# Madeleine II: A Portable and Efficient Communication Library for High-Performance Cluster Computing ⋆

Olivier Aumage [1], Luc Bougé, Jean-François Méhaut, Raymond Namyst

*LIP, ENS-Lyon, 46, Allée d'Italie, F-69364 Lyon Cedex 07, France*

**Abstract**

This paper introduces *Madeleine II*, an adaptive and portable multi-protocol communication library for high performance multithreaded applications. *Madeleine II* has the ability to control multiple network protocols (BIP, SISCI, VIA) and multiple network adapters (ETHERNET, MYRINET, SCI) within the same application session. Moreover, it includes advanced mechanisms to dynamically select the most appropriate transfer method for a given network protocol according to various parameters such as data size or responsiveness user requirements. We report on performance measurements obtained using various protocols and we present preliminary results about porting the MPICH and the NEXUS communication libraries on top of *Madeleine II*.

*Key words:* Multi-protocol communication, high-performance cluster computing, multi-threaded programming environment, metacomputing.

## 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

⋆ A preliminary version of this work has been submitted for presentation at the *Cluster 2000* Workshop, Chemnitz, Germany, December 2000.

[1] Contact: `Olivier.Aumage@ens-lyon.fr`.

such environments mainly involves remote service request (RSR), remote procedure call (RPC) or remote method invocation-like (RMI) interactions. This is obviously true for environments providing a RPC-based programming model such as NEXUS [1] or PM2 [2], but also for others which often provide functionalities that can be efficiently implemented by RPC operations.

We have shown in [3] 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 this first implementation were strongly message-passing oriented. Consequently, the support of non message-passing network protocols such as SCI [4] or even VIA [5] was cumbersome and introduced some unnecessary overhead. In addition, no provision was made to use multiple network protocols within the same application. For these reasons, we decided to design *Madeleine II*, a full multi-protocol version of *Madeleine*, efficiently portable on a wider range of network protocols, including non message-passing ones.

Section 2 presents the generic communication interface provided 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 demonstrates this organization in action while transmitting a message. The implementation of the VIA driver of *Madeleine II* is presented in more detail as a case study. This section is followed by an evaluation of *Madeleine II* over several high performance network protocols and a presentation of *Madeleine II* as a low level communication layer for two famous communication libraries: MPICH [6] and GLOBUS/NEXUS [1].

## 2 An Interface to Multiprotocol Communication

### *2.1 Basic Concepts*

*Madeleine II* aims at enabling an efficient 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: *channels* and *connections*. 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

| | |
|---|---|
| `mad_begin_packing` | Initiates a new message |
| `mad_begin_unpacking` | Initiates a message reception |
| `mad_end_packing` | Finalizes an emission |
| `mad_end_unpacking` | Finalizes a reception |
| `mad_pack` | Packs a data block |
| `mad_unpack` | Unpacks a data block |

Table 1
Functional interface of *Madeleine II*.

with a network protocol, a corresponding network adapter and a set of *connection* objects (much like an MPI communicator). Each connection object virtualizes a point-to-point reliable network connection between two processes belonging to the session. It is obviously possible to have several channels related to the same protocol and/or the same network adapter, which may be useful to split communication into different logical groups within an application. 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 provides a small set of primitives to build RPC-like communication schemes. These primitives actually look like classical message-passing-oriented operations. 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 (FM) [7] or NEXUS [1], *Madeleine II* allows applications to incrementally build messages to be transmitted, possibly at multiple software levels. To illustrate why this is useful, 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 is examined first by the multithreaded runtime (e.g., to extract the name of the function that will be executed by the communication thread), and then by the user application (e.g., 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`. Eventually, the message construction is finalized using `mad_end_packing`. This final step guarantees that the whole message has been actually transmitted to the receiving side through 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, e.g. FM and NEXUS. 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. 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.

On the receiving side, a message is received through a similar series of calls. The reception is initiated through a call to `mad_begin_unpacking` on the given channel object. Then, each data block is extracted in turn from the message through a call to `mad_unpack`. Eventually, the reception is finalized through a call to `mad_end_unpacking`: this final step guarantees that the whole message has been actually extracted from the network.

For the sake of efficiency, the messages are not *self-described* at the level of *Madeleine II*: the data blocks should be *unpacked* precisely in the order as they were *packed*, with the same flag specification. The following flags control the reception of the data blocks:

**`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 are needed to issue the following *unpacking* calls. On some network protocols, this functionality may be available for free. On some others, it may put a high penalty on latency and bandwidth. 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

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);
```

Fig. 1. Sending and receiving messages with *Madeleine II*.

the corresponding data is transmitted as efficiently as possible.

It should be stressed that the emission and reception flags only specify the *external semantics* of the operations. *Madeleine II* remains free to select the most appropriate scheduling to *actually* ship the data blocks over the network, and to extract them from the network. For instance, some network technology may make it more efficient to transmit the successive data blocks at each packing step. Alternatively, it may be more efficient to keep data in place and delay actual transmission as long as possible for some other technologies. It may even be interesting to copy data into driver-preallocated buffers and to delegate the actual shipping to the driver. The role of the `end_paking` and `end_unpacking` finalization calls is precisely to guarantee that the specified actions have actually been completed. This behavior is quite similar to the buffered `write` primitive in Unix systems: only after a call to the `sync` primitive is the user guaranteed that all prescribed actions have actually completed. Allowing such an *asynchronism* is the key for efficient implementations.

### *2.3 Example*

Figure 1 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

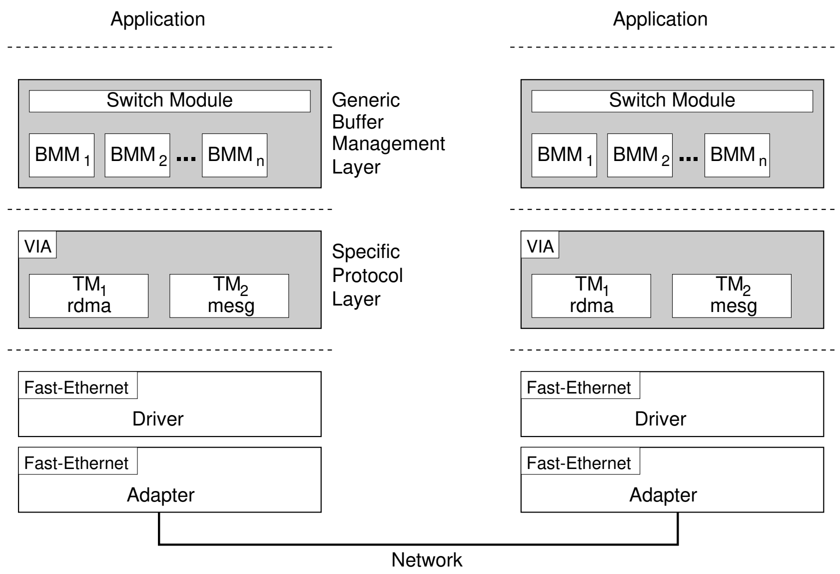


Fig. 2. *Madeleine II*'s modular architecture.

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.

## 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 very tightly fit and optimally exploit the specific characteristics of each target network.

*Madeleine II* is organized as two software layers (Figure 2), following a commonly used scheme. 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 from the supported network protocols and is in charge of the management of buffers. It is made of several *Buffer Management Modules* (BMM), each of these implementing a given buffer management policy.

| | |
|---|---|
| `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 |
| `allocate_static_buffer` | Allocate a protocol dependent buffer |
| `free_static_buffer` | Free a protocol dependent buffer |

Table 2
Functional interface of TMs.

### *3.2 Transfer Management*

One of the goal of *Madeleine II* is to support multimodal protocols such as VIA [5] or SCI [4]. Such protocols provide several data transfer methods, namely a *regular* transfer mode and a DMA (Direct Memory Access) mode. Moreover, it should be able to easily take into account protocols such as BIP [8] which makes a difference between *short* buffers and *long* buffers. As a consequence, *Madeleine II* features specific modules to encapsulate each of these *sub-protocols*. These modules are called Transmission Modules (TM).

Table 2 displays the interface of each TM (note that some functions may not be relevant for a specific TM and will not be implemented in this 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 such as SBP (*Stream Buffered Protocol*, [9]) which provide their own set of preallocated buffers. The asymmetry between buffer group emission and reception will be explicited later in this paper.

### *3.3 Protocol Management*

TMs are grouped into Protocol Management Modules (PMM). There is one PMM for each supported protocol (e.g., BIP/MYRINET or TCP/FAST-ETHERNET). Each PMM implements whole or part of a generic set of functions. This set of functions constitutes the protocol driving interface. It insures independence between the upper layer and the communication protocols.

The protocol management modules are based on a hierarchy of data structures:

**`mad_driver_t`:** This structure contains data common to the whole PMM and virtualizes a *Madeleine II* network driver.

**`mad_adapter_t`:** *Madeleine II* allows each driver to control several Network Interface Cards (NIC). Each of these is represented by a `mad_adapter_t` structure.

**mad_channel_t:** Each adapter may be used by several channels. Channel specific data are contained into the `mad_channel_t` structure.

**mad_connection_t:** A channel contains a set of point-to-point network connections, each corresponding to a `mad_connection_t` data structure. Depending on the PMM's protocol characteristics, connections may be bi-directional (in which case they share their protocol specific data with the reverse connection) or uni-directional.

**mad_link_t:** As mentioned, a PMM may contain several transmission modules. Each TM is conceptually represented as a separate *link* going through a point-to-point connection.

### *3.4 Buffer Management*

While some TM can benefit from grouped buffer transfers, other may not, 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 `allocate_static_buffer` and `free_static_buffer` functions. This aspect is specified through the `buffer_mode` setting, whose value may be either `static` or `dynamic`.

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. On the contrary, a BMM may adopt an eager behavior and send buffers as soon as they are ready. The value of the `link_mode` setting is either `buffer` or `buffer_group`. In the former case, the buffers are sent as soon as they are ready, while in the latter case, the BMM attempts to group buffers before transmitting them to the TM. The buffer management layer may dynamically switch setting from `buffer` to `buffer_group` if immediate transmission is not allowed. This latter case occurs for instance if a `send_LATER/receive_EXPRESS` pack has been requested during the message construction.

The semantics of the `receive_EXPRESS` flag makes an obligation for *Madeleine II* to immediately provide the requested piece of data without waiting for any information about the rest of the message. This raises a difficulty if the data blocks are grouped together on the sending side, as it may occur with the `send_LATER` flag. In this case, *Madeleine II* must be able to process a large message as several *smaller* buffer groups, each time a piece of data is requested in `receive_EXPRESS` mode. This is precisely the intent of the

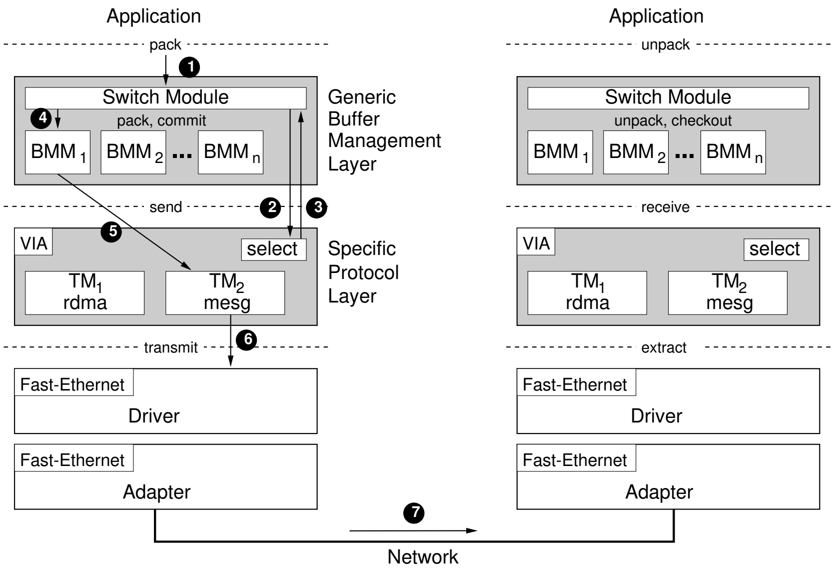


Fig. 3. Conceptual view of the data path through *Madeleine II*'s internal modules.

`receive_sub_buffer_group` function in the TMs interface. The `aggregation_mode` setting indicates whether the underlying network may perform efficient sub-buffer-group extraction, as of TCP for instance. The `asymmetric` mode specifies it is so. If it is not the case, the `symmetric` mode is selected and the BMM shall perform a group flush on the sending side on each `receive_EXPRESS` request.

## 4 A Message Transmission Step-by-Step

We now demonstrate the *Madeleine II* components in operation while transmitting an application message. As an illustration, the implementation of the VIA driver is presented in greater detail.

### 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 in 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 Figure 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 and transmits it to the Driver (Steps 6). The buffer is then eventually shipped to the 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 (*transmit* on Fig. 3) to ensure that any remaining delayed packet has been shipped to the network. A general *transmit* 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 for the sake of efficiency. The Switch Step is performed on each `unpack` and must select the *same* TM as on the sending side. For instance, a packet sent by the DMA Transmission Module of VIA must be received by the same module on the receiving side. The *extract* function (dual to the *transmit* 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. Eventually, the final call to `end_unpacking(connection)` ensures that all expected packets are made available to the user application.

### *4.3 Case Study: VIA*

The functional versatility of VIA [5] makes it a nice example to illustrate the interaction between the TMs and the buffer management layer described in the former paragraphs.

Supporting network protocols like VIA requires polymorphic capacities from the network management layer interface. Indeed, VIA allows data to be sent using the traditional send/receive primitives or by performing remote write operations with direct memory access. Moreover, data areas to be transfered by the VIA protocol must have first been registered — both on the sending side and on the receiving side — to ensure that the corresponding memory pages are pinned into physical RAM. The registering operation is quite expensive on the current implementations of VIA [10,11], and one can consider two alternative strategies: one may either choose to manage a pool of pre-registered buffers and copy user data into and from these buffers; or dynamically register user data areas before network transfers. To sum up, there is a trade-off between the cost of an extra copy and the cost of the registration operation, the former being more rewarding for small-sized pieces of data.

As a consequence we have to consider a set of four possible combinations. Three of them are currently implemented (as three different TMs) into the *Madeleine II* protocol management module of VIA:

- Pre-registered buffer pool, message passing primitives ($<$ 5 kB data blocks).
- Dynamic registration, message passing primitives ($<$ 32 kB data blocks).
- Dynamic registration, remote DMA write primitive ($\geq$ 32 kB data blocks).

The boundaries have been adjusted from our *Madeleine II* VIA driver over the (software) M-VIA [12]/FAST-ETHERNET implementation on a LINUX/PII-450 cluster. The 5 kB limit was experimentally determined and corresponds to the minimal block size for which dynamic registration becomes cheaper than an extra copy.

The 32 kB limit is dictated by the MTU (Maximum Transfer Unit) of M-VIA using our hardware component: it is the maximal size of messages which can be sent through a unique transfer. Hence, transferring a message longer than 32 kB in message-passing mode, would require to implement software flow control at the *Madeleine II* VIA driver level. The RDMA-Write mode of VIA enables a better implementation. The receiver side registers the whole buffer at once, and acknowledges the sender. Then, the sender issues enough RDMA-Write transactions to send the buffer without requiring any operation from the receiver. Only one receiver ack is needed with this method. In contrast, the message-passing way would be much more expensive, even with a credit based flow-control scheme.

It is also possible to take advantage of the gather/scatter capabilities of VIA's message-passing mode to issue one-step burst data transfers when possible. This strategy is rewarding for *medium-size blocks* scattered in user-space and this is why the corresponding TM use the `buffer_group` mode. For *small blocks* accumulated into static buffers however, it is more efficient to immediately transfer buffers as soon as they get full: it suffices to set `link_mode` to `buffer`. This enhances pipelining and overlaps the additional copy involved.

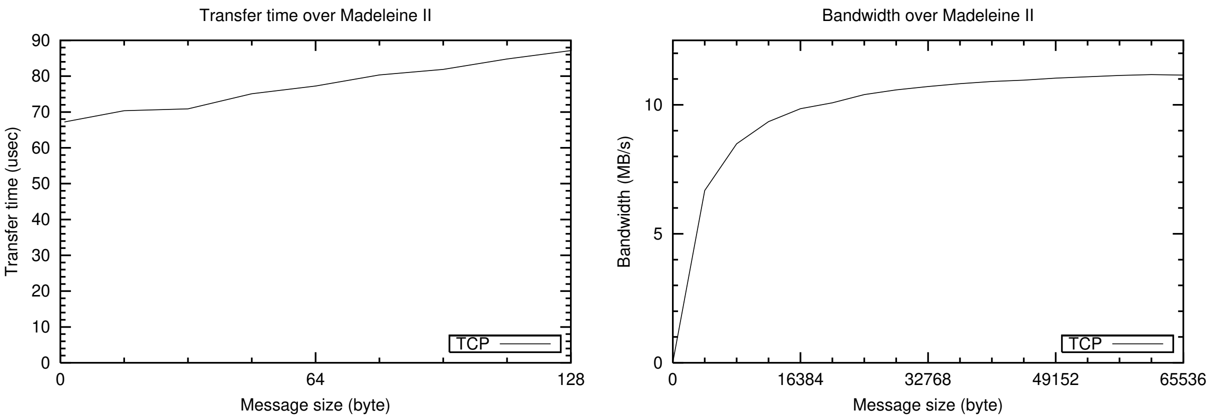


Fig. 4. Latency and bandwidth over TCP/FAST-ETHERNET

## 5 Implementation and Performance

### 5.1 *Testing Environment*

Let us now have a look to the performance achieved by *Madeleine II*. 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 (Kernel 2.1.130 for VIA, and Kernel 2.2.13 otherwise). The cluster interconnection networks are 100 Mb/s FAST-ETHERNET for TCP and VIA, DOLPHIN SCI for SISCI and MYRINET for BIP.

### 5.2 Madeleine II *Drivers*

#### 5.2.1 *TCP*

The results of tests run on the TCP/IP protocol using standard UNIX sockets are plotted on Figure 4. *Madeleine II*'s TCP driver delivers most of the FAST-ETHERNET bandwidth available through TCP with more than 11 MB/s. Minimal latency is below 70 $\mu$s with a uniprocessor-compiled LINUX kernel and around 110 $\mu$s with an SMP LINUX kernel.

#### 5.2.2 *VIA*

Figure 5 illustrates the interest of an adaptive multi-modal protocol support with the VIA protocol using the M-VIA 0.9.2 software implementation from the NERSC (National Energy Research Scientific Computing Center, Lawrence Berkeley Natl. Labs.).

We can see that none of the dynamic user buffer registration method and the pre-registered buffer pool method are *globally* optimal for the whole message size range. In

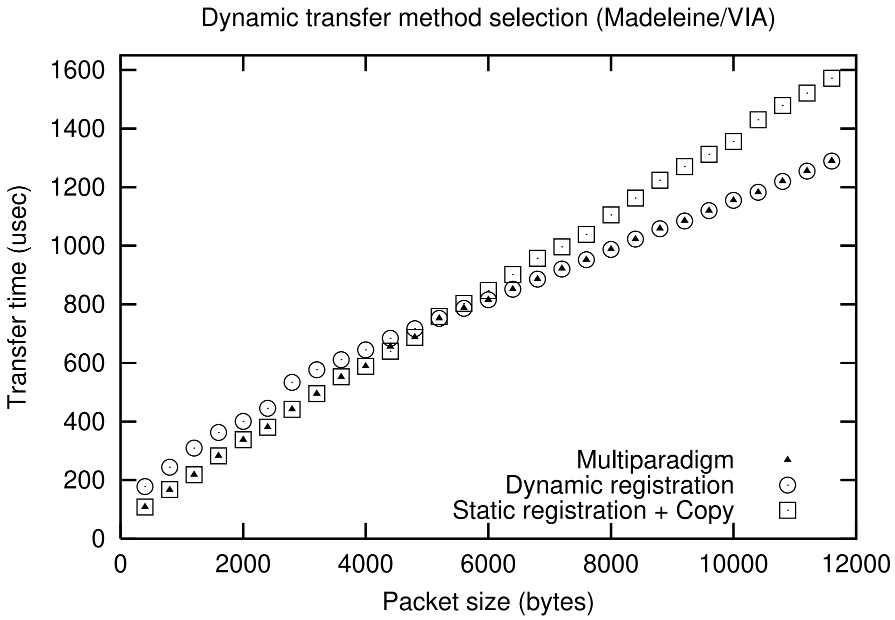


Fig. 5. Adaptive multi-modal protocol support with VIA

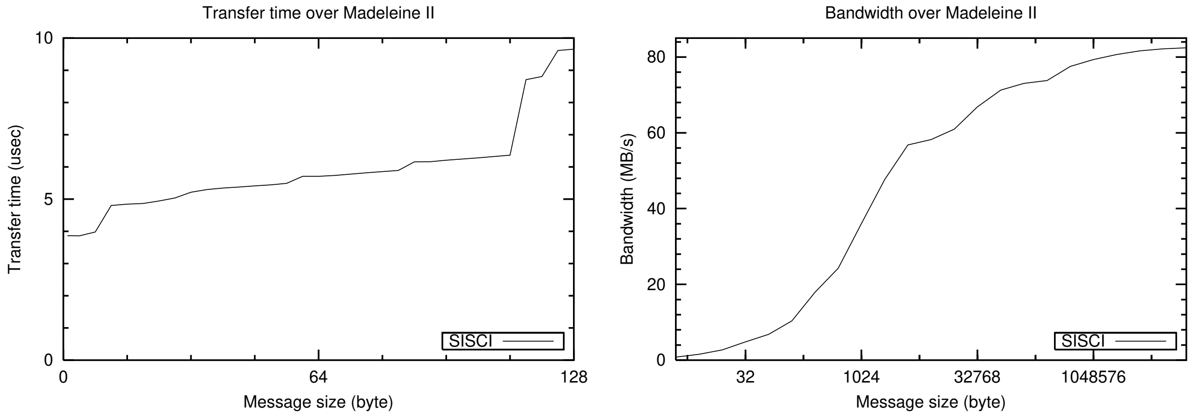


Fig. 6. Latency and bandwidth over SISCI/SCI

contrast, the multiparadigm protocol support provided by *Madeleine II* selects the first VIA TM for messages shorter than 5 kB and the second VIA TM for messages longer than 5 kB which results in the *Madeleine II* VIA driver being efficient on the full message length range.

### *5.2.3 SISCI*

**Results** The performance measurements of the SISCI driver are shown on Figure 6. We can see that the minimal latency is very low (3.9 $\mu$s), thanks to our highly optimized short message TM (see implementation details below). The large latency increase for messages larger than 112 bytes is due to the switch from the TM optimized for small messages to the TM dedicated to middle-sized and large messages.

The bandwidth is also very good thanks to the use of an adaptive double-buffer algorithm (activated for data blocks longer than 8 kB) into the regular SISCI TM which allows *Madeleine II* to deliver a bandwidth of 82 MB/s.

**Implementation details** The SISCI driver handles both transmission modes provided by the SISCI interface: a *regular* remote memory write mode and a *DMA* mode. Note that the DMA mode TM is not currently active because of the very poor performance of the SCI DMA: we have not been able to get more than 35 MB/s as of now. In contrast, the regular transfer mode TM delivers about 82 MB/s.

Some special care is needed to avoid any delay while feeding the network interface card with data. Indeed, if only one 64 kB SCI segment was used to send data to a given remote host, the sender would have to wait for the segment to be read by the receiver each time the segment becomes full. Using an additional segment solves this problem: the sender starts filling the second segment as soon as the first one becomes full. Moreover, data from the first segment is more likely to have been read by the receiver when the sender switches back to the first segment, reducing further the potential delay. Thanks to this *double-buffer* technique, we were able to increase the delivered bandwidth by more than 20 MB/s. This mechanism is activated for messages larger than 8 kB. It is *adaptative* in the sense that it always sends messages in at least 4 balanced chunks of data in order to deliver a large bandwidth for a wide range of message sizes.

Three different transmission modules have been implemented for the SISCI driver. Among them, the *regular* mode uses an additional TM specifically optimized for short message transfer, presented in the following paragraphs.

SCI memory segments use a special caching feature of PENTIUM-like microprocessors called *write-combining buffer*. Memory segments set to use write-combining caching are not written synchronously by the microprocessor. Instead, the processor waits a little while after a write operation to get a chance to aggregate other succeeding write operations into a write combining buffer. Each such buffer is currently 64 bytes long. Then, the whole buffer is written to memory (e.g. the SCI segment in this case) as a single bus transaction. As a counterpart, the order of write operations is not guaranteed to be preserved as seen by the bus.

Consequently, the unoptimized, regular SISCI TM sends a message in two phases:

- The message contents is copied to the SISCI segment and a flush is performed to empty the write-combining buffer.
- A *message-ready* mailbox flag is set in the SISCI segment and the segment is flushed again to indicate to the receiver that a message may now be read.

The first flush is definitely necessary. Otherwise, the *message-ready* flag toggle could be spotted before the message itself on the receiving side (it really happens that way experi-

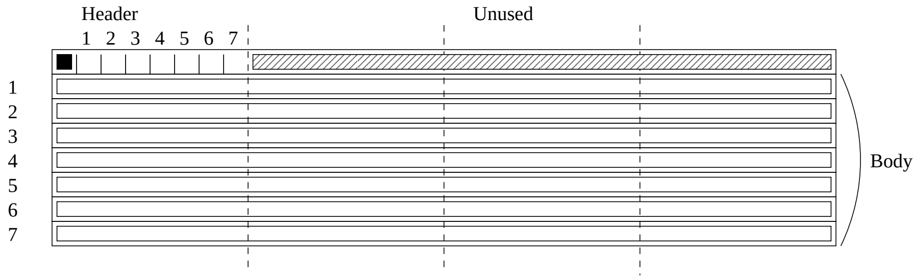


Fig. 7. Area layout for optimized transfer

mentally!).

Yet, these two flush operations are expensive. This is why a second TM implements an optimization mechanism to avoid one of these flushes for small messages. Only the first 128 bytes of the SISCI segment are used by this TM. Initially, this 128-byte zone is filled with 0 and considered as four 32-byte areas. Figure 7 shows the layout of one of these areas. The first 32-bit integer of each area is used to describe the contents of the corresponding area as follow:

- The first bit is always set to 1 (this bit simulates the *message-ready* mailbox).
- The next 7 bits correspond to the 7 following 32-bit integers of user data. Each bit is set to 0 if its corresponding integer is null and 1 otherwise.
- The next 24 bits are reserved for later use.

Let us now see how the message is received. The receiver loops reading the first 32-bit integer (the header) of the segment until it becomes $\neq 0$. Then, for each of the 7 following 32-bit integers, two cases are possible:

- If the corresponding bit in the header is set to 1, the receiver loops until the integer is detected (i.e. becomes not null), copies it into the user destination buffer and resets the integer to 0 in the segment.
- if the corresponding bit is 0, the receiver skips the integer and writes a 0 in the user destination buffer.

This process is iterated for each of the four areas. Hence, up to 112 bytes may be sent this way (hence the 112-byte gap on the *Madeleine II*/SISCI latency curve). Of course, it would have been possible to use areas larger than 32 bits, but this value yields the best results experimentally. Note that this value could by no means exceed the size of the write combining buffer: the header is always written after the user data (it is built on the fly) and would then be sent after the message body, generating additional (unaligned) and expensive bus transactions.

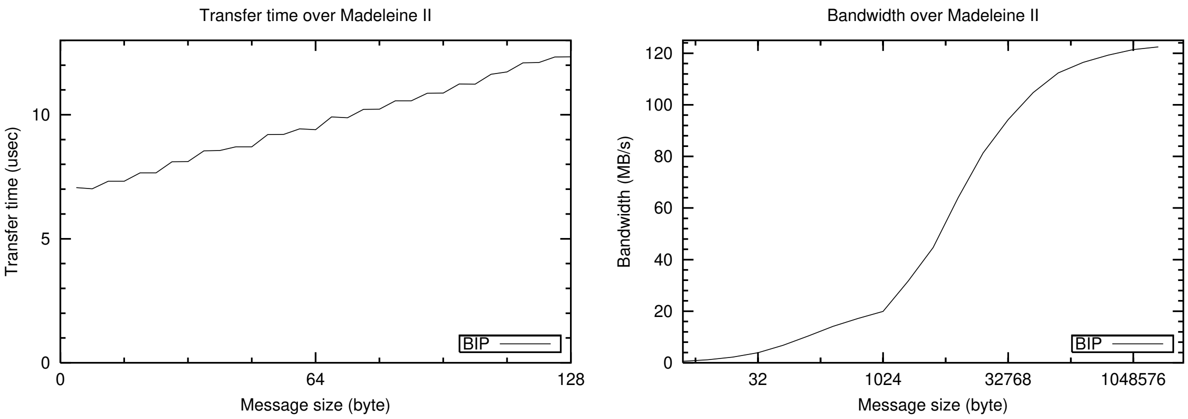


Fig. 8. Latency and bandwidth over BIP/MYRINET

### *5.2.4 BIP*

BIP (Basic Interface for Parallelism) is a low-level communication interface specifically designed for the MYRINET network protocol [8]. The most interesting functionality of BIP is to provide communication control in user space: the application may interact directly with the network interface card. The BIP interface makes a distinction between short messages ($<$ 1 kB) and long messages. Consequently, the BIP driver of *Madeleine II* features two different transmission modes.

In BIP, each node manages a fixed, pre-allocated pool of static buffers to store incoming short messages. This saves a synchronization operation between the sender and the receiver in this case. The buffer pool is a simple circular queue. For the sake of efficiency, BIP does not perform any check about the number of buffers which have been filled by incoming messages and not yet read: it is left to the upper communication layers. Therefore, the *short message* TM implements a credit-based flow-control algorithm to avoid overflows. Credits are piggy-backed in regular messages whenever possible.

In contrast with short messages, BIP strives to avoid copies in transmitting long messages. In this case, a strict rendez-vous synchronization is necessary between the sender and the receiver: the receiver must acknowledge the sender that it is ready to store the data before a message is actually transmitted. Therefore, the *long message* TM implements a *handshake* synchronization scheme, which involves a sequence of three messages on the network.

The results obtained with this *Madeleine II*/BIP driver are very good. The minimal latency (message of 4 bytes) is 7 $\mu$s and the asymptotic bandwidth is of 122 MB/s (Figure 8). Theses results are very close to the raw BIP results: 5 $\mu$s minimal latency and 126 MB/s maximal bandwidth.

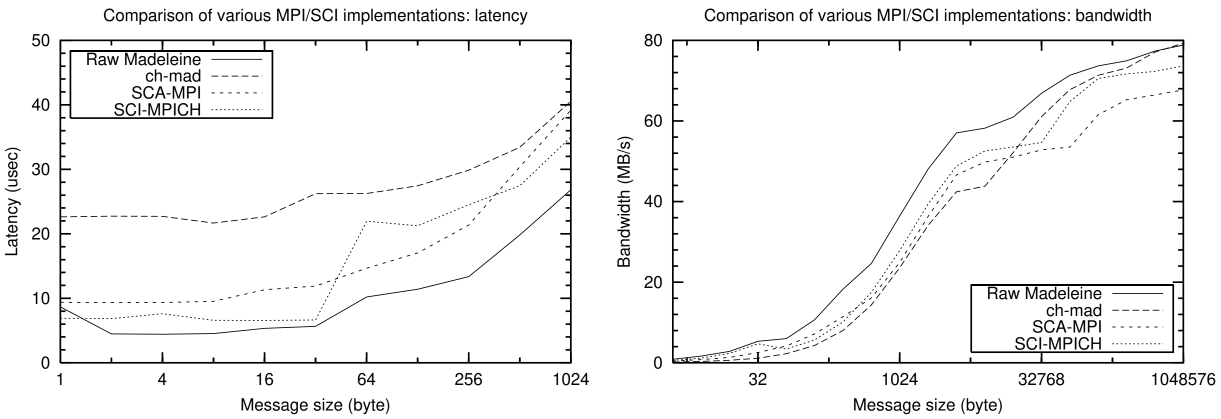


Fig. 9. Comparison of various MPI implementations over SCI

### *5.3* Madeleine II *as a basis for high-level communication libraries*

We now present two implementations of high-level communication libraries — namely MPICH and GLOBUS/NEXUS — over *Madeleine II*.

#### *5.3.1 MPICH*/Madeleine II

MPICH [6] is a free, portable implementation of the MPI specification [13]. MPICH is widely used in the scientific community. *Madeleine II* has been integrated into MPICH by Guillaume Mercier [14] as a new *device* of the *Abstract Device Interface*. The device is called *ch_mad*. The goal was to let MPICH transparently benefit of the multi-protocol features of *Madeleine II*.

The device uses two message transfer modes. The *eager* mode is dedicated to short messages: this modes does not require nodes to be tightly synchronized, because data are copied to an internal buffer on the receiving side, before being actually put into user areas. The *rendez-vous* mode performs zero-copy transmission for large messages. As a counterpart, the receiver must explicitly acknowledge the sender that it is ready to receive the message. The transition boundary between the two modes depends on the underlying protocol: it is 9 kB over SISCI, and 64 kB over TCP/FAST-ETHERNET on our hardware.

Preliminary performance measurements show promising results. Figure 9 compares MPICH/*Madeleine II*/SISCI to two other implementations of MPI over SCI, namely SCI-MPICH [15] and the commercial version SCAMPI [16]. The performance curves of *Madeleine II* over SISCI (without MPICH) are plotted too in order to show the current overhead of our MPI/*Madeleine II* implementation.

While SCI-MPICH and SCAMPI directly control the SCI hardware, our implementation has to transmit data through both *Madeleine II* and SISCI communication layers. Hence, latency results compare defavorably to those achieved by direct implementations of MPI

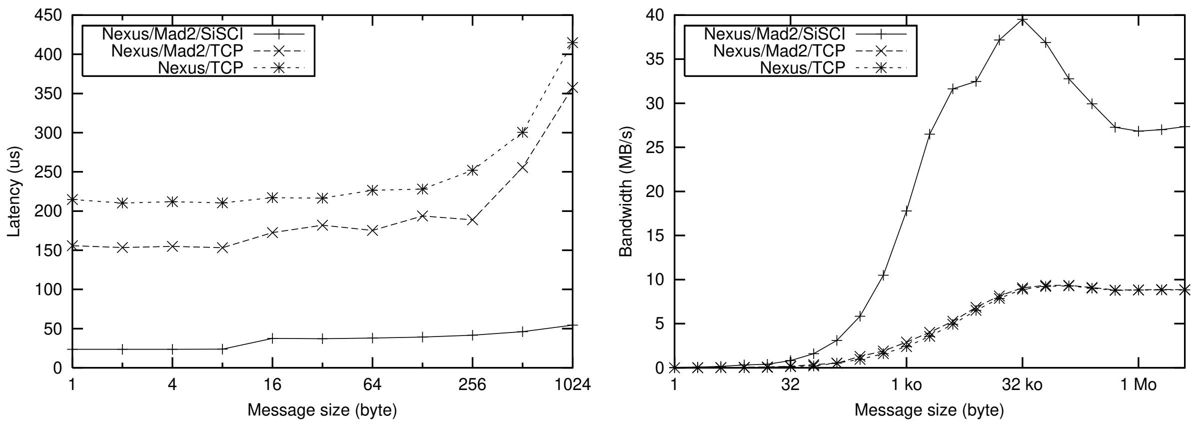


Fig. 10. NEXUS/*Madeleine II* performance

over SCI, although the difference decreases for messages larger than 512 bytes.

Yet, we can see that things are much different as far as bandwidth is concerned. The *ch_mad* device features performance very close to the other implementations, with the *eager* mode (< 9 kB ). It even outperforms these implementations when the *zero-copy* mode is activated. Moreover, this device is able to use most of the bandwidth provided by *Madeleine II* for large messages.

### 5.3.2 *Nexus*/Madeleine II

NEXUS is the communication library of the GLOBUS programming environment devoted to build *computational grids* of large-scale metacomputing [1,17]. *Madeleine II* has also been integrated into NEXUS by Alexandre Denis [18] as a new *module* called `GLOBUS_MAD2`.

While NEXUS has proven fully valuable 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 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 is point-to-point connection-oriented while *Madeleine II* is cluster-oriented. Figure 10 shows the level of performance achieved by our implementation over *Madeleine II*/TCP and *Madeleine II*/SISCI. It is clear that even with a rather heavy interface and without any specific optimization, our NEXUS/*Madeleine II* implementation is very effective on high-performance network like SCI (with a minimal latency below 25 $\mu$s) and offers a more interesting solution as far as cluster computing is concerned. Though, we can see that the two additional copies imposed by NEXUS tremendously affect the bandwidth of the system on large messages: while *Madeleine II* provides up to 82 MB/s using SISCI, NEXUS/*Madeleine II* only reaches 40 MB/s before decreasing down to 27 MB/s.

NEXUS features multiprotocol support [19] and *Madeleine II* is currently seen as one protocol by NEXUS. Hence, we can easily imagine Globus applications using regular TCP/NEXUS protocol for wide area transmission and the '*Madeleine II*' NEXUS protocol for local cluster high-performance computation. These applications would seamlessly benefit of local efficient communication components, without having to be rewritten.

## 6 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, dynamic *most-efficient transfer-method* selection mechanism. It currently runs on top of BIP, SISCI, TCP, VIA, SBP and common MPI implementations. We reported very interesting performance results on top of BIP/MYRINET and SISCI over a SCI network.

We also showed the effectiveness of *Madeleine II* as a foundation for higher level communication libraries and introduced two implementations: MPICH/*Madeleine II* and NEXUS/*Madeleine II*. Preliminary performance figures are highly encouraging. MPICH/*Madeleine II* even outperforms the current best implementations of MPI over SCI as far as bandwidth is concerned.

Other communication libraries have recently been designed to provide portable interfaces and/or efficient implementations to build distributed applications. However, very few of them provide an efficient support for RPC-like communication schemes, support for multi-protocol communications and support for multithreading.

ILLINOIS FAST MESSAGES (FM) [7] provides a very simple mechanism to send data to a receiving node that is notified upon arrival by the activation of a handler. Releases 2.x of this interface provide interesting gather/scatter features which allow an efficient implementation of zero-copy data transmissions. However, it is not possible to issue a transmission with the semantics of the `receive_CHEAPER` *Madeleine II* flag: only `receive_EXPRESS`-like receptions are supported, and it is not possible to enforce aggregated transmissions.

The PANDA multithreaded runtime system [20] has been developed as a support for the ORCA parallel object language. It provides a communication interface which is very close to ours. Its goal is to support very efficient RPC operations on message-passing networks. In contrast with *Madeleine II*, Panda does not provide alternative semantics for packing and unpacking operations: in *Madeleine II* parlance, all packs are issued in a mode similar to `send_LATER`/`receive_EXPRESS`. Also, there is no notion of an incremental construction of messages: it makes it difficult to implement multi-layered protocols stacks.

The NEXUS multithreaded runtime [1] features a multi-protocol communication subsystem very close to the one of *Madeleine II*. The messages are constructed using similar *packing* operations except that no "high level" semantics can be associated to data: there is no notion of `CHEAPER` specifications, which allows *Madeleine II* to choose the best suited strategy. Also, as for FM, *unpacking* operations behave like `receive_EXPRESS` *Madeleine II* transmissions.

We are now actively working on various *Madeleine II* improvements, namely letting *Madeleine II* run across clusters connected by heterogeneous networks and providing efficient routing between different high-performance network protocols. We are also investigating the integration of *Madeleine II* with our user-level multithread 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* [21].

*Acknowledgements*

We would like to thank Alexandre Denis, Guillaume Mercier and Loïc Prylli for their constant support and help. The ports reported in Section 5.3 have been carried out in the framework of Alexandre's and Guillaume's Master Theses at ENS Lyon.

## References

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

[2] R. Namyst, J.-F. Méhaut, PM2: Parallel Multithreaded Machine. a computing environment for distributed architectures, in: Parallel Computing (ParCo'95), Elsevier, 1995, pp. 279–285.

[3] L. Bougé, J.-F. Méhaut, R. Namyst, Efficient communications in multithreaded runtime systems, in: Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99), Vol. 1586 of Lect. Notes Comp. Science, Springer-Verlag, San Juan, Puerto Rico, 1999, pp. 468–182.

[4] IEEE, Standard for Scalable Coherent Interface (SCI), standard no. 1596 (Aug. 1993).

[5] D. Dunning, G. Regnier, G. McAlpine, et al., The Virtual Interface Architecture, IEEE Micro (1998) 66–75.

[6] E. Lusk, W. Gropp, MPICH working note: The second-generation ADI for the MPICH implementation of MPI, Tech. rep., Argonne National Laboratory (1996).

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

[8] L. Prylli, 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), Vol. 1388 of Lect. Notes in Comp. Science, Held in conjunction with IPPS/SPDP 1998, Springer-Verlag, 1998, pp. 472–485.

[9] R. Russell, P. Hatcher, Efficient kernel support for reliable communication, in: 13th ACM Symposium on Applied Computing, Atlanta, GA, 1998, pp. 541–550.

[10] X. Liu, Performance evaluation of a hardware implementation of VIA, Tech. rep., U. of California in San Diego (1999).

[11] L. Bougé, J.-F. Méhaut, R. Namyst, L. Prylli, Using the VI architecture to build distributed multithreaded runtime systems: a case study, in: Applied Computing 2000, Proc. 2000 ACM Symposium on Applied Computing, Vol. 2, ACM, Como, Italy, 2000, pp. 704–705.

[12] M-VIA, Web document, `http://www.nersc.gov/research/FTG/via/`.

[13] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, Scientific and Engineering Computation, The MIT Press, 1994.

[14] G. Mercier, Ch_mad: un support efficace pour l'hétérogénéité des réseaux dans MPI, Master's thesis report, DEA d'informatique fondamentale, ENS-Lyon, France (Jun. 2000).

[15] J. Worringen, T. Bemmerl, MPICH for SCI-connected clusters, in: SCI Europe '99, Bordeaux, France, 1999, pp. 3–11.

[16] Sca-MPI, Web document, `http://www.scali.com/`.

[17] I. Foster, C. Kesselman, Globus: A metacomputing infrastructure toolkit, Intl. Journal of Super-Computer Applications 11 (2) (1997) 115–128.

[18] 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 (Jun. 2000).

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

[20] T. Ruhl, H. Bal, G. Benson, R. Bhoedjang, K. Langendoen, Experience with a portability layer for implementing parallel programming systems, in: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96), Sunnyvale, CA, 1996, pp. 1477–1488.

[21] V. Danjean, R. Namyst, R. Russell, Linux kernel activations to support multithreading, in: Proc. 18th IASTED International Conference on Applied Informatics (AI 2000), IASTED, Innsbruck, Austria, 2000.