


STEFFEN CHRISTGAU<sup>1</sup>, BETTINA SCHNOR<sup>2</sup>

# MPI Passive Target Synchronization on a Non-Cache-Coherent Shared-Memory Processor

---

<sup>1</sup>  0000-0002-8702-8422

<sup>2</sup>Operating Systems and Distributed Systems, University of Potsdam

Zuse Institute Berlin  
Takustr. 7  
14195 Berlin  
Germany

Telephone: +49 30-84185-0  
Telefax: +49 30-84185-125

E-mail: [bibliothek@zib.de](mailto:bibliothek@zib.de)  
URL: <http://www.zib.de>

ZIB-Report (Print) ISSN 1438-0064  
ZIB-Report (Internet) ISSN 2192-7782

# MPI Passive Target Synchronization on a Non-Cache-Coherent Shared-Memory Processor

Steffen Christgau\*  
Supercomputing Department  
Zuse Institute Berlin  
Berlin, Germany  
christgau@zib.de

Bettina Schnor  
Institute for Computer Science  
University of Potsdam  
Potsdam, Germany  
schnor@cs.uni-potsdam.de

## Abstract

MPI passive target synchronization offers exclusive and shared locks. These are the building blocks for the implementation of applications with Readers & Writers semantic, like for example distributed hash tables. This paper discusses the implementation of MPI passive target synchronization on a non-cache-coherent multicore, the Intel Single-Chip Cloud Computer. The considered algorithms differ in their communication style (message based versus shared memory), their data structures (centralized versus distributed) and their semantics (with/without Writer preference). It is shown that shared memory approaches scale very well and deliver good performance, even in absence of cache coherence.

**CCS Concepts** • **Computer systems organization** → **Multicore architectures**; • **Computing methodologies** → *Concurrent algorithms*; • **Software and its engineering** → Message oriented middleware.

**Keywords** process synchronization, programming models and systems for manycores, MPI

## 1 Introduction

Distributed hash tables (DHTs) are a common approach for fast data access in big data and data analytics applications. However, DHTs imply dynamic communication which makes an implementation using two-sided communication, i.e. with SEND and RECV operations, cumbersome. In contrast, one-sided communication with PUT and GET operations is a suited programming model for a DHT. It allows to specify the communication parameters by the local process only and does not require knowledge about the communication on the remote side.

Concerning the process coordination, a DHT application follows the Readers & Writers model: reads may occur concurrently while inserts have to be done exclusively. Hence, a resource has to be locked before it is updated. Typically, writers are given preference to avoid readers reading old data. This coordination scheme maps on MPI's *passive target synchronization* which offers *exclusive locks* (one writer)

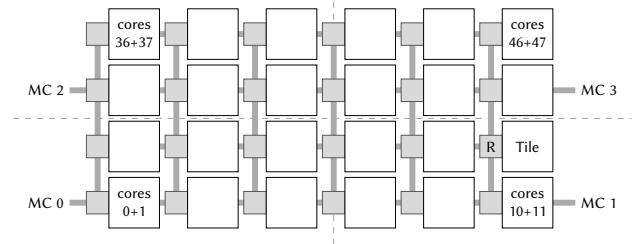


Figure 1. Overview of the Intel SCC.

and *shared locks* (many readers). In addition, an MPI implementation has much freedom to implement the process synchronization for passive target OSC [17, p. 448].

This paper discusses different synchronization algorithms on the non-cache-coherent Intel Single-Chip Cloud Computer (SCC) which is an experimental many-core chip that is comprised of 48 in-order Pentium (P54C) cores [12]. Figure 1 shows an architectural overview of the chip. Figure 1 shows an architectural overview of the chip. While core counts steadily increase, the management of cache coherence becomes a more challenging task due to that high number of cores and high memory bandwidths [18]. Although coherent high-end processors with 64 cores are currently available, non-coherent architectures provide an interesting research domain. It has been shown in previous work that such nCC shared-memory systems can be easily programmed with well established technologies like, e.g., MPI [7].

RCKMPI, the MPI implementation for the SCC, uses messages to implement passive target synchronization. We compare this implementation against two different synchronization schemes from the literature which are suited for the implementation on a shared memory system: a best-effort approach from Gerstenberger et al. [9], and a synchronization scheme with writer preference based on the work of Mellor-Crummey and Scott [15].

Gerstenberger, Besta and Hoefler (GBH) present a synchronization scheme for MPI passive target synchronization for the Cray XC super-computers [9]. We call their synchronization scheme the *GBH best effort approach*. They implement passive target synchronization, but without writer or reader preference. Rather, a best-effort approach is used: Locks are

\*Most of the work was done at the Institute for Computer Science, Potsdam

acquired without respect to other processes. If an acquisition does not succeed, all data structures will be released and the process will try again later with an exponential back-off.

Mellor-Crummey and Scott proposed the so-called MCS locks [15] which are based on linked lists of wait objects that are allocated in shared memory. Those locks have been extended to support different preference use-cases [16] which clearly distinguishes their approach from the GBH concept. The design of Mellor-Crummey and Scott allows to prefer writers and is therefore suited for all use-cases where many readers and less writers are expected.

The contributions of this paper are

- First implementation of *passive target synchronization* on the non-cache-coherent SCC using shared memory,
- investigation on the scalability and their impact on a DHT application of different synchronization algorithms, and
- recommendations for both application and MPI developer regarding which synchronization scheme may be beneficial under which application behavior.

The next section gives an overview over MPI one-sided communication with the focus on passive target synchronization. Section 3 presents related work. The different synchronization schemes and their data structures are described in Section 4, their implementation on the SCC is presented in Section 5. Results from a micro- and application benchmarks are presented in Section 6, followed by a discussion. Section 8 concludes the paper.

## 2 MPI One-Sided Communication

MPI specifies one-sided communication (OSC) since version 2.0 of the standard. There are two main aspects which makes OSC different from point-to-point exchange: First, only one process (called *origin*) specifies the parameters of the communication, like destination and size of communicated data. From the API perspective, the *target* process is not involved in the communication. The second main difference is that communication is separated from synchronization. The latter must be explicitly performed.

Data is exchanged via a *window* that exposes parts of a process' address space to other processes. A *window object* serves as a handle for accessing all windows that have been collectively created by a group of processes. Combined with the process identifier (rank), the window object identifies the destination of communication operations.

An origin communicates with operations like PUT and GET to either replace or fetch window data. Additional operations like ACCUMULATE or FETCH\_AND\_OP combine the data in the window with the provided buffer atomically, but per element, using pre-defined operations.

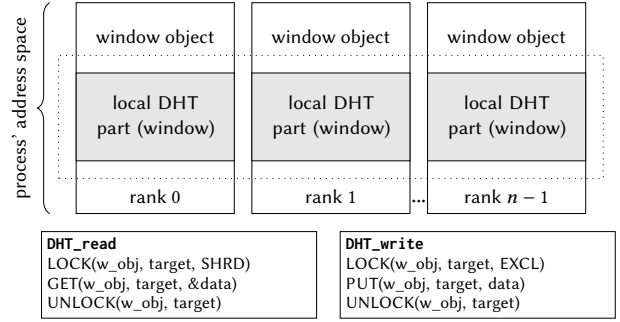


Figure 2. Usage of MPI windows for a distributed hash table.

### 2.1 Passive Target Synchronization

For the implementation of a DHT, MPI's *passive target synchronization* is well suited. Opposite to the active variant where both origin and target have to issue synchronization routines, only the origin has to issue a synchronization routine in the passive variant. This enables a shared-memory like programming style even on distributed memory machines.

The API defines *locks* as means for passive target synchronization. Before an origin process issues communication operations (see previous subsection), it has to issue a synchronization method, like MPI\_WIN\_LOCK. This opens an *access epoch* at the window of a given single process. *Exclusive* and *shared* locks can be acquired. With an exclusive lock, conflicting accesses can be avoided as this lock type ensures that concurrent modifications a target's window do not interfere. In contrast, a shared lock allows several processes to access a window. Shared locks are suited for concurrent read operations.

After the communication operations have been issued, the access epoch needs to be closed with MPI\_WIN\_UNLOCK. The MPI standard defines that RMA operations issued during the epoch will have been completed both at the origin and the target when the call that closes the access epoch returns [17, p. 447].

Additionally, the MPI standard defines MPI\_WIN\_LOCK\_ALL which locks *all* windows of the window object. However, the employed lock type is restricted to shared locks.

### 2.2 An RMA based Implementation of a DHT application

Based on MPI's passive target one-sided communication, DHT's can be implemented. In a DHT application that uses MPI RMA functionality, each process reserves (equally-sized) parts of its local address space as storage for a part of the distributed hash table as shown in Figure 2. Using MPI windows, a global address space is created which allows to access the distributed data via the window object.

In such an application, the hash function returns two information: the process and the offset at which the table's

entry could be found. In case data needs to be looked up, the application computes the hash and acquires a shared lock via `MPI_WIN_LOCK(SHARED, ...)` for the window at the determined process. Using an `MPI_GET` call, the hash table entry is fetched from the offset provided by the hash function. The lock is released by calling `MPI_WIN_UNLOCK`. Special values or markers of the fetched data may indicate the absence of a valid table entry. For a write, an exclusive lock and `MPI_PUT` is used to update the hash table (see Fig 2).

The sketched implementation solely relies on existing MPI routines. Thus, a DHT can be realized easily for different platforms. By tuning the MPI implementation, different semantics can be achieved. In the following, we discuss different designs for the synchronization on nCC platforms with a focus on the DHT use-case.

### 3 Related Work

Several research groups are concerned with the efficient implementation of MPI OSC and its synchronization methods. An early work is the discussion of MPI OSC on InfiniBand clusters [13].

Recently, implementation schemes for NUMA-aware locks on cache-coherent multicore machines are gaining interest [3, 8, 11, 14], but non-cache-coherent architectures are still neglected.

#### 3.1 MPI OSC and Synchronization on the SCC

RCKMPI [23] is a tuned MPI implementation for the SCC that is based on MPICH. It uses messages which are transferred via the SCC's on-chip Message Passing Buffers (MPB). One-sided communication including all synchronization styles is supported, but is based on messages as well.

An implementation scheme for RMA communication on the SCC that avoids messages is discussed in [7]. In case of MPI's *general active synchronization*, Christgau and Schnor have shown that an implementation using shared memory and uncached memory accesses outperforms the message-based approach significantly [5]. Similar, Reble et al. discuss one-sided communication for the SCC, but focus on the active target *fence synchronization* style which they implement on top of an efficient barrier [19].

The authors of [1] investigate barrier synchronization on the SCC and use the Message Passing Buffer to store the synchronization data. In [2], they even exploit unused entries in the rare lookup tables of the chip's memory subsystem. The bottom line of this research is that synchronization data should be placed close to the spinning core.

#### 3.2 Synchronization on Distributed Memory Architectures

Gerstenberger et al. have published performance numbers of a distributed hash table application running on up to

32k cores [9]. They use their own MPI-3.0 RMA library implementation for Cray Gemini and Aries interconnects called foMPI (fast one-sided MPI). Inserts are based on atomic compare and swap (CAS) and atomic `fetch_and_op` operations which are implemented on top of proprietary Cray-specific APIs. The presented synchronization scheme for passive synchronization is described in Section 4.2 and adapted for the SCC (see Section 5).

Schmid et al. have proposed a scheme for Readers & Writers locking dedicated for distributed memory architectures with RMA capabilities like the Cray XC30 [22]. The synchronization data structures are organized hierarchically in a distributed tree.

Subsuming the related work, there are no efforts in *passive target synchronization* for nCC many-core CPUs with shared memory like the SCC.

### 4 Design of Synchronization

This section describes three implementation designs for MPI passive target synchronization. The first two are known from the literature ([9, 15]), the third one describes the default implementation on the SCC. While [9] presents a best-effort approach which scales well on the Cray Gemini which is a distributed-memory machine, the algorithm presented in [15] addresses scaling on shared-memory machines.

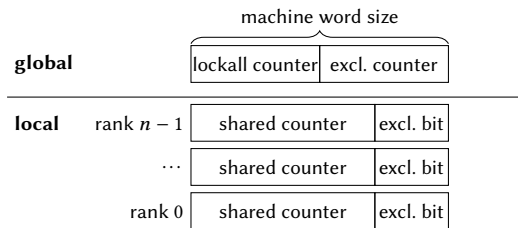
For the implementation of a DHT, shared and exclusive locks are used which are set on a per-target basis with the LOCK operation as depicted in Section 2.2 and Figure 2. Therefore, we omit the discussion of the LOCKALL functionality where possible.

#### 4.1 General Design Principles

Since the SCC is a shared memory architecture, an immediate scheme which actually performs synchronization operations when according methods are invoked should be preferred [7]. This enables direct communication after synchronization. Its counterpart, *deferred* synchronization, delays both the actual synchronization operations and the communication until the end of an access epoch. This approach allows optimizations which are mainly suited for message based approaches, but it is not beneficial for shared memory platforms. [10]

#### 4.2 GBH Best Effort Synchronization

In [9], Gerstenberger, Besta and Hoefler (GBH) present a synchronization scheme for MPI passive target synchronization for the Cray XC super-computers. It is based on atomic remote direct memory operations (RDMA) operations which are supported by the hardware. The design uses two stages of counters for each created window object: a single global counter and per-process local counters. All of them are of machine word size, but their bits are interpreted in different ways. All counters are allocated in memory local to owning



**Figure 3.** Counters used by the GBH scheme.

process. The global counter resides in the memory of a designated process (rank 0). The counters’ memory locations, however, are made accessible for RDMA operations from remote processes as well.

The global counter is used to track active LOCKALL operations and exclusive locks which are mutual exclusive. The per-process counter indicates the number of active exclusive and shared locks. As there can be only one exclusive access at a time, a single bit is used to indicate such epochs. The remaining bits of the variable are used to count the number of active access epochs using shared locks. A value of zero indicates no active access epochs of either type on the associated window. The separation of both values inside the counter is illustrated in Figure 3

When an exclusive lock is to be acquired, an atomic add operation increments the global exclusive counter part and returns the old value including the lockall value. If its value was zero no LOCKALL access epoch is active. Then, an atomic compare-and-swap (CAS) operation is issued on the remote counter variable of the target process. Its value is compared with zero and swapped with 1 such that the exclusive access is indicated if there is no other active access. For shared locks, the counter’s value is incremented with an atomic fetch-and-add (FAA) operation which returns the old value. If the old value indicates the presence of an exclusive access, the increment is undone. In that case, the process attempting to acquire the lock steps back and repeats its attempt. The same applies to an exclusive lock attempt for which the CAS operation did not succeed due to the presence of an exclusive lock.

The UNLOCK operation atomically sets the local counter to zero (exclusive access) or decrements the shared lock part atomically (shared access).

The GBH scheme is not starvation-free. Exclusive write accesses can be delayed indefinitely by a sequence of shared accesses. The indicator for an exclusive access (the single bit inside the per-process counter) is reset for every failed attempt which gives concurrent shared access the possibility to acquire the lock. In addition, there is no guaranteed order of which the processes acquire an exclusive lock. A newly arriving process which wants to acquire an exclusive lock might overtake a process that arrived earlier but stepped back. Thus, even attempts for exclusive locks can

be deferred indefinitely by other attempts of the same type. However, if there is little or no contention on the lock, those disadvantages disappear.

In contrast to the GBH approach, the following design prefers writers. This approach is suited for all use-cases where many readers and less writers are expected.

### 4.3 MCS Locks with Writer Preference

To avoid centralized spin objects which cause high interconnect traffic, Mellor-Crummey and Scott proposed MCS locks [15]. Those are based on linked lists of lock objects that are allocated in shared memory. Each process that wants to enter a critical section by means of MCS locks appends a list entry which consists of a boolean flag `blocked` and a pointer to the next waiting process. The flag is initially set to `TRUE`. A process that wants to acquire the lock repeatedly polls the flag in its list entry until it is set to `FALSE` by another process.

The main advantage of using one list item per process is that spinning is done only on a local data item and not on a globally shared one like a single spin lock, for example. In case of coherent shared memory systems, this reduces the coherence traffic on the interconnect and makes processes to spin on their local caches only.

When a process enters an MCS lock it adds its lock item to the list by using atomic operations. If there are other processes waiting in the list, spinning is started. When a process wants to release an acquired lock, it sets the `blocked` flag of the successor (referenced by the next pointer) to `FALSE`, so the spinning of the waiting process is terminated.

Based on the original MCS locks, which do not differentiate between process types, Mellor-Crummey et al. present specialized locks that give precedence to either reading or writing processes [16]. We have implemented MCS locks with writer preference, since it fits best to the DHT use case where lots of readers and rare writers are expected. We call this lock type *MCS-WP*.

Independent of the precedence, the proposed lock data structures contain lists for waiting reader and writer processes as shown in Figure 4. In addition to the lists, there is a state variable which is a single integer variable. For writer-precedence, the state tracks the number of active readers and provides flags for indicating presence of interested readers, interested writers, and active writers.

The state is manipulated with atomic operations. Those are used to implement `{start/end}_{write/read}` methods. Details can be found in Section 3 of [16]. In case of writers precedence lock, readers can join other readers as long as there is no interested writer. Otherwise, they queue up in the list of readers. On the other hand, writers that arrive after an active writer will be woken up by the leaving writer. In that case, readers are left in the wait state.

For usage with MPI passive target synchronization, every window  $i$  is associated with a lock data structure  $L_i$  as shown in Figure 4.

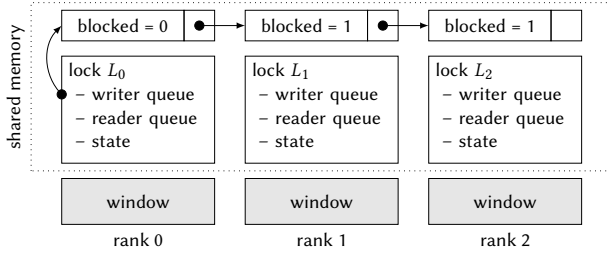


Figure 4. MCS-lock based data structures.

#### 4.4 Message-Based Synchronization

RCKMPI, the MPI implementation for the SCC, uses messages to implement passive target synchronization. This behaviour is inherited from MPICH’s CH3 device implementation but varies depending on the configuration. By default, the LOCK synchronization and subsequent communication operations are deferred until the end of the UNLOCK operation. The library then sends a control message from the origin to the target process, waits for a reply, i.e. *lock granted message*, issues the communication operations, and signals the unlock operation by setting an according field in the message header of the final communication operation.

If the access epoch constitutes only one outstanding RMA operation, a single message that indicates lock acquisition, the operation, and lock release is emitted. Depending on the actual access, the origin then waits for a reply indicating the completion of the RMA operation on the target process. In case the access epoch did not contain communication operations, no messages are sent at all.

However, MPICH/RCKMPI can be configured to send a control message to the target for lock acquisition at the beginning of the access epoch. Similar to the default behavior, all RMA operations are deferred to the epoch end. Nevertheless, messages are not merged together, but the bit that indicates a final communication operation, and therefore the unlock step, is still piggybacked in the last message. When the access epoch ends (by calling UNLOCK), the implementation on the origin waits for a *lock granted message*, performs outstanding RMA operations (if any), and potentially waits a for reply. If there were no RMA operations, MPICH sends an explicit unlock message. Although this implements one-sided communication it actually requires participation of the target to process the synchronization messages.

Independent of the active configuration, the lock requests from different origins are serialized at the target process. Since the received messages are processed in the order at which they are received by the target, there is no preference of readers or writers (or according lock type).

Table 1 summarizes the main characteristics of the presented implementation schemes. All schemes can be used to implement MPI’s passive target synchronization to support

Table 1. Comparison of the discussed designs.

	RCKMPI	GBH [9]	MCS-WP [15]
communication	messages	shared mem.	shared mem.
synchronization	deferred	immediate	immediate
R&W support	yes	yes	yes
fairness	FIFO	best-effort	writer pref.
data structures	–	centralized and distributed counters	distributed counters

Readers&Writers semantics (exclusive and shared locks), but they differ in their fairness behavior.

## 5 Implementation on the SCC

### 5.1 The Single-Chip Cloud Computer

The SCC is not a product but a research vehicle [12]. Each of the 48 cores has two integrated 16 KB L1 caches – one for data and instructions – as well as an external unified 256 KB L2 cache. There is no cache coherence between the caches of different cores.

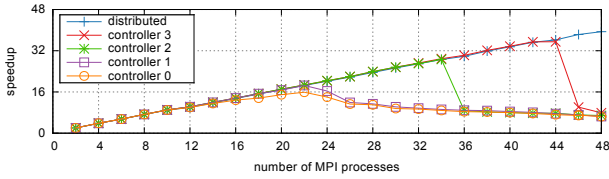
Two cores are placed on a tile. In total, 24 tiles are arranged in a regular  $6 \times 4$  grid, to which four memory controllers are attached as well. In addition to the cores, a tile also provides a fast 16 KB message passing buffer (MPB). The on-chip network allows to access both the MPBs as well as the main memory by plain load and store instructions. Every core can access all memory locations in the system via the mesh network, making the SCC an nCC-NUMA processor.

### 5.2 Handling of Messages in RCKMPI

All of the of the above synchronization schemes have been implemented in RCKMPI. The message-based variant uses the CH3 device code from MPICH. Sending and receiving of the CH3 messages has been optimized for the SCC in the work of Ureña et al. [24]. Messages are exchanged essentially by storing them in the fast on-chip SRAM-based MPB of the core that executes the receiver. The receiver polls the local MPB for new messages, and copies them for further processing. This implementation, in addition with required bug fixes [4], is considered as the baseline version.

### 5.3 Shared Memory Data Structures

The GBH and MCS-WP implementations do not use messages. Instead the required data structures are allocated in shared off-chip DRAM memory. Due to the non-coherent architecture of the SCC, those data structures are accessed with uncached memory operations. Those are enabled by the `mmap`’ing according physical memory regions into a process’ virtual address space via a special device (similar to `/dev/mem`) of the SCC-adjusted Linux kernel. It has been shown in the past that uncached memory accesses can significantly outperform message-based synchronization schemes [7, 21].



**Figure 5.** Speedup of a strong scaling experiment for different placements of the synchronization data structures [6].

In case of the GBH *best effort* and the MCS-WP synchronization schemes, polling on variables is performed. While previous research proved that polling the on-tile MPB or even the Lookup Tables (see Section 3) reduces the traffic on the interconnect, both approaches are hardly feasible in our case. The MPB is of limited size and is completely used by RCKMPI that serves as the foundation of the other implementations. The LUTs have free items, but they require special resource management which is out of the scope of this work. Therefore, we focus on the usage of the external DRAM as resource for the synchronization data. The experimental results will therefore provide an upper bound for the performance of the synchronization but not the optimal values that could be achieved by using memory (like the MPB) that is very close to a spinning core.

#### 5.4 Data Allocation Strategy

To account the NUMA characteristics and to avoid contention on the four memory controllers all data structures are allocated in that DRAM portion which is managed by the memory controller that is closest to the core that allocates the data. The need for the distributed allocation was already derived in our previous work on active target synchronization [5]. Here, a strong scaling experiment with a memory-bound application using stencil computations was conducted with varying locations of the synchronization data. While the per-process application was always allocated at the closest memory controller, the place for the synchronization data was either fixed in one of the four memory controllers or the synchronization data was distributed. In the latter case the data was allocated in the DRAM memory closest to the process.

From Figure 5, it can be seen that the distributed allocation strategy allows linear scaling of the application. Opposite to this, a centralized location of the synchronization data for all processes causes memory contention and results in poor parallel application performance. This emphasizes the need for distributed data structures.

#### 5.5 Atomic Operations

Besides the message-based one, all synchronization schemes require atomic operations. In the GBH scheme, the counter variables need to be modified with FAA and CAS operations.

The MCS-WP variant also requires CAS operations, among others, to modify the linked list and the lock state.

The SCC, however, lacks those primitive atomic operations. To implement any atomic operation on integers or pointers the test-and-set registers (TSR), of which each core owns only one, are employed. Reading such a register returns its last value and sets it to zero atomically. Writing any value resets the register to one. Thus, the TSR can be used to synchronize concurrent access to a resource associated to the core owning that TSR.

For implementing atomic operations, a TSR is acquired in a blocking fashion before the operation is performed. The TSR is reset immediately afterwards. Other means of synchronization or atomic operations which are provided by the system FPGA have been proven to exhibit higher latencies [20]. Although previous research showed that the TSR are prone to unfairness under high contention due to the SCC’s NUMA characteristics [20], no such effects have been observed in case of the active target synchronization implementation [7]. In addition, contention on the TSR is not assumed because access epochs and library overhead between the required atomic operations are considered to be much longer than the critical path in the atomic operation itself. However, for future architectures, the (hardware) support for atomic operations might be critical for efficient software designs.

## 6 Experimental Evaluation

We evaluate the different design schemes using a communication-free microbenchmark and check the impact on the DHT application from Section 2.2 on the SCC.

### 6.1 Environment

The experiments were conducted on a SCC system with cores clocked at 533 MHz and 800 MHz for the mesh network and the memory controllers. A total of 32 GB of RAM was installed on the system. Each core runs Linux 3.1.4 with platform-relevant patches applied. Software is cross-compiled using GCC 4.4.6, and MPICH 3.1.3 was used as the foundation MPI implementation. The MPB-based CH3 channel from RCKMPI was merged with the modifications from [4]. The synchronization functions were overridden to implement the approach presented in this work. The resulting MPI library was compiled with optimization enabled (-O2).

### 6.2 Microbenchmark description

To evaluate the performance of the different synchronization schemes, a microbenchmark is used which measures latency for a pair of LOCK/UNLOCK operations. No communication is performed between those two operations. The time for performing these operations is compared for the GBH and MCS-WP implementations as well as for the message-based



but SCC-optimized RCKMPI. Because the default RCKMPI implementation defers the synchronization, we also measure RCKMPI with a forced message exchange for synchronization (cf. Section 4.4).

Each process of the microbenchmark performs 1000 pairs of LOCK/UNLOCK calls in a tight loop. The type of the employed lock is controlled by an input parameter that specifies the share of shared and exclusive locks each process shall issue. According to that parameter, every process randomly decides between the two lock types. The target process is chosen randomly as well and may include the origin process.

The access mode (shared or exclusive) will obviously have an influence on the results. Therefore, three different ratios of shared and exclusive locks, i.e. readers and writers, were measured: only shared locks (only readers), all accesses are made with exclusive locks (only writers) and a mixture of both where shared and exclusive accesses are equally distributed (see Figure 6–8).

Since we are interested in the scaling of the different synchronization schemes, we run the benchmark with different numbers of processes. The processes are mapped according to the core with matching number. That is, the rows of the SCC’s mesh network are filled before moving to the next row. In case for 24 processes, the lower half of the chip (see Figure 1) is filled.

From each of the 1000 LOCK/UNLOCK cycles, the required time is measured. Finally, all samples from all processes are gathered and the median time from all synchronization operations is computed. This value is shown in the following diagrams for different core counts. We compare MCS-WP, RCKMPI with both immediate messaging (synchronization message upon method call) and default behaviour (no messages), and GBH. In addition to GBH, a version without back-off is included in the evaluation in order to analyze the impact of the back-off on the synchronization latency. For the version with back-off, the initial delay between two lock acquisition attempts is 1  $\mu$ s. This value is doubled for each consecutive failed attempt. It has been shown for the SCC that the usage of back-offs can improve the performance of synchronization primitives [19].

### 6.3 Results from the Microbenchmark

In the following, we present the results from the microbenchmark in the previous section for the three different mixtures of locks.

#### 6.3.1 Shared Locks Only

Figure 6 shows the latency of the different implementations when all accesses are shared. The RCKMPI implementation with immediate messaging has the highest latency due to overhead from sending and processing the control messages. The default RCKMPI implementation includes only library overhead but no message exchange and scales therefore well. It is slightly slower than both GBH versions due to additional

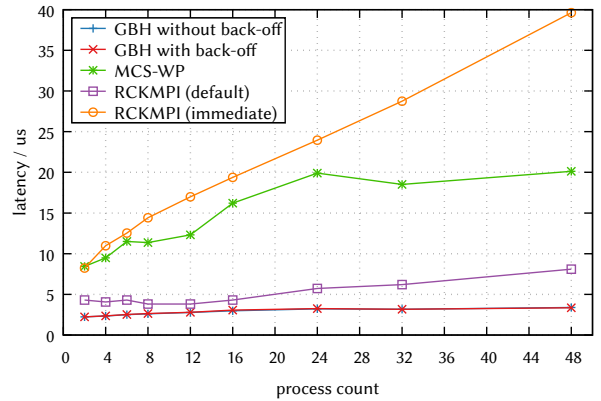


Figure 6. Latency for shared locks.

management in the message-based code path that are not used for the GBH and MCS implementations.

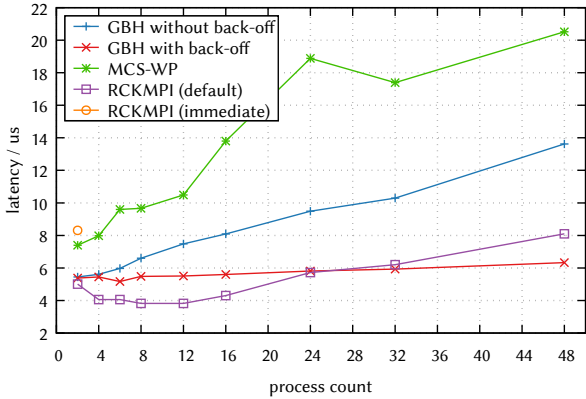
Both GBH versions exhibit nearly constant and identical synchronization latencies because no conflicts occur in the shared-only use-case and thus no back-off is required. Consequently, the two curves overlap in the plot. Similar, the reason for the constant time is that are not mutual exclusive. In the GBH scheme, acquiring a shared lock only involves incrementing the shared counter in the target’s local counter (see Fig. 3). Due to the distribution of the synchronization data and missing exclusive locks, which might cause more attempts to acquire the shared lock, no contention on these counters is observed on the SCC.

In case of the of the MCS-WP, the latency is generally higher than for GBH. The latter only involves incrementing a single per-process counter value, but for MCS the state variable needs to be checked and list data has to be changed. This causes the operations to take longer than for GBH.

From the data one can also note an increasing latency for up to 24 processes. After that, the latency remains nearly constant with a slight drop for 32 processes. This observation can be attributed to the distributed synchronization data. With up to 24 processes, the two lower memory controllers of the chip (see Figure 1) have to handle the polling requests of the 12 processes associated to each of them. Additional processes are then handled by the next memory controllers, but do not increase the load on the already utilized ones.

This is also the reason for the slight latency drop at 32 processes: Since the upper two memory controllers have to serve fewer processes than the lower two, the median latency reduces. Similar behavior can be identified for the switch from 6 (only handled by MC 0) to 8 processes (MC 1 handles additional polling accesses).

Since for 24 processes the two lower memory controllers experience maximum usage and because of the distributed data, no further increase of the lock latency is observed when the number of MPI processes is raised. This is different from



**Figure 7.** Latency when the processes use 50% shared and 50% exclusive locks.

the statement in [22, p. 11], that MCS locks that distinguish between readers and writers do not scale well under heavy read contention. We are not able to confirm this remark by our experiments on the SCC.

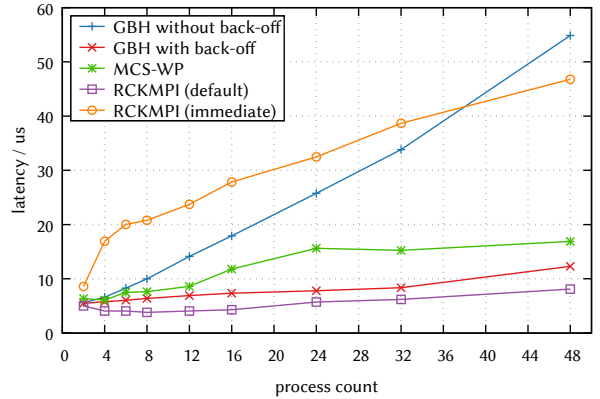
### 6.3.2 Lock Type Mix

In Figure 7, the results for the 50% mix of shared and exclusive locks is displayed. For this workload, no data — except for two processes — could be acquired for the immediate RCKMPI variant. The benchmark deadlocked in those cases. Our assumption is that required responses to control message are not sent when they are expected. This might be due to absent message processing and might be solved by triggering process through `MPI_Test` calls. However, a deeper investigation was out of the paper’s scope.

The default variant of RCKMPI which does not send any message unsurprisingly performs as in case for shared lock.

For GBH, the latency is slightly increased compared to the previous results. The scaling, however, remains nearly identical and still shows a constant time for the synchronization for all process counts. The increased latency can be accounted to the higher probability for an unsuccessful attempt for lock acquisition. In such a case, the processes perform their back-off but are able to acquire the lock in a later attempt very soon, since the median latency only increases by about  $3\ \mu\text{s}$ .

Opposite to GBH with back-off, the version without this feature shows a latency that increases linear with the number of processes. The effect is due to the contention. This can be explained by a competition for both the global and the per-process counter variables. This reduces the chance of a lock acquisition for either process type. Especially, the global counter must be modified both at the beginning and at the end of the lock attempt — notably, this has to be done also in the unsuccessful case. Since the global counter is



**Figure 8.** Latency for exclusive locks.

a centralized data structure, contention on the responsible memory controller is likely.

With the exception of the GBH without back-off and the dysfunctional immediate RCKMPI version, the overall performance and scaling is identical to the previous scenario.

For MCS-WP, an almost identical performance as in the previous experiment is observed. While two different process types are active, the same data structures are used and the same operations (state manipulation and list management) are performed. Thus, the overall performance stays the same.

### 6.3.3 Exclusive Locks Only

Finally, Figure 8 shows the scaling where only exclusive locks are used.

The GBH variant without back-off clearly suffers from the sole usage of exclusive locks and its aggressive best-effort approach. The effect of contention on the global counter from the previous experiment is amplified which causes increased latency.

Contrary to that observation, the other synchronization schemes still perform with identical scaling behavior and similar absolute latency. For GBH with back-off, the latency increases slightly and approaches MCS-WP. This might be caused by an increased number of attempts to acquire the lock.

For MCS-WP, the performance is still equivalent to the previous experiments. Because writers just queue up at the individual per-process queues (cf. Figure 4), the median time to acquire the lock does not increase. Further, the completely distributed data structures pays off as no contention occurs.

The immediate RCKMPI variant works without problems in this experiments. However, the latency is up to about four times higher than for the other implementations. Moreover, linear scaling can be observed.

**Table 2.** Quartiles and interquartile range (absolute and relative to the median) for the LOCK/UNLOCK latency at 48 processes.

scheme	shared locks only					50% lock type mix					exclusive locks only				
	$Q_1$	$Q_2$	$Q_3$	IQR	IQR/ $Q_2$	$Q_1$	$Q_2$	$Q_3$	IQR	IQR/ $Q_2$	$Q_1$	$Q_2$	$Q_3$	IQR	IQR/ $Q_2$
GBH	3.00	3.39	4.14	1.14	0.34	2.94	13.62	61.85	58.91	<b>4.33</b>	23.52	54.86	143.40	119.88	<b>2.18</b>
GBH back-off	2.98	3.35	<b>4.02</b>	1.04	0.31	2.79	6.33	<b>8.88</b>	6.09	0.96	8.90	12.31	<b>19.24</b>	10.34	0.84
MCS-WP	16.39	20.13	24.76	8.37	0.42	15.51	20.52	30.00	14.49	0.71	14.17	16.90	22.06	7.89	0.46

### 6.3.4 Quartile Analysis

After looking at the scaling of the median for the different synchronization schemes we analyze the first ( $Q_1$ ) and third quartile ( $Q_3$ ) as well as the interquartile range (IQR) of both GBH variants and the MCS-WP scheme. We discuss only the stress test with 48 processes. The data allows to assess the latency not only for the medial case (as done before) but also for more extreme cases where processes take longer for a complete LOCK/UNLOCK cycle. The IQR relative to the median reveals how much the latency can vary.

The obtained data is displayed in Table 2 for the case of 48 processes. In the table, all values are given in microseconds, except for the dimensionless relative IQR (IQR/ $Q_2$ ).

In case when there are exclusive locks in the workload, GBH’s best effort approach manifests in a high absolute and relative IQR (4.33 and 2.18). This indicates high deviations between the waiting times of the processes.

In contrast, the back-off version pays off in case of exclusive locks where all quartiles, IQRs and relative IQRs are significantly smaller. This confirms the importance of exponential back-off in the GBH strategy.

Compared to MCS-WP, the  $Q_3$  values of GBH with back-off are much better. In case of the shared-only and the mixed workload they are even better than the  $Q_1$  values of MCS-WP.

As already indicated in the previous sections, MCS-WP’s performance is nearly constant, even for  $Q_1$  and  $Q_3$ . The increase for  $Q_3$  in the mixed scenario can be attributed to failed attempts of shared locks when (preferred) writers are present. The absolute latency is generally higher than for GBH with back-off. Nevertheless, if writers precedence is needed, MCS-WP delivers acceptable performance with small relative deviations in the waiting times.

### 6.4 Application on the DHT

After the microbenchmark evaluation, the impact of the synchronization methods *GBH with backoff* and *MCS-WP* on the DHT application is measured. We focus on the case where a single writer updates the DHT and, at the same time, all other reader processes fetch data from the writer’s DHT portion. It is ensured that the writer gets the lock first and thus the readers have to wait for the lock, which puts traffic on the SCC’s memory system. The writer stores  $k \in \{32, 512, 1024\}$  bytes in the DHT. The measurements are done for different numbers of  $n \in \{1, 11, 47\}$  concurrent DHT readers.

For  $n$  equal 1 and 11, two different mappings are compared: a *close* mapping where all processes are placed in the same memory controller (MC) domain (cf. Fig. 1, cores 0-5 and 12-17), and a *distant* mapping where the readers use a different MC domain (cores 6-11 and 18-23). For  $n = 47$ , all cores are used. The writer always runs on core 0. We measure the time in microseconds for the PUT and UNLOCK operation in `DHT_write` (see Fig. 2) and present the median out of 101 measurements. The results are shown in Table 3.

**Table 3.** Impact of synchronization method on DHT writer.

$n$	map.	GBH with back-off			MCS-WP		
		$k=32$	$k=512$	$k=1024$	$k=32$	$k=512$	$k=1024$
0	—	5.4	18.8	34.3	4.9	18.4	33.9
1	close	6.3	20.2	35.6	8.7	22.0	37.4
1	distant	6.5	19.9	35.4	6.8	22.5	38.1
11	close	13.2	46.0	86.3	18.3	51.7	90.9
11	distant	12.9	45.3	86.1	10.6	24.6	41.0
47	—	90.6	136.4	244.4	23.1	51.7	89.7

For zero and one reader as well as 11 readers using the close mapping, the two synchronization methods have about the same impact on the writer’s performance. MCS-WP shows slightly higher impact due to the local but aggressive spinning whereas GBH uses the back-off. However, for the distant mapping and  $n = 11$ , MCS-WP’s impact on the writer reduces. This can be attributed to the readers spinning on another memory controller whereas for GBH they check the writer’s single local counter (see Fig. 3). This effect is emphasized for 47 readers, i.e. when the whole SCC is used. In that case, GBH has a 2.7 to 3.9 times higher impact on the writer than MCS-WP for  $k = 1024$  and 32, respectively. Note that due to the distributed data structures, the impact of MCS-WP does not change in case of the close mapping when going from  $n = 11$  to  $n = 47$  readers.

## 7 Discussion

From the experimental results in the last section and the properties of the synchronization schemes, the following aspects have to be discussed.

### 7.1 Synchronization Scheme for nCC Many-Cores

Although a tuned implementation for message transfer is available on the SCC, it does not pay off in case for MPI

passive target synchronization. Besides issues with deadlocks, which may be fixable, the observed latency is much higher than for the presented memory-based approaches that use uncached-memory accesses due to the immanent data transfer and processing overhead.

Contrary, the memory-based schemes perform with low latencies and nearly constant scaling and are therefore a favorable choice for nCC shared memory architectures like the SCC. Nevertheless, the attention must be paid to the distribution of data structures and their access pattern. Centralized data should either be avoided, like with the MCS-WP scheme, or access to its data must be rate-limited, as it is done by the GBH scheme with back-off.

While MCS-WP does not employ any centralized data structure, the efforts for maintaining the queue data structures and lock state appear to be higher than the counter-based approach of the GBH scheme. However, the latency for the synchronization is almost constant for every lock type mix. For GBH a trend for increased latency can be observed under heavy write contention (exclusive locks) which might increase further for higher core counts.

Although the microbenchmark results identify GBH with back-off as most efficient synchronization method the application benchmark revealed that MCS-WP is more appropriate for the SCC's architecture and performs better when the whole chip is used.

## 7.2 Support for LOCKALL

As depicted in Section 2, the MPI standard offers a LOCKALL routine to acquire a shared lock on all members of the window's process group. While this access scheme may apply to other applications, it is not necessary for the DHT use case and the support of LOCKALL was not in our focus. While the GBH scheme supports the implementation of LOCKALL, MCS-WP uses no centralized data structures which removes the support for a LOCKALL operation. In order to support this MPI routine, a global state needs to be added to the existing lock variables. However, given the results from Section 6.3 and the previous discussion, the benefits of such an effort might be questionable.

## 7.3 Consequences for MPI Implementations and Users

Depending on the actual application, the need for different synchronization scheme arises. An MPI application, such as a DHT, may give precedence either to processes that acquire exclusive locks (writers) or to ones that use shared locks (readers). As shown in Section 6.3 schemes like MCS-WP that support such a precedence exhibit similar performance compared to best-effort approaches ones like GBH. Nevertheless, an MPI library may offer those options for an improved performance of the application. Since the MPI standard makes no restriction on the actual scheme for the passive target synchronization such options can be offered by the library.

Consequently, an application may mandate a certain scheme that matches its data access pattern. To allow the choice of a synchronization scheme, MPI's Info objects can be used. Those can be specified per window with the MPI\_WIN\_SET\_INFO function call. This gives the MPI library hints on how to optimize internal mechanisms for the application's needs [17, §11.2.7]. In case for the passive target synchronization, all processes must use the same hint to ensure the same scheme is used. However, for different window objects, different hints can be set. That way, the user can control the precedence for lock/process types and may accept restricted but sufficient functionality, e.g missing LOCKALL support (see above).

## 8 Conclusion and Future Work

In this paper, we discussed and evaluated three different synchronization schemes for non-cache-coherent shared memory architectures, like the SCC. Two memory-based schemes known from the literature have been implemented for that platform. The evaluation shows that such schemes are well-suited for nCC many-core architectures both in terms of absolute performance and scalability. Despite they employ uncached memory operations, the approaches even outperform competitors that rely on SCC-optimized message passing. In general, the GBH best effort scheme with back-off performs well and is applicable to most use-cases. However, it suffers from centralized data structures. The distributed design of MCS-WP fits better to the architecture and has less impact on the discussed DHT application.

It was also shown that the MCS-WP scheme which gives precedence to writers can be used on nCC systems without scalability or severe performance degradations. However, a key aspect for the synchronization schemes is the avoidance of centralized data structures. Distributed data structures are essential for a good performance. For schemes like GBH that also use centralized data structures in addition to distributed ones, a back-off mechanism appears to be crucial for the median latency.

Depending on the applications, the choice of the synchronization scheme might be critical. We discussed that the MPI standard offers the possibility to inform the library about desired behavior. In case of a DHT, for example, support for Readers & Writers with writer-precedence may be beneficial. Hence, an info key like PASSIVE\_SYNC\_MODE could be set to `writer-precedence`, which in turn might be realized by MCS-WP. An info key value of `full-support` then indicates desired support for shared and exclusive locks as well as LOCKALL operations for which the GBH locks are a recommended choice.

Future work may also include an analysis how the presented approaches perform on contemporary processors

built from multiple *chipllets* when taking their inherent NUMA-design and hardware support for cache coherence into consideration.

## Acknowledgments

We thank Intel for lending an SCC system to the Institute of Computer Science at the University of Potsdam for the research in that domain.

## References

- [1] Hayder Al-Khalissi, Andrea Marongiu, and Mladen Berekovic. Low-Overhead Barrier Synchronization for OpenMP-like Parallelism on the Single-Chip Cloud Computer. In *Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University, November 29th-30th 2012, Aachen, Germany*, pages 26–31, 2012.
- [2] Hayder Al-Khalissi, Syed Abbas Ali Shah, and Mladen Berekovic. An Efficient Barrier Implementation for OpenMP-Like Parallelism on the Intel SCC. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 76–83, 2014.
- [3] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2015*, pages 215–226, New York, NY, USA, 2015. ACM.
- [4] Steffen Christgau and Bettina Schnor. One-sided communication in RCKMPI for the Single-Chip Cloud Computer. In *Proceedings of the 6th MARC Symposium, 19-20 July 2012, Toulouse, France*, pages 19–23. ONERA, The French Aerospace Lab, 2012.
- [5] Steffen Christgau and Bettina Schnor. Synchronization of One-Sided MPI Communication on a Non-Cache Coherent Many-Core System. In *29th International Conference on Architecture of Computing Systems (ARCS), Nürnberg, Germany, 2016*.
- [6] Steffen Christgau and Bettina Schnor. Design of MPI Passive Target Synchronization for a Non-Cache-Coherent Many-Core Processor. In *Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware: 27. PARS Workshop*, volume 34 of *Mitteilungen*. Gesellschaft für Informatik, 2017.
- [7] Steffen Christgau and Bettina Schnor. Exploring one-sided communication and synchronization on a non-cache-coherent many-core architecture. *Concurrency and Computation: Practice and Experience*, 29(15), 2017.
- [8] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2):13:1–13:42, February 2015.
- [9] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. Enabling highly-scalable remote memory access programming with MPI-3 one sided. *Scientific Programming*, 22(2):75–91, 2014.
- [10] William D. Gropp and Rajeev Thakur. An evaluation of implementation options for MPI one-sided communication. In *12th European PVM/MPI Users' Group Meeting, Sorrento, Italy*, pages 415–424, 2005.
- [11] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 649–662, Berkeley, CA, USA, 2016. USENIX Association.
- [12] Jason Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, February 2010.
- [13] Weihang Jiang et al. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In Dieter Kranzlmüller, Péter Kacsuk, and Jack J. Dongarra, editors, *11th European PVM/MPI Users' Group Meeting, Budapest, Hungary*, volume 3241 of *Lecture Notes in Computer Science*, pages 68–76. Springer, 2004.
- [14] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 603–615, Berkeley, CA, USA, 2017. USENIX Association.
- [15] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.
- [16] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In David S. Wise, editor, *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Williamsburg, Virginia, USA, April 21-24, 1991*, pages 106–113. ACM, 1991.
- [17] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.1. online, June 2015. <http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.
- [18] Timothy Prickett Morgan. More Knights Landing Xeon Phi Secrets Unveiled, March 2015. <http://www.nextplatform.com/2015/03/25/more-knights-landing-xeon-phi-secrets-unveiled/> accessed 2019-10-10.
- [19] Pablo Reble, Carsten Clauss, and Stefan Lankes. One-sided communication and synchronization for non-coherent memory-coupled cores. In *International Conference on High Performance Computing & Simulation, HPCS 2013*, pages 390–397. IEEE, 2013.
- [20] Pablo Reble, Stefan Lankes, Florian Zeitz, and Thomas Bemmerl. Evaluation of Hardware Synchronization Support of the SCC Many-Core Processor. In *4th USENIX Workshop on Hot Topics in Parallelism, Poster Paper*, 2012. <https://www.usenix.org/system/files/conference/hotpar12/hotpar12-final9.pdf>.
- [21] Randolph Rotta. On efficient message passing on the intel SCC. In Diana Göhringer, Michael Hübner, and Jürgen Becker, editors, *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*, pages 53–58. KIT Scientific Publishing, Karlsruhe, 2011.
- [22] Patrick Schmid, Maciej Besta, and Torsten Hoefler. High-performance distributed RMA locks. In Hiroshi Nakashima, Kenjiro Taura, and Jack Lange, editors, *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 - June 04, 2016*, pages 19–30. ACM, 2016.
- [23] Isaías A. Comprés Ureña, Michael Gerndt, and Carsten Trinitis. Wait-free message passing protocol for non-coherent shared memory architectures. In *19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria*, pages 142–152, 2012.
- [24] Isaías A. Comprés Ureña, Michael Riepen, and Michael Konow. RCKMPI - lightweight MPI implementation for intel's single-chip cloud computer (SCC). In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack J. Dongarra, editors, *18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece*, volume 6960 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2011.