Synchronization of MPI One-Sided Communication on a Non-Cache-Coherent Many-Core System

Steffen Christgau and Bettina Schnor
Institute for Computer Science
University of Potsdam
August-Bebel-Straße 89
14482 Potsdam, Germany
{christgau,schnor}@cs.uni-potsdam.de

***Abstract*—This paper discusses the design and implementation of MPI's general active target synchronization on the Intel Single-Chip Cloud Computer, a non-cache-coherent many-core CPU. Measurements show a performance benefit of a factor of four compared to the default SCC-tuned MPI implementation and demonstrate the feasibility of implementing efficiently a shared memory protocol despite the lack of cache coherence. Further, a classification of implementation designs of MPI's general active target synchronization is presented.**

I. Introduction

Cache coherence has been present in multi-CPU and multi-core CPUs since decades. However, with increasing memory bandwidths the bandwidth of the coherence interconnect traffic becomes challenging [1]. This problem gets even more critical with increasing core count in many-core CPUs. Therefore, the investigation of algorithms for non-cache-coherent architectures becomes an important topic.

Previous work [2] has shown that true one-sided communication based on shared memory systems is feasible even when cache coherence is managed in software. This paper presents the design and performance analysis of MPI's synchronization methods used in the SCOSCo approach [2] for one-sided communication (OSC) on the Intel Single Chip Cloud-Computer (SCC) [3], a non-cache-coherent (nCC) architecture.

Since version 2.0, the Message Passing Interface (MPI) standard includes one-sided communication which has been extended in the subsequent versions [4]. Figure 1 shows pseudo-code for only two communicating processes.

Within the MPI standard, memory for remote memory access (RMA) operations is logically attached to a *window* object. Remote memory is addressed together with that window object and the *rank*, the numerical process identifier. The creation of a window object is a collective operation, i.e. all processes inside a group (*communicator*) have to participate in the construction. `MPI_WIN_CREATE` can be used to create a window inside a given communicator and associates that communicator to the created window object. Subsequent RMA operations, like `PUT`, `GET` and `ACCUMULATE` [4, §11.3], operate with the returned window object, but must be non-blocking according to the standard.

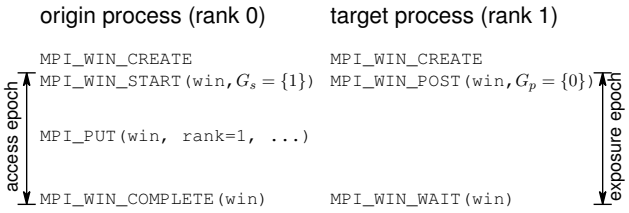

Fig. 1. PSCW Synchronization in MPI One-Sided Communication

II. OSC Synchronization

While message-passing communication includes both data transfer and synchronization between sender and receiver, these functions are separated in the OSC model. In general, RMA operations are only allowed after synchronization calls have been issued. An *origin* performs accesses during an *access epoch* only. Vice versa, a *target* allows such accesses only within an *exposure epoch*.

Previous work already discussed the optimization of MPI's fence synchronization [5]. This paper addresses *general active target communication*, which provides a flexible mechanism to synchronize processes. Based on the names of the methods that have to be invoked in that scheme, the synchronization is often also referred to as PSCW synchronization. An example of its usage for one origin and one target process is shown in Figure 1.

In contrast to the fence synchronization, PSCW allows to synchronize only a subset of the processes that created the window object. In addition, it allows an application to explicitly open access and exposure epochs. An origin opens and closes an access epoch with the `MPI_WIN_START` and the `MPI_WIN_COMPLETE` calls. In the `START` operation, an origin names the processes it (potentially) communicates with during the opened access epoch. The processes are given as a list of ranks, called *start group* G_s. On the other hand, a target invokes `MPI_WIN_POST` to open an exposure epoch and names the processes from which RMA operations are allowed (*post group*, G_p). However, the named processes are actually not required to access the target's window, but only these are allowed to do so. At the end of an exposure epoch, `MPI_WIN_WAIT` is issued to wait for a notification

TABLE I
CLASSIFICATION OF SYNCHRONIZATION PROTOCOLS FOR MPI ONE-SIDED COMMUNICATION.

class	epoch start	communication	overlap
deferred	non-blocking	delayed to epoch's end	no
immediate	blocking	prompt	yes
trigger-only	non-blocking	prompt	yes

that RMA operations have been completed and the window can be accesses without remote interference.

III. CLASSIFICATION OF MPI SYNCHRONIZATION IMPLEMENTATIONS

MPI implementations may implement the synchronization calls in different ways. Gropp and Thakur [6] define two options: *immediate* and *deferred*.

For deferred synchronization (used in MPICH, e.g.), the execution of methods that open epochs and perform RMA operation is delayed until the end of an epoch. This allows an MPI implementation to merge and optimize multiple of such communication calls. The downside is that optimizations like the overlap of communication and computation are not possible.

Within the immediate class (employed by MVAPICH for InfiniBand), the synchronization calls at the beginning of an epoch (`POST` and `START`) perform the synchronization immediately when they are invoked. Usually, this leads to blocking implementations. However, origin and target are ready for communication after synchronization. Since communication calls need to be non-blocking, immediate synchronization offers the possibility for communication computation overlap.

In addition to the classification from [6], we identify a third class that combines the advantages from deferred and immediate synchronization (see Table I). In the *trigger-only* variant, the starting synchronization calls initiate operations but do not block to wait for completion of the synchronization. This task is shifted to the communication calls, like `PUT` and `GET`, that check for those target processes to be synchronized. If the synchronization is still not finalized, the communication call blocks until its single target process has synchronized. In such a case, the call violates the standard. The benefit of this approach (presented in [7]) is that a process waits for process synchronization when it is actually required. In addition, after the initial synchronization with a particular target is completed, all subsequent communication with that process can be performed promptly.

IV. MPI PROCESS SYNCHRONIZATION ON THE SCC

No current MPI implementation efficiently supports PSCW synchronization on an nCC many-core chip. This section presents the design of such a scheme for the Intel SCC.

A. The Intel SCC

The Intel SCC [3] is an experimental tiled many-core chip with 48 Pentium (P54C) cores. Each tile contains two

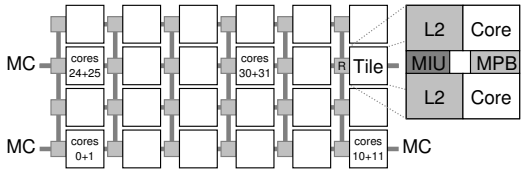

Fig. 2. Overview of the Intel SCC

cores. A router connects each of the 24 tiles to an on-chip network as illustrated in Figure 2. Also, four DDR3 memory controllers are attached to the network. The mesh interface (MIU) unit on each tile translates memory accesses by the cores to network packets and vice-versa. The unit performs a configurable address translation that determines the actual destination of a memory access. If more than one MIUs are configured to translate to the same network address shared memory is created. Further, the MIU contains one atomic test-and-set register per core that has mutex-like semantics.

Although each core has two cache levels, the hardware provides no support for cache coherence at all. This complicates the usage of shared memory in the common sense. In consequence, message passing is the most adequate programming model for this on-chip cluster. To support this, each tile possesses a 16 KB SRAM-based so-called *Message Passing Buffer (MPB)* that enables fast on-chip data transfer without using the external DDR3 memory.

In the default configuration, the MIU provides access to 16 MB per memory controller that is shared between all cores and is called *legacy shared memory*. However, accesses to this memory have to be performed carefully due to the missing cache coherence.

In addition, different memory types are supported. Most relevant for this work is the non-cacheable memory (NCM) type which bypasses all cache levels. It has been shown that usage of this memory can provide better latencies over other (cached) memory types [8].

On top of SCC's hardware, the MPI implementation RCKMPI [9] was developed. It is based on MPICH [10], thus inherently message-based, and exploits the MPB. Due to the derivation from MPICH, the implementation of one-sided communication, including the process synchronization, is also message-based [11].

B. SCOSCo process synchronization

The design of the PSCW synchronization is developed in the context of the implementation of SCOSCo [2], a software-managed cache coherence protocol for one-sided communication.

To efficiently implement the PSCW synchronization pattern on the SCC, the message based approach of RCKMPI is dropped. Instead, the presented solution is based on an optimization of MVAPICH for shared memory systems [7] which belongs to the trigger-only class. It is demonstrated in the following that this approach can be successfully implemented on the non-cache-coherent Intel SCC.

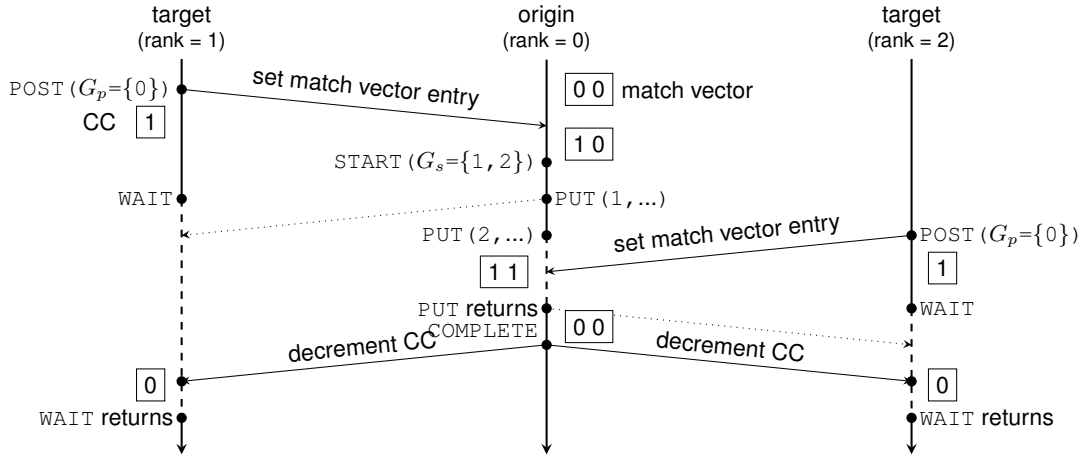

Fig. 3. Sequence diagram of the implemented synchronization protocol for two target processes (left and right) and a single origin (center).

The SCC's ability to define shared memory is exploited to store and access synchronization data. For this purpose, a memory region called *window database* is reserved inside the legacy shared memory (see above). Therein, the required synchronization data structures are allocated. A bit vector for the start of an epoch (called *match vector*) and completion counters are used as space-efficient means for synchronization. For polling these data structures efficiently, uncached memory is used to circumvent coherence problems.

1) Window Creation: The synchronization data structures are allocated in the legacy shared memory which is accessible by all processes in the system. The allocation is distributed among the four memory controllers to avoid contention of the controllers when the match vector and completion counters are polled. This issue was observed in preliminary experiments especially when synchronization was executed in memory-bound applications. Therefore, depending on the core a MPI process is running on, it allocates the synchronization data inside the legacy shared memory such that the nearest (Manhattan distance) memory controller is used for accesses.

The actual allocation is performed during the collective window creation. Since it is unknown to a process whether it becomes target or origin for the created window, space for both match vector (for origins) and completion counter (for targets) is reserved. The size of the bit vector is equal to the window's communicator size. Memory is allocated atomically (with the help of the test-and-set registers) by reading and advancing an offset, which is stored in the windows database, by the amount of allocated memory.

The obtained offset of the reserved memory is exchanged by an MPI all-to-all operation between all the n processes which create the window. Consequently, each process knows the base location of all other processes' synchronization data. Since the match vector size is known locally (and is equal among all processes), the position of the completion counter can be derived locally as well. This approach leads to data duplication of the exchanged offsets which scales with n^2. However, it can be compensated by storing all offsets in the shared memory as well. Anyhow, this is left out of the prototypical implementation.

2) START and POST: When a `POST` operation is issued by a target process, it iterates through the post group G_p that contains all origins (see Section II). To notify each of them, the address of the according match vector in the legacy shared memory is determined. Then, the test-and-set register of the origin's core is locked to prevent concurrent modifications (see Algorithm 1) Subsequently, the byte containing the according bit is read, locally modified and written back using uncached memory. Cached memory is unsuitable for this use-case since it would require an explicit write-back of the according cache line. Such an operation is not supported by the SCC's cores. Only a write-back including an invalidation of the whole cache is available in the instruction set. However, it is obviously far slower than using uncached memory. In addition, the match vector is not accessed by the targets for any other purpose which makes caching unnecessary anyway.

Algorithm 1 Pseudocode for START and POST

```
function START(G_s: Group)
    start_ranks ← ranks of procs ∈ G_s
end function

function POST(G_p: Group)
    CC ← |G_p|                          ▷ init completion counter
    for all origins ∈ G_p do            ▷ notify all origins
        core ← CORE_OF_PROC(origin)
        LOCK_TSR(core)
        match_vector[core][local_rank] ← 1
        UNLOCK_TSR(core)
    end for
end function
```

On the origin side, the `START` function performs only bookkeeping operations, like storing the ranks of the processes

from the start group G_s (see Section II). Polling the match vector is shifted to the communication calls, like `PUT` and `GET`. However, these calls only check for the according target process to have synchronized. Thus, the implementation is classified as trigger-only (see Section III).

Polling is performed with uncached memory since cached reads would prevent the origin to observe a post operation. In addition, caching of the match vector is dangerous in any case. Suppose a match vector is stored in the origin's cache and that it can successfully communicate with all targets of the current access epoch. Independent from this, another target of a subsequent access epoch (but of the same window) performs its `POST` call and modifies the match vector in main memory. Due to the missing coherence, the cached copy is unchanged, but now becomes invalid. In case the origin's cache evicts the line (due to memory accesses by computation, e.g.) that contains the cached and outdated match vector, it overrides the manipulated match vector in the main memory. This would cause a deadlock as the origin would check for a post operation that actually took place but its effect was destroyed by the origin itself and leads to an infinite loop.

However, to speed up polls of already synchronized targets, a local and cacheable copy of the match vector is employed. Inside a communication call the cached vector copy is checked first. If there was no `POST` operation, the uncached match vector is polled and upon detection of the target's post operation, the cached copy is updated.

3) COMPLETE and WAIT: At the end of its access epoch, i.e. in `COMPLETE`, the origin resets the entries corresponding to the targets in G_s in the match vector and in its copy. Subsequently, it decrements the targets' completion counters. However, the notification of completing an access epoch can only be performed after an origin has successfully started its matching exposure epoch. Only then, the target's completion counter is in a valid state and can be modified when the origin completes. Thus, an origin first ensures that every targets in G_s has synchronized (see Algorithm 2).

Then, it iterates through all targets and decrements their completion counter. Similar to the targets' accesses on the match vector, the completion counter is accessed with uncached memory (see Figure 3). Further, to make the decrement atomic, the test-and-set registers of the target's core are used.

On the target's side, the `WAIT` call polls the completion counter with uncached memory as well to observe the decrements made by the origins. Polling is performed until the counter reaches zero.

Algorithm 2 Pseudocode for COMPLETE and WAIT

```
function COMPLETE
    for all targets ∈ start_ranks do                ▷ see START
        repeat                                      ▷ busy wait for all targets
        until match_vector[local_core][target] == 1
        match_vector[local_core][target] ← 0
    end for

    for all targets ∈ start_ranks do                ▷ notify all targets
        core ← CORE_OF_PROC(target)
        LOCK_TSR(core)
        CC[core] ← CC[core] − 1                     ▷ decrement remote CC
        UNLOCK_TSR(core)
    end for
end function

function WAIT
    repeat                                          ▷ busy waiting
    until CC == 0                                   ▷ poll with uncached reads
end function
```

V. Experimental Evaluation

To evaluate the performance of the SCOSCo approach, the runtime required for synchronization is analyzed. 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 together with the modifications from [11] into the MPICH sources. The synchronization functions were overridden to implement the approach presented in this work. The resulting MPI library was compiled with optimization enabled (`-O2`).

A. Microbenchmark

For measuring the synchronization performance, a microbenchmark was created that does not include any one-sided communication like PUT and GET. This enables a fair comparison with other implementations, for example MPICH's deferred approach that performs queued communication in the `COMPLETE` routine.

Algorithm 3 Microbenchmark Pseudocode

```
for i = 0 ... 1000 do
    if rank == 0 then
        t_{s,i} ← TIME(START(G_s = {1 ... k}))
        t_{c,i} ← TIME(COMPLETE)
    else
        t_{p,i} ← TIME(POST(G_p = {0}))
        t_{w,i} ← TIME(WAIT)
    end if
end for
```

Consequently, the microbenchmark only uses the PSCW methods (cf. Figure 1). Further, it consists of a single origin process that synchronizes with a given number of k targets as illustrated in Algorithm 3. The runtime of each of the four PSCW routines is measured by reading the per CPU timestamp counter with the `RDTSC` instruction before and after the call. The synchronization sequence is repeated 1001 times. In case of the targets, all recorded times $t_{p,i}$ and $t_{w,i}$ from all k

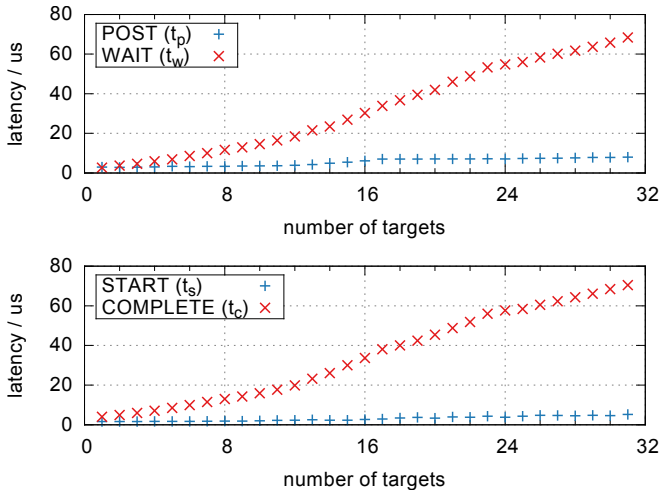

Fig. 4. Scaling of PSCW synchronization on the SCC for the target (top) and origin (bottom) processes.

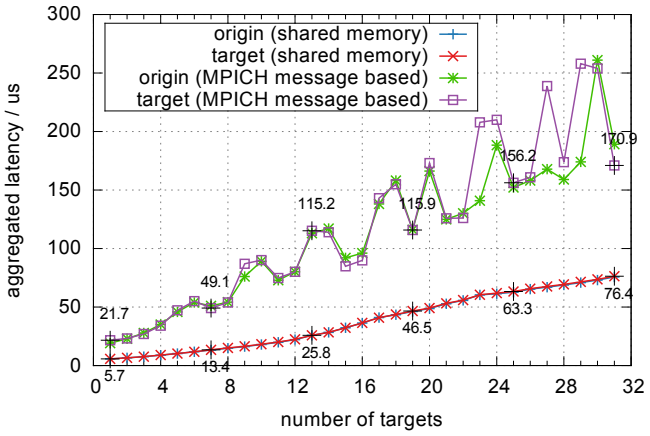

Fig. 5. Comparison of target and origin synchronization times t_o and t_t. Numbered labels inside the plot are given for the target processes.

targets are collected and the medians t_p and t_w are obtained from the $k * 1001$ samples. Besides outliers caused by process scheduling only little deviation from the reported values was observed: For all measurements, the first and third quartile of the four measured timings never deviated by more than 5% for origin and 10% for targets from the according median.

A synchronization at the beginning of each synchronization loop is omitted since after one iteration the processes are synchronized anyway. Additionally, using an MPI barrier caused fluctuations in the measurements because the origin process might leave the barrier later than some targets depending on the number of started processes. In the experiments, the benchmark was compiled with optimization enabled (`-O2`).

B. Scaling

The microbenchmark was used to analyze the scaling of the implemented synchronization scheme. It was executed for up to 32 processes (1 origin, up to $k = 31$ targets). The SCC provides more cores but using (nearly) all of them questions the usage of the PSCW scheme. In such cases, fence is more appropriate.

The results for target and origins are shown in Figure 4. The cores with numbers up to 31 (see Figure 2) were used for this analysis.

The presented results show a nearly constant runtime for the `START` and `POST` operations. For `POST`, this can be accounted to the usage of only one origin process in the experiment. In case of `START`, the constant runtime is attributed to the trigger-only design of the synchronization protocol which does not involve any communication in that routine. Thus, the observed latency is introduced by MPI library overhead only.

Contrary, the `COMPLETE` and `WAIT` runtime exhibit linear scaling. Concerning the `COMPLETE` call, this has two reasons. First, the origin needs to notify all targets which is done in a loop and thus causes linear scaling behavior. To do so, it has to wait for all target to be synchronized (cf. Section IV-B) which also scales linearly. This consequently affects the `WAIT` on the target side which therefore shows linear scaling as well.

C. Comparison to MPICH's Message based approach

Next, the SCOSCo synchronization was compared against the default RCKMPI implementation that uses MPICH's message based CH3 synchronization [11] but transfers messages over the on-chip MPB (see Section IV-A). The microbenchmark and the methodology from Section V-B were reused. However, the recorded median times of the origin, i.e. t_s and t_c, were summed, giving the total time t_o required to perform the PSCW synchronization on the origin side. The same was done for the recorded median of the target times (t_p and t_w) leading to t_t. Figure 5 shows the obtained t_o and t_s for both RCKMPI/MPICH and SCOSCo as well as for different number of targets.

The results reveal that despite RCKMPI's usage of the fast on-chip message passing buffer [11], the performance of the message-based synchronization from MPICH delivers latencies more than four times higher (e.g., 13 targets: 25.8 µs vs. 115.2 µs) than the SCC-aware approach. This can be explained by the overhead of message assembly, sending, reception and processing by MPICH's internal progress engine. This underlines that even in presence of a tuned implementation for message transfer, a shared memory approach is more appropriate on the SCC. Bearing in mind, this is possbile without hardware support for cache coherence.

VI. Summary and Conclusion

We presented the design and implementation of the SCOSCo PSCW synchronization on the Intel SCC, a non-cache-coherent many-core system. The developed approach leverages match vector and completion counters located in shared memory. Even without cache coherence, the approach delivers linear scaling and outperforms optimized message based synchronization that utilizes fast on-chip memory. This

shows that efficient synchronization is feasible also on nCC architectures.

REFERENCES

[1] T. P. Morgan, "More Knights Landing Xeon Phi Secrets Unveiled," Mar. 2015, accessed 2015-11-02. [Online]. Available: http://www.nextplatform.com/2015/03/25/more-knights-landing-xeon-phi-secrets-unveiled/

[2] S. Christgau and B. Schnor, "Software-managed cache coherence for fast one-sided communication," in *Proceedings of the Seventh International Workshop on Programming Models and Applications for Multicores and Manycores, Barcelona, Spain*, P. Balaji and K.-C. Leung, Eds., 2016, to appear.

[3] J. 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*, Feb. 2010, pp. 108 –109.

[4] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 3.1," online, Jun. 2015, http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[5] P. Reble, C. Clauss, and S. Lankes, "One-sided communication and synchronization for non-coherent memory-coupled cores," in *International Conference on High Performance Computing & Simulation, HPCS 2013*. IEEE, 2013, pp. 390–397.

[6] W. D. Gropp and R. 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*, ser. LNCS, vol. 3666. Springer, 2005, pp. 415–424.

[7] P. Lai, S. Sur, and D. K. Panda, "Designing truly one-sided MPI-2 RMA intra-node communication on multi-core systems," *Computer Science - R&D*, vol. 25, no. 1-2, pp. 3–14, 2010.

[8] R. Rotta, "On efficient message passing on the Intel SCC," in *Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*. KIT Scientific Publishing, Karlsruhe, 2011, pp. 53–58.

[9] I. A. C. Ureña, M. Riepen, and M. Konow, "RCKMPI - lightweight MPI implementation for Intel's single-chip cloud computer (SCC)," in *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011. Proceedings*, ser. LNCS, vol. 6960. Springer, 2011, pp. 208–217.

[10] W. Gropp, "MPICH2: A new start for MPI implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, Proceedings*, ser. LNCS, vol. 2474. Springer, 2002, p. 7.

[11] S. Christgau and B. 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*. ONERA, The French Aerospace Lab, 2012, pp. 19–23.