# MPI-GLUE: Interoperable high-performance MPI combining different vendor's MPI worlds

Rolf Rabenseifner
Technical Report, April 1998

Rechenzentrum der Universität Stuttgart, Allmandring 30, D-70550 Stuttgart,
rabenseifner@rus.uni-stuttgart.de,
WWW home page: http://www.hlrs.de/people/rabenseifner/

**Abstract.** Several metacomputing projects try to implement MPI for homogeneous and heterogeneous clusters of parallel systems. MPI-GLUE is the first approach which exports nearly full MPI 1.1 to the user's application without losing the efficiency of the vendors' MPI implementations. Inside of each MPP or PVP system the vendor's MPI implementation is used. Between the parallel systems a slightly modified TCP-based MPICH is used, i.e. MPI-GLUE is a layer that combines different vendors' MPIs by using MPICH as a global communication layer. Major design decisions within MPI-GLUE and other metacomputing MPI libraries (PACX-MPI, PVMPI, Globus and PLUS) and their implications for the programming model are compared. The design principles are explained in detail.

## 1 Introduction

The development of large scale parallel applications for clusters of high-end MPP or PVP systems requires efficient and standardized programming models. The message passing interface MPI [11] is one solution. Several metacomputing projects try to solve the problem that mostly different vendors' MPI implementations are not interoperable. MPI-GLUE is the first approach that exports nearly full MPI 1.1 without loosing the efficiency of the vendors' MPI implementations inside of each parallel system.

## 2 Architecture of MPI-GLUE

MPI-GLUE is a library exporting the standard MPI 1.1 to parallel applications on clusters of parallel systems. MPI-GLUE imports the native MPI library of the system's vendor for the communication inside of a parallel system. Parallel systems in this sense are MPP or PVP systems, but also homogeneous workstation clusters. For the communication between such parallel systems MPI-GLUE uses a portable TCP-based MPI library (e.g. MPICH). In MPI-GLUE the native MPI library is also used by the portable MPI to transfer its byte messages inside of each parallel system. Fig. 1 shows this design which allows any homogeneous and heterogeneous combination of any number of parallel or nonparallel systems.

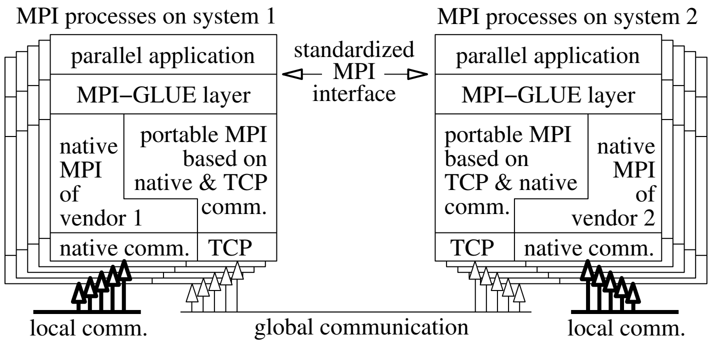


**Fig. 1.** Software architecture

## 3 Related Works

The **PACX-MPI** [1] project at the computing center of the University of Stuttgart has a similar approach as MPI-GLUE. At Supercomputing '97 PACX-MPI was used to demonstrate that MPI-based metacomputing can solve large scale problems. Two CRAY T3E with 512 processors each were combined to run a flow solver (URANUS) which predicts aerodynamic forces and high temperatures effecting on space vehicles when they reenter Earth's atmosphere [2]. Furthermore a new world record in simulating an FCC crystal with 1.399.440.00 atoms has been reached on that cluster [13]. PACX-MPI uses a special module for the TCP communication in contrast to the portable MPI in MPI-GLUE. The functionality of PACX-MPI is limited by the MPI subset of this module. PACX-MPI, therefore, implements only a small subset of MPI 1.1. The TCP communication does not directly exchange messages between the MPI processes. On each parallel system two special nodes are used as concentrators for incoming and outgoing TCP messages. This is no bottleneck as long as such a daemon node is able to communicate with full speed with the network interface and without significant additional latency by forwarding the messages to or from the application node. PACX-MPI can compress all TCP messages to enlarge bandwidth.

**PVMPI** [4, 5] has been developed at UTK, ORNL and ICL, and couples several MPI worlds by using PVM based bridges. The user interface does not meet the MPI 1.1 standard, but is similar to a restricted subset of the functionality of MPI_Comm_join in MPI 2.0 [12]. It is impossible to merge the bridge's inter-communicator into a global intra-communicator over all processes. Therefore, one can not execute existing MPI applications with PVMPI on a cluster of parallel systems.

In the **PLUS** [3] project at PC$^2$ at the University of Paderborn a bridge has been developed too. With this bridge MPI and PVM worlds can be combined. Inefficient TCP implementations of some vendors have been substituted by an own protocol based on UDP. It is optimized for wide area networks. As in PACX-MPI the communication is concentrated by a daemon process on each parallel system. The interface has the same restrictions as described for PVMPI, moreover, only a restricted subset of datatypes is valid. PLUS is part of the MOL (Metacomputer Online) project.

In the **Globus** [6] project and in the **I-WAY** [8, 10] project MPICH [9] is used on top of the multi-protocol communication library NEXUS [7]. NEXUS is designed as a basic module for communication libraries like MPI. On the parallel system itself MPICH is used instead of the vendor's MPI library.

# 4 Design Decisions and Related Problems

Several decisions must be taken in the design of an MPI library usable for heterogeneous metacomputing. Major choices and the decisions in the mentioned projects are viewed in this section.

## 4.1 The glue approach versus the portable multi-protocol MPI implementation

The goals of high efficiency and full MPI functionality can be achieved by the following alternatives:

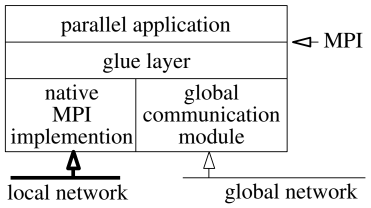


**Fig. 2.** The general glue approach

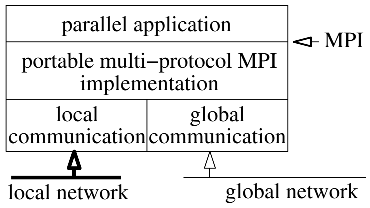


**Fig. 3.** The multi-protocol approach

(a) The general glue approach is shown in Fig. 2. The glue layer exports an MPI interface to the application and imports the native MPIs in each parallel system and a global communication module which allows communication between parallel systems. The glue approach has been implemented in MPI-GLUE, PACX-MPI, PVMPI and PLUS.

(b) In the portable multi-protocol MPI approach, shown in Fig. 3, each parallel system has its own local communication module. A common global communication module enables the communication between the parallel systems. The MPI library uses both modules. In this approach the global communication module is usually very small in comparison with the glue approach: it does only allow byte message transfers. The main disadvantage of this approach is that the performance of the native MPI library can not be used. This would, for instance, slow down the latency on our NEC-SX4/32 by a factor of 3.8 (respectively 2.2) and the bandwidth by a factor of 1.6 (2.1) with the MPI/SX, version 7.1 (7.2). Globus has implemented this approach.

### 4.2 Global world versus dynamically created bridges

The standards MPI 1.1 and MPI 2.0 imply two metacomputing alternatives: (a) The group of the predefined communicator MPI_COMM_WORLD can be the collection of *all* processes or (b) on each partition of the cluster a separate MPI_COMM_WORLD collects the partition's processes and inter-communicators are used to combine the partitions:

(a) In MPI-GLUE, PACX-MPI and Globus one MPI_COMM_WORLD collects all processes in all parallel systems after the processes have been started all together. This approach matches MPI 1.1 and all higher versions.
(b) In PVMPI and PLUS there is an MPI_COMM_WORLD inside each parallel system and some inter-communicators connect the systems. These inter-communicators are *bridges* between these several MPI_COMM_WORLDs. The bridges are established dynamically by the application, after the application has successfully initialized its MPI_COMM_WORLD communicator on each parallel system. MPI 2.0 defines two different interfaces for establishing a bridge (MPI_COMM_SPAWN and MPI_COMM_JOIN) and requires the possibility of building a global intra-communicator in several steps by using MPI_INTERCOMM_MERGE, i.e. the application can build a global intra-communicator over all processes that can be used instead of MPI_COMM_WORLD in existing application. But both projects haven't implemented this merge.

### 4.3 Global communication using an own module versus importing a portable MPI

The global communication module in the glue approach can be:

(a) a portable TCP-based MPI implementation (in MPI-GLUE), or
(b) a special TCP-based module (in PACX-MPI, PVMPI and PLUS).

With (a) it is easy to implement the full MPI 1.1 standard, but including the bridge interface depends on the availability of a portable MPI implementation which includes the MPI 2.0 dynamic process start facilities.

With (b) it is difficult to achieve the full MPI 1.1 standard, because this means to reimplement most of the MPI's functionality inside of this special module. Therefore, in PACX-MPI and PLUS there is implemented only a small subset of the MPI 1.1 standard. In PACX-MPI the subset was defined by the needs of typical applications. In PVMPI there is implemented a larger subset (especially a full datatype support) because its bridges are based on the PVM library.

### 4.4 Using daemons or not using daemons

The global communication can be (a) concentrated by daemons (Fig. 4) or can be (b) done directly (Fig. 5). The *daemon-based approach* (a) has the advantage


**Fig. 4.** Daemon based global communication


**Fig. 5.** Global communication with direct links

that system routines are not called for global communication in the application processes. On PVP and MPP systems the ratio between the latency of a global and a local communication routine may be about 100:1! The overlay of computation and communication, therefore, can be implemented in a more efficient way. The latency of the TCP write and read routines may be hidden.

Furthermore, the number of TCP ports and connections is reduced to one bidirectional daemon or two unidirectional daemons for each global communication link (this may simplify problems occuring with firewalls or limits of operating systems).

On the other hand, this approach has also disadvantages: Some extra processes are needed for running the daemons. This may be a problem if the application needs the node number being quadratic or a power of two, while the user wants to use all processors (compute nodes). Then there must exist enough extra nodes (e.g. command nodes) which can be used as daemons. The daemon nodes should be able to transfer data from the local to the global network and vice versa, sustained with the full bandwidth of the global network. This approach has been implemented in PACX-MPI, PLUS and PVMPI configured with daemons.

The *direct-communication approach* (b) may establish TCP links between each pair of nodes in different parallel systems, if they exchange messages. This approach is to be preferred if the differences between local and global communication are small, e.g. in WAN-based clusters of LAN-based workstation clusters. This approach has been implemented in MPI-GLUE, PVMPI configured without daemons, and Globus.

(a) implies the extra nodes are used as daemons, and (b) implies that they are used as additional application nodes. In the first case all TCP delays are handled in the extra nodes, in the second case the extra nodes compute parts of the application and the TCP delays are handled by **some** application nodes. Which of the both approaches is more efficient depends on the cluster configuration, the number of application nodes that make global communication, whether the daemons are overloaded with TCP latencies or not, and whether the application issues a lot of MPI_Tests or MPI_Iprobes.

In PLUS an additional reason for the daemon approach is, that the UDP-based transport protocol implementation needs daemons to be able to handle necessary retransmissions. By using TCP (in all other projects) this should be handled efficiently by the operating system.

### 4.5 The first progress problem

All testing and blocking MPI calls that have to look at processes connected by the *local* network *and* at other processes connected by the *global* network should take into account that the latencies for probing the local and global network are different. On clusters of MPPs the ratio can be 1:100. Examples are a call to MPI_Recv with source=MPI_ANY_SOURCE, or a call to MPI_Waitany with requests handling local *and* global communication.

More a workaround than a solution are asymmetric polling strategies: if polling is necessary then only in the $n^{\mathrm{th}}$ round the central polling routine will inquire the state of the global interface, while in each round the local interface will be examined. $n$ is given by the ratio mentioned above. Without this trick the latency for local communication may be expanded to the latency of the global communication. But with this trick the latency for global communication may be in the worst case twice the latency of the underlaying global communication routine.

This first progress problem arises in all systems that use no daemons for the global communication (MPI-GLUE, PVMPI configured without daemons, and Globus). In MPI-GLUE the asymmetric polling strategy has been implemented. In Globus it is possible to specify different polling intervals for each protocol. This problem should not exist in daemon-based systems (PACX-MPI, PLUS and PVMPI configured with daemons), as long as they really use an optimal communication method between daemon and application processes (e.g. the native MPI).

### 4.6 The second progress problem

The restriction in MPI 1.1, page 12, lines 24 ff ("This document specifies the behavior of a parallel program assuming that only MPI calls are used for communication") allows that MPI implementations make progress on non-blocking and buffered operations *only inside* of MPI routines, e.g. in MPI_Wait... / _Test... / _Iprobe or _Finalize. This is a weak interpretation of the progress rule in MPI 1.1, page 44, lines 41ff. The rationale in MPI 2.0, page 142, lines 14-29 explicitely mentions this interpretation. This allows that a native MPI makes no progress, while the portable MPI is blocked by waiting for a message on the TCP link, and vice versa.

Therefore, in the glue approach the problem arises that the glue layer must not implement anything by blocking or testing only within the native MPI without giving the global communication module a chance to make progress, and vice versa. The only *hard problem* is the combination of using collective routines of the native MPI (because there isn't any non-blocking alternative) *and* using sends in the global MPI *and* these sends are buffered at the sender *and* the sender is blocked in the native collective operation. This problem can be solved in daemon-based systems by sending all messages (with a destination in another parallel system) immediately to the daemons and buffering them there (implemented in PACX-MPI and PLUS). It can also be solved by modifying the central probe and dispatching routine inside the native MPI.

The analogous problem (how to give the native MPI a chance to make progress, while the glue layer blocks in a global collective routine) usually only exists, if one uses an existing MPI as a global communication module, but in MPI-GLUE this problem has been solved, because the global MPI itself is a multi-protocol implementation which uses the local MPI as one protocol stack.

## 5 Design Details

The notation **"MPI"** is used for the MPI interface exported by this glue layer and for the glue layer software.
The notation **"MPX"** is used for the imported TCP-based MPI-1.1 library in which all names are modified and begin with "MPX_" instead of "MPI_". MPX allows the e**x**change of messages between all processes.
The notation **"MPL"** is used for the native (**l**ocal) MPI 1.1 library. All names are modified ("MPL_" instead of "MPI_"). All MP**L**_COMM_WORLDs are disjoint and in the current version each process is member of exactly one MP**L**_COMM_WORLD (that may have any size$\geq$1).

The MPI process start and MPI_Init is mapped to the MPL process start, MPL_Init, MPX process start and MPX_Init. The user starts the application processes on the first parallel system with its MPL process start facility. MPI_Init first calls MPL_Init and then MPX_Init. The latter one is modified to reflect that the parallel partition is already started by MPL. Based on special arguments at the end of the argument list and based on the MPL ranks the first node recognizes inside of MPX_Init its role as MPICH's *big master*. It interactively starts the other partitions on the other parallel systems or uses a *remote shell* method (e.g. `rsh`) to invoke the MPL process start facility there. There as well the MPI_Init routine first invokes MPL_Init and then MPX_Init. Finally MPI_Init initializes the predefined handles. MPI_Finalize is implemented in the reverse order.

Further details are given in the following table. **"X"** and **"L"** are abbreviations for MPX and MPL.

| MPI constant, handle, routine | impl.ed by using | remarks |
|---|---|---|
| ranks | X | i.e. the MPI ranks are equal to the MPX ranks. |
| group handling | X | i.e. group handles and handling are impl. with MPX. |
| communicators | L and X | and *roots-communicator*, see Chap. 6. |

| MPI constant, handle, routine | impl.ed by using | remarks *continued* |
|---|---|---|
| datatypes | L and X | and a mapping of the L ranks to the X ranks and vice versa. |
| requests | L or X or delayed | MPI_IRECVs with MPI_ANY_SOURCE and communicators spawning more than one MPL region store all arguments of the IRECV call into the handle's structure. The real RECV is delayed until the test or wait calls. |
| NULL handles | 0 | except MPI_GROUP_NULL, see group handling above. |
| MPI_STATUS | official entries and L or X | While copying an MPX status to the official entries, one must check for MPX_ANY_SOURCE/TAG and switch it to MPI_ANY_SOURCE/TAG. While copying an MPL status to the official entries, one must convert the MPL source rank into the MPI source rank, if it is a receive status. |
| MPI_BOTTOM/ _UNDEFINED/... | L | They must be mapped to the corresponding MPX values if they are transferred by the GLUE to MPX routines. |
| sends | L or X | |
| receives | L or X | (only with known source partition). |
| receives with ANY_SOURCE: | | |
| - blocking | L and X | with asymmetric polling strategy. |
| - nonblocking | GLUE | postponed until the wait and test routines, see MPI requests above. |
| wait and test | L and X | or "L or X". |
| MPI_Testall | | Testing inside partial areas of the communicator may complete request handles, but the interface definition requires that only none or all requests are completed. Therefore (only for Testall), requests may be MPL/MPX-finished, but not MPI-finished. |
| MPI_Pack, MPI_Unpack | X | This causes the restriction that typed messages can not be received with datatype MPI_PACKED and then explicitly unpacked with MPI_Unpack, and vice versa. |
| MPI_Wtime, MPI_Tick | L | This causes that the clock is not globally synchronized, but MPL may be faster and more precise than MPX. |
| MPI_Op_create and oper. handles | GLUE | For each user defined operation a special wrapper is needed in the glue layer, because the callback function's datatype argument has different definitions in MPI and MPL/X. |
| MPI_Abort | X | but always with MPI_COMM_WORLD. |
| user buffer | L | and a second malloc'ed duplicated buffer is used for X; therefore, MPI_ERR_INTERN may be returned due to insufficient space for malloc. |
| attribute caching | X | Using MPX for attribute caching simplifies the implementation of the topology functions, because a topology is a special attribute in the sense of the MPI standard. |
| topology | X | i.e. the glue layer does not use any (possibly) native topology support. |
| error codes | L or X | MPI error codes are MPL_SUCCESS or an MPL error code or an MPX error code plus MPL_ERR_LASTCODE. |

| MPI constant, handle, routine | impl.ed by using | remarks *continued* |
|---|---|---|
| error classes | L | |
| errhandler | GLUE | Special error handlers are installed by the glue layer and the user's error handler is cached in the MPI communicator, because the MPI_Errhandler_set callback function argument can not be directly mapped, due to the different definitions of MPI_Comm and MPL/X_Comm. |

Table 1: Design details

## 6 Optimization of Collective Operations in the GLUE

It is desirable that metacomputing applications do not synchronize processes over the TCP links. This implies that they should not call collective operations on a global communicators, especially those operations which make a barrier synchronization (barrier, allreduce, reduce_scatter allgather(v), alltoall(v)). The following schemes help to minimize the latencies if the application still uses collective operations over global communicators. In general a collective routine is mapped to the corresponding MPL_... routine *if the communicator is completely inside of one local MPL world*. To optimize MPI_Barrier, MPI_Bcast, MPI_Reduce and MPI_Allreduce *if the communicator belongs to more than one MPL world*, for each MPI-communicator, there is an additional internal *roots-communicator* that combines the first process of any underlying MPL-communicator by using MPX, e.g. MPI_Bcast is implemented by a call to MPL_Bcast in each underlying MPL-communicator and one MPX_Bcast in the roots-communicator.

## 7 Status of the MPI-GLUE Project

MPI-GLUE implements the full MPI 1.1 standard except some (hopefully minor) functionalities. Some restrictions result from the glue approach:

– It is impossible to allow an unlimited number of user-defined operations.
– A message packed with MPI_Pack and sent with datatype MPI_PACKED must not be received by using the original datatype; it must be received by using the same datatype, i.e. MPI_PACKED.
– A message sent with any datatype except MPI_PACKED must not be received by using the datatype MPI_PACKED and then unpacked with MPI_-Unpack.
– Restrictions in the used MPI implementations are mostly valid for the whole glue system as well; e.g. with MPICH the signal SIGUSR1 must not be used by the application and the commandline arguments can be used only after MPI_Init was called.

Some other (minor) functionalities are still waiting to be implemented (state Jan. 1998):

- Not even a small number of user-defined operations can be created by MPI_Op_create.
- MPI_Cart_create and MPI_Graph_create cannot create a communicator that has less processes than the old one.
- The predefined handles are not initialized before MPI_Init is called.
- The inter-communicator handling routines are not implemented yet.
- MPI_Cancel and MPI_Test_cancelled are not implemented.
- A communicator which has been used in creating a request must not be freed before the request is deallocated.
- Only the collective operations MPI_Barrier, MPI_Bcast, MPI_Reduce and MPI_Allreduce are optimized. The other collective routines use the native MPI routines only if the communicator lies entirely in one of the native MPI worlds, otherwise the portable MPI library is used which must not be worse, because the portable MPI library itself uses the native MPI for messages inside of each native MPI world.
- The solution of the second progress problem isn't implemented for any of the supported native MPI libraries.

MPI-GLUE has been ported to the following native MPI environments: (a) MPICH 1.1.0 with device ch_shmem, (b) MPT 1.1.0.4 on CRAY T3E and (c) Intel Paragon (version 1.2.2.2 in mpi.h). MPICH 1.1.0 with device=ch_p4 and comm=shared is used as the TCP-based portable MPI library.

## 8 How to Get the MPI-GLUE Software

MPI-GLUE is not public domain and the development is ongoing. So the best thing you can do to get a copy of the current version of MPI-GLUE is to contact the author.

## 9 Future Plans

As soon as we can get a dynamic process start extension for MPICH, we want to extend MPI-GLUE with an process start facility that conforms to the standard MPI 2.0. A daemon-based communication module plugged into the p4 interface of MPICH may lead to a merge of MPI-GLUE and PACX-MPI.

Collective routines without inherent barrier synchronization can be implemented in a cluster with local MPI's latency: Bcast and Reduce have already been implemented, Gather and Scatter are planned. These routines may speed up the initialization and the finish up of cluster applications on any platform. Collective routines with inherent barrier synchronization used over more than one partition have always at least twice of the global MPI's latency. The optimization may prohibit that the latency is more than theoretically needed. Barrier and Allreduce have already been implemented, Allgather and Alltoall are planned.

The current Fortran interface needs to store C handles in integer variables. An additional handle wrapper must be implemented for systems with 64 bit addresses and 32 bit integers.

## 10 Summary

MPI-GLUE achieves interoperability for the message passing interface MPI. For this it combines existing vendors' MPI implementations losing neither full MPI 1.1 functionality nor the vendors' MPIs' efficiency. MPI-GLUE targets existing MPI applications that need clusters of MPPs or PVPs to solve large-scale problems. MPI-GLUE enables metacomputing by seamlessly expanding MPI 1.1 to any cluster of parallel systems. It exports a single virtual parallel MPI machine to the application. It combines all processes in one MPI_COMM_WORLD. All local communication is done by the local vendor's MPI and all global communication is implemented with MPICH directly, without using deamons. The design has only two unavoidable restrictions which concern user defined reduce operations and packed messages.

## 11 Acknowledgments

The members of the MPI-2 Forum and of the PACX-MPI project as well as the developers of MPICH have helped me in the discussion which led to the present design and implementation of MPI-GLUE. I would especially like to thank Thomas Beisel, Holger Berger, Thomas Bönisch, Matthias Brune, James Cownie, Terry Dontje, Graham E. Fagg, Mark Fallon, Richard Frost, Edgar Gabriel, Al Geist, Bill Gropp, Shane Hebert, Jürgen Lepper, Lloyd Lewins, Rusty Lusk, Gary Oberbrunner, Heidi Poxon, Michael Resch, Eric Salo, Tony Skjellum, Marc Snir, and George Thiruvathukal. And I would like to thank Oliver Hofmann for implementing a first prototype in his diploma thesis, Christoph Grunwald for implementing some routines during his practical course, especially some wait, test and topology functions, and Matthias Müller for testing MPI-GLUE with his P3T-DSMC application.

## References

1. Beisel, T., Gabriel, E., Resch, M.: An Extension to MPI for Distributed Computing on MPPs. In Marian Bubak, Jack Dongarra, Jerzy Wasniewski (Eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, LNCS, Springer, 1997, 75-83. `http://www.hlrs.de/people/resch/PAPER/PVMPI97/pvmmpi.ps` and `http://www.hlrs.de/people/resch/PROJECTS/PACX.html`
2. T. Boenisch, R. Ruehle: Portable Parallelization of a 3-D Flow-Solver, Parallel Comp. Fluid Dynamics '97, Elsevier, Amsterdam, to appear. `http://www.hlrs.de/news/events/pittsburgh.html`

3. Brune, M., Gehring, J., Reinefeld, A.: A Lightweight Communication Interface for Parallel Programming Environments. In Proc. High-Performance Computing and Networking HPCN'97, LNCS, Springer, 1997.
http://www.uni-paderborn.de/pc2/services/public/1997/97002.ps.Z and
http://www.uni-paderborn.de/pc2/projects/mol/plus.htm
4. Fagg, G.E., Dongarra, J.: Interoperating MPI Applications PVMPI. Presentation for MAPINT 97. http://www.cs.utk.edu/~fagg/MSRC/index.html and
http://www.cs.utk.edu/~fagg/pvmpi/
5. Fagg, G.E., Dongarra, J.J.: PVMPI: An Integration of the PVM and MPI Systems. Department of Computer Science Technical Report CS-96-328, University of Tennessee, Knoxville, May, 1996.
http://www.cs.utk.edu/~library/TechReports/1996/ut-cs-96-328.ps.Z
6. Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. International Journal of Supercomputer Applications (to appear). ftp://ftp.mcs.anl.gov/pub/nexus/reports/globus.ps.Z and
http://www.globus.org/
7. Foster, I., Geisler, J., Kesselman, C., Tuecke, S.: Managing Multiple Communication Methods in High-Performance Networked Computing Systems. J. Parallel and Distributed Computing, 40:35-48, 1997. ftp://ftp.mcs.anl.gov/pub/nexus/reports/jpdc2.ps.Z
8. Foster, I., Geisler, J., Tuecke, S.: MPI on the I-WAY: A Wide-Area, Multimethod Implementation of the Message Passing Interface. Proc. 1996 MPI Developers Conference, 10-17, 1996.
ftp://ftp.mcs.anl.gov/pub/nexus/reports/mdc96_iway.ps.Z
9. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Paralle Computing, 22, 1996. http://www.mcs.anl.gov/mpi/mpicharticle/paper.html and
http://www.mcs.anl.gov/mpi/mpich/
10. I-WAY/SCinet on the Supercomputing '95, Dec. 3-8, 1995, San Diego, CA, USA. http://www.sdsc.edu/, http://www.iway.org/ or http://146.137.96.14/, http://www.crpc.rice.edu/CRPC/newsletters/fal95/director.html,
http://www.mcs.anl.gov/home/itf/isoft_user_guide/
11. MPI: A Message-Passing Interface Standard. Message Passing Interface Forum, June 12, 1995. http://www.mpi-forum.org/docs/docs.html and
http://www.mcs.anl.gov/mpi/
12. MPI-2: Extensions to the Message-Passing Interface. Message Passing Interface Forum, July 18, 1997. http://www.mpi-forum.org/docs/docs.html and
http://www.mcs.anl.gov/mpi/
13. Müller, M.: Weltrekorde durch Metacomputing. BI 11+12, Regionales Rechenzentrum Universität Stuttgart, 1997.
http://www.uni-stuttgart.de/Rus/Bi/BI_html/BI97/11+12/file1.html#HDR108

A short version of this paper is publicated under

Rolf Rabenseifner: MPI-GLUE: Interoperable high-performance MPI combining different vendor's MPI worlds. Proc. Euro-Par '98, University of Southampton, Sept. 1-4, 1998.

http://www.hlrs.de/people/rabenseifner/publ/mpi_glue_europar98.ps.Z or .gz

This technical report can be found under

http://www.hlrs.de/people/rabenseifner/publ/mpi_glue_tr_apr98.ps.Z or .gz