NewMadeleine: An Efficient Support for High-Performance Networks in MPICH2

Guillaume Mercier, François Trahay, Darius Buntinas, Elisabeth Brunet

NewMadeleine: An Efficient Support for High-Performance Networks in MPICH2

Guillaume Mercier, François Trahay, Elisabeth Brunet
Bordeaux University
LaBRI - INRIA
F-33405 Talence, France
{mercier,trahay,brunet}@labri.fr

Darius Buntinas
Argonne National Laboratory
Mathematics and Computer Science Division
Argonne, IL 60439, USA
buntinas@mcs.anl.gov

Abstract

This paper describes how the NewMadeleine communication library has been integrated within the MPICH2 MPI implementation and the benefits brought. NewMadeleine is integrated as a Nemesis network module but the upper layers and in particular the CH3 layer has been modified. By doing so, we allow NewMadeleine to fully deliver its performance to an MPI application. NewMadeleine features sophisticated strategies for sending messages and natively supports multirail network configurations, even heterogeneous ones. It also uses a software element called PIOMan that uses multithreading in order to enhance reactivity and create more efficient progress engines. We show various results that prove that NewMadeleine is indeed well suited as a low-level communication library for building MPI implementations.

1 Introduction

In recent years, the landscape of parallel computing has undergone dramatic changes. Massively multicore CPUs, as well as highly hierarchical clusters based on NUMA nodes are now emerging on the market. Programming such architectures is becoming increasingly challenging since current models become more and more questioned. However, it seems difficult to propose a brand new model that would replace existing standards. Despite their lack of convenience in this context, OpenMP and MPI are tools that are and will still be used for a long time. It is therefore crucial for implementations to be able to take advantage as much as possible of such hardware's evolutions.

One of the major difficulties is that current MPI implementations have to take into account multiple hardware features while maintaining a strict compliance to the actual standard. Implementations now have to take into consideration the increasing number of CPUs and cores available in a computing node. They will also need to take into consideration the memory hierarchy as well as the NUMA factor when accessing data. As far as the network is concerned, exploiting multiple and possibly heterogeneous interconnects raises issues: How can an MPI implementation efficiently utilize all NIC resources despite their different natures? How can we avoid contention on the NICs in the case where all the MPI processes on a given node are sending messages? Could some cores be dedicated to optimize communication progress instead of executing regular application code? Building a complete MPI stack is a complex task and such sophisticated optimizations are often overlooked.

We believe that specialized software tailored to efficiently exploit complex and hierarchical architectures is one of the keys to an efficient MPI implementation in such environments. Indeed a low-level runtime system upon which the MPI stack is ported offers both portability and performance to the application using MPI. All optimization mechanisms developed in such a low-level system can benefit the upper, more generic layers of the resulting MPI implementation.

The PM^2 software suite [11] developed in the Runtime team is able to provide such services. Several software elements compose this suite. The main three are: a high-performance communication library called NewMadeleine, a sophisticated user-level thread package called Marcel [10] and a generic I/O manager called PIOMan [15]. Those elements are able to interact with each other, making it easier, for instance, to mix communication with multithreading. All elements feature numerous characteristics that make them extremely suitable as a runtime system for higher-level programming environments and standards implementations. In this paper, we will show that overall MPICH2 performance is improved due to the PM^2 suite.

The structure of this paper is as follows. In Section 2, we describe the various software that we used to create our MPI implementation. Section 3 presents the details of the integration of the various software elements into an efficient communication device for MPICH2. In Section 4 we present the various performance evaluations we carried out. In Section 5 we conclude this paper and discuss future work.

2 Software architecture of MPICH2-NewMadeleine

In this section, we describe the various pieces that constitute our MPI implementation. We first give an overview of the MPICH2 software stack. We then explain how the PM^2 software suite is organized and which elements are used to create our implementation. In particular, we emphasize the advantages provided by each piece of software selected.

2.1 An overview of MPICH2-NewMadeleine software stack

MPICH2 is an MPI-2 compliant implementation. It features a layered structure, as shown by Figure 1. Several implementation choices are available for porting a new communication library or protocol into this stack. A first solution is to implement a new ADI3 *device*. Building a whole device is likely to yield the best performance, but at a high cost in terms of complexity of development because the number of routines to implement is the largest. This solution is particularly recommended when the underlying interface is an elaborate one, closely matching what the MPI standard itself might propose. For instance, the MPICH2 implementation on top of the Myrinet MX [2] interface follows this philosophy of development and so does the MPICH2 port over Elan networks [9]. Our target as a low-level communication layer is PM^2's communication subsystem, that is, the NewMadeleine library. As we shall see later, its interface is rather simple and the number of routines proposed is small. In this regard, implementing a new device would not be ideal since we would have to reimplement most of the things that have been previously developed in the current CH3 device.

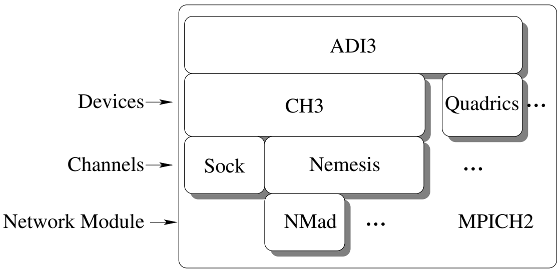


Figure 1. Software layers of MPICH2

Another approach would be to develop a new CH3 *channel*. Since we target nodes that will eventually feature a large number of cores, we need an efficient communication mechanism to move data between processes sharing the same physical node. However, NewMadeleine focuses on network communication and the issue of intra-node communication is left unaddressed. Several CH3 channels in MPICH2 already offer a support for intra-node communication: *shm* (the shared memory channel), *ssm* (the socket and shared memory channel), *sshm* (the scalable shared memory channel) and the *Nemesis* channel.

2.1.1 The Nemesis communication channel

The Nemesis channel [5] is the default channel in MPICH2. It relies on shared memory for intra-node communication and uses network for all inter-node communication. The Nemesis channel currently yields the best performance for intra-node communication among all available channels in MPICH2. It also compares favourably to other MPI implementations [5]. The Nemesis channel uses shared-memory message queues of fixed-size message cells for intra-node communication. These queues are lock-free and allow multiple processes to enqueue cells concurrently. Each process owns one free queue and one receive queue. The free queue holds free cells which the process dequeues and fills with a message (or message fragment when the message is larger than a single cell). To receive a message, a process actively polls on the receive queue until a sender process enqueues a new cell. This approach is scalable since the receiver needs only to poll a single receive queue. This also allows handling the `MPI_ANY_SOURCE` case efficiently. Another interesting feature of Nemesis is that network communication can be added through the use of network modules. As a whole, the Nemesis channel fulfills our requirements of excellent performance for intra-node communication and the ease of porting a new communication layer.

2.1.2 The Nemesis network modules

Nemesis already supports numerous networking technologies thanks to dedicated *network modules*. Currently supported networks are Myrinet (with both GM and MX modules), QsNet, Infiniband, TCP and PSM. A network module implements a relatively small set of routines, especially when compared to the channel or device approaches. Basically the four following routines are required to implement a module: `net_module_init`, `net_module_send`, `net_module_poll` and `net_module_finalize`. There is no `net_module_recv` routine since the `net_module_poll` routine is called by the low-level progress engine in Nemesis and is actually responsible to retrieve all incoming messages from the network.

2.1.3 Current limitations in the modules' design

Implementing a network module implies to use the Nemesis queue system. However, in some cases, unnecessary copies are performed, in and from the queue cells. This happens in the case of short and medium-sized messages and even if memory copies for such sizes are efficient, a performance penalty occurs. Also, a library might implement its own set of protocols such as the *eager* or more important the *rendezvous* protocol. By using the low-level Nemesis module approach the library can't use its own set and has to rely on the CH3 protocols instead. In the case of a large message, the CH3 *rendezvous* protocol starts with a handshake sequence composed of a RTS message from the sending side, followed by a CTS message from the receiving side. Then the data itself is sent over as shown by Figure 2. Each time, the underlying communication library is used to send both messages. In the case of the data message (composed of a single CH3 message) the library might use internally its own *rendezvous* protocol to send only the data. In such case, an additional handshake is employed which could be avoided, thus improving performance. We want to use Nemesis but we would like to shortcut it for small network messages. We also want to avoid CH3 protocols when necessary. The issue is then to be able to expose the underlying communication library's interface without sacrificing the portability of the code located in higher level layers of the stack.

2.2 The NewMadeleine multithreaded communication library

NewMadeleine [3] is the communication library of the PM^2 suite. Most of communication libraries focus only on sheer latency and bandwidth. Usually data is sent over the network as soon as it is passed to the library by the upper layers. NewMadeleine proceeds differently: it works with the networks's activity. When a network is already fulfilled with communication requests, NewMadeleine keeps a window of packets to send. Thus, when a network becomes idle, it has the possibility to apply optimizations on the accumulated communication requests before submitting them to the network. This uncoupled network request submission permits a more global vision of communication flows and the various strategies can be applied over the overal set of messages sharing the same destination. Such strategies may use, for instance, reordering techniques or messages aggregation. Also, NewMadeleine is able to support several NICs at the same time and even several (possibly different) protocols, being natively multirail-enabled. In this case, NewMadeleine proposes several strategies to efficiently take advantage of the multiple network resources. A network sampling mechanism is used to compute an adaptive split ratio tailored to fit each available networks' abilities [4]. Another interesting feature of NewMadeleine lies in its ability to work in a multithreaded environment. Indeed NewMadeleine is thread-safe to boot. It also can rely on the PIOMan I/O manager to handle communication progress in a sophisticated manner.

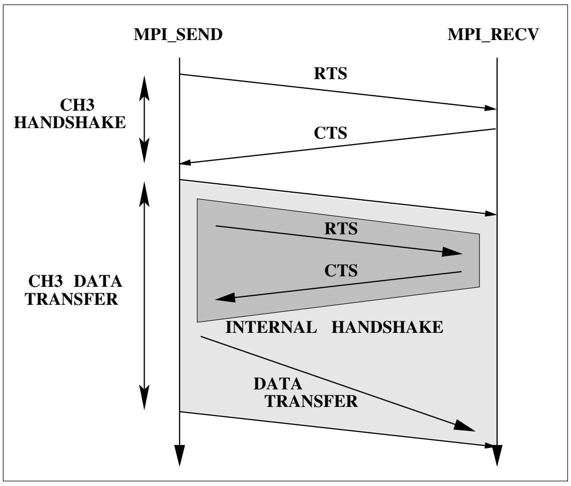


Figure 2. Nested handshakes in *rendezvous* case

2.2.1 The NewMadeleine interface

NewMadeleine supports a wide range of current high-performance networking technologies thanks to its Infiniband, Myrinet and Quadrics ports. These low-level interfaces are accessible through NewMadeleine's generic and message-passing oriented interface. Indeed, some of NewMadeleine's routines do closely match MPI's routines. For instance, the following functions are used to send and receive data:

```
nm_sr_isend( destination, tag, *buffer,
             size, *nmad_request );
nm_sr_irecv( destination, tag, *buffer,
             size, *nmad_request );
```

NewMadeleine's requests are opaque objects allocated internally each time a send or receive operation is submitted. Once this object is created, the user can query NewMadeleine in order to get information about a request's completion. NewMadeleine, however, does not yet support the cancellation of a posted request. Any request that has been previously posted has to be completed at some point during the application's execution.

2.2.2 The PIOMan I/O events manager

PIOMan [15] is the I/O manager of the PM^2 software suite. It aims at providing communication libraries with an event detection service that guaranties a predefined level of reactivity to events. PIOMan works closely with a user-level thread scheduler called Marcel [10] and thus is able to balance the processing of event detections across the whole machine. The collaboration with the thread scheduler allows PIOMan to precisely know the workload of each CPU used within the process. Therefore, the most appropriate detection method (polling or interrupt-based blocking call) is called depending on the context (number of computing threads, available CPUs, etc.) Marcel also schedules PIOMan on some triggers (CPU idleness, context switches, timer interrupts, etc.) so as to ensure a fast detection of communication events.

In a multithreaded context, NewMadeleine delegates communication flow progress to PIOMan. NewMadeleine can thus concentrate its efforts on optimizing communication flows whereas PIOMan handles the issues raised by the use of computing threads. PIOMan is responsible for detecting communication completions, making communication progress in the background and submitting new packets to NICs. The submission of data is thus performed by idle cores when it is possible, reducing the application's threads' workload [6] and allowing the overlap of communication with computation.

2.2.3 Benefits brought by NewMadeleine and PIOMan

The integration of the PM^2 software suite within the MPICH2 software stack provides all of the benefits of PM^2 to the MPICH2 library for free. Thanks to NewMadeleine, aggressive optimizations on communication flows can be applied to reduce the impact of inter-node communication on applications. NewMadeleine's multirail feature allows MPICH2 to efficiently support modern multirail clusters exploiting the high bandwidth systems.

NewMadeleine's multithreaded subsystem also allows MPICH2 to exploit multicore architectures by offloading eager messages submission or by using idle cores to poll networks. The progress of communication flows in the background and the reactivity to both network and shared-memory events is provided by PIOMan.

3 Towards an efficient communication core for MPICH2

In this section, we give details about our NewMadeleine network module implementation. We also describe how we manage the case of `MPI_ANY_SOURCE` and how this impacts performance. Then we explain how the PIOMan software is integrated with the Nemesis communication subsystem.

3.1 NewMadeleine network module internals

In porting MPICH2 over NewMadeleine, we implemented a NewMadeleine network module. Then in order to allow MPICH2 to take full advantage of the features of the NewMadeleine communication library, we modified the CH3 layer to bypass the Nemesis layer and directly call NewMadeleine. These modifications are not NewMadeleine specific and could be used to support other communication libraries that perform tag matching. Below we describe these modifications as well as modifications to NewMadeleine's interface.

3.1.1 Accessing NewMadeleine's interface directly from CH3

In order to avoid unnecessary handshakes during the *rendezvous* protocol, we must either expose the low-level communication library interface at the CH3 level or force NewMadeleine to use only certain types of protocols. However, restricting NewMadeleine in this way, e.g., not utilizing NewMadeleine's ability to perform tag matching, results in suboptimal performance. The solution is to bypass parts of CH3 and Nemesis and allow CH3 to directly call NewMadeleine functions. In this way, intra-node messages are still handled by Nemesis using the shared-memory queues, while inter-node messages are handled directly from the CH3 layer by NewMadeleine.

A mechanism is needed to associate an MPI communication operation with the corresponding NewMadeleine communication operation. When a NewMadeleine communication is completed we would like to be able to mark the corresponding MPI communication as completed too. In the MPICH2 implementation, each communication is managed with a *request* object. Such objects are queued on MPICH2's internal *posted receive queue* and *unexpected queue*. This pair of queues forms the core of the message passing management in MPICH2. NewMadeleine has a similar request object to keep track of its pending communication operations. So to associate the two types of requests, we added a new field to the Nemesis-specific portion of the MPICH2 request which points to the corresponding NewMadeleine request.

3.1.2 Sending operations implementation

As mentioned previously, NewMadeleine's sending and receiving routines have prototypes that closely match their CH3 counterparts (e.g `MPID_Send()`, `MPID_Recv()`). So, we chose to directly call the NewMadeleine functions in the corresponding CH3 routines. However, the implementation differs significantly between the sending and receiving side. On the send side, function pointers were added to MPICH2's per-connection *virtual connection* (VC) structure to allow the various CH3 send functions to be overridden on a per-destination basis. In this way, a call to `MPID_Send()` will result in a call directly to the NewMadeleine send function only when sending to a process on a different node. Because Nemesis is being bypassed when sending to processes on remote nodes, the Nemesis layer will only manage send queues for messages sent to processes on the same node.

3.1.3 Managing receive queues

Because a process can receive messages either from processes on the same node through Nemesis, or from remote nodes through NewMadeleine, receiving messages are handled differently. In order to take advantage of NewMadeleine's tag matching capability, Nemesis does not manage messages received by NewMadeleine. Instead, NewMadeleine maintains its own receive queues, performs tag matching internally, and delivers messages directly to the user buffers. To accomplish this, when the application calls `MPI_Recv()` and a receive request is posted in the CH3 posted receive queue, a corresponding receive operation is posted to NewMadeleine. A pointer to the CH3 request is included in the NewMadeleine request in order to associate the NewMadeleine request is associated with the CH3 request. The NewMadeleine network module periodically polls a new NewMadeleine function which returns a pointer to the CH3 request of any received message. Using this function, once a message is matched and received by NewMadeleine, the NewMadeleine network module can directly mark the corresponding CH3 request as complete, and allow the MPICH2 progress engine to handle the completed request as usual.

3.2 Management of multiple sources on reception

3.2.1 Polling on multiple sources: issues

There is another side effect of bypassing Nemesis for all network communication. Indeed, the handling of any source communication gets more complicated. We don't use Nemesis' receive queue that centralizes all incoming messages from intra and inter-node sources in the case of network modules. In our case, we have intra-node messages in the Nemesis receive queue and inter-node messages are handled internally by NewMadeleine. In both cases, the corresponding ADI requests are stored in MPICH2's Unexpected and Posted Receive Queues. However, we also create specific NewMadeleine requests that correspond to ADI3 requests. The issue is to keep consistency between NewMadeleine and ADI requests. In the case of a regular request (i.e using a well-defined source for the matching), as soon as the NewMadeleine request is completed, we also handle the ADI request and we remove it from the Posted Receive Queue. This is an easy task because for have an one-to-one relationship between these requests. In the case of an ADI request using any source for matching, a solution would be to create multiple NewMadeleine requests, one for each possible incoming source. When one of the NewMadeleine requests that matches its ADI counterpart is completed, then we would cancel the remaining one. Also, if a intra-node communication would match the request, we would cancel all the NewMadeleine requests.

3.2.2 Implementation with requests lists

The major issue here stems from the fact that Newmadeleine does not support the cancellation of requests. Moreover a posted NewMadeleine request has to be eventually matched. So, whenever an any source message appears, we can't create NewMadeleine requests for all possible incoming source. So we created a system where a NewMadeleine request is dynamically created when a message is received that could match the current ADI any source request. We keep track of such pending any source requests in a separate list. Each time a new any source request is posted, we check the list and create a new entry if the MPI message tag hasn't already been used. Then, every time Nemesis polls for incoming messages, we *probe* NewMadeleine to check if a corresponding message has arrived. In this case, the NewMadeleine request is posted. Since the message has arrived and sits in NewMadeleine's buffers, it will be completed shortly after its creation and the ADI request will be marked as completed too. In the meantime, other ADI requests using NewMadeleine and the same MPI tag might have been created. In order to ensure message ordering, they are enqueued in the list of pending any sources and dequeued when the any source NewMadeleine entry is removed. If an other any source request (using the same tag) is present in the sublist, it replaces the former request as list head. But there is another case where an intra-node message might very well match the ADI request also. In that case the entry in NewMadeleine's pending any source queue is sim-

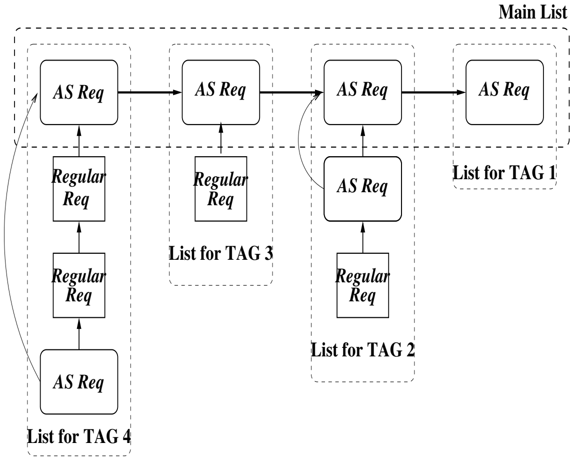


Figure 3. Any source Management Lists in the NewMadeleine Module

ply removed and all requests that might have been posted after are created.

3.3 PIOMan's integration within Nemesis

In this section, we explain the consequence of Nemesis delegating its polling operations to PIOMan and how Nemesis is affected by this change.

3.3.1 Creating a global polling authority

In order to fairly make progress both intra-node and inter-node communication, it is necessary to centralize the detection of communication completions. This permits to have a global view of pending communication and avoids to privilege one type at expense of another. The progression of communication within NewMadeleine being already centralized by PIOMan, the detection of shared memory communication completion has been deferred to PIOMan. This way, the whole software stack benefits from a global view of both intra-node and inter-node communication flows and from the multithreaded polling mechanism provided by PIOMan.

3.3.2 Modification to Nemesis' polling schemes

To make PIOMan handle the detection of completed shared memory requests, a mailbox mechanism has been added to the shared memory subsystem: when Nemesis needs to poll for an incoming message in shared memory, it notifies PIOMan and specifies the address of a counter that is incremented when the message is sent to the other side. PIOMan can thus check the state of shared memory as it checks the state of networks.

In order to fully benefit from PIOMan's progression service, busy-waiting loops have been replaced by blocking primitives that can be viewed as semaphores. So, whenever an application thread waits for a message completion – using the `MPI_Wait` function for instance – it is blocked on a semaphore and another thread can be scheduled to make the application's computation progress. The detection of the message completion is performed in the background by PIOMan during context switches, timer interrupts or when a CPU is idle. When a communication completion is detected, PIOMan unblocks the corresponding thread that can be scheduled.

The use of semaphore-like primitives instead of actively polling will permit interesting features when MPICH2 will provide a `MPI_THREAD_MULTIPLE` thread-safety level: instead of concurrently polling when several threads invoke `MPI_Wait` – which would boil down to wasting CPU time – these threads would relinquish the CPU in order to allow other threads to compute.

4 Performance evaluation

In this Section, we assess the performance of our MPICH2-NewMadeleine implementation. First, we show the basic performance obtained with simple point-to-point experiments. We produce latency and bandwidth figures in the Infiniband high-performance network case as well as in a multirail case where both Infiniband and Myrinet are used. Then we show the influence of the PIOMan software element in terms of both overhead and overlapping improvement. The second part of this performance evaluation is dedicated to application kernels and we consider the NAS parallel benchmarks to test the behaviour of our implementation in a more realistic setting. The other MPI implementations considered for this evaluation are MVAPICH2 1.0.3[8] and Open MPI 1.2.7[13].

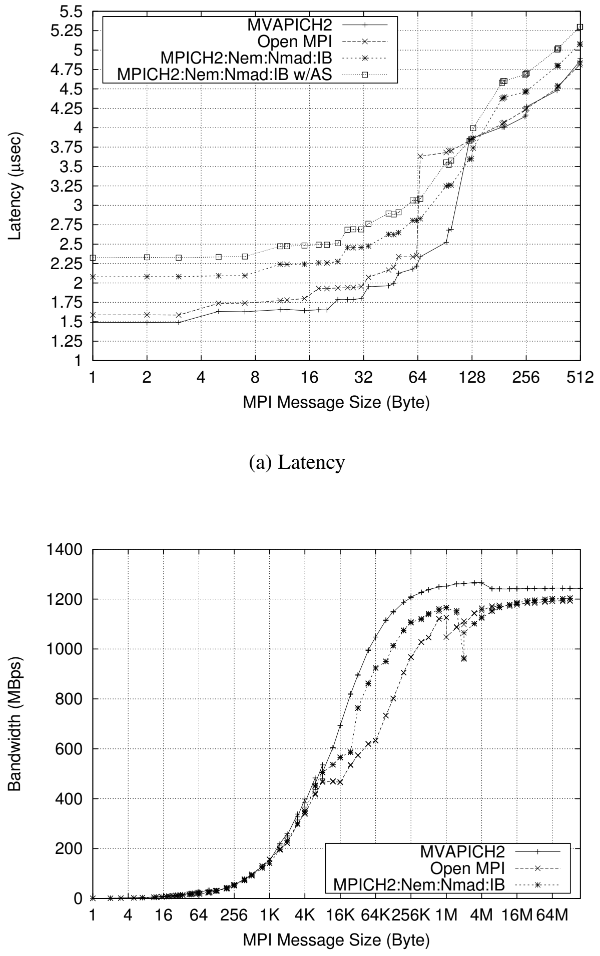
4.1 Point-to-point evaluations

For this series of experiments, we use two boxes composed each of two quadcore 3.16 GHz Intel Xeon CPUs featuring 4 GB of memory. The OS is Linux 2.6.26 and each box is equipped with one Myrinet Myri-10G NIC (with the MX interface) and one ConnectX Infiniband NIC (with the Verbs interface). In the remaining of the Section we consider one Megabyte (1 MB) as 1024 × 1024 bytes.

4.1.1 Latency and bandwidth evaluations

We now present the results obtained with the Netpipe [14] test program. Since NewMadeleine is a generic communication libray, we can easily benchmark several high-performance networks without writing any specific code.

Indeed, we made experiments with Infiniband NICs and with an heterogeneous multirail featuring both Myrinet and Infiniband networks.



(a) Latency

(b) Bandwidth

Figure 4. Infiniband Performance Comparisons

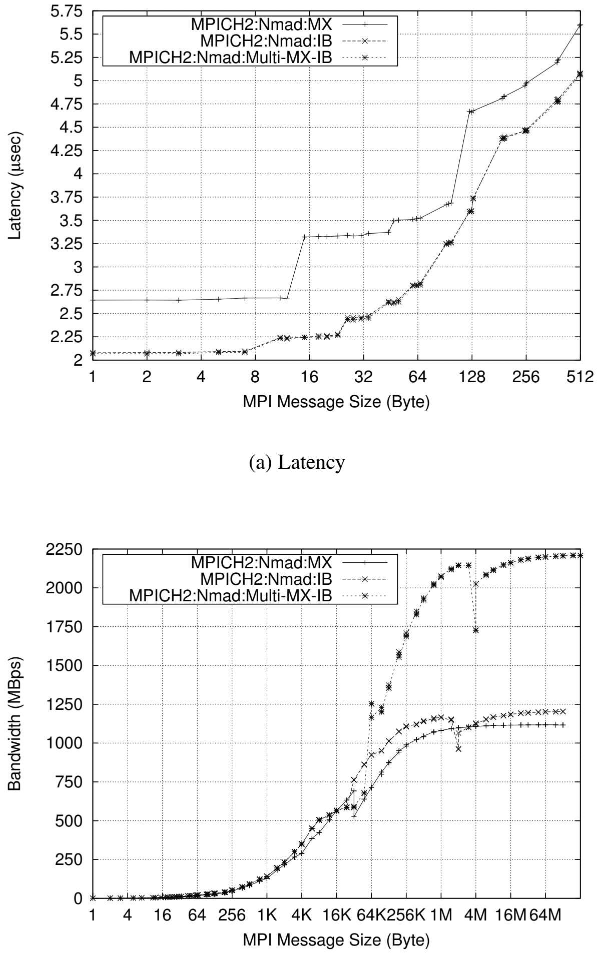
Figure 4 shows the performance comparisons between several MPI implementations using Infiniband. We tested MVAPICH2 which is derived from MPICH2, as well as Open MPI 1.2.7 with its Infiniband support. Actually, Open MPI uses the *openib* BTL but also takes advantage of an Infiniband-tailored MTL as well. This explains why MVAPICH and Open MPI latencies are almost identical (1.5μs vs. 1.6μs) and very close to the hardware's raw performance (1.2μs, not shown on the graphs). Because MPICH2-NewMadeleine relies on its generic layer, the latency is higher (2.1μs) with an overhead of 300 nanoseconds when compared to NewMadeleine (1.8μs, not shown on the graph). As far as bandwidth is concerned, MVAPICH2 outperforms all other solutions but it is interesting to note that MPICH2-NewMadeleine is able to reach a higher bandwidth than Open MPI for medium-sized messages. This performance is very good since NewMadeleine does not use any caching mechanism for large messages and registers dynamically and on-the-fly the needed memory to handle the transfers. As expected, MPICH2-NewMadeleine's latency is affected by a 300 nanoseconds gap when `MPI_ANY_SOURCE` is used. This gap remains constant while message size grows and bandwidth for large messages remains unaffected (the curve on the graph is similar to the regular known-source case and thus not shown). We also did experiments with Myrinet and the same behaviour is to be observed.

Figure 5 exposes the performance level of our NewMadeleine module in the case of an heterogeneous multirail system featuring one Myri-10G NIC and one Infiniband 10G NIC each. We compare the multirail performance of MPICH2-NewMadeleine with the performance obtained in the Myrinet-only and in the Infiniband-only cases. Actually the curve corresponding to the Infiniband-only case in Figure 5 is the same as the one featured in Figure 4. This gives a point of comparison between both graphs. Open MPI also features a multirail support[7] but to the extent of our knowledge, this functionality is not fully operational in the release we tested. The results clearly confirm the behaviour of the multirail strategy enforced by NewMadeleine which is to choose the fastest network for small messages (Infiniband in our case) and to distribute the message chunks across the multiple networks in case of large messages. Indeed, we obtain an aggregated bandwidth that corresponds almost to the sum of the individual Myrinet and Infiniband bandwidths. Since both performs equally, each chunk has the same size (that is, half of the total message size). But NewMadeleine is able to balance the load according to each network's performance when they differ in order to achieve the best results.

4.1.2 Impact of PIOMan's integration into Nemesis

We now study the impact of our centralized progression subsystem on pingpong tests. We also evaluate the progression of asynchronous communication and the ability to overlap communication with computation.

PIOMan's raw overhead We first evaluate the impact of the centralized progression subsystem within MPICH2 for both intra-node and inter-node communication. Figure 6 shows the latency results for the Netpipe program over shared memory and over Myrinet MX. The introduction of PIOMan within the Nemesis software stack significantly af-



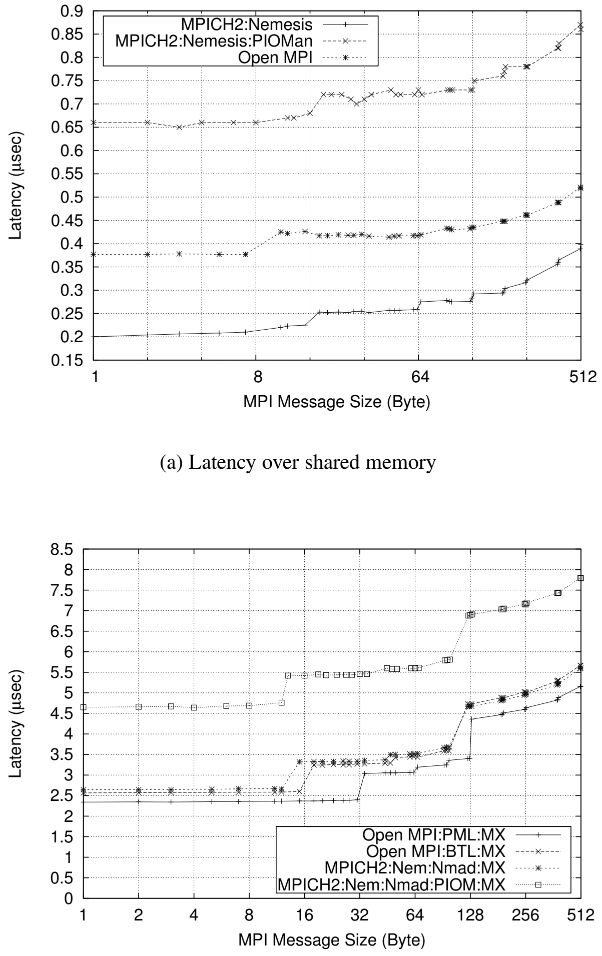
(a) Latency

(b) Bandwidth

Figure 5. MultiRail Performance with Myrinet 10G and Infiniband 10G NICs

fects the latency (roughly 450 ns for shared memory). However, the overhead is constant as the message size grows and becomes negligible for large messages. The overhead introduced is mainly due to synchronization since the progression subsystem is totally thread-safe.

The use of PIOMan within the inter-node communication subsystem also introduces an overhead (roughly 2 μs). This more expensive impact can be explained by the need for a stronger synchronization in that case: to poll a network within NewMadeleine it is required to modify lists of requests that have to be protected from concurrent accesses. Network drivers also have to be protected against concurrent accesses since most of them are not thread-safe.



(a) Latency over shared memory

(b) Latency over Myrinet MX

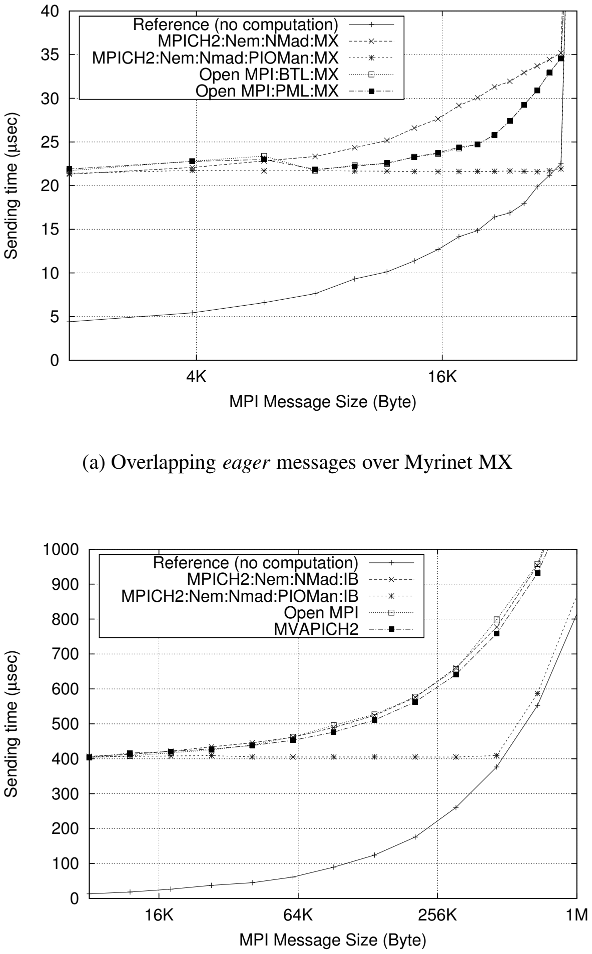
Figure 6. Latency Performance with PIOMan

Overlapping communication with computation In order to evaluate the ability to overlap communication with computation for both small (i.e *eager*) and large messages (i.e messages that require a *rendezvous* handshake), we proceed as follows: the sender calls `MPI_Isend`, computes for a while and waits for the end of the communication (using `MPI_Wait`). Then the sender waits for an incoming message. We measure the time required to send the message and to perform the computation.

Figure 7(a) shows the results obtained for *eager* messages over Myrinet MX with a computation time of 20 μs. Open MPI (both the MX BTL and PML CM versions) and MPICH2 do not overlap communication with computation: the measured sending time roughly corresponds to $sum(computation, communication)$. MPICH2 on top of the multithreaded version of NewMadeleine – that uses PI-

OMan to make communication progress – overlaps communication with computation: the sending time corresponds to $max(computation, communication)$.

Figure 7(b) shows the results obtained for large messages over Infiniband with a computation time of 400 μs. The goal here is to evaluate the progression of the *rendezvous* handshake. Open MPI, MVAPICH2 and MPICH2 do not detect the handshake during the computation and thus the communication is not overlapped. MPICH2 on top the the multithreaded version of NewMadeleine uses an idle core to poll the network to detect the handshake quickly and thus is able to overlap communication with computation.



(a) Overlapping *eager* messages over Myrinet MX

(b) Making *rendezvous* progress over Infiniband

Figure 7. Asynchronous progression of communication

4.2 NAS parallel benchmarks

Beside the previous point-to-point experiments, we also evaluate NAS application kernels. on the Grid5000 [1] testbed. The 10 nodes that we used feature each 4 CPUs (with 2 cores per CPU). The CPUs are 2.6 GHz AMD Opteron 2218 with 2 MB of L2 cache. Each node features 32GB of memory and one Infiniband 10G NIC. As previously, we compare MPICH2-NewMadeleine (with and without the PIOMan support) to both MVAPICH2 and Open MPI. All are compiled using the same optimization level, support for shared-memory communication is enabled, thread support and error checking are both disabled. The experiments are carried out with respectively 8, 16, 32 and 64 computing processes (8 and 32 are replaced with 9 and 36 for kernels requiring a square number of processes). In the 8 (or 9) processes case, only one process runs on a node and no communication uses shared memory. We gathered data for all NAS benchmarks, except for IS. Indeed, IS needs datatypes support and MPICH2-NewMadeleine does not handle yet this functionality. For each kernel, we tested A,B and C classes. However, the behaviour for all classes following roughly the same pattern, Figure 8 shows only class C results. Please note that results for MPICH2-NewMadeleine with PIOMan are only available for the 8, 16 and 32 processes cases. The 64 processes case as well as the MG and LU kernels are not yet available due a problem in the current implementation that leads to deadlocks in these cases.

Figure 8 shows that for the various NAS kernels considered, all MPI implementations exhibit a good scalability factor as the number of processes is growing. There is one exception, SP with 36 processes, where the performance is rather low for all implementations. It is to be noted that Open MPI lags behind other implementations in the EP and the LU kernels cases, regardless of the number of processes involved. In the 8 processes case, shared memory is not used so the intra-node communication support is not the issue here. The LU kernel sends only a limited percentage of large messages and most of the traffic is composed of small messages (a few KBytes, size depending on the class). So, according to Figure 4, latencies and bandwidths are in the same range for all implementations. MPICH2-NewMadeleine's performance level is globally on par with network-tailored MPI implementations, while using a generic communication layer. The version featuring PIOMan delivers the same level of performance, the overhead is usually less than 3%. In the FT and SP kernels, PIOMan improves performance slightly. However, PIOMan's ability to overlap communication with computation proves difficult to test with the NAS. [12] evaluates such a *potential* overlap of communication with computation for the NAS kernels that require application changes. The scheme required by PIOMan where :

1. a non-blocking communication operation request is posted
2. some computation is performed
3. the request is completed

is slightly used in the NAS. Yet, the performance of applications that use more intensively this scheme should be improved by PIOMan.

5 Conclusion and future work

In this paper, we presented our MPICH2-NewMadeleine implementation. We showed that the current CH3 device coupled with the Nemesis channel needs to be modified in order to fully exploit communication interfaces with advanced features (e.g tag-matching capabilites). The software stack takes advantage of both the Nemesis subsystem for intra-node communication and the NewMadeleine generic library for inter-node communication involving high-performance networks. Communication progress is also enhanced thanks to the PIOMan software. This work paves the way to an MPI implementation tailored for clusters composed of multicore CPUs nodes featuring multirail networks. The performance achieved is promising, and compares farouvably to other finely-tuned, specialized MPI implementations.

Even though the current status of this work allows us to execute and run MPI applications such as the NAS computing kernels, we do not support the whole set of MPI functionalities. In particular we think that NewMadeleine's optimization schemes might improve performance for noncontiguous user datatypes. Another challenge would be to efficiently support MPI2 RMA operations without compromising the optimizations implemented. Also, the performance level (latency) obtained when PIOMan is integrated should be improved in order to reduce the gap with the regular version of Nemesis. We also intend to exhibit the benefits of PIOMan on real applications, especially in the overlapping department.

Since we built our implementation on NewMadeleine and PIOMan, some multithreading aspects have been addressed that shall be completed in order to get an MPI implementation that works smoothly with our OpenMP support based on the Marcel thread library [10]. Indeed, we think that MPICH2-NewMadeleine is the first step towards an efficient implementation of an hybrid MPI+Open MP programming model.

Acknowledgments

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357 and by the ACI GRID.

References

[1] Grid'5000. `http://www.grid5000.fr`.

[2] MX. `http://www.myri.com/scs/download-mx.html`.

[3] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. NewMadeleine: a Fast Communication Scheduling Engine for High Performance Networks. In *CAC 2007: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007.*

[4] Olivier Aumage, Elisabeth Brunet, Guillaume Mercier, and Raymond Namyst. High-Performance Multi-Rail Support with the NewMadeleine Communication Library. In *HCW 2007: the Sixteenth International Heterogeneity in Computing Workshop, held in conjunction with IPDPS 2007.*

[5] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem. *Parallel Computing, Selected Papers from EuroPVM/MPI 2006*, 33(9):634–644, September 2007.

[6] François Trahay, Elisabeth Brunet, Alexandre Denis, and Raymond Namyst. A multithreaded communication engine for multicore architectures. In *CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008.*

[7] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings of the Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, 2006.

[8] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. Design of High Performance MVAPICH2: MPI2 over InfiniBand. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid.*

[9] Quadrics Ltd. Elan Programming Manual, 2003. `http://www.quadrics.com/`.

[10] Runtime Team, LaBRI-Inria Bordeaux – Sud-Ouest. Marcel: A POSIX-compliant thread library for hierarchical multiprocessor machines, 2007. http://runtime.bordeaux.inria.fr/marcel/.

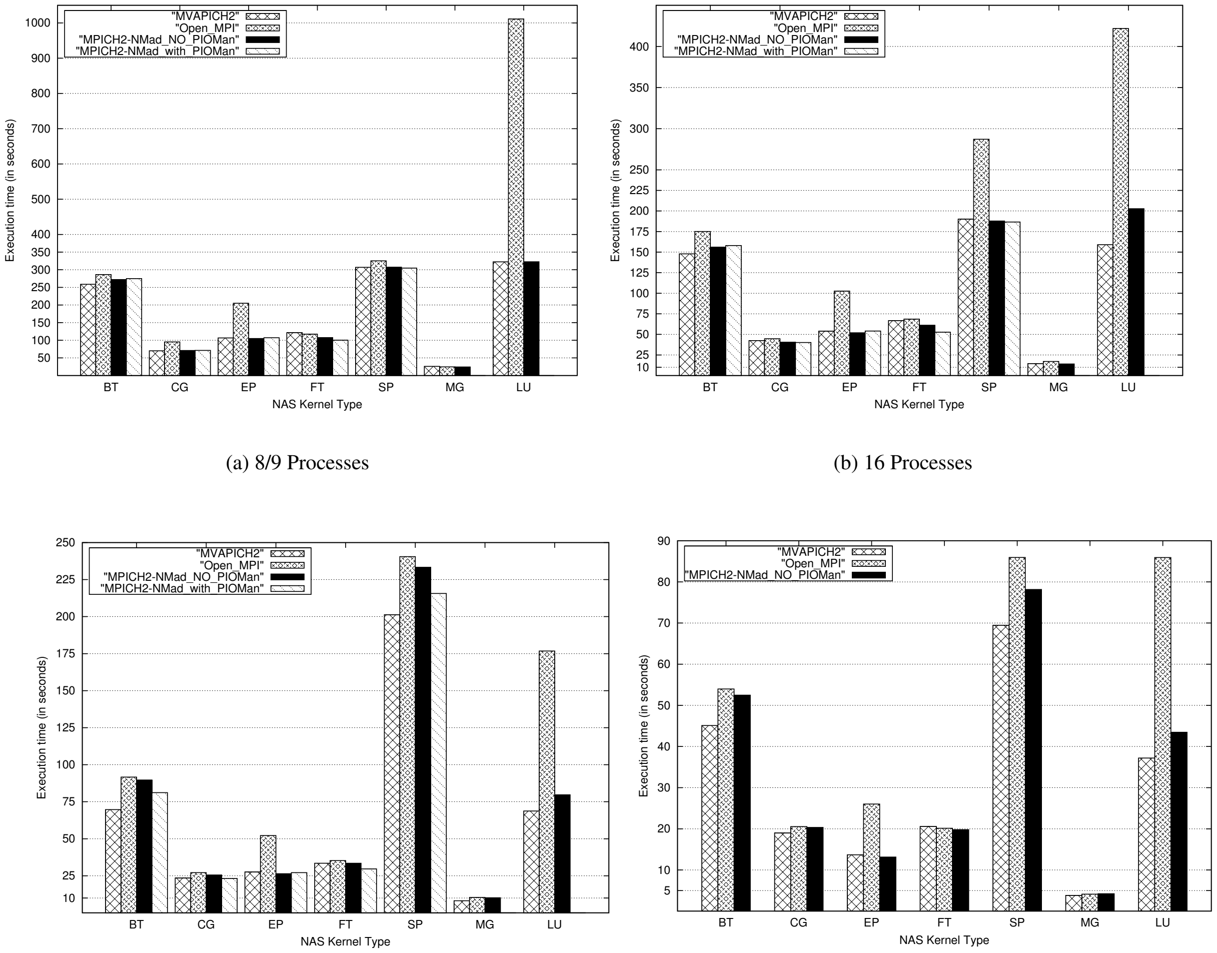
[11] Runtime Team, LaBRI-Inria Bordeaux – Sud-Ouest. PM2 Software Suite, 2008. http://runtime.bordeaux.inria.fr/.

[12] Aniruddha G. Shet, P. Sadayappan, David E. Bernholdt, Jarek Nieplocha, and Vinod Tipparaju. A framework for characterizing overlap of communication and computation in parallel applications. *Cluster Computing*, 11(1):75–90, 2008.

[13] Galen M. Shipman, Ron Brightwell, Brian Barrett, Jeffrey M. Squyres, and Gil Bloch. Investigations on InfiniBand: Efficient Network Buffer Utilization at Scale. In *Proceedings, Euro PVM/MPI 2007*.

[14] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. Netpipe: A network protocol independent performace evaluator. In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[15] Francois Trahay, Alexandre Denis, Olivier Aumage, and Raymond Namyst. Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager. In *Proceedings, Euro PVM/MPI*.



(a) 8/9 Processes

(b) 16 Processes

(c) 32/36 Processes

(d) 64 Processes

Figure 8. NAS Benchmark Performance for Class C