SEVENTH FRAMEWORK PROGRAMME
Research Infrastructures

FP7-ICT-2011-7

DEEP

Dynamical Exascale Entry Platform

Grant Agreement Number: 287530

D4.3
ParaStation component pscom supporting EXTOLL

Approved

Version:	2.0
Author(s):	J.Hauke (ParTec)
Date:	24.02.2014

Project and Deliverable Information Sheet

DEEP Project	**Project Ref. №:** 287530	
	Project Title: Dynamical Exascale Entry Platform	
	Project Web Site: http://www.deep-project.eu	
	Deliverable ID: D4.3	
	Deliverable Nature: Report / Other	
	Deliverable Level: PU *	**Contractual Date of Delivery:** 31 / May / 2013
		Actual Date of Delivery: 31 / May / 2013
	EC Project Officer: Luis Carlos Busquets Pérez	

* - The dissemination level are indicated as follows: **PU** – Public, **PP** – Restricted to other participants (including the Commission Services), **RE** – Restricted to a group specified by the consortium (including the Commission Services). **CO** – Confidential, only for members of the consortium (including the Commission Services).

Document Control Sheet

Document	**Title: ParaStation component pscom supporting EXTOLL**	
	ID: D4.3	
	Version: 2.0	**Status:** Approved
	Available at: http://www.deep-project.eu	
	Software Tool: Microsoft Word	
	File(s): DEEP_D4.3_ParaStation_component_pscom_supporting_EXTOLL_v2.0-ECapproved.docx	
Authorship	**Written by:**	J. Hauke, ParTec
	Contributors:	N. Eicker, JUELICH T. Moschny, ParTec
	Reviewed by:	H.-C. Hoppe, Intel W. Gürich, JUELICH
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	31/May/2013	Final version	Submitted to EC
2.0	24/February/2014	Approved	Approved by EC

Document Keywords

Keywords:	DEEP, HPC, Exascale, ParaStation, pscom, EXTOLL

Table of Contents

List of Figures

List of Tables

Executive Summary

The DEEP programming model employs MPI in a twofold way. On the one hand the MPI communication layer is used on both sides of the DEEP System, the Cluster and the Booster. On the other hand, off-loading of highly-scalable code-parts from the Cluster to the Booster is implemented by means of *MPI_Comm_spawn()* to spawn processes and create communication channels between the Cluster and the Booster.

Both mechanisms require an efficient utilization of the EXTOLL Booster network by ParaStation MPI, ParTec's implementation of the MPI-Standard chosen by the DEEP project.

This document describes the design and the implementation of the ParaStation communication library pscom, the network abstraction layer of pscom and pscom's plugin API for extending the list of supported network architectures. Furthermore it explains the internals of both EXTOLL-plugins of the pscom library which provide efficient communication paths for communicating between Booster nodes.

This report is targeted at developers of high-level communication functionality in the context of the DEEP project. Members of the work package WP4, WP5 and application developers using the global MPI shall take this document to get a better understanding of the various different communication paths which exist in the DEEP architecture. This document highlights the communication path between the Booster Nodes (BN) and explains the internals of both EXTOLL-plugins for the pscom library.

1 Development Environment “The BIC Evaluator”

For development and testing purposes in the context of the DEEP project the BIC evaluator was built (Figure 1). This system consists of three standard x86 servers representing a Cluster Node (CN), a Booster Node (BN) and a Booster Interface (BI). The CN is connected to the BI with InfiniBand, the BN is connected to the BI with the Xilinx FPGA implementation of EXTOLL. The BI is a bridge between both networks. Although the BN in the BIC evaluator is not equipped with a many-core processor based on MIC technology, it is expected that the software stack developed and tested on the BIC evaluator can be easily ported to the final architecture due to the similarities of the MIC many-core architecture to the standard Xeon multi-core processors.

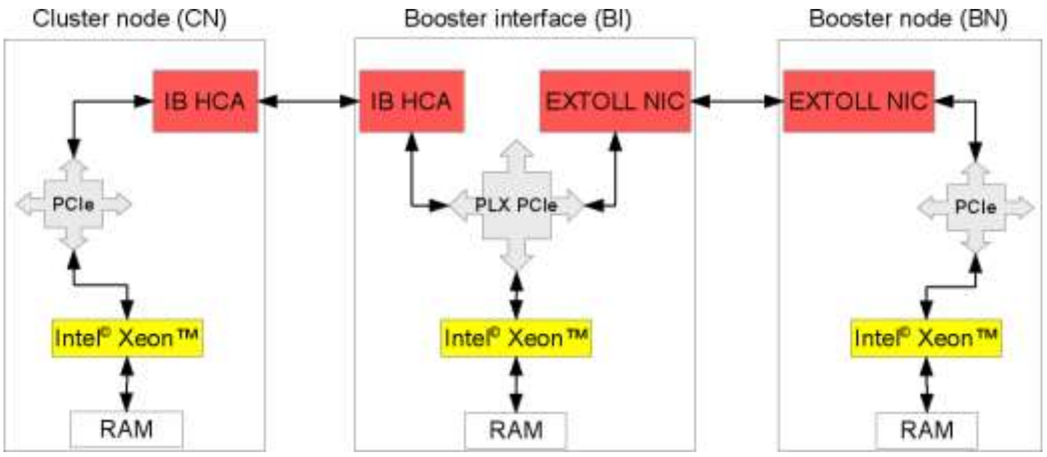

Figure 1: Architecture of the BIC evaluator system

For a hardware description of the BIC Evaluator refer to D4.2 “Cluster Booster protocol” [5].

For the purpose of developing the EXTOLL-plugins of the pscom library we are using the two nodes of the BIC-evaluator equipped with FPGA-based EXTOLL network adapters, namely the "BI" and the "BN". Because both nodes provide EXTOLL access, they could also be used as a development system for BN-to-BN communication. For this task the InfiniBand HCA of the BI is unused.

2 Software Design

2.1 ParaStation MPI

ParaStation MPI [2] provides a standard interface for parallel applications that require MPI-1 or MPI-2 library functionality. ParaStation MPI selects the most appropriate of all available interconnects at runtime. For example, for intra-node communication, ParaStation MPI will select a shared-memory model. Inter-node communication is done using the native networking libraries of the highest performing installed interconnect. MPI applications are started by an mpiexec implementation of ParaStation MPI (Figure 2). It contacts a local process management daemon (psid) to spawn processes on remote nodes. The process management daemons on every node are connected to each other by a highly scalable reliable datagram protocol (RDP) on top of UDP/IP. Spawned processes find each other using the process management interface (PMI) also provided by the psid. For each process, the PMI layer distributes its TCP/IP address to every other process. Using this information, a TCP connection will be established between all processes (precon) which will then be used for a handshake to exchange protocol-dependent data of the pscom plugin in use for this connection.

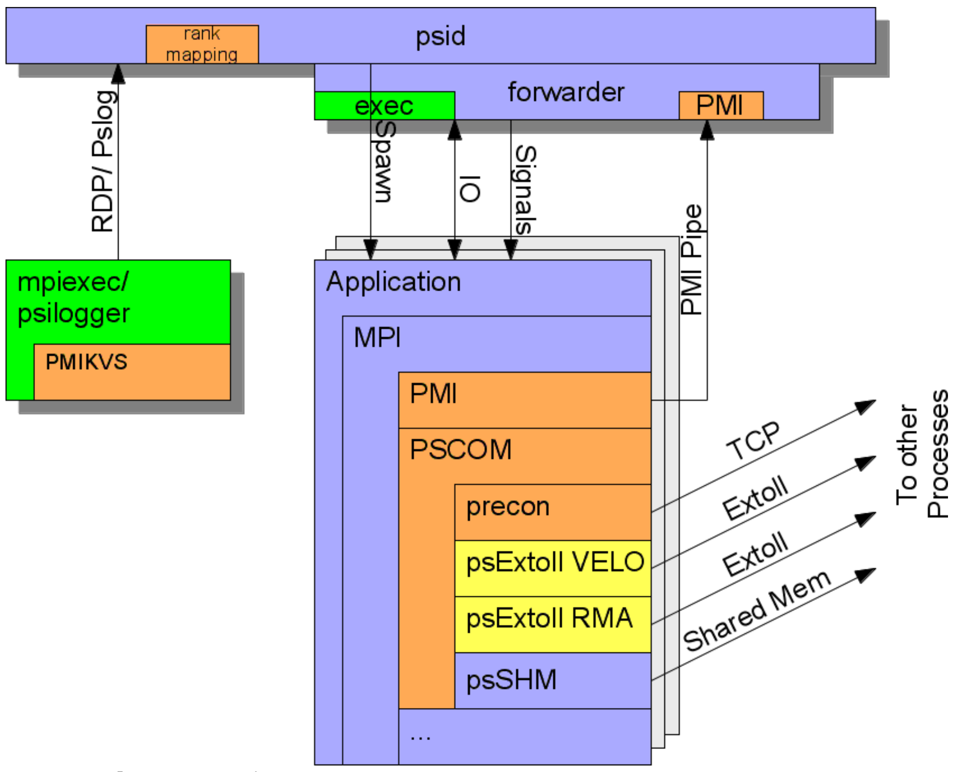

Figure 2: Booster Node: ParaStation MPI

2.2 ParaStation pscom library

ParaStation's pscom library implements efficient inter-process communication to be utilized by ParaStation MPI. It supports various network interconnects for inter-node communication and shared memory for intra-node communication at the same time. The library is extensible through plugins to support more interconnects or protocols. Internally all communication is connection-based. The actual communication path to use is automatically chosen at runtime during the establishment of the connection. This happens independently for each connection with an adjustable priority system to identify the best available path.

Initially designed as a network abstraction layer for the ParaStation MPI implementation, the pscom library can also be used stand-alone. It provides synchronous and asynchronous point-to-point message delivery with send/receive semantics, single sided RMA with put/get and collective operations on groups of connections.

ParaStation MPI uses the pscom library and can therefore be used on all nodes connected through interconnects which are supported by the library. In the case of BN-to-BN communication users can choose between the EXTOLL-plugins described in section 2.3.

In the context of DEEP the pscom library will also be used by the Cluster-Booster protocol (CBP) as an efficient transport layer for the meta data of the protocol. This is described in D4.1 [4] and D4.2 [5] in detail. It implements the control channels between BN and BI, and between BI and CN that are used for acknowledgement delivery, remote calls, etc. In the case of the CN-to-BI control channel pscom uses the InfiniBand verbs layer through the InfiniBand plugin not covered in this document. Between BI and BN it uses EXTOLL's small message/ low latency VELO engine provided by the EXTOLL VELO plugin. The pscom library itself provides the central message processing engine of ParaStation MPI for all plugins. Pscom plugins implement efficient communication channels on top of different low-level network protocols concurrently usable at the same time. Therefore, it fits very well to the heterogeneous network design of the DEEP architecture. Messages of any size are fragmented by pscom into chunks fitting the maximal transfer unit (MTU) of the low-level communication stack of a given connection. An in-order delivery of these chunks is guaranteed by pscom plugins using in turn the respective low level API. As a middleware layer the pscom library together with the plugins connects higher- and lower-level layers of the communication stack.

2.3 PSCOM EXTOLL plugins

The EXTOLL interconnect technology has been specifically developed for High Performance Computing (HPC).

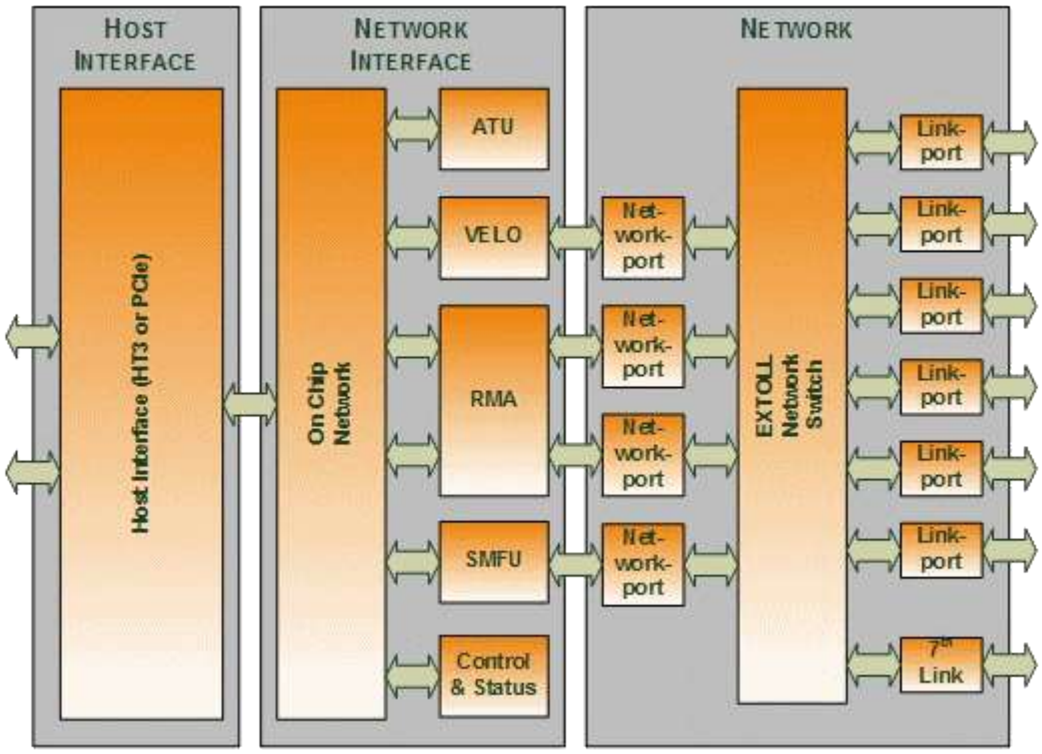

Figure 3: Extoll architecture

EXTOLL implements a lean and optimized network interface controller (NIC). It minimizes the state information on the NIC and provides user-level, virtualized access to the NIC without the need to go through the OS kernel on the node. It enables CPU-offloading, zero-copy and true one-sided protocols. To support these, a Control & Status register file and an Address Translation Unit (ATU) for translation between virtual and physical addresses are included. Each EXTOLL NIC includes its own switch with 6 external links, enabling a 3-D torus topology. In-order message delivery is reliable with automatic retransmissions handled by the hardware.

Currently two primary communication engines are implemented by EXTOLL [3]:

- Remote Memory Access (RMA) realizes direct access to remote memory using put and get operations. It enables CPU-offloading, zero-copy and true one-sided protocols. After triggering a RMA operation by the host CPU the EXTOLL NICs use their own DMA engines to do the data transfer.
- Virtualized Engine for Low Overhead (VELO) implements extreme low latency send/receive style communication. It supports messages of up to 64 byte on the hardware level.

Both engines are accessible for both, direct user-level applications or higher-level libraries by the API libraries libVELO and libRMA. They provide user-level, virtualized access to the hardware; on the critical path operations are initiated bypassing the OS kernel in a secure way. To support both APIs we have implemented two separate pscom plugins.

2.4 Description of the EXTOLL RMA plugin

EXTOLL provides the RMA interface to transfer arbitrary data from a registered memory region on one node to a registered memory region of a remote node. The registration is done by the pscom plugin with a call to *rma2_register()* giving a handle to the memory-region. Data transfers itself are single-sided operations acting on the handles on both sides. Both, "put" and "get" operations are possible. The current implementation of the EXTOLL plugin does not employ the "get" operation and utilizes only the "put" mechanism as realized by the API call *rma2_post_put_bt()*.

Registering a memory region is an expensive operation. Therefore, send and receive buffers are allocated and registered only once, at connection initialization time. Each connection uses its private set of send and receive buffers.

The main pscom layers break up higher level messages into chunks of data to be transferred by the plugin. The only requirement on the plugin is the delivery of these chunks to the pscom layer on the remote site in the same order as the chunks on the local site.

The EXTOLL plugin transmits the chunks as follows:

Operations on the sending side

1. Choose the next unused send buffer. This is done in a round robin fashion.
2. Copy the data-chunk into the chosen buffer
3. Post a "RMA put" request to EXTOLL which actually transfers the chunk from the local send buffer to the remote receive buffer with the same index as the send buffer.
4. Notify pscom about the progress

Actions on the receiving side

5. Notify pscom about the new chunk located in the receive buffer
6. The main pscom layer will copy the received data to its final destination
7. Acknowledge the receive

In 1) an unused send buffer has to be identified. Eventually, all send buffers are busy. This might be the case if there is still an active RMA put request from a previous operation, or the send buffer is not yet acknowledged by the receiver for reuse (see step 7, described below). If there is no unused buffer left, the plugin reports this busy state to the pscom layer and does not continue with step 2. The progress engine of the pscom layer will later retry to send this chunk until success or a fatal error aborts the connection. In this case, the chunk-data stays in the original message buffer.

In 2) the data-chunk itself is copied to the send buffer. At this stage the send buffer is marked as "used". Right behind the data-chunk the plugin appends a small tail containing protocol data. This includes the size of this chunk, piggybacked acknowledgements from previous receives and finally a marker which is used by the receiver to detect a completed message transfer.

In 3) the prepared send buffer is transmitted, including the actual data and the protocol tail. The buffer will stay in the state "used" until the remote side sends back an acknowledgement.

At step 4) pscom is informed about the progress. If the current chunk is the last chunk of a message, the associated higher level send request is marked as "done" and it is now safe to reuse the memory of the original message. Higher layers (e.g. MPI) can optionally be notified via a callback on this fact.

The receiving side waits for the reception of a chunk by polling on the memory location where the RMA transfer will place the marker of the protocol data tail. As the marker is the last byte of the receive buffer, it is safe to assume that the RMA put operation is completed, when the marker is there. In step 5) a pointer to the received data is passed to the pscom layer for copy out. Any acknowledges which might be in the tail are used to mark associated send buffers as "unused". Step 6 then copies the data to a final message buffer. If this was the last chunk of the message, the associated higher level receive request is marked as "done". Higher layers (e.g. MPI) are optionally notified via callback on this fact.

Finally the receive is acknowledged in step 7). Acknowledges are send back in the opposite direction to the sender piggyback with send requests. In the common case of a bi-directional communication there will be no extra RMA requests for acknowledgements. In the case of uni-directional communication too many outstanding acknowledges may accumulate. In this case an empty chunk containing just the protocol tail with acknowledges will be sent back explicitly. The RMA plugin supports this threshold to be configured.

Besides the buffer management, this acknowledgement scheme also puts a flow control into effect. It suspends the communication on the sending side, if the receiver does not consume the incoming data. The non-blocking implementation of pscom with independent event handling for every connection assures that suspending one connection will not block the progress of other connections.

2.4.1 *Buffer layout*

Each connection has its own set of send- and receive-buffers associated with it. All send-buffers and all receive-buffers of a single connection are allocated during connection start-up time as two contiguous blocks aligned to a page boundary with a system call to *valloc()*. Both regions are then registered to the EXTOLL layer for RMA transfers, but only the handle pointing to the receive buffer block is transmitted to the remote side. This is an extra security measure against code bugs since only receive buffers are allowed to be the target of an RMA operation.

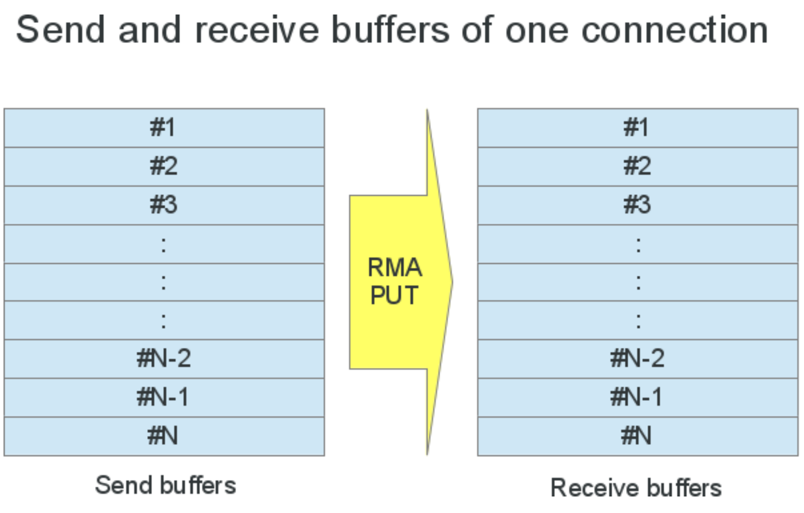

Figure 4: Send and receive buffers of one connection

Each communication buffer consists of three parts as shown in Figure 5. The first part is reserved for the payload with data and meta-data of higher level protocols like MPI. This payload itself will never be examined by the plugin and thus its content does not influence the progress logic. The second part is the protocol data tail with information only needed by the protocol. This tail has a size of 64 bits, which is the lower limit for a beneficial memory alignment and at the same time leaves enough room for protocol data. These are always the last 64 bits of a communication buffer. For performance reasons there are up to 7 padding bytes between payload and tail. This way, the beginning of the payload is always aligned to an 8-byte boundary which increases performance of the memory copy from the message chunk to the send buffer and from the receive buffer back to the final receiving message buffer. The EXTOLL RMA operation is also faster when it starts at an 8 byte boundary and the length is a multiple of 8.

Placing the tail behind the payload and aligning it to the upper bound of the buffer instead of a more commonly used protocol header in front of the payload provides some advantages: Being independent of the size of the payload, the marker will always be at the same position at the end of the communication buffer. This allows the receiver to monitor for a new chunk without knowing the length of the payload. At the same time it permits to send payload and tail with just a single RMA operation. Due to the special alignment of data and tail inside the buffer a different payload length does not change the position of the marker inside the tail, which is important for the receiver. Finally the marker has to be the last transmitted byte to assure that all bytes are transferred. This requirement is fulfilled using our one-RMA-operation-per-chunk approach.

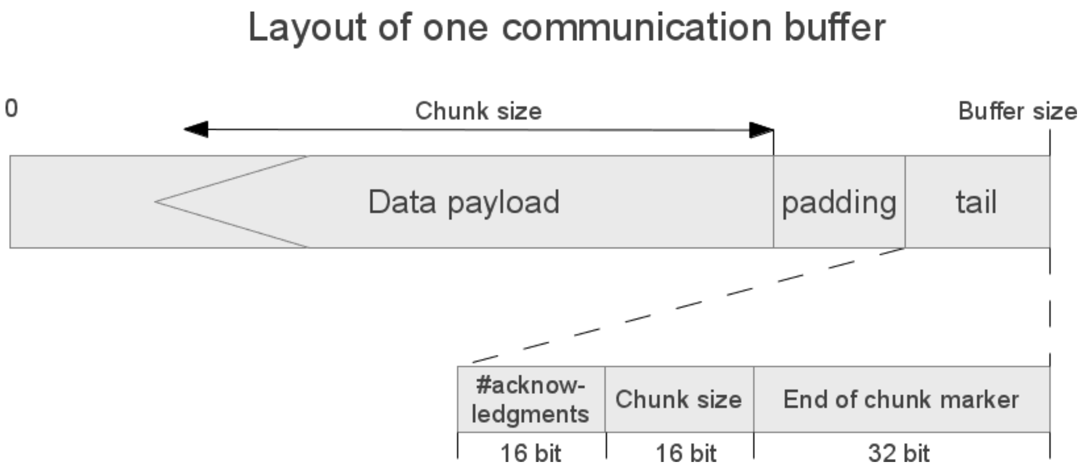

Figure 5: Layout of one communication buffer

2.4.2 *Memory considerations*

The current EXTOLL RMA plugin implementation requires the following amount of memory for buffer space:

$$buffer\ space = \#connections * (\#send\ buffers + \#receive\ buffers) * buffer\ size$$

The number of buffers should not be chosen too small to allow to send some chunks without the need of urgently receiving fresh acknowledgements: If the number is too small, communication might stall due to running out of usable send buffers. The result would be a limited maximal communication bandwidth. On the other hand, if we choose too many buffers, we need a lot of memory for them, which is then missing for the application itself.

The number of send and receive buffers is configurable at runtime through environment variables. The buffer size is fixed at compile time of the library. We have chosen 16 send and 16 receive buffers, each of size 4 Kbyte. Therefore every connection requires 128 Kbyte of buffer space. This means 1024 connections need 128 Mbyte of buffer space. Our experiments show that performance-wise, this is a good parameter set: Increasing the buffer size or buffer count only slightly increases the maximal throughput, decreasing the buffer count significantly lowers the throughput. The influence of these parameters should later be measured again and the parameters should be adopted for the final hardware of the DEEP System.

2.5 Description of the EXTOLL VELO plugin

The RMA protocol of EXTOLL shows very good performance for medium-sized and large messages. Due to the overhead of address translations on both the sending and the receiving side, small messages will be sent more efficiently using the VELO protocol. The EXTOLL VELO protocol is a message based protocol with send/receive semantics. It avoids the overhead of memory address translations and therefore shows a much smaller latency.

In particular for the Cluster-Booster protocol implementation the latency of small messages on the control channel is important to get a good overall performance.

To improve the small message performance we developed an EXTOLL VELO plugin for the pscom library. The implementation has to send chunks of data like the RMA implementation, but these chunks are much smaller. The current FPGA implementation of EXTOLL used inside the BIC evaluator supports VELO messages up to a length of 64 Bytes. This will be increased to 128 Bytes in future implementations of EXTOLL like the ASIC proposed for the final DEEP system.

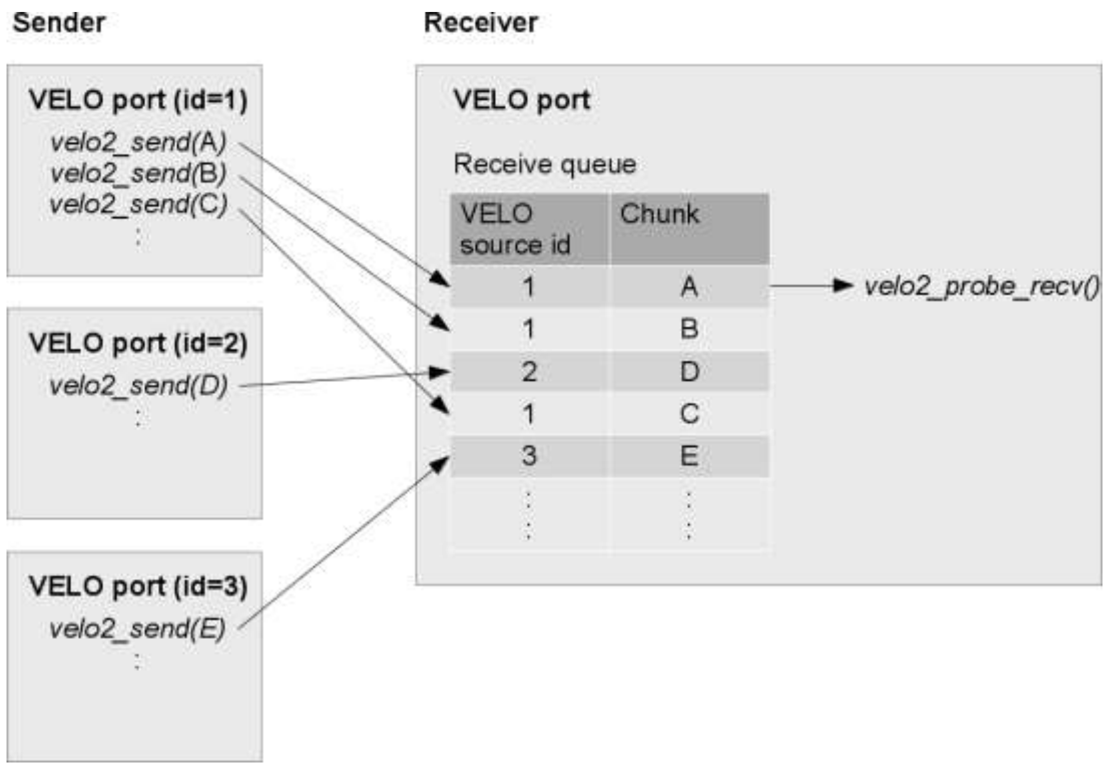

Figure 6: Send and receive with VELO ports

These chunks are directly sent to the VELO layer by the API call *velo2_send()* and received in a non-blocking way by *velo2_probe_recv()* without any address translation. Connection endpoints are VELO ports (Figure 6). Each pscom instance opens one VELO Port at start-up time und uses this port for every VELO I/O operation. A send operation addresses the destination with a handle to the VELO port of the recipient. The receiver has one receive queue shared between all connections. Incoming chunks from every connection are queued there in the order they arrive together with the source id of the originating VELO port. This means, that only one call to *velo2_probe_recv()* probes for the next chunk from any connection without the requirement of looping over every connections. The source id will then be mapped back to a pscom connection and the chunk is handed over to the upper layer for this pscom connection for further message processing. To protect the receive queue from overrunning, the VELO layer implements its own flow control on every connection.

3 Evaluation

Both implementations are functionally complete and running stable. We can choose the implementation to use at runtime by setting the LD_LIBRARY_PATH environment variable. We compared both implementations by running the “Intel MPI benchmark” [6] with modes "PingPong", which measures the overall half round trip latency and "SendRecv", which is good to measure the bidirectional maximal bandwidth. The results are shown in Figure 7 and Figure 8.

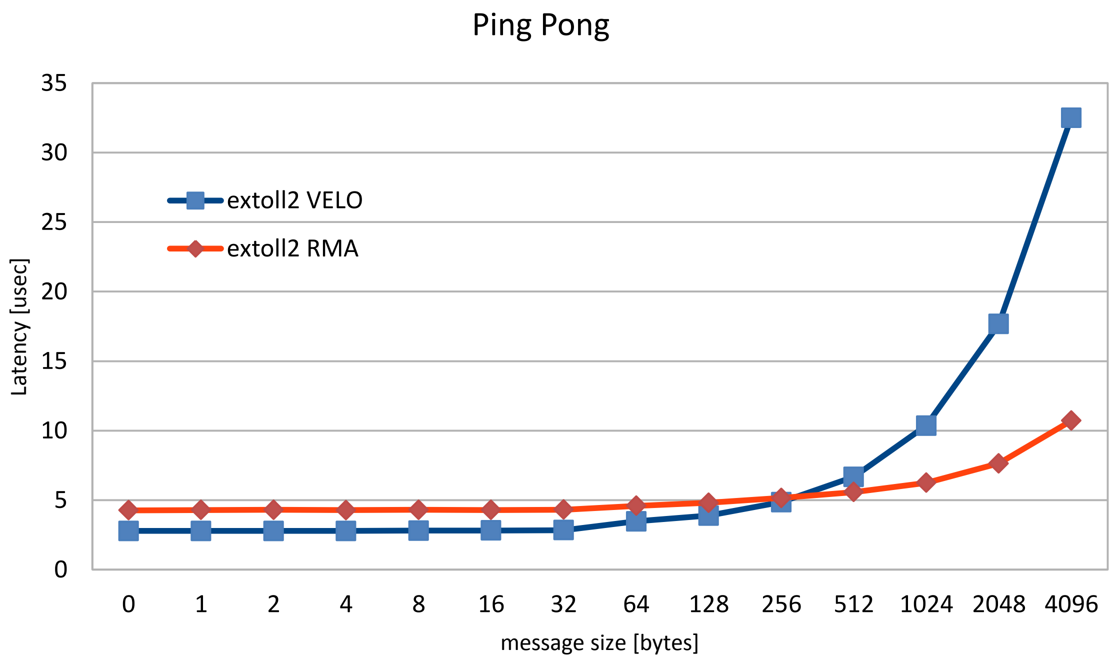

Figure 7: MPI Ping Pong Latency (half round trip time)

The measured VELO latency is nearly constant at 2.8µs for MPI messages up to 32 Bytes. Since the MPI layer adds a 12-Byte header containing *context_id*, *tag* and *rank* information to the message data, the 32 Byte messages are fitting well into one VELO message with an MTU of 64 Bytes. MPI Messages of 64 Bytes and above require more than one VELO message and therefore shows an increasing latency in the measurements:

MPI message size	VELO messages	Latency [µs]
<= 32	1	2.8
64	2	3.5
128	3	3.9
256	5	4.9
512	9	6.7

Table 1: VELO ping pong latency (half round trip time)

The RMA latency is about 4.3µs for messages up to 32 bytes. Compared to VELO this is significantly larger due to the overhead of additional address translations at the sending and the receiving side. Even if the RMA API allows to post "put" requests of up to 4 Kbyte, they will be split by the EXTOLL wire protocol to fragments of 64 Bytes. This explains a similar increase of the latency for messages above 64 Bytes. In contrast to VELO this fragmentation is handled by the EXTOLL adapter resulting in a better throughput. For MPI messages above 512 Bytes, the RMA implementation is faster than the VELO implementation.

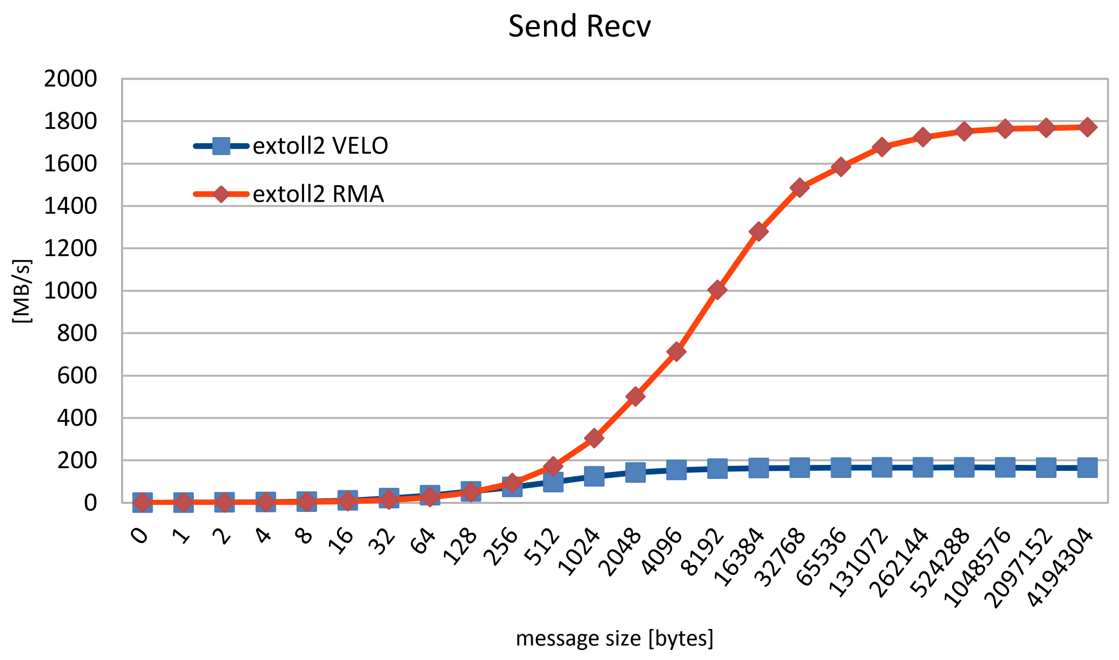

Figure 8: MPI Send Receive throughput

The same is true for the bidirectional bandwidth. Messages larger than 512 Bytes should be sent by the RMA implementation. Due to the software overhead of the message fragmentation for VELO, the throughput is already saturated at 8 Kbyte messages with 160 MB/s. Here the RMA implementation outperforms VELO with a maximal bidirectional bandwidth of about 1760 MB/s. The theoretical maximum bandwidth per link of the FPGA-based EXTOLL implementation is around 1086 MB/s as given by the SERDES capabilities. With the protocol overhead of the full network stack we achieve about 81% of the theoretical bidirectional bandwidth of 2172 MB/s.

References and Applicable Documents

[1] http://www.deep-project.eu
[2] ParaStation MPI: http://www.par-tec.com/products/parastation-mpi.html
[3] Extoll architecture: http://www.extoll.de/index.php/technology
[4] Deliverable D4.1 “Cluster Booster low-level protocol”
[5] Deliverable D4.2 “Full Cluster-Booster protocol without MIC”
[6] Intel MPI benchmark: http://software.intel.com/en-us/articles/intel-mpi-benchmarks

List of Acronyms and Abbreviations

A

API: Application Programming Interface

B

BI: Booster Interface (functional entity)

BIC: Booster Interface Card: Interface card to connect the Booster to the Cluster InfiniBand network

BIC evaluator: A platform consisting of three x86-based nodes equipped with (i) an EXTOLL NIC, (ii) an InfiniBand HCA, (iii) both, EXTOLL NIC and InfiniBand HCA, developed and used only in the DEEP project

BN: Booster Node (functional entity)

BNC: Booster Node Card: A physical instantiation of the BN

BNC evaluator: Same as EXTOLL evaluator

C

CN: Cluster Node (functional entity)

D

DEEP: Dynamical Exascale Entry Platform: EU-FP7 Exascale Project led by Forschungszentrum Juelich

DEEP Architecture: Functional architecture of DEEP (e.g. concept of an integrated Cluster Booster Architecture)

DEEP Booster: Booster part of the DEEP System

DEEP Supercomputer: A future Exascale supercomputer based on the DEEP Architecture

DEEP System: The production machine based on the DEEP Architecture developed and installed by the DEEP project

DMA: Direct Memory Access

E

EXTOLL: High speed interconnect technology for cluster computers developed by University of Heidelberg

EXTOLL evaluator: Platform for evaluation of EXTOLL technology, developed and used in the DEEP project

F

FPGA: Field-Programmable Gate Array: Integrated circuit to be configured by the customer or designer after manufacturing

G

Global MPI: MPI allowing communication between the Booster and Cluster part of the DEEP System. Based on the ParaStation process-management and the Cluster-Booster protocol acting as a plug-in for the pscom library. Provides the MPI_Comm_spawn() call used by application processes running on the CNs to start additional processes on the BNs.

H

HCA: Host Channel Adapter

I

IB: InfiniBand

M

MIC: Intel Many Integrated Core architecture

MPI: Message Passing Interface: API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages

MPICH: Freely available, portable implementation of MPI

P

ParaStation Consortium: Involved in research and development of solutions for high performance computing, especially for cluster computing

ParaStationMPI: Software for cluster management and control developed by ParTec

ParTec: ParTec Cluster Competence Center GmbH, Munich, Germany

R

RMA: Remote Memory Access: A protocol for remote memory access between EXTOLL NICs

S

SMFU: Shared Memory Functional Unit

V

VELO: Virtualized Engine for Low Overhead: An EXTOLL communications channel

W

WP: Work Package