Evaluating the transport layer of the ALFA framework for the Intel® Xeon Phi™ Coprocessor

2015 J. Phys.: Conf. Ser. 664 092021

(http://iopscience.iop.org/1742-6596/664/9/092021)

Journal of Physics: Conference Series **664** (2015) 092021 doi:10.1088/1742-6596/664/9/092021

Evaluating the transport layer of the ALFA framework for the Intel® Xeon Phi™ Coprocessor

Aram Santogidis[1, 2], Andreas Hirstius[3], Spyros Lalis[4]

[1]CERN IT/Openlab, 1211 Geneva, Switzerland
[2]Maynooth University, Maynooth, Co. Kildare, Ireland
[3]Intel GmbH, 85622 Feldkirchen/Munich, Germany
[4]University of Thessaly, 38221 Volos, Greece

E-mail: aram.santogidis@cern.ch, andreas.hirstius@intel.com, lalis@inf.uth.gr

Abstract. The ALFA framework supports the software development of major High Energy Physics experiments. As part of our research effort to optimize the transport layer of ALFA, we focus on profiling its data transfer performance for inter-node communication on the Intel Xeon Phi Coprocessor. In this article we present the collected performance measurements with the related analysis of the results. The optimization opportunities that are discovered, help us to formulate the future plans of enabling high performance data transfer for ALFA on the Intel Xeon Phi architecture.

1. Introduction

ALFA is the concurrency framework supporting the development of the data processing and event reconstruction software for ALICE and FAIR high energy physics experiments [1, 2]. One of the main facilities it provides is the data transport layer, the FairMQ message queue library which is used to connect and coordinate data processing components. By taking advantage of the abstractions provided by FairMQ one can establish a data processing topology spanning over a computing cluster potentially featuring heterogeneous computing hardware.

Distributed applications, such as the O^2 software of ALICE [1], consist of hundreds of loosely coupled processes. One can take advantage of this design by porting compute-intensive (OS) processes on manycore processors, such as the Intel Xeon Phi Coprocessor, without modifying much the overall system.

The motivation of porting such processes to the Intel Xeon Phi Coprocessor is to increase the execution efficiency of software components that can take advantage of the computational capabilities of this particular architecture (high core count, wide vector engines, high memory bandwidth, etc.). The High Energy Physics (HEP) applications are usually throughput driven [3]. Therefore the data transport mechanisms supporting this kind of applications must cope with high data transfer rates in order to keep the computing nodes busy. As part of our effort to optimize the transport layer of ALFA for the Intel Xeon Phi platform, we evaluate its current performance profile and search for optimization opportunities.

2. Background

The O^2 software is a collection of multi-process distributed applications that implement the online and offline processing for the ALICE experiment at CERN [1]. The software is responsible for a number of computing tasks such as data acquisition from the detectors, event reconstruction, data compression and physics analysis. It will run on the next generation computing farm of the experiment, featuring high-end heterogeneous commercial-off-the-shelf (COTS) hardware. The O^2 software implementation is based on the ALFA and FairRoot frameworks (Figure 1). It is designed as a data processing pipeline, as opposed to big, tightly coupled, parallel processes. The data transport technology, interconnecting the processes of the pipeline, is the FairMQ message queue library which is part of the ALFA framework.

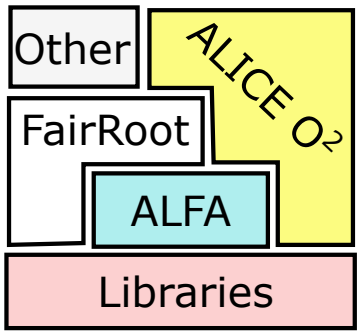

Figure 1. Overview of the external dependencies of the O^2 software.

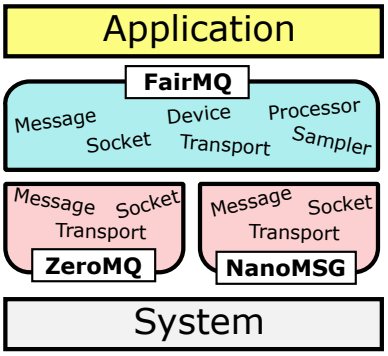

Figure 2. The FairMQ software architecture overview.

2.1. FairMQ on ZeroMQ and NanoMSG

FairMQ, the transport mechanism of ALFA, is a message passing technology that enables asynchronous communication between processes running on a distributed system. The main abstraction it provides is the *device* which represents a process that either generates data (i.e. a source), does message-based processing (e.g. merging, compressing messages, etc.) or content-based processing (data consumption). The user of FairMQ defines a set of devices and connects them to form a processing topology that implements the application logic.

FairMQ is implemented as a thin layer of software over ZeroMQ [4] and NanoMSG [5] message-queue transport libraries (Figure 2). These libraries implement the actual data transfer mechanisms therefore the communication performance of FairMQ is defined by the performance of these libraries since the FairMQ interface introduces marginal overhead to the overall data transfer performance [1]. ZeroMQ and NanoMSG are *message queue* technologies that provide functionality for inter-thread, inter-process and inter-node communication. The message queues are designed without the requirement of a dedicated message broker and for this reason they can support highly scalable concurrent distributed applications. With a socket-like API along with built-in abstractions for common communication patterns such as publish-subscribe, pipeline, request-reply, they can support fast development without compromising stability and performance.

2.2. The Intel Xeon Phi Coprocessor and the SCIF mechanism

In 2012 Intel released the Intel Xeon Phi Coprocessor, codenamed “Knights Corner”, which is a manycore processor with enhanced parallel computing capabilities. It can be attached to

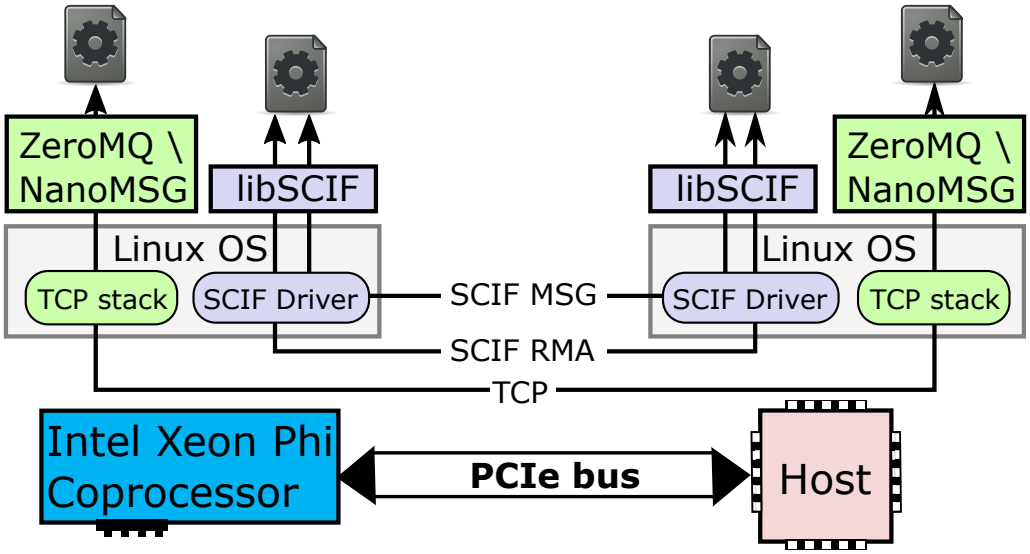

Figure 3. Program runs on the Coprocessor and on the Host system. They communicate over the PCIe bus with the TCP and SCIF transport protocols.

a PCIe[1] slot of a computing node and boost the performance of parallel applications running on that node. It features up to 61 cores with 4 hardware threads per core. Each core has a 512-bit wide vector engine that can speed up applications with data parallelism. Although it is a PCIe peripheral device, it runs its own embedded Linux-based operating system accompanied with a utility software package, the Intel Manycore Platform Software Stack (MPSS). For this reason, it can also be used as a separate computing node. In our research effort we investigate the potential of porting complete OS processes to the Coprocessor and increase their execution efficiency. For this reason we profile the performance of the communication of the Coprocessor with the Host system over the PCIe interconnect.

The Symmetric Communications InterFace (SCIF), included in the MPSS package, abstracts the details of the communication over the PCI express bus [6]. It is used as a socket-like inter-node communication mechanism where nodes can be Intel Xeon Phi Coprocessors and Intel Xeon host processors. Programs using SCIF can communicate by passing messages, typically used for small data payloads, and use Remote (Direct) Memory Access (RMA) operations for bulk data transfers. RMA has also the advantage of supporting one-sided communication, which is useful for algorithms that need to avoid explicit synchronization to increase the execution performance. Figure 3 shows the interactions between programs running on the Coprocessor and on the Host. Programs that run on top of the message queue libraries use the TCP protocol, whereas programs that use the SCIF API use SCIF messaging and SCIF Remote Memory Access (RMA) operations. It is worth noting that SCIF RMA can utilize the DMA engine whereas the TCP implementation for the Intel Xeon Phi Coprocessor does not support DMA transfers.

3. Performance evaluation method

We designed an indicative execution scenario in order to compare the performance of ZeroMQ and NanoMSG against the native SCIF data transport mechanism of the Intel Xeon Phi Coprocessor. We describe the hardware and software setup details as well as the techniques we used to collect and present the performance measurements. We performed the performance evaluation tests on a testbed provided by Intel (see Table 1).

[1] PCIe Gen2 16x lanes

Component	Technical details
CPU	2x E5-2680 "Sandy Bridge EP" | 3.1 GHz | 8 cores
RAM	8x DDR3 4GB DIMMs | 1333 MHz Speed
Operating System	RHEL server 6.6 | Kernel 2.6.32-504.el6.x86_64
Coprocessors	2x Xeon Phi Coprocessor SE10/7120 series | 1.1 GHz | 8 GB GDDR
MPSS version	version 3.5 | Kernel 2.6.38.8+mpss3.5
Bios version	SE5C600.86B.02.04.0003.102320141138
ZeroMQ	Version 4.0.5
NanoMSG	Last commit: d530bfe3ad64e9cf0e5d5f5c97549f5744190b5e
SCIF-perf-bench	Version 1.0

Table 1. The software and hardware technical details of our testbed.

3.1. Porting ZeroMQ and NanoMSG to the Intel Xeon Phi platform

The cross-compilation of the libraries for the Intel Xeon Phi architecture was a matter of providing the required building environment parameters [7]. After that we were able to run ZeroMQ and NanoMSG programs on the Coprocessor that communicate with programs running on the Host system. Essentially this is inter-node communication so, both for ZeroMQ and NanoMSG, the only option we had was to use the TCP over the PCIe bus. We increased the maximum TCP buffer sizes to 128 MB on both, the Coprocessor and the Host, systems[2] without experiencing noticeable improvement. The TCP implementation for the Intel Xeon Phi Coprocessor does not use DMA transfers so the data transfer performance is limited by the single core speed of the device.

3.2. Performance testing programs

Both ZeroMQ and NanoMSG are shipped with message throughput test programs implemented by the developers maintaining these projects. For the evaluation of the messaging libraries we used their respective performance tests. For the evaluation of SCIF, we developed a set of micro-benchmarks, the SCIF-perf-bench [8]. The design and the usage of these benchmarks is the same with the ones shipped with ZeroMQ and NanoMSG.

These benchmarks consist of two processes, one sender and one receiver. The sender initially sends a signaling-message to the receiver, indicating the start of the data transfer, and then transfers the payload in chunks, by sending a number of fixed-size messages. The receiver records the time that elapses between the arrival of the signaling-message and the receipt of the last message (payload chunk), and calculates the average transfer rate based on the following formula:

$$average\_transfer\_rate = \frac{msg\_size \times num\_of\_msgs}{transfer\_time}$$

So for example, to measure the transfer rate from the Coprocessor to the Host, we execute the sender program on the former and the receiver on the latter, and record the output of the receiver.

3.3. Selection of statistical measures

We would like to be able to determine representative values from a set of performance measurements we conduct. In order to choose a suitable measure of central tendency, we have to check the distribution of the samples that we get when we run the performance test programs.

[2] We also configured the high-water mark of ZeroMQ to infinite. This option defines the maximum number of outstanding messages in its outbound and inbound queues.

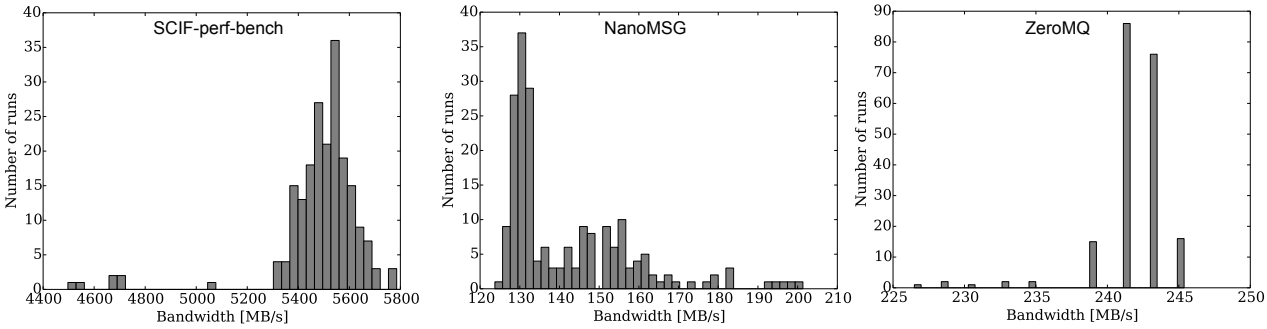

Figure 4. Distributions of the data transfer rate for each technology. The data transfer direction is from the Coprocessor to the Host, transferring 1 GB payload in 2 MB chunks. The distributions with other chunk sizes had similar shape.

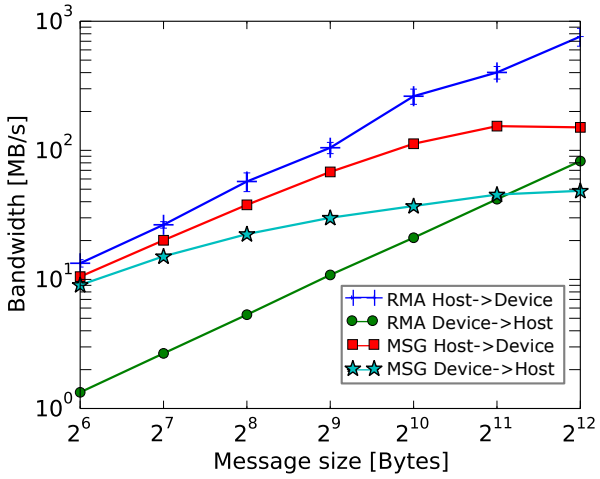

Figure 5. SCIF performance for small data sizes: Using the remote memory access (RMA) and messaging (MSG) mechanism, to transfer 128 MB while varying the chunk size from 64 to 4096 Bytes.

We present the distributions of 201 samples[3] collected for each one of the solutions (Figure 4). We notice that the distributions have significant number of outliers therefore we choose *median* as measure of central tendency. As for the measure of variation we use the Median Absolute Deviation (MAD).

3.4. Choice between SCIF messaging and RMA for small messages

For SCIF, we can use either the messaging interface or RMA operations. For bulk data transfers (e.g. over 4 KB), we use RMA since that is the use case that it was designed for. For small payloads, we executed a performance comparison test in order to determine the preferred solution. The Figure 5 shows the performance of RMA[4] against message passing[5].

We notice that the RMA transfer from the Coprocessor to the Host is an order of magnitude slower than the RMA transfer from the Host to the Coprocessor and generally slower than the message transfers. It is preferable to have consistent performance, regardless if the transfer is

[3] We used an odd number of samples because it is convenient to calculate the median.
[4] We use the scif_vwriteto() function of the SCIF API.
[5] Invoked without the BLOCK flag, which yields slightly better performance.

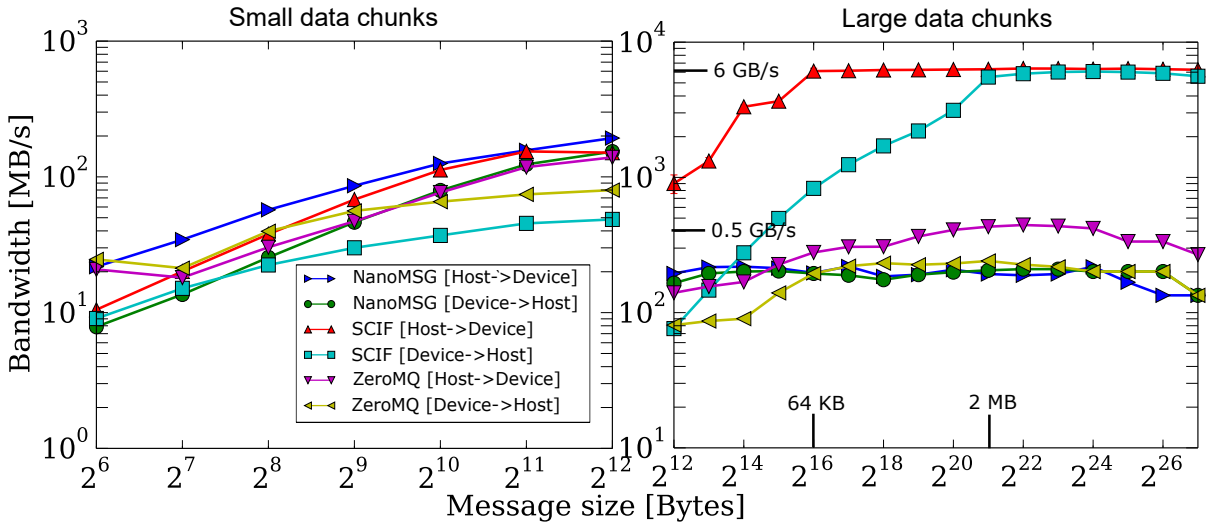

Figure 6. Plots illustrating the performance of SCIF, ZeroMQ and NanoMSG, for small and large payloads. The left plot corresponds to the results collected for the transfer of 128 MB in chunks from 64 B to 4 KB. The right plot corresponds to large data payloads, where 1 GB is transferred in chunks from 4 KB to 128 MB. For the large payloads scenario, SCIF uses the DMA engine.

initiated from the Coprocessor or the Host. Therefore we use SCIF messages for small data payloads in the performance tests that follow.

3.5. Caveats and solutions

3.5.1. NanoMSG last message problem: During our tests we encountered a problem with NanoMSG's throughput test implementation. For large message sizes, sometimes the last message would be dropped presumably because of premature exit from the sender program. We inspected the implementation and introduced the workaround of adding a fixed delay before the sender closes the socket. A better solution would be to synchronize the two processes before closing the sockets.

3.5.2. SCIF RMA Coprocessor to Host performance drop: Initially when we tried to assess the performance of SCIF we noticed some unexpected drops in performance for RMA transfers from the Coprocessor to the Host. This effect was also encountered by Drs. Jan Just Keijser who shared with us that by upgrading the firmware of the host server he solved the problem. We confirmed his solution by upgrading the firmware of our machine and executing the benchmarks anew.

4. Data transfer performance comparison of ZeroMQ, NanoMSG and SCIF

For the comparison of the data transfer performance of the libraries, we collected measurements for small and large payloads according to the methodology presented in Section 3. Figure 6

summarizes the performance comparison results. The left plot refers to small data sizes, where a data payload of 128 MB is transferred in chunks from 64 to 4096 Bytes. The processes running on the Coprocessor and the Host are terminated and restarted for each test. For each chunk size the test is repeated 201 times and the median values are chosen as the representative values for each case (see Sec. 3.3). For the large data sizes the test is the same as for small data sizes, except that we transfer 1 GB of payload, in chunks starting from 4 KB up to 128 MB. We remind that for the small data sizes with SCIF we use the messaging mechanism whereas RMA with DMA transfers for large data sizes.

4.1. Small data payloads

For the small data sizes the performance profile is similar for all options. We notice that in the *Coprocessor to Host* transfer case, and especially the SCIF version, the rate is lower than the *Host to Coprocessor* case. This result can be explained by the fact that the single core performance of the Intel Xeon Phi Coprocessor is lower than that of the Host processor. Therefore the *sender* program running on the Coprocessor is slower than when it runs on the Host. Note that there are no DMA transfers involved in the small data payloads scenario.

One other noticeable result is the fact that the performance of SCIF is less or equal than that of ZeroMQ and NanoMSG. This may be because TCP does message coalescing therefore achieves aggregate transfers. On the other hand the messaging implementation in SCIF-perf-bench directly uses the SCIF API and sends each message separately without aggregating.

4.2. Large data payloads

For the large data payloads, SCIF eventually outperforms ZeroMQ and NanoMSG by more than an order of magnitude. For chunk sizes larger than 64 KB for the Host to Coprocessor case and 2 MB for the Coprocessor to Host case, the transfer bandwidth is already saturated. The fact that the bandwidth saturates for different chunk sizes can be attributed to the single core performance difference between the Coprocessor and the Host.

The primary reason for performance difference between SCIF and the two libraries is because SCIF uses the DMA engine to achieve high bandwidth utilization of the PCIe bus whereas the message queue libraries are limited to the TCP protocol option. The TCP implementation of Intel Xeon Phi Coprocessor is a single-threaded process limited by the poor single-core performance of the Coprocessor. One other important aspect is that TCP is designed for communication over unreliable networks by employing techniques such as delivery acknowledgments, flow control and congestion avoidance. The PCIe bus is a reliable communication medium, therefore the features of TCP introduce unnecessary overhead which further diminishes the data transfer performance.

One can notice that ZeroMQ and NanoMSG show almost constant performance over the increasing data chunk size. Although we increased the TCP maximum buffer size to 128 MB, both on the Coprocessor and the Host systems, the performance did not improve. This fact leads us to the conclusion that the bottleneck occurs at the TCP stack implementation level, and not at the application and system configuration level.

We noticed that, especially from *Host to Coprocessor* transfers, ZeroMQ performs better than NanoMSG. The reasons for this result are unknown but probably this phenomenon is related to the maturity level of the ZeroMQ project compared to the much younger NanoMSG. Evidence, that currently NanoMSG is slower than ZeroMQ, are examined by the project community[6].

[6] https://github.com/nanomsg/nanomsg/issues/300

5. Summary and future plans

In this article we investigated the data transfer performance of the ZeroMQ and NanoMSG libraries of the ALFA framework on top of the Intel Xeon Phi Coprocessor, in comparison with the performance of the SCIF data transfer mechanisms. By collecting a set of performance measurements we showed that for large message sizes (over 4 KB), the SCIF data transport mechanism of Intel Xeon Phi Coprocessor outperforms by over an order of magnitude the ZeroMQ and NanoMSG messaging libraries. We also take into account that the event sizes of O^2 of ALICE is around 24 MB, and for this size of payloads SCIF RMA operates at its maximum bandwidth. This finding motivates us to extend the above messaging libraries with support for the SCIF transport protocol. Such an extension will transparently boost the data transfer performance of processes running on the Coprocessor.

As a next step we plan to design and implement a transport protocol on top of SCIF that will be integrated in the messaging libraries. We will also investigate the possibility of implementing our protocol over other RMA enabled transport protocols such as the Infiniband Verbs protocol. We will also do preliminary investigation in order to prepare our solution for the next generation of Intel Xeon Phi, "Knights Landing", featuring the Intel Omni-Path fabric.

Acknowledgments

Special thanks to Dr. Piotr Umiski from Intel Poland, who provided additional insight in order to explain the performance results for SCIF. Also many thanks to Drs. Jan Just Keijser from the Nikhef institute in Amsterdam, who suggested to upgrade the firmware in order to overcome some performance instabilities we encountered.

This research project has been supported by a Marie Curie Early European Industrial Doctorates Fellowship of the European Communitys Seventh Framework Programme under contract number (PITN-GA-2012-316596-ICE-DIP).

References

[1] ALICE Collaboration, Upgrade of the Online - Offline computing system. CERN-LHCC-2015-004; ALICE-TDR-019.
[2] M. Al-Turany, ALFA: a common concurrency framework for ALICE and FAIR experiments, April 2014, `https://goo.gl/BYlNsn`
[3] John Apostolakis, Ren Brun, Federico Carminati, Andrei Gheata and Sandro Wenzel, The path toward HEP High Performance Computing, J. Phys.: Conf. Ser. 513 052006, 2014
[4] P. Hintjens, `http://zguide.zeromq.org/page:all`
[5] M. Sustrik, `http://nanomsg.org/index.html`
[6] Intel corporation, Symmetric Communications Interface (SCIF) For Intel Xeon Phi Product Family Users Guide, `http://goo.gl/y3L13U`, April 2015
[7] Intel corporation, Autotools and Intel Xeon Phi Coprocessor, `https://goo.gl/Es7TJJ`, December 2013
[8] A. Santogidis, The SCIF-perf-bench micro-benchmark source code, Zenodo.org, http://dx.doi.org/10.5281/zenodo.17665, May 2015