SEVENTH FRAMEWORK PROGRAMME
Research Infrastructures

FP7-ICT-2011-7

DEEP

Dynamical Exascale Entry Platform
Grant Agreement Number: 287530

D4.6
ParaStation MPI supporting EXTOLL on MIC

Approved

Version: 2.0
Author(s): N. Eicker (JUELICH), J. Hauke (ParTec)
Date: 16.01.2015
Project and Deliverable Information Sheet

<table>
<thead>
<tr>
<th>DEEP Project</th>
<th>Project Ref. №:</th>
<th>287530</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project Title:</strong></td>
<td>Dynamical Exascale Entry Platform</td>
<td></td>
</tr>
<tr>
<td><strong>Project Web Site:</strong></td>
<td><a href="http://www.deep-project.eu">http://www.deep-project.eu</a></td>
<td></td>
</tr>
<tr>
<td><strong>Deliverable ID:</strong></td>
<td>D4.6</td>
<td></td>
</tr>
<tr>
<td><strong>Deliverable Nature:</strong></td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td><strong>Deliverable Level:</strong></td>
<td>PU *</td>
<td></td>
</tr>
<tr>
<td><strong>Contractual Date of Delivery:</strong></td>
<td>31 / January / 2014</td>
<td></td>
</tr>
<tr>
<td><strong>Actual Date of Delivery:</strong></td>
<td>31 / January / 2014</td>
<td></td>
</tr>
<tr>
<td><strong>EC Project Officer:</strong></td>
<td>Luis Carlos Busquets Pérez</td>
<td></td>
</tr>
</tbody>
</table>

* - 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

<table>
<thead>
<tr>
<th>Document</th>
<th>Title:</th>
<th>ParaStation MPI supporting EXTOLL on MIC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID:</strong></td>
<td>D4.6</td>
<td></td>
</tr>
<tr>
<td><strong>Version:</strong></td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td><strong>Status:</strong></td>
<td>Approved</td>
<td></td>
</tr>
<tr>
<td><strong>Available at:</strong></td>
<td><a href="http://www.deep-project.eu">http://www.deep-project.eu</a></td>
<td></td>
</tr>
<tr>
<td><strong>Software Tool:</strong></td>
<td>Microsoft Word</td>
<td></td>
</tr>
</tbody>
</table>

**File(s):**
DEEP_D4.6_ParaStationMPI_supporting_Extoll_on_MIC_v2.0_ECapproved

**Authorship**

| Written by: | N. Eicker (JUELICH), J. Hauke (ParTec) |
| Contributors: | T. Moschny (ParTec), M. Rauh (ParTec), A. Galonska (JUELICH) |
| Reviewed by: | U. Bruening, UniHD, E. Suarez, JUELICH |
| Approved by: | BoP/PMT |

Document Status Sheet

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>31.01.2014</td>
<td>Final</td>
<td>EC-submission</td>
</tr>
<tr>
<td>2.0</td>
<td>16.01.2015</td>
<td>Final</td>
<td>EC approved</td>
</tr>
</tbody>
</table>
Document Keywords

| Keywords:       | DEEP, HPC, Exascale, EXTOLL, Xeon Phi, ParaStation |

Copyright notices

© 2011-2015 DEEP Consortium Partners. All rights reserved. This document is a project document of the DEEP project. All contents are reserved by default and may not be disclosed to third parties without the written consent of the DEEP partners, except as mandated by the European Commission contract 287530 for reviewing and dissemination purposes.

All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.
Table of Contents

Project and Deliverable Information Sheet ................................................................. 2
Document Control Sheet ................................................................................................. 2
Document Status Sheet ................................................................................................. 2
Document Keywords ..................................................................................................... 3
Table of Contents ......................................................................................................... 3
List of Figures ................................................................................................................ 4
List of Tables .................................................................................................................. 4

Executive Summary ..................................................................................................... 5
1 Introduction .................................................................................................................. 5
2 ParaStation on the MIC ............................................................................................. 7
3 Pscom shared memory “direct shared memory” ....................................................... 8
   3.1 Implementation ....................................................................................................... 8
   3.2 Evaluation: Pscom shared memory communication on KNC ................................ 10
4 EXTOLL on KNC ......................................................................................................... 11
   4.1 Implementation: Pscom VELO plugin with zero copy RMA ............................... 11
   4.2 Evaluation pscom’s EXTOLL plugins on KNC .................................................... 12
5 Memory considerations ............................................................................................. 14
   5.1 Shared memory ..................................................................................................... 15
   5.2 EXTOLL ................................................................................................................ 15
6 MPI_Comm_spawn ....................................................................................................... 16
7 Cluster Booster protocol .......................................................................................... 17
8 Conclusion ................................................................................................................... 18
9 References and Applicable Documents ................................................................. 19
10 List of Acronyms and Abbreviations ....................................................................... 20

List of Figures

Figure 1: DEEP Software Architecture .................................................................. 6
Figure 2: IMB-MPII: Sendrecv (EXTOLL) on KNC, Gnu compiler vs. Intel compiler ........ 7
Figure 3: IMB-MPII: Sendrecv on KNC, comparison shared memory protocols .......... 10
Figure 4: IMB-MPII: Ping Pong (Half roundtrip time) on KNC, Comparison shared memory protocols .................................................................................................................. 11
Figure 5: IMB-MPII: Ping Pong (Half roundtrip time) on KNC, comparison of EXTOLL protocols .......................................................... 13
Figure 6: IMB-MPII: Sendrecv on KNC, comparison of EXTOLL protocols ............... 13
Figure 7: Schematic view on ParaStation pscom with the CBP as an additional plug-in .... 17

List of Tables

Table 1 Shared memory buffer space .......................................................................... 15
Table 2 EXTOLL buffer space ....................................................................................... 15
**Executive Summary**

MPI plays a crucial role in the software architecture underlying the DEEP Programming environment. Therefore, optimisation of the corresponding libraries is mandatory to achieve a good performance and scalability of the applications running on the DEEP System.

This document describes various aspects of the functional extensions and optimisations on ParaStation MPI – the implementation of the Message Passing Interface standard chosen for the DEEP System. This includes:

- porting ParaStation MPI to the MIC architecture utilised within the DEEP Booster,
- adding support for the EXTOLL interconnect used as the Booster fabric,
- optimising shared memory communication on the Booster Nodes,
- integrating the Cluster-Booster protocol and, last but not least,
- fully supporting MPI_Comm_spawn() and its integration with the resource management.

The present deliverable is addressed to software developers from WP5 and application developers from WP8. WP5 uses MPI as a low-level functionality in order to implement the OmpSs off-loading abstraction that shall hide most of the complexity required to off-load highly-scalable code-part from the Cluster to the Booster. WP8 will use the described extensions and optimisations in order to implement communication within the highly-scalable code-parts running on the Booster part of the DEEP System.

1 **Introduction**

Both parts of the DEEP System – Cluster and Booster – will have their own interconnect (InfiniBand (IB) and EXTOLL, respectively) adapted to the requirements of the specific application code-parts. The DEEP Programming model foresees the use of MPI in various roles: (i) to enable communication within the less-scalable code parts residing on the Cluster and the highly-scalable code parts to be off-loaded to the Booster, (ii) to realise the actual off-loading mechanism using MPI_Comm_spawn(), and (iii) to allow for communication between the code-parts running on the different parts of the system. This list of requirements was collected during the first phase of the DEEP project and is the result of close collaboration between the WP4 and WP5 technical work-packages, based on the input provided by the application developers of WP8. In a second step the capabilities of early prototypes of the MPI components were analysed by WP8 and the feedback of the application developers was used in order to tune and optimise their features.

Since all requirements set by the DEEP programming model are already covered by the MPI standard, the decision to implement a global MPI spanning the whole system and supporting all technologies utilised in the DEEP System was straight forward. This decision is reflected by the central position of the ParaStation Global MPI in the DEEP Software architecture as depicted in Figure 1.

The realisation of such Global MPI on the basis of ParaStation MPI was significantly supported by its modular design. It enables WP4 to extend the communication library pscom by two new transport layers supporting intra-Booster communication and Cluster-Booster communication. Since all transport layers in pscom are implemented within independent plugins, both extensions are straight forward to realise. Furthermore, since plugins are orthogonal by design, this enables developers to work on both tasks in parallel.

Due to the extensive use of MPI an optimal implementation of the various components is crucial for the overall performance of the DEEP System. While the realisation of MPI-
communication within the DEEP Cluster is straight forward due to the use of Xeon processors and the InfiniBand interconnect, all further components are beyond the standard feature-sets of modern MPIs and require special efforts. In the case of communication within the DEEP Booster, the challenge is two-fold: On the one hand the novel interconnect technology provided by EXTOLL has to be supported in an optimised way, on the other hand the specific attributes of the Intel Xeon Phi processors used in the DEEP Booster have to be taken into account. The latter includes the optimisation for its in-order architecture and the challenges stemming from its intra-processor shared-memory architecture.

The specific challenges of the Cluster-Booster communication were already discussed extensively in various deliverables (D4.1 [8], D4.2 [9] and D4.5 [11]) from a low-level point of view. Within this document we describe the actual integration into ParaStation MPI’s communication layer.

![Figure 1: DEEP Software Architecture](image)

Last but not least the off-loading mechanism to be utilised either by the OmpSs offload abstraction layer or directly via the MPI_Comm_spawn() call will be discussed. Beyond starting processes on the Booster part of the DEEP System this mechanism relies on the composition of communication channels between Cluster- and Booster-processes in a stable way. Furthermore the tight integration with the resource-management capabilities to be developed in Task 4.4 is crucial.

Starting with a short description of the porting efforts of ParaStation MPI to Intel’s MIC architecture this document describes the actual implementation details of several plugins for ParaStation’s pscom communication library developed in the course of this project. This includes the VELO plugin that extends EXTOLL’s very efficient low-overhead (VELO [6]) capabilities for small messages by an RMA [7] based optimisation allowing zero-copy transfers for larger messages. A second plugin realising optimised intra-MIC shared-memory communication is discussed in detail, too. Besides that, the according chapters also include the presentation of performance results in order to provide the key figures of the corresponding communication channels. All data was generated on the prototypical hardware available to the work-package at this stage of the project: the Super-BIC evaluator. The
following chapter some considerations concerning the expected memory footprint are discussed. Since memory is a scarce resource on Intel Xeon Phi it turned out to be crucial for application developers to find the sweet-spots for the tuneable parameters of the communication library within a co-design effort between WP4 and WP8.

The next chapter presents the realisation of MPI_Comm_spawn that will be used for the actual off-loading of the highly-scalable code-parts to the Booster. Before we conclude, Chapter 7 gives an update on the integration of the Cluster-Booster protocol into ParaStation MPI.

2 ParaStation on the MIC

All existing components of ParaStation MPI [3] had to be ported to Intel’s MIC platform in order to support MPI on the DEEP Booster. Besides the actual implementation of the MPI library based on MPIch [4] this includes further low-level components like psmgmt and pscm.

The actual port of the components required the adaptation of ParaStation’s build system to the need of cross-compilation on MIC. Nevertheless, an unexpected pitfall was identified in the course of porting and collecting first performance results on the new platform. On all CPU and interconnect platforms used before the choice of the compiler was not crucial. MIC turned out to be different. To give an example, Figure 2 compares the Sendrecv throughput achieved using the EXTOLL fabric on the Knights Corner (KNC) realisation of the MIC architecture provided by different compilers.

![Figure 2: IMB-MPI1: Sendrecv (EXTOLL) on KNC, Gnu compiler vs. Intel compiler](image)

Initially the GNU cross compiler for x86_64-k1om was used to build ParaStation. The corresponding results for Sendrecv-throughput are depicted by the blue data in Figure 2. Due to the disappointing performance results shown by this combination, the Intel compiler for KNC was used for a cross-check. It turned out that the compiler might have a significant
influence on the throughputs that can be achieved depending on the message size to be submitted.

The diagram in Figure 2 shows Sendrecv throughput measured with the Intel MPI benchmark IMB-MPI1[12] communicating over EXTOLL with the RMA2 protocol. It shows a significantly higher throughput (550 MB/s vs. 320 MB/s) for the communication library created by Intel’s compiler for intermediate message sizes between 8 kB and 128 kB. It seems that the GNU compiler tool-chain is not yet highly optimised for the in-order architecture of the KNC. This was underpinned by other experiments that led to the assumption that the Intel compiler comes with a runtime highly optimised for KNC. The GNU compiler is currently lacking a similar feature. Therefore, it was decided to build all libraries for the KNC platform with Intel’s compiler only.

3 Pscom shared memory “direct shared memory”

Due to the high number of cores on a KNC we cannot expect to have only one MPI process per BN. In order to enable an optimised communication between different MPI processes running on the same BN, ParaStation provides a shared memory communication channel. Since the port of the standard plugin for this channel did not show the expected performance, a detailed analysis of the performance numbers unveiled the potential to improve the behaviour significantly.

3.1 Implementation

A first problem of the original implementation is the use of dedicated shmem segments shared by the communicating processes in order to pass data from one address-space to the other. This strategy includes two copy operations for a full data transfer. Since this turned out to be expensive on MIC, a “one copy” shared memory protocol for local communication was implemented. An additional motivation is the reduction of the memory consumption per connection since the dedicated shmem segments require less memory.

For this to work, a hook in glibc’s malloc implementation was used in order to allocate every heap memory region requested by the application inside a Sys V shared memory region. All dynamic allocated memory will reside in this region. This allows any other process with the corresponding permissions to map this region in its own virtual address space. This provides direct access to memory of other processes (hence the name “direct shared memory”). With this approach message data can be transmitted with just one memory copy operation directly from the source buffer into the destination buffer. In our implementation this memcopy is executed by the receiving process. In contrast, the older implementation copied the data twice: First, on the sending side from the source buffer to the dedicated shmem buffer shared by the two processes and then later at the receiving side from the shared shmem buffer to the final destination buffer.

In detail, transmitting with direct shared memory works as follows:

If the message to be sent is larger than the rendezvous size,

1) Check if the message data pointer is pointing into the direct shared memory region. If not, fall back to the double-copy strategy and continue with 1b (see below).

2) Transmit the metadata of the message together with the virtual address of the data to the receiving side, using the small message protocol.

Receiver:

3) Find the destination buffer of a receive request matching the message metadata.
4) Translate the foreign virtual address of the data to a virtual address of the receiver. This is done by adding an offset which is calculated at connection start.

5) Copy the data from the remote source to the local destination buffer via memcpy using the data pointer from 4).

6) After completion of the memcpy operation:
   - Notify the application about the new message.
   - Send an acknowledge control message via send/receive buffers back to the sender.

Sender:
7) Upon receipt of the acknowledge message, notify the application about the completed send. It is now safe to reuse the send buffer.

If the data is not located in the direct shared memory region:
   1b) Dynamically allocate temporary memory inside the shared memory region. If this also fails, fall back to buffered send/receive.

   2b) Copy the message’s data to the temporary buffer. Since this is an expensive operation it should be avoided by using dynamic allocated memory for application data. Nevertheless, copying the data to a single, large temporary buffer is still significantly faster than using the send/receive buffers, where data has to be copied in many small chunks. The latter turned out to be a major bottleneck on KNC.

   3b) Send the virtual address of the temporary buffer to the receiver and notify the application that the send buffer can be reused.

Receiver:
4b) to 6b) analogous to 4), 5) and 6) with applications source buffer replaced by the temporary buffer.

Sender:
7b) Free the temporary buffer after getting the acknowledge message.

To get the full performance of this implementation, applications have to use dynamic allocated memory. For C and C++ applications this is typically the case if they use malloc, realloc or new. FORTRAN applications have to use “allocatable” memory:

```fortran
INTEGER, ALLOCATABLE :: data(:)
allocate(data(15000))
```

Statistics about the used memory for local communication can be dumped at the end of an application run by setting the debug level (PSP_DEBUG) greater or equal to 2, or by explicitly requesting it via the environment variable PSP_DEBUG_STATS=1.

Applications equipped with their own memory management may conflict with our implementation. They can disable the hook into glibc by setting PSP_MALLOC=0, but then they can not benefit from the direct shared memory and will use the buffered send/receive for local communication instead.
3.2 Evaluation: Pscom shared memory communication on KNC

A first measurement was done using local communication via shared memory with the same parameter set that was used on x86_64 Intel Xeon nodes. This just gave about 360 MB/s throughput for Sendrecv at 64 kBytes, and less than 300 MB/s above 256 kBytes message size (Figure 3, buffered shm). To improve this, we had to increase the size of the shared memory buffers from 8 kBytes to 128 kBytes. This caused a huge speedup for messages of size 16 kBytes and above from 345 MB/s to 1600 MB/s (Figure 3, big buffers). Unfortunately, this parameter set consumes a lot of memory for the communication buffers (see also Chapter 5 for a detailed discussion). Since every local connection has its own set of shared communication buffers it is to be expected that – due to the many local connections resulting from the high core-count on KNC – this does not scale well. With the implementation of the “direct shared memory” we have build another local communication protocol with a significantly reduced memory footprint that at the same time gives a much higher throughout (5874 MB/s at 64 kBytes, 2409 MB/s at 4 MBytes). The rendezvous protocol is used for message sizes above 400 Bytes (this default might be adjusted via \texttt{PSP\_SHM\_DIRECT}).

![Send Receive](image)

Figure 3: IMB-MPI1: Sendrecv on KNC, comparison shared memory protocols
4 EXTOLL on KNC

4.1 Implementation: Pscom VELO plugin with zero copy RMA

The VELO plugin of pscom for the EXTOLL network was extended by a rendezvous protocol for zero copy RMA. The goal of this effort is to combine the low latency of EXTOLL’s VELO protocol with the high throughput of the RMA protocol for large messages. VELO is utilised to transfer small messages and to initiate zero copy RMA transfers for large messages. Due to this design, it is sufficient to listen only for incoming VELO messages on the receiving side.

The rendezvous for large messages (i.e. messages larger than the rendezvous size) works as follows.

Sender:

1) Register the data buffer of the message for RMA usage. For the EXTOLL stack this means pinning down all memory pages containing message data, preventing these pages from being swapped out to disk or being moved to a different location in physical memory. After that, map the physical addresses of all affected pages to a consecutive range of EXTOLL’s network logical addresses (NLA).

2) Transmit the metadata of the message together with the NLA of the message data to the receiving side.

Receiver:

3) Identify the destination buffer of a receive request matching the message metadata.

4) Register the destination buffer for RMA usage (pinning and assign a NLA).

Figure 4: IMB-MPI1: Ping Pong (Half roundtrip time) on KNC, Comparison shared memory protocols
5) Initiate a RMA Get operation from the remote source buffer to the local destination buffer. The RMA Get will address the source and destination regions with the NLAs from step 1) and step 4). A completion notification when the RMA is done is requested at the same time.

6) After completion of the RMA operation:
   - Notify the application about the new message.
   - Unregister the receive buffer.
   - Send an acknowledge control message via VELO back to the sender.

Sender:

7) When receiving the acknowledge message:
   - Unregister the send buffer.
   - Notify the application about the completed send. It is now safe for the application to reuse the send buffer.

The registration of data buffers is handled by the librma2 of EXTOLL’s communication stack. The current implementation of this functionality suffers from a significant overhead required to register and release the communication buffers each time a buffer is used. Since typical applications in HPC tend to reuse communication buffers frequently, most often the release and re-registration of memory buffers is unnecessary. Therefore, future versions of the librma2 will cache registrations of memory regions. This will reduce the registration overhead of frequently re-used buffers significantly and probably increase the throughput.

Per default, the new rendezvous protocol is used for messages larger than 1024 Bytes. For smaller messages VELO is used directly. For the sake of flexibility this threshold for the protocol switch can be adjusted at runtime by setting the environment variable PSP_RENDEZVOUS_VELO.

4.2 Evaluation pscom’s EXTOLL plugins on KNC

The MPI half-round-trip time of messages sent by the VELO plugin is 5.5μsec. This has to be compared to the RMA plugin’s latency of 6.4μsec (lower latency, Figure 5). The new rendezvous protocol is used for messages of size 1024 Bytes and greater. Between 64 and 32 kBytes the RMA plugin still achieves a higher throughput than the hybrid VELO/RMA approach (Figure 6). This is most probably caused by the additional overhead for the memory registration on both sides. We expect that caching of these registrations, as planned for the next version of the librma2, will reduce this overhead.

The maximum Sendrecv throughput for “RMA only” of 472 MB/s is reached at a message size of 16 kbytes, but decreases for larger messages down to 326 MB/s. Compared to this the new RMA rendezvous protocol provides a much better result with a final throughput of 875 MB/s at 4 MBytes message size as depicted in Figure 6.
We recommend users of ParaStation MPI to use the new VELO plugin with the rendezvous protocol for zero-copy RMA enabled for their applications. It is expected that it will show
lower latency, higher throughput, and less memory usage at the same time. Only applications without large memory requirements and a typical message size between 64 and 32kBBytes might benefit from the old RMA plugin. Nevertheless, the introduction of an updated libma2 implementing the cached memory registrations might even eliminate this performance benefit of the RMA plugin.

For comparison Figure 5 and Figure 6 also show results attained on the widely used Xeon platform. This platform shows much better results, i.e. a significantly reduced latency and much higher throughput for both protocols. The root cause for the worse behaviour on the KNC platform is still unclear and subject to investigation. Nevertheless, there are various possible reasons:

- KNC is an in-order platform compared to the distinct out-of-order optimisations found on the Xeon platform.
- DMA on KNC might be optimised for exchanging data with the host-processors main memory. I.e. KNC’s DMA engine is optimised for accessing external memory while grasping KNCs memory from the outside – as conducted by EXTOLL – seems to be sub-optimal.
- The large core count of KNC makes each and every operation on memory significantly more sensitive to cache-coherence protocols.

5 Memory considerations

The more MPI processes will be used on each Booster Node, the more important it is to reduce the memory consumption per connection. This is crucial since the number of connections per MPI job exhibits quadratic growth in the number of processes. To give a worst case example: if we would run one MPI process per KNC hardware thread, a total of 29646 local connections have to be established \( n \times (n-1)/2 \) with \( n=244 \). Since each connection requires some dedicated memory in order to be operational, the corresponding memory footprint would render this approach unusable. Significant improvements have to be achieved.

A first step is to reduce the amount of connections by using a hybrid programming model combining OmpSs and MPI for the compute kernels on the Booster. By this means multiple threads of one MPI process can share connections to other processes and thus also share corresponding memory buffers. In this scenario we do not expect more than about 32 MPI processes per Booster Node.

The next step is to establish only such connections that are actually used by the application. In reality most applications in HPC do not exhibit an all-to-all communication pattern but de facto employ only a few connections. In ParaStation MPI’s pscom library this is taken care of by so called “on demand” connections. Even though each MPI process (or rank to stick with MPI nomenclature) is logically connected to every other process (rank) of the same MPI job, the low-level connection will be deferred until the first data actually needs to be sent. At this point of time the actual connection is created “on demand”. If possible, applications shall avoid sending from all ranks to all other ranks with point-to-point MPI calls by looping over all ranks. Instead, the MPI standard advises to use collective operations. The key reason for that is to enable an MPI implementation to apply optimisations like tree-based algorithms to scatter and gather data, etc.

Finally, the reduction of the memory-footprint for each connection is crucial. The current pscom implementation requires at least 400 Bytes per connection to store the corresponding metadata (even for not established “on demand” connections). Additional memory is required for network specific metadata and buffer space. This depends on the chosen network path.
Depending on the actual implementation of the connection protocol the amount of memory – especially for the buffers – might be considerable.

### 5.1 Shared memory

With the implementation of the “direct shared memory” plugin for the pscom we reduced the memory required for communication buffers on local connections to 16 kByte per connection endpoint. Additionally, 400 Bytes are utilised by the connection’s metadata (cf. Table 1 Shared memory buffer space). By comparison, the default implementation for x86_64 Xeon nodes “buffered shared memory” consumes 64 kBytes per endpoint. Nevertheless, this implementation should not be used on KNC, as the throughput is very limited (below 420 MB/s). To improve the throughput on KNC we had to increase the size of communication buffers considerably (“big buffers”) to as much as 1 MByte per connection endpoint. Even for just 32 MPI processes per KNC this would already consume about 1 GBytes for shared memory buffers (32 * (32-1) * 1MByte). In contrast, the preferred new “direct shared memory” plugin achieves the highest throughput on KNC and scales much better (consuming just about 16 Mbytes per KNC node for 32 MPI local processes per node).

<table>
<thead>
<tr>
<th>Shared memory protocol</th>
<th>Buffers per connection</th>
<th>Buffer size [kByte]</th>
<th>Buffer space per connection endpoint [kByte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>“buffered”</td>
<td>8</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>“big buffers”</td>
<td>8</td>
<td>128</td>
<td>1024</td>
</tr>
<tr>
<td>“direct shared memory”</td>
<td>16</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 1 Shared memory buffer space

### 5.2 EXTOLL

By employing EXTOLL’s message-based VELO2 protocol the pscom plugin is able to send and receive small messages without need for large pre-allocated communication buffers. In addition, the send and receive queues of this protocol can be shared between all connections of a single process. Only the memory required to store the connection metadata scales with O(N^2). The new hybrid pscom plugin “VELO2 + RMA2” requests RMA buffers on demand and uses the memory provided by the application (zero copy). Using a rendezvous protocol over VELO2 to initiate a zero-copy RMA2 transfer gives a good throughput with the same scaling behaviour for memory footprint as the VELO2 protocol.

<table>
<thead>
<tr>
<th>EXTOLL protocol</th>
<th>Buffers per connection</th>
<th>Buffer size [kByte]</th>
<th>Buffer space per connection endpoint [kByte]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Send</td>
<td>Receive</td>
<td></td>
</tr>
<tr>
<td>RMA2</td>
<td>16</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>VELO2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VELO2 + RMA2 rendezvous</td>
<td>Zero copy User implicitly provides buffers with each send or receive request.</td>
<td>Individually defined by each send or receive request</td>
<td>dynamic</td>
</tr>
</tbody>
</table>

Table 2 EXTOLL buffer space
6 MPI_Comm_spawn

The off-loading of highly-scalable code parts to the Booster is a central aspect of the DEEP programming model. During the early design-phase of the DEEP System extensive discussions between the developers of the DEEP system-software from WP4 and WP5 and the application developers from WP8 unveiled the requirements for implementing the actual off-loading mechanism and the infrastructure needed for exchanging data between the application parts running on the Cluster and the Booster. A detailed analysis of these requirements manifested their congruence with the functionality provided by MPI’s MPI_Comm_spawn() call and the use of inter-communicators. Therefore, it was decided to stick with the MPI standard.

Off-loading of highly-scalable code-parts from the Cluster to the Booster is implemented by means of MPI_Comm_spawn() to spawn processes. At the same time communication channels between the processes residing on the Cluster and the spawned processes on the Booster are created. MPI_Comm_spawn() starts identical copies of an MPI program, establishes communication channels between them and the calling processes and returns an MPI inter-communicator. In the following the spawned processes are referred to as “children” while the processes collectively calling MPI_Comm_spawn() are referred to as “parents”. Parents and their children will live in the same MPI_COMM_UNIVERSE but in different process groups. The two process groups can exchange data by using the inter-communicator returned by MPI_Comm_spawn() and MPI_Get_Parent(), respectively. MPI_Comm_spawn() is a collective operation and does not return until all spawned children have called MPI_Init().

ParaStation MPI utilises the process management interface (PMI) implemented in MPIch [4] to exchange information between the processes of a parallel application and to spawn new processes via the ParaStation process management (psmgmt). When an MPI process is started, it is connected to the local ParaStation process-management daemon via UNIX domain socket to allow for PMI communication. All processes belonging to the same process-group, i.e. in our case the parents or their children, will share a common key-value space (KVS) in order to exchange information required to establish communication channels between these processes.

When a group of MPI process calls MPI_Comm_spawn() the MPI layer will issue a call to the PMI_spawn() function of one of the involved parent processes. The spawned children will live in a separate MPI_COMM_WORLD i.e. they will form their own process group. From the process management’s point of view the children have their own KVS to store their contact information. Numbering of MPI ranks will start again at 0 within the children’s process group.

As a first step PMI_spawn() will start a new service process hosting the children’s KVS. To actually start the child processes, the corresponding environment is prepared; the main management structures (e.g. holding the user id) are cloned from the calling process and sent by means of a spawn request message. If the job has enough resources allocated, the spawn request will succeed and the binary specified as an argument to the MPI_Comm_spawn() call will be executed.

While the MPI children processes might get a handle to the inter-communicator by calling MPI_Comm_get_parent(), the parent processes will receive the corresponding handle to the inter-communicator as a return value of the MPI_Comm_spawn() call.
7 Cluster Booster protocol

As discussed, the software architecture of DEEP foresees a global MPI spanned over the whole system (Figure 1). On the Cluster, low-level communication is done via InfiniBand while on the Booster the EXTOLL communication engines VELO and RMA come into play.

![ParaStation PSCOM diagram](image)

**Figure 7: Schematic view on ParaStation pscom with the CBP as an additional plug-in**

Between Cluster and Booster the Cluster-Booster communication is done by means of the Cluster-Booster protocol (CBP).

The CBP provides functionality to create direct routes between a CN and a BN utilising InfiniBand and EXTOLL’s SMFU in conjunction (cf. D4.2[9], D4.3[10] and D4.5[11]).

The relation between CBP and pscom is bidirectional: On the one hand pscom is employed by the CBP as a transport layer for protocol metadata. For this, the InfiniBand verbs layer is used between CN and BI, whereas between BI and BN EXTOLL’s VELO or RMA protocols are utilised. On the other hand the CBP itself is also a part of the pscom library serving as an additional plug-in. From pscom’s point of view, this is an additional network protocol. The CBP plug-in has to avoid ending up in a dead-lock when simultaneously using pscom and providing it with a new network protocol. The plug-in provides an important functionality to pscom in order to span the global MPI in DEEP by adding the missing link between Cluster and Booster.

The CBP plug-in provides the following functionalities to the pscom library:

- Mapping of the EXTOLL Shared Memory (SMFU) and initialisation of the InfiniBand verbs layer for the actual connection
- Connection to another process
- Accept connections from another process
- Look-up for new data
- Send data to another process
• Close a CBP connection

The global MPI approach was tested on the Super-BIC evaluator using the Intel MPI Benchmark. The Ping Pong scenario was tested between a Cluster and a Booster Node with a metadata server running on the BI.

8 Conclusion

The DEEP programming model utilises MPI in various places. Besides providing a communication layer within both, Cluster and Booster, it is used to actually off-load highly-scalable code parts to the Booster and to establish efficient communication between Cluster and Booster processes. Due to its wide use, an efficient implementation of all aspects is crucial for the overall performance of the DEEP System.

Especially on the Booster side of the DEEP system memory is a scarce resource. This is due to the fact that KNC today is only available with a maximum memory of 16 GB per socket. Thus, reducing the memory consumption per connection is important for the potential to scale applications without running out of main memory. To achieve this, a new shared memory protocol was introduced providing high throughput for local communication within KNC and at the same time reducing the buffer space required per connection. Additionally, pscom’s VELO plugin was enhanced by a rendezvous protocol to implement zero-copy RMA functionality on top. This combination combines the low latency of the VELO protocol with the high throughput of the RMA protocol. At the same time the good scaling behaviour of VELO due to its small memory footprint is sustained. We expect further improvements for messages between 32Bytes and 64kBytes by implementing an RMA registration cache.

We have implemented MPI_Comm_spawn and its process management features to be able to start highly-scalable code parts on the Booster initiated from the Cluster. The Cluster-Booster protocol was integrated into ParaStation MPI.
9 References and Applicable Documents

[1] [http://www.deep-project.eu](http://www.deep-project.eu)


VELO: A Novel Communication Engine for Ultra-low Latency Message Transfers
37th INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP-08), September 08–12, 2008, Portland, Oregon, USA


A resource optimised remote-memory-access architecture for low-latency communication
The 38th International Conference on Parallel Processing (ICPP-2009), September 22–25, Vienna, Austria


[10] Deliverable D4.3 “ParaStation component pscom supporting EXTOLL”


10 List of Acronyms and Abbreviations

A

API: Application Programming Interface

B

BAR: Base Address Register. Resides in the configuration space of a PCI peripheral device. It describes a memory region exposed by the peripheral to the system. Loosely also used to name said memory region.

BIC: Booster Interface Card: Interface card to connect the EXTOLL based Booster network 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

CBP: Cluster-Booster protocol

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 device 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.

**K**

**KNC:** Knights Corner. Code name of the first generation of Xeon® Phi™ CPUs  
**KVS:** Key-Value Space

**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  
**MPSS:** Intel many-core platform software stack. Software bundle to operate Xeon® Phi™ devices

**N**

**NLA:** Network Logical Address

**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  
**PMI:** Process Management Interface

**R**

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

**S**

**SMFU:** Shared Memory Functional Unit. Used by the EXTOLL network for mapping remote memory regions.

**V**

**VELO:** Very Efficient Low-Overhead

**W**

**WP:** Work Package

**X**

**X86:** The general term for microprocessor architectures derived from the Intel® 80386