Scalable Computing: Practice and Experience
Volume 8, Number 3, pp. 271–280. http://www.scpe.org
ISSN 1895-1767
© 2007 SWPS

# RWAPI OVER INFINIBAND: DESIGN AND PERFORMANCE

OUISSEM BEN FREDJ AND ÉRIC RENAULT*

**Abstract.** This paper presents the design of InfiniWrite, the implementation of a lightweight communication interface called RWAPI over the InfiniBand interconnect for clusters of PCs. Since the specifications of the InfiniBand interconnect provide many ways to transfer data, we are discussing some issues regarding the choices between InfiniBand capabilities. We implemented RWAPI on top of the grid-oriented architecture called GRWA and evaluated the communication performance. GRWA has been developed to provide performance to higher applications on a wide variety of architectures. We obtain a very low latency and a throughput very close to the maximum user bandwidth for messages as small as 4 kilo-bytes.

**Key words.** one-sided communications, cluster and grid computing, high-speed networks, performance.

**1. Introduction.** In recent years, clusters of workstations have become a viable alternative to supercomputer for high performance computing. The success of cluster computing has lead people to interconnect several clusters to form a powerful heterogeneous parallel machine called a cluster of clusters which is now widely used in the industry. High-speed network interconnects that offer low latency and high bandwidth have been one of the main reasons attributed to the success of cluster systems. Some of the leading high-speed networking interconnects include Gigabit-Ethernet, InfiniBand [19], Myrinet [9] and Quadrics [8]. Two common features shared by these interconnects are User-level networking and Direct Memory Access (DMA). The class of communication protocol that best uses these new features is the one-sided protocol. In this class, the completion of a send (resp. receive) operation does not require the intervention of the receiver (resp. sender) process to take a complementary action. RDMA should be used to copy data to (from) the remote user space directly. Suppose that the receiver process has allocated a buffer to store incoming data and the sender has allocated a send buffer. Prior to the data transfer, the receiver must have sent its buffer address to the sender. Once the sender owns the destination address, it initiates a direct-deposit data sending. This task does not interfere with the receiver process. On the receiver side, it focuses on computation tasks, testing if new messages have arrived, or blocking until an incoming message event arises.

The need for a one-sided communication protocol has been recognized for many years. Some of these issues were initially addressed by the POrtable Run-Time Systems (PORTS) consortium [25]. One of the missions of PORTS was to define a standard API for one-sided communications. During the development, several approaches were taken toward the one-sided API. The first one is the *thread-to-thread* communication paradigm which is supported by CHANT [17]. The second one is the *remote service request* (RSR) communication approach supported by libraries such as NEXUS [13] and DMCS [11]. The third approach is a *hybrid* communication that combines both prior paradigms and supported by the TULIP [7] project. All these paradigms are widely used. For example, NEXUS supports the grid computing software infrastructure GLOBUS. MOL [10] extends DMCS with an object migration mechanism and a global namespace for the system. DMCS/MOL is used both in the Parallel Runtime Environment for Multi-computer Applications (PREMA) [5] and in the Dynamic Load Balancing Library (DLBL) [4].

In 1997, MPI-2 [21] (the second MPI standard) has been released with specifications including some basic one-sided functionalities. Although, many studies have integrated one-sided communications to optimize MPI [12]. In 1999, a new communication library called Aggregate Remote Memory Copy Interface (ARMCI) [22] has been released. ARMCI is a high-level library designed for distributed array libraries and compiler run-time systems. IBM has maintained a low-level API, named LAPI [18], implementing the one-sided protocol and running on IBM SP systems only. Similarly, Cray SHMEM [6] provides direct send routines.

At the network layer, many factories have built RDMA features that make it easier the implementation of one-sided paradigms. For example, the HSL [29] network uses the PCI-Direct Deposit Component (PCI-DDC) [16] to offer a message-passing multiprocessor architecture based on a one-sided protocol. InfiniBand [19] proposes native one-sided communications. Myrinet [3, 9] and QNIX [28] do not provide native one-sided communications. But these features may be added (as for example in GM [1] for Myrinet since Myrinet NICs are programmable).

*GET / INT — CNRS UMR 5157 SAMOVAR, Département Informatique, 9, rue Charles Fourier, 91011 Évry, France, Tel: +33 1 60 76 45 73 — Fax: +33 1 60 76 47 80, {ouissem.benfredj, eric.renault} @int-edu.eu

The arrival of these kind of networks has imposed common message-passing libraries to support RDMA (GM [1], VIA [27], VAPI [20]...). Most of these libraries have been extended with one-sided communications to exploit RDMA features. But they do not use these functionalities as a base for their programming model.

**2. RWAPI.** Remote Write is a simple communication protocol that uses asynchronous communications and the one-sided paradigm as a programming model. The Remote-write protocol insure that the sender of a message provides all the information needed to copy a contiguous memory area from one node to another.

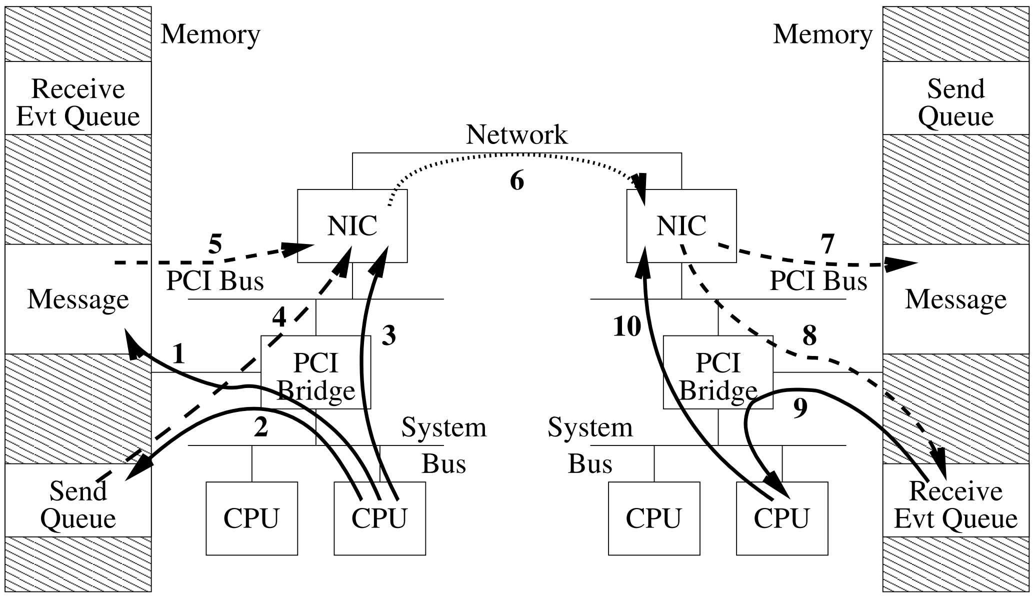


Fig. 2.1. *Message transfer with the remote-write protocol.*

Fig. 2.1 shows the different steps involved in a message transfer using the remote write:

1. the application writes the content of the message in a previously allocated contiguous memory area.
2. the application adds a new entry in the send queue. All information required to perform the message transfer are provided: the local address (where the content of the message is located), the remote address (where the message will be copied on the receiver) and the size of the message.
3. the application informs the NIC that a new entry has been posted in the send queue.
4. the NIC reads the next available entry in the send queue.
5. using the local address and the size of the message, the NIC reads the message content from memory...
6. and sends it to the destination node using the network.
7. when the message arrives on the destination node, it is copied in memory using the remote address and the size of the message.
8. once the message has been received completely, the NIC adds a new entry in the receive event queue.
9. the application is then able to read the content of the receive event queue to take into account new income messages.
10. when a message has been taken into account, the application notifies the NIC to free resources in the receive event queue.

RWAPI (which stands for Remote-Write Application Programming Interface) is a lightweight interface designed to provide a tiny API for the single remote-write primitive. The goal we are trying to achieve is to provide the smallest set of functions that enables to write any parallel programs. This way, we expect:

- to achieve the best performance for communications.
- to require as less development as possible to port our interface to new architectures (and GRWA — see next section — has been defined in this way).

There are two kinds of messages in RWAPI. The first message type requires the destination node identification, both local and remote addresses and the size of the message. Messages in this case can be of any length. The second message type just requires the destination node identification and the message content;

these messages do not require any addresses or size, but are limited to a few 16 bytes. They can be helpfully used to transfer small amounts of information of any kind from one node to another. However, even if they are not limited to this specific use, they are especially useful to exchange addresses before the other message type transfers can occur.

The API is as follows. Unless otherwise mentioned, the value returned by the functions is an error code:

- int rwapi_init ( int, char ** ) must be called before any other RWAPI functions in order to set up the communication interface.
- int rwapi_finalize ( ) should be called after any other RWAPI functions and before exiting the program. This function ensures that all FIFOs are flushed before leaving.
- int rwapi_size ( ) returns the number of nodes in the virtual machine.
- int rwapi_rank ( ) returns the rank of the local node in the virtual parallel machine in the range from 0 to size minus 1.
- void * rwapi_alloc ( size, net * ) allocates a contiguous memory block of the given size in the virtual address space of the process. If the underlying network interface requires the use of contiguous physical memory, it is attached to the application transparently. The value returned by this function is the virtual address in the virtual address space of the process where the contiguous memory block has been attached. The second parameter is the address where the "network" address will be stored when returning from the function. This address is the one that must be used for transferring data.
- int rwapi_free ( void * ) deallocates the memory area provided as a parameter.
- int rwapi_send ( node, small ) sends a small message to another node.
- int rwapi_receive ( node *, small * ) returns information about the oldest incoming message that has not been received whatever the sender. The value returned by the function is 0 if there is no message pending and 1 if a message has been returned. In this case, both the node and the content of the small message are stored at the addresses provided as parameters.
- sid rwapi_write ( node, net, net, size ) sends an arbitrary-long message to another node. Both local and remote "network" addresses must be provided together with the size of the message. The value returned by this function is an SID (Send ID).
- int rwapi_issent ( sid ) returns whether the message identified by the SID has been sent or not. This is useful to determine when a memory area can be reused.

Note that, rwapi_ssend, rwapi_send, rwapi_issent and rwapi_receive are all non blocking functions.

**3. GRWA.** Even if lightweight, RWAPI is generic enough to be implement on top of any computer and communication architectures. In order to avoid rewriting several times the same pieces of code, we developed GRWA (the Global Remote-Write Architecture). GRWA is composed of a set of eight inter-dependable modules. Each module provides a specific set of services which can be used by the others. GRWA allows an application to run on a grid with heterogeneous nodes interconnected with different network hardware. Thus, it guarantees good levels of scalability and portability. Figure 3.1 presents the architecture of GRWA and the relations between the different modules (arrows on the figure represents the dependencies between modules).

*Module ARCH.* aims at providing a generic interface for processor specific functionalities. Typically, this includes the ability to read/write on I/O ports: almost all processors provides a way to let users access I/O ports; however, processor access and operations may vary from one processor to another.

*Module BUS.* implements a separate module for bus management. This allows portability as some NICs are available for more than one I/O bus, and more than one NIC is available for a given I/O bus. It can also improve the efficiency of the interfaces provided by GRWA as it allows to replace the operating-system implementation by any customized ones.

*Module CRYPT.* aims at ensuring the security of "network" addresses used to perform message transfers. At the physical layer, NICs are using physical addresses instead of virtual ones. A solution would consist in translating virtual addresses to physical addresses each time a transfer occurs. However, as whatever the way it is performed, the translation is costly, we consider it is better to provide users the addresses involved in message transfers. Then, provisions have to be made in order to check the validity of "network" addresses provided by the user.

*Module INTERFACE.* provides users a unified API for message passing. This is the only module programmers should see. RWAPI is the default API provided by GRWA. However, other interfaces may be provided. For example, [14] shows it is possible to provide an efficient implementation of other message-passing libraries (like MPI) using only the remote-write primitive.

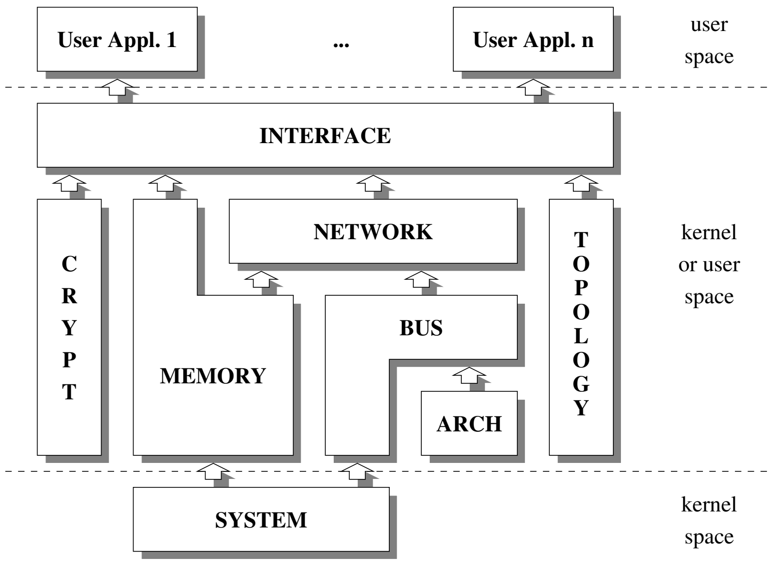


Fig. 3.1. *Global Remote-Write Architecture.*

*Module MEMORY.* manages the memory used to transfer information between nodes. As memory areas used to send and receive data are not necessarily a multiple of a page size, and as memory is a critical resource (especially when the network interface requires the use of physical contiguous memory blocks), it is important to manage the memory resource carefully. As a result, this module manages physical and virtual memory undistinctively, in a similar way as the malloc-free couple of functions does.

*Module NETWORK.* is in charge of managing NICs and routers. In order to improve performance, this management should be performed at the lowest level. This is the case for the implementation [23] on top of the Multi-PC machine [15] (using a HSL network [29]) which requires the application to deal directly with the on board NIC component. However, it is possible to provide implementations on top of any other message-passing libraries, as long as the NETWORK module API is respected. This is useful to provide our interfaces on top of any new architecture until a native implementation becomes available.

*Module SYSTEM.* aims at exporting data from kernel space to user space or at providing applications an access to functionalities which are traditionally reserved to priviledged users.

*Module TOPOLOGY.* provides a set of functions to determine which node number is attached to a host-name, the number of nodes in the parallel virtual machine and useful topological information to route messages.

## 4. InfiniBand.

According to the last list of the TOP500 [2, 24], InfiniBand is the third most used interconnect. The InfiniBand Architecture (IBA) [19] is a new industry-standard architecture for server I/O and inter-server communication. It was developed by the InfiniBand SM Trade Association (IBTA) to provide the levels of reliability, availability, performance, and scalability necessary for present and future server systems, levels significantly better than what can be achieved with bus-oriented I/O structures.

The paradigm of the InfiniBand Architecture is message-passing. It incorporates many of the concepts of the Virtual Interface Architecture. Each vendor had a different software stack with a proprietary added value. However, there are multiple vendor-independent access layers that support different HCA (Host Channel Adapter or network card) simultaneously. A vendor-independent access layer is called a verb.

Figure 4.1 shows that the software stack of the Mellanox IB-Verbs API (VAPI [20]) interface provides a set of operations that closely parallel the proto verbs of the InfiniBand standard, plus additional extension functionalities in the areas of enhanced memory management and adapter properties specifications.

In addition to the above interfaces that can be used to communicate with the Host Channel Adapters' provider drivers directly, there also exist portable interfaces that hide the channel access interface from the user point of view. These include SRP, IPoIB, SDP, and SM. SRP (the SCSI RDMA Protocol) enables access to remote storage devices across an InfiniBand fabric. IPoIB provides standardized IP encapsulation over InfiniBand fabrics as defined by the IETF. This driver operates only within an InfiniBand fabric. SDP (the Sockets Direct Protocol) is an InfiniBand specific protocol defined by the Software Working Group (SWG) of

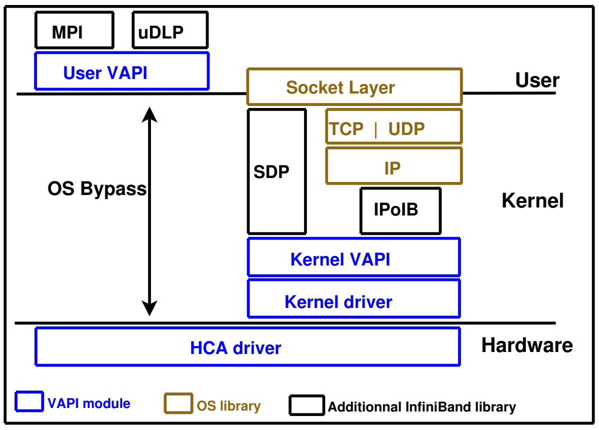


Fig. 4.1. *The Mellanox InfiniBand-Verbs API (VAPI) software stack.*

the InfiniBand Trade Association (IBTA). It defines a standard wire protocol over IBA fabric to support user-mode stream sockets (SOCK_STREAM) over IBA. SM (the Subnet Manager) handles several areas related to the operation of the subnet including: discovery, monitoring and configuration of the ports connected to the subnet, responding to subnet administrative (SA) queries, configuration of I/O units with host channel drivers, performance management, and baseboard management.

InfiniBand is an I/O channel based architecture [26] and not a register based one. A channel based architecture offloads the interprocessor communication overhead from the CPU to provide the maximum available performance. For large multiprocessor applications the performance gain is significant. The use of DMA engines at every InfiniBand node is critical to offloading the CPU. In a traditional load/store model, the data must pass through all levels of the memory hierarchy on its way to and from a CPU register. The device driver is responsible for maintaining the queue of transfers for the device. In an InfiniBand based system, I/O operations are scheduled as queued DMA operations and are handled by the node hardware rather than the CPU. In addition to preserving memory bandwidth and cache resources, this architectural feature also minimizes the number of interrupts that must be serviced by the CPU during a data transfer.

The primary architectural element is the Queue Pair (QP). Each QP consists in an outbound queue, and an inbound queue. Queue Pairs are not shared between applications; therefore, once they are set up, they can be managed at the application level without incurring the overhead of system calls. The QP is the mechanism by which quality of service, system protection, error detection and response, and allowable services are defined. An application can use many QPs, each one with a different quality of service. Creating a QP requires support from the operating system which handles the HCA and initialize memory regions to be used by QPs to manage communication requests and memory operations.

The following transfer types are possible between Queue Pairs: *Send* which includes a scatter/gather capability; *RDMA-Write*; *RDMA-Read*; *Atomic Operations* like Compare & Swap and Fetch & Add in remote memory; *Bind Window* for remote memory management; and *Multicast*.

Each QP is configured for a particular type of service independently. These service types provide different levels of service and different error recovery characteristics. The available transport service types include: Reliable Connection (RC), Unreliable Connection (UC), Reliable Datagram (RD), Unreliable Datagram (UD) and Raw.

**5. InfiniWrite: the RWAPI interface over InfiniBand.** This section presents the design of InfiniWrite which is an implementation of RWAPI over InfiniBand. The InfiniBand architecture is a complex and hierarchical structure comprising many components, libraries, programming interfaces, and modules along with multiple protocols that can be used based on application requirements. Thus, a native implementation of RWAPI over InfiniBand was not planned. Although, we chose to implement RWAPI on top of VAPI which is the lowest reliable layer of the Mellanox software stack.

Like any software and middleware developed on top of InfiniBand, RWAPI processes must use the same InfiniBand type of service. At this time, InfiniWrite uses RC which guarantees the highest level of reliability. However, the RC service requires a complex initialization and a big amount of NIC memory. Actually, rwapi_initialize() creates $(N-1)$ QPs ($N$ is the number of processes for the RWAPI application). Then, it sends the QP numbers to remote processes, one QP number for each remote process. In order to do so, it calls the *exchange* collective operation provided by the bootstrap module called *Fughara*. The initialization function also communicates the mapping process identifier (LID) to the other processes.

In this context, any external memory module could not be used since the VAPI memory module is devoted to the NIC driver. In fact, the VAPI_register_mr() function pins the memory pages as requested for subsequent DMA transfers, translates virtual addresses of the pinned pages into physical addresses, and transmits the obtained physical addresses to the NIC on board driver.

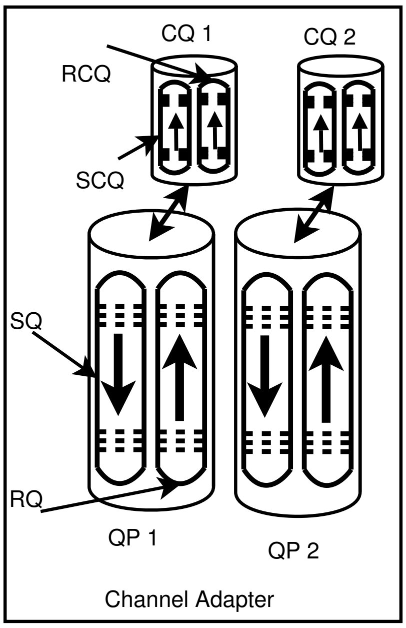


Fig. 5.1. *The organization of the InfiniWrite communication queues.*

Communications with VAPI are regulated by the previously allocated QPs. Each QP encompass a send work queue (SWQ) and a receive work queue (RWQ). The RC type of service supports all transfer types (SEND, RDMA-Read, RDMA-Write, Atomic Operations, and Bind Window). Sending a small message with rwapi_send() can be performed easily by EVAPI_post_inline_sr() which posts a small amount of data in the SWQ using the *Send* transfer type and the PIO mechanism (without any virtual to physical memory address translation). Rwapi_write() takes advantage of the *RDMA-Write* transfer type. In the case of VAPI, the "network" address which describes a rwapi-allocated memory region is mapped into a structure containing the virtual address and two other VAPI-related information, ie. a key and a memory region number.

Since the SWQ size can be set up to a great number of entries (several thousands), there is no need for a host send queue. The InfiniWrite's send functions are non-blocking operations as long as the SWQ is not full. When the SWQ overflows, which is quite infrequent, the function blocks until a send request is executed and a SWQ entry is freed.

InfiniBand is based on the rendez-vous communication model which requires that a receive request is posted before the corresponding send request. Thus, InfiniWrite associates a pre-allocated receive buffer to each entry of the $(N-1)$ RWQs. When a small message arrives, InfiniWrite copies the received data to the user destination, free the receive buffer, and posts it again in the same RWQ entry. In order to perform this last task, InfiniWrite tags each receive buffer with its associated RWQ entry. Then, when a new receive arises, InfiniWrite uses the receive buffer which tag is the RWQ entry number.

In order to control message reception, InfiniWrite sets up at least $(N-1)$ receive completion queue (RCQ), one RQC for each remote process. Rwapi_receive() then polls on the $(N-1)$ RCQ to check if a message has been received. Note that VAPI provides a interrupt-driven receive control. However, InfiniWrite does not use this feature because the management of interrupts is very expensive. This is especially costly for small messages because the interrupt time is comparable with the total communication time.

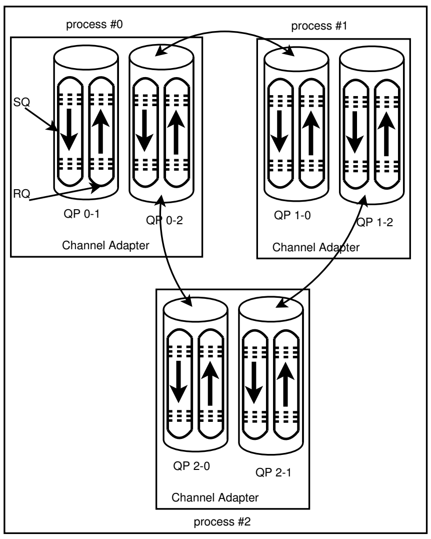
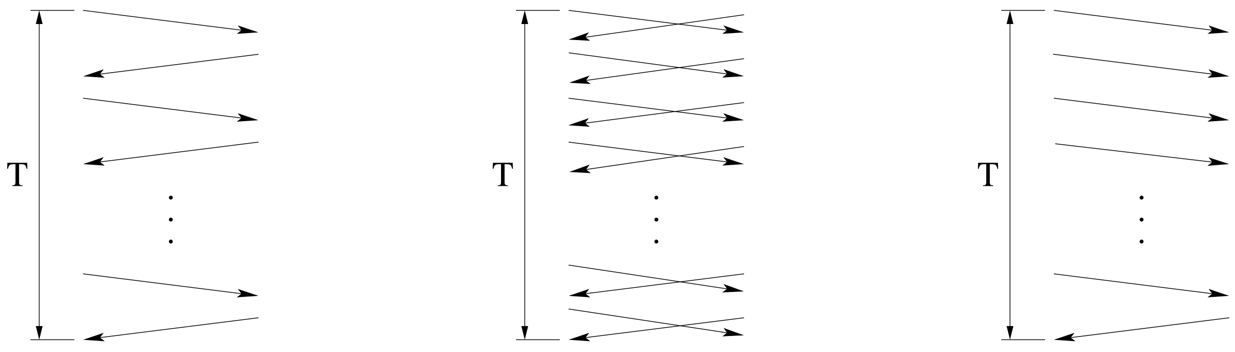


FIG. 5.2. *InfiniWrite design using three processes.*

InfiniWrite implements Rwapi_issent() with $(N-1)$ send completion queues (SCQ). Each time it is called, rwapi_issent() polls SCQs and increments a global variable which stores the total number of the send requests performed locally. The function returns *true* if the value of the argument is greater than the value of the global variable.

**6. Performance measurements.** We compared our implementation with two other existing libraries on top of the InfiniBand Technology: VAPI, the native interface available on top of InfiniBand, and MPI, developed on top of VAPI for InfiniBand. In order to make the comparison between these communications libraries, we use three separate benchmarks (see Fig. 6.1).



(a) Round-trip. (b) Bidirectional RT. (c) One-way.

FIG. 6.1. *Performance benchmarks.*

The first benchmark (Fig. 6.1(a)) is the classic ping-pong in which a message can be sent only once the previous one has been received. This benchmark is used to determine the one-way latency of the network for various message size. The second one (Fig. 6.1(b)) is a bidirectional ping-pong which is used to highlight if a library takes benefits of the bidirectional links. The last benchmark (Fig. 6.1(c)) is the burst. Data is flowing in one direction only and the sender does not need to wait acknowledgment from the receiver before it sends the next one. This benchmark essentially measures the reciprocal of the gap $g$ (the interval between successive sends when the buffering capacity of the network is saturated).

Performance have been measured on Muse, one of the clusters of the Institut National des Télécommunications. Muse is composed of 16 nodes connected using both an InfiniBand 4× interconnection network for data and a Gigabit Ethernet interconnection network as a control network. Each node includes a 1.8-GHz AMD

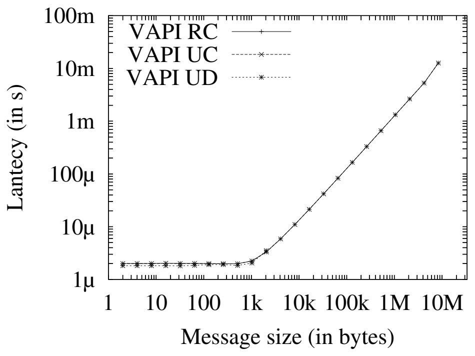
Opteron processor with a 1-MB cache and 2 GB of memory. For mass storage, each node is associated a 40-GB IDE hard drive, except for the first node (the front-end node) which is associated a 80-GB IDE hard drive; note that users accounts are stored on the front-end.

The measurement protocol is as follows: for each message size, each benchmark is run ten times. The duration of a run is one minute (this ensures a high consistency in results and we have determined that the confidence interval is greater than or equal to 90%). The system time is registered before the first message is sent ($t_1$) and after the last message is received on the same node ($t_2$). Let the elapsed time $t$ be the difference between both. For a given run, let $s$ be the size of messages and $n$ be the number of effectively transmitted messages.

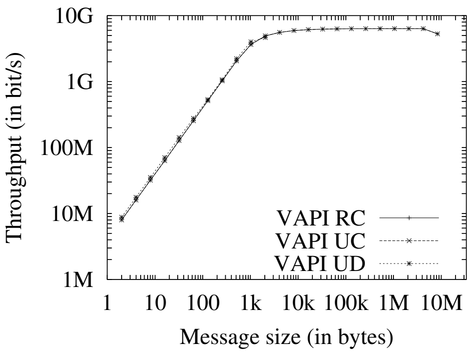
Let the end-to-end latency $L$ (in the following we use the term latency) be the ratio between the elapsed time $t$ and the number of effectively transmitted messages $n$. And let the user throughput $T$ (in the following we use the term throughput) be the ratio between the amount of data (number of effectively transmitted messages $n$ times the size of a message $s$) and the elapsed time $t$.

$$t = t_2 - t_1 \qquad L = \frac{t}{n} \qquad T = \frac{n \times s}{t}$$

Fig. 6.2 provides both latency and throughput for all three service types. These graphs show that they all roughly provide the same communication performance. As a result, we chose to use RC only to make the comparison with the other two libraries (MPI and our remote-write message-passing interface RWAPI) as it is easy to work with and provides reliable communication.



(a) Latency.



(b) Throughput.

FIG. 6.2. *VAPI performance.*

Fig. 6.3 presents a comparison of performance for VAPI, MPI and InfiniWrite message-passing libraries. Fig. 6.3(c) and Fig. 6.3(f) show that InfiniWrite performance are very close to the native VAPI performance.

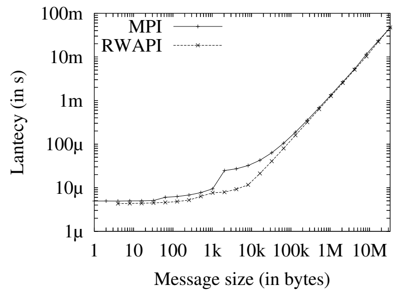
In a general way, Fig. 6.3 shows that InfiniWrite performance are always better than MPI performance. More specifically, the maximum ratio between the minimum latency achieved by RWAPI and the minimum latency achieved by MPI is up to 5.5 $\mu$s for small messages (i. e. 1.76 $\mu$s for InfiniWrite and 9.71 $\mu$s for MPI using the one-way benchmark).

Both InfiniWrite and MPI are able to achieve the maximum user throughput for long messages. However, InfiniWrite is able to provide this maximum user throughput for messages as short as 4 kilo-bytes while MPI cannot do the same for messages smaller than few hundreds kilo-bytes.

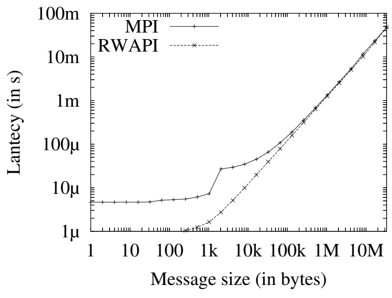
InfiniWrite, as a lightweight interface, coupled with the InfiniBand interconnect, which is characterized by a low-latency routing, provides a good area for scalability in terms of number of nodes.

Finally, the curves on Fig. 6.3 shows that there is an important difference in the management of short and long messages for MPI, represented with a significant decrease of the throughput between 1 and 2 kilo-bytes.

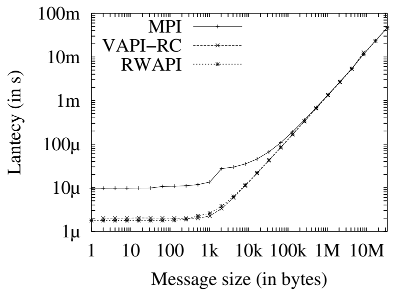
**7. Related Works.** Besides RWAPI, there are other high-performance libraries which are implemented over InfiniBand. Some of the examples use one-sided communications as a programming model and the others use two-sided communications. The first class includes ARMCI GASNET and the extended version of MPI-2. ARMCI is a communication library that optimizes communications in the context of distributed arrays. It



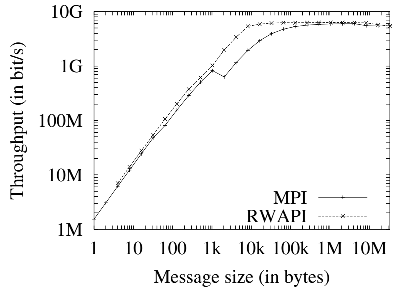
(a) Round-trip.



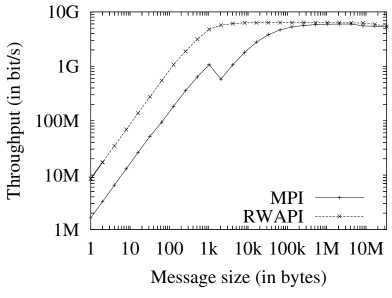
(b) Bidirectional round-trip.



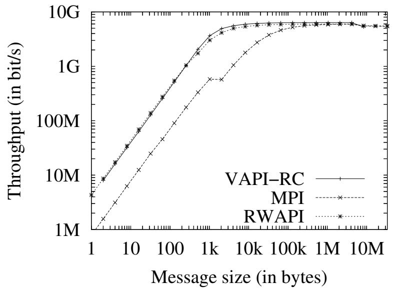
(c) One-way.



(d) Round-trip.



(e) Bidirectional round-trip.



(f) One-way.

FIG. 6.3. *Performance comparison.*

is suited only for fine-grained communications like with a distributed compiler. GASNET provides a global shared-memory abstraction to the user, regardless the hardware implementation and based on the Active-Messages paradigm with a modified Interrupt implementation. However, Active-Messages threads add an extra overhead due to the creation of threads and their management in addition to the thread-safe problem. MPI-2 extends MPI with some one-sided communication routines. The second class of libraries mainly includes the various implementations of MPI. These implementations do not benefit from RDMA capabilities. Recent studies [14] have tried to implement the MPI interface with one-sided routines but the performance is still lower.

**8. Conclusion and Future Works.** In this paper, we have proposed a design for RWAPI over InfiniBand called InfiniWrite. This design takes full advantages of the InfiniBand hardware such as OS-Bypass and RDMA, thus eliminating the involvement of the operating system and the receive process. In addition, InfiniWrite allows the overlap between communication and computation.

In order to decrease the end-to-end latency of small messages, we used the Programmed-IO facility instead of RDMA to copy data to the network card, so as to remove one long-startup-time DMA transaction.

Through performance evaluation, we have shown that our design can achieve a low latency (about 1.76 $\mu$s) and a high user throughput (more than 6300 Mb/s, ie. the maximum available bandwidth for the user) even for short messages. As a comparison, the lowest latency provided by MPI over the same platform is 4.96 $\mu$s and the maximum user throughput cannot be achieved for messages smaller than several hundreds of kilo-bytes.

The lowest InfiniBand communication layer (VAPI) let the user choose between producing events for transfer operation completion or not. This does not suit RWAPI as disabling events does not allow the user to be notified for the completion of send operations and enabling events adds an extra overhead due to the unnecessary receive completion.

Currently, InfiniWrite uses RC as the type of service provided by the InfiniBand packet management. RC requires a connection between each remote HCA and thus consumes much HCA memory resources. Consumed memory is mainly used to store data reassembly informations for each connection. To achieve better scalability, we are working on applying the RD type of service which bypasses any connection management and maintains a reliable communication.

## REFERENCES

[1] *GM: A message-passing system for myrinet networks 2.0.12*, 1995.

[2] *TOP500 list*. `http://www.top500.org` November 2006.

[3] ANSI/VITA 26-1998, *Myrinet-on-VME Protocol Specification.*, 1998.

[4] M. Balasubramaniam, K. Barker, I. Banicescu, N. Chrisochoides, J. Pabico, and R. Cariño, *A novel dynamic load balancing library for cluster computing*, in Proceedings of the IEEE International Symposium on Parallel and Distributed Computing (ISPDC/HeteroPar 2004), Cork, Ireland, July 2004, IEEE Computer Society, IEEE, pp. 346–353.

[5] K. J. Barker, *Runtime support for load balancing of parallel adaptive and irregular applications*, PhD thesis, 2004. Adviser-Nikos Chrisochoides.

[6] R. Barriuso and A. Knies, *SHMEM User's Guide*, Cray Research Inc, 1994.

[7] P. Beckman and D. Gannon, *Tulip: Parallel run-time support system for pC++*.

[8] J. Beecroft, D. Addison, F. Petrini, and M. McLaren, *Quadrics QsNet II: A network for Supercomputing Applications*, in In Hot Chips 15, Stanford University, Palo Alto, CA, August 2003.

[9] N. Boden, D. Cohen, R. Flederman, A. Kulawik, C. Seitz, J. Selzovic, and W. Su, *Myrinet—A Gigabit-per-Second Local-Area Network*, vol. 15, 1995, pp. 29–36.

[10] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel, *Mobile object layer: A runtime substrate for parallel adaptive and irregular computations*, Advances in Engineering Software, 31 (2000), pp. 621–637.

[11] N. Chrisochoides, I. Kodukula, and K. Pingali, *Data Movement and Control Substrate for parallel scientific computing*, in First International Workshop on Communication and Architectural Support for Network-Based Parallel Computing, D. Panda and C. Stunkel, eds., vol. 1199 of Lecture Notes in Computer Science, San Antonio, TX, February 1997, IEEE Computer Society, Springer-Verlag, pp. 256–268.

[12] J. Dobbelaere and N. Chrisochoides, *One-sided communication over MPI-1.*

[13] I. Foster, C. Kesselman, and S. Tuecke, *The Nexus task-parallel runtime system*, in Proc. 1st Intl Workshop on Parallel Processing, Tata McGraw Hill, 1994, pp. 457–462.

[14] O. Glück, *Optimisations de la bibliothèque de communication MPI pour machines parallèles de type "grappe de PCs" sur une primitive d'écriture distante*, PhD thesis, Université Paris VI, July 2002.

[15] O. Glück, A. Zerrouki, J. Desbarbieux, A. Fenyo, A. Greiner, F. Wajsbürt, C. Spasevski, F. Silva, and E. Dreyfus, *Protocol and Performance Analysis of the MPC Parallel Computer*, in Proceedings of the 15th International Parallel and Distributed Processing Symposium, San Francisco, CA, April 2001, pp. 52–58.

[16] A. Greiner, J. Desbarbieux, J. Lecler, F. Potter, F. Wajsbürt, S. Penain, and C. Spasevski, *PCI-DDC Specifications*, Laboratoire MASI, Université Paris VI, September 1996.

[17] M. Haines, P. Mehrotra, and D. Cronk, *Chant: Lightweight threads in a distributed memory environment*, 1995.

[18] IBM Corporation, ed., *LAPI Programming Guide*, no. IBM Document Number: SA22-7936-00 in IBM Reliable Scalable Cluster Technology for AIX L5, First Edition, Poughkeepsie, NY, September 2004.

[19] InfiniBand Trade Association, *The InfiniBand Architecture, Specification Volume 1 & 2*, June 2001. Release 1.0.a.

[20] Mellanox Technologies Inc, *Mellanox ib-verbs api (vapi)*, 2001.

[21] Message Passing Interface Forum MPIF, *MPI-2: Extensions to the Message-Passing Interface.* Technical Report, University of Tennessee, Knoxville, 1996.

[22] J. Nieplocha and B. Carpenter, *ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems*, Lecture Notes in Computer Science, 1586 (1999), pp. 533–546.

[23] E. Renault and P. David, *Performance and Analysis of the PCI-DDC Remote-Write Implementation*, in 2001 IEEE International Conference on Cluster Computing, D. Katz, T. Sterling, M. Baker, L. Bergman, M. Paprzycki, and R. Buyya, eds., Newport Beach, CA, October 2001, pp. 197–203.

[24] E. Strohmaier, *Top500—top500 supercomputer*, in SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, New York, NY, USA, 2006, ACM Press, p. 18.

[25] The PORTS Consortium, *The PORTS0 Interface*, Technical Report ANL/MCS-TM-203, Mathematics and Computer Science Division, Argonne National Laboratory, February 1995.

[26] Tom Shanley, *InfiniBand Network Architecture*, PC System Architecture, MindShare, Inc., 2003.

[27] *Virtual Interface Architecture Specification, Version 1.0, published by Compaq, Intel, and Microsoft.* December 1997.

[28] Amelia De Vivo, *A Light-Weight Communication System for a High Performance System Area Network,* Universitá di Salerno—Italy, November 2001.

[29] C. Whitby Strevens and al., *IEEE Draft Std P1355—Standard for Heterogeneous Interconnect—Low Cost Low Latency Scalable Serial Interconnect for Parallel System Construction*, 1993.

*Edited by:* Dana Petcu
*Received:* May 31, 2007
*Accepted:* June 17, 2007