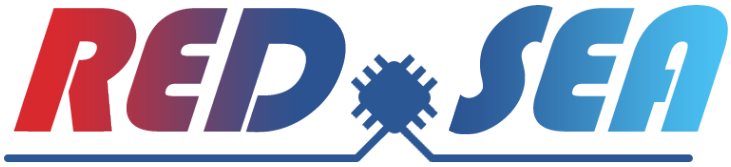
Network Solution for Exascale Architectures



D4.3: Planned MPI-related optimizations

Document Properties

Contact Number	955776
Contractual Deadline	30/09/2022
Dissemination Level	Public
Nature	Report
Edited by:	Hugo Taboada - CEA
Authors	Gilles Moreau - CEA Hugo Taboada - CEA Marc Pérache - CEA Simon Pickartz - ParTec Carsten Clauss - ParTec
Reviewers	Timo Schneider - ETHZ Guiseppe Piero Brandino - eXact lab
Date	September 2022
Keywords	MPI, ParaStation MPI, MPC-MPI, Optimizations
Status	Final
Release	1.0

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955776. The JU receives support from the European Union's Horizon 2020 research and innovation programme and France, Greece, Germany, Spain, Italy, Switzerland.

History of changes

Release	Date	Author, Organization	Description of Changes
1.0RC01	08/31/2022	CEA	Ready for internal review
1.0	09/22/2022	CEA	Final version including the comments and changes from the internal reviewers

Contents

List of Figures

List of Tables

Executive Summary

This document presents the two contributions from Commissariat à l'énergie atomique et aux énergies alternatives (CEA) and ParTec on the development of Message-Passing Interface (MPI) optimizations on their respective platform Multi-Processor Computing (MPC) and ParaStation MPI. This work has been conducted in the context of WP 4, Task4.5. The objective of this intermediary deliverable is the implementation of first prototypes of MPI-related optimisations for BullSequana eXascale Interconnect (BXI) networks.

CEA developed a first prototype of a feature called multirail to take advantage of new node infrastructure containing multiple Network Interface Cards (NICs). Developments in the meantime also layed the ground for potential collaboration with the DEEP-SEA project on the integration of a gateway feature used in supercomputers with Modular Supercomputer Architecture (MSA), WP3 Task3.3. In this context, ParTec worked on the extension of ParaStation MPI by support for BXI networks. This does not only enable efficient communication over BXI using the ParaStation communication stack but also benefits from MSA-aware features such as gateway communication. Both contributions show promising results that pave the way for upcoming studies and optimisations.

1 Introduction

The MPI standard takes an important role in the network software stack revolving around WP 4 of the RED-SEA project. Existing implementations act as a middleware abstracting complex network architectures to send messages between processes in an efficient and scalable manner. In particular, this document presents how the MPI implementations MPC and ParaStation MPI developed by CEA and ParTec respectively, support a highly scalable network interconnect such as the BXI.

In the first part of this deliverable, CEA presents the development of the multirail feature in its MPI implementation MPC. This gives the ability to take advantage of recent network architectures composed of multiple NICs per computing node—a feature particular important when applications put high demands on the network throughput. Also, based on state-of-the-art MPI implementation, CEA explains modification on the current software architecture to overcome potential performance issues. This first part finally presents preliminary performance results leveraging the multirail capabilities.

In the second part, ParTec presents the design and implementation of BXI support in ParaStation MPI. This is an MSA-enabled implementation of the MPI standard. After providing a brief introduction to the software architecture of ParaStation MPI, the section provides a comprehensive presentation of the *pscom4portals* plugin efficiently integrating BXI support into pscom—the low-level communication layer of ParaStation MPI. This approach not only enables efficient communication via the BXI interconnect in ParaStation MPI, but also supports the integration of BXI into MSA systems exhibiting a heterogeneous landscape of interconnects by relying on pscom's bridging capabilities [23]. This section concludes with the presentation of a preliminary performance evaluation on two RED-SEA testbeds: (1) the Dibona cluster being based on the ARM platform; and (2) the modular DEEP System at Jülich Supercomputing Centre (JSC).

Overall, this deliverable regroups contributions from both CEA and ParTec and aims to give a comprehensive overview of the work done since the beginning of the project.

2 LowComm Protocol: a new MPC layer for fragmentation and more

2.1 Introduction

Nowadays, HPC applications scale to several hundred nodes, and MPI has become the *de facto* standard for internode communication. It abstracts Point-to-point (P2P) communication to the application developer while taking advantage of the underlying network. More particularly, it abstracts the case where nodes possess multiple Network Interface Cards (NIC) in a feature called *multirail*. This feature also allows splitting large messages on parallel communication channels in a process called *striping* (*fragmentation* ?). A similar process called *fragmentation* exists in lower-level communication protocols where physical limitations limit the fragment size sent on the wire (MTU). The subject of this study is the implementation of such mechanisms in the MPC platform (Multi-Processor Computing).

These different features are the reasons that led state-of-the-art MPI Communication Modules (CM) to the definition of a two-layered architecture with a *protocol* and a *transport* layer. The concept of layer helps to provide as much abstraction and modularity as possible. It also delimits perimeters unambiguously between them to avoid redundancy. The transport layer interfaces lower-level driver's API such as InfiniBand, TCP, or Portals 4 for P2P communication while the protocol layer implements the MPI specifications and protocols. There exist two main protocols: eager which is more latency-oriented and rendez-vous which is bandwidth-oriented. For the latter, modern MPI implementations use *zero-copy* mechanisms using RDMA [24, 13] or tag-matching interface [4] over InfiniBand were proposed to copy data from the initiator's buffer straight to the target's buffer, thus by-passing kernel or intermediary copies. Two of the CM of state-of-the-art MPI implementations that will be discussed here (OpenMPI PML [9] and UCX [22]) features these technics as well as multi-rail.

The MPC platform (Multi-Processor Computing) is the CEA implementation of the MPI standard [17]. Similarly, its CM could be modelized as a two-layered structure with a similar lower layer called *rail* that abstracts the variety of networks while the upper layer handles other MPI features such as tag-matching, message ordering... The main difference is that the transport layer is responsible for protocols. Consequently, developing the multi-rail with the current rail (or transport) API would possibly incur the application of a rendez-vous protocol for each fragment sent. Moreover, protocols can usually be applied undifferentiable of the underlying transport, so replicating them for each transport is counterproductive. In the context of the RED-SEA project, we expose our implementation of the multi-rail feature with architectural changes to the MPC communication module inspired by other MPI distributions. In particular, we target these developments on BXI networks.

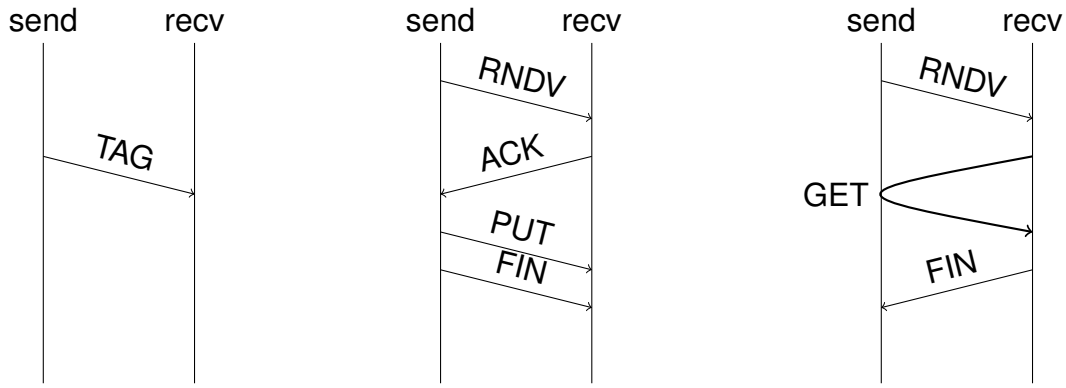
This report has been divided into four parts. Firstly, we discuss on protocols and their implementation in OpenMPI and UCX. Then we analyze the architecture of MPC and expose its limitations in terms of architecture and implementation. Finally, we present our contributions to implementing the multi-rail feature on the BXI network.

2.2 Discussions on protocols and their implementation

The protocol and transport layers form the base of many MPI implementation. We first give some background on the different types of protocols available. Then we focus on the implementation of two state-of-the-art MPI Communication Modules (CM): OpenMPI PML and UCX.

2.2.1 From the eager to the fragmentation protocols

MPI is the *de facto* standard for communication in high-performance environments. It provides primitives for communication between several MPI processes. Those primitives can be split into two kinds: the two-sided and the one-sided. The first communication model involves both the sender and the receiver, and synchronization is achieved implicitly through communication operations. In the second model, one process specifies all communication parameters, and synchronization is done explicitly to ensure the completion of communication. In the first model, the asynchronism between the two independent processes implies that a send may be initiated before a receive operation. Thus, application buffers intended to receive the data may not be allocated. Depending on the buffer size, MPI implementations solve this problem with different protocols that optimize important communication metrics such as latency or bandwidth. They also take advantage of underlying network technologies such as RDMA (Remote Direct Memory Access) that lets the NIC perform the data transfers without the involvement of the CPU [14, 8]. For example, InfiniBand provides primitives for one-sided and two-sided semantics.



(a) Eager (b) Rendez-vous Put (RPUT) (c) Rendez-vous Get (RGET)

Figure 1: Commonly known protocols. RPUT and RGET are specific for RDMA communication.

Usually, the transport of small messages ($size < 8kB$) is initiated directly, but the receiver may not have posted its request, so temporary buffers need to be available to store the data, which will then be copied to the actual application buffer when available. It is called the *eager protocol*, see Figure 1a. The eager protocol could also be implemented with RDMA over InfiniBand, thus avoiding copies [13]. However, in most cases, the memory footprint and copies become prohibitive for large messages. Thus synchronizations are needed beforehand to make sure application buffers are

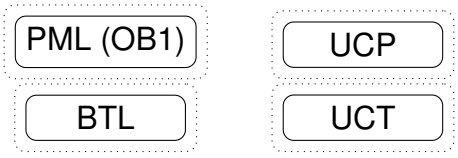
available. In that case, a handshake protocol between the two processes is realized before sending the actual data. It is called the *rendez-vous protocol*. Such protocols make *zero-copy* mechanisms possible by using RDMA operations to avoid intermediary copies and exchanging data for RDMA during the handshake. Figures 1b and 1c illustrates to possible zero-copy protocol: RPUT uses RDMA Write operations while RGET uses RDMA Read operations. It was shown that the latter improved overlapping of the communication and computation [24]. Additionally, specialized Network Interface Cards (NIC) offer similar zero-copy mechanisms while offloading tag-matching to deposit data straight to the application buffer. For example, Mellanox ConnectX-5 network architecture has introduced this feature [4] as well as BXI networks [7]. The latter will be of interest in this study. As said in [24], it is also important to note that the receiver may post a much larger buffer than what the sender chooses to send. As a consequence, the protocol choice relies on the sender passing this information through control messages as an example.

Finally, the *multirail* feature is also present in modern MPI distributions and has also been studied. For example, [12] describes a first use case called *multiplexing* which consists of sending messages successively on different transport available using dynamic scheduling. It also defines *striping* as the fragmentation of very large messages and their distribution on several transports. However, it is based on InfiniBand networks while we will be focusing here on the Portals 4 tag-matching interface. [21] reports another use case around reliability and fault-tolerance.

We now evaluate the architecture of the communication module in OpenMPI and UCX as well as their multi-rail design.

2.2.2 Software architecture of state-of-the-art MPI implementation

We propose here to give a brief overview of two widely used MPI implementations, OpenMPI and UCX, or more precisely, their communication module. We will first discuss their software architecture and then focus on the multi-rail.



(a) OpenMPI (b) UCX

Figure 2: Protocol and transport layers.

2.2.2.1 Protocol and transport layers

With the increasing complexity of software and hardware, most modern MPI implementations, and particularly Open MPI, are turned into component-based architecture [9]. This architecture helped tackle the vast heterogeneity of platforms and environments by giving more flexibility. In particular, OpenMPI and UCX can be divided into two layers: *transport and protocol layers*, see Figure 2. The transport interface provides primitives to handle point-to-point data delivery over the network. The interface accommodates the variety of network architecture,

and they provide sufficiently fine-grained communication API first to implement both two-sided and one-sided semantics and second to adapt to message size and MPI communication models: *buffered, synchronous, standard, and ready*. For example, UCT (Unified Communication Transport) interface from UCX segregate two-sided semantic based on message size with the `short`, `bcopy` and `zcopy` calls and also provide `put`, `get` calls for one-sided semantic. The same is available for the OpenMPI BTL (Bytes Transfert Layer) interface. `short` is used to send a few bytes of data that can be sent with very low latency, *i.e.* without any copies or sanity check. Then `bcopy` and `zcopy` differ in the sense that `bcopy` creates a buffered copy before sending data so that the application buffer can be released and reused by the application immediately after the call returns. On the contrary, `zcopy` sends the data without intermediary copies so that the buffer can only be released when the data has been fully sent. Thererfore, `bcopy` is more suited for latency-oriented communication, while `zcopy` is better for bandwidth-oriented communication. More detailed views of both OpenMPI PML and UCX are respectively described in [26] and [16].

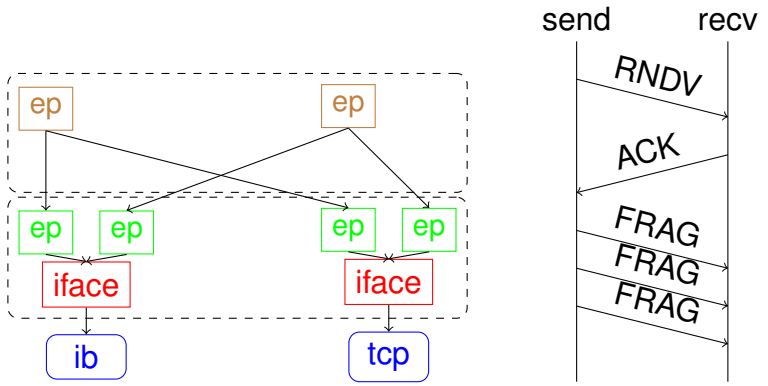
The transport API gives sufficient flexibility for the *protocol layer* to optimize the communication and enforce the specifications of the MPI standard. In OpenMPI and UCX, the Point-to-point Management Layer (PML) and the Unified Communication Protocol (UCP), see Figure 2, chose the optimal *data path* depending on multiple factors: network availability, message size,... More generally, the protocol layer is responsible for selecting transports for communication, message fragmentation, and multi-rail communication. In UCX, protocol selection is based on a piecewise linear model representing latency in the function of message size and is estimated before runtime. Moreover, they similarly integrate protocols such as rendez-vous in order to improve latency, bandwidth, and computation/communication overlap based on the Remote Direct Memory Access (RDMA).

We now give more details on their implementation of the multi-rail feature which relates to fragmentation.

2.2.2.2 Endpoints and multirail

To understand the multi-rail and how it has been implemented, it is essential to focus on what we call the transport and protocol endpoints.

While there can be some slight differences, Figure 3a represents the data structures in use during communication for UCX and OpenMPI PML. The interface (iface) exposes the communication primitives implemented for the underlying network (IB, TCP, Portals 4,...) and also collects its configuration information (eager size, rendez-vous threshold, max message size,...). During communication, the target is modelized in the transport layer by a *transport endpoint* that contains the connection information specific to the type of transport used (file descriptor for TCP, QP for Infiniband, nid/pid for Portals 4,...). The multiplicity of transports is abstracted by the *protocol endpoint*. It may have multiple transport endpoints, giving a coherent view of the underlying transport by "merging" their configurations. Indeed, transport heterogeneity can create complex combinations. For example, the RGET protocol cannot be used if one of the underlying transport does not support such operations, whereas it could be the case with a combination of InfiniBand and TCP.



(a) Multirail (b) Fragmentation

Figure 3: Multirail data structures and fragmentation algorithm.

In concordance to the previous modelization, multi-rail is just the fact of handling multiple transport endpoints simultaneously. Consequently, fragmentation is implemented in the context of multiple transports by distributing the application buffer to the different transport endpoints in a round-robin fashion. Fragment sizes may change depending on the type of transport. A request is considered completed when all data has been sent or received, and it is tracked either by piggybacking a unique id (UCX) or direct pointer (OpenMPI) in the request header. The protocol layer is responsible for the message reconstruction on the receiver side.

In most MPI implementations, *connection establishment* is done using out-of-band requests, usually on the backbone network through PMI requests. Indeed, this centralized server is needed for the processes to get the connection information of the MPI rank they are trying to communicate with. While OpenMPI performs one PMI request per transport endpoint, UCX recovers all transport information with only one request. These requests are also called *on-demand* requests, and they are called only once when communication is needed with a peer and are usually expensive.

The MPC implementation resembles these two layers' architecture with some fundamental differences that we will explain now.

2.3 MPC's communication module

Like its counterparts, MPC could be modeled as two-layered architecture. We first discuss this architectural design, how the workflow is articulated around it, and then the implementation design. Especially, we explain why it limited the implementation of fragmentation.

2.3.1 Principal designs and limitations

The design of the fragmentation protocol revealed some limitations of the current transport API that we propose to expose here.

First, MPC can be architecturally split into two layers. The upper layer, called Inter Thread Comm (ITC), is responsible for tag matching, connection establishment, and

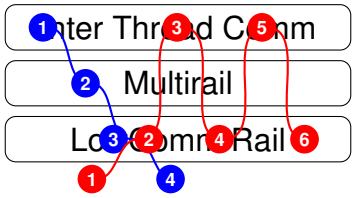


Figure 4: Message workflow: send and receive.

multi-rail. In Figure 4, the multi-rail has been shown explicitly but is part of the ITC. In the current implementation, multi-rail only elects the rail that sends data and initiates on-demand connections, but it cannot fragment the message if necessary. The lower layer, or Rail layer, abstracts different drivers implementation (IB, TCP, Portals 4) in the same manner as the transport layer previously described. The most significant difference with state-of-the-art MPI implementation is that it implements protocols (eager and rendez-vous) and initiates packing/unpacking.

Moreover, the data workflow is represented in Figure 4 for the two-sided semantic. For the send, (1) the ITC encapsulates user data in the lowcomm header, (2) select (or create) a communication endpoint (if it does not exist), and finally, (3) the rail passes the message to (4) the underlying transport. In receive workflow, (1) the rail receives the network data, and (2) the communication header is retrieved along with all tag-matching information. (3) It is moved up the ITC to perform tag-matching, and if matched, (4) the payload is copied to the application buffer by the rail. Indeed, each transport possesses its data layout, so each must implement its copy and free function that (5-6) are called by the ITC. The receive workflow is more complicated but avoids intermediary buffer copies. As a consequence, the MPC Lowcomm architecture cannot be modeled like others as pure transport and protocol layers, creating limitations.

The first limitation is related to the design of the transport API. Indeed, eager and rendez-vous protocols are implemented transparently by each rail through the `send_msg` call, see Listing 1, which additionally creates code replication. Building the fragmentation algorithm using this API would imply performing a rendez-vous for each fragment in case its size exceeds the rendez-vous threshold, while only one would be needed. For two-sided semantic, the current transport API is as follow:

```
/* ptp_msg_t contains header and data information */
/* transp_ep contains the transport endpoint information. */
void send_msg(ptp_msg_t *msg, transp_ep *ep);
/* ptp_msg_t contains header and data information */
/* transp_iface is the transport interface. Call is
   needed for tag-matching interface. */
void recv_msg(ptp_msg_t *msg, transp_rail *rail);
```

Listing 1: Current rail API for two-sided semantic

The second limitation is related to the implementation. MPC's lowcomm layer is centralized around one data structure `ptp_msg_t` attached to any message sent on the network ("one header to rule them all"). However, the design of other protocols such as fragmentation requires specific control messages with additional metadata that would extend the size of the current structure. This increase may not be profitable regarding network traffic, especially when not all the metadata in the header is needed. For

example, there are control messages that do not require tag information. Even more, the current design does not allow us to add more data to the lowcomm header, which forced us to send the metadata as user payload and thus to go through the received workflow described in Figure 6. Enforcing data encapsulation by extending the rail API with more flexible and generic calls would help the design of protocols.

2.3.2 Details on Portals 4 driver

We introduce some of the Portals 4 specifications and useful information on the driver implementation in MPC. We refer the reader to the Portals 4 documentation for more information [3].

Portals 4 seeks to provide a highly scalable, high-performance network programming interface while being sufficiently general to support different communication models (MPI, PGAS,...). In particular, it provides an adapted addressing semantic to easily support two-sided, and one-sided semantics through methods called `put` and `get`. Moreover, it is connectionless, unlike VIA or TCP/IP sockets, enabling network independence. That is, communication is progressed without requiring the host processor activity. Along with user-level and OS bypasses, Portals 4 enables zero-copy protocols through its tag-matching interface. Then, there are mechanisms to handle unexpected messages that avoid flow control and protocol overhead which is not always supported by other interconnects. Finally, Portals 4 are asynchronous and thus rely on the publication of events to inform the user of the request progress. Portals 4 specifications have also been implemented in MPI distributions [2, 20]

We now describe how the MPC driver implementation uses the unexpected message mechanism. A remote address is needed in a classic RDMA interface to send data to the correct location. As per the two-sided semantics in Portals 4, it is necessary for the sender and the receiver to post what we call memory descriptors to identify the memory region involved in the communication that is then stored in Portals 4 internal lists. Portals 4 natively provides an expected list called Priority List and an unexpected list called Overflow List, whose behavior does not differ from usual MPI lists called PRQ and UMQ lists. Along with memory information, the descriptor contains an unsigned 64 bits integer used for tag matching. Consequently, upon receiving a Portals 4 request and when the receive memory descriptor has been posted before, data will be deposited to the matched entry, directly into the application buffer, thus resulting in a zero-copy protocol. However, to handle the case where the corresponding memory descriptor is unavailable, the request is posted to the Overflow List that has been previously populated with temporary buffers at init time and continuously updated to ensure memory is always available to store unexpected message data. It is important to know that, although Portals 4 support a non-matching interface, the application buffer is always associated with a matching tag in the MPC driver. In other words, the matching is necessary in order to enable zero-copy mechanisms.

Finally, the driver currently supports eager, rendez-vous protocols but implements fragmentation. The latter is RGET-based, see Section 2.2.1. Also, because Portals 4 is event-oriented, the interface provides polling functions called by the MPC's progress engine. The messages are then progressed depending on the event generated. In

MPC, the progress engine is called numerous times thanks to the integrated MPC scheduler [17].

We now describe our new implementation of the protocol layer in MPC.

2.4 Implementation of MPC protocol layer

The following protocol layer has been designed to overcome some earlier limitations. To accommodate its implementation, we first extended the current transport API. We then implemented the multi-rail based on the connection management model described in Figure 3a. Moreover, we finally introduced the protocols through the paradigm of active messages and handlers.

2.4.1 Extending the transport API

The first step to implementing protocols was to extend the current rail API, see Listing 1.

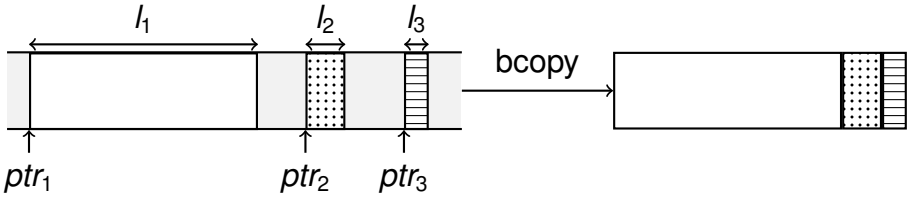


Figure 5: Encapsulation with buffered copy or zcopy with iovec (ptr_i and l_i).

Indeed, the implementation of protocols such as Rendez-vous requires control messages (RTS, RTR, ATS, ...) to synchronize and exchange necessary protocol data that are part of the header. This variety of control messages is not consistent with the current unmutable `ptp_msg_t` structure that we extended as follows:

```
/* packs with copy the header before user data using the callback
argument. */
/* id: message identifier for active message. */
/* transp_ep: transport endpoint. */
int send_bcopy(transp_ep *ep, int id, pack_callback_t cb, void *arg);
/* id: message identifier */
/* hdr and hdr_l: header for protocol data and header length*/
/* iovec: user data */
int send_zcopy(transp_ep *ep, int id, void *hdr, int hdr_l, struct
    iovec *iov);
/* tag: tag for tag-matching */
/* cb: callback to pack user of header data */
/* ctx: context passed to Portals\,4 for request tracking */
int send_tag_bcopy(transp_ep *ep, uint64_t tag, int id,
    pack_callback_t cb, void *arg, context_t *ctx);
int send_tag_zcopy(transp_ep *ep, uint64_t tag, int id, void *hdr,
    int hdr_l, struct iovec *iov, context_t *ctx);
/* ign_tag: ignored tag, support for MPI wildcards */
```

```
int recv_tag_zcopy(transp_ep *ep, uint64_t tag, uint64_t ign_tag,
    struct iovec *iov, context_t *ctx);
```

Listing 2: Extension of the transport API

The differences are two ways: first, weakly-typed arguments enable arbitrary headers, and second, we give finer granularity to adapt to message size. Indeed, the most significant difference relies on the fact that with `send_bcopy`, the user buffer can be relaxed earlier, thus speeding up request completion. Buffered copy is more suited for small data, whereas zero-copy is more suited for large data. The API has been only implemented for Portals 4 and TCP, support for InfiniBand will come later (we exclude tag-based calls).

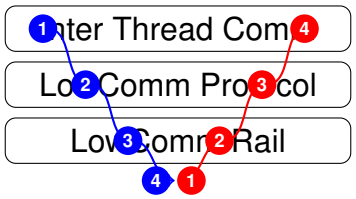


Figure 6: Message workflow: send and receive.

Additionally, we enforced data encapsulation between layers to help simplify the data workflows as presented in Figure 6. The transport layer mainly performs this encapsulation, but in principle, going down the layers, it places a header before the payload. Going up the layers removes the headers, passing only the payload to the upper layer. The header is positioned either through copy (when message size is not too large) or iovec are used, see new rail API.

2.4.2 Implementing eager and rendez-vous protocols

We exposed in Section 2.3.1 the drawbacks of implementing protocols in the transport layer. They have been moved now to the protocol layers using the active message and handler paradigm. First, we mention that connection management has been implemented based on the model represented by Figure 3a. To satisfy a modular, object-oriented approach, a protocol endpoint exists in the protocol layer and can handle multiple transport endpoints from the transport layer. When multiple transport endpoints are available, messages are scheduled based on a globally shared rail priority to ensure coherency when tag-matching interfaces (or rails) such as Portals 4 are used. Upon on-demand, all transport endpoints are connected.

As described earlier, messages can be sent either eagerly without any synchronization for small messages or through a rendez-vous protocol to accommodate large messages. In our implementation, we use the RPUT rendez-vous since we wanted support for heterogeneous transport and TCP does not support `get` operation. The different message types are handled using the active message paradigm, which consists in sending a message with a particular `id` upon which the receiver will call a particular handler. This `id` is specified in the function signature described earlier. The control messages involved are listed below:

- `TAG`: regular header with tag information, used with eager protocol.
- `RTS` (Ready To Send): `TAG` header extended with necessary rendez-vous information.
- `RTR` (Ready To Receive): regular ack message.
- `FRAG`: fragment message.

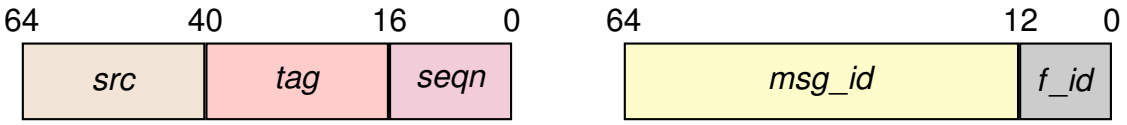
As a first implementation, even though underlying transports support Put/Get semantic, we used a Write-based rendez-vous. In future work, implementing Read-based rendez-vous, when possible, could limit the number of control messages and improve computation to communication overlap [24].

The eager protocol does not require progression monitoring since request completion is immediately after sending the message. However, it is required for rendez-vous and fragmentation to track the request until its completion, when all data has been sent. Each request is assigned a message identifier (a randomly generated integer) and stored in a hash table. It is generated by the initiator of the communication and piggy-backed onto control messages so that the receiver can assign the id to the matched request when the RTS control message is received or retrieved and progress when it is a FRAG message. During fragmentation, fragments are scheduled in a round-robin fashion on the available transport endpoints owned by the protocol endpoint. Protocol endpoints support heterogeneous interfaces, so tag-matching interfaces (Portals 4) can be used in conjunction with standard interfaces (TCP). Better protocol endpoint configuration should be performed to identify the best protocol to be used (RPUT or RGET).

In the next section, we explain how we used the zero-copy mechanism proposed by Portals 4 in the context of very large messages with fragmentation.

2.4.3 Preserving Portals 4 zero-copy during fragmentation

MPC Portals 4 driver based its implementation on the Portals 4 tag matching feature, which provides optimized features such as tag offloading and zero-copy mechanism for large messages. We explain our implementation to preserve these features in the context of fragmentation and multi-rail.



(a) `TAG` with MPI matching information

(b) `FRAG` with message and fragment identifiers

Figure 7: Structure of the Portals 4 matching bits.

During the fragmentation algorithm, first a `RTS` message with a `TAG` matching bits, cf Figure 7a, is initiated. The communicator is not present in the tag, but the information is retrieved using specific Portals 4 structure. *seqn* is the sequence number and is used

for message ordering. This control message does not contain user data but is used to match the receive request and synchronize to ensure receive buffers are available. As said in Section 2.2.1, the receiver cannot know in advance the protocol that the sender will initiate, but it should be known as soon as possible to perform the required action. To do so, each Portals 4 request contains a 64 bits field made available directly when the event is polled and thus filled with the correct `id` to call the corresponding handler in the context of the active message paradigm. The metadata in the `RTS` message contains the `msg_id` assigned to the request, and that will be used along with the `frag_id` to build the tag of the Portals 4 memory descriptors of all the fragments, see Figure 7b. The fragment identifier, `f_id`, is incremented for each fragment. Once all memory descriptors have been posted, receiver sends the `RTR` message with the same `msg_id` upon which the sender starts sending the fragments. Therefore, the synchronization ensures fragments will be matched in the Priority List, and zero-copies will be enabled.

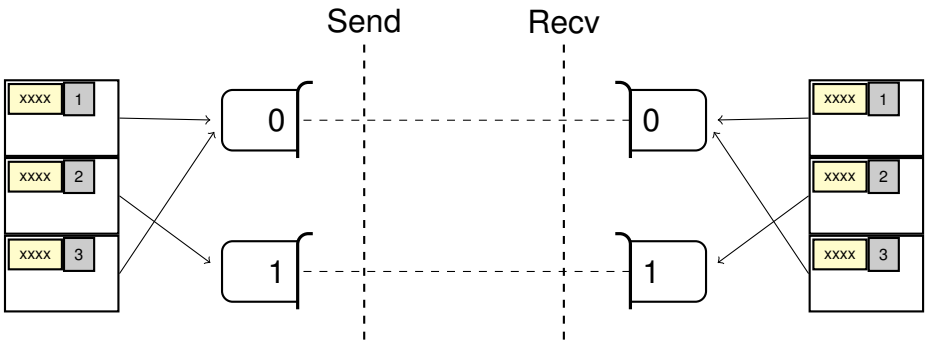


Figure 8: Matching fragments with Portals 4.

Figure 8 illustrates this mechanism. In addition, we need to make sure that all fragments will be matched. In other words, to make sure that fragment of tag X is sent to the interface on which fragment of the tag X has been posted. To do so, a one-to-one relationship has been created between Portals 4 NICs. Each NIC is associated with an ID at init time for each process, and NIC number i can only communicate with remote NIC i. Then, the first fragment is posted to the lowest NIC ID, and the followings are scheduled in a round-robin fashion. In future works, it could be interesting to quantify the overhead of tag-matching compared to multi-rail RDMA, where only the remote address is needed to perform the transfer.

We now present validation results for the multi-rail feature.

2.4.4 Validating the multirail feature

Validation benchmarks have been run on machines provided by the CEA Inti supercomputer, which features multi-NIC nodes with the BXI network. They are composed of bi-socket AMD Milan (2×64 cores) with 256GB of RAM, and each node possesses $4 * 100 GB/s$ BXI network cards.

For our performance analysis, we ran Pingpong from IMB benchmarks for message sizes ranging from 4Bi to 4096MBi. In the experiment, fragment size has been set to 64MBi which is the maximum message size allowed by the Portals 4 library on this

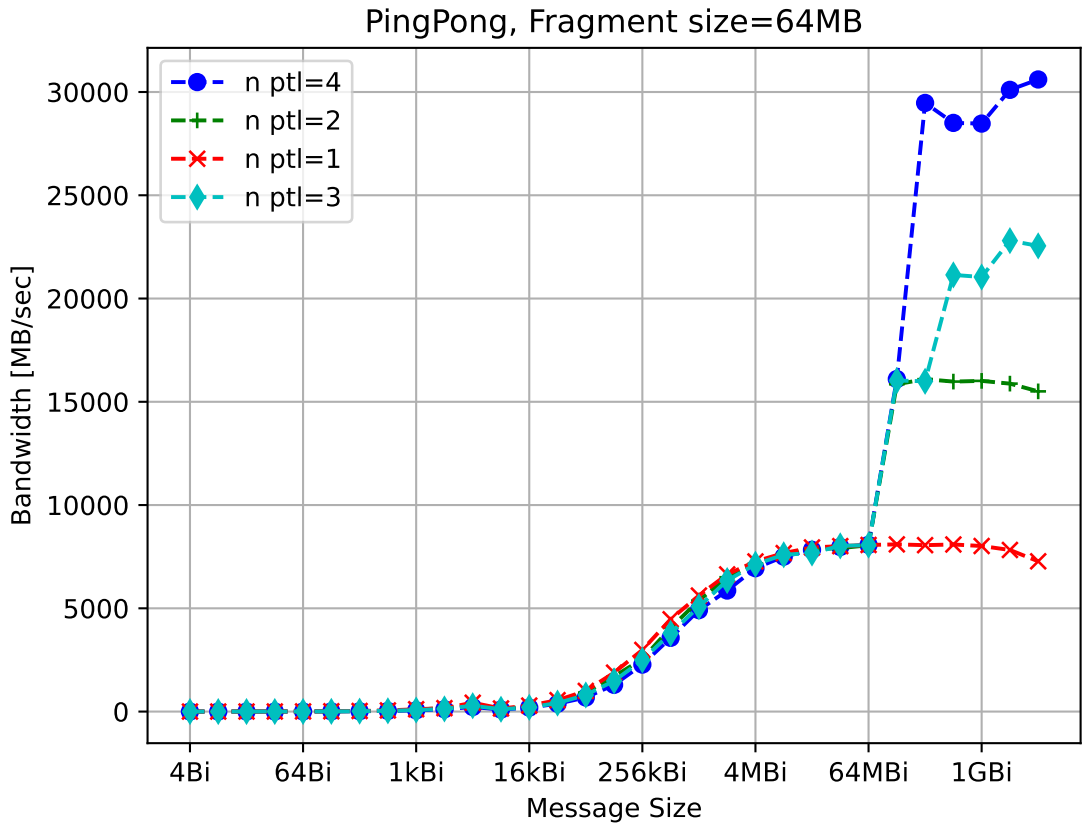


Figure 9: Average bandwidth for 1000 iterations on the IMB PingPong benchmark with increasing number of BXI network cards.

network. Figure 9 acts as a Proof of Concept of the multirail feature and a more careful analysis will be performed.

Messages ranging from 4Bi to 64MBi have been showned to better appreciate the outcome of the multirail feature for sufficiently large messages. Additionnally, we show that it introduces a limited overhead. To complete the study on this message range, finer comparisons will have to be undertaken with state of the art MPI implementations, more especially to evaluate the performance of the eager and rendez-vous protocols.

For messages ranging between 64MBi to 4GBi, the implementation shows coherent results with good scalability. With 4 NICs, bandwidth reached more than 30 000MBi/s which make relevant the use of multirail feature in this context. However, a decrease appears for very large messages ($\geq$ 2GBi) and for 1 and 2 NICs which may be improved by an optimized use of Portals 4 memory descriptors. For now, each fragment has is own memory descriptor while only one would suffice accompanied with offset management. This will be studied for the next deliverable. Moreover, focusing on 256MBi message size, one can notice that 3 NICs are as fast as 2 NICs. Indeed, since all fragments are of equal sizes, the message will be split in 4 fragments which will be sent for both configuration in two round (2 + 2 for dual-NIC and 3 + 1 for 3-NICs). Data distribution could also be improved in future work.

2.5 Conclusion

To conclude, we exposed our principal designs for implementing the multi-rail feature on MPC while comparing what has been done in other state-of-the-art MPI implementations. We first extended the existing transport API with support for active message paradigm to alleviate some limitations of the current one. We implemented a simplified protocol layer with sufficient handlers for eager and rendez-vous protocols. Finally, we validated the feature with the IMB PingPong benchmark showing decent speedup with an efficiency of more than 85% for large message sizes but the study will have to be completed.

In future work, latency for large message sizes should be improved by using RGET protocol whenever possible. Thoughtful code instrumentation will give more insight into the current bottlenecks. Furthermore, implementing this transport API for InfiniBand could be useful.

3 Extending ParaStation MPI by BXI Support

This section presents the work of adding support for the BXI network to the ParaStation MPI library. After a short introduction to its software architecture in Section 3.1, the design and the implementation of the BXI support is presented in Section 3.2.

3.1 An Introduction to ParaStation MPI

The ParaStation MPI communication stack is a central pillar of ParaStation Modulo. This is a comprehensive software suite especially designed for MSA systems. It is the selected middleware powering the DEEP projects but is also extensively used in production environments, e. g., on the JUWELS Cluster/Booster system at JSC [1] and the modular MeluXina system in Luxembourg.

ParaStation MPI is an MPICH [10] derivate relying on the pscom library [19] for point-to-point communication. It is used to implement the PSP device (cf. Fig. 10b) which, in turn, implements the ADI3 interface. ADI3 separates the hardware-independent communication facilities (e. g., datatype handling, collective communication, etc.) from the hardware-related counterpart (e. g., transport-specific Remote Memory Access (RMA) operations), as indicated in Figure 10a. It is a rich interface with over 80 function prototypes that have to be implemented by the so-called *devices*. Although MPICH already ships with an exemplary implementation of this interface by means of the CH4 device, a distinct device comes with several advantages. On the one hand, the PSP device is a thin software layer since its main purpose is the integration of the point-to-point communication logic provided by the pscom into the MPICH software stack. On the other hand, further optimisations can be more easily integrated due to fewer dependencies to the lower layers, i. e., instead of relying on multiple, different low-level communication layers as this is the case for the CH4 device, the PSP device and the pscom are closely aligned with each other. Following this concept, the PSP device implements MSA awareness in ParaStation MPI, e. g., by providing multi-level, hierarchy-aware collectives taking the topology of MSA systems into account.

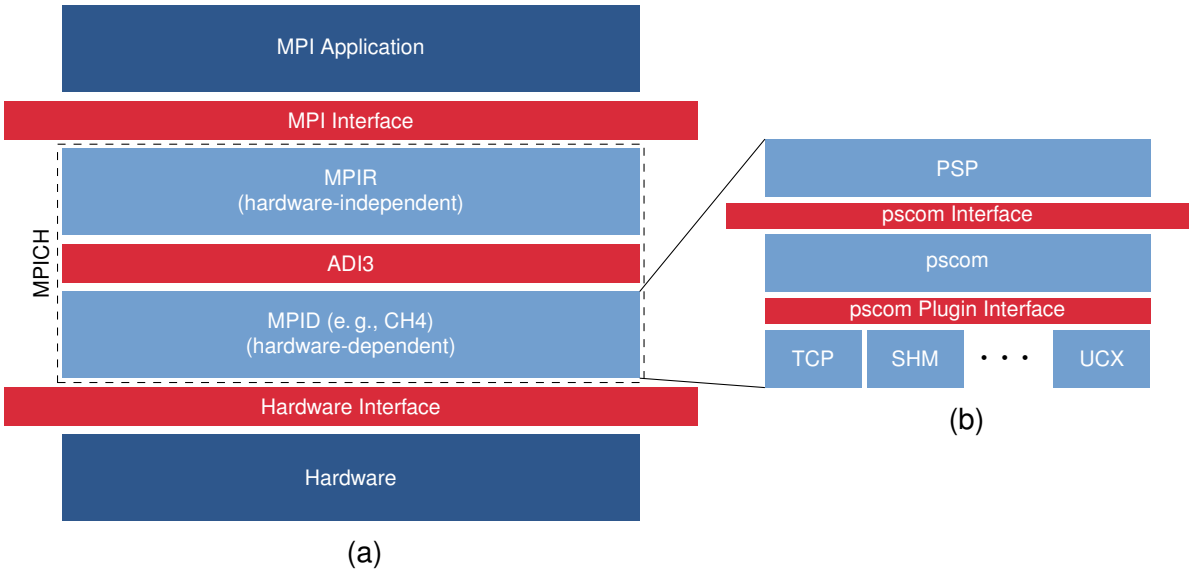


Figure 10: The architecture of ParaStation MPI: The pscom is used to implement the psp layer (b) for an integration into the general MPICH software stack (a) as an ADI3 device. (based on [18])

The pscom is a low-level communication library and is especially designed for High Performance Computing (HPC) systems. It targets high-performance communication and ships with a variety of plugins supporting different interconnects and interfaces relevant to the HPC domain, e. g., InfiniBand (IB) [11], UCX [25], Extoll [15], and Omni-Path [5]. These transports and protocols can be used concurrently and transparently by the processes of a parallel application, i. e., processes residing on the same computing node leverage a plugin dedicated to efficient shared-memory communication while inter-process communication relies on plugins optimised for the respective interconnect.

The pscom itself is divided into different software layers (cf. Fig. 10b): The hardware-independent layer facilitates the session management by enabling the establishment of bi-directional connections between different pscom processes. The hardware-dependent layer features a modular design that supports different communication interfaces and protocols by means of *plugins*.

The session management of the pscom behaves similar to the Berkeley Socket API and distinguishes the active *connecting* process and the passive *listening* process. Initially, both processes set up a Transmission Control Protocol and the Internet Protocol (TCP/IP) connection that is used for negotiating the actual transport by choosing the appropriate communication plugin. In accordance with this session management model, each connection is established explicitly and asymmetrically by means of the `listen` and `connect` calls offered by the pscom Application Programming Interface (API). This way, a fully connected graph among all processes in the initial `MPI_COMM_WORLD` group of an MPI session is created. This simple approach may result in an excessive waste of resources, i. e., $n \cdot \frac{n-1}{2}$ possible connections right upon session startup, although commonly only a fraction of these connections is required during a communication session. Therefore, the pscom provides an *on-demand* mechanism to overcome this waste of resources for huge MPI sessions comprising thousands of ranks. This is achieved by implementing

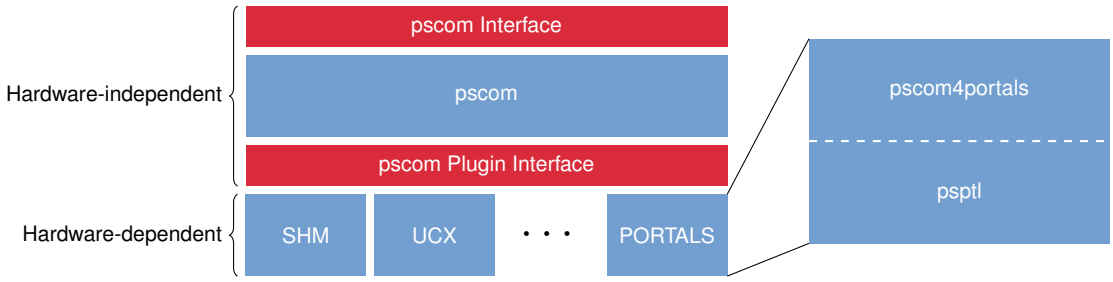


Figure 11: The software layout of the pscom4portals plugin and how it integrates into the layered architecture of the pscom library.

a lazy connect approach: the connection setup is postponed and only triggered on occurrence of the first send request.

3.2 Design and Implementation of the pscom4portals Plugin

Enabling efficient BXI communication in modular systems is the main goal of Task (Tk) 4.5 with respect to ParaStation MPI. As discussed in the previous section, ParaStation MPI abstracts from the hardware by relying on the facilities provided by the pscom library. Therefore, the natural way for adding support for BXI to the ParaStation MPI communication stack is to add an appropriate plugin to the pscom.

This approach not only enables MPI communication over the BXI network but also allows applications to run on heterogeneous network landscapes supported by the gateway facility of pscom [23]. Additionally, this aspect creates a direct link to the DEEP-SEA project by providing the ability to use BXI-based computing modules with MSA systems.

3.2.1 Architecture

The *pscom4portals* plugin relies on the Portals 4 API [6] for interfacing with the BXI hardware. Similar to other pscom plugin, this plugin is logically divided into two layers (cf. Fig. 11): (1) the upper layer implementing the interface to the hardware-independent part of the pscom and the lower *psptl* implementing a point-to-point communication channel by leveraging the Portals 4 API.

The main purpose of the upper layer is to implement the handshake mechanism of the pscom. Additionally, this layer provides the necessary callbacks for sending and receiving data over pscom4portals connections. During the handshake procedure, the pscom4portals negotiates and exchanges the required endpoint information for communication between two peer processes. This is basically the Node ID (NID) and the Process ID (PID) provided by the Portals 4 layer, i. e., message matching is done based on these two identifiers in the first place.

3.2.2 Communication Protocols

Depending on the message size, the plugin uses either *eager* or *rendezvous* communication. For the latter, it implements an RPUT protocol relying on Remote Direct Memory

Variable	Description
PSP_PORTALS_MAX_RNDV_REQS	Maximum number of outstanding rendezvous requests per communication pair
PSP_PORTALS_RNDV_FRAGMENT_SIZE	Maximum size of the fragments being sent during communication
PSP_PORTALS_RENDEZVOUS	The rendezvous threshold

Table 1: Important environment variables to influence the runtime behavior of the rendezvous protocol of the pscom4portals plugin.

Variable	Description
PSP_PORTALS_BUFFER_SIZE	The size of the buffers in the send/recv queues
PSP_PORTALS_SENDQ_SIZE	Number of send buffers per communication pair
PSP_PORTALS_RECVQ_SIZE	Number of receive buffers per communication pair

Table 2: Important environment variables to influence the runtime behavior of the eager protocol of the pscom4portals plugin.

Access (RDMA) write operations (cf. Fig. 1b). Both communication paths are separated by allocating dedicated Portals Table Entries (PTEs) each, i. e., each process initialising the pscom4portals plugin allocates two PTEs in total. The threshold when to switch the protocols can be defined by setting the PSP_PORTALS_RENDEZVOUS environment variable (cf. Tab. 1).

Eager Communication leverages pre-allocated send and receive buffers for the data transfers. These are allocated during the creation of a pscom4portals connection and exclusively used between a given pair of processes. The user may influence the number and size of these buffers by setting the according environment variables (cf. Tab. 2). On the sending side, the next available buffer is used to temporarily store the user data and pscom-internal headers. This is then transferred to the destination by triggering a put operation. The PTL_EVENT_ACK issued by the Portals 4 layer indicates whether this transmission was successful or not and the plugin simply resends the buffer in case of message drops on the receiving side, otherwise the buffer is marked as *free*.

Data arrival at the receiving process is indicated by the PTL_EVENT_PUT. However, as the sender might transfer the buffers out-of-order (due to the resend approach), we have to keep track of sequence numbers which are encoded in the header field of the Portals 4 event. If the sequence number matches that at the receiving process, the data can be directly processed. Otherwise, incoming buffers have to be queued until a buffer with the expected sequence number arrives.

Rendezvous Communication As mentioned above, the pscom4portals Plugin implements an RPUT protocol for rendezvous communication. The control logic synchronising sender and receiver is already provided by the hardware-independent part

	Dibona	DEEP System
Architecture	TSMC ThunderX2 99xx ARMv8.1	Intel(R) Xeon(R) Gold 5122
Node Count	12	4
Socket Count	2	1
Memory per Node	48 GiB	48 GiB
Interconnect	BXI1.3	BXI 1.3

Table 3: The testbeds being used for the evaluation of the BXI support in ParaStation MPI.

of the pscom. The pscom4portals has to provide callbacks for the registration and de-registration of memory regions as well as for issuing the actual put operation.

The psptl layer of the pscom4portals plugin supports the fragmentation of a single rendezvous message into smaller chunks as the network hardware might impose a limit to the maximum message size that can be transferred in a single call to `PtlPut()`. The default fragment size matches the limit imposed by the hardware. However, further performance tuning of the rendezvous protocol is possible by setting the `PSP_PORTALS_RNDV_FRAGMENT_SIZE` environment variable. In contrast to eager communication, the rendezvous implementation does not generate `PTL_EVENT_PUT` at the receiver. Instead, it waits for the `PTL_EVENT_ACK` of the last fragment and triggers the plugin-independent control flow provided by the pscom.

To avoid mismatches of multiple rendezvous transfers between the same pair of processes, the pscom4portals plugin leverages the matching capabilities provided by Portals 4. Therefore, each rendezvous transfer has a unique identifier being used in the `match_bits` field of the Matching List Entry (ME). This way, multiple rendezvous transfers can be processed concurrently without interfering each other.

3.3 Evaluation

For a preliminary evaluation of the BXI support in ParaStation MPI we used two hardware testbeds: (1) the Dibona system and (2) the DEEP system. This approach enables the assessment of the functionality and performance on two very different platforms. While the Dibona system is based on the ARM architecture, the DEEP system is a supercomputing system built by following the MSA approach. Further details of both systems can be found in Table 3.

Figures 12 and 13 present the communication throughput obtained with both ParaStation MPI and Open MPI on the DEEP system and the Dibona cluster respectively. The numbers have been obtained by running the OSU bandwidth test in the default configuration for 100 iterations per message size and a warm-up phase of 10 iterations respectively to avoid side effects caused by cold caches. ParaStation MPI is reaching a comparable throughput for large size messages. This leads to the conclusion that the implemented rendezvous protocol based on RMA write operations already yields a decent performance. However, small to midsize messages suffer from performance penalties as compared with Open MPI which is also reflected by the latency results shown in Figures 14 and 15. The latencies were obtained by running the latency test of

the OSU benchmark suite for 10 000 iterations per message size and a warm-up phase of 100 iterations.

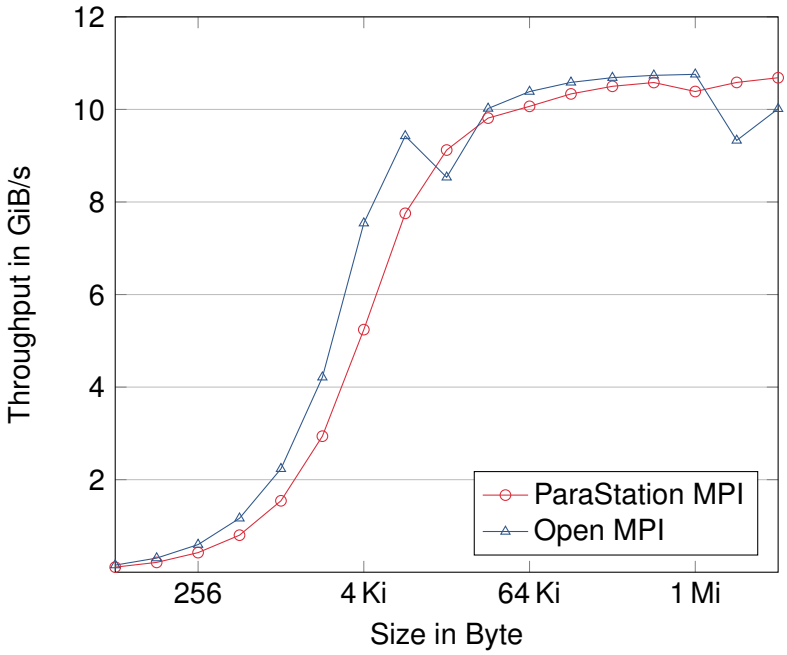


Figure 12: The BXI throughput of ParaStation MPI compared to Open MPI on the DEEP system.

A reason might be the additional copy operations on both the sender and receiver side. However, further investigation is necessary to identify the root-cause and to apply according optimisations. Additionally, further tweaking of the runtime parameters to the respective platform may yield additional performance improvements.

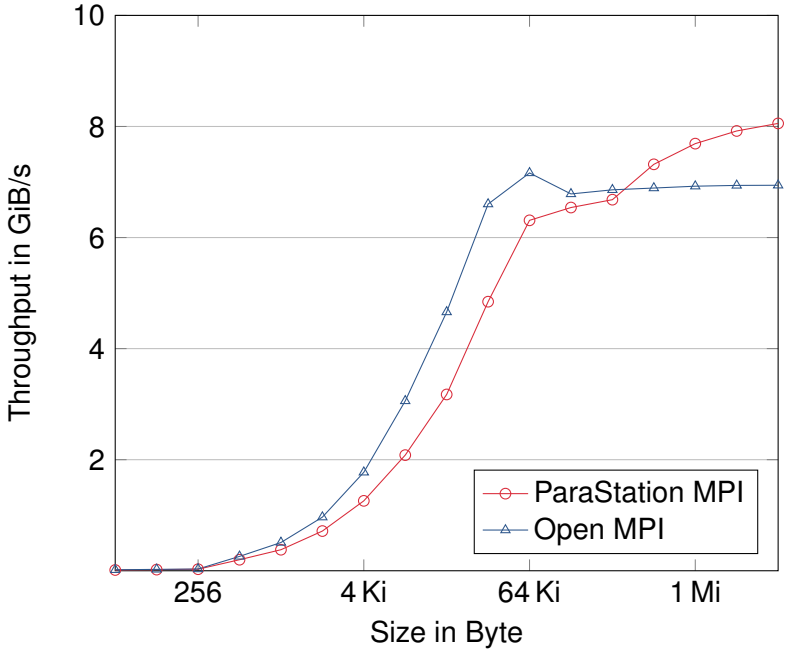


Figure 13: the BXI throughput of ParaStation MPI compared to Open MPI on the Dibona cluster.

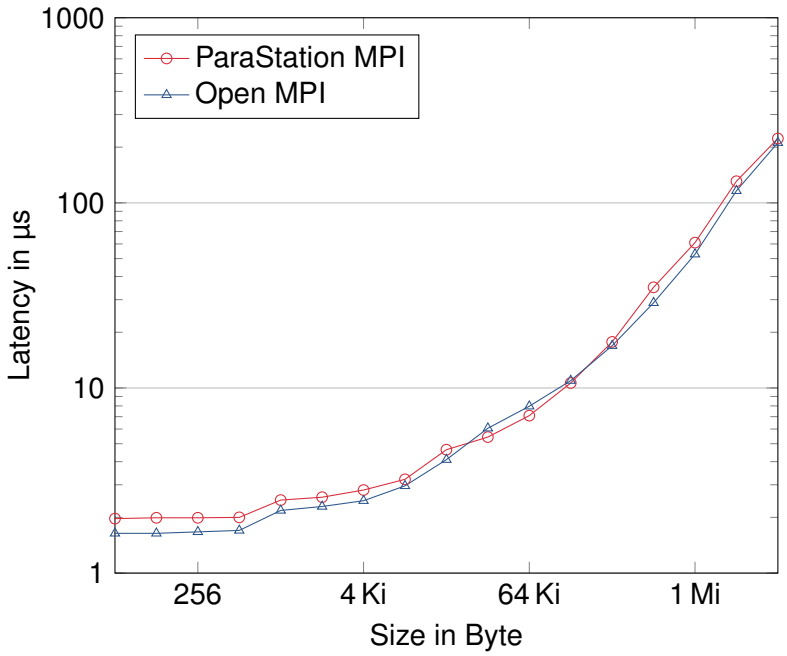


Figure 14: Latency analysis of ParaStation MPI and Open MPI over BXI the DEEP system.

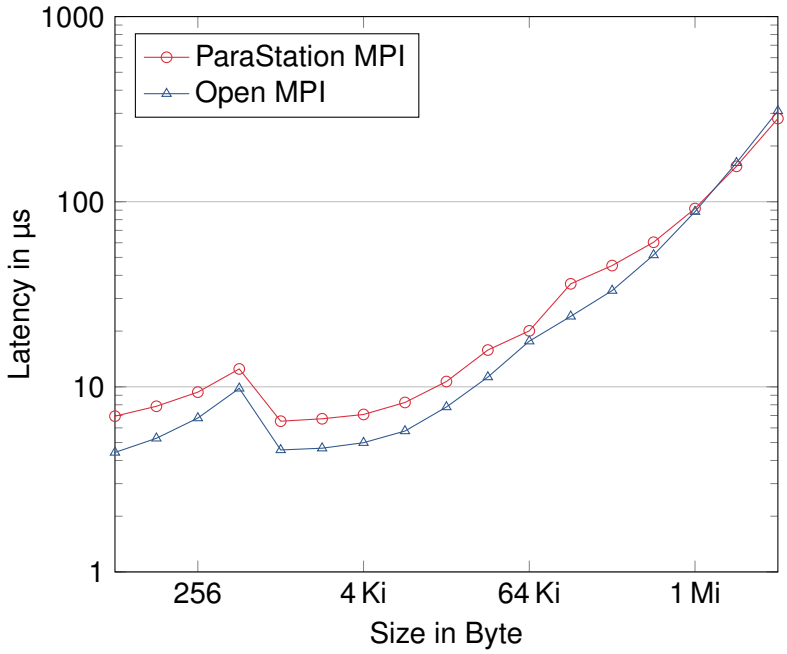


Figure 15: Latency analysis of ParaStation MPI and Open MPI over BXI the Dibona cluster.

4 Conclusion

MPI implementations have a pivotal role in networks stacks of current and next supercomputers in that they are a building block in the efficient exploitation of state-of-the-art network architectures such as the BXI interconnect. This deliverable describes how MPC from CEA and ParaStation MPI from ParTec took on this challenge. The first by including multirail support and the second by adding support for the BXI interconnect through the implementation of an additional plugin to the low-level communication layer pscom of ParaStation MPI.

Indeed, CEA showed the implementation details of the multirail feature to efficiently perform point-to-point communication for large messages in MPC. The architectural changes made to more accurately differentiate the protocol and transport layers enabled the efficient implementation of these communication protocols and to preserve the capabilities of the BXI network. Preliminary results show that efficient speedups can be obtained by the multiplication of NICs. While already present in most state-of-the-art MPI implementation, support for message fragmentation could lay the ground for new optimisations in the future.

Moreover, ParTec presents their efficient implementation of BXI support in ParaStation MPI by means of the newly developed *pscom4portals* plugin of pscom. By supporting both eager and the rendezvous communications semantics, the plugin provides two different protocols optimised for latency and throughput respectively. The preliminary evaluation results show that the performance of ParaStation MPI is aligned with Open MPI for large message sizes. The communication latency of small-sized messages still leaves room for improvements. According developments will be the objective of the next studies.

Finally, the document presented the architectural decisions taken by both partners to optimise their MPI implementation of BXI networks and whose prototypes act now as a proof-of-concept for the remaining part of the RED-SEA project. The next actions will be then to improve and stabilise the current designs.

Acronyms and abbreviations

A

API *(Application Programming Interface)* 21, 22, 29
ARM Family of Reduced Instruction Set Computing (RISC) architectures for computer processors, configured for various environments. Formerly standing for Advanced RISC Machine, or Acorn RISC Machine 7, 24

B

BXI *(BullSequana eXascale Interconnect)* 6, 7, 20, 22, 24–28

C

CEA *(Commissariat à l'énergie atomique et aux énergies alternatives)* Commissariat à l'énergie atomique et aux énergies alternatives, France 6, 7, 28

D

DEEP-SEA DEEP – Software for Exascale Architectures 6, 22

H

HPC *(High Performance Computing)* 21, 29

I

IB *(InfiniBand)* A networking communication standard for HPC clusters 21

J

JSC *(Jülich Supercomputing Centre)* Jülich Supercomputing Centre GmbH, Jülich, Germany 7, 20

M

ME *(Matching List Entry)* 24
MPC *(Multi-Processor Computing)* The MPC framework regroups an MPI, an Open Multi-Processing (OpenMP), and a pthread implementation in the same software, for better interoperability 6, 7, 28
MPI *(Message-Passing Interface)* An API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages 6, 7, 20–22, 24, 28–30
MSA *(Modular Supercomputer Architecture)* 6, 7, 20, 22, 24

N

NIC *(Network Interface Card)* 6, 7, 28
NID *(Node ID)* 22

O

OpenMP *(Open Multi-Processing)* Application programming interface that support multi-platform shared memory multiprocessing 29
Open MPI An MPI implementation maintained by the Open MPI Project 24, 28

P

ParaStation Software for cluster management and control developed by ParTec 6, 7, 20, 22, 24, 28, 30
ParTec ParTec AG, Munich, Germany. 6, 7, 28, 30
PID *(Process ID)* 22
pscom The low-level communication layer of ParaStation MPI 7, 20–22, 24, 28
PTE *(Portals Table Entry)* 23

R

RDMA *(Remote Direct Memory Access)* 22
RED-SEA Network Solution for Exascale Architecturesfor Exascale Architectures Project 7, 28, 30
RISC *(Reduced Instruction Set Computing)* 29
RMA *(Remote Memory Access)* 20, 24

T

TCP/IP *(Transmission Control Protocol and the Internet Protocol)* 21
Tk *(Task)* Followed by a number, term to designate a Task inside a Work Package of the RED-SEA project 22

References

[1] Damian Alvarez. “JUWELS Cluster and Booster: Exascale Pathfinder with Modular Supercomputing Architecture at Juelich Supercomputing Centre”. In: *Journal of large-scale research facilities* 7 (2021), A183. DOI: 10.17815/jlsrf-7-183.

[2] Brian W. Barrett, Ron Brightwell, and Keith D. Underwood. “A Low Impact Flow Control Implementation for Offload Communication Interfaces”. In: *Recent Advances in the Message Passing Interface*. Ed. by Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 27–36. ISBN: 978-3-642-33518-1.

[3] Brian W. Barrett et al. *The Portals 4.0.2 Networking Programming Interface*. Tech. rep. Nov. 2014. DOI: 10.2172/1561686. URL: https://www.osti.gov/biblio/1561686.

[4] Mohammadreza Bayatpour et al. “Design and Characterization of InfiniBand Hardware Tag Matching in MPI”. In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 2020, pp. 101–110. DOI: 10.1109/CCGrid49817.2020.00-83.

[5] M. S. Birrittella et al. “Intel Omni-path architecture: Enabling scalable, high performance fabrics”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. Aug. 2015, pp. 1–9. DOI: 10.1109/HOTI.2015.22.

[6] Ronald Brightwell et al. “The Portals 4.3 Network Programming Interface”. In: (June 2022). DOI: 10.2172/1875218. URL: https://www.osti.gov/biblio/1875218.

[7] Saïd Derradji et al. “The BXI Interconnect Architecture”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015, pp. 18–25. DOI: 10.1109/HOTI.2015.15.

[8] Rossen Dimitrov and Anthony Skjellum. *Efficient MPI for Virtual Interface (VI) Architecture*.

[9] Edgar Gabriel et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104. ISBN: 978-3-540-30218-6.

[10] William Gropp. “MPICH2: a new start for MPI implementations”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Vol. 2474. 2002, p. 7.

[11] *Infiniband Architecture Specification*. Tech. rep. InfiniBand Trade Association, Nov. 2016.

[12] Jiuxing Liu, A. Vishnu, and D.K. Panda. “Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation”. In: *SC ’04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. 2004, pp. 33–33. DOI: 10.1109/SC.2004.15.

[13] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. “High Performance RDMA-Based MPI Implementation over InfiniBand”. In: *International Journal of Parallel Programming* 32 (2003), pp. 167–198.

[14] Jiuxing Liu et al. *MPI over InfiniBand: Early Experiences*. Tech. rep. 2003.

[15] M. Nüssle et al. “An FPGA-based custom high performance interconnection network”. In: *2009 International Conference on Reconfigurable Computing and FPGAs*. Dec. 2009, pp. 113–118. DOI: 10.1109/ReConFig.2009.23.

[16] Nikela Papadopoulou, Lena Oden, and Pavan Balaji. “A Performance Study of UCX over InfiniBand”. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2017, pp. 345–354. DOI: 10.1109/CCGRID.2017.149.

[17] Marc Pérache, Hervé Jourdren, and Raymond Namyst. “MPC: A Unified Parallel Runtime for Clusters of NUMA Machines”. In: *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*. Euro-Par ’08. Las Palmas de Gran Canaria, Spain: Springer-Verlag, 2008, pp. 78–88. ISBN: 978-3-540-85450-0. DOI: 10.1007/978-3-540-85451-7_9. URL: http://dx.doi.org/10.1007/978-3-540-85451-7_9.

[18] Simon Pickartz. “Virtualization as an enabler for dynamic resource allocation in HPC”. PhD thesis. 2019. DOI: 10.18154/RWTH-2019-02208.

[19] Simon Pickartz et al. “Non-Intrusive Migration of MPI Processes in OS-bypass Networks”. In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016.

[20] Ken Raffenetti, Antonio J. Pena, and Pavan Balaji. *Toward Implementing Robust Support for Portals 4 Networks in MPICH*. Undetermined. DOI: 10.1109/CCGrid.2015.79.

[21] S. Pai Raikar et al. “Designing Network Failover and Recovery in MPI for Multi-Rail InfiniBand Clusters”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 2012, pp. 1160–1167. DOI: 10.1109/IPDPSW.2012.142.

[22] Pavel Shamis et al. “UCX: An Open Source Framework for HPC Network APIs and Beyond.” In: *Hot Interconnects*. IEEE Computer Society, 2015, pp. 40–43. ISBN: 978-1-4673-9160-3. URL: http://dblp.uni-trier.de/db/conf/hoti/hoti2015.html#ShamisVLBHIDSGL15.

[23] Estela Suarez et al. *Modular Supercomputing Architecture – A success story of European R&D*. Tech. rep. ETP4HPC, May 2022.

[24] Sayantan Sur et al. “RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits”. In: *In PPoPP ’06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles*. 2006, pp. 32–39.

[25] *UCX*. URL: https://openucx.org (visited on 08/29/2022).

[26] T. S. Woodall et al. “Open MPI’s TEG point-to-point communications methodology: Comparison to existing implementations”. In: *In Proceedings, 11th European PVM/MPI Users’ Group Meeting*. 2004, pp. 105–111.

