Madeleine II: a Portable and Efficient Communication Library for High-Performance Cluster Computing

Olivier Aumage* Luc Bougé* Alexandre Denis* Lionel Eyraud* Jean-François Méhaut*
Guillaume Mercier* Raymond Namyst* Loïc Prylli*

Abstract

This paper introduces Madeleine II*, a new adaptive and portable multi-protocol implementation of the* Madeleine *communication library.* Madeleine II *has the ability to control multiple network interfaces (BIP, SISCI, VIA) and multiple network adapters (Ethernet, Myrinet, SCI) within the same application session. We report on performance measurements obtained using BIP/Myrinet and SISCI/SCI and we present preliminary results about our MPICH/*Madeleine II *and Nexus/*Madeleine II *ports. We also discuss an extension of* Madeleine II *for clusters of clusters which is able to handle heterogeneous networks. In particular, we present the fast internal data-forwarding mechanism that is used on gateway nodes to speed up inter-cluster transmissions. Preliminary experiments show that the resulting inter-cluster bandwidth is close to the one delivered by the hardware.*

1 Introduction

Due to their ever-growing success in the development of distributed applications on clusters of workstations and SMP machines, today's multithreaded programming environments have to be highly *portable* and *efficient* on a large variety of architectures. For portability reasons, most of these environments are built on top of widespread message-passing communication interfaces such as PVM or MPI. However, the implementation of such environments often involves remote service request (RSR), remote procedure call (RPC) or remote method invocation-like (RMI) interactions. This is obviously true for environments providing an RPC-based programming model such as Nexus [6] or PM2 [10], but also for others which often provide functionalities that can be efficiently implemented by RPC operations.

We have shown in [1] that message passing interfaces such as MPI do not meet the needs of RPC-based multithreaded environments with respect to efficiency. Therefore, we have proposed a portable and efficient communication interface, called *Madeleine*, which was specifically designed to provide RPC-based multithreaded environments with *both* transparent and highly efficient communication. However, the internals of the first implementation were strongly message-passing oriented [1]. Consequently, the support of non message-passing network interfaces such as SISCI/SCI [16] or even VIA [4] was cumbersome and introduced some unnecessary overhead. In addition, no provision was made to use multiple networks within the same application. For these reasons, we decided to design *Madeleine II*, a full multi-protocol implementation of *Madeleine*, efficiently portable on a wider range of network interfaces, including non message-passing ones.

Much work has been devoted to high-performance communication interfaces in the context of *homogeneous* clusters. Yet, cheap and powerful platforms for parallel computing can be obtained by interconnecting several clusters together. In general, the resulting configuration is highly *heterogeneous*. In opposite to the assumption for Grid-computing, the inter-cluster connections may be as powerful as the intra-cluster ones: environments designed for grids are not suitable for such clusters of clusters. We demonstrate in this paper that *Madeleine II* can be extended with a fast data-forwarding mechanism to handle such a heterogeneous configuration in a uniform way.

Section 2 presents the generic communication interface featured by *Madeleine II* and the explicit control over message construction it provides to the application. Then, we describe the internal structure of our library in Section 3 through an in-depth study of its highly modular organization. Section 4 displays this organization in action while transmitting a message. Section 5 reports on the performance of *Madeleine II*. It also demonstrates how *Madeleine II* can be used as a low-level communication layer for two famous communication libraries: MPICH [8] and Globus/Nexus [6]. Section 6 describes some prelim-

*LIP, ENS-Lyon, 46, Allée d'Italie, F-69364 Lyon Cedex 07, France. Contact: {Olivier.Aumage@ens-lyon.fr}

mad_begin_packing	Initiates a new message
mad_begin_unpacking	Initiates a message reception
mad_end_packing	Finalize an emission
mad_end_unpacking	Finalize a reception
mad_pack	Packs a data block
mad_unpack	Unpacks a data block

Table 1. Functional interface of Madeleine II.

inary work to extend *Madeleine II* with an efficient inter-device data-forwarding facility.

2 An Interface to Multiprotocol Communication

2.1 Basic Concepts

Madeleine II aims at enabling an efficient use of the complete set of underlying communication software and hardware available on a given cluster. It is able to deal with several networks (through possibly different interfaces) within the same session and to manage multiple network adapters (NIC) for each of these networks. The library provides an explicit control over communication on each underlying network. The user application can dynamically switch from one network to another, according to its communication needs.

This control is offered by means of two basic objects. The *channel* object defines a closed world for communication. Communication over a given channel does not interfere with communication over another channel. A channel is associated with a network interface, a corresponding network adapter and a set of *connection* objects. Each connection object virtualizes a point-to-point reliable network connection between two processes belonging to the session. It is of course possible to have several channels related to the same interface and/or the same network adapter. This feature may be used to logically split communication from two different modules. Yet, in-order delivery is only enforced for point-to-point connections within the same channel.

2.2 Message Construction

The *Madeleine II* programming interface is essentially the same as the *Madeleine* interface but a few minor modifications and improvements. Like *Madeleine*, it provides a small set of primitives to build RPC-like communication schemes. These primitives actually look like classical message-passing-oriented primitives. Essentially, this interface provides primitives to send and receive *messages*, and several *packing* and *unpacking* primitives that allow the user to specify how data should be inserted into/extracted from messages (Table 1). Just like Fast-Messages [12] or Nexus [6], *Madeleine II* allows applications to incrementally build messages to be transmitted, possibly at multiple software levels. To illustrate this, let us consider a remote procedure call which takes an array of unpredictable size as a parameter. When the request reaches the destination node, the header must be examined both by the multithreaded runtime (to extract the name of the function that will be executed by the server thread) and by the user application (to allocate the memory where the array should be stored).

A *Madeleine II* message consists of several pieces of data, located anywhere in user-space. It is initiated with a call to `mad_begin_packing`. Its parameters are the remote node *id* and the channel object to use for the message transmission. Each data block is then appended to the message using `mad_pack`. The message construction is eventually finalized by calling `mad_end_packing`. This last operation ensures that each previously packed piece of the message has actually been flushed to the network.

In addition to the data address and size, the packing primitive features a pair of *flag* parameters which specify the semantics of the operation. This is an original specificity of *Madeleine II* with respect to other communication libraries. For example, it is possible to require *Madeleine II* to enforce a piece of data to be immediately available on the receiving side after the corresponding `mad_unpack` call. Alternatively, one may completely relax this constraint to allow *Madeleine II* to optimize data transmission according to the underlying network as explained below. The expression of such constraints by the application is the key point to provide an optimal level of performance through a generic interface. The available emission flags are the following:

`send_SAFER` This flag indicates that *Madeleine II* should pack the data in a way that further modifications to the corresponding memory area should not corrupt the message. This is particularly mandatory if the data location is reused before the message is actually sent.

`send_LATER` This flag indicates that *Madeleine II* should not consider accessing the value of the corresponding data until the `mad_end_packing` primitive is called. This means that any modification of these data between their packing and their sending shall actually update the message contents.

`send_CHEAPER` This is the default flag. It allows *Madeleine II* to do its best to handle the data as efficiently as possible. The counterpart is that no assumption should be made about the way *Madeleine II* will access the data. Thus, the corresponding data should be left unchanged until the send operation has completed. Note that most data transmissions involved in parallel applications can accommodate the `send_CHEAPER` semantics.

The following flags control the reception of user data packets:

`receive_EXPRESS` This flag forces *Madeleine II* to guarantee that the corresponding data are immediately available after the *unpacking* operation. Typically, this flag is mandatory if the data is needed to issue the upcoming *unpacking* calls. On some network protocols, this functionality may be available for free. On some others, it may result in poor performance. The user should therefore extract data this way only when necessary.

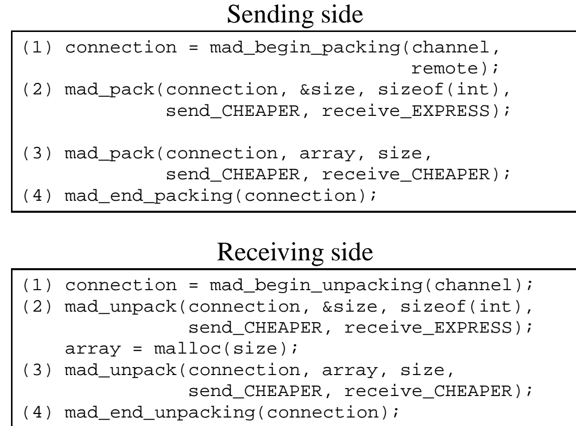
`receive_CHEAPER` This flag allows *Madeleine II* to possibly defer the extraction of the corresponding data until the execution of `mad_end_unpacking`. Thus, no assumption can be made about the exact moment at which the data will be extracted. Depending on the underlying network protocol, *Madeleine II* will do its best to minimize the overall message transmission time. If combined with `send_CHEAPER`, this flag guarantees that the corresponding data is transmitted as efficiently as possible.

It should be stressed that this message construction is in fact virtual. *Madeleine II* may well choose at any pack step to send data over the network or to keep data in place and delay transmission or even to copy data into protocol-specific preallocated buffers. There is no restriction about the combinations of the send and receive modes in the current implementation.

However, *Madeleine II* messages do not contain any information about which mode was selected for each piece of data, for the sake of optimal latency. Hence, one should ensure that packing and unpacking sequences are strictly symmetrical (regarding both packet sizes and combinations of packing/unpacking modes). Unspecified behavior would occur otherwise.

2.3 Example

Figure 1 illustrates the power of the *Madeleine II* interface. Consider sending a message made of an array of bytes whose size is unpredictable on the receiving side. Thus, the receiver has first to extract the size of the array (an integer) before extracting the array itself, because the destination memory has to be dynamically allocated. In this example, the constraint is that the integer must be extracted `EXPRESS` *before* the corresponding array data is extracted. In contrast, the array data may safely be extracted `CHEAPER`, striving to avoid any copies. It is fine to do so, as the size of the array is expected to be much larger than the size of an integer. The `end_unpacking` call ensures that the array has actually been filled with the expected piece of data.



Sending side

```
(1) connection = mad_begin_packing(channel,
                                    remote);
(2) mad_pack(connection, &size, sizeof(int),
             send_CHEAPER, receive_EXPRESS);

(3) mad_pack(connection, array, size,
             send_CHEAPER, receive_CHEAPER);
(4) mad_end_packing(connection);
```

Receiving side

```
(1) connection = mad_begin_unpacking(channel);
(2) mad_unpack(connection, &size, sizeof(int),
               send_CHEAPER, receive_EXPRESS);
    array = malloc(size);
(3) mad_unpack(connection, array, size,
               send_CHEAPER, receive_CHEAPER);
(4) mad_end_unpacking(connection);
```

Figure 1. Sending and receiving messages with Madeleine II.

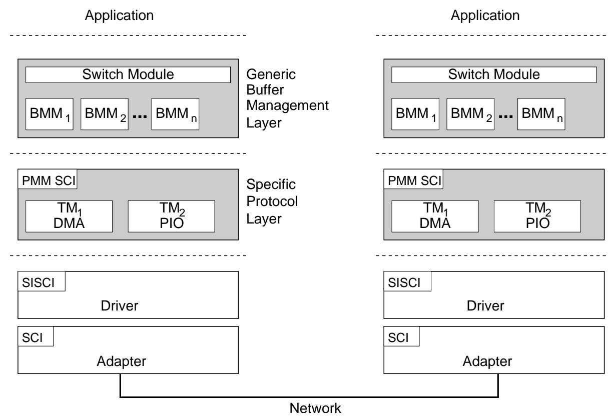


Figure 2. Madeleine II's modular architecture.

3 The Core Structure of Madeleine II

3.1 Global Organization

Nowadays communication libraries have to reach two seemingly contradictory goals. They are expected to provide both an effective portability over a wide range of hardware/software combinations, whilst achieving a high efficiency using these components. To meet these goals, *Madeleine II* follows a modular approach built around a highly flexible architecture. This approach allows the library to tightly fit and optimally exploit the specific characteristics of each target network.

Madeleine II is organized as two software layers (Fig. 2), following a commonly used scheme. *Protocol/network-specific interfacing* is realized by the lower layer, providing the portability of the whole library. This layer relies on a set of network specific *Transmission Modules* (TM). The upper layer is independent of the supported network interfaces and

send_buffer	Send a single buffer
send_buffer_group	Send a group of buffers
receive_buffer	Receive a single buffer
receive_sub_buffer_group	Receive a group of buffers
obtain_static_buffer	Obtain a protocol level buffer
release_static_buffer	Release a protocol level buffer

Table 2. Functional interface of TMs.

is in charge of the *generic buffer management*. It is made of several *Buffer Management Modules* (BMM), each of these implementing a given buffer management policy.

3.2 Transfer Management

One of the goals of *Madeleine II* is to support multi-modal interfaces such as VIA [4] or SISCI/SCI [7]. Such interfaces provide several data transfer methods. For instance, regular Processor IO (PIO) and Direct Memory Access (DMA) are available for Dolphin SCI NICs. Moreover, it should be able to easily take into account interface implementations like BIP/Myrinet [13] which make a difference between *short* buffers and *long* buffers. As a consequence, *Madeleine II* features specific modules to encapsulate each of these *sub*-interfaces. These modules are called *Transmission Modules* (TM).

Table 2 shows the common interface of the TMs (note that some functions may not be relevant for a specific TM and will not be implemented in such case). We can see that TMs provide single buffer transmission support and potentially optimized scatter/gather multi-buffer transfers. Depending on the underlying network properties, they may also implement protocol-specific buffer allocation routines. This feature is needed for protocols which provide their own set of preallocated buffers.

3.3 Network Management

TMs are grouped into *Protocol Management Modules* (PMM). There is one PMM for each supported network interface (e.g., BIP or TCP). Each PMM implements whole or part of a generic set of functions. This set of functions constitutes the protocol driving interface. It ensures independence between the upper layer and the communication networks. The protocol management modules are based on a hierarchy of data structures that virtualize each basic object involved during a data transfer: Driver, Adapter, etc.

3.4 Buffer Management

While some TMs can benefit from grouped buffer transfers, others may behave worse depending on the functionalities implemented by the underlying network. Each TM should thus be fed with its optimal shape of data. As a result, each TM is associated with a *Buffer Management Module* (BMM) from the Buffer Management Layer. Of course, it is expected that several TMs share the same shape so that BMMs can be reused, which results in a significant improvement in development time and reliability.

Each BMM implements a generic, protocol-independent management policy. A BMM may either control *dynamic buffers* (the user-allocated data block is directly referenced as a buffer) or *static buffers* (data is copied into a buffer provided by the TM), but not both. The static buffer BMMs work together with TMs implementing the `obtain_static_buffer` and `release_static_buffer` functions. These functions provide the BMMs with a generic access to pools of protocol specific buffers. The work of copying user pieces of data into and from static buffers is done by the BMMs.

Moreover, each BMM may implement a specific aggregation scheme to group successive buffers into a single virtual piece of message in order to exploit optional scatter/gather protocol capabilities. However, a BMM may also adopt an eager behavior and send buffers as soon as they are ready.

4 A Message Transmission Step-by-Step

We now show the *Madeleine II* components in operation while transmitting an application message.

4.1 Sending

The application initiates the construction of an outgoing message through a call to `begin_packing(channel, remote)`. The `channel` object selects the protocol module, and the adapter to use for sending the message. The `remote` parameter specifies the destination node. The `begin_packing` function returns a `connection` object.

Using this `connection` object, the application can start packing user data into packets by calling `pack(connection, ptr, len, s_mode, r_mode)`. Entering the Generic Buffer Management Layer, the packet is examined by the *Switch Module* (Step 1 on Fig. 3). It queries the Specific Protocol Layer (Step 2) for the best suited *Transmission Module*, given the length and the send/receive mode combination. The selected TM (Step 3) determines the optimal *Buffer Management Module* to use (Step 4). Finally, the Switch Module forwards the packet to the selected BMM. Depending on the BMM, the packet may be handled as is (and considered as a buffer), or copied into a new buffer, possibly provided by the TM. Depending on its aggregation scheme, the BMM either immediately sends the buffer to the TM or delays this operation for a later time. The buffer is eventually sent to the TM (Step 5). The TM immediately processes

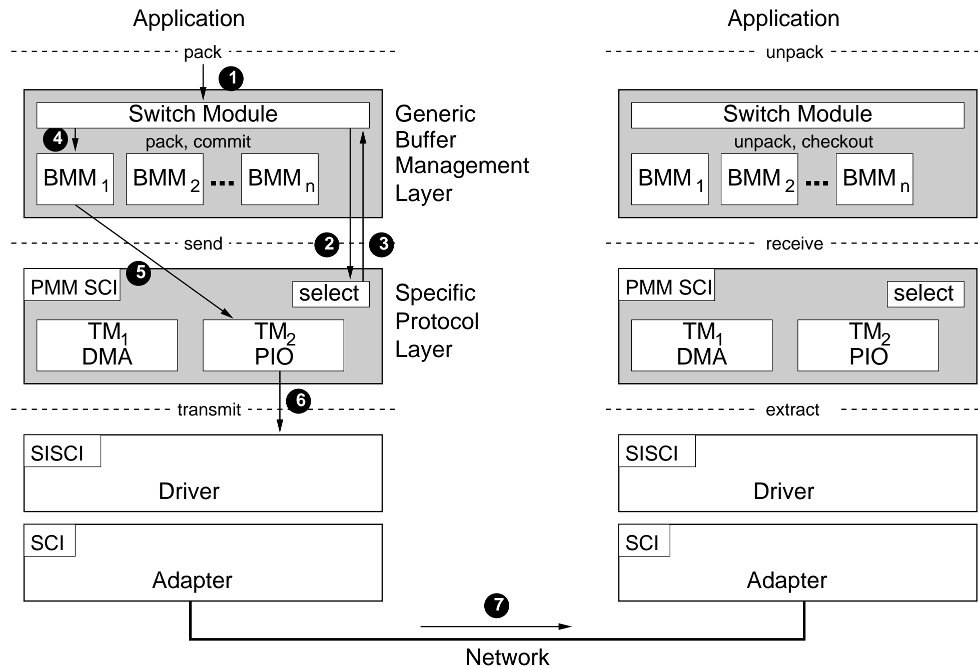


Figure 3. Conceptual view of the data path through Madeleine II's internal modules.

and transmits it to the network driver/library (Step 6). The buffer is then eventually shipped to the network adapter (Step 7).

Special attention must be paid to guarantee the delivery order in presence of multiple TMs. Each time the Switch Step selects a TM differing from the previous one, the corresponding previous BMM is flushed (*commit* on Fig. 3) to ensure that any remaining delayed packet has been shipped to the network. A general *commit* operation is also performed by the `end_packing(connection)` call to ensure that no delayed packet remains waiting in the BMM.

4.2 Receiving

Processing an incoming message on the destination side is just symmetric. A message reception is initiated by a call to `begin_unpacking(channel)` which starts the extraction of the first incoming message for the specified channel. This function returns the `connection` object corresponding to the established point-to-point connection, which contains the remote node identification among other things.

Using this `connection` object, the application issues a sequence of `unpack(connection, ptr, len, s_mode, r_mode)` calls, symmetrically to the series of `pack` calls that generated the message. Exact symmetry between `pack` and `unpack` call series is mandatory because *Madeleine II* messages are not self-described (in order to optimize efficiency). The Switch Step is performed on each `unpack` and must select the same sequence of TM as on the sending side. For instance, a packet sent by the DMA Transmission Module of SCI must be received by the same module on the receiving side. The *checkout* function (dual to the *commit* one on the sending side) is used to actually extract data from the network to the user application space: indeed, just like packet sending could be delayed on the sending side for aggregation, the actual packet extraction from the network may also be delayed to allow for burst data reception. Of course, the final call to `end_unpacking(connection)` ensures that all expected packets are made available to the user application.

5 Implementation and Performance

5.1 Testing Environment

The following performance results are obtained using a cluster of dual Intel Pentium II 450 MHz PC nodes with 128 MB of RAM running Linux version 2.2.13. The cluster interconnection networks are Dolphin SCI (D310 NICs) for SISCI and Myrinet (NICs specs: LANai 4.3, 32-bit bus, 1 MB SRAM) for BIP. Please note that the latency measurements are one-way transfer time measurements.

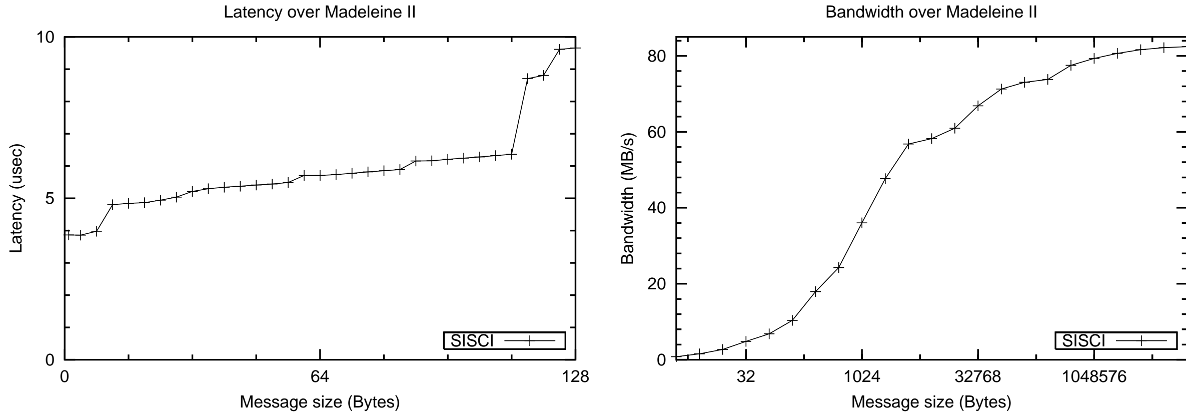


Figure 4. Latency and bandwidth over SISCI/SCI

5.2 Madeleine II on Top of Existing Interfaces

5.2.1 SISCI/SCI

Results The performance measurements of the SISCI PMM are shown in Figure 4. We can see that the minimal latency is very low (3.9 μs), thanks to our highly optimized short message TM (see implementation details below).

The bandwidth is very good, too, thanks to the use of an adaptive, dual-buffering algorithm. This algorithm is activated for data blocks larger than 8 kB within the regular SISCI TM as clearly seen on Figure 4. This optimization allows *Madeleine II* to deliver a bandwidth of 82 MB/s.

The SISCI PMM handles both transmission modes provided by the SISCI interface: a regular PIO mode and a DMA mode. Three transmission modules are currently implemented, as the regular PIO mode uses an additional TM specifically optimized for short message transfer. Note that the DMA mode TM is implemented but not active in the current version, because of the poor performance of the SCI DMA: we have not been able to get more than 35 MB/s with Dolphin SCI D310 NICs.

5.2.2 BIP/Myrinet

BIP (Basic Interface for Parallelism) is a low-level communication interface specifically designed for the Myrinet network protocol [13]. The main advantage of BIP is to provide communication control in user space: the application may directly interact with the network interface card. The BIP interface makes a distinction between short messages (< 1 kB) and long messages. Short messages are temporarily stored into internal buffers (preallocated by BIP) on the receiving side. No participation of the receiver is necessary. In contrast, long messages are directly delivered at their final location without any intermediate copy. In this latter case, a strict synchronization is necessary between the sender and the receiver: the receiver must acknowledge the sender that it is ready to receive before a message is actually transmitted.

The BIP PMM of *Madeleine II* handles both transmission modes. The *short message* TM uses a credit-based flow control algorithm to make sure that each message can be stored into a buffer. The *long message* TM implements the receiver-acknowledgment synchronization scheme. This *Madeleine II* BIP PMM achieves top performance results with a minimal latency of 7 μs and a bandwidth of 122 MB/s (Figure 5). Theses results are very close to the raw BIP results: 5 μs minimal latency and 126 MB/s maximal bandwidth.

5.3 Madeleine II as a Basis for High-Level Communication Libraries

While *Madeleine II* has also been ported (quite straightforwardly) on top of MPI, it may also be used as a low-level multiprotocol communication component for MPI implementations (as well as other for communication interfaces) too. We now present two implementations of high-level communication libraries–namely MPICH [8, 9] and Globus/Nexus [6, 3]–over *Madeleine II*.

5.3.1 MPICH/Madeleine II

Madeleine II has been integrated into MPICH as a new *ch-mad* module. Our goal was to let MPICH benefit from the multi-protocol features of *Madeleine II*. Preliminary performance measurements are quite encouraging. Figure 6 compares MPICH/*Madeleine II*/SISCI to two other implementations of MPI over SCI, namely SCI-MPICH [16] and the commercial version ScaMPI [15]. The performance curves of *Madeleine II* over SISCI (without MPICH) are plotted

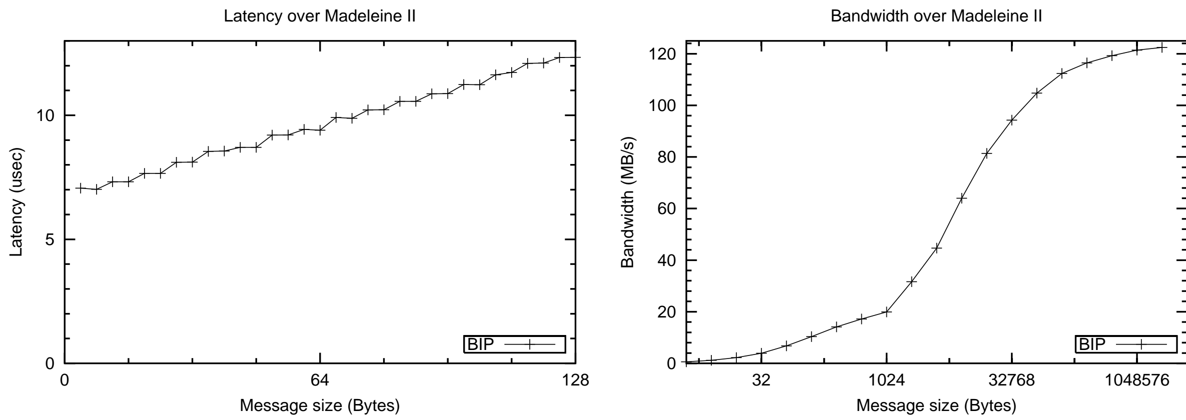


Figure 5. Latency and bandwidth over BIP/Myrinet

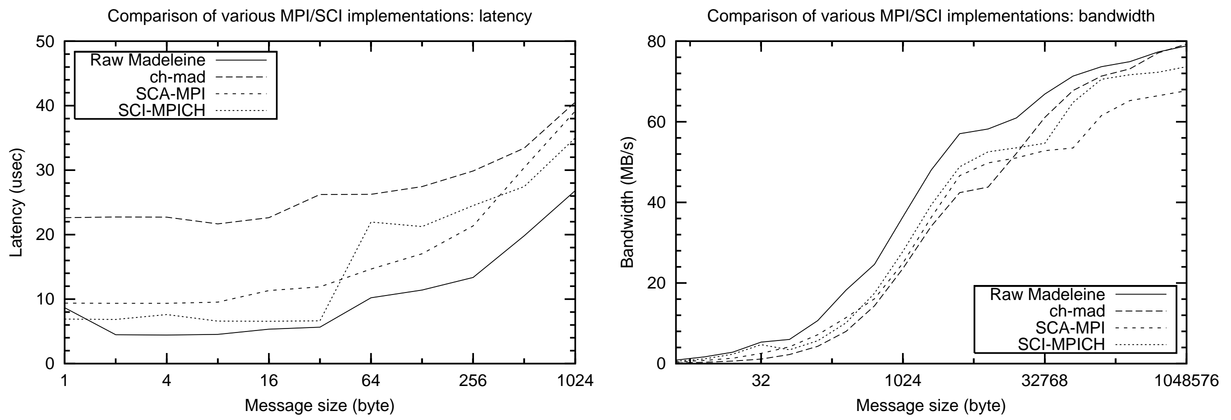


Figure 6. Comparison of various MPI implementations over SCI

too, in order to provide an idea of the current overhead of our MPI/*Madeleine II* implementation.

Though latency does not compare favorably to direct implementations of MPI over SCI, we can see that things are much different as far as bandwidth is concerned. Our *ch-mad* module provides the best results for messages of 32 kB and above. Moreover, this module is able to use most of the bandwidth provided by *Madeleine II* for large messages.

5.3.2 Nexus/Madeleine II

While Nexus is valuable–as part of Globus–for interconnecting supercomputers and clusters of workstations with wide area networks (WAN), it suffers from its heavy mechanisms when it comes to perform high performance application communication at the cluster scale. In contrast, *Madeleine II* was specifically designed to provide applications with highly efficient access to cluster network resources. Hence, it was interesting to investigate merging these two communication libraries in order to get the best of both worlds. The problem is the different models adopted by these communication interfaces: Nexus initialization scheme is point-to-point connection-oriented while *Madeleine II* is cluster-oriented.

Figure 7 shows the level of performance achieved by our implementation of Nexus over *Madeleine II*/TCP and *Madeleine II*/SISCI. It is clear that even with a rather heavy interface and without any sophisticated optimization, our Nexus/*Madeleine II* implementation is very effective on a high-performance network like SCI (with a minimal latency below 25 μs) and offers a more interesting solution as far as cluster computing is concerned.

Nexus features multiprotocol support [5] and *Madeleine II* is currently seen as one protocol by Nexus. Hence, we can easily imagine Globus applications using regular the TCP/Nexus protocol for wide area transmission and the "*Madeleine II*" Nexus protocol for local cluster high-performance computation.

6 Efficient Inter-Device Data-Forwarding in *Madeleine II*

The success of cluster computing in both academic institutions and companies led to consider interconnecting several clusters to form powerful heterogeneous infrastructures for parallel computing. However, developing runtime systems for such architectures raises many research issues. Among them, the design of the communication subsystem is perhaps the most challenging. Because network links between clusters may be as fast as internal cluster links, clusters of clusters significantly differ from grid architectures where inter-cluster links are assumed to be slow Consequently, communication environments originally designed

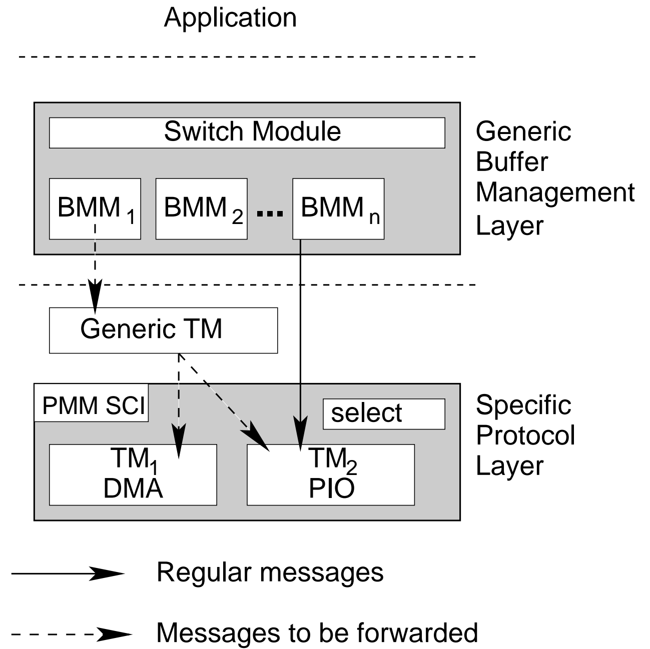


Figure 8. Integrating the Generic Transmission Module into *Madeleine II*: Emitting a message.

for grids, such as Nexus [6], cannot be efficiently used in this new context.

It has been proposed (e.g., PACX-MPI [11]) to *glue together* heterogeneous communication libraries. In contrast, we propose to extend the natively multi-device communication library *Madeleine II* with an additional facility to efficiently transfer messages across devices. Doing so, the inter-device data-transfer mechanism is completely hidden to the upper layers and the low-level characteristics of network devices can be used to optimize transfers (pre-allocated buffers, DMA operations, etc.) Then, higher-level traditional routing mechanisms can be efficiently implemented on top of this extended *Madeleine II* interface. Our approach retains *Madeleine II*'s portability while being as efficient as possible with regards to the capabilities of high-speed networks. The only change in the interface is due to the necessity to provide additional information about the configuration of the network when a channel is created: instead of a single channel using a given network protocol, one has to specify a *virtual channel* that includes a sequence of real channels.

6.1 Implementation Principles

Forwarding mechanism Where should the forwarding mechanism be implemented within the layered architecture of *Madeleine II*? Because Transmission Modules (TMs) are

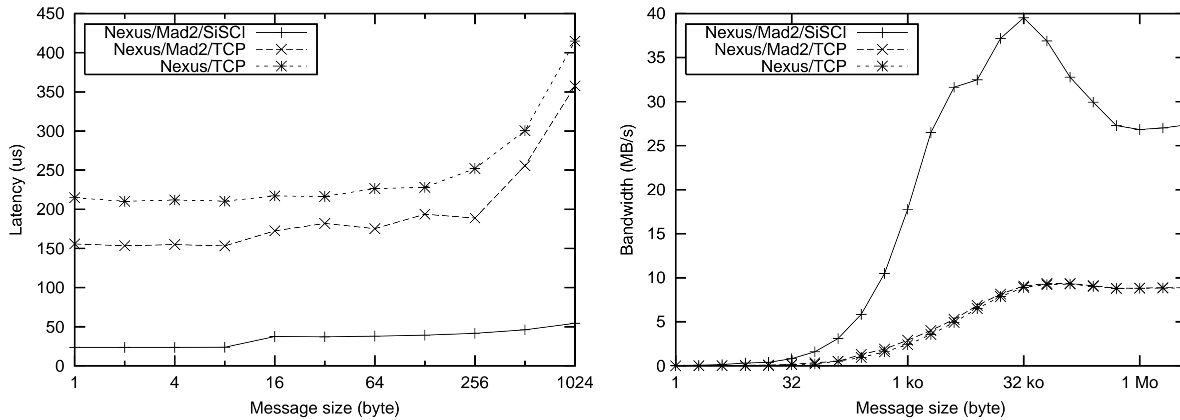


Figure 7. Nexus/Madeleine II performance

strongly protocol-dependent, it is not possible to implement the forwarding mechanism at this level without compromising *Madeleine II*'s portability. Alternatively, modifying the Buffer Management Layer could be considered, but a prohibitive development cost would be expected because of the number of modules in that layer. Furthermore, additional *conversion* modules between Buffer Management Modules (BMMs) would have to be introduced because the BMMs used on each side of a virtual channel may differ. Finally, we could merely implement this mechanism on top of *Madeleine II*, which would perfectly meet our needs in terms of portability. However, this would also have a dramatic impact on the efficiency because all data transfers would have to through *Madeleine II* twice on the gateway nodes, with the need for extra copies in temporary buffers. Hence, the best solution seems to insert the forwarding mechanism between BMMs and TMs.

Data transfers How can actual data transfers be performed? The most efficient way would be to consider raw transfers between transmission modules. However, such a raw forwarding is impossible because *Madeleine II* may use different BMMs for different network devices in order to optimally exploit the characteristics of the various underlying networks. Buffers may thus be grouped in a specific way for each single device, and they should be ungrouped and then regrouped in a different way at each gateway. To circumvent this difficult problem, we have decided that all inter-cluster traffic should be handled by a *generic* TM. This TM, used by both the sender and the receiver of a message as an interface between BMMs and real TMs (see Fig. 8), guarantees that data is handled in the same way on both ends. Some optimizations are lost, but the cost of ungrouping and regrouping buffers is definitely saved.

Self-described messages Within homogeneous *Madeleine II* applications, messages need not be self-described, since the user provides the necessary information on receiving a message. Yet, this information is not available to the gateway, unless the code of that gateway is written as part of the application, which is precisely what we try to avoid. Hence, self-describing messages are mandatory to our transparent forwarding mechanism. The generic TM is in charge of transparently inserting information needed by the gateway to get the size and actual destination of a message.

Generic Transmission Modules The Generic Transmission Module (Generic TM) guarantees that intermediate gateways can merely forward the buffers without considering the initial grouping method.

The Generic TM is also in charge of determining the *common*, optimal packet size (MTU, Maximum Transmission Unit) to be used along the route. To optimally use the pipeline mechanism on the gateways, the messages have to be fragmented into several packets. The size of those fragments is defined so that each network is able to send them without having to fragment them further. In the current version, the appropriate packet size is specified at compile time because the network configuration is statically configured.

The Generic TM is also used to add self-description information to the messages that go through the gateways. This description is compulsory as gateways know nothing about what messages are to be expected. Since some information (e.g., the destination of the whole message) is common to several buffers, it is sent only once, as part of the first packet. Buffer-specific information, such as the size and the emission/reception constraints, is sent together with each buffer.

Avoiding copies on forwarding Efficiently using high-performance communication interfaces demand avoiding copies. This is easily done for interfaces based on *dynamic buffers*. Yet, some interfaces (e.g., SBP [14]) require data to be written in specific buffers before being sent. In that case, using an additional temporary buffer to receive data should definitely be avoided. *Madeleine II*'s support for static buffer protocols allows us to avoid any extra copy as follows. If the sending-side interface uses static buffers while the receiving-side one uses dynamic buffers, it suffices to request a static buffer from the outgoing protocol/network TM: it can be used to receive data, thereby saving one copy. Obviously, one extra copy cannot be avoided when *both* networks require static buffers.

6.2 Experimental Evaluation

We report below on a series of tests run on a two-cluster configuration: a Myrinet cluster and a SCI one, connected through a shared node with both interfaces. Both clusters are built with dual Intel Pentium II 450 MHz PC nodes, equipped with 128 MB of RAM and with a 33 MHz, 32-bit PCI bus. The operating system is Linux version 2.2.13. The Myrinet cluster uses LANai 4.3 NICs, with a 32-bit bus and 1 MB SRAM. The SCI cluster uses Dolphin SCI D310 NICs. The communication interfaces are BIP [13] for Myrinet and the Dolphin SISCI for SCI.

We run inter-cluster ping tests between a regular node (i.e., not the gateway node) of one cluster and a regular node of the other, through the common gateway, first from SCI to Myrinet, and then from Myrinet to SCI. For each case, the ping program repeatedly transmits messages of the given size using the heterogeneous *Madeleine II* in one direction through the gateway. At each message, a small acknowledgment is sent back using the common Fast-Ethernet connection. Since we exactly know the latency of the acknowledgment, we are able to infer the one-way message transmission time from the total round-trip time.

6.2.1 Packet-forwarding pipeline architecture

Our forwarding mechanism is essentially designed to provide a high bandwidth on forwarding messages among clusters interconnected by high speed networks. However, low latency should not be expected from this design: 1) The overall latency of a inter-cluster transmission includes at least the native latencies of each networks; 2) It also includes a significant amount of software overhead at the gateway. Actually, the size of packets handled by the gateway is fixed by design in the current implementation, so that no optimization of the pipeline startup latency is possible at this time. In the sequel, we only discuss bandwidth performance.

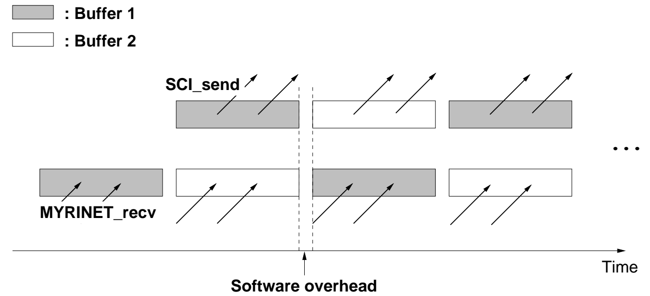


Figure 9. Packet-forwarding pipeline on the gateway node.

On the gateway node, our implementation uses two separate threads to pipeline the packet forwarding from one network to the other with a *dual-buffering* strategy. The best performance is achieved if: 1) Sending and receiving packets approximately take the same time (Figure 9); 2) The software overhead of having the threads exchange their buffers is small. Then, one buffer can be sent while the other is received with a perfect overlap.

However, the gateway node bridges two different networks and the respective transmission times for a given packet size differ in general. For instance, SCI achieves very good performance for small messages, whereas Myrinet behaves better for large messages. In fact, *Madeleine II* achieves approximately the same performance on top of Myrinet and SCI for messages of size 16 kB (latency: ca. 250 μs, bandwidth: ca. 60 MB/s), which suggests that the correct packet size should be set to 16 kB. Unfortunately, we will see in the next sections that several other factors have significant impact on the pipeline behavior, making it quite difficult to predict its actual performance.

6.2.2 Forwarding from SCI to Myrinet

We first report on the performance of our forwarding mechanism in the *SCI-to-Myrinet* direction (Figure 10) with packet sizes ranging from 8 kB to 128 kB. The bandwidth obtained when using 8 kB packets is only 36.5 MB/s. For larger packets, the bandwidth is greater than 45 MB/s and even close to 50 MB/s for 128 kB packets. This can be considered as satisfactory, since the theoretical maximum bandwidth one can achieve on a machine equipped with a single 33 MHz PCI bus is 66 MB/s.

For 8 kB packets, a pure *Madeleine II* ping-pong program achieves a bandwidth of 58 MB/s over SISCI/SCI and 47 MB/s over BIP/Myrinet. Thus, the period of the pipeline (i.e., the duration of a single step) for 8 kB packets is at least 166 μs. In practice, the observed bandwidth is 36.5 MB, which means that the effective pipeline period is closer to 215 μs. This seems to indicate that the software overhead

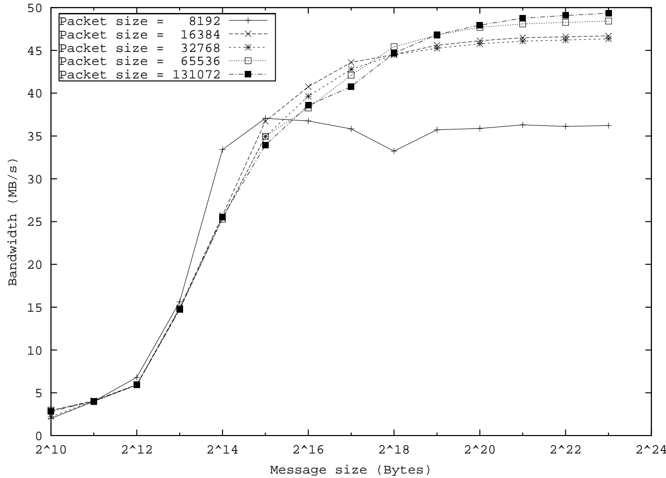


Figure 10. Forwarding bandwidth: from SISCI/SCI to BIP/Myrinet.

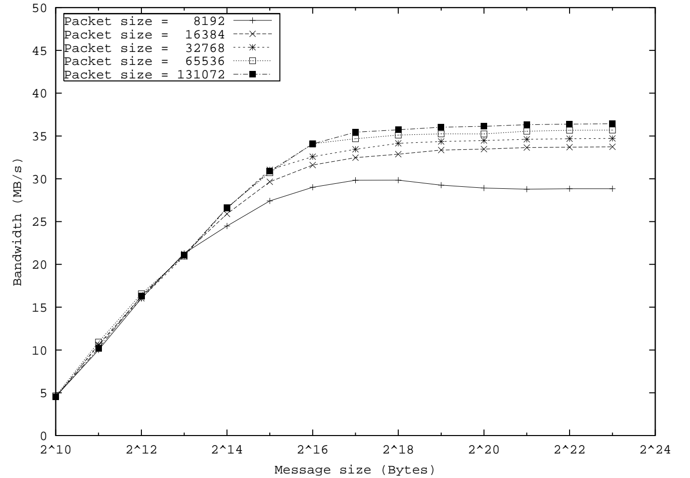


Figure 11. Forwarding bandwidth: from BIP/Myrinet to SISCI/SCI.

of this approach is 50 μs per step. We are unfortunately not able to analyze this behavior in greater detail as of today.

For larger packets however, another phenomenon appears. Indeed, for packet sizes greater that 16 kB, a pure *Madeleine II* ping-pong program achieves a bandwidth of more than 60 MB/s, which can be considered as the maximum one-way bandwidth one can get over a 32-bit PCI bus in practice. Thus, the outgoing packets cannot be sent with a bandwidth greater than 60 MB/s. However, we only reach an asymptotic bandwidth of 49.5 MB/s in practice: we assume that this is due to some conflicts raised on the PCI bus when doing intensive full-duplex communications.

6.2.3 Forwarding from Myrinet to SCI

We run similar tests in the opposite direction for packet sizes ranging from 8 kB to 128 kB (Figure 11). The observed performance is by far lower than the previous one. The bandwidth obtained when using 8 kB packets is only 29 MB/s (instead of 36.5 MB/s), and the asymptotic bandwidth obtained for larger packets remains under 36.5 MB/s (instead of 50 MB/s)!

Obviously, such disappointing results cannot be due to some software overhead nor to the saturation of the PCI bus. In fact, we run several additional experiments that seems to indicate that the problem is related to the priority of the involved PCI transactions: the DMA PCI transactions initiated by the Myrinet card seem to have a greater priority than the PIO PCI transactions initiated by *Madeleine II*. Consequently, during a (Myrinet-) buffer receiving, the sending of the other buffer over SCI is slowed down by a factor of two. We are currently investigating several techniques to reduce this phenomenon.

7 Conclusion

Madeleine II is a new high-performance communication library for distributed programming environments. Our library features full multi-protocol, multi-adapter support as well as an integrated new dynamic *most-efficient transfer-method* selection mechanism. It currently runs on top of BIP, SISCI, TCP, VIA and common MPI implementations. We reported very interesting performance results on top of BIP/Myrinet and SISCI over a SCI network.

In addition, we have shown the effectiveness of *Madeleine II* as a foundation for higher level communication libraries and introduced two implementations: Nexus/*Madeleine II* and MPICH/*Madeleine II*. Here again, results are highly encouraging. MPICH/*Madeleine II* even outperforms the current best implementations of MPI over SCI as far as bandwidth is concerned.

Madeleine II can also be extended with a portable data-forwarding mechanism for network-heterogeneous clusters of clusters. We showed that targeting the right abstraction level can make this additional mechanism completely transparent from the application point of view, portable on a wide range of network protocols, while remaining efficient. Zero-copy techniques together with pipelining strategies are mandatory to keep a high bandwidth over inter-cluster links. However, the sharing of the gateway internal system bus bandwidth appears to be a central issue: some sophisticated *bandwidth control* mechanism is needed to regulate the incoming communication flow on gateways. This is a point we intend to investigate in the future.

We are now actively investigating the integration of *Madeleine II* with our user-level multithreading library

Marcel by the design and development of advanced adaptive polling/interruption network interaction mechanisms coupled to an extensive support of our implementation of the *Scheduler Activations* [2]

References

[1] L. Boug, J.-F. Mhaut, and R. Namyst. Efficient communications in multithreaded runtime systems. In *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 468–482, San Juan, Puerto Rico, Apr. 1999. Springer-Verlag.

[2] V. Danjean, R. Namyst, and R. Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, volume 1800 of *Lect. Notes in Comp. Science*, pages 1160–1167, Cancun, Mexico, May 2000. Springer-Verlag.

[3] A. Denis. Adaptation de l'environnement générique de metacomputing Globus à des réseaux haut débit. Master's thesis report, DEA d'informatique fondamentale, ENS-Lyon, France, June 2000.

[4] 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*, pages 66–75, Mar. 1998.

[5] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

[6] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal on Parallel and Distributed Computing*, 37(1):70–82, 1996.

[7] H. Hellwagner and A. Reinefeld, editors. *SCI: Scalable Coherent Interface, Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *Lect. Notes in Comp. Science, State-of-the-Art Surveys*. Springer-Verlag, 1999.

[8] E. Lusk and W. Gropp. MPICH working note: The second-generation ADI for the MPICH implementation of MPI. Technical report, Argonne National Laboratory, 1996.

[9] G. Mercier. Support efficace de l'hétérogénéité des réseaux dans MPI. Master's thesis report, DEA d'informatique fondamentale, ENS-Lyon, France, June 2000.

[10] R. Namyst and J.-F. Méhaut. PM2: Parallel Multithreaded Machine; a computing environment for distributed architectures. In *Parallel Computing (ParCo'95)*, pages 279–285. Elsevier, Sept. 1995.

[11] PACX-MPI. `http://www.hlrs.de/structure/organisation/par/projects/pacx-mpi/`.

[12] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, Apr. 1997.

[13] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*, volume 1388 of *Lect. Notes in Comp. Science*, pages 472–485. Springer-Verlag, Apr. 1998.

[14] R. Russell and P. Hatcher. Efficient kernel support for reliable communication. In *13th ACM Symposium on Applied Computing*, pages 541–550, Atlanta, GA, Feb. 1998.

[15] Sca-MPI. `http://www.scali.com/`.

[16] J. Worringen and T. Bemmerl. MPICH for SCI-connected clusters. In *SCI Europe '99*, pages 3–11, Bordeaux, France, Sept. 1999.