Multiple devices under MPICH

Sven Schindler, Wolfgang Rehm
({svsc,rehm}@informatik.tu-chemnitz.de)

Technische Universität Chemnitz

Fakultät für Informatik

Straße der Nationen 62, 09107 Chemnitz

Abstract

Rapid developments in networking technology and a rise in cluster computing have driven the construction of clusters of cluster. Obviously there is a need for a portable, easy–to–use, efficient software environment for those cluster type.

In recent years, the MPICH implementation of MPI has become a widely used message passing interface because of its powerful and efficient layered approach to simplify porting MPI to new systems. Thereby the so called Abstract Device Interface (ADI) takes a key position.

In this paper we describe an enhanced MPICH architecture. Whereas other MPICH implementations support only one communication medium for internode communication at a time our enhanced MPICH implementation supports different one's too.

The basic idea is to introduce a so–called multidevice besides individual devices, so called subdevices, each of them supporting a certain communication medium.

Some relevant aspects of our implementation are discussed. Finally, performance measurement shows that when introducing the general concept of a multidevice we scarcely have to put up with performance losses.

I. Motivation

Our research group investigates clusters of cluster as shown in Figure 1. All nodes are connected with an unique network. Typically, a partial cluster has it's own high–speed interconnection that may be different from the other ones.

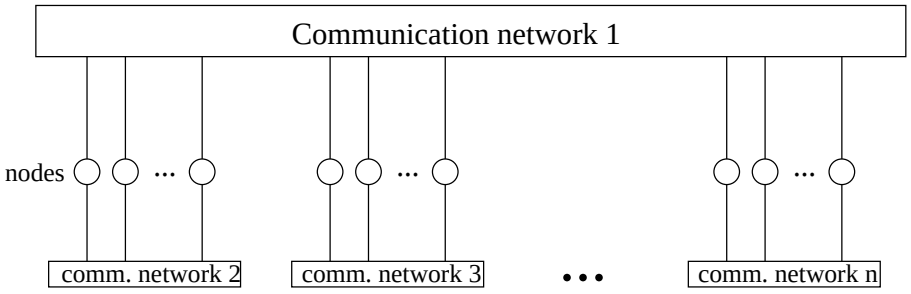

Fig. 1. The investigated cluster architecture

An example for such a type of cluster is the so called OSCAR (**O**pen **S**calable **C**luster **Ar**chitecture)[1][2] that is used at our department as a research prototype. As depicted in Figure 2 all nodes are connected with Fast Ethernet. Additionally, a certain number of nodes forming a partial cluster are connected by Myrinet[3], others using the Scalable Coherent Interface[4][5] to accelerate their communication paths.

The applications came from the area of mathematics and physics and are based on MPI.

An application may use a partial cluster only as well as the total cluster. When using a partial cluster as it were stand–alone only various MPI implementations are available for homogenous networking, e.g. ScaMPI[6] for SCI or MPICH–PM/CLUMP[7] for Myrinet. If an application would

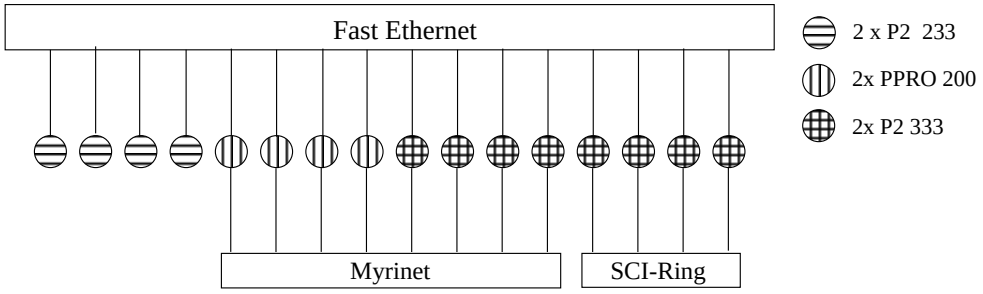

Fig. 2. The example implementation OSCAR IV

like to use the total cluster the things are changing. Currently, there is no MPI–implementation which supports clusters of cluster as shown in Figure 1.

The goal of this work is to develop a portable, easy–to–use MPI implementation for heterogenous networks which allows the shared usage of the existing communication networks. An additional goal is the extensibility of the implementation, what means, supporting new communication hardware can be done with small effort.

A. The structure of MPICH

MPICH (**M**essage **P**assing **I**nterface **CH**ameleon)[8] is a result of a cooporation of the Argonne National Laboratory and of the Mississippi State University. The project was started the same time the MPI–1 standard[9] was developed, so MPICH became a reference implementation. Today MPICH is the most common implementation of the MPI-1 Standard.

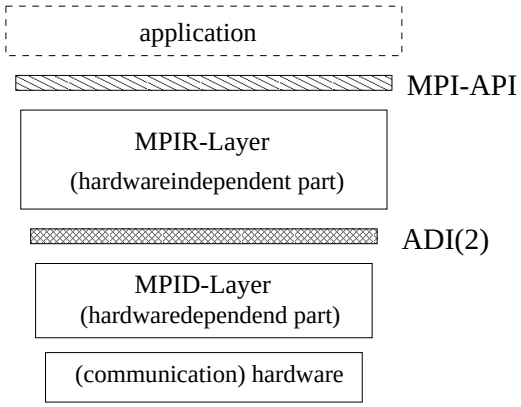

Fig. 3. Structure of MPICH

In Figure 3 the internal structure of MPICH is shown. MPICH devides the MPI–1 functionality in a hardwaredependent and a hardwareindependent part (layer). Due to this structure it is possible to generate a MPI implementation for a new (or not yet supported) communication hardware with small effort.

The hardware independent layer contains the following functionality:

- Administration of process groups, communicators and contexts
- Building and administration of MPI–datatypes
- Building of virtual topologies
- Optional global operations

The hardwaredependent layer is responsible for point–to–point communication. Within this layer a realization of global operations is possible and useful, if that way the performance in comparision to the implementation in the hardwareindependent layer can be increased. Futhermore,

things like pack/unpack operations and timing routines should be provided by the so called device.

The interface between hardwaredependend layer and hardwareindependent layer is called Abstract Device Interfacee (ADI). With the completion of MPICH 1.0.13 the developement of ADI–2[10] (an enhanced version of ADI) was finished, too. We have used the ADI–2 as a general base for our implementation.

II. The multidevice

A. The general approach

For a better understanding of the following explanations it is necessary to imagine a parallel MPI–application on a cluster of workstations. Such an application mostly runs on several nodes, maybe on each node some local tasks. For communication between MPI–tasks on a single SMP node shared memory should be used and communication between different nodes may be carried out using the fastest available network.

The MPI–API and likewise the ADI–2 expects a network service, that exhibits a flat fully–connected structure, at least virtualy. Like shown in Figure 1 the real communication network structure is much more complicated, therefore a so called multidevice that maps the real network structure to the flat virtual structure, was introduced.

An "ordinary" ADI–2 device performs only the mapping of an unique network to a flat virtual structure. In such a case there is only one communication device. Whereas in a heterogenous network several principles, e.g. global shared memory (SCI) as well as packet or stream based ones, are available for increased performance. This is why the multidevice has to support several devices, thus an auxiliary interface within multidevice is needed.

Furthermore an importent goal of the multidevice should be not to loose performance due to underlying layers.

III. Structure of the multidevice

Figure 4 shows the structure and the integration of the multidevice in the MPICH environment.

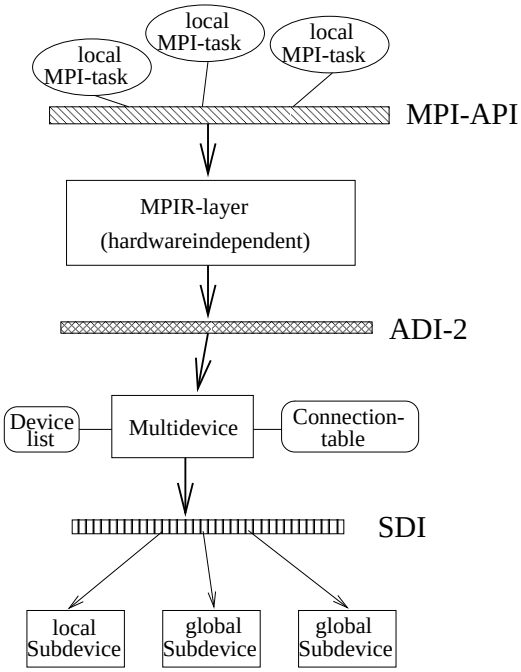

Fig. 4. integration of the multidevice in MPICH

From viewpoint of MPIR–layer the multidevice just connects several MPI–tasks on different nodes with "any" network. Internally, this network consists of several networks. In accordance to the explanations above II-A the multidevice provides an interface to several subdevices. The interface that exhibits a subdevice to the multidevice is a well–defined function set, called **S**ub**D**evice **I**nterface (SDI). The SDI is very similar to the ADI–2. The creation of a subdevice from a "ordinary" ADI–2 Device can be performed with small effort. Mainly the SDI contains routines for data transfer, for management of communication requests and for management of the subdevice (initialization, termination and other stuff). The major differences between SDI and ADI–2 is semantic of the routines for initialization and communication which are specified in the next section. A complete description of the SDI can be found in [11]. The creation of a subdevice is described in section V. The multidevice is fully compliant with the ADI–2 specification, so that further usage of the multidevice is guaranted on following versions of MPICH.

The structures `devicelist` and `connectiontable` are required for the correct work of the multidevice. Selecting the required subdevices must be done during runtime. Therefore subdevices are bound dynamically. An entry of the `devicelist` contains a pointer for each function of a subdevice. The `deviceslist` ist the only way to call a subdevice function.The `connectiontable` contains all informations needed for communication between any MPI–task, for instance which subdevice to use for a dedicated message size. A node specific configuration file makes the data for `devicelist` and `connectiontable` available.

All components below the ADI–2 are developed by the TU Chemnitz. The multidevice concept was developed in a framework of a diploma thesis[11]. Currently available are a multithreaded subdevice for local communication as well as a SCI subdevice. Work on the multithreaded device was started in [12], port to ADI–2 in [13] , and a transformation into a subdevice was done within this work. For communication via different nodes, a subdevice for SCI, that is based on a former ADI–2 device was developed by our resaerch team.

IV. Some aspects of the implementation

The realization of the multidevice lead to some problems. The first problem was to find an algorithm for startup of the whole system. That includes the start of all MPI–tasks and the initialization of the whole network (see IV-A). Another important problem is the design of the communication protocols (see IV-B). If between two nodes no direct connection exists, the concept of indirect communication (see IV-C) offers a solution to further usage of the multidevice.

A. Initialization

After start of a MPI application (under MPICH) only one MPI–task is active. This MPI–task is responsible for the start of all other MPI–tasks. Therefore the first MPI–task is called master. The initialization works as follows:

1. Reading of the procgroup file (master only)
2. Starting a process on every participating node (master only)
3. Distribution of the network information
4. Reading of the configuration file
5. Loading of the subdevices
6. Starting of all local MPI–tasks
7. Initialization of the subdevices for global communication (1 task per node)
8. Initialization of the global data

The configuration file contains the node specific data for the `connectiontable` and the names of the used subdevices. The building of a configuration file is described in section VI. A socket connection which was established in step 2 is used for the distribution of the network information. The integration of subdevices is done dynamically at runtime using a simple mechanism (see

section V-C). For start of local subdevices the multidevice is responsible. Meanwhile the local subdevice starts the additional local MPI–tasks, thus the local subdevice gets a special status.

B. Communication protocols

The classification in communication with or without (simple communication) receive from **MPI_ANY_SOURCE** is based on the fact, that the receive from **MPI_ANY_SOURCE** is difficult in this MPI implementation and not only in this.

The only problem of simple send/receive is to decide, which subdevice should be used. The decision is made by courtesy of the `connectiontable` which contains the network information. The criteria are the global rank of the communication partner and the message size. If a request is used for nonblocking communication the selected subdevice should be saved in the request because after the initialization the access to a nonblocking communication operation is possible only by means of the request.

The general problem of the receive from **MPI_ANY_SOURCE** is to decide which subdevice has to be used for a receive operation, because for such a decision the rank of the sender is required and with **MPI_ANY_SOURCE** not given.

In the blocking case a simple solution is possible and no relevant changes in the subdevices are required. The ADI–2 offers the function **MPID_Iprobe** which allows a nonblocking test regardless whether a message can be received or not. Each subdevice has a corresponding function. By courtesy of this function the multidevice tests the subdevices in sequence. This process stops when a subdevice signals that the message can be received. Thereafter the multidevice calls the receive function of the signaling subdevice.

The nonblocking case is a little bit tricky, because the receive has to return if the receive is not possible immediatly. First the multidevice tests if the message is already available. If not, the multidevice duplicates the request so that every subdevice gets its own copy of the request[1]. Because only one subdevice is allowed to receive the message, all copies of the request get a shared mutex. If a subdevice is ready to receive, first it has to lock the mutex to indicate that the request is already in use. If the lock fails, an other subdevice was faster and the subdevice has to look for another request. If the lock succeeds, the subdevice receives the data and fills in the status information. After that the subdevice calls a specific multidevice function (called **MULTI_CancelAnyRequest**), that dequeues the duplicated request in the subdevices by the courtesy of **RecvCancel**[2]. Thereafter status information is copied into the original request and the original request is marked complete.

C. System messages and indirect communication

The current multidevice supports only cluster architectures as described in section, Figure 1. It is required that a direct connection exists between any 2 nodes and between any 2 MPI–tasks likewise.

For reason to support clusters as shown in Figure 5[3], too, the concept of indirect communication is introduced. Another advantage of indirect communication is the fact, that it is sometimes better to use 2 very fast connections instead of a slow one. Presently the indirect communication is in the stage of a concept only. The term indirect has the meaning "over an intermediate node".

Let's suppose a message should be transfered from a MPI–task on node A to a MPI–task on node C via the intermediate node B[4]. It is possible for node C to receive this message from node B and for node A to send this message to node B, but node B knows nothing about such a

[1] The multidevice keeps the orginal
[2] This function corresponds with **MPID_RecvCancel** from the ADI–2.
[3] Now it is not required that a connection have to exist between any 2 nodes.
[4] It is possible to use several intermediate nodes.

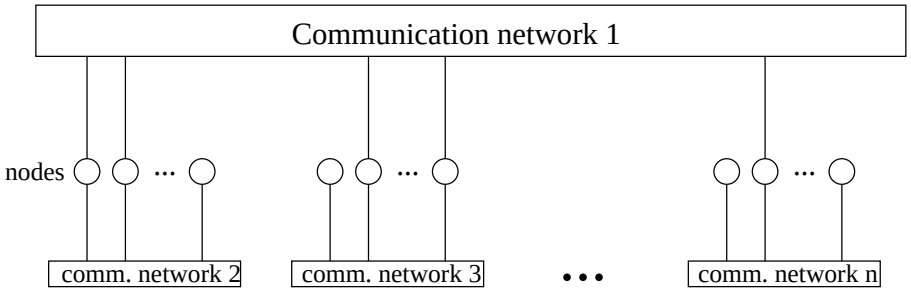

Fig. 5. The enhanced cluster architecture

communication and so it does nothing. A solution for this problem is to use so called system messages.

A system message is a special message, which is not initiated by the application. It is started explicitly on sender side, the receive happens implicitly. The principle of the system messages is similar to the active messages[14]. Principally, a system message is a nonblocking operation what implies the requirement of a message request. On sender side a system message is marked by a negative tag. The subdevice is obligated to transfer such a message instantly. Because the receive is implicit the subdevice must provide a receivebuffer for the system message. When receive of the data is finished the system message handler of the multidevice is called. The system message handler executes an action which is specified by the tag. After the return of the system message handler the subdevice is allowed to free the receivebuffer. The condition for the correct work of system messages is to implement a subdevice as a thread, because otherwise the receive, and consequently the system message handler, couldn't start immediatly.

When using indirect communication on sender side the multidevice packs userdata in a buffer and prefixes a header with own rank, rank of the destination and original tag. By courtesy of a system message this buffer is transfered to the intermediate node. For this purpose the message gets a specific negative tag that indicates that the message as a part of indirect communication. The system message handler on the intermediate node initiates the transfer of the message to the destination task. Therefore it uses nonblocking communication, so it has to create a request. The destination task receives the message as usual and sends the acknowledgement via a system message to the intermediate node, which forwards the acknowledgement to the source node. The system message handler on the source node marks the request complete in nonblocking case or wakes up the waiting MPI–task in blocking case.

When using indirect communication we have to respect that not only the transmission from one subdevice to another on the intermediate node needs time, but also the load on the rather not involved intermediate node increases. Therefore the advantage of indirect communication has to be evaluated in every case.

V. Subdevice — changes from a ADI–2 device

This section describes the modifications, which have to be made to transform an "usual" ADI–2 device (described in [10]) into a subdevice.

The fundamental difference between an ADI–2 device and a subdevice is the fact, that a subdevice requires to be multithread–safe. This is necessary, because several local MPI–tasks (e.g. threads) can call a function of the subdevice simultaneously. This requires that structures, which could be written by several threads, have to be protected.

A. Initialization

The initialization routine is the main difference between subdevices for local communication and subdevices for global communication. While a local subdevice is responsible for starting the local MPI–tasks, the initialization routine of a global subdevice has no permission to start new MPI–tasks.

In contrast to its predecessor ADI–2 device the subdevice doesn't set up data from the MPIR–layer. This is the task of the multidevice.

During the creation of a subdevice for global communication some more changes are necessary. The first problem is the determination of all nodes, on which the subdevice is active. For this purpose the multidevice provides an auxiliary function called **GetDeviceNodeList**. In an "ordinary" ADI–2 device the first active process starts a process on every node involved in the MPI–application. Thereby the first process has the facility to deliver some needed parameters with the startup to the new process and consequently to the device. Because a subdevice for global communication is forbidden to start new processes, a new solution for the transfer of parameters has to be found. If the subdevice is active on the masternode, the socket connection of the multidevice may be used. Otherwise a new connection has to be established.

B. Send and receive

The point–to–point communication in subdevices for local communication doesn't need significant changes.

Because several local MPI–tasks share the multidevice, it's useful to think about the management of communication requests. The following solutions are conceivable:
1. All requests are held in one common structure. That means, that this common structure must be protected for concurrent accesses, especially for concurrent writing. In this solution only the locking of the complete structure is possible.
2. Every local MPI–task has an own structure for the management of communication requests. So concurrent accesses are prevented as far as possible.

I prefer the second solution, because the finegranular locking stands for better performance in comparision with the first solution. In the first solution the risk of deadlocks additionally exists.

If indirect communication should be used, it's neccessary to implement system messages. The precondition for the use of system messages is the fact, that the subdevice must be implemented as a thread.

C. Binding to the multidevice

The subdevice has to provide a function named *Subdevicename*_**LoadDevicePtr**. The function fills an entry of the devicelist with pointers to the functions of the subdevice. The function will be called in the initialization routine by the multidevice (in step 5).

VI. Configuration of the multidevice — mdconfig

Besides the data of the procgroup file, the multidevice needs information about network and used subdevices. Therefore every node reads a specific configuration file in the initialization process. It's very expendable and fault–prone to write this configuration files for every node by hand. For this reason the tool mdconfig[5] was developed.

The Figure 6 shows the functioning of the tool mdconfig. The input of the tool is the global description of the network. The description contains available subdevices, performance of these subdevices and all connections that exist in the network. The parser produces a weighted graph and by courtesy of a modified algorithm by Dijkstra the fastest connections and the subdevices,

[5]The name stands for **m**ulti**d**evice **conf**igurator.

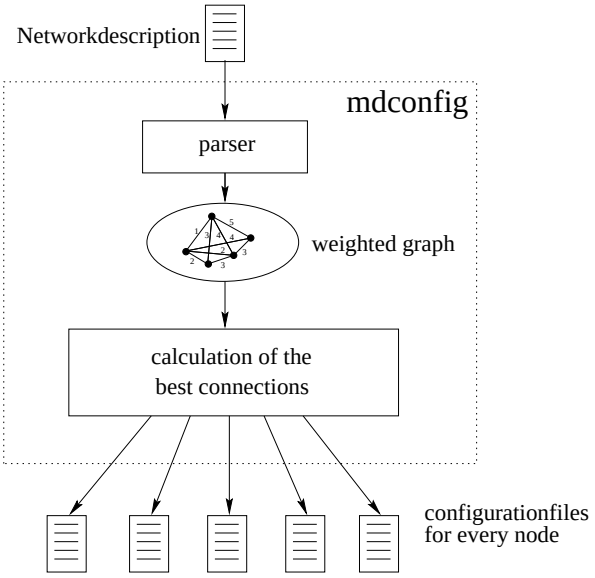

Fig. 6. functioning of mdconfig

which have to be used for the connection. Then these informations are saved for every node in a specific file.

The network parameters in the input file refer to 3 different message size ranges, so it's possible to use several subdevices between the same 2 MPI–tasks for different message sizes. The control of indirect communication (if implemented) is possible with a weighted value for the transmission of a message between 2 subdevices. With a large value indirect communication will be switched off practically for the case, that transfer form node A to node B plus transfer from node B to node C is faster than transfer from node A to node C.

More information about the tool mdconfig and an entire description of the configuration language can be found in [11].

VII. Results

The functionality and correctness of the MPI–implementation (of course without indirect communication) was tested by means of the MPICH testsuite.

message size in Byte	multidevice transfer time in ns	ADI–2 device transfer time in ns
1	51.8	50.8
16	52.0	51.3
64	52.2	51.5
256	52.7	51.8
1024	54.6	55.7
32768	208	199
524289	6184	6159

TABLE I
TRANSFER TIME: MULTIDEVICE VS. ADI–2 DEVICE (MT)

A measurement, which shows that the multidevice is no "performanceeater" can be seen in the comparision of the multithreading subdevice and its predecessor, the usual ADI–2 device (Table

I). The measurement shows a simple pingpong, which is based on usual blocking communication. In the diagram only the bandwidth for messages from 1 Byte to 256 Bytes is shown, because on larger messages the time for the initialization of a communication operation is much smaller than the time for the data transfer. Because the transfer mechanism was not changed during the transformation of the ADI–2 device into the subdevice (and so the time for the pure transfer will be the same), a deterioration of the perfomance must be visible especially on small messages. The diagramm shows, that the bandwidth of small massages is nearly equal to the ADI–2 device, so that the initialization of a communication operation in multidevice plus subdevice needs nearly the same time like in a pure ADI–2 device. This means that the additional effort in the multidevice is very small.

VIII. Further work

The next steps in the creation of our MPI implementation are the developement of subdevices for Myrinet and Fast Ethernet(TCP/IP). For this developement ADI–2 devices for Myrinet (see [15]) und FastEthernet (the ch_p4 device from MPICH) are the basic work. Another work to do is to adapt the SCI subdevice on the actual ADI–2 device and so to improve the performance of the SCI connections. After this is done, the resulting MPI implementation supports all communication networks of our cluster called OSCAR. The next step is the implementation of system messages and indirect communication. Thereafter practical investigations about indirect communication are possible.

First tests have shown, that for instance the implementation of a barrier in the MPIR–layer is suboptimal in heterogenous networks. So investigations about the implementation of some global operations are useful.

References

[1] Carsten Dinkelmann, Wolfgang Rehm, and Marko Meyer. Eine Konfiguration– und Managementlösung für ein dediziertes Linux–Cluster. *Tagungsband zum 2. Workshop Cluster–Computing, Chemnitzer Informatik Berichte CSR-99-02, TU Chemnitz, Fakultät für Informatik*, pages 95–103, March 1999.

[2] http://www.tu-chemnitz.de/informatik/RA/projects/oscar/oscar.html.

[3] http://www.myri.com/.

[4] IEEE. IEEE Standard for Scalable Coherent, Interface (SCI) (Std. 1596 – 1992). The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, August 1993.

[5] Dolphin Interconnect Solutions. PCI–SCI Cluster Adapter Specification, May 1996.

[6] http://www.scali.com/.

[7] http://pdswww.rwcp.or.jp/mpich-pm/doc/index.html.

[8] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High–Performance, Portable Implementation of the MPI message–passing standard. Argonne National Laboratory and Mississippi State University, 1996.

[9] Message Passing Interface Forum. MPI: A message–passing interface standard Vers. 1.1. http://www.mcs.anl.gov/mpi/standard.html, June 1995.

[10] William Gropp and Ewing Lusk. MPICH Working Note: The Second–Generation ADI for the MPICH Implementation of MPI. Argonne National Laboratory, Mathematics and Computer Science Division, 1996.

[11] Sven Schindler. Entwurf und Implementierung eines ADI–2 Multidevices. Diplomarbeit,TU Chemnitz, Fakultät für Informatik, Lehrstuhl Rechnerarchitektur, March 1999.

[12] Uwe Beyer. Optimierung einer multithreaded MPI–Implementierung. Diplomarbeit,TU Chemnitz, Fakultät für Informatik, Lehrstuhl Rechnerarchitektur, April 1997.

[13] Sven Schindler. Weiterentwicklung der MPICH–Implementation für SCI–cluster. Studienarbeit, TU Chemnitz, Fakultät für Informatik, Lehrstuhl Rechnerarchitektur, November 1997.

[14] Thorsten von Eiken, David E. Culler, Seth Copen Goldstein and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architekture*, 1992. available from `http://www.cs.cornell.edu/Info/Projects/CAM/isca92.ps`

[15] Carsten Dinkelmann. Implementierung einer effizienten MPI–Schnittstelle für Myrinetkarten auf der Basis von Fast Messages. Diplomarbeit, WH Zwickau, FB Physikalische Technik/Informatik, March 1999.