A Portable and Efficient Communication Library for High-Performance Cluster Computing

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

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Portable and Efficient Communication Library for High-Performance Cluster Computing

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

LIP, ENS Lyon

No 3976

Juillet 2000

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--3976--FR+ENG

A Portable and Efficient Communication Library for High-Performance Cluster Computing

Olivier Aumage * Luc Bougé* Alexandre Denis* Jean-François Méhaut*
Guillaume Mercier* Raymond Namyst* Loïc Prylli*
LIP, ENS Lyon

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n°3976 — Juillet 2000 — 18 pages

Abstract: This paper introduces Madeleine II, a new adaptive and portable multi-protocol communication library. 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 BIP and SCI and we present preliminary results about our Nexus/Madeleine II and MPICH/Madeleine II ports.

Key-words: Multiprotocol, multiparadigm, dynamicity, Nexus, MPI.

(Résumé : tsvp)

* Laboratoire de l'Informatique du Parallélisme, École normale supérieure de Lyon, F-69364 Lyon, France. Email : `{Olivier.Aumage}@ens-lyon.fr`

Unit´e de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
T´el´ephone : 04 76 61 52 00 - International: +33 4 76 61 52 00
T´el´ecopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Une biblioth‘eque de communication portable et efficace pour le calcul haute-performance sur grappes de stations

Résumé : Cet article présente Madeleine II, une nouvelle bibliothèque de communication portable et adaptative. Madeleine II est capable de contrôler plusieurs protocoles réseaux (BIP, SISCI, VIA) et plusieurs types de cartes d'interface (Ethernet, Myrinet, SCI) au cours d'une même session. De plus, elle intègre un système de sélection dynamique de la méthode de transfert la plus appropriée pour chaque protocole réseau, d'après divers paramètres tels que la taille des données ou la réactivité requise. Nous présentons les mesures de performance sur des réseaux rapides tels que BIP et SCI ainsi que les premiers résultats de nos adaptations de Nexus et MPICH au-dessus de Madeleine II.

Mots-clé : Multi-protocole, multi-paradigme, dynamicité, NEXUS, MPI.

his paper introduces *Madeleine II*, a new adaptive and portable multi-protocol communication library. *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 BIP and SCI and we present preliminary results about our NEXUS/*Madeleine II* and MPICH/*Madeleine II* ports.

Table des matières

1 Introduction

Due to their ever-growing success in the development of distributed applications on clusters of workstation and SMP machines, today's multithreaded programming environments have to be highly *portable* and *efficient* on a large variety of architectures. For portability reasons, most of these environments are built on top of widespread message-passing communication interfaces such as PVM or MPI. However, the implementation of such environments 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 [9] or PM2 [14], 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 [10] or even VIA [7] 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. The fourth section displays this organization in action while transmitting a message. The implementation of the VIA driver of *Madeleine II* is explained 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 : GLOBUS/NEXUS [9] and MPICH [12]. The last section concludes this paper and introduces on-going and future work.

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. The *channel* object defines a closed world for communication. Communication over a given channel do 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 (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 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.

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

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

TAB. 1: Functional interface of *Madeleine II*.

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 (Table 1). Just like FAST-MESSAGES [15] or NEXUS [9], *Madeleine II* allows applications to incrementally build messages to be transmitted, possibly at multiple software levels. To illustrate this, let us consider a remote procedure call which takes an array of unpredictable size as a parameter. When the request reaches the destination node, the header is examined both by the multithreaded runtime (to extract the name of the function that will be executed by the server thread) and by the user application (to allocate the memory where the array should be stored).

A *Madeleine II* message consists of several pieces of data, located anywhere in user-space. It is initiated with a call to `mad_begin_packing`. Its parameters are the remote node *id* and the channel object to use for the message transmission. Each data block is then appended to the message using `mad_pack`. The 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 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.

The following flags control the reception of user data packets :

`receive_EXPRESS` This flag forces *Madeleine II* to guarantee that the corresponding data are immediately available after the *unpacking* operation. Typically, this flag is mandatory if the data is needed to issue the 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 the corresponding data is transmitted as efficiently as possible.

It should be stressed that this message construction is in fact virtual. *Madeleine II* may well choose at any pack step to send data over the network or to keep data in place and delay transmission or even to copy data into driver-preallocated buffers.

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 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);
```

FIG. 1: Sending and receiving messages with *Madeleine II*.

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 characteritics of each target network.

Madeleine II is organized as two software layers (Fig. 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 of the supported network protocols and is in charge of the management of buffers. It

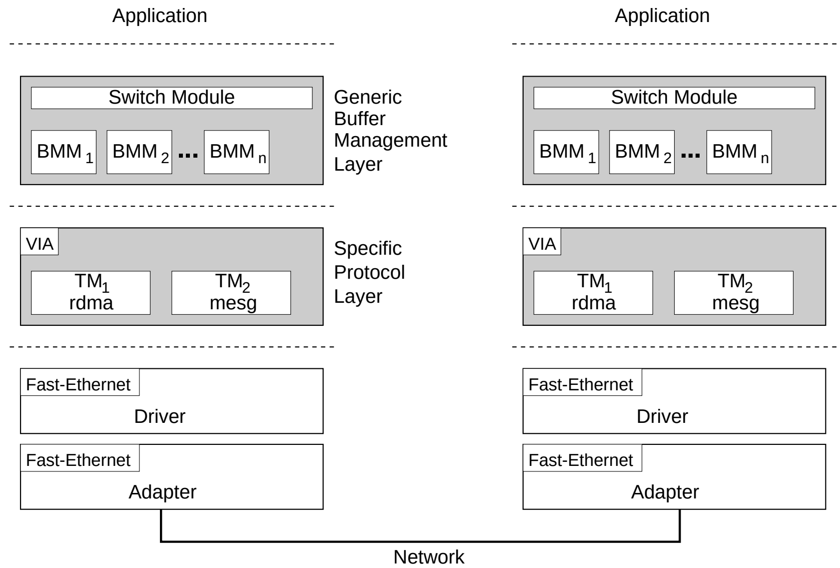

FIG. 2: *Madeleine II*'s modular architecture.

`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

TAB. 2: Functional interface of TMs.

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

3.2 Transfer Management

One of the goal of *Madeleine II* is to support multimodal protocols such as VIA [7] or SCI [10]. 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 like BIP 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).

The Table 2 shows 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, it may also implement protocol-specific buffer allocation routines. This feature is needed for protocols like SBP which provides its 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 or TCP). 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 Card (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 mentionned, 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 will beneficiate from grouped buffer transfers, other may behave worse depedending 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.

Moreover, each BMM may implement a specific aggregation scheme to groups 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 behaviour and send buffers as soon as they are ready. Currently, two settings control which BMM should be selected to work with a given TM :

`buffer_mode` This setting may either be `static` or `dynamic`;

`link_mode` The value of this setting is either `buffer` or `buffer_group`. In the former case, the buffers will be sent as soon as they are ready, while in the latter case, the BMM will attempt to group buffers before transmitting them to the TM. The buffer management layer may dynamically change `buffer` into `buffer_group` if immediate transmission is not allowed (this case occurs if a `send_LATER/receive_EXPRESS` pack has been requested during the message contruction) ;

A third setting (`aggregation_mode`) controls how buffer groups are built on both sides of a connection. It is related to the `receive_EXPRESS` flag. The semantic of this flag makes an obligation for *Madeleine II* to immediately provide the requested piece of data without having any information about the other blocks following this piece of data in the message. Hence, while the sending side groups data regardless of the receive flag, the receiving side must be able to extract this large message as several *smaller* buffer groups, each time a piece of data is request in `receive_EXPRESS` mode (hence the `receive_sub_buffer_group` function in the TMs interface). The `aggregation_mode` precisely indicates if the underlying network may perform efficient sub-buffer-group extraction (TCP for instance), which selects the *asymmetric* mode. If this is not the case, the *symmetric* mode will be selected and the BMM will perform a group flush on the sending side on each `receive_EXPRESS` request.

4 A Message Transmission Step-by-Step

We now displays the *Madeleine II* components running while transmitting an application message. A case study of the implementation of the VIA driver follows these paragraphs.

4.1 Sending

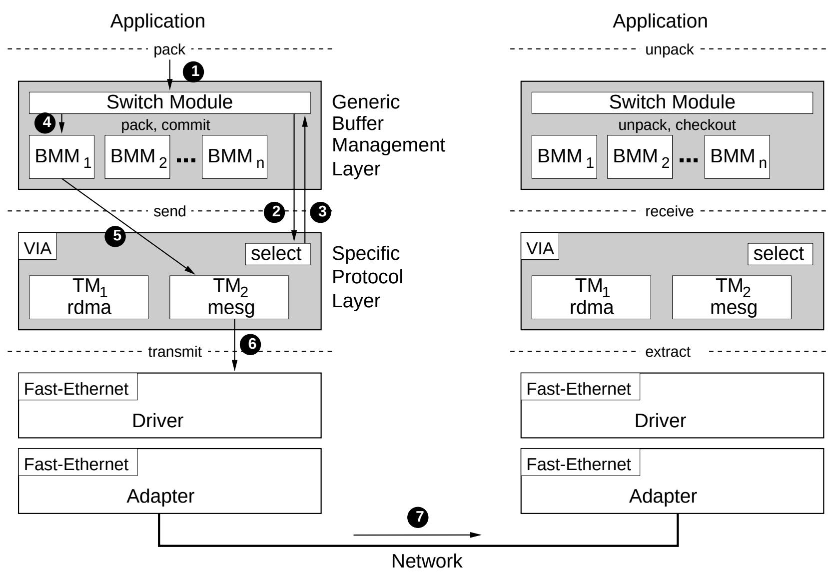

FIG. 3: Conceptual view of the data path through *Madeleine II*'s internal modules.

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

Using this `connection` object, the application can start packing user data into packets by calling `pack(connection, ptr, len, s_mode, r_mode)`. Entering the Generic Buffer Management Layer, the packet is examined by the *Switch Module* (Step 1 on Fig. 3). It queries the Specific Protocol Layer (Step 2) for the best suited *Transmission Module*, given the length and the send/receive mode combination. The selected TM (Step 3) determines the optimal *Buffer Management Module* to use (Step 4). Finally, the Switch Module forwards the packet to the selected BMM. Depending on the BMM, the packet may be handled as is (and considered as a buffer), or copied into a new buffer, possibly provided by the TM. Depending on its aggregation scheme, the BMM either immediately sends the buffer to the TM or delays this operation for a later time. The buffer is eventually sent to the TM (Step 5). The TM immediately processes 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 (*commit* on Fig. 3) to ensure that any remaining delayed packet has been shipped to the network. A general *commit* operation is also performed by the `end_packing(connection)` call to ensure that no delayed packet remains waiting in the BMM.

4.2 Receiving

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

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

4.3 Case Study : VIA

The functional versatility of VIA [7] 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 traditionnal 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 ([11], [4]), and one can consider two alternative solutions : one may either choose to manage a pool of preregistered 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 tradeof 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 get 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 :

– Pregistered 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 frontiers comes from our *Madeleine II* VIA driver over the M-VIA [1]/Fast-Ethernet implementation. The 5 kB limit was determined experimentally (on a LINUX/PII-450 cluster) and corresponds to the minimal block size for which dynamic registration becomes cheaper than an extra copy. As for the 32 kB limit, it is equal to the MTU (Maximum Transfer Unit) of M-VIA using our hardware components and is the size limit for messages to be sent with a unique transfer. Hence, transfering 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 allows a better implementation. The receiver side register the whole buffer at once and acknowledge the sender. Then the sender emits enough RDMA-Write transactions to send the buffer without requiring any operation from the receiver. While only one receiver ack is used with this method, the message-pasing way would much more expensive, even with a credit based flow-control scheme.

Going back to VIA features, it is possible for instance 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 most efficient to immediately transfer buffers as soon as they get full (just set `link_mode` as `buffer`) : this enhances pipelining and overlaps the additional copy involved.

Let us now have a look to the performance achieved by *Madeleine II*.

5 Implementation and Performance

5.1 Testing Environment

The following performance results are obtained using a cluster of dual Intel Pentium II 450 MHz PC nodes with 128 MB of RAM running LINUX (Kernel 2.1.130 for VIA, and Kernel 2.2.13 otherwise). The cluster interconnection networks are 100 Mbit/s Fast Ethernet for TCP and VIA, Dolphin SCI for SISCI and Myrinet for BIP.

5.2 *Madeleine II* Drivers

5.2.1 VIA

Figure 4 illustrates the interest of adaptive multi-modal protocol support with the VIA protocol use the M-VIA 0.9.2 implementation from the NERSC (National Energy Research Scientific Computing Center, Lawrence Berkeley Natl Labs).

We can see that both the dynamic user buffer registration method and the pre-registered buffer pool method are not optimal for the whole packet size range. In 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.

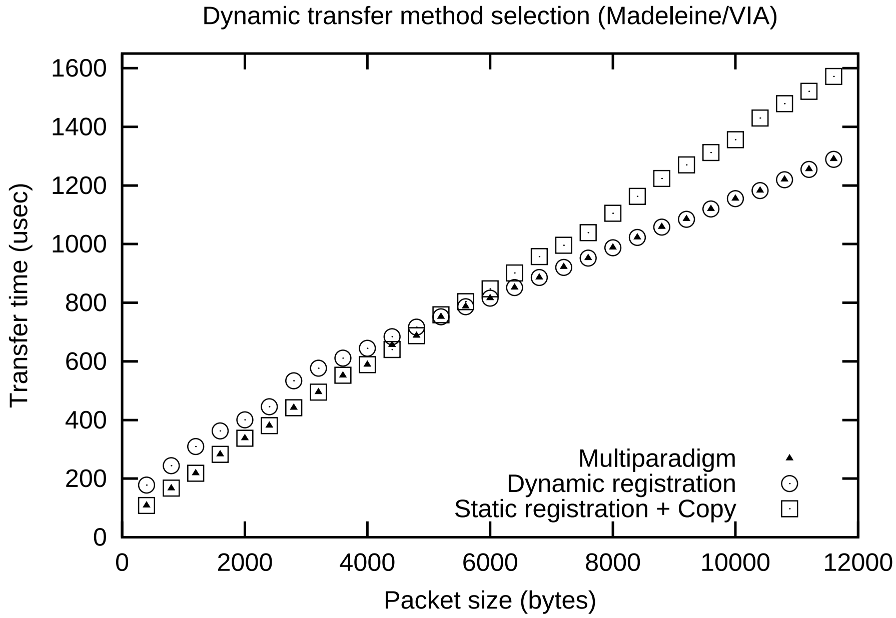

FIG. 4: Adaptive multi-modal protocol support with VIA

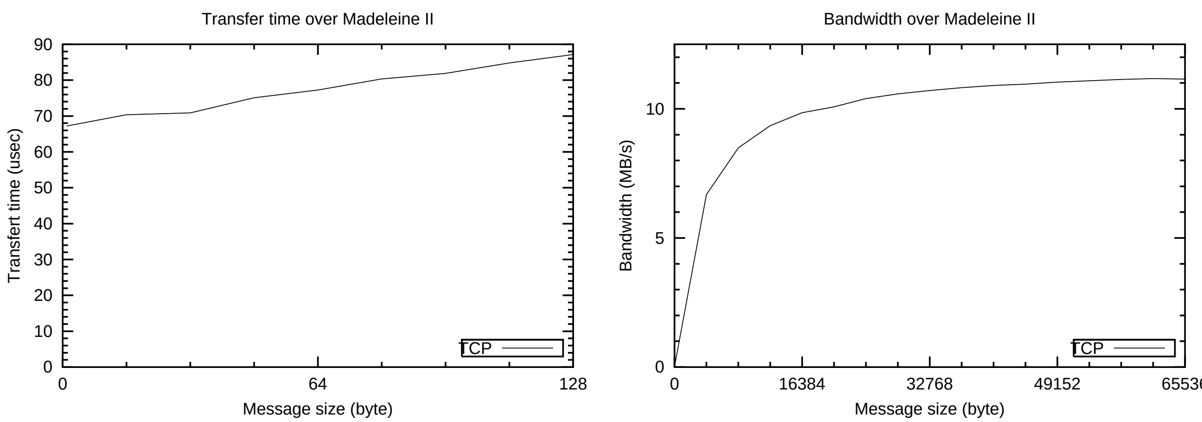

FIG. 5: Latency and bandwidth over TCP/Fast-Ethernet

5.2.2 TCP

The results of tests run on the TCP/IP protocol using standard UNIX sockets are plotted on Figure 5. *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 μs with a uniprocessor-compiled LINUX kernel and around 110 μs with an SMP LINUX kernel.

5.2.3 SISCI

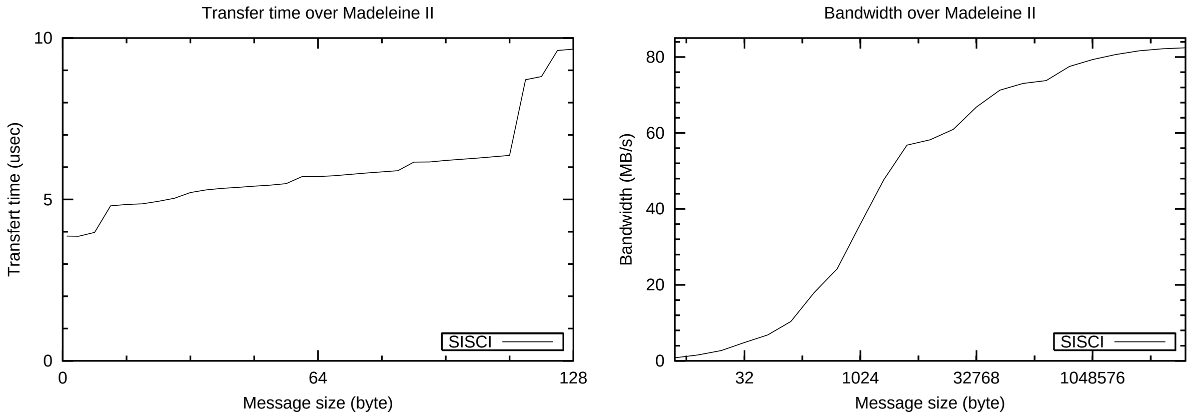

FIG. 6: Latency and bandwidth over SISCI/SCI

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 μs), thanks to our higly optimized short message TM (see implementation details below).

The bandwidth is very good too because of 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 perforance of the SCI DMA : we have not been able to get more than 35MB/s as of now ! Three transmission modules are currently implemented. Indeed, the *regular* mode uses an additional TM specifically optimized for short message transfer.

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 experimentally !).

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 :

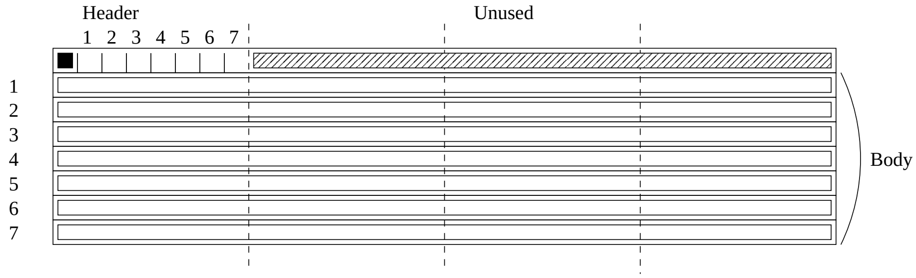

FIG. 7: Area layout for optimized transfer

– The first bit is always set to 1 (this bit simulate 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. 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.

5.2.4 BIP

BIP (Basic Interface for Parallelism) is a low-level communication interface specifically designed for the Myrinet network protocol [16]. The main advantage 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. Short messages are being stored into internal static buffers (preallocated by BIP) on the receiving side. Long messages however are not copied during their transmission. In this latter case, a strict synchronization is necessary between the sender and the receiver : the receiver must acknowledge the sender that it is ready to receive before a message is actually transmitted.

The BIP driver of *Madeleine II* handles both transmission modes. The *short messages* TM uses a credit based flow control algorithm to make sure that each message may be stored into a static buffer. The *long message* TM implements the receiver-acknowledgement synchronisation scheme. This *Madeleine II* driver also gives nice results with a minimal latency of 7 μs and a bandwidth of 122 MB/s (Figure 8). Theses results are very close to the raw BIP results : 5 μs minimal latency and 126 MB/s maximal bandwidth.

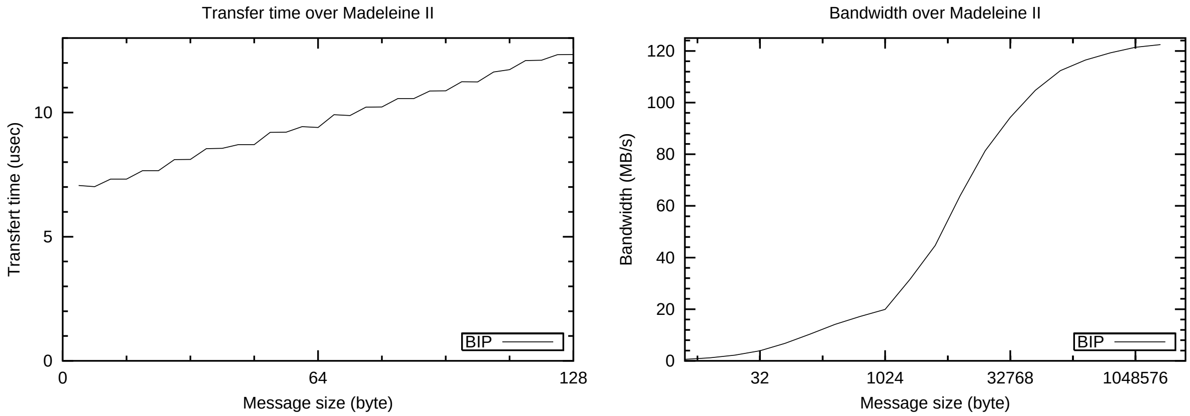

FIG. 8: Latency and bandwidth over BIP/Myrinet

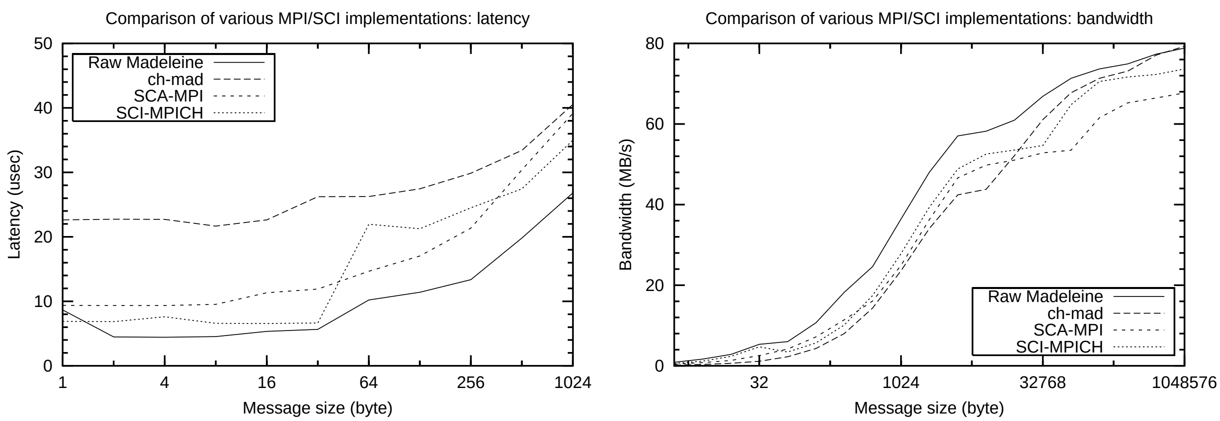

FIG. 9: Comparison of various MPI implementation over SCI

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

We now present two implementations of high-level communication libraries — namely MPICH [12, 13] and GLOBUS/NEXUS [9, 6] — over *Madeleine II*.

5.3.1 MPICH/*Madeleine II*

Madeleine II has been integrated into MPICH as a *ch-mad* module. Our goal was to let MPICH beneficiate of the multi-protocol features of *Madeleine II*. First performance measurements show very interesting results. Figure 9 compares MPICH/*Madeleine II*/SISCI to two other implementations of MPI over SCI, namely SCI-MPICH [17] and the commercial version ScaMPI [2]. The performance curves of *Madeleine II* over SISCI (without MPICH) are plotted too in order to provide an idea of the current overhead of our MPI/*Madeleine II* implementation.

Though latency compares defavorably to direct implementations of MPI over SCI, we can see that things are much different as far as bandwidth is concerned. Our *ch-mad* module provides the best results for messages of 32 kB and above. Moreover, this module is able to use most of the bandwidth provided by *Madeleine II* for large messages.

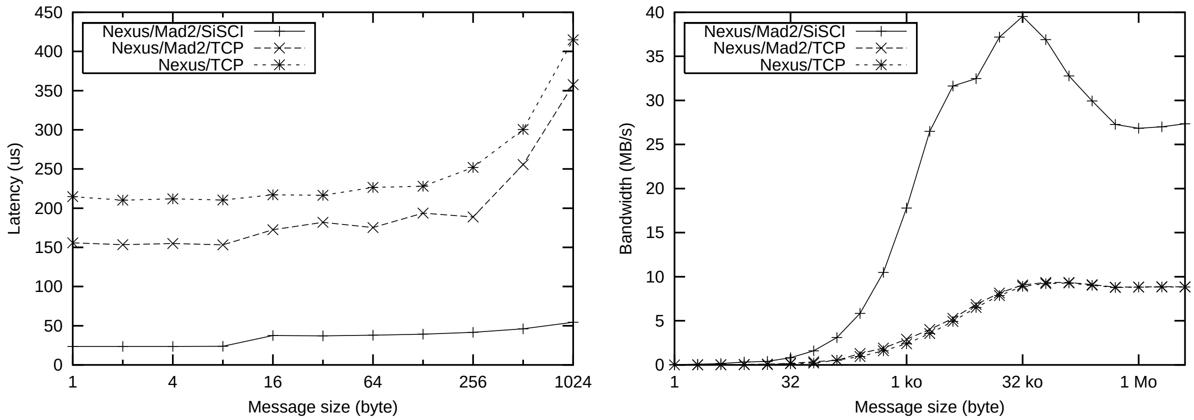

FIG. 10: NEXUS/*Madeleine II* performance

5.3.2 Nexus/*Madeleine II*

While NEXUS is very much 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 application communication at the cluster scale. In contrast, *Madeleine II* was specifically designed to provide applications with highly efficient access to cluster network resources. Hence, it was interesting to investigate merging these two communication libraries in order to get the best of both worlds.

The problem is the different models adopted by these communication interfaces : NEXUS 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 μs) and offers a more interesting solution as far as cluster computing is concerned.

NEXUS features multiprotocol support [8] 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.

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 new 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 : NEXUS/*Madeleine II* and MPICH/*Madeleine II*. Here again, results are highly encouraging. MPICH/*Madeleine II* even outperforms the current best implementations of MPI over SCI as far as bandwidth is concerned.

We are now actively working on various *Madeleine II* improvements, namely having *Madeleine II* running 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* [5].

Références

[1] MVIA. `http://www.nersc.gov/research/FTG/via/`.

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

[3] Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Efficient communications in multithreaded runtime systems. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes Comp. Science*, pages 468–182, San Juan, Puerto Rico, April 1999. Springer-Verlag.

[4] Luc Bougé, Jean-François Méhaut, Raymond Namyst, and Loïc 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*, volume 2, pages 704–705, Como, Italy, March 2000. ACM.

[5] Vincent Danjean, Raymond Namyst, and Robert Russell. Linux kernel activations to support multithreading. In *Proc. 18th IASTED International Conference on Applied Informatics (AI 2000)*, Innsbruck, Austria, February 2000. IASTED.

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

[7] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne-Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–75, Mar-Apr 1998.

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

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

[10] IEEE. *Standard for Scalable Coherent Interface (SCI)*, August 1993. Standard no. 1596.

[11] Xin Liu. Performance evaluation of a hardware implementation of via. Technical report, University of California San Diego, 1999.

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

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

[14] 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, September 1995.

[15] 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.

[16] L. Prylli and B. Tourancheau. BIP : A New Protocol designed for High-Performance Networking on Myrinet. In *Proc. of PC-NOW IPPS-SPDP98*, Orlando, USA, March 1998.

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

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399