Evaluation of Low-Level Libraries to Leverage the Performance of InfiniBand in Java Applications

Stefan Nothaas
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
stefan.nothaas@hhu.de

Fabian Ruhland
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
fabian.ruhland@hhu.de

Michael Schoettner
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
michael.schoettner@hhu.de

***Abstract*—Low latency network interconnects, such as InfiniBand, are commoly used in HPC centers and even accessible and affordable with todays cloud providers offering equipped instances for rent. This allows many application domains with distributed applications including web-based highly interactive applications like social networks, search engines, simulations or big-data analytics to run their systems and frameworks on low-latency capable hardware. But, with many of the used backend systems/frameworks written in Java, the JVM environment alone does not provide interfaces to utilize InfiniBand networks. In this paper, we present and evaluate the currently available (and supported) "low-level" solutions to utilize InfiniBand in Java applications. We created the "Java InfiniBand Benchmark" suite to evaluate socket- and verbs-based libraries using typical network microbenchmarks regarding throughput and latency. The benchmarks are executed on currently typical hardware configurations with 56 Gbit/s and 100 Gbit/s InfiniBand NICs. With transparency often traded for performance and vice versa, the results presented serve as a reference to help developers studying the pros and cons of each solution and support them in their decision which solution is more suitable for their use-case.**

***Index Terms*—High-speed networks, Distributed computing**

I. Introduction

RDMA capable devices have been providing high throughput and low-latency to HPC applications for several years [18]. With todays cloud providers offering instances equipped with InfiniBand for rent, such hardware is available to a wider range of users without the high costs of buying and maintaining it [25]. Many application domains such as social networks [20], [29], [31], search engines [24], [35], simulations [36] or online data analytics [21], [40], [41] require large processing frameworks and backend storages. Many of these are written in Java, e.g. big-data batch processing frameworks [28], [33], databases [4], [5] or backend storages/caches [6]–[8], [34]. These applications benefit from the rich environment Java offers including automatic garbage collection and multi-threading utilities. But, the choices for inter node communication on distributed applications are limited to Ethernet based socket interfaces (standard ServerSocket or NIO) on the commonly used JVMs OpenJDK and Oracle. They do not provide support for low latency InfiniBand hardware but there are various external solutions available each with its advantages and drawbacks. This raises various questions if a developer wants to chose a suitable solution for a specific use-case or application: What's the throughput/latency on small/large payload sizes? Is the performance sufficient when trading it for transparency requiring less to no changes to the existing code? Is it worth considering an implementation using the native API to gain maximum control with chances to harvest the performance available by the hardware?

In this paper, we address these questions by evaluating the currently existing and supported solutions that allow Java applications to leverage the performance of InfiniBand hardware. With different advantages/drawbacks as well as limitations, we evaluate the three socket-based libraries and implementations, IP over InfiniBand, libvma and JSOR, as well as two verbs-based implementations, native C-verbs and jVerbs. There are also full higher level network subsystems offering higher level primitives for messaging or message passing, e.g. MPI, but are not the focus of this paper. We discuss and evaluate these in a seperate publication (**TODO ref ibdxnet paper**). We implemented a benchmark suite with typical network microbenchmarks to analyze the throughput on uni- and bi-directional communiation as well as the uni-directional latency using a ping-pong benchmark. The results presented cannot provide a general recommendation but instead serve as an important foundation to determine a suitable solution to leverage InfiniBand in different types of Java applications.

The remaining paper is structured as follows: Section II discusses related work with socket-based (§II-A) and verbs-based (§II-B) libraries. Section III presents the Java InfiniBand Benchmark Suite which is used to evaluate two verbs-based solutions and three socket-based solutions in the following Section IV regarding uni-directional (§IV-B) and bi-directional (§IV-C) throughput, as well as one-sided latency (§IV-D) and full round-trip-time using a ping-pong benchmark (§IV-E). Conclusions are presented in Section V.

II. Related Work

This section elaborates on existing "low-level" solutions/libraries that can be used to leverage the performance of InfiniBand hardware in Java applications. This does not include network or messaging stacks/subsystems implementing higher-level primitives such as the Massage Passing Interface like the Java-based FastMPJ [22] which also provides a special transport to use InfiniBand hardware. We address the different

low-level solutions in two categories: Socket-based wrappers or injected libraries (§II-A) and verbs-based libraries (§II-B).

A. Socket-based Libraries

The **socket-based libraries redirect the traffic of send and receive calls of socket-based applications transparently over InfiniBand host channel adapters (HCAs)** with or without kernel bypass depending on the implementation. Thus, existing applications do not have to be altered to benefit from improved performance due to the lower latency hardware compared to commonly used Gigabit Ethernet. The following three libraries are still supported to date and evaluated in Section IV.

IP over InfiniBand (IPoIB) [27] is not a library but actually a kernel driver that exposes the InfiniBand device as a standard network interface (e.g. *ib0*) to the user space. Socket-based applications do not have to be modified to send/receive data but use the specific interface. However, the driver uses the kernel's network stack and, thus, has to execute context switching between user and kernel space, and the CPU is more involved when handling data. Naturally, this solution trades performance for transparency.

libvma [10] is a library developed by Mellanox and included in their OFED software package [11]. It is pre-loaded to any socket-based application (using *LD_PRELOAD*). It enables full bypass of the kernel network-stack by redirecting all socket-traffic over InfiniBand using unreliable datagram with native C-verbs. Again, the existing application code does not have to be modified to benefit from increased performance.

Java Sockets over RDMA (JSOR) [39] redirects all socket-based data traffic in Java applications using native verbs, similar to libvma. It uses two paths for implementing transparent socket streams over RDMA devices. The "fast data path" uses native verbs to send and receive data and the "slow control path" is used for managing RDMA connections. JSOR is developed by IBM for their proprietary J9 JVM and, thus, not available on other JVMs.

The following libraries are also known in literature but are not supported or maintained anymore.

The **Sockets Direct Protocol (SDP)** [23] redirects all socket-based traffic of Java applications over RDMA with kernel-bypass. It supported all available JDKs since Java 7 and was part of the OFED package until it was removed with OFED version 3.5 [12].

Java Fast Sockets (JFS) [38] is an optimized Java socket implementation for high speed interconnects. It avoids serialization of primitivate data arrays and reduces buffering and buffer copying with shared memory communication as its main focus. However, JFS relies on SDP (deprecated) for using InfiniBand hardware.

Speedus [17] is a native library that optimizes data transfers for applications especially on intrahost and intercontainer communication by bypassing the kernel's network stack. It is also advertised to support low-latency networking hardware for internode communication but the latest available version to date (2016-09-08) does not include such support.

B. Verb-based Libraries

Verbs are an abstract and low-level description of functionality for RDMA devices (e.g. InfiniBand) and how to program them. Verbs define the control and data paths including RDMA operations (write/read) as well as messaging (send/receive). RDMA operations allow **reading or writing directly from/to the memory of the remote host without involving the CPU of the remote** at all. Messaging follows a more traditional approach by providing a buffer with data to send to the remote host and the remote has to provide a buffer to receive the data to.

The programming model heavily differs from traditional socket-based programming. Using different types of asynchronous queues (send, receive, completion) as communication endpoints. The application uses different types of work-requests for sending and receiving data. When handling data to transfer, all communication with the HCA is executed using these queues. The following libraries are verbs implementations that allow the user to program the RDMA capable hardware directly. The first two libraries presented are evaluated in Section IV.

C-verbs are the native verbs implementation included in the OFED software package [13]. Using the Java Native Interface (JNI) [30], this library can be utilized in Java applications as well in order to create a custom network subsystem [22] (**TODO ref ibdxnet**). Using the Unsafe class [32] or Java DirectByteBuffers, one can easily allocate memory off-heap which must be used for sending and receiving data using InfiniBand hardware (buffers must be registered with a protection domain which pins the physical memory).

jVerbs [37] are a proprietary verbs implementation for Java developed by IBM for their J9 JVM. Using a JNI layer, the OFED C-verbs implementation is accessed but not just wrapped. Instead, "stateful verb methods" (*StatefulVerbsMethod* Java objects) encapsulate the verb to call including all parameters. The parameters are serialized to native space before the verb is executed. Once the object is prepared, it can be executed which actually calls the native verb. These objects can be re-used for further calls with the same parameters to avoid repeated serialization of parameters to native space and creating new objects which burdens garbage collection.

Jdib [26] is a library wrapping native C-verbs function calls and exposing them to Java using a JNI layer. According to the authors, various methods, e.g. queue pair data exchange on connection setup, are abstracted to create an easier to use API for Java programmers. The fundamental operations to create protection domains, create and setup queue pairs, as well as posting data-to-send to queues and polling the completion queue seem to wrap the native verbs and do not introduce additional mechanisms like jVerbs's stateful verb calls. We were not able to obtain a copy of the library for evaluation.

III. JIB-Benchmarks: A Benchmark Suite for Evaluating InfiniBand Libraries for Java

To evaluate and compare the different libraries available to utilize InfiniBand hardware in Java applications, a common set

of benchmarks had to be implemented for each of the libraries. Existing solutions like the iperf [9] tools for TCP/UDP or the ibperf tools included in the OFED package [13] cannot cover all libraries we want to evaluate and do not implement all necessary benchmark types. In this paper, we want to evaluate the still supported and available libraries (§II) in a fundamental **point-to-point setup** regarding throughput and latency. Like other benchmark suits (e.g. OSU point-to-point microbenchmarks [14]), we want to determine the **maximum throughput** on **uni-directional** and **bi-directional** communication (e.g. application pattern asynchronous "messaging"), as well as **one-sided latency** and **full round-trip-time** (RTT) with a **pingpong communication** pattern (e.g. application pattern "request-response"). These benchmarks allow us to determine the fundamental performance of the presented solutions and are commonly used to evaluate network hardware or applications [9], [13], [14]. The *JIB-Benchmark* (Java InfiniBand Benchmark) suite provides implementation of the listed benchmarks for two verbs-based solutions (c-verbs, jVerbs) and three socket-based solutions (IPoIB, libvma, JSOR). It is open source and available at Github [1].

Each benchmark run is configurable using command line parameters such as the benchmark type (uni-/ bi-directional, one-sided latency or ping-pong), the message size to send/receive and the number of messages to send/receive. For convenience, we refer to the payload size sent as messages independent of how it is transfered (e.g. sockets, verbs RDMA or verbs messaging). The context, all buffers and other allocations are executed before the benchmark is started. Afterwards, the current instance connects to the remote specified via command line parameters. Once the connection is established, a dedicated thread is started for sending data and another thread for receiving. Today, we can expect this to run on a multicore system with at least two physical cores to ensure that the send and receive thread can be run simultaniously to avoid blocking one another. The benchmark instance sends the specified number of messages to the remote and measures the time it takes to send the messages. Furthermore, we utilize the performance counters of the InfiniBand HCA to determine the overhead added by any software defined protocol which is especially relevant for the socket-based libraries (see Section IV-A).

For socket-based libraries, the benchmark is implemented in Java using TCP sockets with the ServerSocket, DataInputStream and DataOutputStream classes. Sending and receiving data is executed synchronously in a single loop on each thread. No further adjustments are required because all libraries redirect the normal send and receive calls of the socket libraries. With IPoIB, we use the address of the exposed *ib0* device. The libvma library is pre-loaded to the benchmark using *LD_PRELOAD*. In order to use JSOR, we run the benchmark in the J9-JVM and provide a configuration file specifying IP-addresses whose traffic is redirected over the RDMA device.

The verbs-based benchmarks are implemented in C and Java. Both implementations use RC queue pairs for RDMA and message operations. UD queue pairs can also be used for message operations but this option is currently not implemented. On RDMA operations, we did not include immediate data with a work request which would require a work completion on the remote. This is often used to allow signaling the remote on incoming RDMA operations but is optional and not desired on transparent remote operations and for determining the maximum performance. When sending RDMA operations to the HCA to determine the maximum throughput, we do not repeatetly add one work request to the send queue, then poll the completion queue waiting for that single work request to complete. This pattern is commonly used in examples [16] and even larger applications [15] but does not yield optimal performance because the queue of the HCA runs empty very frequently. To keep the HCA busy, the send queue must be kept filled at all times. Thus, we fill up the send queue to the maximum size configured, first. Then, we poll the completion queue and once at least one completion is available and processed, we immediately fill the send queue again. Naturally, this pattern cannot be applied to the ping-pong latency benchmark executing a request-response pattern.

This data path is implemented in both, the C-verbs and jVerbs implementation. The C implementation uses the verbs implementation included in the OFED package and serves as a reference for comparing the maximum possible performance. To establish a remote connection, queue pair information is exchanged using a TCP socket. The jVerbs implementation has to implement the operations of the data path using the previously described stateful verbs methods. Thus, the sending of data on the throughput benchmark had to be altered slightly. A single stateful verb call for posting work requests to the send queue always posts 10 elements. Hence, new work requests are added to the send queue once at least 10 work completions were polled from the completion queue. We create all stateful verbs calls before the benchmark and re-use them to avoid performance penalties. On connection creation, queue pair information is exchanged with the API provided by jVerbs which is using the RDMA connection manager.

IV. Evaluation

In this Section, we present the results of the evaluation of the socket-based libraries/implementations IPoIB, libvma and JSOR and the verbs-based libraries C-verbs and jVerbs using our benchmark suite (§III). We analyze and discuss basic performance metrics regarding throughput and latency using typical network benchmarks with a two node setup with 56 Gbit/s and 100 Gbit/s interconnects. In Section IV-A, we analyze and discuss the network and protocol overhead of the different libraries. The results of the uni-directional throughput benchmark are presented in Section IV-B and for bi-directional communication in Section IV-C. Section IV-D presents the results of the one-sided latency benchmark and Section IV-E the results of the ping-pong benchmark to determine full round-trip-times. Due to space constraints, we limit the discussion of the results to selected conspicuities of the plotted data.

We use the term “message” to refer to the unit of transfer which is equivalent to the data payload. The size of a message does refer to the payload size only and does not include any additional protocol or network layer overhead. Each throughput focused benchmark run transfers 100 million messages and each latency focused benchmark run transfers 10 million messages. We evaluated payload sizes of 1 byte to 1 MB in power-of-two-increments. When discussing the results, we usually focus on the message rate on small messages with payload sizes less than 1 kB and on the throughput on middle sized and large messages starting at 1 kB.

The throughput results are depicted as line plots with the left y-axis describing the throughput in million messages per second (mmps) and the right y-axis describing the throughput in MB/s. For the latency results, the left y-axis describes the latency in µs and the right y-axis the throughput in mmps. The dotted lines always represent the message throughput while the normal lines represent either the throughput in MB/s or the latency in µs, depending on the benchmark. For the overhead results, a single y-axis describes the overhead in percentage in relation to the amount of payload transfered on a logarithmic scale. On all plot types, the x-axis depicts the size of the payload in power-of-two increments from 1 byte to 1 MB. Each benchmark run was executed three times and the min and max as well as average of the three runs are visualized using error bars.

The following releases of software were used for compiling and running the benchmarks: Java 1.8, OFED 4.4-2.0.7, libvma 8.7.5, IBM J9 VM version 2.9, gcc 8.1.0. We ran our experiments on two setups with two nodes each connected with an InfiniBand fabric with the following specifications:

1) Mellanox ConnectX-3 HCA, 56 Gbit/s InfiniBand, MTU size 4096, Dual socket Intel Xeon E5-2697v2 (2.7 GHz) 12 core CPUs, 128 GB RAM, CentOS 7.4 with the Linux Kernel version 3.10.0-693
2) Mellanox ConnectX-5 HCA, 100 Gbit/s InfiniBand, MTU size 4096, Dual socket Intel Xeon Gold 6136 (3.0 GHz) 12 core CPUs, 128 GB RAM, CentOS 7.4 with the Linux Kernel version 3.10.0-693

When using libvma, flow steering must be activated to redirect all traffic over InfiniBand correctly. This must be enabled in the InfiniBand kernel module by setting the parameter *log_num_mgm_entry_size* to *-1* in the configuration file */etc/modprobe.d/mlnx.conf*. Otherwise, libvma falls back to sockets over Ethernet.

In the following subsections, we focus the analysis and discussion on the results with 100 Gbit/s hardware. Figures depicting the results with 56 Gbit/s hardware are also included but not further discussed if they do not differ significantly. Results of IPoIB are labeled as ”JSocketBench(msg)“ in all following figures.

A. Overhead

In this Section, we present the results of the overhead measurements of the described libraries/implementations. As overhead, we consider the additional amount of data that is sent along with the payload data of the user. This includes any data of any network layer down to the HCA. We measured the amount of data emitted by the port using the performance counter *port_xmit_data* of the HCA.

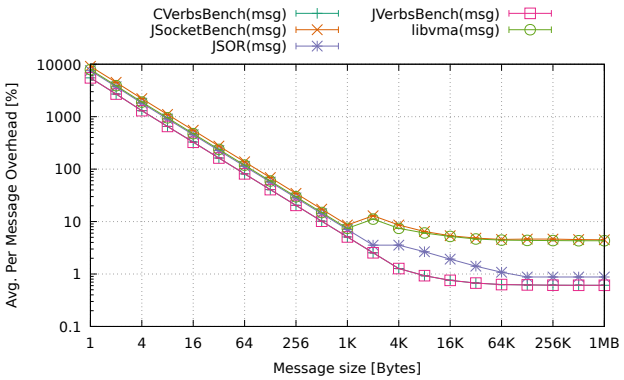

Fig. 1. Average per message overhead in percentage in relation to the payload size transfered (ping-pong benchmark).

IPoIB and libvma are implementing buffer/message aggregation when sending data. Applications on high load sending many small messages benefit highly from increased throughput and the overall per message overhead is lowered. However, in order to determine the general per message overhead, we used the pingpong benchmark which does not allow aggregation due to its nature. To save space, the results of both types (sockets/verbs) are depicted in a single figure.

We try to give a rough breakdown of the overhead involved with each method evaluated. A precise breakdown is rather difficult with just the raw amount of data captured from the ports as re-transmission of packages are also captured (e.g. RC queue pairs or custom protocols based on UD queue pairs).

The results show that the overhead for msg operations of C-verbs and jVerbs are identical. For a single byte of payload, an additional 54 bytes are required which corresponds to two 27 byte headers which are part of the low-level InfiniBand protocol. Required by the RC protocol, one package is used for sending the ping and the other package to receive the pong. The metadata consists of a local routing header (8 bytes), base transport header (12 bytes), invariant CRC (4 bytes) and variant CRC (2 bytes) [19]. This makes a total of 26 bytes which is close to the measured 27 bytes (errors due to *port_xmit_data* yielding values in octets). For RDMA operations, an additional RDMA extended transport header (16 bytes) is added which makes a total of 42 bytes of “metadata“ for such a paket. Naturally, this overhead cannot be avoided. As expected with jVerbs using the native verbs directly without adding another software protocol layer, the overhead added is identical to C-verbs’s. The overhead stays constant which leads to an overall decreasing per message overhead with increasing payload size. Starting with 8 kB payload size, the overhead ratio drops below 1%.

The overhead of the socket-based solutions is overall slightly higher. Again, considering 1 byte messages, JSOR

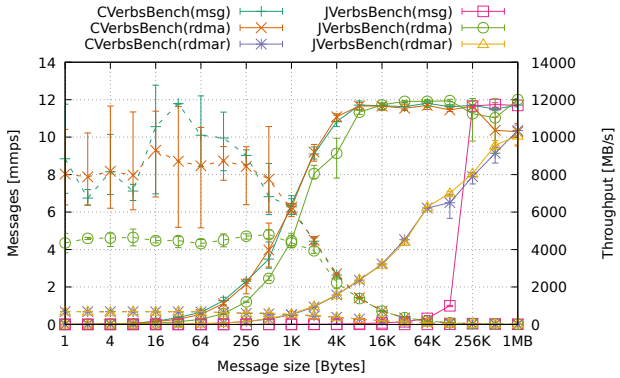

Fig. 2. Throughput results of the **uni-directional** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**100 GBit/s** hardware).

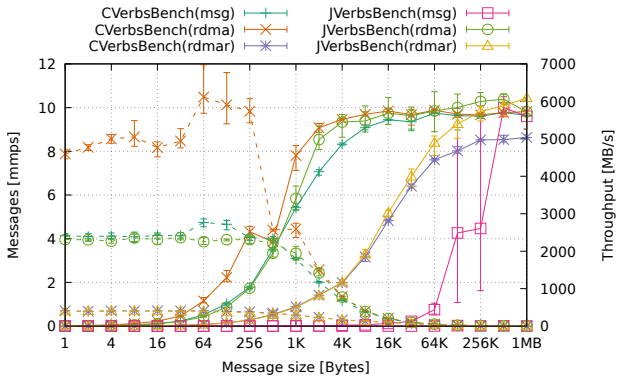

Fig. 3. Throughput results of the **uni-directional** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**56 GBit/s** hardware).

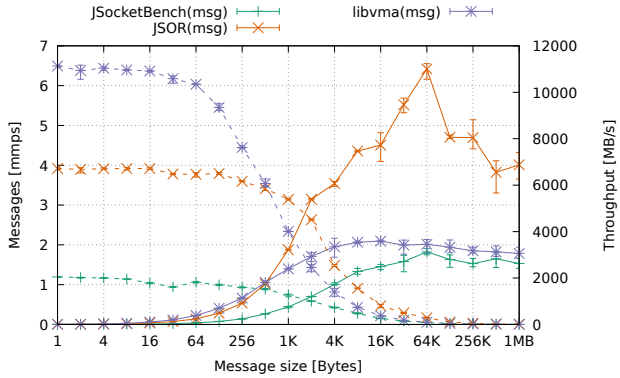

Fig. 4. Throughput results of the **uni-directional** benchmark using **socket-based** libraries with increasing message size (**100 GBit/s** hardware).

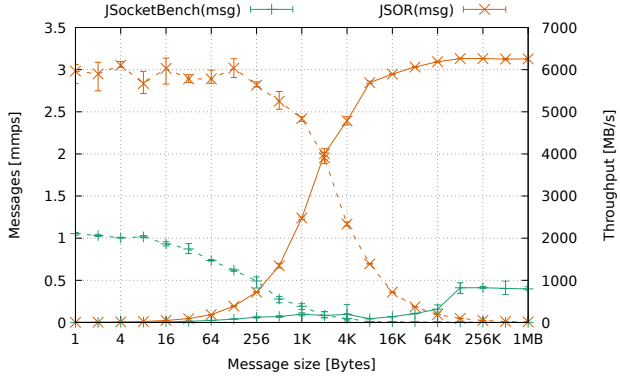

Fig. 5. Throughput results of the **uni-directional** benchmark using **socket-based** libraries with increasing message size (**56 GBit/s** hardware).

adds an additional 7500 %, libvma 7900 % and IPoIB 9100 % overhead to each pingpong transfer. libvma and IPoIB rely on UD messaging verbs which add a datagram extended transport header (8 bytes) to the InfiniBand header and include additional information to allow IP-address based routing of the packages. The IPoIB specification describes an additional header of 4 octets (4 bytes) and IP header (e.g. IPv4 20 btyes + 40 bytes optional) which are added alongside the message payload [27]. libvma adds an IP-address (4 bytes) and Ethernet frame header (14 bytes) [10]. Remaining data is likely committed towards a software signaling protocol. Regarding JSOR, we could not find any information about the protocol implemented (closed source).

B. Uni-directional Throughput

This Section presents the throughput results of the uni-directional benchmark. Starting with the verbs-based results depicted in Figure 2, jVerbs RDMA write message throughput (4.3 - 4.6 mmps) is about half of C-verbs's RDMA write throughput (7.9 - 9.3 mmps) for small payload sizes up to 512 byte. The RDMA write performance of C-verbs is nearly double the throughput of C-verbs messaging but with high jitter. When increasing the payload size starting at 1 kB, jVerbs's RDMA write throughput stays clearly below both C-verbs's RDMA write and message send throughput. Interesting to note that C-verbs's messaging is significantly better, though highly jittery, on small messages up to 512 bytes and middle sized messages up to 4 kB. Both C-verbs operations saturate throughput with 8 kB payload size and low jitter at around 11.7 GB/s. We could not determine the reason for the very poor performance of jVerbs's msg verbs on both 56Gbit/s and 100Gbit/s hardware.

In comparison, the results of socket-based libraries are depicted in Figure 4. On small payload sizes up to 256 bytes, IPoIB achieves a message throughput of approx. 1 - 1.2 mmps. With increasing payload size, the throughput starts saturating at 32 kB message size and peaks at 64 kB with 3.1 GB/s throughput. The results of libvma show an highly increased throughput of 6.0 to 6.5 mmps for up to 64 byte messages. Overall throughput for middle sized and large messages surpasses IPoIB's peaking at 3.5 GB/s with 8 kB messages but also starting to decrease down to 3.0 GB/s when increasing the message size up to 1 MB. JSOR achieves a significantly lower throughput of 3.8 - 3.9 mmps for up to 128 byte messages. However, JSOR provides a much higher throughput starting at 1 kB message size compared to IPoIB and libvma. Throughput peaks at 64 kB message

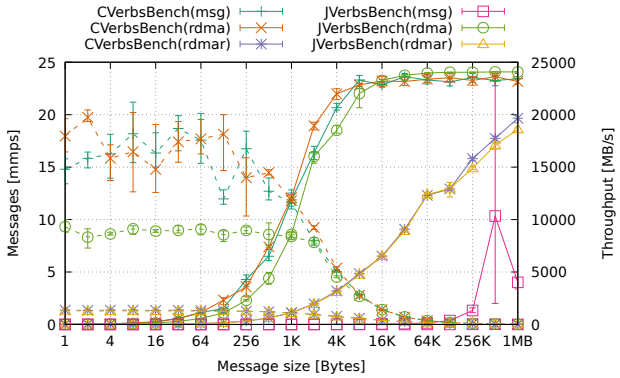

Fig. 6. Throughput results of the **bi-directional** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**100 GBit/s** hardware).

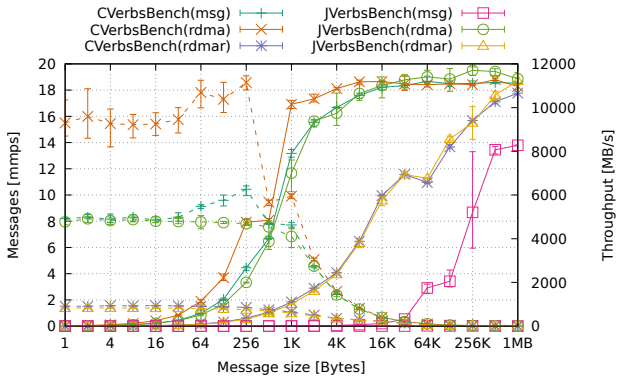

Fig. 7. Throughput results of the **bi-directional** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**56 GBit/s** hardware).

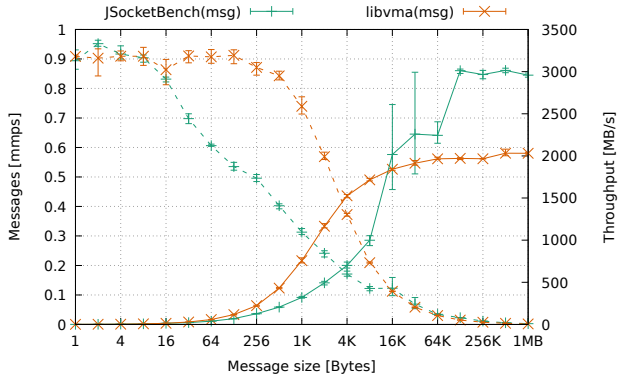

Fig. 8. Throughput results of the **bi-directional** benchmark using **socket-based** libraries with increasing message size (**100 GBit/s** hardware).

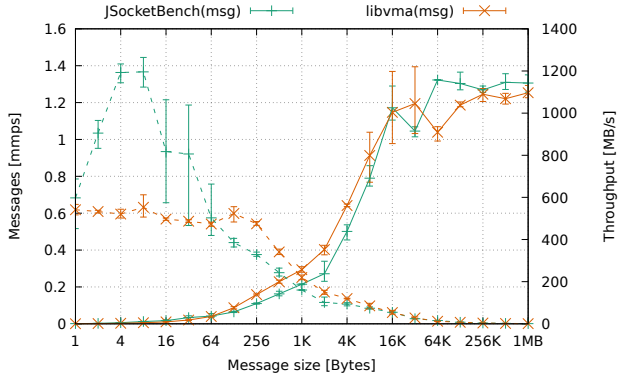

Fig. 9. Throughput results of the **bi-directional** benchmark using **socket-based** libraries with increasing message size (**56 GBit/s** hardware).

size with 11 GB/s but drops down to approx 6.5 GB/s with 512 kB messages afterwards. As described in Section IV, we determined that JSOR's performance degrades considerable on payload sizes of 128 kB and greater which required us to increase the RDMA buffer size (to 1 MB). But, even with an increased buffer size, this problem could not be resolved entirely. The results on 56Gbit/s hardware (see Figure 5 do not include results of libvma because the benchmarks failed repeatetly with a non fixable "connection reset" error by libvma.

C. Bi-directional Throughput

This Section presents the throughput results of the bi-directional benchmark. With InfiniBand supporting full-duplex communication, we expect roughly double the throughput of the uni-directional results in general. Figure 6 depicts the results of the verbs-based implementations and, as expected, all implementations show roughly double the message rate on small messages and roughly double the throughput on large messages compared to the uni-directional results (§IV-B). Regarding overall performance, C-verbs RDMA writes are yielding the best performance here (15 - 17 mmps for 1 - 128 byte payload size, 11.5 GB/s peak performance at 32 kB payload size) followed by C-verbs messages (8.3 - 8.9 mmps for 1 - 64 bytes, 11.5 GB/s peak performance at 64 kB payload size) and jVerbs RDMA writes with a significant worse performance on payload sizes up to 1 kB (3.3 - 3.8 mmps) but slightly exceeding both C-verbs implementations with a peak performance of 11.6 GB/s at 256 kb payload size. Again, it is notable that the throughput of C-verbs RDMA writes jitters on small payloads as well as throughput of jVerbs for middle sized payloads (2 kB to 64 kB). The incomprehensible poor performance of jVerbs msg verbs is also present here.

Figure 8 depicts the socket-based results of the bi-directional benchmark. Due to unresolvable errors causing disconnects, especially on message sizes smaller 512 bytes, we cannot provide results for JSOR. This seems to be a known problem [2] but increasing the send and receive queue sizes as described does not resolve this issue. Furthermore, the benchmark does not terminate anymore on message sizes greater 32 kB. The proposed solution to increase the buffer size does not resolve this problem either [3].

The results available show a very low overall performance for IPoIB and libvma compared to their results on the uni-directional benchmark (§IV-B). IPoIB achives an aggregated throughput of 0.89 to 0.95 mmps for just up to 16 byte

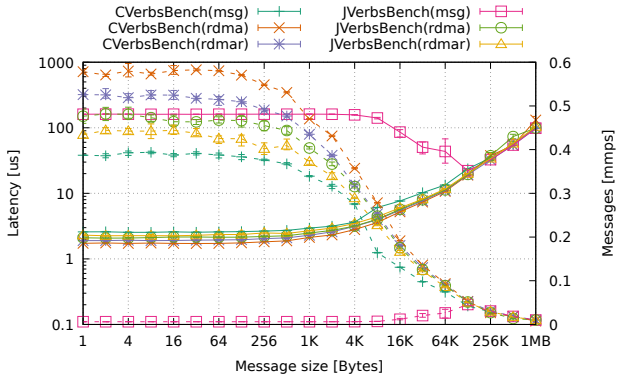

Fig. 10. Average latency results of the **one-sided latency** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**100 GBit/s** hardware).

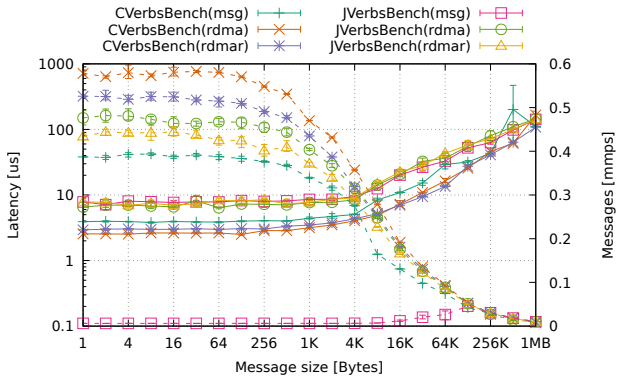

Fig. 11. 99.9th percentile results of the **one-sided latency** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**100 GBit/s** hardware).

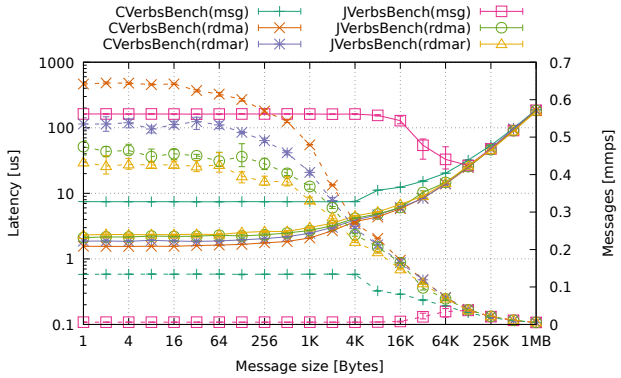

Fig. 12. Average latency results of the **one-sided latency** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**56 GBit/s** hardware).

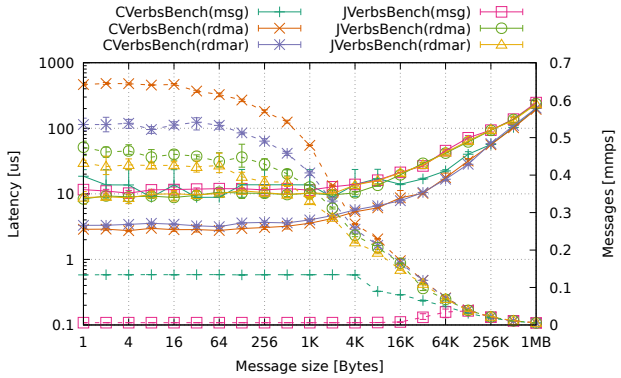

Fig. 13. 99.9th percentile results of the **one-sided latency** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**56 GBit/s** hardware).

messages with a considerable drop in performance for small messages afterwards. But, throughput increases with increasing message size starting middle sized messages saturating and peaking at 128 kB message with 3.0 GB/s aggregated throughput. Further notable are high flucatuations for 16 kB to 64 kB. On small messages, libvma can at least provide a constant performance for small messages up to 128 bytes with 0.9 mmps. Throughput increases with increasing message size with saturation starting at approx. 32 kB messages with 1.9 GB/s aggregated throughput. A lower Peak performance than IPoIB is rached at 512 kB messages with 2.0 GB/s.

D. One-sided Latency

This Section presents the results of the one-sided latency benchmark to determine the latency of a single operation. Section IV-E further discusses full RTT latency for a ping-pong communication pattern. Results are separated by socket-based and verbs-based again and include the average latency as well as the 99.9th percentiles.

Figure 10 shows the average latencies and Figure 11 the 99.9th percentiles determined for the verbs-based benchmarks. The results of all native verbs-based communication as well as jVerbs RDMA write and read are as expected providing a low average close to 1 µs latency. From lowest latency to ”highest“: c-verbs RDMA write, c-verbs RDMA read, jVerbs RDMA write, jVerbs RDMA read and c-verbs msg. As expected, jVerbs adds some overhead leading to a minor increase in latency. But, the average latency results of jVerbs msg are unexpected. Up to 4 kB message size, which equals the used MTU size, the latency is very high and constant at approx. 160 µs. With further increasing message size, the latency is lowered and approximates the average latencies of the other transfer methods. At 128 kB message size, it goes along perfectly with the other results. This is also present on the results on 56 Gbit/s hardware for jVerbs msg but also for C-verbs msg (see Figure 12) wherelse the C-verbs msg results on 100 Gbit/s hardware are fine.

To further analyze this issue, we depicted the 99.9th percentiles in Figure 11. Again, the other five transfer methods are showing expected behviour and overall low latency. But, jVerbs msg also shows very low latencies around 8 µs latency for 99.9th of all messages transfered. Thus, just a small amount of messages (10,000 worst out of 10 million) yields latencies higher than 8 µs. We further analyzed the 99.99th percentiles (i.e. 1,000 worst out of 10 million) which results in a latency of

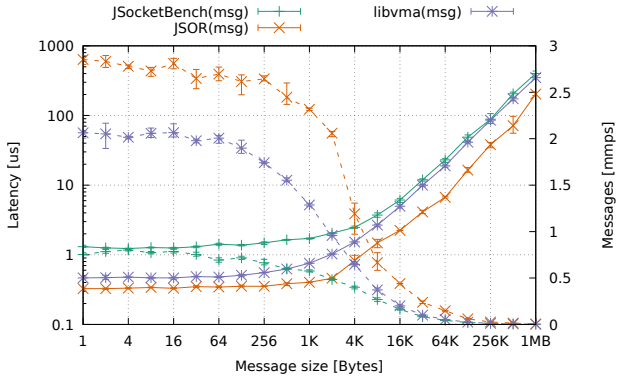

Fig. 14. Average latency results of the **one-sided latency** benchmark using **socket-based** libraries with increasing message size (**100 GBit/s** hardware).

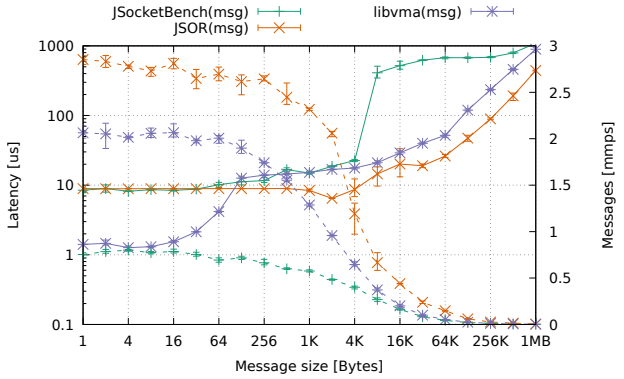

Fig. 15. 99.9th percentile results of the **one-sided latency** benchmark using **socket-based** libraries with increasing message size (**100 GBit/s** hardware).

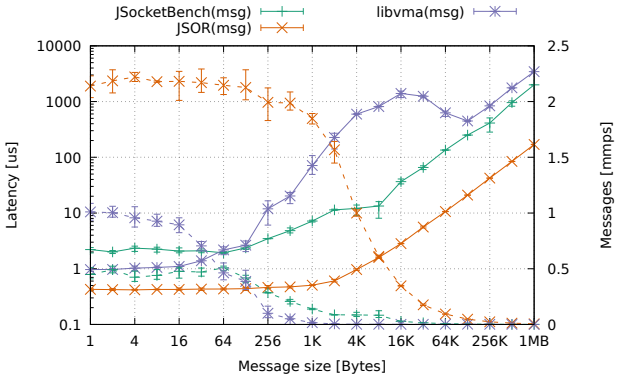

Fig. 16. Average latency results of the **one-sided latency** benchmark using **socket-based** libraries with increasing message size (**56 GBit/s** hardware).

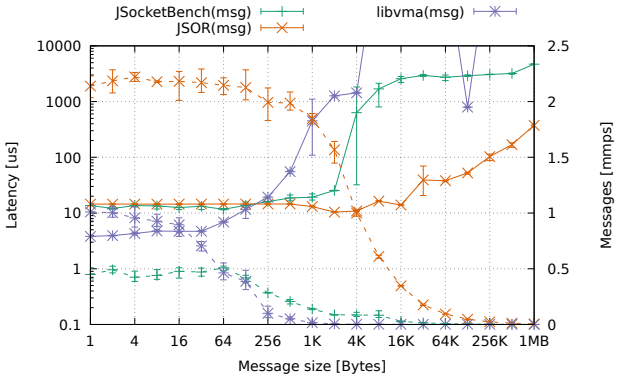

Fig. 17. 99.9th percentile results of the **one-sided latency** benchmark using **socket-based** libraries with increasing message size (**56 GBit/s** hardware).

approx. 1500 µs. This shows, that there is a very small amount of messages with very high latency causing a rather overall high average latency. This might be caused by receiver-not-ready errors on high loads which triggers an automatic retry after a specific delay when using RC queue pairs. However, that delay (*min_rnr_timer*) cannot be configured with jVerbs and it is not documented which value was chosen. Other configuration parameters such as queue sizes did not have any impact on the results. **TODO add sentence explaining possible hardware issues with bull chassis regarding c-verbs msg?**

The average latencies in Figure 14 show that JSOR performs best with an average per operation latency of 0.3 - 0.4 µs for up to 1 kB messages. With further increasing payload size, latency increases as expected. libvma shows similar results with a slightly higher latency of 0.4 - 0.5 µs for small messages. IPoIB follows with a further increased average latency of 1.3 to 1.5 µs for small messages. These results, especially JSOR's, seem unexpected low at first glance. But, when considering the socket-interface, it does not provide means to return any feedback to the application when data is actually sent to the remote. With verbs, one polls the completion queue and as soon as the work completion is received, it is guaranteed that the data is fully sent to the remote and received by the same. A socket send-call however, does not guarantee that the data is sent once it returns control to the caller. Typically, a buffer is used to allow aggregation of data before putting it on the wire. JSOR, libvma and IPoIB implement message aggregation before actually sending out any data. This is further proofen by the ping-pong benchmark which does not allow any aggregation to be applied by the backend (see Section IV-E). On 56 Gbit/s hardware (see Figure 16), JSOR and IPoIB show similar results with a slightly higher latency but libvma shows significantly worse results for 256 byte to 64 kB message sizes compared to running on 100 Gbit/s hardware.

The 99.9th percentiles in Figure 15 show further interesting aspects not just limited to message aggregation. libvma starts with very low 99.9th latencies for up to 16 byte messages indicating a rather low threshold for aggregation benefitting small messages by keeping the total of worst latencies low. However, with 16 to 128 byte messages, the latency increases considerably. JSOR and IPoIB show similar results for small messages up to 1 kB. JSOR's stays even constant with 9 µs up to 512 byte messages. This indicates a flush threshold based on the number of messages instead of a total buffer size. But, the latency of IPoIB starts to increase already starting with 64 byte message size and even jumps significantly up to over 400

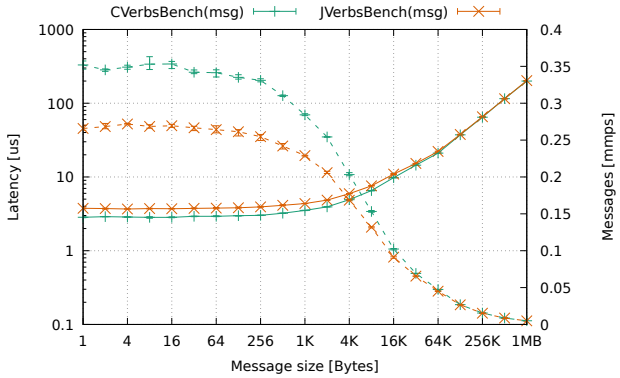

Fig. 18. Average latency results of the **ping-pong** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**100 GBit/s** hardware).

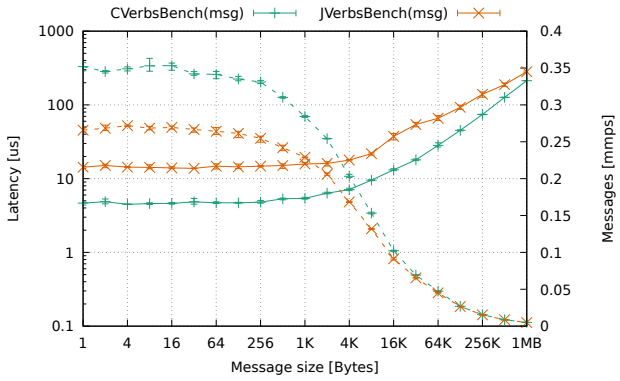

Fig. 19. 99.9th percentile results of the **ping-pong** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**100 GBit/s** hardware).

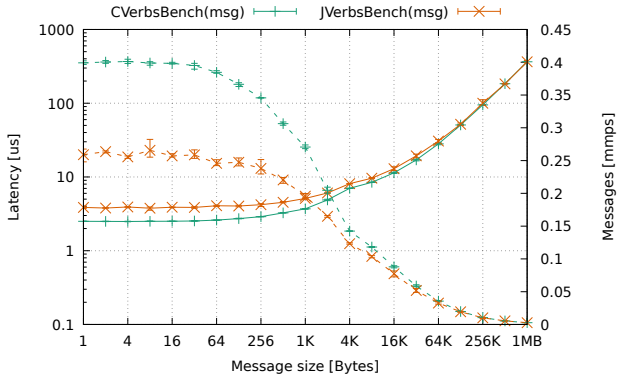

Fig. 20. Average latency results of the **ping-pong** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**56 GBit/s** hardware).

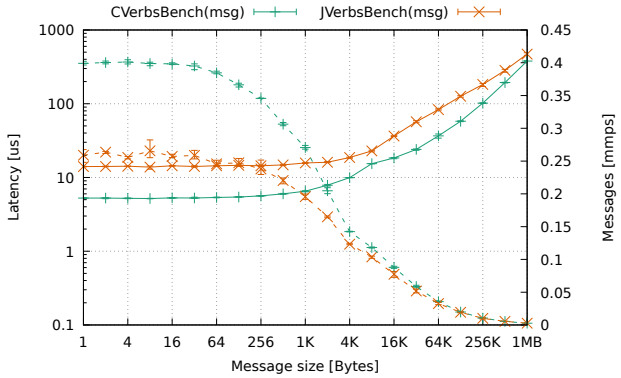

Fig. 21. 99.9th percentile results of the **ping-pong** benchmark using **verbs-based** libraries with different transfer methods and increasing message size (**56 GBit/s** hardware).

µs with 8 kB message size. This might indicate that additional allocations are involved for large(r) messages increasing the overall worst latencies significantly. On 56 Gbit/s hardware, JSOR's and IPoIB's results are similar with a slightly higher latency but libvma's 99.9th percentile latency is significantly increasing with 256 byte message size and even reaching 600 ms on multiple sizes greater 4 kB.

E. Ping-pong Latency

In this last section, we present the results of the ping-pong latency benchmark. Due to the nature of the communication pattern, the methods of transfer are limited to messaging operations for verbs-based implementations. Using RDMA operations is also possible but requires additional data structures and control logic which is currently not implemented. The average latencies, i.e. full round-trip-times, are depicted in Figure 18. C-verbs messaging achieves an average latency of 2.8 to 3.2 µs for up to 512 byte messages. jVerbs shows similar behaviour but with a slightly higher latency of 3.7 to 4.1 µs. Further increasing the message size of both, C-verbs and jVerbs increases the latency as expected. The 99.9th percentiles, depicted in Figure 19, show a similar picture with slightly higher latencies of 4.6 to 4.8 µs for C-verbs and 14.2 to 15.0 µs for small messags up to 512 bytes. On 56 Gbit/s hardware (see Figures 20 and 21), both implementations show similar results.

The results of the socket-based methods are depicated in Figure 22 for the average latencies and in Figure 23 for the 99.9th percentiles. Both, JSOR and libvma show low average latencies of 2.3 to 3.5 µs and 3.7 to 5.3 µs for message sizes up to 512 bytes. With increasing message sizes, the average latency also increases as expected. A small latency "jump" of around 1 µs is notable on both libraries from 64 byte to 128 byte message size. IPoIB shows a similar behviour but with a significant higher averate latency of 42.7 to 43.2 µs up to 1 kB message size. The same latency "jump" can be seen from 1 kB to 2 kB message size. This increase of latency might be caused due switching to a different buffer size which might involve additional buffer allocation. The 99.9th percentile results show a similar picture but with increased latencies of 6.7 to 7.8 µs (JSOR), 7.3 to 9.2 µs (libvma) and 96.5 to 99.2 µs (IPoIB) for small messages up to 512 bytes. On 56 Gbit/s hardware (see Figure 24), libvma's average latency increases notably to up to 17 to 22 µs wheras IPoIB's latency decreased to 30 to 33 µs for up to 512 byte messages.

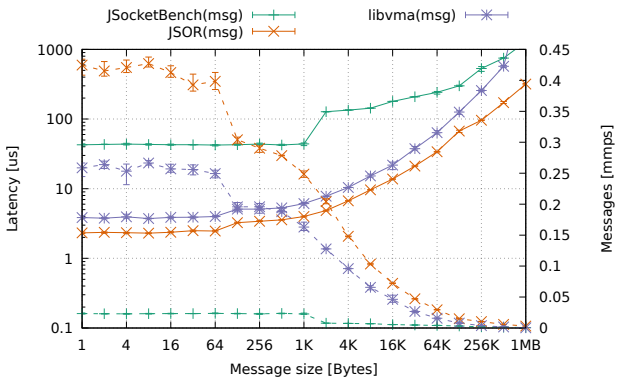

Fig. 22. Average latency results of the **ping-pong** benchmark using **socket-based** libraries with increasing message size (**100 GBit/s** hardware).

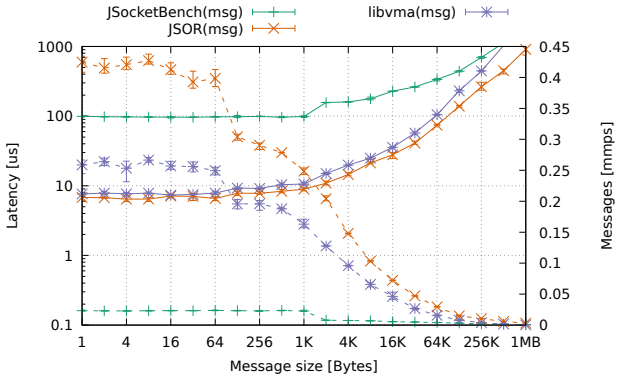

Fig. 23. 99.9th percentile results of the **ping-pong** benchmark using **socket-based** libraries with increasing message size (**100 GBit/s** hardware).

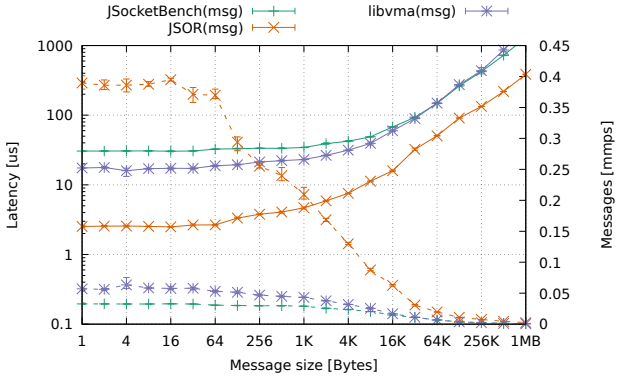

Fig. 24. Average latency results of the **ping-pong** benchmark using **socket-based** libraries with increasing message size (**56 GBit/s** hardware).

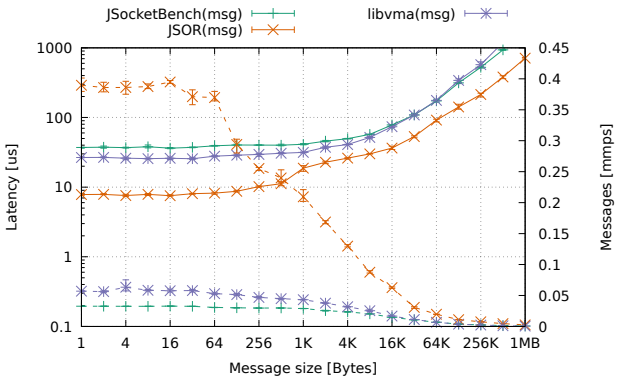

Fig. 25. 99.9th percentile results of the **ping-pong** benchmark using **socket-based** libraries with increasing message size (**56 GBit/s** hardware).

V. Conclusions

We presented our JIB-Benchmark suite and the evaluation results of three socket-based solutions and two verbs-based solutions to leverage InfiniBand in Java applications. The socket-based solutions provide a transparent solution requiring low effort to get additional performance from InfiniBand hardware for existing socket-based solutions without changing existing applications. But, this comes at the price that the full potential of the hardware cannot be exploited, especially on bi-directional communication. Compared to the performance of Gigabit Ethernet, latency is at least halved on 56 Gbit/s hadware and can even be as low as 2-5 µs for small messages. Regarding throughput, one can get an at least ten-fold increase and it is even possible to saturate 56 Gbit/s hardware on uni-directional communication.

To leverage the true potential of the hardware, the verbs-based solutions are a must. Overall, jVerbs is performing very well and brings nearly native performance on RDMA operations to the Java space with a few minor performance differences. But, the inexplicable poor performance of jVerbs messaging verbs does not allow any meaningful usage in applications. With C-verbs, the full potential of the hardware can be exploited on all communication methods. Thus, one has to decide whether to stay entirely in Java space but having to rely on the proprietary JV9 JVM or having the freedom to write a custom network subsystem using C-verbs with JNI which is more time consuming and challenging.

From the results and our experience so far, we consider libvma a good solutions to benefit from some of the performance of InfiniBand hardware without having to invest a signifcant amount of time and work. But, we think that it is worth spending additional work and time on implementing a custom network subsystem based on C-verbs to leverage the true performance of InfiniBand hardware if required for a target application.

References

[1] https://github.com/hhu-bsinfo/jib-benchmarks/, Title = JIB-Benchmarks Github Repository.

[2] https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/diag/problem_determination/rdma_jsor_connection_reset.html, Title = IBM. RDMA connection reset exceptions.

[3] https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/diag/problem_determination/rdma_jsor_hang.html, Title = IBM. RDMA communication appears to hang.

[4] Apache ignite - database and caching platform. https://ignite.apache.org/.

[5] Cassandra. https://cassandra.apache.org/.

[6] Gemfire - in-memory data grid powered by apache geode. https://pivotal.io/pivotal-gemfire.

[7] Hazelcast - an in-memory data grid. https://hazelcast.com.

[8] Infinispan. http://infinispan.org/.

[9] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/.

[10] libvma github. https://github.com/Mellanox/libvma/.

[11] Mellanox. https://www.mellanox.com/.

[12] Ofed 3.5 release notes. https://downloads.openfabrics.org/OFED/release_notes/OFED_3.5_release_notes.

[13] Openfabrics alliance. https://openfabrics.org/.

[14] Osu micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[15] Ramcloud source code. https://github.com/PlatformLab/RAMCloud.

[16] Rdmamojo - blog on rdma technology and programming by dotan barak. https://www.rdmamojo.com.

[17] Speedus website. http://speedus.torusware.com/index.html.

[18] Top500 list.

[19] Infiniband architecture specification volume 1, release 1.3. http://www.infinibandta.org/, 2015.

[20] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[21] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 1094–1095, 2005.

[22] R. R. Exposito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Fastmpj: a scalable and efficient java message-passing library. *Cluster Computing*, 17:1031–1050, Sept. 2014.

[23] D. Goldenberg, T. Dar, and G. Shainer. Architecture and implementation of sockets direct protocol in windows. *2006 IEEE International Conference on Cluster Computing*, pages 1–9, 2006.

[24] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 902–903, 2005.

[25] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.

[26] W. Huang, H. Zhang, J. He, J. Han, and L. Zhang. Jdib: Java applications interface to unshackle the communication capabilities of infiniband networks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 596–601, 10 2007.

[27] V. Kashyap. Ip over infiniband (ipoib) architecture. https://www.ietf.org/rfc/rfc4392.txt, April 2006.

[28] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *NetDB 2011: 6th Workshop on Networking meetsDatabases*, 2011.

[29] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, 2010.

[30] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[31] X. Liu. Entity centric information retrieval. *SIGIR Forum*, 50:92–92, June 2016.

[32] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. *SIGPLAN Not.*, 50:695–710, Oct. 2015.

[33] S. Mehta and V. Mehta. Hadoop ecosystem: An introduction. In *International Journal of Science and Research (IJSR)*, volume 5, June 2016.

[34] Oracle. Oracle coherence. https://www.oracle.com/technetwork/middleware/coherence/overview/index.html.

[35] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, November 1999. Previous number = SIDL-WP-1999-0120.

[36] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. sson, D. van der Spoel, B. Hess, and E. Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29:845–854, 2013.

[37] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14, New York, NY, USA, 2013. ACM.

[38] G. L. Taboada, J. Touriño, and R. Doallo. Java fast sockets: Enabling high-speed java communications on high performance clusters. *Comput. Commun.*, 31:4049–4059, Nov. 2008.

[39] S. Thirugnanapandi, S. Kodali, N. Richards, T. Ellison, X. Meng, and I. Poddar. Transparent network acceleration for java-based workloads in the cloud. https://www.ibm.com/developerworks/library/j-transparentaccel/, January 2014.

[40] X. Wu, X. Zhu, G. Q. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26:97–107, Jan. 2014.

[41] P. Zhao, Y. Li, H. Xie, Z. Wu, Y. Xu, and J. C. Lui. Measuring and maximizing influence via random walk in social activity networks. pages 323–338, Mar. 2017.