figure 5.1 demonstrates it very well where we used the LB for the measurement. foMPI-Spin has at least double the latency of the RMA-MCS due to a big amount of coherence traffic. Everyone has a copy of the value in their cache that gets checked before acquiring. As soon as the holder of the lock changes his copy during the release, everybody else’s cache entry is invalidated. Therefore, all of them want to have the modified version, which causes a lot of communication that slows the whole system down. In contrast, MCS lock implementations ensure that processes are local spin-waiting until they get notified by the predecessor. The result is the absence of a communication storm like in a spin-lock.
5. Evaluation

![Graph comparing Latency (sec) vs. MPI-Processes for RMA-MCS and foMPI-exclusive]

Figure 5.1: Comparison of RMA-MCS and foMPI-Spin in terms of latency for acquiring and releasing a lock when the contention is increased.

In Figure 5.2, one can see the four different benchmarks for throughput. One clearly visible point is the rise and drop of RMA-MCS right at the beginning. This is because the first data point is measured for eight MPI-processes. As already stated, this is actually less than the maximum number of processes on one node. Therefore, 16 processes is the optimized setting for RMA-MCS. For every number of MPI-processes that is bigger than 16, there is not only intra-node but inter-node communication as well. This is much slower and thus it is clearly impacting the throughput. Interesting enough, foMPI-Spin has later the decrease than RMA-MCS. This is because we used foMPI to lock a mutex object only on one MPI-process. Therefore, only the MPI-processes on other computing nodes have a slower latency for the communication to that specific process. Since for 32 processes only one half of them are on another computing node, it seems that this produces the highest throughput for that case. For more processes, the contention gets again bigger and there are more MPI-processes on other computing nodes. This results in a degrading throughput for a higher number of processes.
5.3 Distributed Reader Writer Lock

Before we compare the RMA-RW to other locking schemes, we first optimize the following four parameters:

- The number of counter in the system
- The threshold that regulates after how many consecutive writer lock acquisition the lock gets passed to the readers
- The thresholds on each level of our tree design that controls the conflict between the writers
- The threshold for the readers that will make the readers to pass the lock to the writers after reaching

Figure 5.2: Comparison of MCS and foMPI exclusive. These four graphs show the results of the benchmarks that measure the throughput for different numbers of MPI-processes.

5.3 Distributed Reader Writer Lock

Before we compare the RMA-RW to other locking schemes, we first optimize the following four parameters:

- The number of counter in the system
- The threshold that regulates after how many consecutive writer lock acquisition the lock gets passed to the readers
- The thresholds on each level of our tree design that controls the conflict between the writers
- The threshold for the readers that will make the readers to pass the lock to the writers after reaching

35
5. Evaluation

5.3.1 Counter Distance

In this section, we analyze how the distance between the counters influences the throughput of the RW distributed lock. In this context, distance means the distance between MPI-processes having a counter. We have an example in Figure 5.3 with a counter distance of four. We tested different settings with either several counters residing on one computing node or only a counter on every few nodes. In Figure 5.4, one can see the results for these different configurations. It is generated with a simple counter plus some local calculations in the critical section.

Figure 5.3: An example for a counter distance of four.

Figure 5.4: Comparison of different counter distances. The benchmark SOB was used to produce the results.

First, different peaks of throughput can be observed. The curves of the
counter distance two and four have their climax at lower contention. This makes sense since the shorter the distance the more work has a writer to do for changing the lock mode on a big number of MPI-processes. However, the values higher than four have their peaks at about the same contention. Because moving to higher process counts makes the differences between the peaks smaller. Next, one can discuss how well different counter distances perform compared to each other. One can clearly see that the bigger the counter distance is the higher the throughput will be but only up to a certain point. First, the outcome was surprising. We expected that a counter distance of 16 would be best since we have one counter per computing node. We conjecture that the cost of the readers, which have to access a counter on a computing node nearby, is smaller than the gain for the writers that have to check and change fewer counters. Nevertheless, this is true only up to a certain point. The counter distances 32 and 64 have about the same throughput. Therefore, the bigger cost of the readers balances out the gain of the writers.

Overall, a counter distance of 32 has the highest throughput. Therefore, it is optimal for readers since the counter lies on the same node or on one nearby and for the writers, there are not that many counters to change as for a lower counter distance.

### 5.3.2 Writer Thresholds

Other parameters that we want to optimize are the thresholds for the writers. For an architecture with \( N \) level there are also \( N \) thresholds – one on each level. The top most level threshold is actually the one that regulates after how many writers the lock gets passed to the readers. The total number of writer lock acquisitions before this threshold is reached is at most the product of all \( N \) thresholds. However, since our system has two level in the topology, we only have two thresholds to optimize. This leads to a problem because real world applications have a very small percentage of write operations. With that small number of writers it is unlikely that two of them work on the same computing node in our system since there are at most 16 processes on one node. Thus, we use the benchmarks with the writer extension for this section.

We first want to find the best fit for the product of our two level thresholds. One can see the result of that experiment in Figure 5.5. Our findings are that the fewer writers we have the higher the throughput. This makes sense because the latency for acquiring a writer lock is much higher than for a reader lock. Therefore, with a smaller writer threshold a lot more readers are able to acquire the lock and with that the throughput increases. But after the last reader releases the lock, there will be a lot of writers that have not finished yet. This decreases the fairness significantly. In the worst case, a
5. Evaluation

Figure 5.5: Different values for the writer threshold. We used the benchmark W-SOB to produce the figure.

writer has to wait for all readers to finish that arrived later than the writer itself. Thus, one has to decide about the trade-off between performance and fairness. We believe that a threshold of 1’000 seems like a good fit because we set the reader threshold to 4’000 and the writer percentage is 25%. Therefore, the readers have a small advantage since they have a better ratio between the percentage and the threshold.

After finding the best threshold for the writers, we optimize the lock with respect to the thresholds on each level in the hierarchy. We only have two thresholds and thus we vary the two so that the product of them are always equal to 1’000. We compare the values 10, 25 and 50 for the threshold on level 2 and for the one on level 1 the corresponding values 100, 40 and 20. We choose these values because for level 2 we want to have a sufficiently large number to exploit the locality and for level 1, we need a value that lets processes on different nodes acquire the lock before handing it to the readers. The outcome is shown in Figure 5.6. In this case, when more writers consecutively acquire the lock on one computing node, the throughput is higher as expected. Nevertheless, one can see that the difference between them is fairly small (up to 25%). However at the count of 1024 MPI-processes, the
thresholds 50 for level 2 and 20 for level 1 have about 50% more throughput. We assume that the thresholds do not have a strong effect for low counts of processes because of the design of the topology of which we only consider two levels in the memory hierarchy. Therefore, on level 2 there are only up to 16 MPI-processes compared to level 1 where all MPI-processes are contained. We believe that considering only two levels limits the impact of adjusting the thresholds on each level.

An interesting fact can be observed while comparing the latency values (Figure 5.7) for the lock acquire and release. The time values are the other way round than expected. We believe that the reason is the better fairness of smaller thresholds for the level 2 since more processes of different nodes can acquire the lock. With high values for the threshold, only a few selected processes on a small number of nodes can consecutively acquire the lock and therefore have a short latency. The average latency however is higher because a lot more other writers have to wait for a much longer time.

Figure 5.6: Comparison of different values for the writer thresholds on each level in the hierarchy. In the legend, the left number shows the threshold for level 2 and the right the value for level 1. We used the benchmark W-SOB.
5. Evaluation

Figure 5.7: Comparison of different values for the writer thresholds on each level in the hierarchy. In the legend, the left number shows the threshold for level 2 and the right the value for level 1. We used the benchmark W-LB for these results.

5.3.3 Reader Threshold

The next parameter, that we try to optimize, is the reader threshold. We started the search by measuring the throughput with the benchmark WCSB. The reader rate was set to 99.8% because this shows which value for the threshold performs best with a high number of readers. Figure 5.8 reveals the outcome.

As one can easily see, the throughput for thresholds 1’000 and 2’000 drops significantly for more than 512 MPI-processes. Thus, these numbers are not big enough to keep a good balance between readers and writers. An interesting observation is that increasing the reader threshold only improves performance up to a value of 3’000. We therefore provide more results for the thresholds 3’000, 4’000 and 5’000 that have about the same throughput. Now we will vary the reader rate on the reader threshold because these two parameters are dependent on each other. Here again, we use the WCSB for measuring the throughput. We present the outcome in Figure 5.9.

The evaluation shows no advantage of one threshold over the others. This actually emphasizes the fact that increasing the reader threshold only results
5.3. Distributed Reader Writer Lock

Figure 5.8: Different values for the reader threshold for a reader rate of 99.8%. We used the benchmark WCSB to produce the figure.

in a higher throughput up to a certain point. We expected such a result since each reader has a latency for acquiring, executing the critical section and releasing the lock again. This duration limits how high the throughput is for one reader. Therefore, to increase the aggregate throughput one has to utilize more MPI-processes.

5.3.4 Reader/Writer Rate

After tuning the parameters for the highest throughput, we now depict how well the RMA-RW lock performs compared to foMPI-RW lock with different writer respectively reader rates. For this compromise, we use the benchmark WARB. We have chosen this benchmark because we wanted to scale down the contention. The reason for it is that foMPI-RW achieves better results for fewer processes trying to grab a lock.

The outcome is visible in Figure 5.10. As expected, the RW distributed lock provides the highest throughput for a writer rate of 0.2%. This is because readers have a lower latency for acquiring a lock than writers do. Moreover, readers can grab the lock concurrently. At their peaks, the difference between the rates 0.2% and 2% is a factor of 2.8 and between 0.2% and 5% a
factor of 5.2. We tested further writer percentage configurations up to a rate of 100%. We found out that for percentages higher than 30% the throughput remains about the same. At such rates, the throughput is mainly restricted by the throughput of writers that are executing the critical section consecutively. This leads actually to another interesting fact. The throughput for such high rates is comparable to the throughput of the foMPI-RW lock. It means that even when there are only writers trying to acquire an exclusive lock of RMA-RW, it scores better results than foMPI that has shared and exclusive locks.

5.3.5 Comparison to the State-of-the-Art

Figure 5.11 shows our findings of the comparison between the RMA-MCS, the RMA-RW lock and the state-of-the-art foMPI-RW lock. As one can see, the MCS lock does not scale as well as the other two. The main reason is that the MCS lock is based on a queue for all processes. It does not make any difference between processes that execute only read operations and processes...
5.3. Distributed Reader-Writer Lock

that execute read and write operations in the critical section. However, the other two lock implementations allow readers to grab a lock simultaneously. This is of course an advantage of the RW lock scheme. Therefore, foMPI-RW provides better results than RMA-MCS. There is a big gap between the MCS and the RMA-RW lock. This has two reasons: the already mentioned RW scheme and its topology-oblivious behavior. It can happen that each consecutive process in the queue is on a different computing node. Thus, notifying the successor to proceed to the critical section is more expensive than informing one on the same node. Moreover, every process tries to put himself at the tail of the queue. This means that there is a single point that all need to check or modify while acquiring or releasing. The consequence is a bad scalability behavior.

In the implementation of foMPI-RW, a trade-off between simplicity and fairness has been made. Writers acquire the exclusive lock only if there are neither readers nor writers holding the lock but readers can obtain a shared lock even if there are other readers that are in their critical section. Thus, there could always be at least one reader which can result in a starvation

Figure 5.10: Different values for the ratio of writers and readers. We compared the values for the RMA-RW and the foMPI-RW lock. This data is produced with WARB.
5. Evaluation

Figure 5.11: Comparison of RMA-MCS, foMPI-RW and RMA-RW. These four graphs show the results of the benchmarks that measure the throughput for different numbers of MPI-processes.

of the writers. This is leading to an interesting phenomenon that causes degradation of the throughput. At the start of a benchmark, a certain small amount (in this setting 2%) of the processes are writers and the big part are readers. While most of the readers remain one, some of them evolve to writers. On the other side, writers will turn with a high chance to readers after their turn but with a lower rate than the readers since only one at a time can obtain the lock. Moreover, it is for writers difficult to grab the lock because there can always be a reader that blocks them as already said. Thus, the number of writers is rising in the system and this increases the latency. The last point why foMPI-RW is slower than the RMA-RW lock is the not being aware of the topology. There is still the problem that the locality of a multi-level system is not exploited. A lot of communication is inter-node
that is much slower than the intra-node. That is why the throughput of 
foMPI is only decreasing for higher contention in contrast to the constant 
increase shown by the RW distributed lock.

Overall, the RW distributed lock is the best scalable of the three as expected. It has a higher throughput than the other two even for little contention except for eight and 16 MPI-processes where the MCS lock is optimized very well. Reaching its peak performance, RMA-RW outperforms foMPI up to a factor of 15 and RMA-MCS up to a factor of 32. Of course the topology-awareness has a big influence whether a lock is scalable on a two-level system or not. But there is another reason as well since we outfitted the RMA-RW with thresholds that regulate how many readers respectively writers can consecutively acquire a lock before a lock mode change. With that regulation the balance between readers and writers is equalized. Thus, the described phenomenon, which the foMPI-RW lock suffers from, is not a problem for the RMA-RW. Furthermore, the overall fairness is better enforced than in the foMPI implementation. Nevertheless, for a high count of MPI-processes, the RW lock’s performance is decreasing. It is caused by the spread of the counters. A trade-off has been made between the locality of the counter for the readers and the work for writers to change the lock mode. There is still the possibility to just increase the distance between the counter nodes. This introduces more overhead for little contention although it gets compensated with a rising number of processes.
Chapter 6

Related Work

An important aspect in today’s high performance systems is the choice of a locking algorithm. There are several approaches that base on a queue lock. Especially the CLH [9, 24] and the MCS lock [25] are well-known. But these locking schemes have a limitation because they are NUMA-oblivious. Nevertheless, with the constant growth of large-scale processing machines, the implementations need to consider the underlying NUMA architecture.

Several ideas show a way to exploit the memory locality. Radovic and Hagersten [29] proposed a design of a hierarchical backoff lock. The main idea is that a thread reduces its backoff delay if it notices that another thread from the same cluster owns the lock. This increases the chance to acquire the lock for the waiting thread. However, backoff locks are vulnerable to starvation. The queue designs in contrast do not suffer from starvation and thus Luchangco et al. [23] presented a hierarchical CLH queue lock. In this scheme, threads from the same cluster build up a local queue which is spliced into a global queue. Their key idea is that the cost of the splicing is only a single CAS operation. But this approach considers only two levels of the NUMA hierarchy. Chabbi et al. [5] found a solution that works for any number of hierarchy levels. They propose that each level has its own MCS queue lock. In order to acquire the global lock, a thread has to obtain the MCS lock on each level starting at the processor level. Thresholds on each level control that one part of the NUMA domain does not get preferred over the others.

RW locks have been a subject of quite a number of researches. One of the first solutions was based on a simple counter scheme [8]. Courtois et al. recognized that there are different fairness schemes. On the one hand, reader can get preferred by allowing a reader to join the group of current readers even if there is a writer waiting. This scheme maximizes the throughput by reducing the delay for readers. On the other hand, a writer preference lock ensures that updates of the writers can be seen as soon as possible. It is en-
forced by letting readers wait if there is a current or waiting writer. Clearly, this does neither prevent starvation of the readers or writers nor is a scalable implementation. Mellor-Crummey and Scott [26] extended their MCS queue lock that it supports reader-writer exclusion. It allows readers in the queue to access the critical section if the predecessor is an active reader. Furthermore, this process notifies his successor (if it is a waiting reader) to access the critical section as well. There is a reader counter and a pointer to the next writer in the queue so that the last active reader can notify the writer to acquire the lock. This algorithm however does not scale well under heavy read contention.

Krieger et al. [21] introduced a wait-queue design in which the readers can splice themselves out of the queue during the release despite having a predecessor and successor that are active readers. They achieve that by using a doubly linked list. Nevertheless, there is still a single point which all threads need to access to add themselves to the end of the queue. To eliminate this contention, Hsieh and Weihl [18] suggest to trade writer throughput for reader throughput. This is achieved by having a private mutex object on each thread. The readers can acquire the lock by acquiring their private mutex object but the writers need to obtain all mutex objects of all threads. For a small system with few writers this scales quite well. However, for a large number of threads, this introduces a massive overhead for the writers.

More recent approaches yielded elaborate data structures like the Scalable Non-Zero Indicator (SNZI) [22]. In this locking algorithm, readers are traced by arriving at a leaf in the SNZI tree. This tree is constructed such that threads arrive at leaves that are associated to the same NUMA node as the thread itself. The approach does not use a centralized data structure and therefore is scalable for readers. However, writers stay NUMA-oblivious. An answer to that problem was given by Calciu et al. [4] in which they designed RW locks that are NUMA-aware for readers and writers. This is done by batching the readers together so that they access their critical section concurrently. On the other hand, writers of the same NUMA domain get batched together so that they grab the lock one after another. The conflict between readers and writers is resolved via the lock cohorting technique [11]. Writers have to acquire the cohort lock in order to get the exclusive lock and before accessing the critical section, they ensure that no reader is executing or about to execute their critical section. This is done by checking the so called read indicators. These indicators are distributed throughout the system (one per NUMA node) and they show whether a reader has arrived on a node or not. This design exploits the memory locality of up to two levels in a NUMA hierarchy.

As far as we know, we are the first to introduce an RW lock scheme for distributed memory machines that is aware of the underlying topology.
Chapter 7

Conclusion

Large-scale processing machines are growing rapidly nowadays. This results in a bigger need of architectures with a distributed memory design. Thus, synchronization primitives and concurrent algorithms have to be redesigned to adopt to these new circumstances. We first depicted an implementation of the well-known MCS lock on such a distributed system. In our test environment, the MCS adopted to newer technology has about half the latency of a spin-lock scheme that was previously adjusted to a distributed system.

As far as we know, we introduced the first design for a topology-aware distributed reader-writer lock that outperforms prior lock algorithms. It is based on two main ideas. Firstly, the writers acquire the global lock by obtaining the queue lock on every level in the hierarchy tree. Secondly, on every i-th node there is a counter that indicates the lock mode of the system. Every reader increments it during the acquisition and decrements it at leaving. Thus, writers can verify whether a reader is active by checking these counters. We optimized this locking algorithm by adjusting its three different types of parameters. The first one defines the counter distance that also decides how near a counter is to a reader and how many counters a writer has to check. The second parameter type is the writer threshold. There are N thresholds for N level NUMA hierarchy. They not only regulate how many locks can consecutively be acquired on one hierarchy level queue but also how many writers can grab a lock before passing it to the readers. The last parameter is controlling after how many readers a lock mode change is issued.

Microbenchmark experiments suggest that a well-optimized RW distributed lock outperforms the beforehand explained MCS lock up to a factor of 32. But since the MCS lock is only an exclusive lock, we compared it to a state-of-the-art RW lock that is part of a MPI implementation. We outmatched it as well up to a factor of 15.
7. Conclusion

These numbers show that there is a large potential in topology-aware lock algorithms. They exploit the locality of data which makes quite a difference regarding the latency of communication. Therefore, we encourage researchers to find synchronization structures and concurrent algorithms that utilize this potential.
Appendix A

A.1 Counter Distance

In Section 5.3.1, we showed the results of the comparison of different values for the counter distance. Figures A.1 and A.2 emphasize our findings.

Figure A.1: Comparison of different counter distances in terms of the total latency of acquiring and releasing a lock.
Figure A.2: Comparison of different counter distances with the specified benchmarks that measure the throughput for different numbers of MPI-processes.
A.2 Writer Thresholds

The following Figures (A.3, A.4, A.5) show the results for the optimization of the writer thresholds. They provide similar outcomes as the figures in Section 5.3.2.

![Graph showing the comparison of different writer thresholds in terms of the total latency of acquiring and releasing a lock.](image)

Figure A.3: Comparison of different writer thresholds in terms of the total latency of acquiring and releasing a lock.
Figure A.4: Comparison of different writer thresholds with the specified benchmarks that measure the throughput for different numbers of MPI-processes.
A.2. Writer Thresholds

Figure A.5: Comparison of different writer thresholds for level 1 (right number) and 2 (left number) with the specified benchmarks that measure the throughput for different numbers of MPI-processes.
A.3 Reader Threshold

The Figures below (A.6, A.7, A.8, A.9) support our findings in Section 5.3.3.

![Graph showing comparison of different reader thresholds in terms of the total latency of acquiring and releasing a lock.](image)

Figure A.6: Comparison of different reader thresholds in terms of the total latency of acquiring and releasing a lock.
A.3. Reader Threshold

Figure A.7: Comparison of different reader thresholds with the specified benchmarks that measure the throughput for different numbers of MPI-processes.
Figure A.8: Comparison of different reader thresholds and different reader percentages in terms of the total latency of acquiring and releasing a lock.
Figure A.9: Comparison of different reader thresholds with different reader percentages. The specified benchmarks measure the throughput for different numbers of MPI-processes.
A.4 Reader/ Writer Rate

The next two Figures (A.10, A.11) provide further results for different reader and writer percentages for RMA-RW and foMPI-RW. They are congruent with our findings in Section 5.3.4.

![Comparison of RMA-RW and foMPI-RW with different writer percentages in terms of the total latency of acquiring and releasing a lock.](image_url)

Figure A.10: Comparison of RMA-RW and foMPI-RW with different writer percentages in terms of the total latency of acquiring and releasing a lock.
Figure A.11: Comparison of RMA-RW and foMPI-RW with different writer percentages. The specified benchmarks measure the throughput for different numbers of MPI-processes.
A. Appendix

A.5 Comparison to the State-of-the-Art

The last Figure (A.12) suggests similar observations as in Section 5.3.5.

![Figure A.12: Comparison of RMA-MCS, foMPI-RW and RMA-RW in terms of the total latency of acquiring and releasing a lock.](image-url)
Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

High-Performance RMA Reader-Writer Lock

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):
Schmid

First name(s):
Patrick

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Oberrieden, 30.09.2015

Signature(s)

P. Schmid

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.