Message passing on InfiniBand RDMA for parallel run-time supports

Alessandro Secco*, Irfan Uddin*, Guilherme Peretti Pezzi*, Massimo Torquati†
*Department of Computer Science,
University of Torino, Corso Svizzera, 185, 10149 Torino, Italy.
†Computer Science Department,
University of Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa - Italy.
E-mails: *{alessandro.secco,mirfanud,peretti}@di.unito.it, †torquati@di.unipi.it

***Abstract*—InfiniBand networks are commonly used in the high performance computing area. They offer RDMA-based operations that help to improve the performance of communication subsystems. In this paper, we propose a minimal message-passing communication layer providing the programmer with a point-to-point communication channel implemented by way of InfiniBand RDMA features. Differently from other libraries exploiting the InfiniBand features, such as the well-known Message Passing Interface (MPI), the proposed library is a communication layer only rather than a programming model, and can be easily used as building block for high-level parallel programming frameworks. Evaluated on micro-benchmarks, the proposed RDMA-based communication channel implementation achieves a comparable performance with highly optimised MPI/InfiniBand implementations. Eventually, the flexibility of the communication layer is evaluated by integrating it within the FastFlow parallel framework, currently supporting TCP/IP networks (via the ZeroMQ communication library).**

I. Introduction

The TCP/IP protocol stack is nowadays the most largely exploited in distributed applications; many of them rely on its features, either directly or indirectly (via a more abstract communication library). Its popularity it is also due to the fact that TCP/IP does not need specialised hardware and supports the majority of the applications. TCP/IP modules are usually implemented at the operating system (OS) level, which means that OS is involved in every operation including the buffer copy on both ends of the cable. For instance, if an application wants to send a message, the OS places the bytes into an anonymous buffer of the main memory and when the transfer is complete the OS copies the data in the receiving buffer of the application. This process is repeated each time a message arrives until the entire byte stream is received, which can reduce the performance when messages are short and/or the distance is long. While most of the features of TCP/IP, such as congestion control, are appreciated for long distance connections, they are a limiting factor in the High Performance Computing (HPC) area in terms of performance. HPC requires high-performance and low-latency InterProcess Communication (IPC) in order to provide performance and scalability to applications [1], which is typically achieved by fast user-level protocols such as Active Messages [2] and VIA [3].

InfiniBand/OFED [4], has been introduced by Mellanox to solve performance issues in HPC at cost of using dedicated hardware. InfiniBand is a communication layer, which provides the application programmer with a communication interface able to do kernel bypass, Remote Direct Memory Access (RDMA) and asynchronous messaging operations. When a message is sent over the Reliable Connection (RC) transport, the InfiniBand hardware segments the message into packets and delivers them directly into the application buffer provided at the destination side (where the segments are reassembled into a complete message). The receiving application gets asynchronous notification once the entire message has been received. This entire process happens without any system calls. The combination of InfiniBand transport layer together with the application transport interface allow to implement RDMA operations at application level [5].

In this paper we present a minimal communication library on top of the native InfiniBand layers implementing *send* and *receive* primitives using InfiniBand RDMA mechanisms. The library provides the application programmer with only efficient and reliable *point-to-point* communication channels able to transport messages of any size.

In order to assess the usability of the proposed library, we integrate the library at the lower layer of the FastFlow parallel framework [6]. The currently available FastFlow distributed implementation (v.2.0.1) is based on the ZeroMQ (v2.2.0) messaging layer [7], which lacks support for InfiniBand networks. We replaced ZeroMQ with our native InfiniBand communication library.

The main contributions of this paper are:

- a careful design and implementation of a minimal message passing communication layer implemented on top of InfiniBand RDMA mechanisms;
- the transparent porting of the FastFlow parallel framework on InfiniBand network.

The rest of the paper is organised as follows. Recent research works using InfiniBand for distributed computation are presented in Sec. II. Section III briefly presents InfiniBand background. Section IV discusses the library implementation choices, whereas Sec. V presents the transparent porting of FastFlow over native InfiniBand providing also some basic FastFlow information. Section VI presents the experimental

results obtained. Finally, Sec. VII proposes potential future directions and draws conclusions.

II. Related work

In this section we briefly describe research works in the domain of distributed computing which use the InfiniBand communication layer.

In the HPC domain, InfiniBand is the most commonly used network [8] and MPI represents the de-facto standard for implementing parallel applications. Existing implementations of MPI over InfiniBand use send/receive operations for small data and control messages, and RDMA operations for large data messages. Among these implementations MVAPICH [9] and OpenMPI [10] are two important cases. The InfiniBand implementations of MPI are highly optimised and fully compatible with the TCP/IP implementations, giving a simple way for the programmers to migrate their software.

An open source RDMA layer for InfiniBand is *librdmaPlus* developed in the Kitten Project [11], [12]. The library, available under GPLv2, supplies some abstraction of read and send functions, as well as connection functionality and memory sharing. While it is an interesting experiment and we used the source code as a tutorial, the library seems tailored on different needs than data flow streaming.

Other libraries have been recently developed, aiming at giving a general purpose communication library over InfiniBand [13],[14]. The project *libvma* [13] targets streaming applications connected via standard sockets. This experiments is interesting where it is required to keep standard socket API.

Works from the Ohio State University [15] [16] show ways to integrate tools with high network communication requirements such as Hadoop, Hadoop Distributed File System, HBase and MemCached replacing sockets with InfiniBand verbs communication. The purpose was to provide the Big Data community with fast communication libraries in order to take advantage of modern clusters infrastructures to improve speed and scalability of Hadhoop middleware.

Example of projects that natively use the InfiniBand network are distributed file systems because they need the highest performance in bandwidth and latency: the porting of PVSF [17] has been reported to give 40% to three times better performance improvement in bandwidth.

High Performance-networking Block Device (HPBD) [18] uses low-level InfiniBand network layers to access remote memories in a cluster environment with an experienced reduction of latency of orders of magnitude compared to Gigabit Ethernet.

III. InfiniBand background

A. InfiniBand API

In InfiniBand Reliable Connection (RC) protocol, the connection occurs between two peers; the one supplying the connection is named *passive* side and the requester is the *active* side. RDMA has hardware-managed protocol allowing the direct transfer of memory areas between communicating peers. The protocol supplies a way to register the local memory area for being accessed by a remote peer, asynchronous RDMA operations functions and a message notification system. In Linux, the API for InfiniBand is named Verbs [5], [19], as defined in the InfiniBand specification. The OFED (OpenFabrics Enterprise Distribution) [20] is a packaging of the InfiniBand related code, drivers and protocols. The main data structures and concepts related to our implementations are given below:

- `Event Channel (EC)`: it is a communication channel used to manage events like connection and disconnection between remote nodes. It is supplied by the *RDMA Connection Manager*, an OFED module aiming at simplifying the connection stage.
- `Protection Domain (PD)`: it is an object where its members, such as memory regions and queue pairs, are allowed to interact to each other.
- `Queue Pair (QP)`: it is the coupling of receiving and sending queues used to transfer data between the nodes.
- `Memory Registration`: this is the Verbs call that registers specific memory for the local kernel and HCA. In order to be used by RDMA operations, the Rkey, pointer and size need to be passed to the remote side.
- `Work Request (WR)`: it is a request for the data transfer. For a write request, this includes a list of addresses and sizes included into the registered memory to be transferred. A read request on the receiver must always match a write request on the sender, and this may generate a completion event on both sides, depending on the operation flags.
- `Completion Queue (CQ)`: it is a queue containing asynchronous completions. Completions are sent to receiver or sender peers depending on the flags set on the Work Request.

IV. RDMA-based message-passing library

In this section, we will describe the implementation of the minimal send-receive library. The main requirements are the following:

- The library must be single threaded and shouldn't call any pthread function. This will avoid adding overhead to the software developer at higher layers.
- Sending data can be asynchronous, for achieving better performance, but reading must be synchronous to easily integrate to the FastFlow programming model.
- It should be possible to connect many active peers to a single passive peer. An implementation of communication patterns is not required at this level.

A. Design of the library

The schematic representation of the point-to-point channel for the InfiniBand network is sketched in Fig. 1. It connects two communicating peers: sender and receiver. Sending and receiving data is performed via *push* and *pop* operations on the *messages queue*. Both nodes have to allocate the memory for the messages queue, but only the sender can actually write data into it. The sender also manages the head of the queue, a pointer to the last sent message, and it calculate the tail while

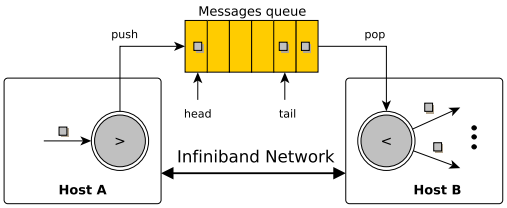

Fig. 1: Schematic representation of the point-to-point channel.

processing the Completion Queue. The receiver can just read data from the queue, it updates the pointer to the last read message (tail) and it calculates the head.

B. Connecting nodes

In InfiniBand, it is possible to initiate a reliable connection using the *RDMA Connection Manager*. It allows to use IP address and TCP ports, which simplify the integration to other system tools.

In TCP, when the server listens to a port and a request is received from a client, it opens a random port to be used for the actual communication. In InfiniBand there is no such protocol and the choice on how to connect multiple nodes to a passive side is left to the developer. In a managed cluster environment, it is possible that a subset of TCP ports may be reserved to a specific program. Therefore we implemented a simple connection protocol with a static port assignation:

- The passive side listens to a number of ports equal to the number of active sides.
- The active side connects to the passive side through a system or user statically assigned port.
- The passive side processes all the incoming connections, one by one.
- The connection is done when all the passive sides are connected to all the active sides. As a connection is established, a *rdma_cm_id* structure for each active side will be created on the passive side, while, on the active side, a single *rdma_cm_id* structure is created to identify the opened connection.

Once the connection is setup, the `Protection Domain`, `Queue Pair` and `Completion Queue` structures are created and can be used to perform communication between the peers. The first operation performed is to allocate all data structures needed and to initiate the phase of memory registration.

C. Memory Registration

The operation of `Memory Registration` is needed to define the memory areas where the network card can access directly: for such areas, virtual to physical mapping cannot be changed by the system and they must be protected from unauthorized remote access. It consists of ”pin down” memory by the OS and exchanging Rkeys between the peers. Our implementation follows this order:

- Memory allocation: For performance reasons, in InfiniBand, it is recommended to allocate memory using *posix_memalign*, that gives an aligned buffer, or a buffer where starting address is multiple of a given value.
- Locally record the allocated memory: This operation is performed with the function *ibv_reg_mr*, that returns a structure of type *ibv_mr*. Each allocated area can be read or write locally or remotely. If more memory area is desired, then the function *ibv_reg_mr* is called more times.
- Rkey dispatch: The is contained in the structure *ibv_mr* and must be sent to the other peer. Unlike the Data Dispatch, we use the Send/Receive model for Memory Registration, instead of RDMA. The function *ibv_post_send* is called with a Work Request operation having the *opcode* flag set to *IBV_WR_SEND* and it must contain the key to send and its size. The peer, after having received the key, will be able to use the local copy of the registered memory area, respecting the permissions declared.

Our implementation includes two methods to automatically send and wait the reception of the needed key. The operation is synchronous: the passive side sends the key and the active side sends back its key after the message is received. To allocate and register the memory, we developed the class *registeredMemory*. It contains the pointers and size of the memory area to register. There is also a flag indicating if the memory is local or a copy of a remote accessible area. The class type to be allocated is specified using the C++ template.

D. Work Request

Once registered, the memory areas may be fully or partially sent to the other side. The memory may be sent through the creation of two lists; Work Request (WR) and Scatter Gather Elements (SGE). We have implemented the *workRequest* class to simplify the creation of the lists with the methods *addRDMASendWithImm*, *addRDMASend* and *addSendMsg*.

A new instance of the class corresponds to a new empty WR list. The methods add different kind of `Work Request`. The method *addRDMASendWithImm* prepares a WR to sends a portion of registered memory to a remote one, attaching to the message an integer. The methods *addRDMASend* and *addSendMsg* are used to prepare a RDMA transfer or a simple message transfer. For each call it is necessary to specify the pointer to the outgoing message, included into the registered memory, and its size.

The possibility to perform inline sending (i.e. a way to send small messages where the send buffer is not checked) is supported: a finite sequence of messages smaller than a fixed size (typically 512K) will be automatically set as inline. When the message is created, the *rdmaSend* method of the class *ofedConnection* is called to send the data.

Usage of immediate data, sending inline messages, sending only a WR at a time, and having a single scatter/gather element helped to achieve the best performance.

E. Shared FIFO Queue

The data structure implemented for sending and receiving the messages in the distributed FastFlow is a First In First

Out (FIFO) queue. This way it is ensured that messages exit the queue in the same order as they enter. A requisite of RDMA is to store the queue in a single virtual contiguous memory area by calling a single *malloc*. The queue has been implemented to circularly use this area, defining a head pointer (the location where a new data has to be inserted), and a tail pointer (where the next data will be removed). The implementation manages the situation of full and empty queue, as well as the passage of the pointers from the end to the beginning of the buffer. The structure created to manage the FIFO queue is *rdmaQueue*. The constructor accepts an object of type *senderSharedMemory*, which contains the buffer used to implement the queue. The queue contains the pointers to the head, last send and tail, which have to be computed when receiving a completion.

The queue is utilized through the methods (*enqueue*) and (*dequeue*): their task is to insert and read a data, managing the pointers. These methods return *false* when the queue is empty (after dequeue) or full (after enqueue), otherwise *true*. We keep a pointer to the last shipped data, to support further developments in which data may be sent asynchronously. The queue also supplies the methods to recompute the pointers when receiving a completion: the sender updates the tail pointer, while the receiver updates the head.

To implement the *onDemand* protocol, The memory shared by the receiver also contains a counter to the request of new data; this has to be incremented and sent each time that the node is available to receive a new message from the sender.

F. Data Dispatch

Data dispatch is done when calling the *send* method of the *ofedConnection* class. Sending data is asynchronous, as allowed by InfiniBand: when data is sent, it is copied into the pipe and a RDMA write with immediate data `Work Request` is sent. When dealing with data streams, the time lost in copying the buffer is lower than the time employed for the network communication, so it is possible to queue a list of send `Work Request`s and keep the network device busy. The improvements given by this solution is evident when comparing the same code with the TCP equivalent.

Other than data shipment, the send method takes care of other aspects, such as flow control and `Completion Queue` processing.

G. Data Reception

The method *recv* of the *ofedConnection* class is in charge of data reception. Because of the asynchronous nature of InfiniBand, the recv method has to enforce synchronization in order to satisfy the requirements of the data streaming model, in which the receiver side is always synchronous on the incoming data. Data reception is made by polling the `Completion Queue` until a new message is received, and calling the dequeue method of the FIFO queue. Data are left on the queue, so it is in charge of the user to consume it before it will be overwritten by new incoming data. However, the library offers the possibility to configure the channel to make a copy of the data, at the cost of introducing additional latency.

H. Completion Queue processing

The Mellanox OFED User's manual [5] shows examples where the `Completion Queue` (CQ) is processed by a separate thread in order to ensure that the CQ is readily emptied (to not saturate). However, in the present case this cannot happen since the CQ is guaranteed to never saturate by our protocol design because:

- There is CQ for each point-to-point channel, thus the number of enqueued completion events can be kept bound by throttling sender and receiver speed.
- The protocol is cooperatively executed in two "dedicated" threads (one for sender one for receiver) which are not meant to execute parallel business code thus are always fully responsive in processing completion events.

The design choice is motivated by the fact that the proposed library is intended to be used in a multi-threaded programming framework, where business code is typically executed in separated threads.

Specifically, in the receiver, the CQ is polled when calling the recv and the FIFO queue does not contain any new message. Polling is performed with the RDMA call *ibv_poll_cq*, which checks for any completion in the queue and return the number of completions; this function also gives a list of work completions with the inline data that, in our case, is used to specify the size of the message. The completion processing is blocking, so that if no completion are received, the receiver will wait for incoming data indefinitely polling the queue. When more than a completion is received, the queue is updated with all the incoming messages, and the next recv would not have to poll the CQ.

For the sender peer, the completion processing is more complex. The CQ is polled when invoking the send method, with a non blocking poll. This allows to update the tail of the FIFO queue leaving the send call asynchronous. However, it is possible that the send is invoked and no completions are received. This may cause the WR to fill the outgoing queue, or the FIFO queue to be filled. To solve this issue, a blocking polling loop is performed in such cases.

When the data flow is ended, a new loop will consume the remaining completions, to avoid the situation in which the receiver may block forever waiting for the last completions.

I. Flow control

InfiniBand gives a partial responsibility to the programmer to decide if and how to regulate the amount of data sent between the peers. In our implementation, the flow control is important to avoid filling the InfiniBand outgoing queues and overwriting the buffer not yet completed by the send operations.

The controls performed on the FIFO queue, partially regulate the data flow. The sender is blocked when the queue is full, slowing down the flow of data. The buffer size, which actually affects the number of messages to be placed in the

buffer, should be big enough to allow the system to achieve good performance, and to allow data not being overwritten before it is consumed. Thus, a more effective control is performed comparing the number of send WR with the number of received sent completion. When the flow is blocked, the completion queue is continuously polled: the only event in fact able to unblock the sender is the reception of a new completion. Thus, the flow control block is actually an busy waiting performing a CQ polling.

The flow may also be controlled by the amount of read `Work Requests` available on the receiver side. A completion is received by the sender when a RDMA write work request meets a `Work Request` on the receiver. Thus, it is possible to write more data at once to more effectively saturate the bandwidth. However, this can be risky because when the sender receives a completion, he is legitimated to reuse the buffer. A possible solution is to use a small amount of `Work Requests` in the receiving queue so that it is not possible to overwrite messages that are still not consumed.

V. Porting FastFlow on InfiniBand network

We are interested to explore the native InfiniBand support for communication in real distributed computation in order to exploit the low latency feature and low CPU usage during communication. We have chosen the FastFlow framework [21] mainly because:

- it has support for distributed computation;
- it uses only point-to-point communication channels at the lower level in the distributed implementation;
- it is relatively easy to integrate new communication library.

A. Background

In this section we provide some basic details of the FastFlow framework. FastFlow is a structured parallel programming framework, originally designed to target shared-memory multi-core and many-core architectures for fine-grain parallel applications. It is implemented in C++ using POSIX threads and mainly targets stream parallelism [22]. Other kinds of parallelism, e.g. data and task parallelism, are supported via stream parallelism. A FastFlow program can be represented as a graph of independent nodes executing simultaneously on subsequent or independent data. As in Kahn process networks [23], graph edges represent true data dependencies Each node of the graph, reads one or more tasks from the input stream, applies some computations on the stream and writes one or more output tasks to the output stream.

FastFlow provides stream parallelism to the programmer in the form of high-level patterns [24]. Basic patterns are: *pipeline*, *farm* and *farm-with-feedback*.

A pipeline models the functional composition of its stages, being either node or other patterns. A farm (a.k.a. master-workers) is a set of nodes that can run in parallel on independent data items. A farm is equipped with an *emitter*, which distributes the data on the worker and an optional *collector* which collects the computed results from the workers. A farm-with-feedback connects the collector to the emitter and can be used to implement parallel recursion. All patterns can be nested and therefore several other patterns can be derived from the basic one. For example, *map* and *reduce* can be derived from farm by properly configuring *emitter* and *collector* to scatter data and reduce gathered data, respectively. *Divide&Conquer* can be implemented by using the farm-with-feedback skeleton.

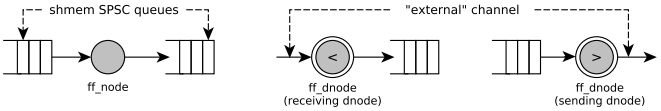

Fig. 2: The schematic representation of FastFlow nodes.

FastFlow has been extended to support distributed computation on cluster of multi-core platforms [25]. It executes programs by coordinating, in a structured way, the fine-grain parallel execution running on a single workstation. The objectives of distributed memory implementation are twofold: i) to exploit the distributed memory on a cluster of multi-core systems, and ii) to investigate process multi-programming vs multi-threading implementation on a single multi-core node.

The FastFlow nodes are shown in Fig. 2. A typical FastFlow application is a graph of nodes (typically a pipeline having farm and map nodes with feedback channels) that are executed on a single machine. In the distributed memory version there are distributed nodes (the *dnode*s) at the edges of the pipeline/-farm structure in order to connect communicating programs on different machines. The distributed nodes have communication channels referred as "external channels" which connect a graph of threads with other graphs of threads running on separated hosts. The external channel in the distributed version of FastFlow has been implemented on top of ZeroMQ [7] communication library.

B. Architectural choices

As mentioned, the present work also aims to provide the FastFlow programming framework with a native InfiniBand compatibility layer. In the design, we identified two different approaches to achieve this result:

- Use the existing interface of FastFlow. The advantage of this approach is reduced development costs from modifying the FastFlow framework. The disadvantage is that the performance may be affected by the fact that the internal FastFlow queues work independently from RDMA queues. The RDMA queue will be treated as a buffer in FastFlow and the same CPU clock will be used to copy data to the RDMA queue.
- Develop a new interface of FastFlow. We can identify the data structure to export and review FastFlow internal structure to support RDMA and find a strategy to allocate user's memory regions directly in the RDMA buffer, with all the advantages in terms of latency of a real ZeroCopy solution. The disadvantage is high development cost of changing the FastFlow internal structure.

Clearly, the development of a new interface in FastFlow would have given the best performance to the framework but the development cost was getting out of the time and budget constraints so we decided to use the existing interface.

C. Requirements

The main requirements for using InfiniBand natively in FastFlow, which must be given by our send-receive compatibility layer and the FastFlow integration layer, are given below:

- Better performance compared to using ZeroMQ.
- Support most of the FastFlow skeletons on the InfiniBand network.
- Scalability on up to 1024 processors (8 processors x 128 hosts).

D. Work description

Once having developed the message-passing library, the integration of FastFlow and InfiniBand was trivial. The TCP descriptor were replaced by our connection abstraction, and all the TCP send and receive were changed to the InfiniBand related calls. Most of the difficulties were about the implementation of one-to-many and many-to-one patterns in which we had to replace the protocols developed for ZeroMQ to a more simple protocol tailored for our solution. The tested patterns were: one-to-one, scatter, allGather, onDemand and fromAny.

VI. Experimental evaluation

In order to evaluate the prototype we presented in this paper, we use three different applications written in FastFlow. These applications are:

- *Unidirectional bandwidth test* for measuring the maximum bandwidth for communication.
- *Ping Pong* for measuring the latency in communication.
- *CWC* for testing a real application developed on FastFlow.

A. Configuration of the host framework

All the experiments have been executed on a 48 cores (4 nodes) cluster connected by 40 Gb/s (4X QDR, MT26428) InfiniBand cards. We have performed our tests with the GNU compiler for all the implementations. The available MPI implementation is MVAPICH2 1.9, which supplies native InfiniBand support. The tests for FastFlow were made both on the native interface and the ZeroMQ implementation on the top of TCP/IP over InfiniBand network. The basic RDMA benchmark shipped with InfiniBand (rdma_bw) scores an average of 25.8 Gb/s for a message size of 65535 bytes and a average of 13.4 Gb/s (with a peak of 21 Gb/s) for a size of 1024 bytes.

B. Unidirectional bandwidth test

In this test, messages are sent continuously from a process located in peer A to a process in peer B. The message sizes range from 10 bytes to 400,000 bytes. We show the results of a single pipe from two implementations: FastFlow using native InfiniBand support and TCP support through ZeroMQ. We also compare the results with the command ib_write_bw, a standard diagnostic test for RDMA, supplied with OFED, giving a reference bandwidth measure, and a similar MPI test.

The following pseudo code shows how the test was developed. In MPI the send and recv are just replaced with *MPI_Send* and *MPI_Recv*. The FastFlow test is composed by a producer generating a number of messages of the same size and a consumer, which processes such messages. The distributed pipe connecting the two nodes is, in one case, shared through the RDMA mechanism and, in the second case, the ZeroMQ implementation running on the top of TCP/IP over InfiniBand.

```
Host A:
_______
init connection
message = new Message(size)
for (i=0; i<LOOPS; i++)
        send(message)

Host B:
_______

init connection
start timer
for (i=0; i<LOOPS; i++)
        message = recv()
end timer
```

The results shown in Table I prove the good performance improvement for FastFlow on InfiniBand using our native library with respect to ZeroMQ using TCP/IP over InfiniBand (IPoIB). It also shows that our solution gives similar results of MPI for most of message sizes, with some advantage in the range of 25k/64k sized messages. MPI gives better performance for messages bigger than 200k and smaller than 10K, because of the highly optimized implementation. While we consider hard to reach the same transfer rate of MPI, we think it is possible to improve our solution with better transfer rates with some future software optimization.

We expected a big difference between using native InfiniBand against ZeroMQ on TCP/IP over IB, but the second performed under expectations for the biggest sized messages. We suspect that the network cards we tested may be better configured to achieve a better bandwidth. In any case, the support for TCP/IP is attractive only when the network does not have InfiniBand hardware.

C. Ping pong

The *ping-pong* benchmark is a standard test used for the measurement of latency in a distributed environment. It sends a single message from a node to another and back, measuring the round trip time, for different message sizes. The pseudo-code of the *ping-pong* test is shown in the following:

```
Host A:
_______
init connection
message = new Message(size)
```

Message size (bytes)	ib_write_bw (Mb/s)	MPI (Mb/s)	FastFlow /IB (Mb/s)	FastFlow/ZMQ /IPoIB (Mb/s)
10	300	192	129	0.7
100	3,600	1,816	1,300	7.0
1,024	22,900	13,936	10,591	70.0
5,000	25,200	23,880	19,761	300.0
10,000	25,500	25,128	20,479	500.0
25,000	25,700	12,408	20,051	1,100.0
50,000	25,800	16,232	21,019	1,950.0
65,536	22,900	17,472	20,889	1,980.0
200,000	25,800	21,208	21,211	3,800.0
400,000	25,800	22,532	21,226	6,200.0

TABLE I: Comparing throughput of different implementations of the *unidirectional bandwidth* test for several message sizes.

message size (bytes)	MPI/IB (us)	FastFlow/IB (us)	FastFlow/ZMQ/IPoIB (us)
10	1.47	1.66	45
100	1.75	1.92	46
1,024	3.91	3.96	49
5,000	6.66	6.56	61
10,000	9.13	9.03	71
25,000	13.74	16.39	84
50,000	21.5	28.34	120

TABLE II: Communication latency (microseconds) of the *ping-pong* benchmark for different message sizes and implementations

```
for (i=0; i<LOOPS; i++)
        send(message)
        message=recv()

Host B:
_______

init connection
start timer
for (i=0; i<LOOPS; i++)
        message = recv()
        send(message)
end timer
```

Table II compares the *mpptest* [26] ping pong implemented using MPI, against the equivalent FastFlow version using ZeroMQ over InfiniBand and using the IPoIB driver, and the same version using our native InfiniBand library. This test again shows the benefits, for FastFlow, to adopt our solution in a native InfiniBand environment instead of using IPoIB driver and a TCP/IP-based communication library.

Comparing to *mpptest*, our implementation behaves well for small sized messages, giving good results. It suffers some extra latency when the message size increases, because of the memory copy on the sender side. In fact, while data streams may benefit from sending data asynchronously, the ping pong test makes data sending synchronous, because every send has to wait for a read. This makes the latency of the send call more evident. In our tests, a simulation of a zero-copy send where the copy is just not performed, cancels the gap between the two versions.

Nodes/core-per-node	FastFlow/IB Elapsed	FastFlow/ZMQ-IPoIB Elapsed
1/12	141	148
2/12	104	143
4/12	95	138

TABLE III: Elapsed time (seconds) for CWC on InfiniBand and TCP/IP over InfiniBand. CPU-U is the CPU time utilisation (seconds) measured for the master process.

D. CWC: Calculus of Wrapped Compartments

CWC is a rewriting-based calculus for the representation and simulation of biological systems [27]. It originally was implemented using multi-threading on multi-core using FastFlow, and then taking advantage of the FastFlow evolution, it was possible to run a distributed version of CWC in the Cloud [28]. We wanted to compare the existing distributed version, running on TCP/IP over InfiniBand using the IBoIP protocol, against the same code where the ZeroMQ communication layer has been replaced by the proposed RDMA library. The objective was to test if the library is able to provide benefits on real applications.

The schema adopted in the implementation of the distributed version of the CWC application, is a classic master/slave, where the master is connected to the slaves by using a scatter pattern, and it receives the results back using a non-deterministic collection pattern (fromAny).

The obtained results are reported in Table III. The tests show that, for 400,000 samples of the test *Phosphate Regulation Mechanism in Escherichia Coli*, running 192 simulations on 48 cores, the elapsed time for the ZeroMQ-based implementation is about 138 seconds, whereas on InfiniBand, it is about 95 seconds. The performance improvement obtained just replacing the communication library is more than 30% in this test.

VII. Conclusion and Future work

TCP/IP-based libraries suffer from high latency and low bandwidth that is not acceptable in the high performance computing setting. The latency can be dramatically reduced using InfiniBand as a native communication layer. In this paper we present and assess a minimal message-passing RDMA-based communication library for InfiniBand networks. Tests performed on simple benchmarks, demonstrate that the proposed library is able to obtain performance close to ideal and up to five times better than that achieved using TCP/IP over InfiniBand using the IPoIB driver.

We also port the distributed version of the FastFlow parallel framework, which natively uses ZeroMQ communication layer, on InfiniBand network by transparently integrating our library at the lower level of the framework. We tested the performance of our library on a real-world application already developed in the distributed version of FastFlow. The results obtained demonstrate that our RDMA-based library is able to improve the performance of the application more than 30% with a consistent reduction of CPU time utilisation with respect to the original TCP/IP implementation.

As future work we are interested to tightly integrate the native InfiniBand network in the FastFlow framework. We would like to modify the code to provide tight integration in order to provide internal data structure directly mapped in RDMA and to extend the FastFlow memory allocator to allocate new messages directly in the RDMA channel avoiding extra memory copies.

ACKNOWLEDGEMENTS

This work has been supported by the EU FP7 grant IST-2011-288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems" and Compagnia di San Paolo project id. ORTO11TPXK "IMPACT: Innovative Methods for Particle Colliders at the Terascale".

REFERENCES

[1] P. Grun, "Introduction to InfiniBand for End Users - Industry standard value and performance for high performance computing and the enterprise," in *Introduce to InfiniBand for End Users*, 2010.

[2] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proceedings of the 19th annual international symposium on Computer architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 256–266.

[3] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The virtual interface architecture," *IEEE Micro*, vol. 18, no. 2, pp. 66–76, Mar. 1998.

[4] InfiniBand trade association. (2013, July) Infiniband. [Online]. Available: http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband

[5] Mellanox Technologies, "RDMA aware networks programming user manual," 2013.

[6] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds. Wiley, 2013, ch. 13.

[7] iMatix Corporation. (2013, January) 0MQ - The Intelligent Transport Layer. [Online]. Available: http://www.zeromq.org

[8] Top500. (2013) Statistics of the most commly used networks (Category: Interconnect Family). [Online]. Available: http://www.top500.org/statistics/list/

[9] The OHIO State University. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. [Online]. Available: http://mvapich.cse.ohio-state.edu

[10] Indiana University. A High Performance Message Passing Library. [Online]. Available: http://www.open-mpi.org

[11] J. R. Lange, K. T. Pedretti, T. Hudson, P. A. Dinda, Z. Cui, L. Xia, P. G. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell, "Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing." in *IPDPS*. IEEE, pp. 1–12.

[12] Sandia National Laboratories. Source code for librdmaPlus. [Online]. Available: https://code.google.com/p/kitten/source/browse/user/runtime/librdmaPlus

[13] Mellanox Technologies. Mellanox's Messaging Accelerator. [Online]. Available: https://code.google.com/p/libvma

[14] E. Salomon (Mellanox). Lightweight messaging library layered on top of rdma. [Online]. Available: https://github.com/accelio/accelio

[15] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda, "Scalable memcached design for infiniband clusters using hybrid transports," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 236–243.

[16] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance rdma-based design of hdfs over infiniband," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 35:1–35:35.

[17] J. Wu, P. Wyckoff, and D. Panda, "Pvfs over infiniband: design and performance evaluation," in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, 2003, pp. 125–132.

[18] S. Liang, R. Noronha, and D. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," in *Cluster Computing, 2005. IEEE International*, 2005, pp. 1–10.

[19] A. Raj (Intel), "InfiniBand host channel adapter verbs implementer's guide," 2003.

[20] OpenFabrics alliance. (2013, July) Ofed overview. [Online]. Available: https://www.openfabrics.org/index.php

[21] M. Aldinucci, M. Meneghin, and M. Torquati, "Efficient Smith-Waterman on multi-core with fastflow," in *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, M. Danelutto, T. Gross, and J. Bourgeois, Eds. Pisa, Italy: IEEE, feb 2010, pp. 195–199.

[22] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies, "Language and compiler design for streaming applications," *Int. J. Parallel Program.*, vol. 33, no. 2, pp. 261–278, Jun. 2005.

[23] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, Amsterdam, Aug 1974, pp. 471–475.

[24] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *in Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana*, 2012, p. 13.

[25] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Targeting distributed systems in fastflow," in *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing*, ser. LNCS, vol. 7640. Springer, 2013, pp. 47—56.

[26] W. D. Gropp and E. Lusk, "Reproducible measurements of MPI performance characteristics," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, J. Dongarra, E. Luque, and T. Margalef, Eds., vol. 1697. Springer Verlag, 1999, pp. 11–18, 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 1999.

[27] M. Coppo, F. Damiani, M. Drocco, E. Grassi, and A. Troina, "Stochastic Calculus of Wrapped Compartment," in *Eighth Workshop on Quantitative Aspects of Programming Languages (QAPL) (affiliated with ETAPS 2010)*, 2010.

[28] M. Aldinucci, M. Torquati, C. Spampinato, M. Drocco, C. Misale, C. Calcagno, and M. Coppo, "Parallel stochastic systems biology in the cloud," *Briefings in Bioinformatics*, June 2013.