GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Using High-Performance Networks

Tim Dettmar

MatrNr: 26327113

Supervisor: Sebastian Krey

Georg-August-Universität Göttingen
Institute of Computer Science

April 15, 2024

Abstract

High-Performance Computing (HPC) clusters rely heavily on fast networking to achieve good performance: fast communication to networked storage reduces the amount of time spent waiting for data, and fast inter-node communication allows for more effective horizontal scalability of parallel computing tasks. To that end, several network adapters used in HPC are equipped with custom hardware offload technologies to accelerate communication and reduce overhead. In order to take advantage of these offloads, networking libraries separate from those of traditional sockets are required. To determine the performance uplift and usage complexity, Libfabric, a commonly used high-performance communication library, is evaluated.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High-Performance Computing

RDMA Remote Direct Memory Access

MPI Message Passing Interface

EP Endpoint

PEP Passive Endpoint

CQ Completion Queue

MR Memory Region

EQ Event Queue

AV Address Vector

UCX Unified Communication X

RoCE RDMA over Converged Ethernet

API Application Programming Interface

TCP Transmission Control Protocol

UDP User Datagram Protocol

MMU Memory Management Unit

DMA Direct Memory Access

CPU Central Processing Unit

1 Introduction

Many networked applications use protocols such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), as well as any higher-level protocols that may be layered on top of these. In day-to-day workloads, these protocols perform adequately: no significant effort currently exists to replace them. The HTTP protocol used on the Internet, for instance, runs on TCP and UDP [Intc; Intd]. However, the landscape is different in HPC. Networking performance is often an important factor for many workloads in this field: the fastest supercomputing clusters have fast networking in the hundreds of gigabits of per-node throughput [TOP]. However, many of these networks also have another capability not available on regular consumer network adapters: full-stack data transfer offload, or Remote Direct Memory Access (RDMA).

Such offloads exist on network adapters marketed for use in HPC, and claim to reduce the burden of data transfer on the system's processor, while lowering latency and increasing throughput significantly. With traditional socket (TCP, UDP or similar) communications, system calls are made to the kernel socket interfaces [Ker10, ch. 59]. These interfaces handle buffering, packet assembly, loss recovery (if applicable to the protocol), firewalling, and other lower-level tasks abstracted away from the user's view. These tasks are complex, especially in the case of TCP [Ros14, p. 90, 269, 318–326]. With full-stack offloads, the theoretical benefit is the increase in processing time available for the actual workload due to the lack of involvement by the kernel. These new networking technologies have emerged over time in the HPC sector: among them are InfiniBand, RDMA over Converged Ethernet (RoCE), Omni-Path, as well as other custom designs.

While the theoretical benefits are clearly defined, this report aims to evaluate the usability and performance aspects of using the RDMA stack over the more commonly used socket interfaces.

2 HPC Network Programming

2.1 Introduction

The understanding of how high-performance networking libraries are implemented is useful for optimizing system performance to users' requirements, as well as for developing high-performance applications. For instance, system administrators may use the understanding of the network capabilities to set default network or job configuration options that work well for the majority of use cases. Middleware developers, such as those developing OpenMPI, can also design their communication backends to take advantage of such high-performance networks [The].

There are a few commonly used RDMA libraries: the InfiniBand Verbs Library (ibverbs), Unified Communication X (UCX), as well as Libfabric [Lin; UCF; Opel]. As the name suggests, the design of ibverbs is closely tied to that of InfiniBand, while the other libraries were specifically designed to address the modern HPC landscape, providing more generic, transport-independent abstractions for multiple high-performance networking technologies. They also provide for some higher-level functionality used by developers of HPC middlewares such as Message Passing Interface (MPI): an example can be found in Libfabric's implementation of collective communication functionalities [Opef]. From an

end-user perspective, these libraries are rather low-level. However, users of HPC systems do not necessarily need to interact with the libraries directly, in the same way as they might not be expected to write a socket application from scratch. Instead, domain-specific applications (e.g., OpenFOAM [Opem]) and abstraction layers such as MPI can be used, which often have support for the aforementioned high-performance networking libraries either directly or indirectly. An example of how these layers interact is shown in Figure 1.

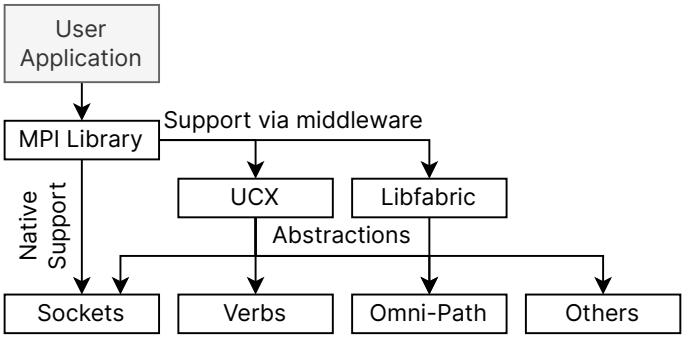


Figure 1: Data transfer abstraction layers

In this report, the focus will be on the Libfabric library, as it has the largest number of transports and operating systems supported. Theoretically speaking, Libfabric can be thought of as a write-once, run-anywhere HPC communication library. Its architecture abstracts many transport-technology-specific implementation details into a unified interface, which supports regular TCP and UDP sockets, InfiniBand- and Ethernet-based RDMA technologies, Omni-Path, and other proprietary transports [Opel].

2.2 Transport Selection

Libfabric is designed to present as similar of an interface as possible for various transport types, in order to improve the portability of applications written with it. The first call into Libfabric that a programmer would normally make is one to `fi_getinfo()`. This call can be used to list all fabric providers [1] (transports) on the system, or only those matching specific criteria. Typically, the application calls this function indicating the features required by it, which destination is desired, and capabilities the application is prepared to support, among others [Opeg]. An example is shown in Listing 1.

```
1  struct fi_info * hints = fi_allocinfo();
2  if (!hints)
3    return -ENOMEM;
4
5  // Specify the transport capabilities we want
6  // Reliable datagram with messaging and remote memory access capabilities
7  hints->ep_attr->type = FI_EP_RDM;
8  hints->caps          = FI_MSG | FI_RMA;
9
10 // Get available transports matching criteria, API support level 1.10
11 struct fi_info * infoList = NULL;
12 ret = fi_getinfo(FI_VERSION(1, 10), 0, 0, 0, hints, &infoList);
13 fi_freeinfo(hints);
14 if (ret < 0)
15   return ret;
```

Listing 1: Available transport request

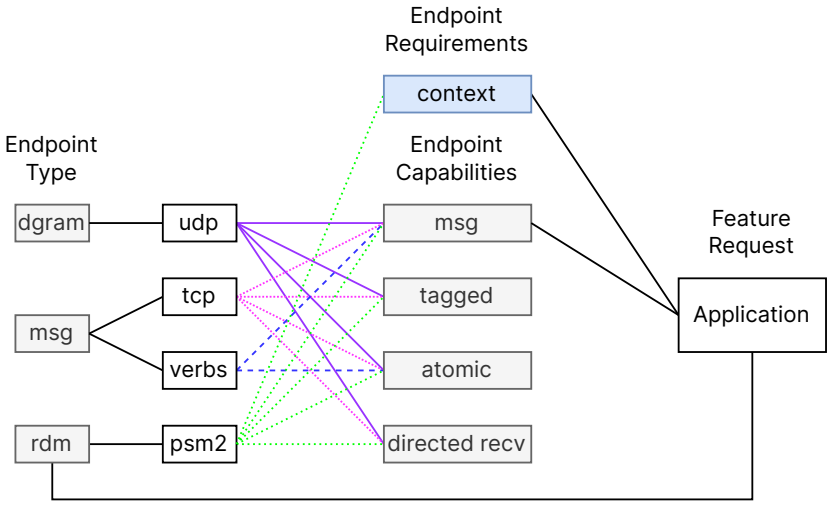


Figure 2: Transport selection

Once this call is made, Libfabric returns a list of transports that satisfy the criteria to the user. In the example shown in Figure 2, only the PSM2 transport is returned, as it is the only transport supporting the RDM (reliable datagram) endpoint type [2]. If the application instead did not support the context requirement and requested tagged functionality over datagram endpoints, then only the UDP transport would have been returned. In theory, the application itself would then be able to support new networking technologies when they become available in Libfabric, as long as the technology supports the features requested by the application, without significant changes to the codebase. If the application were instead using the underlying transport interfaces directly rather than going through the Libfabric abstraction layer, completely separate implementations would have to be written for each transport. Such applications are likely to have larger codebases and thus could be more bug-prone and difficult to maintain. For instance, the GWDG SCC [Ges] contains nodes with regular network adapters, InfiniBand adapters, and Omni-Path adapters. Therefore, to write an application that works well on all compute nodes, a programmer could either write three separate network backend implementations or write a single implementation based on Libfabric.

2.3 Resource Abstraction Layers

The Libfabric Application Programming Interface (API) is based on an object model, with defined roles and interactions between these objects [Opeb]. These objects are then implemented using transport-specific APIs, such as those in the underlying TCP socket or Verbs interfaces—the implementation details are not relevant to the user. It does not matter, for instance, that TCP is a stream protocol: Libfabric can emulate message-oriented communication over it.

In Figure 3, how resources from a native transport interface could be mapped into Libfabric is shown. In this case, the Verbs and RDMA Connection Manager APIs [Mela] are mapped to Libfabric resources. The list of resources exported by the libraries in the figure is not exhaustive, and the solid arrows refer to rough equivalents in terms of functionality rather than a direct mapping.

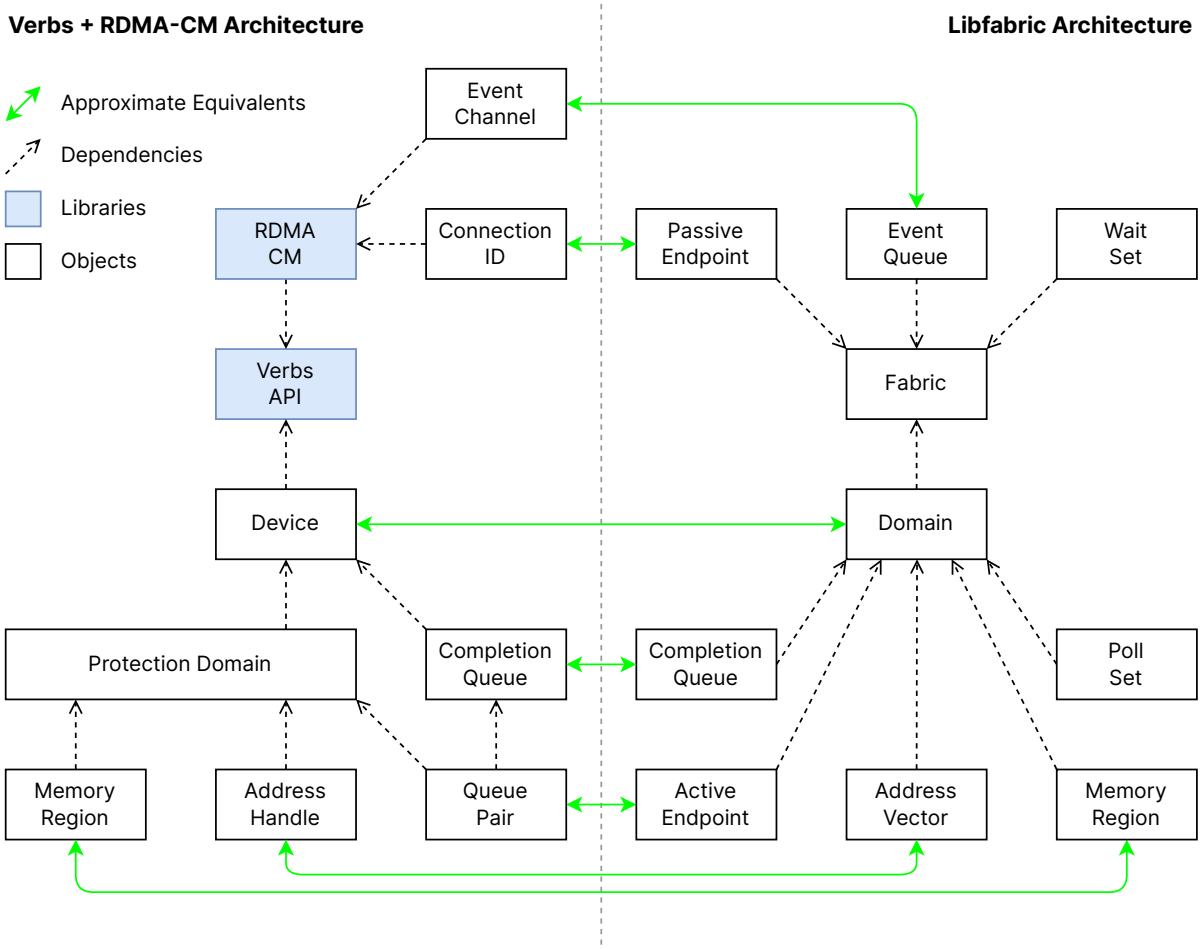


Figure 3: Libfabric abstraction layers

The resource abstractions have the following functionality:

- **Fabric**: Represents a group of mutually reachable resources. For instance, if two network adapters connected to the same subnet are installed in a node, these network adapters are considered part of the same fabric.
- **Domain**: Consists of a single transport on a single physical or virtual interface. On an InfiniBand NIC, for instance, the RDMA and TCP communication modes are presented as part of separate domains as they cannot communicate with each other. Several additional resources used in communication depend on the domain object.
- **Passive Endpoint (PEP)**: Like a passive socket, passive endpoints are used on connection-oriented endpoint types to accept incoming connections.
- **Event Queue (EQ)**: Used to report events, such as incoming connections, asynchronously to the user.
- **Completion Queue (CQ)**: Used to report completed data transfer operations.
- **Active Endpoint (EP)**: Represents an endpoint usable for communication with other peers.
- **Address Vector (AV)**: Stores mappings from Libfabric's abstract address handle to the native address of the transport in use, for connectionless endpoints.

- **Memory Region (MR)**: Block of memory made available for direct access by network hardware.
- **Poll/Wait Set**: Used for efficient polling/waiting across multiple completion queues.

2.4 Resource Initialization

Resources must be allocated or deallocated in an order allowed by the dependency chain, as shown in Figure 3. Therefore, the fabric object must always be allocated first, usually followed by the domain. Once these core resources are initialized, the application may choose which other resources are to be initialized based on its requirements. In the simplified example shown in this report, event queues and poll/wait sets will not be used. In all further examples, the following structure is used to group all required resources together for simplicity.

```
1  struct TestFabric {
2    struct fid_fabric * fabric;
3    struct fid_domain * domain;
4    struct fid_av     * av;   // Address vector
5    struct fid_ep     * ep;   // Endpoint
6    struct fid_cq     * cq;   // Completion queue
7    fi_addr_t           peer; // Target peer
8    void              * mem;  // Memory buffer
9    struct fid_mr     * mr;   // Memory region (from above buffer)
10 };
11
12 // Variable used in data transfer operations
13 struct TestFabric * f;
```

Listing 2: Communication resources

2.4.1 Completion Queue

For many data transfer tasks, a CQ is necessary. CQs contain information regarding completed data transfer operations, which is especially useful when reliable delivery of messages is required, or multiple messages are intended to be simultaneously in-flight [3]. The granularity of reported data can also be tuned by modifying the CQ format from a selection of available formats, as shown in Figure 4.

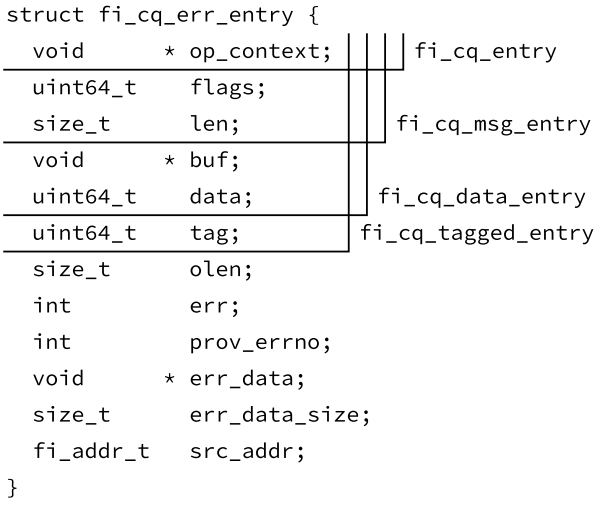


Figure 4: CQ entry types

The item "op_context" is particularly useful. This value can be provided by the user alongside any data transfer operation - and the completion associated with this operation always includes the user-defined value. As it is large enough to store a pointer, this could be any value from a simple integer to a complex user-allocated region of memory containing instructions on the further processing of the message. Upon creation, the CQ may be configured by the user, using the structure fi_cq_attr.

```
1  struct fi_cq_attr {
2    size_t                size;      // Number of entries
3    uint64_t              flags;     // Control flags
4    enum fi_cq_format     format;    // Entry format (see fi_cq_*_entry)
5    enum fi_wait_obj      wait_obj;  // Blocking wait object type
6    int                   signaling_vector; // Interrupt vector
7    enum fi_cq_wait_cond  wait_cond; // Blocking wait condition variable
8    struct fid_wait     * wait_set;  // Existing wait set to bind the CQ to
9  };
```

Listing 3: CQ attributes

The larger the CQ size, the more transfers can safely be posted simultaneously. Having multiple pending transfers can occur when communicating with multiple peers simultaneously, or sending small amounts of data at high speeds. For instance, when sending high-frequency sensor data for immediate processing, waiting until the peer has processed the previous data point before sending the next could lead to additional latency. The pending transfer count should be less than the CQ size, to prevent the CQ from being full or over-running, which can lead to data transfer stalls and undefined behaviour. Listing 4 demonstrates how a CQ may be created.

```
1  struct fi_cq_attr cq_attr;
2  memset(&cq_attr, 0, sizeof(cq_attr));
3  cq_attr.wait_obj = FI_WAIT_UNSPEC;    // This allows us to use fi_cq_sread()
4  cq_attr.format   = FI_CQ_FORMAT_DATA; // See 'CQ entry types'
5  cq_attr.size     = 64;                // Maximum number of elements
6  ret = fi_cq_open(f->domain, &cq_attr, &f->cq, 0);
7  if (ret < 0)
8    return ret;
```

Listing 4: CQ creation

2.4.2 Memory Registration

In socket interfaces, users may specify arbitrary buffers to be used for data transfers [IEE]. The kernel drivers and other subsystems implicitly abstract and handle the necessary steps to send the bytes contained within the buffer over the network. However, in RDMA-enabled communication, the network adapter should be able to read and write directly to user-space memory buffers. This presents several issues which need to be solved before such accesses are possible [Ros14, p. 381] [Oped]. Primarily, the RDMA subsystem operates on physical pages from the memory of the host Central Processing Unit (CPU) [4]. However, pages associated with user memory buffers may be swapped to disk as part of normal memory management operations. At this point, the network adapter would not be able to access paged-out memory regions via Direct Memory Access (DMA) unless they are paged back into system memory. In addition, because modern computers have a Memory Management Unit (MMU), the addresses used by the user's application do not correspond to the physical addresses accessible by the network adapter over the expansion bus.

The memory registration process involves kernel support in fixing these issues. Registered memory pages are locked (never paged out), and the kernel performs a lookup of virtual-to-physical memory mappings to ensure that the network adapter uses the correct address when accessing the memory region [5]. Once this is complete, the buffer provided by the user may be used in data transfer tasks. The example in Listing 5 demonstrates how a user-defined buffer may be registered.

```
1  // Since the RDMA subsystem operates on pages, we also want to
2  // page-align our memory allocation (1MB) for optimal access.
3  int ret = posix_memalign(&f->mem, sysconf(_SC_PAGESIZE), 1048576);
4  if (ret != 0)
5    return ret;
6
7  // FI_READ/_WRITE        = Local NIC read/write
8  // FI_REMOTE_READ/_WRITE = Remote NIC read/write (one-sided ops)
9  ret = fi_mr_reg(f->domain, f->mem, 1048576,
10                 FI_READ | FI_WRITE | FI_REMOTE_READ | FI_REMOTE_WRITE,
11                 0, 0, 0, &f->mr, 0);
12 if (ret < 0)
13   free(f->mem);
14
15 return ret;
```

Listing 5: Memory registration

2.4.3 Address Vector

An AV is used in connectionless endpoints - datagram (dgram) and reliable datagram (rdm) - in order to store peer addressing information. Data fabrics are diverse: Ethernet, InfiniBand, and Omni-Path may all have different, incompatible addressing formats. Despite this, Libfabric is designed to present a unified interface for all of them, which presents challenges when performing data transfer operations. Instead, Libfabric uses its own address format in data transfer functions [Opee].

```
1  ssize_t fi_recv(struct fid_ep *ep, void * buf, size_t len, void *desc,
2                  fi_addr_t src_addr, void *context);
3  ssize_t fi_send(struct fid_ep *ep, const void *buf, size_t len, void *desc,
```

```
4                   fi_addr_t dest_addr, void *context);
```

Listing 6: Libfabric send/receive functions

In order to support the translation of Libfabric's abstracted address format (fi_addr_t) into a native transport address, the AV is used. An AV contains a mapping from fi_addr_t addresses into native addresses, which are the actual addresses used internally for data transfers. The code snippet in Listing 7 demonstrates how one could use a socket address in a data transfer.

```
1  ssize_t ret;
2  struct fi_av_attr attr;
3  memset(&attr, 0, sizeof(attr));
4  attr.type  = FI_AV_UNSPEC; // Address vector type (unspec = auto-select)
5  attr.count = 16;           // Capacity of the address vector
6
7  struct fid_av * av;
8  ret = fi_av_open(f->domain, &attr, &f->av, NULL);
9  if (ret < 0)
10   return ret;
11
12 // We have a peer 10.0.0.10:12345 that we want to communicate with
13 struct sockaddr_in addr;
14 memset(&addr, 0, sizeof(addr));
15 addr.sin_family      = AF_INET;
16 addr.sin_addr.s_addr = inet_addr("10.0.0.10");
17 addr.sin_port        = htons(12345);
18
19 int ret2;
20 ret = fi_av_insert(f->av, &addr, 1, &f->peer, FI_SYNC_ERR, &ret2);
21 if (ret != 1)
22   return ret2;
```

Listing 7: Transport-specific address conversion

The address stored in `f->peer` is then suitable for all data transfer operations, such as those shown in Listing 6, and corresponds to the peer at 10.0.0.10:12345. However, any providers that do not use socket addresses natively must construct and exchange addresses in some other fashion, possibly out-of-band.

2.4.4 Endpoint

An endpoint can be compared to a file descriptor in regular sockets: it represents a channel on which data can be sent. The initialized fabric information and domain structures are used to derive transport type and addressing information. Once this endpoint is created, it must be enabled and bound to other created resources. It can then be used for data transfer operations. An example is shown in Listing 8.

```
1  // struct fi_info * info -> the structure received from a call to fi_getinfo()
2  // This structure is used to determine the endpoint type (e.g. TCP or RDMA)
3  // and source address, among many other parameters
4  ssize_t ret = fi_endpoint(f->domain, info, &f->ep, 0);
5  if (ret < 0)
6    return ret;
7
8  // The endpoint is bound to the CQ, such that data transfer operations of
9  // this endpoint are reported to the specified CQ
10 ret = fi_ep_bind(f->ep, &f->cq->fid, FI_TRANSMIT | FI_RECV);
```

```c
  if (ret < 0)
    return ret;

  // For connectionless endpoints, it is also necessary to bind the address vector
  // to the endpoint
  if (f->av) {
    ret = fi_ep_bind(f->ep, &f->av->fid, 0);
    if (ret < 0)
      return ret;
  }

  return fi_enable(f->ep);
```

Listing 8: Endpoint creation

2.5 Data Transfer

Once the previously mentioned resources have been initialized, the program can begin sending data across the network. However, programmers must be more careful in their data transfer tasks in comparison to regular socket programming. RDMA data transfer operations are mostly asynchronous without implicit temporary buffers where messages can be stored before processing. Therefore, in order to perform a successful send operation, a corresponding receive operation must also be posted on the receiving peer. Buffers must also be sized appropriately to receive the entire contents of a message [Opeh].

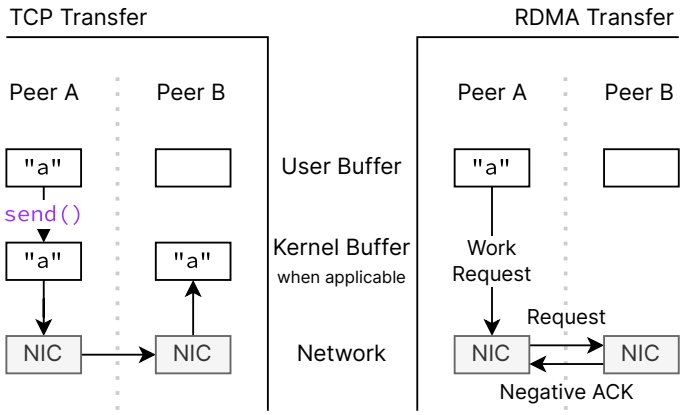


Figure 5: Unexpected data transfers with TCP/RDMA

Unexpected, in the case shown in Figure 5, refers to a connection which has already been established between two peers, but where a send has been called without a corresponding receive on the receiving side. In practice, a posted send without a receive might actually submit successfully without a negative acknowledgement, but never generate a completion notification on the sending side—this is the behaviour exhibited in the benchmark program that was written for this report.

Data transfer operation types in Libfabric (and Verbs) can be broken down into the following: Send, Receive, Read, Write, and Atomic. Send and receive operations are the RDMA equivalents of the same socket operations with some semantic differences—notably the lack of intermediate buffering as demonstrated in Figure 5. Read, write, and atomic operations are known as one-sided operations. These still involve communication between two peers, but they are named so because the target peer is not informed when

the operation occurs [Opei]. The initiator of a read operation copies the contents of a remote memory region into local memory, while write operations perform the opposite task. Atomic operations allow the network adapter to perform atomic fetch-add and compare-and-swap (CAS) operations on remote memory. This atomicity is only defined at the library level; accessing a region of memory directly through a pointer could result in non-atomic updates being observed [Opec]. Like regular atomics, one could use Libfabric atomics to build higher-level constructs such as locks, mutexes, and semaphores. Unlike the operating-system-provided counterparts, however, these atomics function over the network. As such, they can be used for synchronization across multiple nodes. This makes it suitable for writing code that maintains thread-safety even over distributed memory.

Submitting a data transfer request always returns immediately (when the system is working normally, and if there are enough resources available). The previously mentioned CQ in Section 2.4.1 is what informs the application once the transfer has been completed. To optimize for message rates and latency, one could request data from the CQ in a busy-wait loop. However, most applications have useful work which needs to be performed on received data in any case. Instead of using resources to poll for completions, one could only poll for data when computation is completed, leaving more processing time for useful work. This is significant for RDMA transports because the data transfer is offloaded to hardware; with TCP, the kernel still performs data transfer tasks with the CPU in the background.

2.5.1 Parallelism

Multiple requests may be posted simultaneously to increase aggregate throughput. This may be useful, for instance, when communicating with multiple peers simultaneously, or when sending high-frequency streaming data. The former is often the case in distributed parallel computing. In such cases the context parameter, an arbitrary 64-bit value, is useful; when the completion queue is read, each send and receive can contain a different context parameter to determine which transfers have completed from all of the previously posted tasks.

Figure 6 demonstrates one method for how multiple messages may be sent and received simultaneously using only the context parameter for directing completions to the correct messages. In this example, a large memory region is sliced into several logical sub-regions in which messages can be placed. Separately, a memory block is used to store context information for each slice. Some additional context information is defined in the figure, such as the intended target of the message. When a message send or receive request is posted, the corresponding context is filled by the user with information relevant to the user's application, and provided as the context parameter in data transfer calls. For instance, if the application intends to send data to 10 peers, the parameter targetAddr may be set to the target address of each peer. When the provided context is then returned in a completion, the application has information on which peer the transfer was completed for.

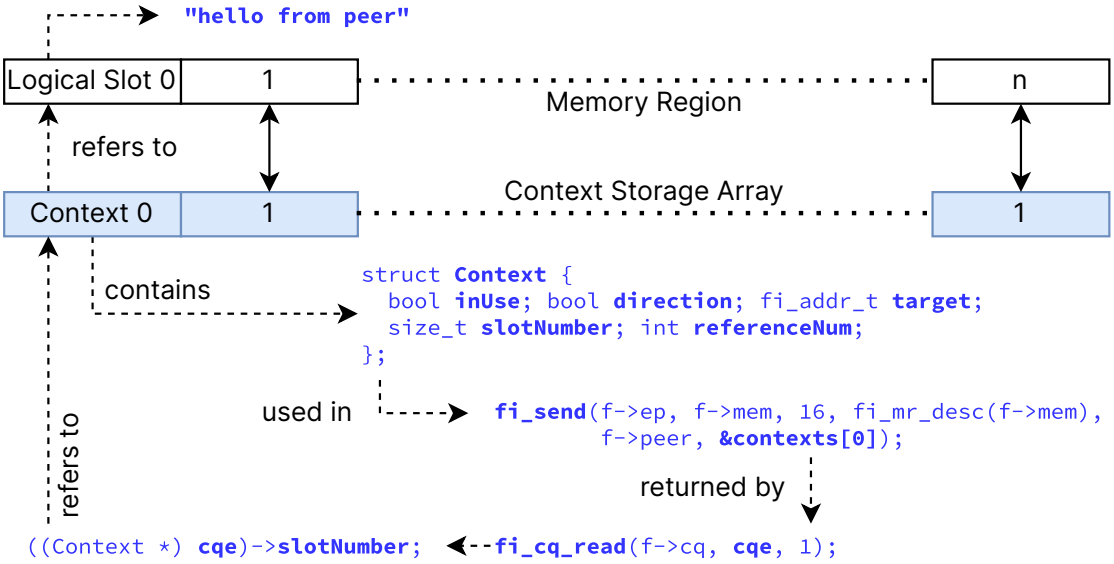


Figure 6: Context parameter message tracking

3 Evaluation

3.1 Configuration

To evaluate Libfabric, a benchmark application was written, where two endpoints communicate with each other over various transports. A client sends arbitrary data to the server, which receives it. Then, performance was evaluated based on the message size with and without parallelism enabled. For serial runs, the application waits until each message has been sent before sending the next message. In parallel runs, the application allows for multiple messages to be sent simultaneously. For native InfiniBand, the parallelism level is 64. For Omni-Path, it was set to 16. The number of messages sent per message size in the benchmark varies from 200 to 2000, and the second half of the collected results are used in the performance calculation. All of the different benchmark configurations and operating modes may be found in the appendix in Section A.1.

A lower number of parallel transfers were required in Omni-Path due to the unreliable data transfer behaviour above 16 transfers. This is likely due to the limited Omni-Path acceleration capabilities relative to InfiniBand. The 16 parallel transfers correspond to the 16 SDMA engines on the network card, checked using the steps described in the manual [Inta, p. 80-81]—this configuration reliably completed all benchmark runs.

3.2 Performance

The data transfer throughput results are shown with the benchmark application optimized for throughput (i.e., asynchronous mode busy-wait polling). This uses 100% of at least a single core when polling for data completions.

Detailed results for both systems can be found in the appendix in Section B. The performance results for InfiniBand are shown in Figure 7. The categories on the X-axis represent the message payload size in bytes, while the Y-axis represents the average throughput in gigabits per second. The labels represent the transport, test type, and

parallelism level, in that order. The peak network throughput is 100Gbps; further information regarding the system configuration may be found in the appendix in Section A.2.

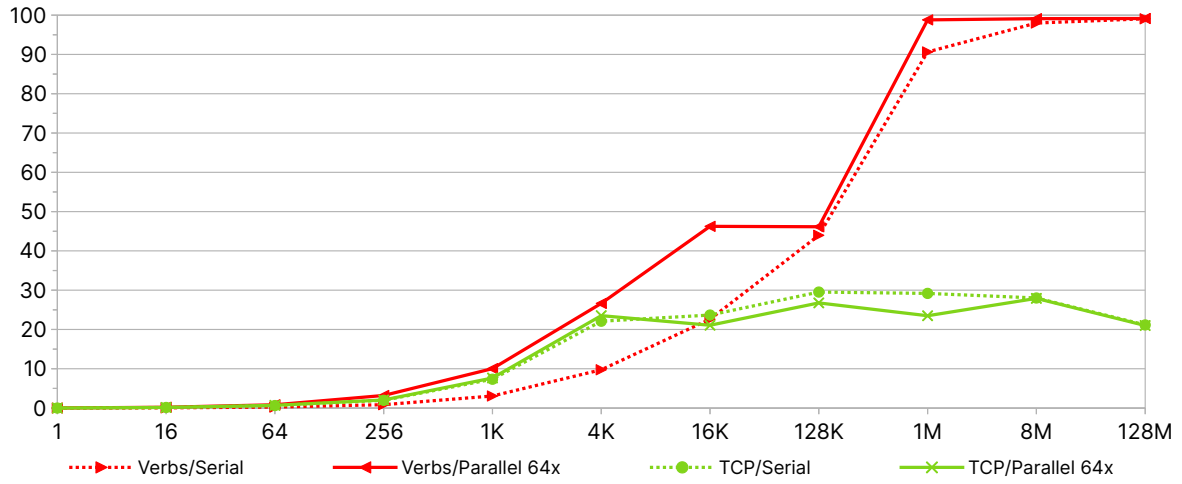


Figure 7: Data transfer throughput (InfiniBand)

The performance results for Omni-Path with 100 Gbps networking are shown in Figure 8. Note that the throughput results are not directly 1:1 comparable between InfiniBand and Omni-Path due to the different system configuration [6]. The system configuration may be found in Section A.3. There are three main methods to interact with the Omni-Path fabric: via the native Libfabric PSM2 and OPX providers, or via Verbs. At the time of this report, the OPX provider was missing a connection management system used by the benchmarking application. Therefore, only the PSM2, Verbs, and TCP providers were tested.

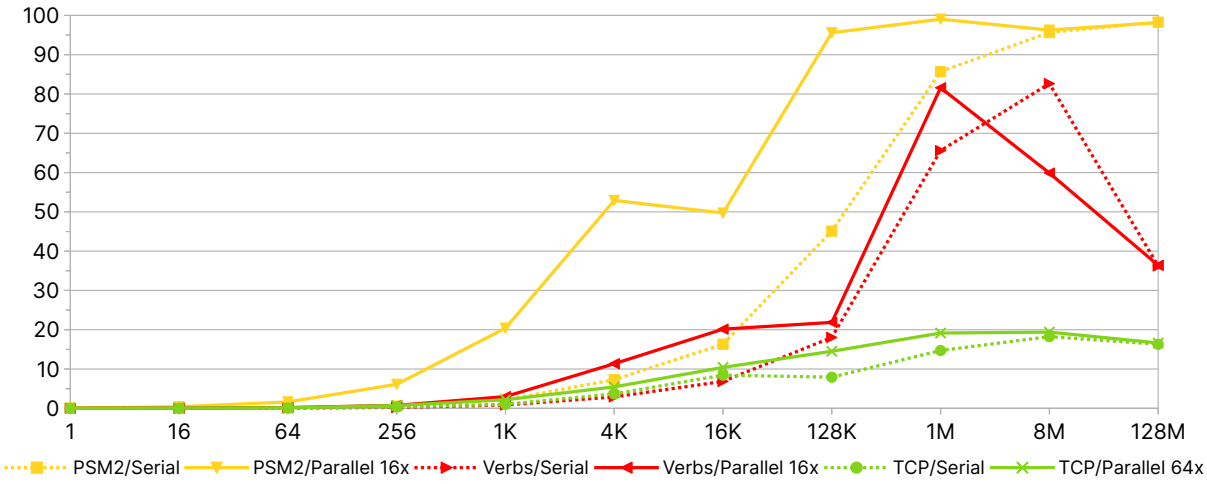
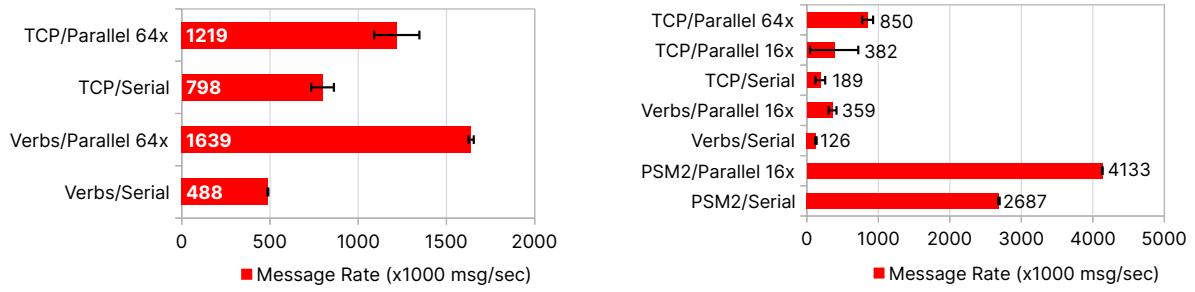


Figure 8: Data transfer throughput (Omni-Path)

Overall throughput with accelerated transports (Verbs, PSM2) was significantly higher than that of regular TCP across the majority of test configurations. An exception to this is when Verbs on Omni-Path is compared to TCP at small message sizes, which could be due to the overhead of running a non-native transport combined with the background

optimization performed by the Libfabric TCP layer, as well as the kernel itself. The InfiniBand system has significantly more consistent performance relative to the Omni-Path system. Above 1MB, the performance of Verbs over Omni-Path varies significantly (cf. Table 1). This could be due to the translation overhead causing CPU resource contention at larger parallelism levels and message sizes. There is little documentation on this topic, however, one recommendation from a server vendor mentions this issue [Len]. This theory also explains the drop in parallel performance before the drop in serial performance, as there are more transfers for the CPU and network adapter to manage in parallel mode.

Higher throughput at the same message size implies higher message rates, which is shown in Figure 9. Message rates are calculated by dividing the number of messages sent by the total runtime at each stage of the benchmark. The smallest message size, 1 byte, is used to ensure that link bandwidth is not the limiting factor.

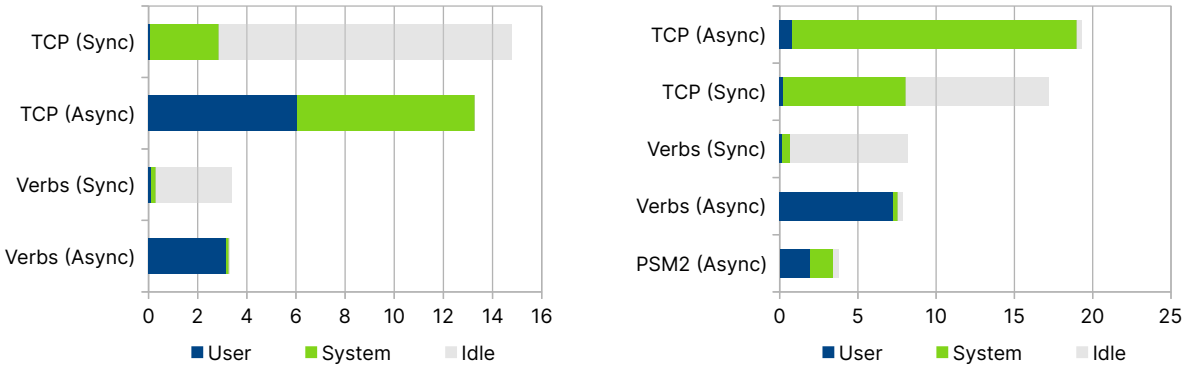


(a) InfiniBand (b) Omni-Path

Figure 9: Message rates

The performance of Omni-Path on any transport other than PSM2 was relatively low. While 126,000 messages per second might still be acceptably fast for many use cases, the discrepancy in performance is significant compared to InfiniBand, which performs relatively well regardless of the transport in use. However, it is possible that the newer kernel version on the InfiniBand equipped system has received more performance enhancements in the kernel network stack and associated drivers.

Another important factor to consider is CPU usage. In Figure 10, the serial data transfer results from both message-rate optimized (Async) and CPU-usage optimized (Sync) tests are compared with each other. The bars in the chart are stacked and represent the amount of CPU-seconds the benchmark spent in each state. User represents time spent in userspace, System represents time in the kernel, and Idle represents the time the program was waiting without involving the CPU (e.g., waiting for data to arrive). The sum of the stacked charts represents the total runtime of the program in seconds.



(a) InfiniBand (b) Omni-Path

Figure 10: CPU time results

Using an accelerated transport on both systems significantly improved runtimes relative to TCP. This was already apparent given the higher throughput. Instead, when considering the actual amount of consumed CPU time (user + system), both InfiniBand and Omni-Path consumed less actual processor time with accelerated transports relative to regular TCP. The difference is significant: On the InfiniBand system, the synchronous Verbs implementation consumed only 9.84% of the CPU time of the synchronous TCP implementation. Likewise, on the Omni-Path system, it was 3.43%. The PSM2 transport in Libfabric did not work reliably in sync mode. Surprisingly, there is only a very small runtime benefit when using Verbs in async mode, with a significant increase in CPU usage. Therefore, users may want to use sync mode for large messages and async mode for small messages, or when latency between receiving and processing a message must be minimized.

This significant CPU time reduction could be useful in two main ways: more computing power would be available for the user to take advantage of, which increases the amount of work that can be done in the same amount of time. Otherwise, if the application is not compute-bound (in user-space), more users are able to share the CPU time that would previously be consumed by networking, increasing density and possibly reducing the total number of servers required.

When all results are considered, Libfabric fulfils its claims of being high-performance and demonstrates the reason why such libraries exist. As modern HPC networks continue to improve in throughput, the use of an accelerated networking library will likely only become more important over time. However, these results also show that although Libfabric provides a unified interface for all of them, not all HPC transports are equal in terms of capabilities. Very compute-heavy applications, which also require large amounts of data, might benefit more from the use of InfiniBand over Omni-Path due to the lower CPU overhead, and such decisions may have to be considered from a use-case, purchasing, and return-on-investment perspective as well, rather than simply raw performance results.

3.3 Usability

For users that use MPI or scientific applications directly, changing the transport type can be as straightforward as changing a runtime argument [The, § 11-12]. OpenMPI, for instance, has integrated support for several high-performance transports via Libfabric, as well as other libraries. At a higher level, an application such as GROMACS uses MPI

[GRO], and can thus implicitly benefit from the faster transport when the underlying MPI library is built with RDMA capabilities. Based on the performance results, users can expect higher throughput for applications with very high message rates or bandwidth, which could lead to a reduction in runtime if these were the limiting factors.

System administrators can also benefit from the performance advantages in the backend, such as with faster storage and reduced overhead, enabling higher compute efficiency. However, support is often highly dependent on the application in use. Experimental support for such transports in Ceph would likely be ill-suited for a production HPC storage cluster if reliability is of importance. In addition, due to the lack of available documentation and limited support, debugging performance issues that could have cluster-wide impacts is likely to be more complex and could result in longer maintenance downtimes. This could present an unacceptably high risk for time-sensitive applications: the weather forecast for tomorrow may not be particularly interesting when delivered next week, and significant operational impacts could occur to a business with non-functional storage. However, middleware such as MPI, used by users directly or by higher-level applications in which the intended targets are HPC users, often support libraries such as Libfabric in the backend, and could be a more maintainable solution. As these libraries and end-user applications often have multiple transports that can be switched by the user, a failed transport type due to incomplete support or mismatched software versions can be mitigated by switching the transport type in use (e.g. from native InfiniBand to TCP/IP over the same InfiniBand network). Although this would result in degraded performance, this could be preferable to complete downtime while the issue is being addressed.

Viewing the interfaces from a developer's perspective—those developing middleware such as MPI, as well as standalone applications such as Ceph that cannot rely on MPI and have previously implemented RDMA support from scratch using the Verbs interface [Cepa]—is where usability issues with Libfabric, Verbs, and similar interfaces are the most apparent. Getting good performance from an accelerated transport does not only consist of using these libraries without consideration. Rather, the tight coupling of the application's functionality with the networking library, as well as an understanding of the limitations of individual transports, are required for the best results. The proof of concept program written in this report is significantly longer, with large amounts of boilerplate code, relative to an equivalent program written with regular sockets. In particular, the quirks of each transport must be managed by the programmer: different addressing formats, transport requirements, environment variables, and required resources must be handled on a transport-by-transport basis. In the case of the program written for this report, independent code paths were added specifically to enable partial support for the PSM2 transport. Thus, some of the benefits of using a unified library such as Libfabric relative to a transport-specific library such as Verbs might be limited by the difficulty in actually working with these additional transports. These issues are known, however, and are slated to be addressed in a future Libfabric release [Sea].

Making use of RDMA effectively is also made more difficult by the lack of available documentation, both in terms of interface documentation and code samples. While the Libfabric documentation contains descriptions of its functions, information on how to combine these functions into a usable program is sparse. This is especially the case when using less common transports such as Omni-Path. While this situation is somewhat better with Verbs, as the documentation provided by Mellanox is more extensive [Melb], the level of support available from public resources and community support falls short relative to regular socket programs. Socket-based programs also have a more diverse ecosystem of

software unrelated to HPC. One can easily find TCP and UDP networking code used in smart devices, games, and web browsers; the software landscape for Libfabric or Verbs-powered software is comparatively small. Thus, the barrier to entry for developers is relatively high. As demonstrated in the performance results in Figure 7, supported hardware in combination with high-performance networking software is required for optimal performance, and significant additions or changes are required to existing codebases—additions that may not be maintainable in the long term. The GlusterFS file system dropped support for RDMA transports as a result [Glu], while Ceph documentation for using its InfiniBand Verbs support is sparse due to its experimental nature [Cepb]. File systems such as Lustre with a specific focus on HPC, however, have official support for fully-offloaded transports [Lus].

Therefore, the limitation of support to mostly HPC applications suggests that it could only make sense from a cost and developer resource allocation standpoint to program codepaths for high-performance networks when the software is expected to run primarily on HPC clusters. Based on the gathered results, a file transfer program used by regular users on slower networks (i.e., under 10 Gbit/s) might see little to no benefit: with messages 16K or higher in size, the TCP/UDP stack is not a bottleneck at such speeds.

Regardless, these libraries are still powerful tools for high-performance communication in and outside of the HPC space that can be taken advantage of by developers. It provides clear performance benefits when used correctly, especially with modern high-performance datacenter networks reaching as high as 400Gbps of throughput. In such cases, the use of accelerated networking protocols that these libraries enable is necessary for taking full advantage of the network's capabilities and justifying the hardware costs.

4 Conclusion

From a raw performance standpoint, libraries such as Libfabric and Verbs that allow for direct access to supported networking hardware are extremely powerful and can be useful in applications limited by communication performance. Especially at high throughput, message rates, or both, the use of offloaded transports showed clear performance advantages over TCP, as well as an over ten-fold decrease in CPU utilization.

However, these libraries fall short in terms of usability. Documentation is especially limited, which exacerbates the usability issues of an already complex and relatively niche library. Middleware designed with HPC in mind, such as OpenMPI, have integrated support for Libfabric transports, which, if the application can take advantage of it, mostly abstracts the usability problem away from end-users. However, applications that cannot rely on such layers, such as storage applications, are left to interface with the lower-level libraries directly. Therefore, the high level of expertise required in combination with the lack of documentation likely limits the potential benefit of these libraries. Several applications exist that do not use Libfabric or a similar library, despite the performance benefits and use cases, and the examples demonstrated in this report have shown that complexity and maintainability are contributing factors to this issue.

It is possible that if new libraries emerge, or if existing libraries become more straightforward to use, more widespread adoption of the technologies detailed in this report would be enabled.

References

[Cepa] Ceph Authors and Contributors. *Ceph - src/msg/async/rdma/Infiniband.h*. https://github.com/ceph/ceph/blob/main/src/msg/async/rdma/Infiniband.h.

[Cepb] Ceph Authors and Contributors. *Network Configuration Reference*. https://docs.ceph.com/en/reef/rados/configuration/network-config-ref/.

[Ges] Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen. *Scientific Compute Cluster (SCC)*. https://gwdg.de/en/hpc/systems/scc/.

[Glu] Gluster Developers. *RDMA Transport*. https://docs.gluster.org/en/main/Administrator-Guide/RDMA-Transport/.

[GRO] GROMACS Development Team. *Getting good performance from mdrun - Parallelization over multiple nodes via MPI*. https://manual.gromacs.org/current/user-guide/mdrun-performance.html.

[IEE] IEEE / The Open Group. *The Open Group Base Specifications Issue 7, 2018 edition - Networking Services*. https://pubs.opengroup.org/onlinepubs/9699919799/idx/networking.html.

[Inta] Intel Corporation. *Intel® Omni-Path Fabric Performance Tuning*. https://www.intel.com/content/dam/support/us/en/documents/network-and-i-o/fabric-products/Intel_OP_Performance_Tuning_UG_H93143_v18_0.pdf.

[Intb] Intel Corporation. *Intel® Performance Scaled Messaging 2 (PSM2) Programmer's Guide*. https://www.intel.com/content/dam/support/us/en/documents/network-and-i-o/fabric-products/Intel_PSM2_PG_H76473_v14_0.pdf.

[Intc] Internet Engineering Task Force. *RFC 9112 - HTTP/1.1 - 9. Connection Management*. https://datatracker.ietf.org/doc/html/rfc9112.

[Intd] Internet Engineering Task Force. *RFC 9114 - HTTP/3*. https://datatracker.ietf.org/doc/html/rfc9114.

[Ker10] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010.

[Len] Lenovo Group. *Omni-Path accelerated RDMA settings for Verbs transport*. https://datacentersupport.lenovo.com/us/en/products/solutions-and-software/converged-hx/hx5510/8695/solutions/ht504160-omni-path-accelerated-rdma-settings-for-verbs-transport-lenovo-x86-servers.

[Lin] Linux RDMA Contributors. *Introduction*. https://github.com/linux-rdma/rdma-core/blob/master/Documentation/libibverbs.md.

[Lus] Lustre Developers. *Lustre Networking (LNET) Overview*. https://wiki.lustre.org/Lustre_Networking_(LNET)_Overview.

[Mela] Mellanox Technologies. *InfiniBand*. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/rdma-aware+programming+overview.

[Melb] Mellanox Technologies. *Programming Examples Using IBV Verbs*. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/programming+examples+using+ibv+verbs.

[NVIa] NVIDIA Corporation. *Optimized Memory Access*. https://docs.nvidia.com/networking/display/mlnxofedv461000/optimized+memory+access.

[NVIb] NVIDIA Corporation. *Overview - GPUDirect RDMA 12.4 Documentation*. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[Opea] Open Edge HPC Initiative. *HAICGU 0.1 Documentation*. https://haicgu.github.io/index.html.

[Opeb] OpenFabrics Interfaces Working Group. *fi_arch(7)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_arch.7.html.

[Opec] OpenFabrics Interfaces Working Group. *fi_atomic(3)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_atomic.3.html.

[Oped] OpenFabrics Interfaces Working Group. *fi_av(3)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_mr.3.html.

[Opee] OpenFabrics Interfaces Working Group. *fi_av(3)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_av.3.html.

[Opef] OpenFabrics Interfaces Working Group. *fi_collective(3)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_collective.3.html.

[Opeg] OpenFabrics Interfaces Working Group. *fi_getinfo(3)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_getinfo.3.html.

[Opeh] OpenFabrics Interfaces Working Group. *fi_msg(3)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_msg.3.html.

[Opei] OpenFabrics Interfaces Working Group. *fi_rma(3)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_rma.3.html.

[Opej] OpenFabrics Interfaces Working Group. *fi_rxd(7)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_rxd.7.html.

[Opek] OpenFabrics Interfaces Working Group. *fi_rxm(7)*. https://ofiwg.github.io/libfabric/v1.20.0/man/fi_rxm.7.html.

[Opel] OpenFabrics Interfaces Working Group. *Open Fabric Interfaces*. https://github.com/ofiwg/libfabric.

[Opem] OpenFOAM Foundation. *Running applications in parallel*. https://www.openfoam.com/documentation/user-guide/3-running-applications/3.2-running-applications-in-parallel.

[Ros14] Rami Rosen. *Linux Kernel Networking Implementation and Theory*. Apress, 2014.

[Sea] Sean Hefty. *libfabric 2.0 release*. https://github.com/ofiwg/libfabric/discussions/8049.

[The] The Open MPI Project. *FAQ: Tuning the run-time characteristics of MPI InfiniBand, RoCE, and iWARP communications*. https://www.open-mpi.org/faq/?category=openfabrics#ib-btl.

[TOP] TOP500.org. *Sublist Generator*. https://www.top500.org/statistics/list/.

[UCF] UCF Consortium. *UCX - Unified Communication X*. https://openucx.org/.

A Benchmark Configuration

A.1 Run Configurations

There are 5 different parameters tuned between different benchmark runs: parallelism level, synchronous mode, and long test mode. The parameters have the following meanings:

- Parallelism level: The maximum number of in-flight, unconfirmed messages
- Synchronous mode: Whether the application polls in a loop for data completion or blocks until one is received
- Long test mode: Send large (128MB) blocks of data only. Otherwise, various message sizes are tried, from 1 byte to 128MB.

Certain providers (transports) are limited in terms of functionality, so certain features and parallelism levels were disabled. All combinations of enabled functionality were tried.

- PSM2: Parallelism levels 1+16, sync mode off [7], long test mode on+off
- Verbs on Omni-Path: Parallelism levels 1+16, sync mode on+off, long test mode on+off
- Verbs on InfiniBand: Parallelism levels 1+16+64, sync mode on+off, long test mode on+off
- TCP on Omni-Path: Parallelism levels 1+16+64, sync mode on+off, long test mode on+off
- TCP on InfiniBand: Parallelism levels 1+16+64, sync mode on+off, long test mode on+off

The average of 3–4 benchmark runs (depending on result consistency) were taken to produce the results shown in the evaluation benchmark section. The benchmarks were conducted on two separate nodes: one as the client, one as the server, with the following configurations.

The number of messages sent during the test for message sizes of 1 byte–1 MB is 2000, 8MB is 1000, and 128MB is 200. For the long test mode, 1000 128MB messages were sent, but the latest 90% of results are considered rather than 50%.

A.2 InfiniBand Benchmark Node Configuration

Results were gathered on the HAICGU cluster of the University of Frankfurt [Opea] on the ARM nodes with exclusive access.

Processor	Kunpeng 920, 64 cores x 2 sockets, no SMT
Operating System	Rocky Linux 8.8
Kernel Version	4.18.0-477.27.1.el8_lustre.aarch64
Libfabric Version	1.20.0 (compiled from source)
Compile Options	Verbs, TCP, RXM
Network Adapter	Mellanox ConnectX-5
Network Configuration	100Gbps InfiniBand

A.3 Omni-Path Benchmark Node Configuration

Results were gathered on the GWDG SCC [Ges] on the "amp" nodes with exclusive access.

Processor	Intel Xeon Platinum 9242, 48 cores x 2 sockets, no SMT
Operating System	Scientific Linux 7.9
Kernel Version	3.10.0-1160.95.1.el7.x86_64
Libfabric Version	1.20.0 (compiled from source)
Enabled Transports	PSM2, Verbs, TCP, RXM
Network Adapter	Intel Omni-Path HFI100
Network Configuration	100Gbps Omni-Path

B Results

The following tables contain a subset of the gathered performance data from both clusters. The raw dataset is provided with the supplementary materials included with this report.

Parameters		Throughput mbit/sec			Message Rate x1000 msg/sec			Std. Deviation Relative SD %		
Parallelism	**Size**	**PSM2**	**Verbs**	**TCP**	**PSM2**	**Verbs**	**TCP**	**PSM2**	**Verbs**	**TCP**
1	1	21.49	1.01	1.51	2686.57	126.44	188.87	0.40	8.16	35.92
	16	37.08	16.18	24.79	289.71	126.37	193.63	0.78	8.33	46.23
	64	148.17	64.18	91.82	289.39	125.36	179.35	0.43	7.93	41.38
	256	571.32	224.56	336.79	278.96	109.65	164.45	1.54	8.89	36.73
	1024	2160.95	852.39	1011.73	263.79	104.05	123.50	0.14	8.61	18.21
	4096	7301.00	2908.30	3671.95	222.81	88.75	112.06	0.10	7.18	1.55
	16384	16315.10	6852.23	8440.31	124.47	52.28	64.39	0.77	8.12	7.62
	131072	45104.09	18044.73	7925.56	43.01	17.21	7.56	1.86	8.64	1.63
	1048576	85687.27	65592.73	14712.07	10.21	7.82	1.75	0.79	1.50	1.31
	8388608	95646.36	82628.44	18246.00	1.43	1.23	0.27	0.06	1.43	2.38
	134217728	98273.86	36223.65	16291.09	0.09	0.03	0.02	0.12	0.53	1.20
16	1	33.06	2.88	3.06	4132.57	359.44	382.06	0.13	15.45	88.41
	16	392.69	46.34	31.38	3067.87	362.05	245.12	0.62	15.00	28.25
	64	1632.21	181.66	199.72	3187.92	354.80	390.08	1.00	14.77	86.72
	256	6113.15	754.25	477.82	2984.94	368.29	233.31	1.61	11.18	36.45
	1024	20401.04	2983.14	2380.67	2490.36	364.15	290.61	3.31	10.77	38.74
	4096	52913.52	11348.85	4787.17	1614.79	346.34	146.09	0.35	10.88	17.56
	16384	49704.76	20142.44	9982.30	379.22	153.67	76.16	0.16	4.08	2.45
	131072	95590.27	21866.03	13756.34	91.16	20.85	13.12	0.69	6.73	9.28
	1048576	99033.03	81578.92	17103.14	11.81	9.72	2.04	0.15	3.34	17.80
	8388608	96260.99	59878.69	17597.89	1.43	0.89	0.26	3.86	31.09	18.60
	134217728	98175.08	36386.25	16295.96	0.09	0.03	0.02	0.28	0.21	1.30
64	1			6.80			850.35			8.97
	16			62.21			486.04			62.46
	64			131.72			257.26			21.54
	256			726.73			354.85			76.95
	1024			2177.56			265.82			10.52
	4096			5449.44			166.30			1.94
	16384			10421.15			79.51			2.22
	131072			14512.02			13.84			1.36
	1048576			19163.31			2.28			1.41
	8388608			19393.98			0.29			2.77
	134217728			16627.83			0.02			2.03

Table 1: Omni-Path message rate benchmarks

Parameters		Throughput mbit/sec		Message Rate x1000 msg/sec		Std. Deviation Relative SD %	
Parallelism	**Size**	**Verbs**	**TCP**	**Verbs**	**TCP**	**Verbs**	**TCP**
1	1	3.90	6.38	487.94	797.79	0.46	8.08
	16	61.66	116.31	481.73	908.68	3.23	3.89
	64	250.72	455.79	489.69	890.21	0.78	4.30
	256	858.44	1630.90	419.16	796.34	0.17	3.24
	1024	3074.93	5687.94	375.36	694.33	7.12	2.54
	4096	9748.05	15227.70	297.49	464.71	1.95	29.06
	16384	22605.80	21518.23	172.47	164.17	0.24	9.36
	131072	43975.58	31009.95	41.94	29.57	1.69	6.30
	1048576	90636.79	28458.38	10.80	3.39	0.02	9.73
	8388608	97982.20	26517.61	1.46	0.40	<0.01	5.57
	134217728	99065.15	21276.38	0.09	0.02	<0.01	3.47
16	1	12.89	7.87	1610.64	983.84	2.23	6.61
	16	208.92	154.05	1632.19	1203.51	2.14	1.80
	64	813.19	637.81	1588.26	1245.71	0.43	1.26
	256	3346.21	1976.36	1633.89	965.02	0.45	3.75
	1024	4039.31	7342.31	493.08	896.28	0.61	4.75
	4096	15731.27	22099.77	480.08	674.43	0.79	6.32
	16384	47310.2	23668.91	360.95	180.58	0.55	7.86
	131072	46405.65	29516.79	44.26	28.15	3.31	30.90
	1048576	98379.76	29178.87	11.73	3.48	0.12	4.54
	8388608	99053.74	28009.94	1.48	0.42	<0.01	0.24
	134217728	99134.13	21154.73	0.09	0.02	<0.01	3.31
64	1	13.11	9.75	1639.37	1218.56	0.93	10.52
	16	215.40	159.55	1682.79	1246.49	0.29	6.30
	64	839.99	690.88	1640.60	1349.38	1.35	3.28
	256	3208.22	2065.83	1566.51	1008.70	0.78	4.89
	1024	10037.24	7654.52	1225.25	934.39	4.86	3.33
	4096	26636.63	23518.47	812.89	717.73	3.01	8.80
	16384	46248.73	21070.44	352.85	160.75	0.54	18.19
	131072	46149.34	26763.02	44.01	25.52	0.45	25.70
	1048576	98789.72	23493.17	11.78	2.80	0.01	29.12
	8388608	99097.11	27912.66	1.48	0.42	<0.01	7.22
	134217728	99137.27	21005.58	0.09	0.02	<0.01	3.53

Table 2: InfiniBand message rate benchmarks

Parameters		Transport														
		PSM2					Verbs					TCP				
Mode	**Parallelism**	**User**	**Sys**	**Idle**	**Time**	**CPU**	**User**	**Sys**	**Idle**	**Time**	**CPU**	**User**	**Sys**	**Idle**	**Time**	**CPU**
Async	1	1.97	1.46	0.31	3.74	91.72%	7.24	0.30	0.37	7.92	95.33%	4.03	14.67	0.34	19.05	98.20%
	16	1.31	2.02	0.31	3.64	91.48%	6.63	1.81	0.35	8.80	95.96%	4.24	14.93	0.35	19.51	98.22%
	64											4.54	15.37	0.35	20.25	98.27%
Sync	1						0.16	0.49	7.84	8.49	7.67%	0.20	7.83	9.13	17.15	46.79%
	16						0.26	1.20	7.07	8.52	17.05%	0.42	8.01	9.04	17.46	48.27%
	64											1.15	8.72	9.23	19.11	51.66%

Table 3: Omni-Path runtime benchmarks

Parameters		Transport									
		Verbs					TCP				
Mode	**Parallelism**	**User**	**Sys**	**Idle**	**Time**	**CPU**	**User**	**Sys**	**Idle**	**Time**	**CPU**
Async	1	3.14	0.10	0.05	3.29	98.58%	6.04	7.22	0.02	13.28	99.85%
	16	3.10	0.18	0.04	3.33	98.70%	6.22	6.86	0.02	13.09	99.87%
	64	3.09	0.42	0.06	3.57	98.41%	5.92	7.56	0.02	13.50	99.85%
Sync	1	0.10	0.18	3.11	3.39	8.26%	0.06	2.78	12.66	15.51	18.38%
	16	0.10	0.21	3.02	3.33	9.30%	0.06	2.67	12.50	15.23	17.92%
	64	0.10	0.42	3.04	3.57	14.75%	0.04	4.09	9.23	13.36	30.55%

Table 4: InfiniBand runtime benchmarks

Definitions

User	Seconds spent in userspace
Sys	Seconds spent in kernel space
Idle	Seconds waiting for data without CPU involvement
Time	Total wall-clock runtime
CPU	The percentage of time the CPU was active during the test

C Endnotes

1. The term “fabric” can often be seen when describing computer networks in general. In the context of this report, however, it is mostly used to refer to high-performance computer networks.

2. The example is simplified. If built with this capability, Libfabric supports emulation of RDM endpoints for DGRAM (datagram) and MSG (connection-oriented) endpoints using the RXD [Opej] and RXM [Opek] utility providers; often these are also returned as layered providers, e.g “verbs;ofi_rxm”.

3. It is possible to use a counter instead of a CQ, which only provides the number of completed transfers, not which transfers have completed. The reasons to choose a CQ over a counter object or vice versa are complex and out of the scope of this report.

4. Operating on graphics memory or the memory of other devices in the system is also possible, but outside of the scope of this report. See GPUDirect RDMA [NVIb] for details.

5. Modern RDMA-capable network adapters have a feature known as On-Demand-Paging, where explicit memory registration is not required. However, there are certain limitations and issues to consider before the use of this feature. Details of this feature are outside of the scope of this report. See [NVIa] for details.

6. In particular, the architectures (x86 vs. ARM) as well as the specifics of each system’s design makes direct comparison beyond rough trends difficult. Factors such as system memory bandwidth limitations, slower interconnect bandwidths, CPU performance, and varied inter-core/inter-chip latency can affect gathered results. Therefore, systematic errors will manifest differently on each system.

7. In testing, the Libfabric PSM2 transport does support synchronous operation to some extent, even though this is not a recommended configuration. However, the latency can be extremely unreliable and tests have spuriously failed to complete during benchmark runs. According to the Libfabric documentation [Intb], “The psm2 provider requires manual progress. The application is expected to call fi_cq_read or fi_cntr_read function from time to time when no other libfabric function is called to ensure progress is made in a timely manner. The provider does support auto progress mode. However, the performance can be significantly impacted if the application purely depends on the provider to make auto progress.”.