Design of a VIA based communication protocol for LAM/MPI Suite

Massimo Bertozzi, Marco Panella, and Monica Reggiani
Dipartimento di Ingegneria dell'Informazione
Università di Parma
Parco Area delle Scienze, 181A Parma, ITALY
bertozzi,panella,reggiani@ce.unipr.it

Abstract

The increasing use of System Area Network (SAN) demands efficient communication to benefit of SAN features through a direct access to network resources and avoiding kernel intervention in communication path.

Recently, a consortium composed by Microsoft, Compaq and Intel authored a new standard, the Virtual Interface Architecture (VIA), designed to reduce software overhead in data transfers.

This paper describes the communication protocol proposed in order to allow a complete implementation of MPI based on VIA. This protocol is needed because the plain use of the two VIA data transfer models does not allow the implementation of MPI based on VIA, due to the large number of MPI communication flavors. To validate the goodness of the proposed protocol, a new communication layer based on VIA has been introduced in the LAM/MPI suite.

The reported results, referring to a software VIA implementation for Fast Ethernet networks, exhibits a significant reduction in latency time of LAM/MPI based on VIA with respect to the same library based on the TCP/IP protocol.

1 Introduction

Several MPI libraries allow implementation and execution of parallel applications, providing both source code portability and high performance. MPI libraries running on latest generation PCs, are today able to exploit almost full Fast Ethernet bandwidth [2]; unfortunately, transmission latency performances have not reached similar excellent results yet. The main reason is the adoption of TCP/IP as protocol for interprocess communication. Due to its layered structure, TCP/IP incurs in performance penalties, unbearable in System Area Networks (SAN) , appointed to speedup connection among proximal and homogeneous nodes. Besides, TCP/IP stack is generally build into operating system kernel [22], so every data transfers involves operating system intervention. Finally data copies performed through TCP/IP layers, and from kernel space to user space or vice versa, represent another cause of lack of performance.

In order to overcome TCP/IP inadequacy for SAN, several research efforts have been carried out [26, 19, 25, 21, 5, 1, 12]. One of the core concept is the removal of kernel intervention in the critical path of communication, supplying a user process with direct access to network interface. In spite of some impressive results, they have remained confined into research area and have not come out in the industry market yet.

In order to define a standard to address low latency communication issues in System Area Networks, a consortium leaded by Intel, Microsoft and Compaq, proposed the Virtual Interface Architecture (VIA), a specification for a user level communication protocol, independent from operating system and hardware architecture [9].

Currently first VIA implementations and applications have come out: Berkeley researchers have produced VIA software emulations for existing network adapters [20]; Giganet already sells a VIA network card [10]; MPI Software Technologies has released first MPI library based on VIA network cards [7].

This paper presents an implementation of the LAM/MPI suite on the top of the Virtual Interface Architecture using Fast Ethernet network cards. Thanks to M-VIA [15], a Linux software module that emulates VIA and provides programmers with VIA APIs, an efficient and complete MPI implementation has been developed.

This paper is organized as follows: section 2 discusses main VIA features; section 3 describes LAM, the used MPI library, while section 4 presents the design of both the communication protocol and the flow control for implementation of all MPI functions within LAM. The results and performance are presented in section 5. Section 6 ends the paper with some final remarks.

1066-6192/01 $10.00 © 2001 IEEE

2 The Virtual Interface Architecture

The increasing use of System Area Network to interconnect cluster's nodes, for a number of applications ranging from database to numerical computation, has boosted interest in network research. Actually, the most used network protocol (TCP/IP) is designed for wide area network, so it must deal with high error rates and heterogenous networks, but is unable to take fully advantage of SAN features such as low error rates, more performant network hardware both in bandwidth and latency.

In order to address SAN's features a number of solutions have been proposed, such as U-Net [26], Active Messages [25], Fast Messages [19], Virtual Memory Mapped Communication [1] and BIP [21]. Most of these solutions share the underlying idea of giving direct access to network resources to user processes, avoiding kernel intervention in communication critical path. Though some excellent results, none of the above projects has been widely used outside research field. Recently, starting from these experiences, a consortium composed by Microsoft, Compaq and Intel authored a new standard, the Virtual Interface Architecture (VIA) specifically designed for fast interprocess communication in System Area Network environment [23, 6, 9]. This effort of standardization seems encouraging because both it inherits the previous positive research experiences and is held by skilled companies.

The main goals followed in VIA design are:

- communications should feature small overhead;
- small CPU intervention is needed in communications;
- 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.

To obtain these targets VIA model eliminates kernel intervention during data transfers; this is achieved thanks to the VIA four components [6]:

- Virtual Interfaces;
- Completion Queues;
- VI Provider;
- VI Consumers.

A user process with the interface (*VI User Agent*), that permits the use of VIA functions, constitutes the *VI Consumer*.

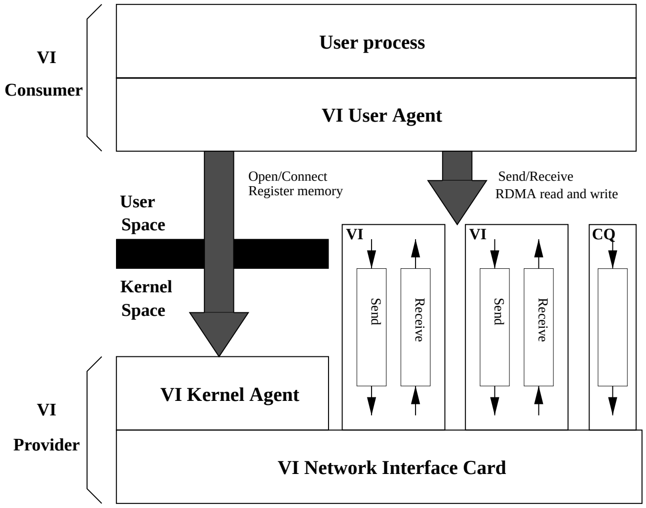

Figure 1. The four VIA components of VIA architectural model: VI Consumer, VI Provider, Virtual Interfaces and Completion Queues.

A VI Consumer is provided with the exclusive use of a *Virtual Interface* (VI), that represents the mechanism allowing a process to gain access to the network interface in order to exchange data. VIA is a connection oriented communication protocol: a VI must be connected with another VI to exchange data. The VIs are featured by two queues, one for sending (*Send Queue*) and one for receiving data (*Receive Queue*) (see figure 1).

Sending or receiving data by polling or blocking on a queue does not involve any operating system call, which is on the other side needed in setting up VIs. This work is performed by *Kernel Agent*, which acts as interface driver and performs the setup and manages the resources needed for communications between processes, thus not taking charge of the actual transmission of data.

In order to avoid intermediate copies of data, structures called *descriptors* are posted on the VI queues to send or receive data. When a process exchanges data with another, it puts a descriptor on the top of the appropriate VI queue and acquaints the VI Network Interface Card (NIC) with the presence of new data in the queue, writing in a memory mapped register (the *doorbell*). When the descriptor is posted in the VI queue, the VI NIC becomes its owner. According to the information contained in the *control segment* of the descriptor, VI NIC performs the communication. As the VI NIC can access directly to the location of the data in the user process address space using the descriptor field *data segment*, the copies of data are avoided.

The drawback of this mechanism is that the area of memory to be transferred must be *registered*, because the VI NIC needs to know the memory physical address, thus requiring an operating system intervention; this operation could be generally demoted in an initialization phase and a registered memory may be reused hereafter.

When VI NIC ends a data transmission, the descriptor previously posted on a queue is marked completed and the value of the doorbell is changed; a process can become aware of completed operation checking the doorbell, thus the application of these memory mapped registers avoid the use of interrupts to complete the data transfer, eliminating one of the major penalty for modern super-scalar CPUs [11].

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 addresses of memory regions containing the data to send or where the received data will be placed. Moreover, a Send operation requires a descriptor already posted on the Receive Queue of the connected VI. In RDMA, the communication initiator specifies both the source buffer and the destination buffer of the data to transfer, which may be read or written in a process memory region. Additional descriptor address segments are required to perform RDMA operations to contain the virtual address of remote memory region, but differently from Send/Receive there is no need of a posted descriptor ready to receive.

Besides, VIA offers the possibility to define a *Completion Queue*, which allows a VI Consumer to coalesce notification of descriptor completions from different VIs in a single point. The drawback is that if a VI queue is associated to a Completion Queue, each synchronization operation must take place on the Completion Queue and no more on the VI queues.

Although the concepts behind the VIA standard are not new [27], its main advantage is that it has been carefully designed to be independent of operating system and processing architecture [23], therefore sharing a significant goal with the MPI standard [14] and allowing programmers to write fully portable applications.

2.1 VIA implementations: M-VIA

VIA has been designed to be easily integrated in silicon as well as emulated via software [9]. Recently first VIA network cards have been released by hardware producers (Giganet [10]). Moreover, a number of research groups have produced software emulation of VIA that allows use of VIA standard even in absence of a hardware VIA interface.

Among them, a Berkeley team is working on a VIA emulation for Myrinet [8, 4, 20] and the *National Energy Research Scientific Computing Center* of the *National Laboratories Lawrence Berkeley* is developing a VIA software emulation [15], M-VIA, for clusters of Linux boxes and Fast– or Giga–Ethernet interconnections.

Since one of the main goals of our cluster project is to maximize performance/price ratio through use of off-the-shelf components [24], we chose Fast Ethernet network connections among cluster nodes and M-VIA.

M-VIA allows the coexistance with traditional network protocol such as TCP/IP. Besides, M-VIA code is freely available.

In addition, M-VIA 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 [2, 3]. On our cluster (see section 5) M-VIA features a latency of 61 μs while TCP/IP latency is 100 μs.

3 The LAM MPI suite

On distributed-memory architectures, message passing is one of the most used communication facilities. The most widespread standard interfaces is Message Passing Interface (MPI) which allows code portability [14].

Interest around MPI is attested by the number of MPI implementations which target different parallel architectures and operating systems. MPICH [13] from the *Mississippi State University* and *Argonne National Laboratory*, and LAM/MPI [17] from the *Supercomputer Center* of *The Ohio State University* and *University of Notre Dame* seem among the most used MPI libraries.

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 5), carried out at the beginning of our project, demonstrated that LAM/MPI is, generally, slightly faster than MPICH [2].

Moreover, the internal structure of LAM/MPI seems clear and simple. LAM/MPI is spread into two layers [16]: an Upper Layer which translate each MPI function in a set of simpler operations, common to all hardware architectures and transport protocols, and a Lower Layer which provides an efficient and reliable message transfer protocol.

The interface between these two completely independent layers is composed by eight functions (Request Progression Interface). A new Lower Layer, implementing the eight functions of Request Progression Interface, is required in order to add a new communication protocol in LAM/MPI. This is the work that has been carried out: the fifth LAM Lower Layer has been designed using VIA standard functions.

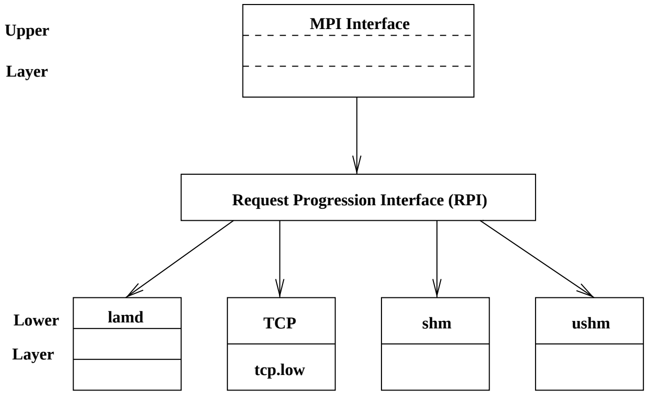

Figure 2. LAM software structure, spread into two layers: an Upper layer, that provides a MPI user interface and a Lower Layer, that performs actual data transfers. Interface between the two layers is the Request Progression Interface.

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

MPI standard offers several variations on how the sender of a message can interact with the execution of the process [18]. Actually, a communication function may return control to the caller routine before the completion of data transfer (*non-blocking*) or wait for it (*blocking*). In addition, a communication function may or may not be provided with information about the completion of operation (*synchronous* or *asynchronous*).

4.1 Communication protocol

In order to design the communication suitable for MPI, the two VIA data transfer models are available: Send/Receive and RDMA.

Nonetheless, the plain use of these communication models provided by VIA does not allow a complete implementation of MPI based on VIA, due to the large number of MPI communication flavors.

In fact, Send and Receive operations require some kind of synchronization: the receiver must post a descriptor on a Receive Queue before delivery of sender's data; otherwise the communication will fail. In RDMA communication model the receiver process is not in charge of any operation during the transfer and no notification is given to the remote node about completion of request. Immediate Data, a descriptor's additional field, can be used to notify the receiver of a RDMA Write operation. Specifying Immediate Data in the initiator of an RDMA Write request conveys information about operation completion, but on the other hand, leads to the consumption of a descriptor on the remote VI, requiring a previously posted descriptor as for Send and Receive model.

Therefore Send and Receive and RDMA Write with Immediate Data follow both a synchronous communication model, while RDMA follows a pure asynchronous model. Since MPI requires synchronous and asynchronous communication functions, use of RDMA alone is insufficient for MPI communication protocol, thus a protocol supporting the whole set of MPI communication features, based on the combination of VIA instructions has been conceived.

A communication protocol constructed with Send and Receive allows straightforward implementation of synchronous, blocking and non blocking point-to-point communications. In order to carry out asynchronous communications the sender should always be able to deliver messages at any time, without prior knowledge of receiver availability. As VIA allows pre-posting of descriptors on a Receive Queue before VIs connection, the solution is to prearrange a suitable number of descriptors on every Receive Queue. Obviously, a reuse of those already consumed is required in order to execute a number of Send/Receive larger than the pre-posted descriptors.

The critical point is that all memory areas where the messages are located, both on sender and receiver side, must be registered by the Kernel Agent. This operation is computational expensive because of kernel intervention, so a new memory area registration during each communication will result in poor performance. Therefore memory areas appointed to messages storage have to be registered during the starting phase, before any communication takes place; than messages may be copied in these memory areas, which can be re-used for further messages.

Nevertheless, registering a memory area equal to the maximum dimension of message for each descriptor is a waste of memory. A solution is to register a smaller memory size for each descriptor and split longer messages. However, this choice conveys larger number of communications and does not exploit full bandwidth.

The use of RDMA operations may avoid both previous difficulties: all received messages are stored in a large memory area, registered in a starting phase. This prevents the need of registration of a great number of buffers on the receiver side, though it requires a sophisticated management and notification of address of this area.

Moreover, the use of RDMA with Immediate Data allows notification of operation completion as in Send/Receive model, allowing the implementation of all MPI functions semantics.

The communication between two MPI processes using RDMA with Immediate Data proceeds as follows:

- VI NIC initialization;

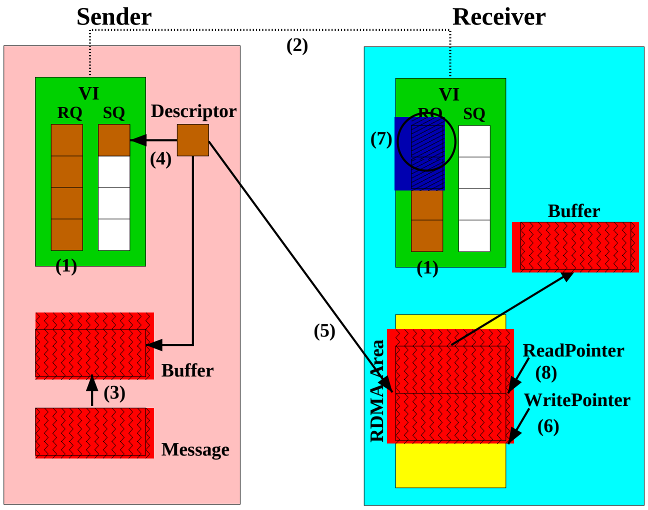

Figure 3. MPI VIA–based communication protocol.

- Descriptors and buffers registration;
- VIs creation;
- Preposting of descriptor on every Receive Queue (1 figure 3);
- VIs connection (2);
- Exchange RDMA addresses;
- Send or receive messages.

After exchange of the addresses of the memory regions which will contain the messages, the processes can communicate.

To carry out a send, a process copies the message in a pre-registered buffer (3), initializes a descriptor with Immediate Data field equal to message length, then the descriptor is inserted on the top of the Send Queue (4) of the VI connected with the receiver process. From this point VI NIC takes charge of data delivering, and so the message is written in the RDMA Area of the receiver starting at address *WritePointer* (5), as indicated by the sender in the remote address descriptor field. Finally variable showing available position for the next send operation in RDMA Area, *WritePointer*, is updated (6).

On the other side in order to execute a Receive, the receiver inspects the top of one of its Receive Queues with a polling operation (7). If a completed descriptor is on a Receive Queue, the process has already a message in its RDMA Area, otherwise it may choose whether it has to wait for an incoming message or to exit from the function. When a descriptor has completed on a Receive Queue, the process reads the message in RDMA Area at address *ReadPointer*, which is updated with message length carried by Immediate Data (8). Finally the completed descriptor is reset and reposted on the Receive Queue to leave the queue full of descriptor ready to receive.

4.2 Flow Control

Flow control purpose is to prevent exhaustion of communication resources, in particular:

- RDMA Area,
- pre-posted descriptors.

Since RDMA Area has a finite size, it has to be carefully managed in order to arrange every incoming message. Thus, when a process has to send a message, checks whether receiver's RDMA Area has enough space to store the message, inspecting *RDMAmaxwrite* variable. In case RDMA Area does not meet the requirement, sender examines *RDMAmaxread* variable, that indicates the amount of the free part of RDMA Area and is updated asynchronously with RDMA operation by the receiver. Receiver process must verify if descriptor with negative Immediate Data arrives; this means the sender has started writing again at the beginning of RDMA Area.

Similar mechanism is used to prevent lack of descriptors preposted on Receive Queues: sender maintains in a variable (*descLeft*) the number of descriptors it has not already consumed on Receive Queue, decrementing at every send. When *descLeft* decreases under a threshold value, it notifies through RDMA write the receiver, which may answer with the number of descriptors preposted on its Receive Queue.

Therefore flow control does not imply any performance penalty, because it exploits asynchronous communication operation such as RDMA Write, and does not force any process to wait for some event. The exception happens when all allocated communication resources are unavailable, that is all the RDMA Area has been fulfilled by messages and none has been read, or many messages have been sent but none read. To avoid this situation the amount of RDMA Area and the number of preposted descriptors must be carefully tuned.

4.3 MPI functions implementation

The protocol described above has been inserted in the LAM/MPI library. LAM/MPI gathers all pending communication requests in a linked list; the MPI Lower Layer takes care to perform communications according to requests in the list. Designed protocol allows straightforward implementation of basic MPI functions, but in order to implement all MPI functions, it has been enhanced with some features.

MPI receive function supplies possibility to recognize between different types of messages, using a *tag*; as a consequence some messages may be discarded by a receive, because of wrong tag, and picked up by a next receive. Besides, a process may decide to receive a message without specifying a sender, with *MPI_ANY_SOURCE* wildcard. Since VIA offers a point-to-point model, in which every VI is connected only to another VI, receiving from any source requires querying multiple VIs. VIA provides Completion Queues, a structure to merge notifications of descriptor completion from different VIs in a single point. Associating to a unique Completion Queue every Receive Queue of all VIs owned by a process prevents this process to poll or block on one of its Receive Queue, therefore every synchronization request must take place on the Completion Queue instead of VI queues.

To take advantage of these VIA features, the protocol has been changed as follows: a process maintains a linked list of messages already arrived but not yet consumed by a receive operation. When a receive function is performed, a process first checks the list for a matching message, then if the message it was looking for is not found, polls the Completion Queue, inserting in the list every not matching message. At the end, if the message has not yet been delivered, the process may choose whether to block on the Completion Queue according to the semantic of the receive operation.

For send operation the protocol remains unchanged; only synchronous sends require a rendez-vous protocol, in which the receiver has to send an acknowledge message to notify the sender of completed data transfer.

5 Performance results

The implementation of LAM Lower Layer over VIA has been developed and tested on the our cluster of PCs. The cluster is composed of four dual 450 MHz Pentium II PCs with 256 Mbyte RAM each. They are interconnected by a switched Fast Ethernet network with full-duplex capable Digital DC21140 Fast–Ethernet network adapters; all nodes are equipped with Linux operating system, kernel 2.2.12 and M-VIA modules, release 0.9.3.

In order to evaluate performance, we used a common *ping-pong* test involving two processes, on a single node (LAM local in table 1) and on different nodes (LAM remote in table 1) of our cluster. Pingpong test is quite simple: a process issues a standard send (MPI_Send) and then waits on a blocking receive (MPI_Recv). The other process first gets ready to receive a message, then as soon as message arrives, sends it back. The whole loop of sending and receiving is repeated several times, measuring the overall time, from which one-way latency is inferred.

Table 1 shows the one-way latencies obtained with LAM/MPI 6.3-b1 on VIA and on TCP/IP for different message sizes, using client-to-client communication in homogeneous cluster of PCs.

Message size	**LAM remote** (μs)		**LAM local** (μs)	
(Bytes)	**TCP/IP**	**VIA**	**TCP/IP**	**VIA**
1	114	63	63	10
2	114	64	63	10
4	114	64	63	10
8	116	66	63	10
16	116	67	63	10
32	120	69	63	10
64	126	77	63	11
128	138	88	64	12
256	163	120	65	18
512	213	169	67	21
1024	310	271	72	27
2048	426	401	81	32
4096	609	590	117	49
8192	996	975	175	106
16384	1743	1756	305	210
32768	3219	3277	529	407

Table 1. Transmission times for LAM/MPI on VIA or TCP/IP.

VIA succeeds in dropping down latency, which features 45% reduction with LAM/MPI on VIA with respect to the same library with TCP/IP. This ratio is much higher on local communication, in which kernel intervention represents the major part of overhead. However, the absolute latency decrease obtained by VIA use is similar in local and remote communications, so that we can estimate as 50μs overhead due to operating system intervention on nodes.

Anyway for larger messages, transmission times are closer, and for 16k message MPI on VIA messages takes more transmission time than TCP ones. In fact as message length increases, the memory copy introduced by our implementation (as explained in section 4) becomes significant and reduces the advantages of using VIA. Different solutions to this drawback have been evaluated: the message protocol engine may initially check message size, deciding whether to copy data on a pre-registered buffer or to register data on the fly. In order to avoid any data copy, it is necessary to work on the whole LAM MPI code, registering in a startup phase data structure appointed to carry MPI messages, thus avoiding buffer requirements.

6 Conclusions and future work

The goal of the porting of VIA within LAM/MPI is to provide a MPI implementation which can drastically decrease latency times. LAM/MPI on VIA has been proved to reduce latency of 45% with respect to the same library on the TCP/IP protocol, even if we used switched Fast Ethernet network and M-VIA instead of an actual VIA hardware interface. This prototypal versione of LAM/MPI on VIA still require some enhancements. Actually, MPI assumes that the transport protocol is reliable, but, while VIA standard endorses reliable and ordered data packet delivery and reception, M-VIA does not implement such feature. Therefore error control and recovery functionalities have to be added to the designed protocol itself.

LAM/MPI implementation over VIA is freely downloadable as a patch for LAM version 6.3.2b from web page: `http://www.ce.UniPR.IT/research/pardis/parma2/index.html`.

References

[1] M. A. Blumrich, K. Li, R. D. Alpert, C. Dubnicki, and E. W. Felten. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. *Proc. 21st Int'l Symp. Computer Architecture*, Apr. 1994.

[2] 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.

[3] L. Bougé, J.-F. Méhaut, and R. Namyst. Efficient Communications in Multithreaded Runtime Systems. *Lecture Notes in Computer Science*, 1586:468–482, Feb. 1999.

[4] P. Buonadonna, A. Geweke, and D. Culler. Implementation and analysis of the Virtual Interface Architecture. Nov. 1998.

[5] G. Ciaccio. Optimal Communication Performance on Fast Ethernet with GAMMA. *Lecture Notes in Computer Science*, 1388:534–547, Mar. 1998.

[6] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification*, Dec. 1997.

[7] R. Dimitrov and A. Skjellum. An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. `http://www.mpi-softtech.com` MPI Software Tecnologies Inc.

[8] C. Dubnicki, A. Bilas, C. Yuqun, S. N. Damianakis, and L. Kai. Myrinet communication. *IEEE Micro*, 18(1):50–52, Jan.–Feb. 1998.

[9] 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*, 18(2), Mar. 1998.

[10] Giganet. Giganet cLAN. `http://www.giganet.com/products/`.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[12] 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.

[13] Mathematics and Computer Science Division, University of Chicago, Chicago. *User's guide for MPICH, a Portable Implementation of MPI*, 1995.

[14] MPI Forum. MPI A Message Passing Interface Standard. Technical report, University of Tennessee, June 1995.

[15] National Energy Research Scientific Computer Center. M-VIA: A high performance Modular VIA for Linux. `http://www.nersc.gov/research/FTG/via/` .

[16] N. Nevin. *Porting the LAM MPI 6.0 Communication Layer*. LAM team, May 1996.

[17] Ohio Supercomputer Center. *MPI Primer/Developing with LAM*. The Ohio State University, Nov. 1996.

[18] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[19] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, Apr. 1997.

[20] Philip Buonadonna and Berkeley researchers. Berkeley via project. `http://www.cs.berkeley.edu/ philipb/via/`.

[21] L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. *Lecture Notes in Computer Science*, 1388:472–485, Mar. 1998.

[22] D. A. Rusling. *The Linux Kernel*. Linux LDP, Mar. 1998.

[23] VIA Org. Specification for the Virtual Interface Architecture. `http://www.viarch.org/default.htm` .

[24] M. Vignali. ParMa^2 White Paper. Technical report, Dipartimento di Ingegneria dell'Informazione - University of Parma, Sept. 1998. Available at ftp://ftp.ce.unipr.it/pub/ParMa2.

[25] T. Von Eicken and al. Active Messages: A Mechanism for Integrated Communication and Computation. In *In Proceedings of the 19th Symp. Computer Architecture*, Gold Coast, Qnd. Australia, May 1992.

[26] T. Von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net:a User-Level Network Interface for Parallel and Distributed Computing. In *In Proceedings of the 15th SOSP*, Copper Mountain, CO, Dec. 1995.

[27] T. Von Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, Nov. 1998.