High-Throughput Sockets over RDMA for the Intel Xeon Phi Coprocessor

Aram Santogidis[1,2](✉) and Spyros Lalis[3]

[1] Maynooth University, Maynooth, Ireland
aram.santogidis@cern.ch
[2] CERN, Geneva, Switzerland
[3] University of Thessaly, Volos, Greece
lalis@uth.gr

Abstract. In this paper we describe the design, implementation and performance of Trans4SCIF, a user-level socket-like transport library for the Intel Xeon Phi coprocessor. Trans4SCIF library is primarily intended for high-throughput applications. It uses RDMA transfers over the native SCIF support, in a way that is transparent for the application, which has the illusion of using conventional stream sockets. We also discuss the integration of Trans4SCIF with the ZeroMQ messaging library, used extensively by several applications running at CERN. We show that this can lead to a substantial, up to $3x$, increase of application throughput compared to the default TCP/IP transport option.

Keywords: RDMA · Fast data transfer · Stream sockets · Manycore processors · Intel Xeon Phi · ZeroMQ · High performance computing

1 Introduction

One of the systems used at CERN to process the data generated from the LHC experiments [12] is the O^2 online-offline distributed system, developed by the ALICE collaboration [1]. O^2 consists of over hundred different kinds of processes that perform data acquisition from the particle detectors, particle trajectory reconstruction, data compression and storage, as well as detector monitoring and calibration. They form a distributed data processing pipeline, interconnected via a message passing fabric based on the ZeroMQ [7] and NanoMSG [18] libraries.

With the introduction of the Intel Xeon Phi coprocessor [5], we started to investigate the possibility of taking advantage of this manycore architecture in order to increase the efficiency of O^2 workloads. Indeed, several O^2 computations could greatly profit from the high core count and high memory bandwidth of the Intel Xeon Phi coprocessor. However, our tests [17] have shown that the host-coprocessor communication throughput of ZeroMQ and NanoMSG over TCP/IP is up to $20x$ lower than what could be achieved using the RDMA support of the Symmetric Communication Interface (SCIF) [8], the native transport mechanism of the Intel Xeon Phi platform.

© Springer International Publishing AG 2017

For this reason, we decided to provide a high-throughput transport service over SCIF-RDMA, called Trans4SCIF, with two goals in mind. On the one hand, it should be straightforward to integrate this transport with the ZeroMQ messaging library, so that the O^2 stack can enjoy improved performance in a transparent way. On the other hand, this transport should be easy to use in other applications as well, offering the familiar abstraction of streaming sockets.

This paper describes the implementation of Trans4SCIF and discusses its integration with ZeroMQ. The main contributions are: (i) we present a socket-based RDMA-capable transport library with streaming semantics for the Intel Xeon Phi coprocessor; (ii) we introduce a novel synchronization algorithm for RDMA-based transport mechanisms; (iii) we discuss how the ZeroMQ library was extended with support for RDMA-based data transfers through Trans4SCIF (iv) we provide an evaluation showing that Trans4SCIF can lead to significant performance improvements vs. TCP/IP based data transfers for intra-node communication. We note that the developed support is also relevant for the second generation Intel Xeon Phi coprocessor (given that this was released in Q2 of 2017, after the paper was written, here we report results only for the first generation).

The rest of the paper is organized as follows. Section 2 describes the implementation of the Trans4SCIF library. Section 3 discusses the integration of Trans4SCIF with ZeroMQ. Section 4 provides a performance evaluation. Section 5 gives an overview of related work. Finally, Sect. 6 concludes the paper and points to some directions for future work.

2 The Trans4SCIF Library

We give an overview of the Symmetric Communication Interface of the Intel Xeon Phi coprocessor, and describe how Trans4SCIF was implemented on top of it. The code is available for download at *goo.gl/ynrmSL*.

2.1 The Symmetric Communication Interface (SCIF)

SCIF supports intra-node communication over the PCIe bus [8]. For small data transfers, it offers familiar POSIX-like *send()/recv()* operations. For bulk transfers, SCIF offers an RDMA interface that can fully utilize the capabilities of the PCIe bus. While this can lead to much higher throughput, it is harder to use due to the memory management and synchronization issues that must be handled by the programmer, in particular, registering the memory regions to be used for remote reading/writing, and detecting the completion of RDMA transfers. These issues also exist in other RDMA implementations [13].

Besides the classic RDMA *read/write* operations, SCIF offers the *scif_mmap()* function, which maps a pre-registered remote address region into the address space of the calling process. If successful it returns a pointer that can be used to transparently access that memory region of a remote process. This method enables a direct sharing of data structures between processes running on the coprocessor and the host. It also has the lowest communication latency [9], which makes it attractive for inter-process synchronization via shared state.

Data transfers via RDMA occur concurrently to normal program execution. One way to notify the program that the requested data transfers have been performed, is to use the *scif_fence_signal()* function. When called, it internally marks all transfers that have been scheduled so far, and upon their completion writes a given value into a specified local or remote memory location (or both). This is done asynchronously, and the program must check/read that memory location in an explicit way to determine whether the transfers have been completed.

2.2 Trans4SCIF API

To make SCIF-RDMA transport more accessible to application programmers, as well as to pave the way towards exploiting it through the messaging libraries of the O^2 stack, we have developed Trans4SCIF, a user-level library that uses the RDMA mechanism of SCIF and exposes an easy to use socket-like interface.

```
class Socket {
  {uint8_t*, size_t}  getSendBuffer(); // free region of internal send buffer
  size_t send(const uint8_t *data, size_t data_size); // non-blocking
  size_t recv(uint8_t *data, size_t data_size); // non-blocking
  void waitIn(long timeout); // block until there is data to receive
};
```

Fig. 1. Basic API of the Trans4SCIF library.

The basic primitives of the Trans4SCIF API are shown in Fig. 1 in simplified C++ syntax. In a nutshell, *send()* copies the data for transmission to an internal pre-registered buffer and schedules a corresponding RDMA-write operation. If the internal buffer is full, *send()* returns zero, indicating that the application should retry at a later point in time. To avoid data copying, the application can get a handle on the internal transport buffer via *getSendBuffer()*, and write data directly into it. Data reception is done via *recv()*, which immediately returns zero when no data is available. If desired, the application can block until data becomes available by calling *waitIn()*.

2.3 Trans4SCIF Implementation

We now turn to the implementation of the sending and receiving side of Trans4-SCIF, henceforth referred to as *sender* and *receiver*, respectively. Each side maintains its own pre-registered data buffer, the sender for the data that is written by the application, and the receiver for the data that is read by the application. Data copying between the two buffers is performed via RDMA-write. The synchronization between the sender and the receiver is done using two auxiliary data structures, the so-called *Buffer Records Table* (BRT) and *Write Records*

Table (WRT). These are shared between the two sides via *scif_mmap()*. The size of the data buffers can be set by the application at the initialization time; the size of the WRT and other parameters can be set at library compilation time.

The BRT resides in the memory of the sender, and is used to keep track of the free buffer space at the receiver. Each entry contains the starting and ending offset of a region in the remote receiver buffer that is available for writing over RDMA. For example, in Fig. 2, the sender checks the BRT and discovers that there are two regions available for RDMA writes, the first being [0x0..0x400] and the second [0xE00..0x1000]. Given that data chunks are written in the receiver buffer in the spirit of a circular buffer, there can be at most two regions available for write operations, thus the BRT only needs to have two entries.

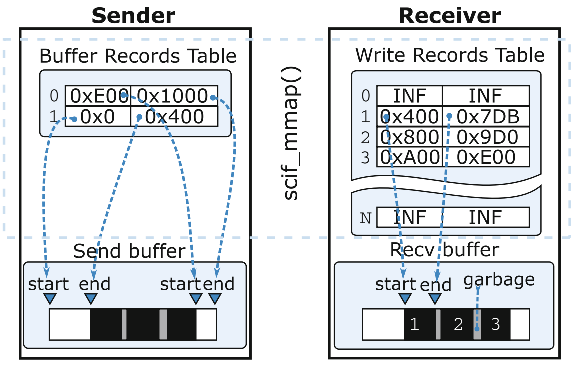


Fig. 2. Snapshot of the registered address spaces of a pair of Trans4SCIF endpoints. The sender's space contains the send buffer and the BRT, while the receiver's space contains the receive buffer and the WRT.

The WRT resides in the memory of the receiver, and is used to keep track of the RDMA writes performed by the sender in the receiver's data buffer. Similarly to a BRT entry, each WRT entry contains the start and end offset of a region in the receiver's data buffer. Taking a look at Fig. 2, the receiver knows that several writes have been performed in its data buffer, the first one in the region [0x400..0x7DB] followed by [0x800..0x9D0] and [0xA00..0xE00]. Note that a separate entry is needed for each individual data transfer. This is because although a write starts at a cache-aligned offset it may end at an arbitrary (non cache-aligned) offset. As a result the receiver's data buffer may have gaps that contain garbage, to be skipped when reading out data. Some WRT entries, like the first and last one in Fig. 2, can be empty (free to use by the sender to denote subsequent RDMA transfers), in which case the start and end fields have an invalid value (infinity). Like the receiver's actual data buffer, the WRT is filled by the sender and consumed by the receiver in the spirit of a circular buffer.

The pseudo-code in Algorithm 1 gives a high-level description of the sender and receiver logic. In a nutshell, the sender checks the BRT to see if there is available space in the receiver's data buffer, in which case it subsequently checks the next WRT entry to see if it is empty, and if so, proceeds with the data transfer and updates the BRT and WRT accordingly. Similarly, the receiver checks the next WRT entry to see if it is filled, in which case it reads out the corresponding region of its data buffer and updates the BRT and WRT. If the next WRT entry is empty the receiver knows no data is available.

Algorithm 1. High-level sender and receiver logic of Trans4SCIF

```
1:  procedure SEND($data, data\_size$)
2:      if $buf\_space = 0$ or $free\_WRT\_slots = 0$ then
3:          return 0
4:      $sz \leftarrow min(buf\_space, data\_size, BUFSIZE/2)$
5:      $sz \leftarrow round\_up(sz)$                        ▷ to cacheline size boundary
6:      $memcpy(send\_buf, data, sz)$                      ▷ destination, source, size
7:      $rdma\_write\_to(recv\_buf, send\_buf, sz)$        ▷ schedule asynchronous RDMA
8:      $scif\_fence\_signal()$             ▷ commission update of WRT upon completion
9:      $scif\_send(token)$                     ▷ send notification (blocking but fast)
10:     $update(BRT)$
11:     return $sz$ + SEND($data + sz, data\_size - sz$)
12:
13: procedure RECV($data, data\_size$)
14:     if $pending\_notifications > 0$ then
15:         $scif\_recv(tokens)$                 ▷ consume notifications (non-blocking)
16:     if buf_fill = 0 then                                       ▷ buffer is empty
17:         return 0
18:     $sz \leftarrow min(buf\_fill, data\_size)$
19:     $memcpy(data, recv\_buf, sz)$                      ▷ destination, source, size
20:     $update(BRT); update(WRT)$
21:     if WRT entry was consumed then
22:         $pending\_notifications++$
23:     return $sz$ + RECV($data + sz, data\_size - sz$)
24:
25: procedure WAITIN($timeout$)
26:     if $pending\_notifications > 0$ then
27:         $scif\_recv(tokens)$                  ▷ consume all notifications (blocking)
28:         $pending\_notifications \leftarrow 0$
29:     $scif\_poll(timeout)(timeout)$              ▷ wait for notification (blocking)
```

The tail recursion in the *send* and *recv* procedures is merely for presentation purposes; in reality, this is implemented using a loop. At the sender, a repetition is performed when wrapping-around the sender's or the receiver's data buffer, in which case two distinct RDMA transfers are scheduled. At the receiver, a repetition is required when wrapping-around the local data buffer, leading to two distinct memory copy operations into the application buffer.

Since the BRT and WRT are directly shared via *scif_mmap()* they are transparently synchronized with the lowest possible latency. Special attention was paid to avoid race conditions, by eliminating concurrent writes on a single field. Importantly, when no data is available, the receiver only accesses a single entry of the (local) WRT. In a similar vein, when the receiver's data buffer is full, the sender only accesses the (local) BRT. Note that in principle it is possible for the receiver's data buffer to have free space and all WRT entries to be filled—in this case the sender cannot proceed with any further data transfers. This can happen if the sending program writes many small messages and the receiving program does not retrieve these messages fast enough. We consider this to be a marginal case given that Trans4SCIF is intended for large data transfers. Also, the application can avoid this by choosing a suitable size for the WRT.

Memory copies, local or remote, are faster when memory addresses are aligned to cacheline boundaries. This is even more crucial for DMA transfers over the PCIe bus [8]. Thus, to achieve good performance, the sender rounds up the amount of data to send to the cacheline boundary and communicates the actual data size to the receiver via the WRT entry. Moreover, the sender bounds the size of each data transfer up to the half of the buffer size. This way it becomes possible to pipeline consecutive data transfers, as the local data copy operation into the sender's buffer (of the next transfer) can be performed in parallel to the RDMA operation into the receiver's buffer (of the previous transfer); note that the asynchronous update of the WRT, when the scheduled transfer completes, is performed by SCIF outside the scope of the Trans4SCIF *send()* operation.

Finally, for each scheduled RDMA transfer the sender sends a notification message to the receiver. This allows the receiving side to block via *scif_poll()* (which in turn invokes the *poll()* system call) instead of busy-waiting until the next WRT entry becomes valid, in case the application wishes to wait until data arrives. Otherwise these notifications do not cause significant overhead since they are small and can be consumed by the receiver in lazy/non-blocking manner.

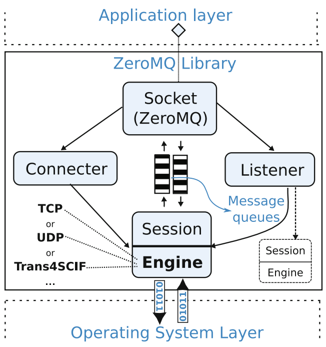
3 Integration of Trans4SCIF with ZeroMQ

ZeroMQ is a versatile and portable messaging technology for building distributed systems. Compared to other technologies, such as MPI, it provides higher-level communication abstractions that can lead to better programmer productivity. Many groups at CERN, including the ALICE collaboration, have chosen ZeroMQ as one of the main communication technologies for performance-critical distributed computing. The popularity of ZeroMQ at CERN as well as elsewhere [16,19] motivated us to extend it with support for SCIF in order to improve performance for programs running on the Intel Xeon Phi coprocessor. In the following, we give an overview of ZeroMQ, and describe how this was extended to support high-bandwidth data transfers via the Trans4SCIF transport.

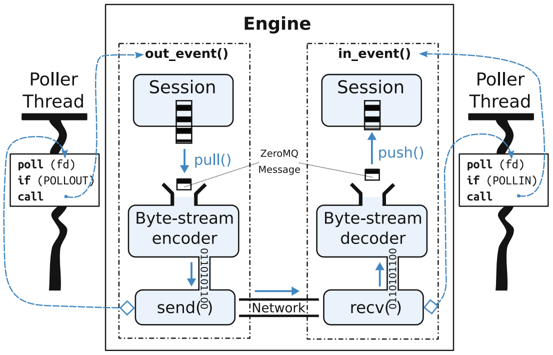
3.1 Technical Overview of the ZeroMQ Messaging Library

The API of ZeroMQ is based on sockets, which can be configured to employ different lower-level transports, such as TCP/IP, UDP/IP and SCTP for communication over the network, Unix domain sockets for local inter-process communication, and shared memory between threads. Also, ZeroMQ sockets can be connected to several peers at the same time in order to form elaborate communication topologies. For instance, one can develop publish-subscribe schemes and processing pipelines with out-of-the-box load balancing and reconnection functionality. Another key feature of ZeroMQ is that it works in a direct peer-to-peer fashion, and does not require an intermediate messaging server/broker.

Once a connection is established between two ZeroMQ sockets, each socket instantiates a *session* object, which is used to keep the state of the connection. In turn, each session object is associated with an *engine* object, which is responsible for sending and receiving data over a lower-level transport service, e.g., TCP/IP or UDP/IP. Figure 3a depicts the relationship between theses objects. The transport engine provides two callback methods for sending and receiving data through the underlying transport. These callbacks are invoked by a so-called *poller* thread, which monitors a file descriptor for input/output readiness events (I/O events). The engine registers this file descriptor with the ZeroMQ runtime environment as part of the initialization procedure. To improve performance, the ZeroMQ runtime may be configured to keep a pool of poller threads, which are shared between the sockets created by the application.



(a) Internal socket structure.



(b) Operation of a transport engine.

Fig. 3. The ZeroMQ architecture.

Figure 3b illustrates the relationship between a poller thread and the engine's file descriptor and callback functions. The poller thread monitors the file descriptor using a suitable POSIX operation, such as *epoll()*, *kqueue()* or *select()*. When a POLLOUT event is raised, indicating that the file descriptor is ready for output, the *out_event()* callback of the engine is invoked. This pulls the next application message from the session's output queue, encodes it into a byte-blob according to the ZMTP protocol [7] of ZeroMQ, and sends it to the other side using the underlying transport service. Similarly, a POLLIN event leads to the invocation of the *in_event()* callback, which retrieves the raw byte-blob from the underlying transport, decodes it into a ZeroMQ message, and pushes it into the session's input queue.

3.2 The Trans4SCIF Engine for ZeroMQ

To enable the usage of SCIF-RDMA through ZeroMQ, we have developed a new ZeroMQ engine that uses Trans4SCIF as the underlying transport service, in the spirit of Fig. 3b. The application can select the Trans4SCIF engine for a ZeroMQ socket simply by prefixing the target address with `scif://` (e.g., instead of the prefix `tcp://` for TCP/IP). The API of the ZeroMQ library is left untouched and can be used in the same way as for all other transports.

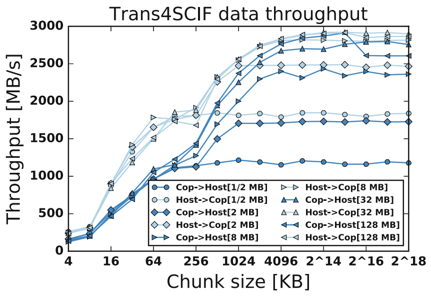
When invoked by the ZeroMQ poller threads, the Trans4SCIF engine performs the data transmission and reception via the *send()* and *recv()* operations of the Trans4SCIF library. Recall that ZeroMQ requires the underlying transport to be accessible through a proper file descriptor that can be monitored through the standard POSIX polling mechanism. Fortunately, the SCIF API provides access to the underlying OS file descriptor that corresponds to a SCIF endpoint (and each Trans4SCIF socket is internally associated with such an endpoint). But note that the I/O readiness of this file descriptor depends on the state of SCIF's internal message buffers, and is not related to the actual RDMA transfers. At the receiver, POLLIN events are properly generated thanks to the arrival of the respective notification tokens that are issued by the sender for each RDMA transfer. This triggers the invocation of the *in_event()* callback which in turn calls the Trans4SCIF *recv()* function to retrieve the data from the local buffer. At the sender, the SCIF file descriptor is always ready for writing, and POLLOUT events lead to the invocation of *out_event()* and the Trans4SCIF *send()* operation, irrespectively of the state of the local data buffer. These invocations are needed to poll the BRT and determine when free space is created so as to proceed with the next transfer.

Finally, the Trans4SCIF engine unregisters the sender's file descriptor from the ZeroMQ polling mechanism when the session output queue has no more application messages. The file descriptor is registered back again when ZeroMQ informs the engine to restart output operation when an application message is added to the output queue. In a similar vein, the receiver's file descriptor is unregistered when the input message queue reaches its capacity, and is registered again as soon as ZeroMQ asks the engine to resume input operation.

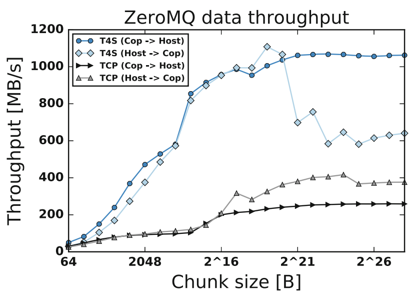
4 Performance Tests

Our experimental testbed consists of two Intel Xeon Phi 7120 coprocessors with 61 cores clocked at 1.23 GHz and 16 GB GDDR memory. The host is a dual socket Intel Xeon E5-2690 server with 64 GB RAM. We run the Intel MPSS v3.8.1 on CentOS Linux kernel 3.10.0-514.2.2.el7.x86_64. For software building we have used the icc compiler v17.0.2 (gcc 6.2 compatibility) with optimizations enabled. Finally, we used ZeroMQ v4.2 and Trans4SCIF v2.4 for the experiments.

Figure 4a shows the results obtained when using the standalone Trans4SCIF library for host-to-coprocessor and coprocessor-to-host transfers. We transfer a total of 1 GB, in chunks ranging from 4 KB up to 256 MB. The benchmark is executed with varying internal buffer sizes, from 0.5 up to 128 MB.



(a) The Trans4SCIF data throughput for different internal buffer sizes.



(b) The ZeroMQ data throughput with the TCP/IP and Trans4SCIF engines.

Fig. 4. Trans4SCIF and ZeroMQ throughput results.

The data points are the (arithmetic) mean values of 100 repetitions for each chunk size. As can be seen, throughput stabilizes at 2.5–3 GB/s for chunk sizes larger than 4 MB. Note that increasing the internal buffer size of Trans4SCIF above 32 MB does not improve performance. In previous work [17] we observed that the maximum throughput that could be achieved with zero-copy RDMA over SCIF was slightly over 6 GB/s on average. Although Trans4SCIF performs memory copies to/from intermediate buffers and needs to synchronize the sender and receiver in order to properly manage buffer occupancy, it still delivers over 40% of raw SCIF performance, which we consider quite acceptable. One also observes that coprocessor-to-host transfers are consistently slower than the ones in the reverse direction. This is attributed to the fact that the RDMA transfers scheduled by the coprocessor are slower than the ones scheduled by the host; this is in line with our previous observations [17].

Figure 4b shows the performance results obtained with ZeroMQ using the Trans4SCIF engine vs. the TCP/IP engine. In the same spirit as above, we transfer again a total of 1 GB, in chunks ranging from 64 B (one cacheline) up to 256 MB, with the internal data buffers of Trans4SCIF and TCP/IP set to 16 MB. It can be seen that ZeroMQ-Trans4SCIF transfers are 2-3x faster than ZeroMQ-TCP (but one has to keep in mind that the former is limited to communication over the PCIe bus whereas the latter can also be used for communication over a network). This is a significant improvement for the ZeroMQ-based applications targeting the Intel Xeon Phi platform. However, the throughput achieved by ZeroMQ-Trans4SCIF is only 50% of that of standalone Trans4SCIF. This heavy drop in performance can be explained considering that encoding and decoding of the data stream to ZeroMQ messages incurs non-trivial computational overhead. Moreover, the receiver makes one additional data copy from the Trans4SCIF buffer into the decoder's internal buffer, which further diminishes the performance (at the sender side, the encoder avoids an extra memory copy by writing directly into the internal Tans4SCIF buffer). We believe that this memory copy is also responsible for the sharp performance drop in host-to-coprocessor transfers with ZeroMQ-Trans4SCIF for chunk sizes larger than 2 MB. When chunk sizes are small, the RDMA transfers are pipelined to a certain extent with the memory copies performed by the decoder. However, as chunk sizes grow, RDMA transfers scale better than the respective memory copies, which in turn eliminates this pipelining effect. The coprocessor-to-host transfers are not severely affected due to the better single-core performance of the host CPU vs. the coprocessor. But even this highly non-optimal host-to-coprocessor throughput of ZeroMQ-Trans4SCIF at roughly 600 MB/s is still $3x$ faster than ZeroMQ-TCP at slightly over 200 MB/s.

We also measured the round-trip-times for the above transports. For chunk sizes up to 64 KB the RTT is stable at about 110 microseconds for Trans4SCIF and 1 millisecond for ZeroMQ-TCP and ZeroMQ-Trans4SCIF. We attribute this order of magnitude difference mainly to two reasons. First, ZeroMQ performs extra encoding/decoding on the application messages, whereas standalone Trans4SCIF leaves application data untouched. Secondly, ZeroMQ blocks for incoming data by waiting to receive an explicit (notification) message from the sending side, whereas standalone Trans4SCIF directly polls the WRT which is updated via the fast *scif_mmap()* method. Still, we do not expect this increased latency to have a notable effect on O^2 computations, which typically push large messages upstream along a uni-directional data-flow pipeline.

5 Related Work

Work on circumventing the limitations of TCP/IP on the Intel Xeon Phi coprocessor by exploiting SCIF-RDMA has also been done in [4,11], in the context of the ROOT software package [2]. However, the approach is more mission-specific, geared towards the parallel composition of output files, and also tightly coupled with the internal architecture of ROOT. In contrast, Trans4SCIF offers a general-purpose stream-based transport abstraction, which is also reused to enhance the performance of ZeroMQ.

Extensive research has been done to optimize MPI for the Intel Xeon Phi coprocessor. For instance, the implementation described in [14,15] employs a zero-copy rendezvous protocol over SCIF to achieve high data throughput for intra-node communication on the coprocessor MPI proxy. While the goal is similar to ours, such support is not easily reusable in the context of ZeroMQ, because the internal design of ZeroMQ does not support integration of RDMA-based transports. In particular, there is no consideration for memory registration and aligned allocations, which is a requirement not only for SCIF but also for numerous other RDMA-enabled interconnects.

An extension for the MVAPICH2 MPI library has been implemented to support transparent data movement between GPUs in a cluster environment with MPI primitives [20]. To hide the overhead of data movement over the PCIe bus, GPU-to-host memory copies are pipelined with node-to-node MPI RDMA transfers. We have adopted a similar approach in Trans4SCIF for the Xeon Phi coprocessor, by pipelining the memory copies to the internal buffer with the RDMA transfers. Also, our data transfer mechanism comes in the form of a standalone library which can be used for socket-oriented host-coprocessor communication. However, MVAPICH2-GPU also enables GPU-to-GPU communication over the network, while Trans4SCIF only works over the PCIe bus.

The work in [10] discusses the performance improvement of UNH-EXS library for data streaming over RDMA. A hybrid data transfer algorithm is presented, which under certain conditions switches to non-zero copy transfers by employing an intermediate circular receive buffer. Once data is copied out from this buffer, the receiver sends notifications back to the sender. Trans4SCIF differs from this approach by eliminating the receiver-to-sender notifications via explicit messaging. Instead, the desired synchronization on the sender side is achieved through shared data structures that are polled locally. However, Trans4SCIF does adopt an explicit notification approach in order to eliminate polling at the receiver side and avoid busywaiting when applications wait for data/messages to arrive.

The Rsockets protocol [6], a successor of the Sockets Direct Protocol (SDP) [3], aims at supporting TCP/IP-like streaming over RDMA by performing remote write operations into pre-exposed data buffers. As these buffers are consumed, new ones become available at the receiving end, for which the sender is notified via control messages. As mentioned, in Trans4SCIF the sender does not need to receive/handle such notifications. To avoid polling at the sender side, Trans4SCIF could be extended following a similar approach. However, the reception of notification messages at the sender would also complicate integration with ZeroMQ significantly.

6 Conclusions

In this paper we have described the design, implementation and performance of the Trans4SCIF library and its integration with the ZeroMQ library. We believe that the synchronization algorithm of Trans4SCIF is generic enough to be used with other RDMA-based transport protocols. Our performance tests show that standalone Trans4SCIF can achieve high data throughput over a second generation PCIe, even with relatively modest internal buffers of a few megabytes. Furthermore, when used through the ZeroMQ messaging library, Trans4SCIF yields a significant improvement over the TCP/IP transport option.

In the future we wish to extend Trans4SCIF to support zero-copy and blocking transfers on both the sender and receiver side, and to exploit these features through ZeroMQ. We will also investigate whether data encoding/decoding can be bypassed in the next versions of ZeroMQ-Trans4SCIF. Last but not least, we plan to port Trans4SCIF on the next generation of the Xeon Phi coprocessor and measure the performance enhancement on actual O^2 workloads.

Acknowledgments. Many thanks for the great support we received from Kristina Gunne, Omar Awile and Luca Atzori from CERN openlab and the CERN IT department.

References

1. ALICE Collaboration: Upgrade of the Online - Offline computing system (CERN-LHCC-2015-004; ALICE-TDR-019)
2. Antcheva, I., et al.: ROOT - A C++ framework for petabyte data storage, statistical analysis and visualization. Comput. Phys. Commun. **180**(12), 2499–2512 (2009)
3. Balaji, P., et al.: Sockets Direct Protocol over InfiniBand in clusters: is it beneficial? In: IEEE International Symposium on Performance Analysis of Systems and Software, pp. 28–35, IEEE (2004)
4. Farrell, S., Dotti, A., Asai, M., Calafiura, P., Monnard, R.: Multi-threaded Geant4 on the Xeon-Phi with complex high-energy physics geometry. In: IEEE Nuclear Science Symposium and Medical Imaging Conference, pp. 1–4 (2015)
5. George, C.: Intel Xeon Phi Coprocessor, the architecture. Intel Whitepaper (2014)
6. Hefty, S.: Rsocket, https://goo.gl/2uOsmZ
7. Hintjens, P.: ZeroMQ: Messaging for Many Applications. O'Reilly, Sebastopol (2013)
8. Intel Corporation: Symmetric Communications Interface (SCIF) For Intel Xeon Phi Product Family Users Guide , revision: 3.5 (2015)
9. Linux. https://www.kernel.org/doc/Documentation/mic/mic_overview.txt
10. MacArthur, P., Russell, R.D.: An efficient method for stream semantics over RDMA. In: IEEE International Parallel and Distributed Processing Symposium, pp. 841–851 (2014)
11. Monnard, R.: Concurrent I/O from Xeon Phi accelerator cards. Masters thesis, Haute Ecole Specialisee de Suisse Occidentale de Fribourg, Switzerland (2015)
12. Nowak, A., et al.: Does the Intel Xeon Phi processor fit HEP workloads?. J. Phys. Conf. Seri. 513(5) (2014). article no. 052024
13. Pfister, G.F.: An introduction to the infiniband architecture. High Perfor. Mass Storage and Parallel I/O **42**, 617–632 (2001)
14. Potluri, S., Hamidouche, K., Bureddy, D., Panda, D.K.: MVAPICH2-MIC: A high performance MPI library for Xeon Phi clusters with Infiniband. In: Extreme Scaling, Workshop, pp. 25–32 (2013)
15. Potluri, S., Venkatesh, A., Bureddy, D., Kandalla, K., Panda, D.K.: Efficient intra-node communication on Intel-MIC clusters. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 128–135 (2013)
16. Radford, N.A., et al.: Valkyrie: NASA's first bipedal humanoid robot. J. Field Robot. **32**(3), 397–419 (2015)
17. Santogidis, A., Hirstius, A., Lalis, S.: Evaluating the transport layer of the ALFA framework for the Intel Xeon Phi Coprocessor. J. Phys. Conf. Ser. 664(9) (2015). article no. 092021
18. Sustrik, M.: NanoMSG. http://nanomsg.org/
19. Toshniwal, A., et al.: Storm@ twitter. In: ACM SIGMOD International Conference on Management of Data, pp. 147–156 (2014)
20. Wang, H., et al.: MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. In: Comput. Sci. Res. Dev. 26(3–4), p. 257 (2011)