An MPI Implementation on the Top of the Virtual Interface Architecture⋆

Massimo Bertozzi, Franco Boselli, Gianni Conte, and Monica Reggiani

Dipartimento di Ingegneria dell'Informazione
Università di Parma, I-43100 Parma, Italy
{bertozzi,boselli,conte,reggiani}@ce.unipr.it

Abstract. This paper describes an implementation of the LAM MPI suite on the top of the Virtual Interface Architecture.
The Virtual Interface Architecture (VIA) is an emerging standard designed by Intel, Microsoft, and Compaq aimed at the reduction of communication latency for cluster of workstations or system area networks.
Thanks to M-VIA, a Linux software module that emulates VIA and provides programmers of VIA APIs, it has been possible to develop an MPI implementation even in absence of a hardware VIA interface. Nonetheless, M-VIA module high optimization as well as VIA protocol simplicity permitted a reduction of latency time with respect to the use of the TCP/IP protocol on Ethernet network interfaces.

1 Introduction

Thanks to the widespreading of NOW prototypes [1], message passing is going to be the most used communication paradigm for parallel machines. Standard programming environment and libraries have been designed to enable the programmers to write fully portable parallel code. Amongst others, the *Message Passing Interface* (MPI) has rapidly received large acceptance because it has been carefully designed to maximize performance on a large variety of systems [8].

Unfortunately, the use of MPI on cluster of workstations generally relies on OS services such as the TCP/IP protocol. This has an obvious disadvantage, mainly the high latency associated to the requirements that TCP enforces in order to work in almost all conditions across highly heterogeneous platforms and WANs. The main advantage is that the programmer is automatically endowed of a truly multi user, reliable programming environment. In addition, most NOW prototypes rely on off-the-shelf network solutions like Ethernet (generally Fast– or, even, Giga–Ethernet), switched Ethernet, ATM, or FDDI. While these solutions provide a good price/performance ratio from the point of view of the bandwidth, they require the operating system intervention for dispatching messages from and to the applications, therefore introducing another latency in the link between application and network.

⋆ The work described in this paper has been carried out under the financial support of the Italian *Ministero dell'Università e della Ricerca Scientifica e Tecnologica* (*MURST*) in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project.

Many solutions to these problems have been proposed, ranging from the development of hardware components dedicated to speedup communications [2, 6, 3] to the implementation of light communication protocols [11, 15, 10]. Nevertheless, no one of the proposed solutions has gained a wide acceptance.

Starting from previous experiences [16], a consortium composed by Intel, Compaq, and Microsoft introduced a new standard for cluster of workstation: the *Virtual Interface Architecture* (VIA), that seems the most promising solution for reducing the communication latency and speeding up the application–network link. The VIA standard, in fact, endorses a direct interface between user applications and network layer without requiring participation of the operating system. The first products based on the VIA standard are now appearing and the proposing consortium seems strong enough to guarantee a wide acceptance of this network standard.

This paper presents an implementation of the LAM MPI suite on the top of the Virtual Interface Architecture. Thanks to M-VIA, a Linux software module that emulates VIA and provides programmers of VIA APIs, it has been possible to develop an MPI implementation even in absence of a hardware VIA implementation.

This paper is organized as follows. Section 2 discusses communication issues and introduces VIA. Section 3 presents the implementation, while section 4 discusses the results and performance of the final product. Section 5 ends the paper with some final remarks.

2 The Communication Architecture

An ideal communication interface/protocol for cluster of workstations or system area networks should feature the following characteristics: low communication latency, high bandwidth, small overhead, and small CPU intervention.

A number of solutions have been proposed to overcome these problems: U-Net [16], Active Messages [17], or Fast Messages [14]. These solutions share the underlying idea of giving to user processes direct access to network resources. Unfortunately, none of the above solutions has reached a wide acceptance.More recently, starting from these experiences, a consortium composed by Microsoft, Compaq and Intel authored a new standard, the Virtual Interface Architecture specifically designed for fast interprocess communication in a cluster or System Area Network environment.

2.1 The Virtual Interface architecture

The basic concepts that have been used for the VIA design are the following:

- the process has to execute a small number of operations for starting, ending, or managing a communication session;
- these operations have to be as simple as possible;
- I/O operations must not require the use of interrupts;
- context switches as well as copies of data have to be avoided whenever possible;
- concurrency for the communication interface must be managed without operating system intervention.

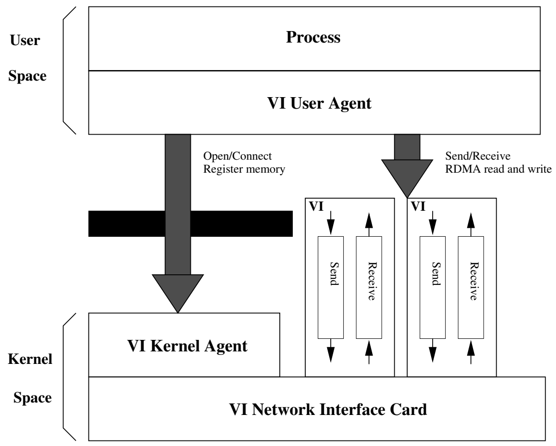

Fig. 1. VIA architectural model.

In the VIA model, the interface driver (usually referred to as the *kernel agent*) performs the setup and manages the resources needed for communications between processes, thus not taking charge of the actual transmission of data. Each process is provided with the exclusive use of a *virtual interface* (VI) for each other process with which it needs to exchange data. The VIs are featured by two queues for sending and receiving data (see figure 1).

VIA provides two types of data transfer facilities: Send/Receive and Remote Direct Memory Access (RDMA). The Send/Receive model follows a standard point-to-point communication model. On both sides, the sender and the receiver must specify the memory regions containing the data to send or where the received data will be placed. In RDMA, the communication initiator specifies both the source buffer and the destination buffer of the data to transfer.

In order to avoid intermediate copies of data, *descriptors* are used: when a process exchanges data with another, it puts a descriptor at the end of the appropriate queue and notifies the VI Network Interface Card writing in a memory mapped register (the *doorbell*). Since now and until the end of the transmission only the interface takes care of the data to be transmitted. The drawback of this mechanism is that the area of memory to be transferred must be *registered* to allow interface access, thus requiring an operating system intervention; this could be generally performed in an initialization phase and not for each transmission.

Similarly, when VIA ends a data transmission the descriptor is completed and the value of the doorbell is changed. The use of these memory mapped registers allows the transfer to proceed without the use of interrupts, that represent a major penalty for modern super-scalar CPUs [9].

Each descriptor is composed by at least two segments. The control segment contains information about the descriptor type, the number of following segments, and the length of data to be transferred. The data segments carry information about local memory regions and their size. For RDMA operation additional address segments are used containing the virtual address of remote memory region to be transferred.

VIA manages each queue in a FIFO fashion but the completion order of descriptors posted on different VIs is not defined. Synchronization between the user agent and VIA could be obtained both using blocking primitives or polling the head of each queue.

Although the concepts behind the VIA standard are not new, its main advantage is that it has been carefully designed to be independent of operating system and processing architecture, therefore sharing a significant goal with the MPI standard [12] and allowing programmers to write fully portable applications. As a consequence, an MPI implementation with the VIA standard support is straightforward.

2.2 The M-VIA Linux Module

The VIA architecture has been created to be easily integrated in silicon as well as emulated via software [7]. For this reason, a number of research groups have produced a software emulation of VIA that allows a quick port of applications toward the VIA standard even in absence of a hardware VIA interface.

Amongst others, the *National Energy Research Scientific Computing Center* of the *National Laboratories Lawrence Berkeley* developed a VIA software emulation [13] for clusters of Linux boxes and Fast– or Giga–Ethernet interconnections.

The M-VIA module allows development of VIA code and, in addition, features higher communication performance with respect to the TCP/IP protocol thanks to the specific network adapter enhancements and to the simpler VIA communication protocol [4, 5]. On the ParMa^2 cluster (see section 4) M-VIA features a latency of 61 μs while TCP/IP latency is 100 μs.

3 Implementation

3.1 The MPI Suite

There are several MPI implementations that target various parallel architectures and operating systems. Amongst others, the most used and widespread are MPICH from the *Mississippi State University* and *Argonne National Laboratory*, and LAM-MPI from the *Supercomputer Centre* of *The Ohio State University* and *University of Notre Dame*. Both these implementations support Linux as operating system and cluster of workstation as architecture. In addition, they are multiprotocol and designed to allow an easy integration of new communication layers. In both cases, the source code is freely available and therefore they seem good choices for adding VIA support.

In order to choose one of these implementations, they have been compared on the performance side. A number of benchmarks on a cluster of homogeneous SMP Linux PCs interconnected by Fast Ethernet (see section 4) demonstrated that LAM-MPI is, generally, slightly faster than MPICH [4]; more specifically:

- the peak bandwidth reached using LAM-MPI is 10% larger than the one reached by MPICH and is nearly the maximal bandwidth for Fast Ethernet;
- the latency of LAM-MPI is larger than the MPICH one;

- LAM-MPI is faster when primitives for process synchronization are used;
- in a SMP environment LAM-MPI improves the overall performance through the use of shared-memory, whilst MPICH does not support shared-memory operations.

Moreover, the internal structure of LAM-MPI seemed to us clearer and simpler than the one of MPICH.

3.2 Implementation of a VIA-based protocol for LAM/MPI

Several variations on how the sending of a message can interact with the execution of the process are available in the Message Passing Interface standard. Common distinctions are between synchronous or asynchronous sends, and blocking or non-blocking communications.The building blocks to construct MPI communication instructions are the two data transfer facilities available in VIA: Send/Receive and RDMA.

To obtain a successful communication with Send and Receive operations, the receiver must post a descriptor on a Receive Queue of adequate size before the sender's data arrives. In RDMA communication the receiver process is not in charge of any operation during the transfer and no notification is given to the remote node that the request has completed. To achieve a synchronous transfer RDMA with Immediate Data must be used. In fact specifying Immediate Data in the initiator of an RDMA Write request leads to the consumption of a descriptor on the remote end when the data transfer is completed.

The plain use of the communication models provided by VIA, Send/Receive and RDMA, does not allow implementation of MPI based on VIA, due to the large number of communication flavors provided by MPI. Therefore a protocol based on the VIA instructions supporting the whole set of MPI communication features should be conceived.

In order to carry out asynchronous communications it is required to have descriptors in the Receive Queue (RQ) available for future sends. As VIA allows pre-posting of descriptors before the VI connection, the solution is to prearrange a suitable number of descriptors in the Receive Queue. Obviously, in order to execute a number of Send/Receive larger than the pre-posted descriptors a reuse of those already consumed by the Sender is required.

The other critical point is that the memory area where the messages are located must be registered by the Kernel Agent. This operation is computational expensive so it is impossible to obtain good performance if a new memory area is registered during each communication. Therefore the area where the message will be placed must be registered during the starting phase, before any communication. Of course, it is a waste of memory to register an area equal to the maximum dimension of the messages for each descriptor in the RQ. Another solution is to register a smaller area size for each descriptor and split longer messages. However, this choice results in larger number of communications.

The use of RDMA operations avoids the previous difficulties: as all the communications can use the same RDMA area no memory is wasted due to a higher estimation of memory requirements. Moreover, the use of Immediate Data option in RDMAs allows

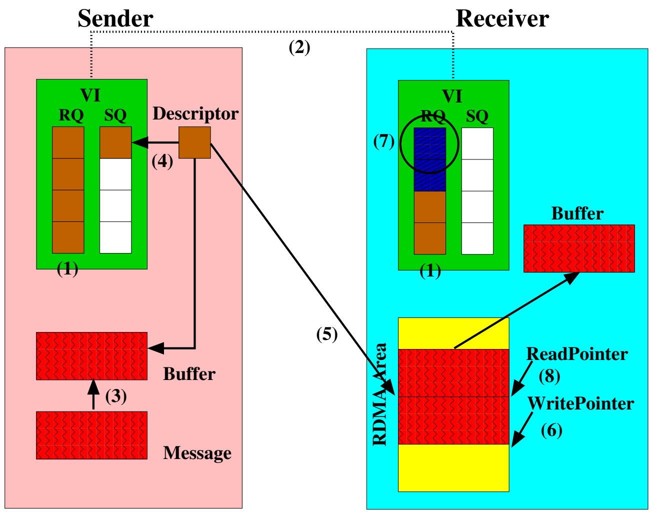

Fig. 2. MPI VIA–based communication protocol.

synchronous transfer. Using RDMA results also in a simpler protocol because only the dimension of the RDMA area must be assessed.

The communication between two MPI processes using RDMA with Immediate Data proceeds as follows. During the starting phase the descriptors are pre-posted in the Receive Queue (1 figure 2), the VIs of the two processes are connected (2) and the RDMA Areas are registered. After exchange of address of the memory regions which will contain the messages, the processes can communicate. To carry out a send, a process must verify whether there is enough free space for the message in the RDMA area of the receiver using the *ReadPointer* and *WritePointer* variables that define the unused area.

When this prerequisite is satisfied the Sender copies the message in the buffer (3) then using the RDMA Write operation a descriptor is inserted in its Send Queue (4) and the message is written in the RDMA Area of the receiver starting at address *WritePointer* (5). Then the number of available descriptors and the variable showing position available for the next send operation in RDMA Area are updated (6).

In the other side in order to execute a Receive, the receiver must verify whether there is a completed descriptor in its Receive Queue (7). When this occurs it reads the message located in the RDMA area at address *ReadPointer*. Then the value of *ReadPointer* is updated (8) and the descriptor is posted again in the RQ.

During the VI connection lifetime the value of the number of descriptors available on the Receive Queues and of the variables ReadPointer and WritePointer are kept consistent using asynchronous RDMA operations between Sender and Receiver.

The protocol described above has been inserted in the LAM/MPI library. As LAM is composed by an Upper Layer which defines the MPI user functions, and a Lower

Layer performing the actual data transfer on the network only this second layer has been modified in order to use the Virtual Interface Architecture as communication protocol.

4 Results

The implementation of VIA support for LAM-MPI has been developed and tested on the ParMa^2 cluster of PCs. ParMa^2 is composed by four dual 450 MHz Pentium PCs with 256 Mbyte RAM each. They are interconnected by a switched Fast Ethernet network based on a CISCO Fast Ethernet switch and full-duplex capable Digital DC21140 Fast–Ethernet network adapters.

The target of this work has been to add the support for VIA to LAM-MPI. In absence of a hardware VIA implementation, a software emulation has been used, therefore performance issues should not matter.

Nonetheless, thanks to the simplicity of the VIA protocol and to the ability of M-VIA in using specific features of supported network adapters, latency of LAM-MPI on VIA is smaller than the one of LAM-MPI on TCP/IP.

In order to evaluate performance, a simple *ping-pong* involving two processes on different nodes of the ParMa^2 cluster has been used. One process issues an MPI_Send and then waits on an MPI_Recv. The other process receives the message and retransmits it back. Table 1 shows the latencies obtained with LAM-MPI on VIA and on TCP/IP for different message sizes.

Message size (byte)	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
LAM on M-VIA (μs)	98	98	98	98	100	104	111	125	151	213	328	470	675	1070	1884	3854
LAM on TCP/IP (μs)	132	132	132	132	137	140	146	158	183	233	335	486	680	1122	1931	3718

Table 1. Latency for LAM-MPI on VIA or TCP/IP.

For small messages, the latency of LAM-MPI on VIA is nearly one quarter smaller than the one of LAM-MPI on TCP/IP. Anyway, for larger messages the memory copy introduced by our implementation (as explained in section 3.2) becomes significant and reduces the advantages of using VIA. A solution to this drawback is currently under implementation: the communication layer initially checks the message size and then switches on the fastest available protocol.

5 Conclusions

In this paper an implementation of LAM-MPI on top of the Virtual Interface Architecture has been presented.

Thanks to a software emulation of VIA, the porting has been possible even in absence of a hardware interface. Nevertheless, thanks to the simplicity of the VIA communication protocol LAM-MPI on VIA has been proved to reduce latency with respect to the same implementation on the TCP/IP protocol.

The VIA based LAM-MPI implementation is freely downloadable from: `http://www.ce.unipr.it/pardis/parma2/`.

References

1. G. Bell. 1995 Observations on Supercomputing Alternatives: Did the MPP Bandwagon Lead to a Cul-de-Sac? *Communications of the ACM*, 30(3):325–348, 1996.
2. M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Procs. International Symposium on Computer Architecture'94*, pages 142–153, 1994.
3. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, Jan.–Feb. 1995.
4. F. Boselli. Integrazione della Virtual Interface Architecture (VIA) all'Interno della Libreria di Message Passing LAM/MPI. Master's thesis, Università degli Studi di Parma - Facoltà di Ingegneria, 1999.
5. L. Bougé, J.-F. Méhaut, and R. Namyst. Efficient Communications in Multithreaded Runtime Systems. *Lecture Notes in Computer Science*, 1586:468–482, Feb. 1999.
6. C. Dubnicki, A. Bilas, C. Yuqun, S. N. Damianakis, and L. Kai. Myrinet communication. *IEEE Micro*, 18(1):50–52, Jan.–Feb. 1998.
7. D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merrit, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–69, Mar.–Apr. 1998.
8. R. Hempel. The MPI Standard for Message Passing. *Lecture Notes in Computer Science*, 797:247–252, Sept. 1994.
9. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.
10. H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Procs. Supercomputing'95*, 1995.
11. P. Marenzoni, G. Rimassa, M. Vignali, M. Bertozzi, G. Conte, and P. Rossi. An Operating System Support to Low-Overhead Communications in NOW Clusters . *Lecture Notes in Computer Science*, 1199:130–143, Feb. 1997.
12. MPI Forum. MPI A Message Passing Interface Standard. Technical report, University of Tennessee, June 1995.
13. National Energy Research Scientific Computer Center. M-via: A high performance modular via for linux. `http://www.nersc.gov/research/FTG/via/` .
14. S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages: efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):90, Apr. 1997.
15. T. Sterling, D. Savarese, B. Fryxell, K. Olson, and D. J. Becker. Communication Overhead for Space Science Applications on the Beowulf Parallel Workstation. In *Procs. High Performance Distributed Computing - HPDC'95*, Pentagon City, Virginia, USA, 1995.
16. T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. *Operating Systems Review*, 29(5):40–53, Dec. 1995.
17. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrated communication and computation. In *Procs. 19th Symp. Computer Architecture*, Gold Coast, Qnd. Australia, May 1992.