

Providing a High-Performance VIA-Module for LAM/MPI

Torsten Mehlan, Wolfgang Rehm, Ralph Engler, Tobias Wenzel

Chemnitz University of Technology

Faculty of Computer Science

Straße der Nationen 62, 09111 Chemnitz, Germany

{tome,rehm}@cs.tu-chemnitz.de, {raen,wtob}@s1998.tu-chemnitz.de *

Abstract

The Virtual Interface Architecture (VIA) was introduced to define a common set of features that are suitable to build high-speed networks. Today the interface of VIA serves as access point to a wide range of system area networks. M-VIA is a software that provides the VIA interface on top of several Ethernet cards. The overhead of TCP/IP protocols is avoided by running M-VIA. To benefit from this performance we developed a LAM/MPI module that utilizes VIA functions to transmit data. The key concepts of data transmission and memory management are presented by this paper. Furthermore a comparative performance analysis is included.

1. Introduction

The VIA specification [16] contains the definition of a programming interface. The properties of this interface are adjusted to the needs of high-speed networks. In contrast to traditional network interfaces the programmer has to deal with memory registration, work queues and Remote Direct Memory Access (RDMA). These concepts were carefully identified to meet special requirements of system area networks. Even the InfiniBand technology utilizes important concepts introduced by VIA many years before.

The M-VIA software [5] was developed as part of the NERSC PC Cluster Project [6]. The framework of M-VIA identifies common tasks that have to be accomplished by any VIA provider. The modular structure of M-VIA enables plugging of third-party modules to access special network hardware. Even conventional Ethernet cards may be accessed using an appropriate kernel module. Thus the

TCP/IP overhead is avoided when Ethernet is used for time critical data transmission.

We developed a module for the Request Progression Interface (RPI) of the LAM/MPI distribution [7] [8]. Thus any MPI application may benefit from the use of M-VIA. Since the M-VIA software is frequently used at Ethernet clusters we expect a performance improvement within this environment. The module follows the conventions of RPI modules [9].

The main subject of this paper describes the challenges of managing the properties of VIA in order to fit into the LAM/MPI framework. A basic introduction into VIA and LAM/MPI is given in section two. Section three contains a discussion of the key concepts. Several main problems had to be solved. A reasonable solution is carried out for each of them. Section four presents some performance results. Moreover these performance results are contrasted with related work. Finally in section five we want to conclude.

2. Basic concepts of VIA and LAM/MPI

The Virtual Interface Architecture not only specifies a programming interface. Since this paper does not focus on VIA, we just give a summary of the VIA interface. The user process creates Virtual Interfaces (VI) to exchange data between other processes. Prior to starting any data transmission each VI must be connected to exactly one peer. The VI is composed of one send queue and one receive queue. The user process has to prepare several data structures that are called descriptors. These data structures describe the locations of the source buffers or target buffers together with some additional information. The descriptors have to be posted to one work queue. The actual data transfer is performed asynchronously.

The newest version of LAM/MPI is labeled with the major release number seven. This version implements the entire MPI 1.2 specification. Additionally many functions of MPI 2 are provided by LAM/MPI. The LAM/MPI software is distributed under an open source license. This fact to-

* The research is part of the project 7455/1180 and funded within the framework for technology promotion by means of the European Fund for Regional Development (EFRE) 2000-2006 as well as by means of the SMWA Saxony ministry.

gether with the well fashioned structure of LAM/MPI are sufficient reasons for choosing this software. In contrast to MPICH the LAM/MPI distribution supports four different module interfaces. Thus checkpointing/restart capabilities can be plugged into the MPI library. MPICH does not define a similar interface. Even if MPICH2 aims to implement the entire MPI-2 specification the final release is still not published. The wide range of service interfaces together with the ability to select modules dynamically justify the decision to use LAM/MPI.

The Request Progression Interface (RPI) [9] of LAM/MPI defines a set of functions that are called to process point-to-point transmission of MPI requests. An RPI module implements these functions in order to interface between the upper layer of LAM/MPI and the underlying network API. The function set is well designed and documented. Furthermore a specific module may be selected during startup of the MPI application.

The last thing to mention is M-VIA. As already stated before this software is used to provide a modular VIA implementation. Device specific modules may be developed in order to enable M-VIA to run on new hardware. Several Ethernet drivers are available providing VIA services over Ethernet hardware. All of our tests and measurements are based on the M-VIA 1.0 distribution.

3. Creation of a LAM-RPI for VIA

The RPI module implements several functions. Among others there are functions to process transmission requests. At the RPI level each transmission is a point-to-point transmission. The upper layers of LAM/MPI care for handling collective operations. Each RPI module has to handle only point-to-point communication. Either there is a specific module implementing collective communication, or the collective data transmission is split into several point-to-point transmissions.

The RPI interface identifies four phases for each request. The first one is used to build the request. At this stage the VIA-RPI module allocates some memory in order to manage RPI specific data belonging to this request. The second phase starts the request. The envelope of the request becomes known to the RPI module. The envelope contains sufficient information to determine the next steps. Especially the message size is used to select an appropriate treatment. Look at section 3.3 for more details. The third stage is responsible for processing the request. At this point data transmission takes place. Possibly this stage is entered several times for one request. Furthermore send events and receive events are handled. The last phase is responsible for cleanup the resources allocated during the first two phases.

In the following we present the most important concepts of the VIA-RPI module. These concepts were developed for

efficient use of VIA services within the LAM/MPI framework. At first the data transfer protocols are discussed. Next the administration of pinned memory areas is shown. Finally some notes about the challenge of correct classification of incoming messages are made.

3.1. Data Transfer

The actual kind of data transfer depends on the size of the message. There are three different kinds of protocols. They are called tiny protocol, short protocol and long protocol. At least the distinction between short and long messages is widely used. But we found very good reasons to add a further protocol type. All data is transferred by using RDMA. The send/receive semantics of VIA are never used within our module. Using RDMA the receiver does not have to specify the location and the size of the target buffer within the appropriate receive descriptor. Thus the sender is enabled to write directly into a predefined buffer. Obviously the sender side administers the receive side buffer without requiring a handshake protocol.

Now we have a look at the tiny protocol. Since memory registration is a time consuming operation, the overhead becomes significant for tiny messages. Therefore tiny messages are copied into a preregistered buffer at the sender. The message body succeeds the envelope inside this buffer. Only one descriptor is needed to issue the transmission. The message is written into a preregistered buffer at the receiver. As a result two copies and no memory registration is needed for tiny messages. Compared to memory registration any data copy needs ca. $2 \mu\text{s}$ less time as long as the message is smaller than 16 kbyte. This value was retrieved from the appropriate M-VIA functions.

The short protocol is used for messages that may be too large to be copied around several times. On the other hand short messages are too short to put up with the overhead of a handshake protocol. Hence the sender registers the data buffer. The data is still transmitted to a preregistered buffer at the receiver. Thus we need one copy of the data and one memory registration. It should perform well even if we need two messages because of the envelope. Due to the lazy memory deregistration, frequently used memory keeps registered for a long time. Therefore the registration overhead is avoided in most cases. For messages larger than 16 kbytes a memory copy is much slower than a memory registration. But the receiver still has to perform one copy. This protocol performs well as long as the time for an additional handshake message is larger than the time needed to copy the data at the receiver. If a small message would take around $80 \mu\text{s}$, this protocol is valuable for messages from ca. 16 kbyte to ca. 32 kbyte. Thus the default message sizes for the tiny protocol start from 1 byte up to 16

kbyte. Per default the short protocol is used from 16 kbyte up to 32 kbyte.

Large messages are handled without using any preregistered memory at all. Since the length of a message is not restricted by MPI it could be impossible to keep sufficient memory in reserve. Furthermore the registration of the original data buffer at sender and receiver is more efficient as copying of large data volumes. The protocol has to realize a handshake in order to announce the address of the target buffer. If the message is larger than the current MTU, it will be divided into several smaller data transfers.

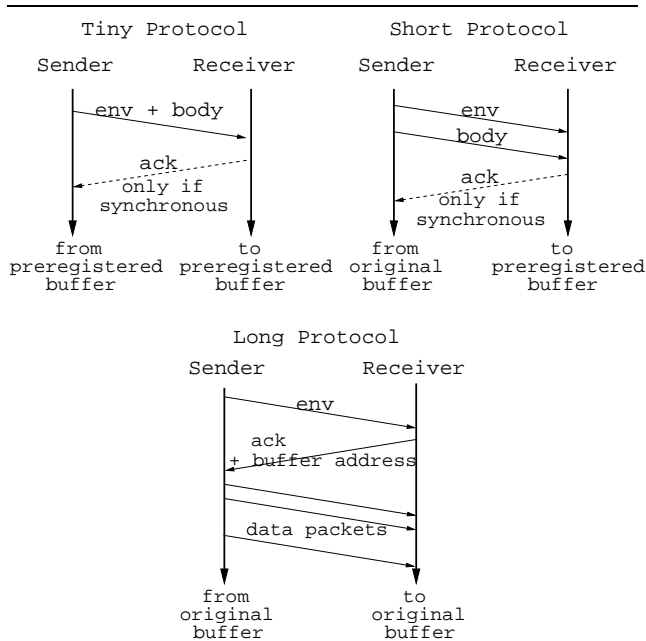


Figure 1. The three protocol types of the RPI module.

3.2. Memory management

Many high-speed networks require a special treatment of memory. To maintain address translation tables some kind of registration must be performed. The participating memory pages need to be locked to make them resident. Also our VIA-RPI module has to care for this task. Unfortunately the memory registration is a time consuming task. To prevent from falling behind TCP/IP speed frequent calls to these functions have to be avoided. Management of registered memory is a research field. In [1] mainly preregistered buffers that do not change during program execution are used.

In order to reduce the number of calls to the memory management functions of VIA we use a lazy memory engine. This name suggests that memory keeps registered even if the current data transfer has finished. It is assumed that the application uses the same memory area several times for communication purposes. Each registration or deregistration of memory is done by the functions of this module. In contrast to the corresponding VIA functions the memory may stay registered. If the same memory area is reused, a time consuming registration becomes unnecessary. This approach is necessary to achieve good performance. On the other hand some problems arise. The behaviour of the `malloc()` function has to be modified. Memory areas have to stay allocated after `free()` is called.

In [2] the concept of the lazy memory engine was introduced. The ideas described by this paper were used several times before [3] [4]. Normally a hash-table was created to maintain the registered memory areas. The start address of a memory area had served as key to identify the matching table entry. This solution works well as long as one assumption is true. One has to assume that memory areas will never interleave each other. Since there is no guarantee for the assumption to be true we have to care about this case. Especially M-VIA does not support multiple registrations of the same memory. If a common part of two memory areas is registered twice, the next deregistration causes the common part to be deregistered also. Refer to figure 2 to get an impression of the effects of multiple registrations.

We decided to discard the hash-table but to use a binary search-tree. Thus it becomes possible to manage interleaved memory while preserving a good performance. The search tree needs $\log_2 n$ steps to find one entry out of n entries. Three basic scenarios have to be handled. In the first one the new memory area is a subset of the old memory area. The second case appears when both memory areas are completely disjoint. In the third case the new memory area partially interleaves the old one. Refer to figure 3 to get an idea of these three cases. We need a criterion to traverse the tree properly. Thus a comparison function is defined. The listing shows the behaviour of the comparison function:

- $sa(y)$ returns the start address of memory area y
- $ea(y)$ returns the end address of memory area y
- Given two memory areas x and y
- $x < y \iff sa(x) < sa(y) \wedge ea(x) \leq ea(y)$
- $x > y \iff sa(x) \geq sa(y) \wedge ea(x) > ea(y)$
- $x = y$ else

Using this criterion we are able to clearly identify the cases shown in figure 3. Since one data block has to be associated with one memory handle we may be forced to deregister a partly interleaved area. The appropriate super-set of

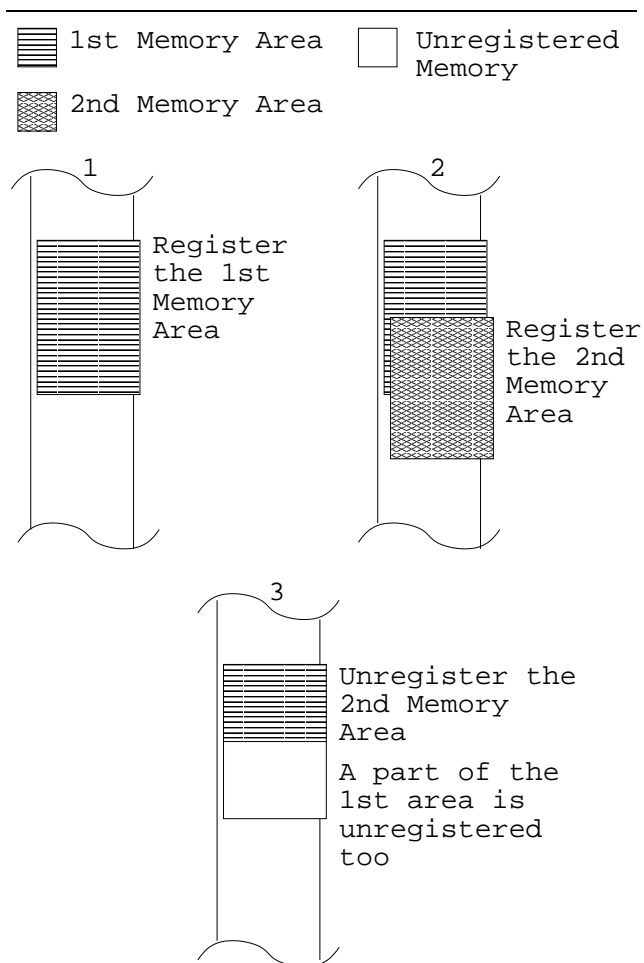


Figure 2. The effect of multiple registrations of one memory area.

memory has to be registered in order to retrieve a valid handle. To avoid errors the memory engine has to keep track of the use count of each memory area. Deregistration is only possible if no data transfer is going on using the memory.

3.3. Classification of incoming messages

Each incoming message causes a receive descriptor to be consumed. Even if RDMA is used solely the specification of immediate data requires a corresponding receive descriptor [16]. Due to this behaviour we are notified about incoming data.

Nevertheless an incoming message has to be classified. At first we have to decide to which connection the message belongs. Next the message type has to be identified in order to take appropriate actions.

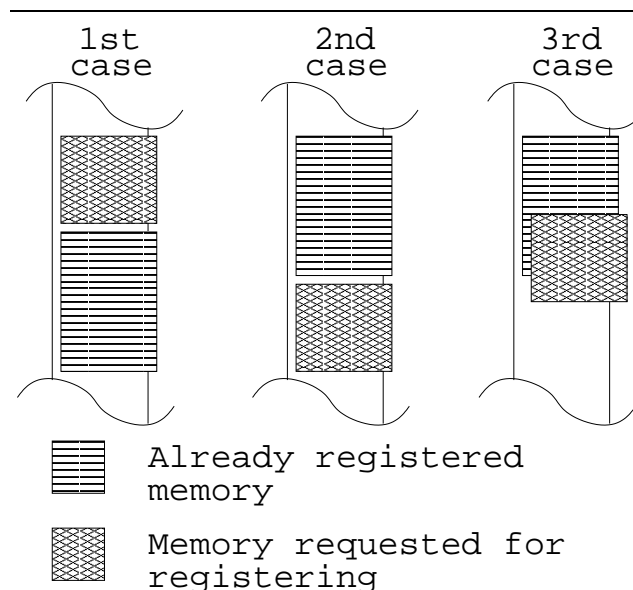


Figure 3. Three cases of memory overlapping.

After the completion of a receive descriptor is recognized the appropriate VI is known. Thus we could map from the VI to the data structure representing the remote process. But we found a more clear way to do this job. The immediate data field of the receive descriptor carries meaningful information. Fifteen bits are used to express the number of the remote process. This is used as index into an array.

Next the type of data has to be determined. This may be either an envelope or a message body. The immediate data field helps to make the decision by adding a special flag. An envelope is accessed appropriately. Thus the fields of the envelope clearly specify what to do next. The message may be an acknowledgement of a synchronous operation, an announcement of a target buffer or a regular envelope.

Finally we have to care for message bodies of long messages and short messages. There is no information inside the request that may help us. Thus we maintain a list of all requests being on the fly. Long messages need a handshake. We exploit the queue discipline of any VI. A long request is added to the head of the list when the rendezvous-message of the receiver arrives at the sender. The acknowledgements carrying buffer information are sent in the same order back. Due to the strict queue discipline of the VI work queues nothing gets confused. The initial order is preserved. Since short message bodies follow directly after the corresponding envelope we can handle this easily. When a short envelope is detected we queue the request on top of this list. Thus an incoming message body always belongs to the re-

quest at the head of the list.

4. Performance

This section presents some performance results. We made a comparison between our VIA-RPI module and already existing solutions. The following software was tested:

- MPICH 1.2.5 using the TCP channel device [14]
- LAM/MPI 7.1 using the TCP RPI [13]
- MVICH based on MPICH 1.2.2.3 using M-VIA [12]

This list contains all solutions that enable MPICH and LAM/MPI 7.x to run with Ethernet connections. Just one software is missing. The ParMa [15] software represents an M-VIA implementation for LAM/MPI 6.3.2. Due to source code errors this software could not be compiled and tested.

In the following the properties of our testbed are listed:

- 2 x Athlon MP 1600+ CPU 1.4GHz real core freq., 2x133MHz FSB, 256K L2 Cache
- 1 x Tyan Tiger MPX mainboard (AMD MPX chipset)
- 1 x 512 MB CL2 Unregistered PC2100 DDR SDRAM Module
- 1 x 3COM 3C920 Fast Ethernet on-board NIC
- 1 x Intel Pro/100 S Desktop Fast Ethernet NIC

All tests used the Intel Pro/100 S Ethernet card for data transmission. The other interfaces were not used during the tests. We run the PALLAS benchmark PMB [11] to determine the results. The PALLAS micro benchmark performs simple MPI operations on two nodes. The Round Trip Time (RTT) is measured. The results give an impression of the performance of an MPI library.

Figure 4 shows the latency of the transmission of messages up to 256 bytes. Our VIA-RPI module shows the best performance. Even the M-VIA based MVICH implementation is a bit slower. This figure also shows the impact of TCP/IP while using the native protocol stack. As expected the TCP devices show a higher latency. Bypassing the TCP/IP stack saves time that would be spent in redundant tasks.

Figure 5 shows the bandwidth of messages from 16 kbytes up to 512 kbytes. The VIA-RPI performs very well and clean. But the LAM/MPI TCP-RPI seems to be better up to 65 kbytes. We tried to determine the reason. Thus we measured the native network latency of TCP. We subtracted the RTT from the results of the PALLAS benchmark. The resulting overhead became lower than zero. Considering this fact the good results of the TCP-RPI become implausible even if we can not explain this effect.

Refer to figure 6 to get an impression of the differences between the raw device speed and the MPI speed. The

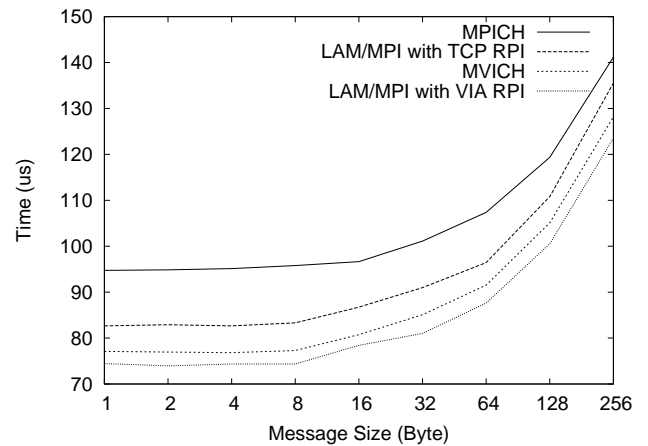


Figure 4. PingPong from 1 byte up to 256 byte.

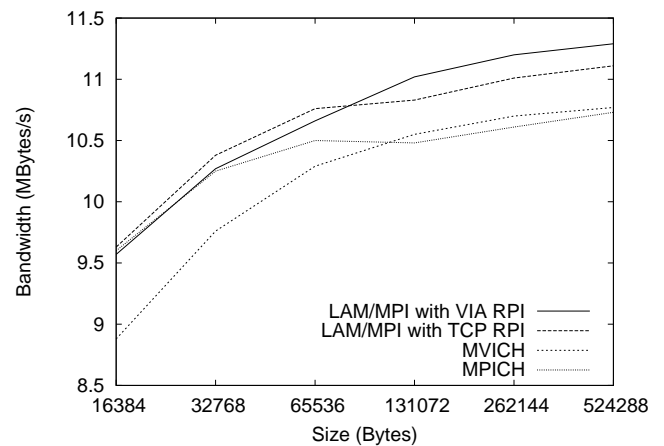


Figure 5. PingPong from 16 kbyte up to 512 kbyte.

RTT of TCP/IP is determined using the NetPIPE benchmark [10]. This RTT was subtracted from the RTT of the LAM/MPI TCP-RPI. The raw RTT of M-VIA was determined using a ping-pong data transmission. This RTT was subtracted from the RTTs of MVICH and LAM/MPI with VIA-RPI. The TCP/IP device shows a difference lower than zero. This would imply that MPI over TCP/IP is faster than direct access to the TCP socket.

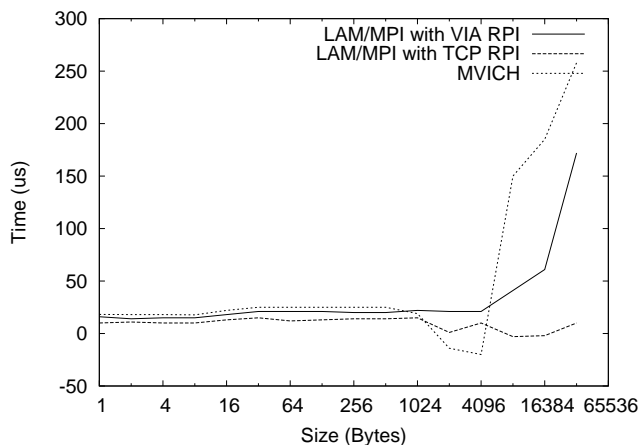


Figure 6. The differences of raw device speed and MPI speed.

5. Conclusion and outlook

This paper presented the new VIA-RPI module for version 7.1 of LAM/MPI. The module was developed with focus on M-VIA. The goal and field of M-VIA was mentioned. Also any other VIA library may be linked to the VIA-RPI module. We started with an introduction of basic concepts of VIA and LAM/MPI. The main part of the paper discussed the challenges of creating the VIA-RPI module. We concentrated on the transmission protocols, the memory management and the message classification. Finally the performance results were shown.

We intend to improve the VIA-RPI device in future. Shared memory support may be added to manage multiple processes at one node. Moreover the module does not handle empty receive queues at the moment. This will be a subject to future work. The M-VIA software also needs to be improved. Especially memory management and device drivers require some changes. The work may be extended to MPICH2. As soon as version 1.0 of MPICH2 appears we will consider the possibility to extend this work to the MPICH2 distribution.

References

- [1] Jiuxing Liu, Jiasheng Wu, Sushmitha P. Kini, Pete Wyckoff, Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand, In the Proceedings of 17th Annual ACM International Conference on Supercomputing, June 2003
- [2] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, Yutaka Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication, In the Proceedings of the 12th International Parallel Processing Symposium, March 1998
- [3] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, William Gropp, Rajeev Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand, In Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 04), April 2004
- [4] René Grabner, Frank Mietke, Wolfgang Rehm. An MPICH2 Channel Device Implementation over VAPI on InfiniBand, In Proceedings of the 18th International Parallel & Distributed Processing Symposium, April 2004
- [5] Paul H. Hargrove, Michael Welcome, Eric Roman. M-VIA Documentation, <http://old-www.nersc.gov/research/FTG/via>, October 2002
- [6] National Energy Research Scientific Computing Center. PC Cluster Project, <http://old-www.nersc.gov/research/FTG/pcp/index.html>, April 2003
- [7] Greg Burns, Raja Daoud, James Vaigl. LAM: An Open Cluster Environment for MPI, Proceedings of 8th Supercomputing Symposium, June 1994
- [8] Jeffrey M. Squyres, Andrew Lumsdaine. A Component Architecture for LAM/MPI, Proceedings, 10th European PVM/MPI Users' Group Meeting, September 2003
- [9] Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine. Request Progression Interface (RPI) System Services Interface (SSI) Modules for LAM/MPI API Version 1.0.0/SSI Version 1.0.0, <http://www.lam-mpi.org/papers/ssi/1.0-ssi-rpi.pdf>, August 2003
- [10] Q. O. Snell, A. Mikler, J. L. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator, In Proceedings of IASTED International Conference on Intelligent Information Management and Systems. 1996
- [11] Karl Solchenbach, Hans-Joachim Plum, Gero Ritzenhoefer. Pallas Effective Bandwidth Benchmark – source code and sample results, ftp://ftp.pallas.de/pub/PALLAS/PMB/EFF_BW.tar.gz, June 2004
- [12] Michael Welcome, Paul H. Hargrove. MVICH Documentation, <http://old-www.nersc.gov/research/ftg/mvich/index.html>, August 2001
- [13] Andrew Lumsdaine, Jeff Squyres, Brian Barrett, Prashanth Charapalli, Amey Dharurkar, Prabhajan Kambadur, Vishal Sahay, Nihar Sanghvi, Sriram Sankaran, Shashwat Srivastav. The LM/MPI Homepage, <http://www.lam-mpi.org/>, May 2004
- [14] W. Gropp, E. Lusk, N. Doss, A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard, In Parallel Computing Volume 22, January 1996
- [15] M. Bertozzi, M. Panella, M. Reggiani. Design of a VIA based communication protocol for LAM/MPI suite, In 9th Euromicro Workshop on Parallel and Distributed Processing, September 2001
- [16] Intel, Microsoft, Compaq, VI Architecture Specification, <http://www.viarch.org>, 1998