W. Karl, J. Keller (Eds.): PARS 2017
Proc. 27th PARS-Workshop

Design of MPI Passive Target Synchronization for a Non-Cache-Coherent Many-Core Processor

Steffen Christgau[1], Bettina Schnor[1]

Abstract: Distributed hash tables are a common approach for fast data access. For this kind of application, a synchronization scheme with Readers and Writers semantic is well suited. This paper presents the design of an implementation of MPI passive target synchronization with Readers and Writers semantic. The implementation is discussed for the Single-Chip Cloud Computer, a non-cache-coherent many-core CPU with shared memory.

Keywords: MPI, one-sided communication, synchronization, distributed hash table

1 Introduction and Motivation

Distributed hash tables (DHT) 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 is a suited programming model for 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.

The Message Passing Interface standard (MPI) defines one-sided methods to perform remote memory accesses (RMA) which enable a shared-memory-like programming style. Gerstenberger et al. showed that implementing a DHT with MPI RMA can efficiently scale up to 32k cores on Cray machines [GBH13].

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) and *shared locks* (many readers).

Gerstenberger et al. published an approach for implementing passive target synchronization, but the proposed scheme does not account Readers & Writers semantics [GBH13]. Rather, a best-effort approach is used: Locks are acquired without respect to other processes. If an

[1] University of Potsdam, Insitute for Computer Science, Operating Systems and Distributed Systems, August-Bebel-Strasse 89, 14482 Potsdam, Germany, email: lastname@cs.uni-potsdam.de

acquisition does not succeed, all data structures will be released and the process will try again later with an exponential back off.

Regarding the RMA methods, the MPI standard also addresses non-coherent architectures. Although caches are performance critical components in CPUs, maintaining the cache coherence becomes a critical problem when both core counts and memory bandwidth increase [Mor15]. Therefore, the investigation of algorithms for non-cache-coherent (nCC) architectures becomes an important topic.

Consequently, this paper discusses the design for implementing passive target synchronization inside the MPI library for the Intel Single-Chip Cloud Computer (SCC). This is an experimental many-core chip that is comprised of 48 in-order Pentium (P54C) cores. It is not a product but a research vehicle [H+10]. 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 6×4 grid, to which four memory controllers are attached as well. A tile also provides a fast 16 KB message passing buffer (MPB) to the cores. The on-chip network allows to access both the MPBs as well as the main memory by regular memory load and store instructions.

The next section introduces MPI one-sided communication with the focus on passive target synchronization followed by a survey over related work. In Section 4, we present the design of passive target synchronization with Readers & Writers semantic. Section 5 concludes the paper and gives an outlook on future work.

2 MPI One-Sided Communication

Since version 2.0 the MPI standard defines one-sided communication (OSC) as an additional way to exchange data between processes. Different to point-to-point data exchange, communication is separated from synchronization. In addition, only one process (called *origin*) specifies the parameters (destination and message size, e.g.) of the communication. The *target* process is not involved in the communication from the API level.

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 using pre-defined operations.

2.1 Passive Target Communication

To communicate successfully, origin and target processes have to synchronize each other. The MPI standard defines two different classes for this purpose: *active* and *passive* target synchronization. Opposite to the active variant, target processes do not have to issue synchronization routines 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 allows only one process at a time to lock a window of a certain process. In contrast, a shared lock allows several processes to access a window. Shared locks are suited for concurrent read operations.

A process can lock *all* windows of the window object by using `MPI_WIN_LOCK_ALL`, but this sets a shared lock only. After the communication has been finished, the access epoch needs to be closed with `MPI_WIN_UNLOCK` or `MPI_WIN_UNLOCK_ALL`, respectively. The MPI standard defines that RMA operations issued during the epoch will have completed both at the origin and the target when the call that closes the access epoch returns [Mes15, p. 447].

2.2 An RMA based Implementation of a DHT application

In this section, the implementation of a distributed hash table where inserts and lookups are based on MPI RMA operations and passive synchronization is outlined.

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 1. Using MPI windows, a global address space is created which allows to access the distributed data via the window object.

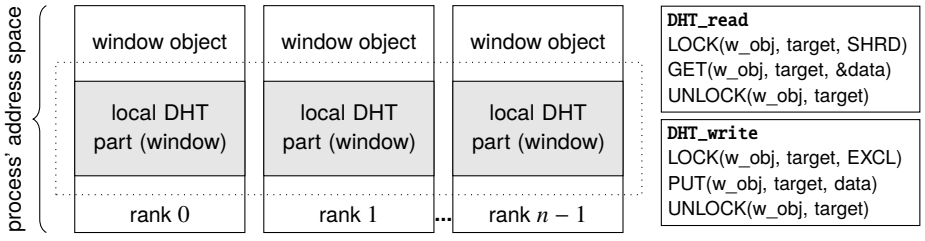

Fig. 1: Usage of MPI windows for a distributed hash table.

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 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. Both cases are illustrated in Figure 1.

2.3 Classification of Implementations

With respect to the synchronization calls, the MPI standard gives an implementer much freedom to realize the above methods [Mes15, p. 448]. The only guarantee is that communication is completed when an UNLOCK operation returns.

Gropp and Thakur [GT05] identified two classes of implementation options: *immediate* and *deferred*. For immediate synchronization a call like MPI_WIN_LOCK blocks until the according locks is acquired. This method is prone to process skew because the delay of another process may delay the synchronization. However, as soon as the processes are synchronized communication can be performed promptly which is beneficial especially for shared memory systems [GT05]. This also enables overlap of communication and computation. For the deferred scheme, the reverse applies. Here, the synchronization is non-blocking and the execution of the communication operations is delayed until the closing synchronization. While this is more tolerant to process skew, communication is lately performed and no overlap with computation is possible.

In previous work, we identified the *trigger-only* scheme as favorable since it combines the advantages of the other schemes: it is less prone to process skew but enables overlap of communication and computation [CS16]. In the trigger-only scheme, the process tries to acquire the lock at the beginning of an access epoch. If it does not succeed immediately it will not try again until a communication method (PUT or GET) is invoked.

For nCC many-core processors with shared memory, a deferred scheme does not provide advantages. Both immediate or trigger-only schemes should be considered. The scheme of Gerstenberger et al. [GBH13] is an immediate method. Jiang et al. [JLJ+04] propose that MPI_WIN_LOCK should be non-blocking, i.e. it should not wait until the lock is acquired, supporting a trigger-only solution.

While there may be applications where computations are performed inside a pair of LOCK and UNLOCK calls (access epoch), we assume that this is barely the case in a DHT application. Thus, a process only performs communication with a single target process during an access epoch with no chance for overlap. Hence, a trigger-only approach will provide no benefits. Consequently, an immediate method is considered in the remainder of this paper.

3 Related Work

RCKMPI [UGT12] 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. One-sided communication including the all synchronization styles is supported, but is based on messages as well. An implementation scheme for RMA communication for many-core architectures that avoids messages is discussed in [CS17]. 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 [CS16]. 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 [RCL13].

Gropp and Thakur present general concepts for implementing MPI one-sided synchronization operations [GT05]. Träff et al. [TRH00] mention a message-based MPI implementation for the nCC NEC SX-5. In case of passive target synchronization, the target has to check for synchronization messages in every of its MPI calls. Gerstenberger et al. [GBH13] present a design for contemporary Cray supercomputers that relies on the atomic RDMA operations provided by the hardware architecture. Similar work is provided in the earlier work of Jiang et al. [JLJ+04] for InfiniBand clusters. The authors also discuss general design issues for passive one-sided communication. Santhanaraman et al. investigate an implementation for clusters based on InfiniBand's atomic operations [SNP08] with cache coherent multi-core nodes [SBG+09]. However, Cardellini et al. [CFF16] state that despite these efforts MPI libraries, like MVAPICH and Open MPI, use two-sided approaches which require participation of the targets to implement the synchronization.

Recently, Schmid et al. have proposed a scheme for Readers & Writers locking dedicated for distributed memory architectures with RMA capabilities [SBH16]. The synchronization data structures are organized hierarchically in a distributed tree.

Subsuming the related work, the usage of atomic operations is common to optimized schemes. Nevertheless, there are no efforts in *passive target synchronization* for nCC many-core CPUs with shared memory like the SCC.

4 Passive Target Synchronization with Readers & Writers Semantic

For the implementation of a DHT, a Readers & Writers (R&W) semantic is beneficial and an implementation of MPI's RMA synchronization supporting this semantic would be preferable. The MPI API offers the suited methods: The writer may lock a resource exclusively via `MPI_WIN_LOCK` with the lock type `MPI_LOCK_EXCLUSIVE`, and a reader may check whether reading is allowed by calling `MPI_WIN_LOCK` with lock type `MPI_LOCK_SHARED` (see section 2.2). In the following, the data structures and algorithms for the MPI library are discussed.

Mellor-Crummey and Scott presented different synchronization algorithms for shared-memory multiprocessors [MS91b]. The essential advantage of the presented algorithms is that each process owns a local data structure which is used for spinning in case a lock could not be acquired immediately. Compared to shared objects on which all waiting processes spin (as used by Gerstenberger [GBH13]), this avoids contention on the memory interface. In the following, we show that the approach of Mellor-Crummey and Scott, using the same data structures for synchronization, can also be applied to nCC architectures.

Since the DHT application (cf. section 2.2) follows the R&W semantic and therefore does not need global locks (provided by `MPI_WIN_LOCKALL`), the proposed design can be built upon a classical R&W solution like given in [MS91b]. Among the presented algorithms in [MS91b], schemes for reader or writer preference are discussed. Depending on the workload, either type may be adequate for a DHT application. However, we focus on a writer preference lock algorithm in this paper in order to allow readers to fetch the most recent data.

4.1 Data Structures

The algorithm is built on a lock data structure L_i that is allocated per each local window at process i (see Figure 2) in shared memory. The lock data structure was proposed in [MS91b] and contains distributed linked lists, known from MCS locks [MS91a], that serve as wait queues for readers and writers.

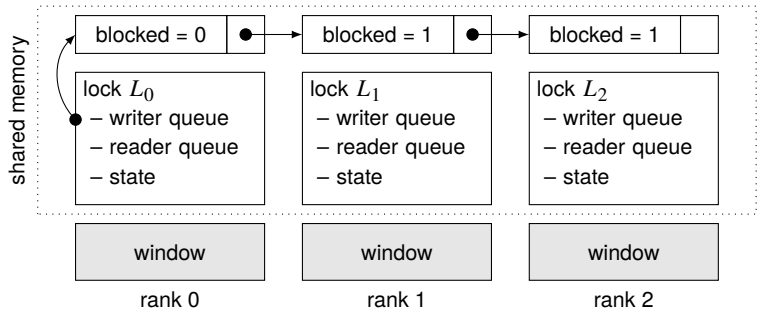

Fig. 2: Data structures for Readers & Writers synchronization.

In order to ensure that writers gain precedence over readers, an unsigned integer variable `state` in the lock L_i is used. It counts the number of active readers, but also records the presence of writers and interested readers with flags. Interested readers are blocked upon arrival when an active or interested writer is present. This ensures that writers gain preference over readers that arrive later.

For the protoype implementation of MPI's passive target synchronization, each MPI process manages locally an array `entry` to store an entry element for every possible target. An entry element consists of a boolean flag `blocked` that indicates whether the process is blocked. A

process which has to wait spins on this flag until it is set to *unblocked* by another process. Additionally, the entry contains a pointer to the next list entry within the corresponding wait queue. A list entry is allocated in shared memory to allow distributed wait queues, but it is local to the calling process (see Figure 2).

4.2 Algorithm

Algorithms 1 and 2 show the usage of distributed readers and writers lock of Mellow-Crummey and Scott [MS91b] inside the MPI implementation. The re-used functions `start_read, start_write, end_read` and `end_write` have two parameters. The first parameter is the lock L of the target window and the second parameter points to the entry element describing the calling process.

If a reader or writer arrives at the lock, it enqueues itself to the appropriate wait queue by creating a new list entry e (see Algorithm 1) and inserting itself atomically into the list (within `start_read` and `start_write`), respectively. Figure 2 illustrates an example, where three writers want to access the window at rank 0 and where rank 0 successfully obtained the lock. The entering process indicates its intention by setting the according fields in the `state` variable of the target's lock L. If the process obtains the lock immediately, the process' wait queue entry is set to unblocked. If a reader becomes unblocked, it also wakes up all other readers that entered before the next writer to allow concurrent read operations.

In addition, the lock type is stored in the list entry e (see Algorithm 1) to call the corresponding routine in the `unlock` method.

As soon as a process releases a lock, it unblocks its successor (either reader or writer) by setting the boolean flag in the successor's entry element to false. This is done within `end_read` and `end_write`. From the target rank the entry element e of the calling process can be derived which is the second parameter in `end_read` and `end_write`.

4.3 Implementation on the SCC

The algorithms presented in [MS91b] require five atomic operations for the manipulation of the `flag` variable of a lock L_i: `fetch_and_{store,or,and}`, `atomic_add`, `compare_and_swap`. On the SCC, all these operations can be built using the single test-and-set register that is available on every core.

The shared data structures are accessed with uncached memory operations to account the nCC aspects of the SCC. This is achieved with the help of an SCC-specific device driver inside the Linux kernel that enables to map arbitrary memory regions with specific caching policies. Uncached reads and writes prevent, e.g., infinite waits caused by reading cached blocked variable of a list entry that has been updated in the RAM but was not updated in the

Algorithm 1 Lock Acquisition

```
function LOCK_SHARED(target, win)
    e = entry[target] = alloc_list_entry()
    e.lockState = SHARED
    start_read(win.locks[target], e)
end function

function LOCK_EXCLUSIVE(target, win)
    e = entry[target] = alloc_list_entry()
    e.lockState = EXCLUSIVE
    start_write(win.locks[target], e)
end function
```

Algorithm 2 Lock Release

```
function UNLOCK(target, win)
  e = entry[target]
  if e.lockState == SHARED then
    end_read(win.locks[target], e)
  else
    end_write(win.locks[target], e)
  end if
end function
```

Fig. 3: Algorithms for R&W lock implementation. The `{start|end}_{read|write}` functions are reused from Mellow-Crummey's and Scott's solution [MS91b].

cache due to the missing coherence. Hence, on the SCC the processes are not doing local-spinning. This seems to be a drawback, but the synchronization data structures are allocated in the memory which is closest to the process. Therefore, the spinning is distributed over all the four memory controllers.

Experiments on the SCC prove the scaling of this approach. Figure 4 shows the speedup of an application including synchronization depending on the location of the synchronization data. The latter is varied between a distributed variant and centralized versions which use one of the four memory controllers. The baseline for the presented speedup is the sequential version of the application. Application data is always distributed across the memory controllers. So, memory accesses causes by the application (not by the synchronization) from the MPI ranks 0 to 11 go to controller 0. Controller 1 handles application traffic for process 12 to 23, and so forth.

The results in Figure 4 clearly show that synchronization data should be distributed and the SCC architecture sustains the traffic generated by both the applications memory accesses and the uncached ones caused by the synchronization. The centralized versions using only a single memory controller (labeled as *controller x* in the Figure) suffer from contention on the memory controllers. Here, the memory traffic from the application plus the synchronization traffic apparently cause to much load on the memory controllers. Due to the distributed application data, the performance degradation depend on the number of MPI processes.

Alternatively to the conventional RAM, the data structures could be stored in the fast on-chip Message Passing Buffer (MPB) which is local to the cores. This would enable pure local spinning. However, the MPB is used by the MPI implementation for two-sided communication and therefore not available for other purposes.

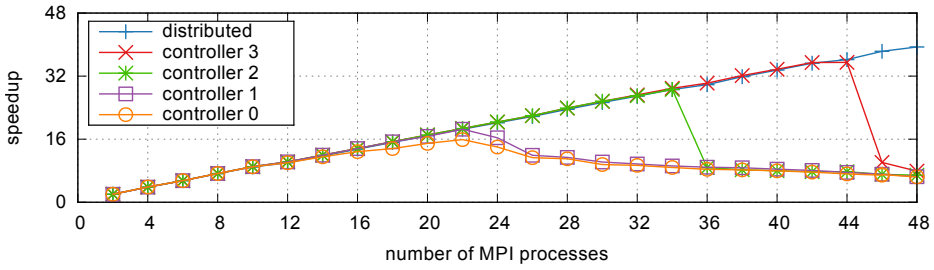

Fig. 4: Scalability depending on the allocation strategy of the synchronization data.

4.4 Discussion

We have shown that the data structures and algorithms proposed by Mellor-Crummey and Scott can also be applied to nCC architectures.

The presented approach has the following characteristics:

Concurrency Each window of a process has its own lock data structure L_i. Therefore, concurrent accesses of different processes on different targets are possible.

Per-process RW semantics The accesses to the same target window are synchronized with readers and writer semantic.

Contention avoiding spinning Since the wait list elements (e) are allocated in a distributed fashion and located close to the spinning process/core (according to [MS91b]), spinning on these structures avoids contention.

Jiang et al. [JLJ+04] state design issues for efficient passive target synchronization. It should only add little overhead, and when there is high contention on the lock there should be as little delay as possible. The presented measurement results show that this is fulfilled on the SCC by distributing the spin load over all four memory controllers.

Further, origin processes should be able to continue with their operations without participation of the targets. The design for the SCC exploits the possibility to create shared memory by LUT re-configuration. Hence, the synchronization data structures can be accessed directly without any participation of the respective target process. This also avoids both the usage of asynchronous agents [Mes15], such as threads, and the usage of messages, which have been proven to introduce higher latencies in case of active target synchronization [CS16].

5 Conclusion and Future Work

This paper shows how a classical Readers & Writers solution can be reused to implement MPI's passive target synchronization. The implementation is discussed for the SCC, a shared memory non-cache-coherent many-core processor. Synchronization data structures are allocated in shared memory, but accessed with uncached read/write operations. We argue that an immediate scheme is best suited for passive synchronization in a DHT application.

The presented approach fulfills the requirements from [JLJ+04]: little overhead under contention (due to distributed spin load), no participation of targets, and co-existence of point-to-point and one-sided communication.

Currently, we are working on a prototype. Further, we are working on a proposal for MPI RMA where an application may specificy the locking semantic, for example, Readers or writers preference.

Acknowledgement

We thank Intel for lending us an SCC system for our ongoing research in that domain.

References

[CFF16] Valeria Cardellini, Alessandro Fanfarillo, and Salvatore Filippone. Overlapping Communication with Computation in MPI Applications. Technical Report RR-16.09, Universit'a di Roma Tor Vergata, Dipartimento di Ingegneria Civile e Ingegneria Informatica, Rome, Italy, February 2016. online, `http://www.opencoarrays.org/uploads/6/9/7/4/69747895/mpiprog.pdf`, accessed 2017-03-13.

[CS16] Steffen Christgau and Bettina Schnor. Synchronization of One-Sided MPI Communication on a Non-Cache Coherent Many-Core System. In *ARCS 2016 - 29th International Conference on Architecture of Computing Systems, Workshop Proceedings, April 4-7, 2016, Nuremberg, Friedrich-Alexander University, Erlangen-Nürnberg*. VDE Verlag / IEEE Xplore, 2016.

[CS17] 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.

[GBH13] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 53:1–53:12. ACM, 2013.

[GT05] William D. Gropp and Rajeev Thakur. An Evaluation of Implementation Options for MPI One-Sided Communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Proceedings*, volume 3666 of *LNCS*, pages 415–424. Springer, 2005.

[H+10] 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.

[JLJ+04] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, Darius Buntinas, Rajeev Thakur, and William D. Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, volume 3241 of *Lecture Notes in Computer Science*, pages 68–76. Springer, 2004.

[Mes15] 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`.

[Mor15] 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 2017-03-13.

[MS91a] 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.

[MS91b] 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.

[RCL13] 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.

[SBG+09] Gopalakrishnan Santhanaraman, Pavan Balaji, K. Gopalakrishnan, Rajeev Thakur, William Gropp, and Dhabaleswar K. Panda. Natively Supporting True One-Sided Communication in. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2009, Shanghai, China, 18-21 May 2009*, pages 380–387, 2009.

[SBH16] P. Schmid, M. Besta, and T. Hoefler. High-Performance Distributed RMA Locks. In *Proceedings of the 25th Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)*, Jun. 2016.

[SNP08] Gopalakrishnan Santhanaraman, Sundeep Narravula, and Dhabaleswar K. Panda. Designing passive synchronization for MPI-2 one-sided communication to maximize overlap. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–11. IEEE, 2008.

[TRH00] Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The Implementation of MPI-2 One-Sided Communication for the NEC SX-5. In *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*, page 1. IEEE Computer Society, 2000.

[UGT12] Isaías A. Comprés Ureña, Michael Gerndt, and Carsten Trinitis. Wait-Free Message Passing Protocol for Non-coherent Shared Memory Architectures. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490 of *Lecture Notes in Computer Science*, pages 142–152. Springer, 2012.