Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

SPI

MPICH/Madeleine: a True Multi-Protocol MPI for High Performance Networks

Olivier Aumage
Guillaume Mercier
Raymond Namyst

October 2000

Research Report N° RR2000-30

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

INRIA

MPICH/Madeleine: a True Multi-Protocol MPI for High Performance Networks

Olivier Aumage
Guillaume Mercier
Raymond Namyst

October 2000

Abstract

This report introduces a version of MPICH handling efficiently different networks simultaneously. The core of the implementation relies on a device called ch_mad which is based on a generic multiprotocol communication library called *Madeleine*. The performance achieved with tested networks such as Fast-Ethernet, Scalable Coherent Interface or Myrinet is very good. Indeed, this multi-protocol version of MPICH generally outperforms other free or commercial implementations of MPI.

Keywords: MPI,High-Performance Network,Heterogeneity,Multi-Protocol,Cluster Computing.

Résumé

Ce rapport présente une version de MPI basée sur MPICH utilisant efficacement plusieurs réseaux simultanément. Le cœur de la mise en œuvre repose sur un module appelé ch_mad, lui-même basé sur une bibliotheèque de communication générique et multiprotocole : *Madeleine*. Les performances obtenues sur les réseaux testés (Fast-Ethernet, SCI, Myrinet) sont excellentes. Cette version multiprotocole de MPICH arrive à des niveaux de performances égaux voire supérieurs á ceux atteints par d'autres implémentations (libres ou commerciales) de MPI.

Mots-clés: MPI, réseaux haut-débit, hétérogénéité, multi-protocole, calculs sur grappes

1 Introduction

Nowadays, clustering technology is fully widespread among research laboratories thanks to the good tradeoff between cost and performance. Moreover, the diversity of gigabit/s networks (Myrinet, Giganet, SCI) leads to the building of heterogeneous clusters of clusters (i.e., a kind of *meta-cluster*). This obviously raises numerous new software needs. One major issue is that common communication libraries should be able to support network heterogeneity in an efficient manner.

The case of the Message Passing Interface library (MPI) is archetypal. The MPI standard [4] does not forbid nor fosters the support of multi-protocol features, but by examining most of the current MPI implementations (MPICH, LAM, etc.), we observe that no one offers such abilities. Nevertheless, several solutions exist. The most obvious approach is to get multi-protocol features through interoperability. With such class of solutions (PACX-MPI [8], MPICH-G [6]), interconnection of several different MPI libraries is privileged with one dedicated version of the library per cluster. MPI libraries specifically optimized for a particular network communicate with each other, thus allowing good performance for communication taking place within a cluster member of the *meta-cluster*.

In the MPICH implementation of MPI, the existence of an intermediate interface called the *Abstract Device Interface* (ADI) allows to plug different network support modules (aka *devices*) into the layered structure of MPICH. It is then theoretically possible to support network heterogeneity in MPICH, since the ADI data structures are to some extend multi-device-ready. In practice, however, taking advantage of such a support in MPICH turns out to be a hard issue. Indeed, a rather heavy integration work has to be done each time a new device is to be supported, in order to preserve inter-device coexistence. As a consequence, there is currently no MPICH version supporting network heterogeneity.

An alternate solution is to get a multi-protocol version of MPICH through the use of a generic multi-protocol communication library such as *Madeleine* [3], the communication subsystem of the *PM*^{2} environment. There are two key points in this approach: software re-usability (since *Madeleine* has not to be modified) as well as evolutivity (avoiding ADI modifications prevents from future incompatibilities due to MPICH changes). Nevertheless, one may wonder if such an approach could really be efficient. This report answers in the affirmative by reporting several results obtained on a number of high-performance networks. Comparisons with other MPI implementations prove that there is no significant loss of performance using our proposal (we even got improvements in some cases). This fact is made possible because *Madeleine*'s conception was multi-protocol-oriented from the very beginning.

The rest of this report is organized as follows: Section 2 briefly describes the MPICH structure and its limitations. Section 3 and 4 respectively present the *Madeleine* interface and show its integration as an MPICH device. Results are given in Section 5 and Section 6 concludes this report.

2 MPICH

MPICH (*MPI-CHameleon*), one of the most famous existing MPI implementations, was designed by W. Gropp and E. Lusk at the Argonne National Laboratory. The goal was to combine portability and high performance in a single implementation [5].

2.1 Overall Structure

The main challenge was obviously the design of an implementation architecture that could adapt to various hardware platforms while taking advantage of their specific features. These architectures include high performance switches (IBM SP2, Cray T3D), shared memory machines (SGI Onyx, Power Challenge) and clusters of workstations (Ethernet, Myrinet, SCI).

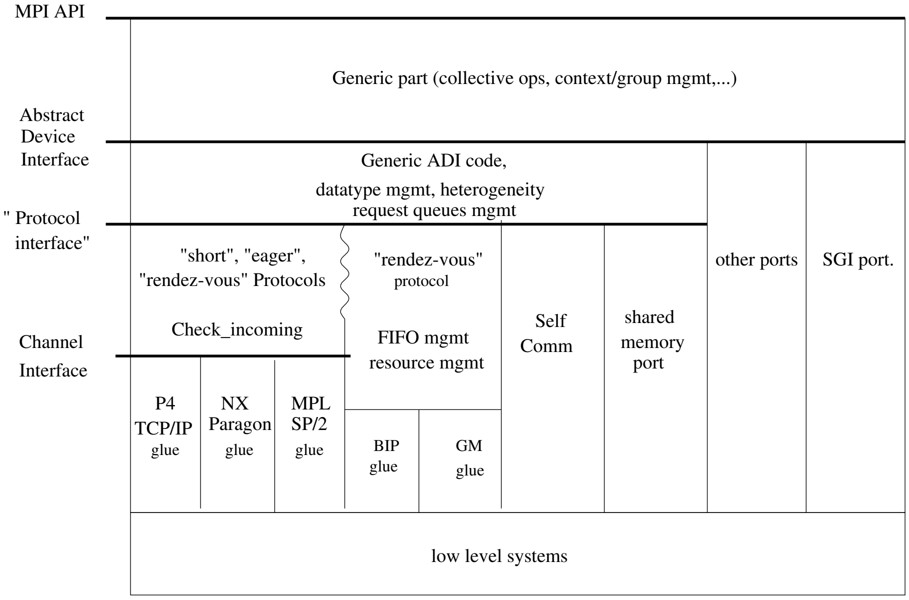


Figure 1: MPICH Structure

To this end, the internal structure of MPICH is organized as a number of software layers so as to maximize code-sharing across implementations (Figure 1). This is a classical approach, where upper layers usually implement high-level functionalities (such as collective operations, for instance) on top of simpler ones. Moreover, the architecture of MPICH makes it possible to build a new implementation by just rewriting the functions in the lowest level layer, and then to improve this implementation by gradually replacing parts of the shared code by platform-specific optimized code.

2.2 The Abstract Device Interface

The main component of this layered architecture is the Abstract Device Interface (ADI) which provides a portable message passing interface to the generic upper layer (see Figure 1). The ADI has thus to meet the heavy requirement of being portable efficiently on a wide range of network hardware ([11], [10]). That for, the ADI features a rich set of functions that may access the full power of many sophisticated networks but also provide software emulations of any functions that may not be supported by some devices.

More precisely, the ADI features a number of function definitions in terms of which the user-callable MPI functions may be expressed. These functions may be classified into four sets which are respectively intended to perform the message transmissions, to move the data between the generic API and the hardware, to process the queues of pending messages and to handle the queries about the configuration.

Obviously, the ADI may be implemented directly on top of the underlying network interface (as for the Cray T3D) to achieve maximal performance. Another alternative is to implement the ADI on a lowest layer that would yet be portable on several devices. The idea is to maximize portability (by potentially reducing the performance) by using a layer that would require only a few functions to be implemented for the device and by emulating all the necessary data transfer modes required by the ADI. This layer is called the "Channel Interface" in MPICH and is actually the quickest way to port MPICH on a new hardware.

Note that some implementations of MPICH are so optimized that the ADI is directly implemented over the underlying hardware protocol (e.g. the SGI port on Figure 1). Obviously, these implementations share only the "generic code" of MPICH and provide no opportunities for reusing code or for building multi-device implementations.

2.2.1 The Channel Interface

MPICH features a portable implementation of the ADI on an abstraction called the *channel interface*. The channel interface is a low-level communication interface that features only about five functions that are responsible for sending and receiving contiguous messages (carrying data or control information). On top of this simple interface, the MPICH portable ADI implements different data exchange protocols. Here are the ones which are mostly used:

Short The data is delivered together with the control information (message envelope).

Eager The data is delivered without waiting for the receiver to request it.

Rendez-vous The data is not delivered until the receiver requests it.

Each time the ADI is requested to send (or receive) a piece of data, one exchange protocol has to be dynamically selected. Such a choice depends on several parameters of the underlying hardware such as the time to send a control packet, the cost to perform a memory-to-memory copy within a process, etc. That for, protocol selection in MPICH is based on a set of device-specific parameters defined at initialization time. In particular, these parameters specify at which points (in terms of packet size) the ADI should switch from one protocol to the next one.

2.3 Getting MPICH Multi-Protocol

Our goal is to provide a fully multi-protocol implementation of MPI that would not only allow to run applications on top of heterogeneous architectures (such as clusters of clusters) but that would also allow applications to access all the communication facilities available between each pair of hosts. The MPICH implementation appears to be a good candidate since its architecture allows the building of implementations handling multiple devices simultaneously.

Indeed, most instantiations of MPICH use several devices since the loop-back facility is usually implemented as a particular additional device called `ch_self`. Furthermore, some multiprocessor versions of MPICH feature three devices: a regular device for inter-node communications (using TCP for instance), a loop-back device and an intra-node (inter-processor) device to allow processes on the same machine to communicate via shared memory (for instance `smp_plug`). When the ADI is requested to send a message, the appropriate device is selected and then the most suited exchange protocol is chosen.

However, to our knowledge, the selection of the appropriate device is straightforward in all current implementations of MPICH and only depends on the location of the destination process.

	TCP	BIP	SISCI
Latency	$121\mu s$	$9.2\mu s$	$4.4\mu s$
Bandwidth (8 MB Message)	11.2 MB/s	122 MB/s	82.6 MB/s

Table 1: Latency and bandwidth for Various Network Protocols

In particular, no available implementation allows an application to simultaneously use two different networks (e.g. Myrinet + SCI) within a single cluster. Moreover, building such a multi-device implementation would require several heavy modifications to the generic ADI code to realize functionalities such as transparent dynamic device selection.

For this reason, we did not subscribe to the classical MPICH philosophy of building a multi-device implementation. Instead, we have built a single-device implementation of MPICH on top of a true multi-protocol communication interface called *Madeleine*.

3 The Madeleine Multi-Protocol Communication Library

The *Madeleine* programming interface provides a small set of primitives to build RPC-like communication schemes. These primitives actually look like classical message-passing-oriented primitives. Basically, 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.

3.1 Overview

Madeleine aims at enabling an efficient (see Table 1) and exhaustive use of underlying communication software and hardware functionalities. It is able to deal with several network protocols within the same session and to manage multiple network adapters (NIC) for each of these protocols. The library provides an explicit control over communication on each underlying network protocol. The user application can dynamically switch from one protocol 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 (much like an MPI communicator). Communication over a given channel does not interfere with communication over another channel. A channel is associated with a network protocol, 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 protocol and/or the same network adapter, which 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.

3.2 Message Building: Principles and Example

Madeleine allows applications to incrementally build messages to be transmitted, possibly at multiple software levels. A *Madeleine* 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 last step uses `mad_end_packing` to finalize the message. 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* with respect to other communication libraries, e.g. FM [14] and Nexus [7]. For example, it is possible to require *Madeleine* 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* to optimize data transmission according to the underlying network. The expression of such constraints by the application is the key point to provide an optimal level of performance through a generic interface.

Figure 2 illustrates the power of the *Madeleine* 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.

Sending side

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

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

Receiving side

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

Figure 2: Sending and Receiving Messages with *Madeleine*.

3.3 Inter-Device Awareness

Madeleine has been targeted from the very beginning to be used as a communication support for distributed multi-threaded runtime systems involving remote service request (RSR), remote procedure call (RPC) or remote method invocation-like (RMI) interactions. As a consequence, its internals were designed to be both *thread-safe*, and to some extend *thread-aware*, in order to ensure a good level of reactivity (a very important parameter for RPC-like mechanisms).

Moreover, it was extended to take advantage of the advanced polling mechanisms provided by the user-level multi-threading library *Marcel* while both libraries cooperate to form the core part of the PM^2 environment [12]. Indeed, *Marcel* features the ability to regularly call *Madeleine*'s polling functions and to factorize several polling requests as a unique polling call, therefore limiting the polling overhead while maximizing the reactivity.

The polling frequency may also be selected on a per-protocol basis, enabling low latency networks with cheap polling mechanisms to be polled more frequently than TCP-like networks only providing the expensive `select` system call.

4 Integrating Madeleine as an Efficient MPICH Device

4.1 Structure Overview

Our solution follows the structure given by Figure 3. Three devices are concurrently used to handle communication. Basically, one specific device is dedicated to each type of communication taking place within a cluster of cluster:

- The `ch_self` device handles *intra-process* communication.
- The `smp_plug` device handles *intra-node* communication (case of SMP nodes in the configuration).
- The `ch_mad` device handles any *inter-node* communication.

The `ch_self` and `smp_plug` devices are parts of the SMP implementation of MPI-BIP ([9], [16]) and won't be further discussed in this report.

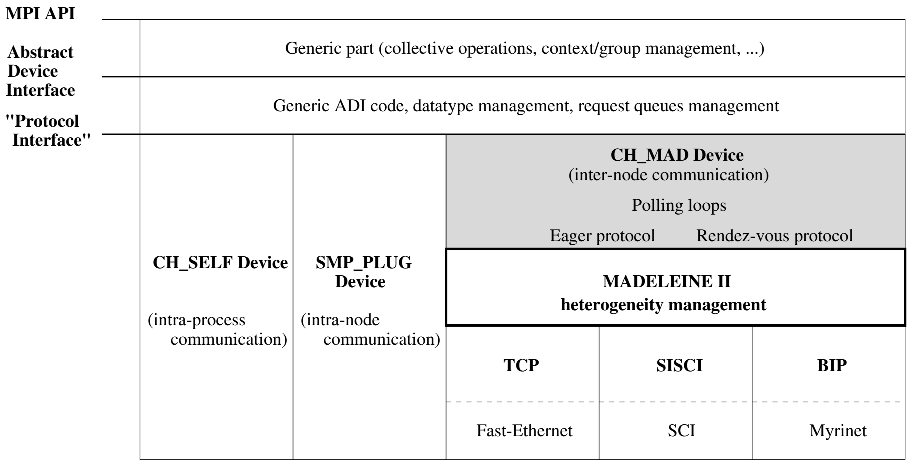


Figure 3: Software Global Organization

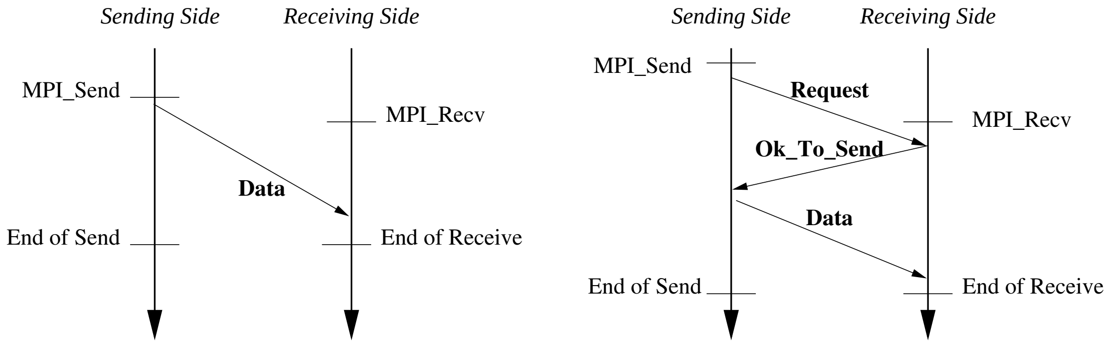
Network heterogeneity is hidden by the *Madeleine* software layer as shown by Figure 3. This heterogeneity is handled through the use of the *channels* objects of *Madeleine*, where one channel is mapped onto a single protocol. One may note that our proposed architecture significantly differs from other solutions in that no distinction is made between intra- and inter-cluster communication. On one hand, TCP sockets are no more dedicated to inter-cluster communication, while on the other hand the cluster-interconnecting network may now be used at full speed.

Our implementation uses *Marcel* user-level multi-threading library to improve both reactivity and efficiency. This library was preferred because of its excellent performance (creation, destruction and yield operations), its optimized interaction with the *Madeleine* communication library and for its polling features possible with *Madeleine* (cf. 3.3). We assign one thread per *Madeleine* channel and dedicate another one to application computation (the *main* thread). Since *Madeleine* communication primitives are blocking, *Marcel* threads are also employed to implement MPI non-blocking functions (e.g. `MPI_Isend`).

The `ch_mad` device provides two transfer modes: the *eager* mode and the *rendez-vous* mode. The mode selection is dynamically performed, according to the message size.

Eager mode: sends data directly from the sending to the receiving process. This mode is optimized for latency, at the cost of an intermediary copy on the receiving side. It is selected for short messages.

Rendez-vous mode: enforces a synchronization between the sending side and the receiving side before transmitting data. This mode involves two additional messages (the sender request and the receiver acknowledgement) but it delivers most of the underlying network bandwidth by performing zero-copy data transfers. It is used for long messages.



(a) *Eager* mode (b) *Rendez-vous* mode

Figure 4: Available Transfer Modes

Figure 4 shows the basic functioning of both transfer modes. The difference between a *short* and a *long* message strongly depends on the underlying protocol used to perform communication between processes. As a consequence, the switch point beyond which the rendez-vous transfer mode replaces the classical eager mode differs from protocol to protocol. This particular point will be discussed further.

4.2 Implementation Details

4.2.1 Channels Use and Packet Structure

During a MPICH-*Madeleine* session, each process accesses the multiple networks through their corresponding *Madeleine* channels. Consequently, it would be possible for a process to receive messages coming from several channels, to send messages to different processes over different channels, to send messages to one process over different channels.

However, this latter scheme is not yet supported by the current `ch_mad` device, as *Madeleine* does not provide in-order delivery guarantee across channels. Each MPI message is sent over a single channel and internally handled as unique *Madeleine* message. It is build up with one or two *Madeleine* packets: the number of packets has to be kept low to ensure a high level of performance, since each pack operation induces a significant overhead.

The ADI provides a rather convenient message structure. It is not completely reusable though, as polling loops executed by threads must group all receiving operations for both transfer modes.

Figure 5 depicts the classical header/body structure of the packets. The header is always sent following the *Madeleine* `EXPRESS` semantics (it contains data needed to unpack the body) while the body itself is sent using the `CHEAPER` semantics. A packet body is only sent as part of messages containing user/MPI data (`MAD_SHORT_PKT` and `MAD_RNDV_PKT` cases), the other messages do not have a body (thus avoiding unnecessary and expensive pack operations). The header is composed of a `type` field (an integer) followed by a buffer. The contents of this buffer depends on the message type:

`MAD_RNDV_PKT`: type of the rendez-vous mode data messages. The buffer contains a data structure (`sync_address`) whose type is implementor-definable in the ADI: `MPID_RNDV_T` (a convenient hook provided to implement rendez-vous transfer mode, see 4.2.2).

`MAD_SHORT_PKT`: type of the eager mode data messages. The buffer contains an ADI-defined type (`MPID_PKT_HEAD_T`) packet. Actually, this packet is the header of ADI short packets (`MPID_PKT_SHORT_T`).

`MAD_REQUEST_PKT`: type of the rendez-vous mode request messages. The buffer contains an ADI-defined type (`MPID_PKT_REQUEST_SEND_T`) packet.

`MAD_SENDOK_PKT`: type of the acknowledgement message of the rendez-vous mode. The buffer contains an ADI-defined type (`MPID_PKT_OK_TO_SEND_T`) packet.

`MAD_TERM_PKT`: type for program termination message. The buffer is empty.

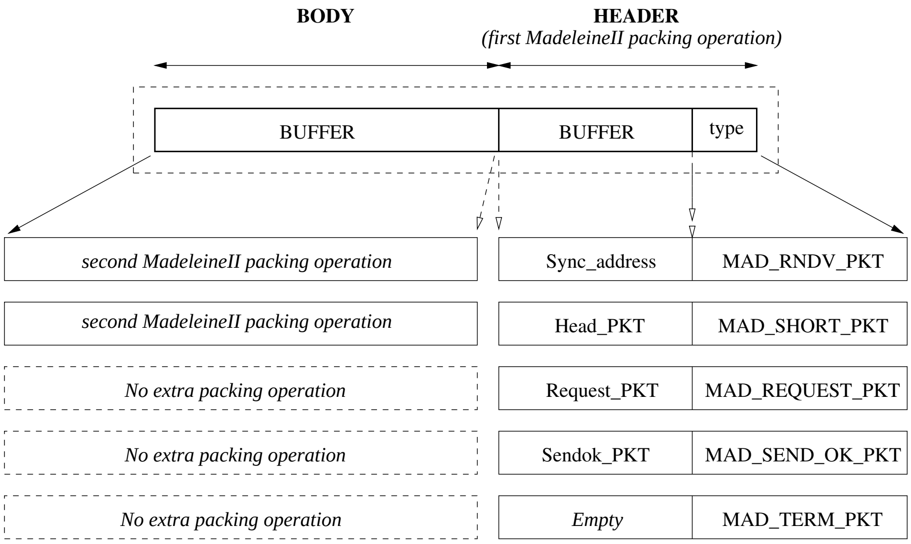


Figure 5: Packet Structure

4.2.2 Transfer Modes Issues

Switch Points Choice The message size threshold for selecting the rendez-vous mode rather than the eager mode is network-dependent. Experiments pointed out that the switch point values for TCP/Fast-Ethernet, SISCI/SCI and BIP/Myrinet were respectively of 64 KB, 8 KB and 7 KB using our hardware. Those values could be determined automatically in future works.

Yet, the ADI does not provide the adequate data structure to cope with this aspect of network heterogeneity. The `MPID_Device` structure of the ADI only reserves a single integer field to store the transfer mode selection threshold for a given device. In the `ch_mad` device case, although all networks are supported within a single MPICH device, a unique threshold had to be elected.

Experimentally, the network with the most influent switch point value over is SCI. So, when supporting multiprotocol feature, the switch point value for the `ch_mad` device is 8 KB (if SCI is a network supported within the material configuration). If not, the switch point of the most performant network is elected. That is, the SCI switch point value is prefered to the Myrinet value in the case of an hybrid SCI-Myrinet material configuration.

The Eager Mode This mode is used to send short messages (`MAD_SHORT_PKT` type). The ADI packet structure dedicated for short messages is not convenient for the case of a single device supporting network heterogeneity. Indeed, the ADI `MPID_PKT_SHORT_T` type definition includes a buffer of constant size (`MPID_PKT_MAX_DATA_SIZE`). This constant is defined in the ADI and should match the switch point value of a particular network (longer messages are sent with the rendez-vous mode).

Let us consider a cluster supporting multiple networks (e.g. SCI and TCP). Since those networks have different switch point values, the buffer size should at least equals the maximum possible switch point value among all networks. With the SCI switch size of 8 KB and TCP's one of 64 KB, all short messages should be sent within a 64 KB buffer. In the case of a message sent on the SCI network, the buffer will never be completely filled and a lot of *null* data will be sent too, thus wasting most of *Madeleine* capabilities.

Such a problem may be solved by splitting the ADI short packet in two parts: the header is sent within the buffer present in `ch_mad` message header, whereas the body of this ADI packet (that is, a user buffer) is directly sent as the body of`ch_mad` messages. On the receiving side the ADI short packet is reconstructed with those two chunks. This solution avoids a copy on the sending side and the `MPID_PKT_MAX_DATA_SIZE` ADI constant is only used on the receiving side to size the buffer storing incoming packets.

The *Rendez-vous* Mode This is the other transfer mode, used in the case of long messages. On receiving side, transaction is handled by an ADI `rhandle` structure. This structure has a field whose type is `MPID_RNDV_T`. In our case, it corresponds to a synchronization structure containing a semaphore and the address of the `rhandle` it belongs to. A transaction is identified with this structure. Since a `rhandle` is responsible for a communication, it identifies a unique transaction until its end. The same `rhandle` can then be reused to handle another communication (thus identifying another transaction).

Communication using the rendez-vous transfer mode involves three steps: 1) The sending process emits a request message to the other process; 2) As soon as the user data location address is known (i.e. a `rhandle` is chosen and is in charge of this transaction), the receiving process replies with an acknowledgement message containing the address of the synchronization structure associated with the `rhandle`; 3) The sending process may now proceed to data exchange with the other process since data destination is known, thus avoiding any intermediate copies. Indeed, the polling thread receiving the MPI data may find the `rhandle` responsible for the transaction because it receives the synchronization structure address from the sending process within the `ch_mad` message header.

During these steps, the main thread of the receiving process is blocked on the synchronization structure semaphore in order to wait for MPI data arrival. As soon as data are received, the polling thread of the receiving process releases the semaphore and the main thread may resume execution.

4.2.3 Using threads to Optimize Network Polling

Let us suppose that n different types of networks are present within the target cluster. One thread is used to poll each *Madeleine* channel dedicated to one particular network. Nevertheless, one other thread is necessary to perform calculation and execute application's MPI code. This latter thread is called the main thread or *MPI control thread*. Those $n+1$ threads are *persistent* threads, i.e. created during the application initialization phase (`MPI_Init`) and they remain active until the end phase of the program (`MPI_Finalize`). They also spawn *temporary* threads during a session: the MPI control thread creates a thread for each non-blocking send operation (typically, `MPI_Isend`), whereas each polling thread creates threads in order to perform request and acknowledgement operations of the rendez-vous transfer mode. Indeed, a polling thread must not proceed by itself to any send operation because deadlock situations might appear. Finally, although this MPICH version is multi-threaded in its conception, please note that it does not lead to a thread-safe version of MPICH since all threads are hidden to the user.

5 Performance Evaluation

This section exposes the performance evaluation obtained by the `ch_mad` device. The context is briefly explained and is followed by figures given for three different protocols/networks: TCP on Fast-Ethernet, SISCI on SCI and BIP on Myrinet. Those figures were obtained by compiling the device in a mono-protocol fashion (exactly like other regular devices) in order to prove that this *Madeleine*-based approach is well-founded: while network heterogeneity is supported, the `ch_mad` device also compares favourably to other devices specifically optimized for a particular protocol-network pair.

5.1 Preliminary Remarks

The hardware configuration used to perform the experiments is the following: the nodes are dual-PentiumII 450 MHz with 64 MB memory, the Myrinet boards are 32 bit,LANai 4.3-based with 1 MB on-board, the SCI boards are Dolphin's D310 and the Fast-Ethernet boards are DEC's 21140. The running operating system is Linux with kernel version 2.2.13 compiled in SMP mode.

The test program is a ping-pong between two nodes. The `mpptest` program was used to generate the results for the `ch_mad` device. Note that several performance figures have been furnished by the developing teams of other MPI implementations.

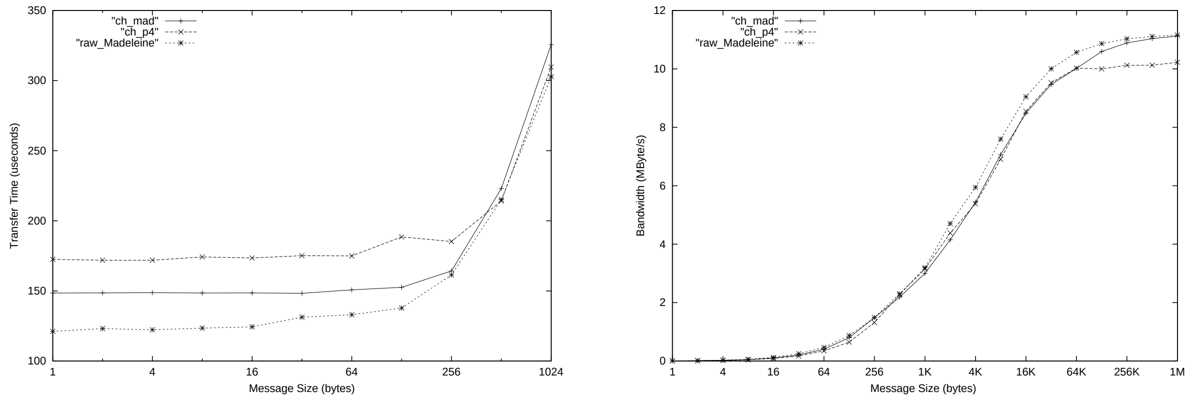
The figures show a performance gap between *Madeleine* and the `ch_mad` device. The explanation is generic and therefore given here. In order to measure a n-byte message latency and bandwith using *Madeleine*, only one pack (on sending side) or unpack (on receiving side) operation is required and used. In the `ch_mad` device case, we pointed out that several packing operations are mandatory for a single MPI message (cf. 4.2.1), hence generating significant overhead. A second overhead source is induced by MPICH since the time lost by polling threads to handle messages is taken into account for performance evaluation.

All results are expressed in Megabytes where 1 MB represents $1024 * 1024$ bytes.

5.2 TCP/Fast-Ethernet

We first show the results for the classical Fast-Ethernet network used with the TCP protocol. Comparison is made between *Madeleine* and the `ch_p4` MPICH device for TCP. Figure 6(a) shows that despite the introduced overhead, the `ch_mad` device performs better than the `ch_p4` device for messages size not exceeding 256 bytes. For longer messages, the difference between `ch_mad` latency and `ch_p4` latency is limited. Raw *Madeleine* latency is 121 μs while `ch_mad`'s one is 148μs. The 28 μs maximum overhead between raw *Madeleine* results and the `ch_mad` device results may be split in two parts: the cost of the additional packing operation is estimated at 21 μs (and decreases with message size) whereas the cost of messages handling is estimated to 7 μs. Those estimations were obtained experimentally.

Let us now analyze the bandwidth results (Figure 6(b)). There is a slight difference between raw *Madeleine* and the `ch_mad` device for messages whose size ranges from 1 KB to 256 KB. The maximum bandwidth gap is reached for 64 KB messages: beyond this size the rendez-vous mode is used and the gap is reduced. So, almost 100% of *Madeleine* bandwidth is delivered by the `ch_mad` device for long messages. As far as `ch_p4` is concerned, bandwidth performance is similar to `ch_mad`'s for messages smaller than 64 KB. For larger messages, `ch_p4` reaches its ceiling of 10 MB/s bandwidth whereas `ch_mad`'s still increases and even exceeds 11 MB/s.



(a) Transfer Time

(b) Bandwidth

Figure 6: Comparison between `ch_mad`, MADELEINE II and `ch_p4`

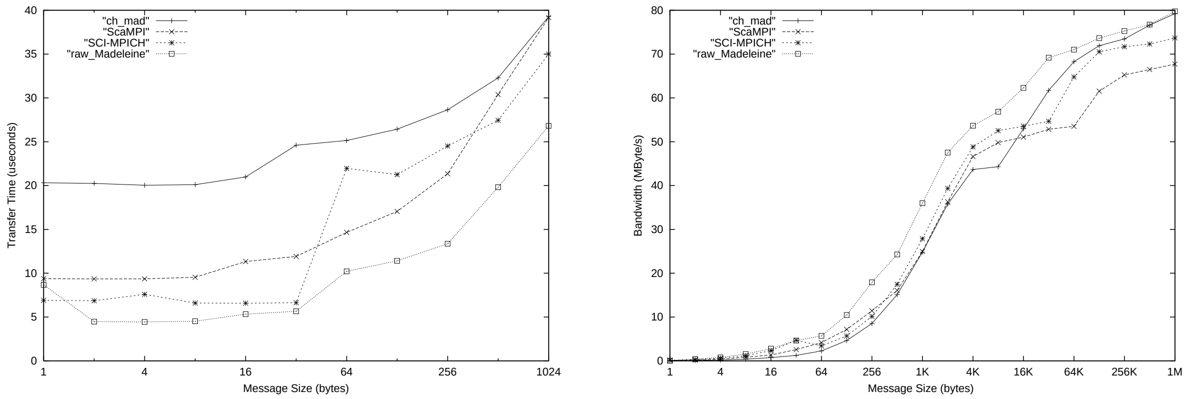
5.3 SISCI/SCI

This part is dedicated to results for SCI network with the SISCI protocol. The `ch_mad` device is compared to *Madeleine* and two other existing SCI-based versions of MPI:

- SCI-MPICH [17] which uses a device called `ch_smi` and is freely available and developed at RWTH Aachen.
- ScaMPI [2] is a commercial implementation of MPI above SCI network. It is developed by Scali Inc.

Latencies comparisons (Figure 7(a)) are not favourable to the `ch_mad` device since intermediate software layers are introduced (*Madeleine* and *Marcel*). As the two other versions of MPI above SCI are directly implemented on top of SCI hardware, they allow their designers to more tightly tune their possible optimizations for short messages. Nevertheless, the difference remains reasonable for messages larger than 512 KB. Raw *Madeleine* latency is 4.5 μs while *Madeleine*'s latency is roughly 20μs. Here again, the overhead introduced over *Madeleine* (15 μs) is decomposed into extra pack/unpack operation (6.5 μs) and message handling overhead (8.5 μs).

Bandwidth comparisons (Figure 7(b)) show better results for `ch_mad`, even if the gap between *Madeleine* and the `ch_mad` device is larger than for the TCP/Fast-Ethernet case. This fact is expectable since SCI is a more efficient network, thus more sensitive to the overhead introduced. But, just as in the TCP case, the device once again delivers almost all *Madeleine* bandwidth capabilities for large messages. The `ch_mad` bandwidth curve clearly shows the switch point between the eager transfer mode and the rendez-vous mode located at 8 KB. Before this point, even if `ch_mad` bandwidth is inferior or equal to other devices' bandwidth, it is still a valuable alternative. Once beyond the 8 KB point, the zero-copy feature of the rendez-vous mode strives to quickly reach the *Madeleine* bandwidth level. As a consequence, the `ch_mad` device outperforms both other devices for messages is larger than 16 KB with a sustained bandwidth of 80 MB/s and more.



(a) Transfer Time

(b) Bandwidth

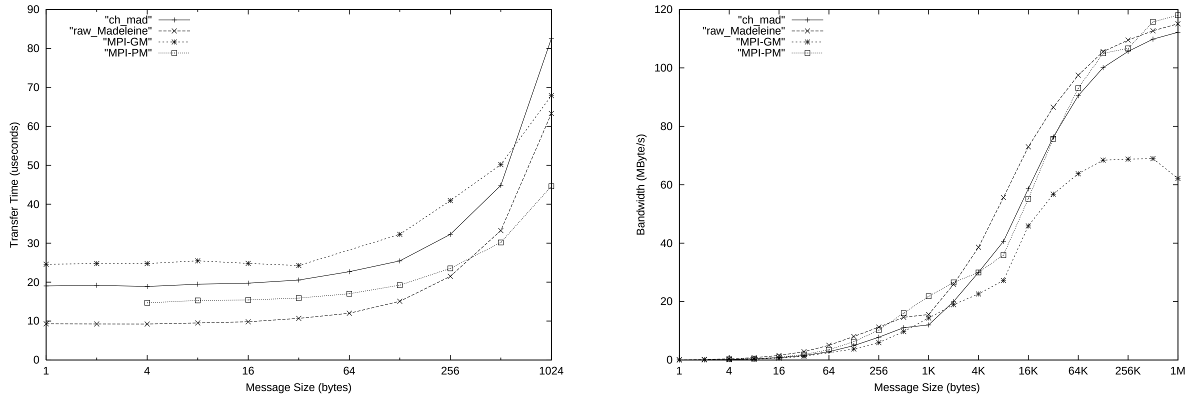
Figure 7: Comparison between `ch_mad`, *Madeleine*, ScaMPI and SCI-MPICH

5.4 BIP/Myrinet

We now expose performance evaluation for the Myrinet network. The protocol used was BIP [15]. Comparisons are made between raw *Madeleine* performance and two other MPI versions over Myrinet: MPI-GM ([1]) and MPICH-PM ([13]). The Myrinet hardware is the same in all cases, but the performance of MPICH-PM were measured on RWC PC Cluster II, which is a Pentium Pro 200 MHz-based cluster (and not a dual-PentiumII 450 MHz-based). The libraries used in these two other versions of MPI are respectively Myricom's GM (1.2.3 version) and RWCP's PM.

Latency results (8(a)) are the following: raw *Madeleine* latency is 9 μs and *Madeleine* latency is 20 μs. The overhead over *Madeleine* is 11μs (extra packing/unpacking operation: 4.5μs plus messages handling overhead: 6.5μs). For messages smaller than 512 bytes, `ch_mad` performs better than MPI-GM and presents a slight gap (5μs) with MPICH-PM. For larger messages, `ch_mad` compares unfavorably to MPI-GM (and even more to MPICH-PM).

Bandwidth comparisons (Figure 8(b)) show that MPI-GM is definitely outperformed by both `ch_mad` and MPICH-PM. The particular point for 1 KB-messages on th `ch_mad` curve is due to BIP's implementation. The switch point for the Myrinet network in the `ch_mad` device case is located around 7 KB. For messages smaller than 4 KB and larger than 256 KB, MPICH-PM takes the advantage over `ch_mad`. For messages between 4 KB and 256 KB, bandwidth performance is roughly the same.



(a) Transfer Time

(b) Bandwidth

Figure 8: Comparison between `ch_mad`, *Madeleine*, MPI-GM and MPICH-PM/SCore

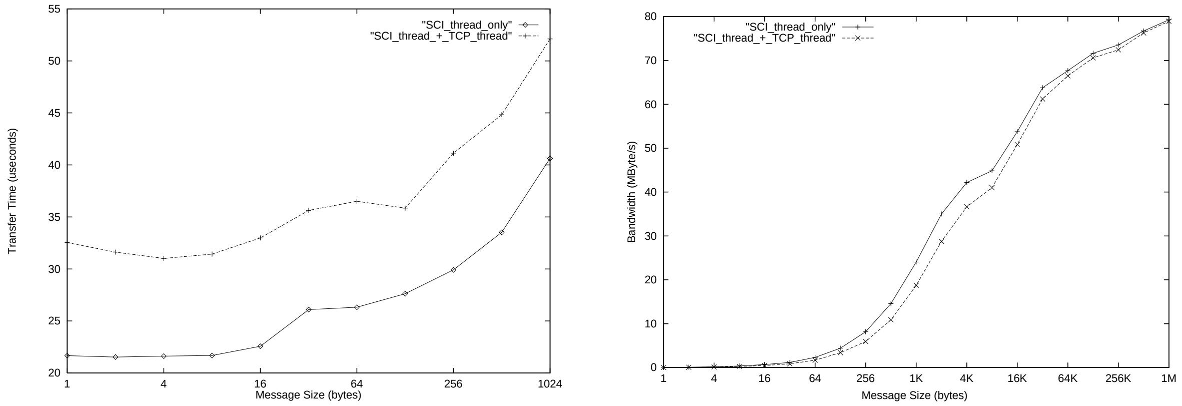
5.5 Of the Impact of Multi-Protocol Features

This section exposes performance evaluation for the multi-protocol feature. The curves show latency and bandwidth obtained when two polling threads are active. Figure 9 shows SCI performance when another TCP polling thread is also running. During the test, all communication is performed over the SCI network. The results show the loss of performance due to the existence of this second polling thread.

It is impossible to measure exactly this loss because the MPI application has a lot of influence on networks utilization. Nevertheless, the performance gap is directly linked with the secondary protocol supported (it depends on the *Madeleine* polling function implemented for a particular protocol). Indeed, additional TCP support determines the upper bound for more efficient networks such as SCI or Myrinet. In any cases, the gap remains limited and the performance of the `ch_mad` device with multi-protocol feature is very close to the device performance in mono-protocol mode.

6 Conclusion & Future Work

This report presents the design of MPICH/*Madeleine*, a true multi-protocol implementation of MPI for high performance networks. Our proposal is to build an efficient single-device implementation of MPICH on top of a natively multi-protocol communication interface called *Madeleine*. This is in major contrast with other approaches which either propose a multi-device implementation of



(a) Transfer Time

(b) Bandwidth

Figure 9: Comparison between SCI Alone and SCI + TCP

	TCP	BIP	SISCI
Latency (0 Byte Message)	130μs	16.9μs	13μs
Latency (4 Byte Message)	148.7μs	18.9μs	20μs
Bandwidth (8 MB Message)	11.2 MB/s	115 MB/s	82.5 MB/s

Table 2: Summary of Performance

MPICH (e.g. MPI-BIP) or interconnect multiple MPI implementation using standard TCP connections (e.g. PACX-MPI). The current implementation of MPICH/*Madeleine* allows applications to simultaneously exploit Myrinet-clusters, SCI-clusters and Ethernet-clusters connected together.

We report various performance numbers which highlight the relevance of our approach. In some cases, our multi-protocol MPICH implementation outperforms some native MPI ports in terms of latency and bandwidth! On another hand, we also made some experiments showing that the overhead of the multi-protocol management (due to network polling) is neglectible thanks to the careful integration of multi-threading into the *Madeleine* library.

Currently, our prototype is not able to forward packets across heterogeneous networks: all nodes have to be connected two-by-two by a direct network link. We are working on a low-level high-performance forwarding mechanism within *Madeleine* that will allow messages to cross gateway nodes while keeping the associated overhead as low as possible (especially in terms of bandwidth). We intend to investigate the impact of this mechanism within the MPICH/*Madeleine* in a near future.

References

[1] MPICH-GM. `http://www.myri.com/scs/index.html`.

[2] ScaMPI. `http://www.scali.com`.

[3] Olivier Aumage, Luc Bougé, and Raymond Namyst. A Portable and Adaptative Multi-Protocol Communication Library for Multithreaded Runtime Systems. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, volume 1800 of *Lect. Notes in Comp. Science*, pages 1136–1143, Cancun, Mexico, May 2000. Held in conjunction with IPDPS 2000. IEEE TCPP and ACM, Springer-Verlag.

[4] Jack Dongarra, Steven Huss-Lederman, Steve Otto, Marc Snir, and David Walker. *MPI : The Complete Reference*. The MIT Press, 1996.

[5] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Technical report, Argonne National Laboratory, 1996.

[6] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, and G. Thruvathukal ans S. Tuecke. Wide-Area Implementation of the Message Passing Interface. In *Parallel Computing*, volume 24, pages 1735–1749, 1998.

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

[8] Edgar Gabriel, Michael Resch, Thomsa Beisel, and Rainer Keller. Distributed Computing in a Heterogeneous Computing Environment. In Vassil Alexandrov and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Sciences. Springer, 1998.

[9] Patrick Geoffray, Loïc Prylli, and Bernard Tourancheau. BIP-SMP : High Performance Message Passing over a Cluster of Commodity SMP's. In *Supercomputing (SC'99)*, November 1999.

[10] Ewing Lusk and William Gropp. MPICH Working Note : the implementation of the second generation ADI. Technical report, Argonne National Laboratory.

[11] Ewing Lusk and William Gropp. MPICH Working Note : The Second-Generation ADI for the MPICH Implementation of MPI. Technical report, Argonne National Laboratory, 1996.

[12] Raymond Namyst and Jean-François Méhaut. PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.

[13] Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. *The Design and Implementation of Zero-Copy MPI using Commodity Hardware with a High Performance Network*, pages 243–250. ACM SIGARCH ICS'98. July 1998.

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

[15] Loïc Prylli and Bernard Tourancheau. BIP: a new protocol designed for high performance networking on myrinet. In *Parallel and Distributed Processing, IPPS/SPDP'98*, volume 1388 of *Lecture Notes in Computer Science*, pages 472–485. Springer-Verlag, April 1998.

[16] Loïc Prylli, Bernard Tourancheau, and Roland Westrelin. The design for a high performance MPI implementation on the Myrinet network. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Proc. 6th European PVM/MPI Users' Group (EuroPVM/MPI '99)*, volume 1697 of *Lect. Notes in Comp. Science*, pages 223–230, Barcelona, Spain, September 1999. Springer Verlag.

[17] Joachim Worringen. SCI-MPICH: The Second Generation. In Geir Horn and Wolfgang Karl, editors, *Conference Proceedings of SCI Europe 2000*, pages 11–20. SINTEF Electronics and Cybernetics, 2000.