An analysis of the impact of multi-threading on communication performance

François Trahay, Elisabeth Brunet, Alexandre Denis

An analysis of the impact of multi-threading on communication performance

François Trahay Élisabeth Brunet
Alexandre Denis
INRIA, LABRI, Université Bordeaux 1
351 cours de la Libération
F-33405 TALENCE, FRANCE
{trahay,brunet,denis}@labri.fr

Abstract

Although processors become massively multicore and therefore new programming models mix message passing and multi-threading, the effects of threads on communication libraries remain neglected. Designing an efficient modern communication library requires precautions in order to limit the impact of thread-safety mechanisms on performance. In this paper, we present various approaches to building a thread-safe communication library and we study their benefit and impact on performance. We also describe and evaluate techniques used to exploit idle cores to balance the communication library load across multicore machines.

1 Introduction

The current trend in cluster architecture leads toward an increase of the number of cores per node. It becomes common to have 8 or 16 cores per node and the evolution of processors is leading to tens or maybe hundreds of cores per node. Thus, the approach to exploit clusters has to evolve since the classical "pure MPI" model suffers from scalability limitations: the increasing number of MPI processes per node may for instance exhaust the memory or TLB space. In order to override these limitations, hybrid solutions that mix the use of threads and MPI processes seem to be the best candidate. Such paradigms allow to pool the hardware resources and to exploit them as much as possible. However the use of threads requires some precautions in communication libraries so as to avoid race conditions when threads access concurrently the library.

In this paper, we describe various solutions for multi-threading support in communication libraries, and we analyze the benefits and cost of each of them. We consider two main levels of multi-threading support. The first level is thread-safety, which means that a multi-threaded application can perform communication in multiple threads. In MPI, this level is known as `MPI_THREAD_MULTIPLE`. Many approaches for locking can be used when ensuring thread safety in a communication library. We also consider one level further of multi-threading support which consists in a multi-threaded communication engine. We consider several techniques to use multiple cores in order to make non-blocking communication primitives really progress in background. We evaluate the impact of such mechanisms on raw performance as well as their potentiel benefits for applications.

Thread-safety and communication engine multi-threading may be implemented in various different ways. In this paper, we aim at decomposing each step of thread support and we analyze precisely the cost and the benefits of each part. We have implemented all presented features in our NEWMADELEINE communication library and we have extensively profiled the code.

The remaining of this paper is composed as follows. Section 2 presents the software and hardware of the experimental testbed used to conduct our experiments. Section 3 analyzes the behavior and cost of various ways for ensuring thread-safety. Section 4 presents the cost and benefits a communication library can take from being itself multi-threaded. Section 5 presents related works. Finally, Section 6 draws a conclusion of this study and shows directions for further work.

2 Experimental testbed

In order to analyze the impact of multi-threading on communication, we have conducted experiments with our PM2 software suite, composed of a communica-

tion library (NEWMADELEINE), a multi-threading library (MARCEL), and an I/O event manager (PIOMAN).

Our communication library for high performance networks is called NEWMADELEINE [1] and is available over MX/Myrinet, Verbs/InfiniBand, Elan/QsNet, and TCP/Ethernet. As depicted in Figure 1, NEWMADELEINE has a 3-layer architecture with its activity driven by the underlying NICs, in contrast with the usual behavior of most communication libraries driven by the send/recv from the application. The core layer applies dynamic scheduling optimizations on multiple communication flows such as packet reordering, coalescing, multirail distribution, etc. NEWMADELEINE implements both a specific API and a MPI interface called Mad-MPI.

The application provides messages to the collect layer. When a NIC becomes idle, the optimization layer is invoked so as to compute the best message arrangement (by aggregating messages, splitting messages, etc.) and to submit the next packet to send to the transfert layer that uses the NICs' drivers to transmit the arranged packet.

For multi-threading, we used our MARCEL [9] multi-threading library. It features a two-level thread scheduler that achieves the performance of a user-level thread package while being able to exploit SMP machines. It is highly tunable and includes hooks usable for asynchronous communication progression. The communication engine of the PM2 software suite is called PIOMAN [11]. It handles polling in behalf of the communication library and works closely with the thread scheduler. It is able to perform polling inside MARCEL hooks (when a core is idle, on context switch, on timer interrupts) or within tasklets in order to exploit any core of the machine.

The benchmarks in the following Sections have been performed on a set of quad-core 3.16 GHz Xeon X5460 boxes with 4 GB of main memory running Linux version 2.6.26. Nodes are interconnected through Myricom Myri-10G NICs (with the MX 1.2.7 driver) and ConnectX Infiniband DDR (MT25418, with the OFED 1.3.1 driver). Latency graphs presented in this paper have been obtained on Myrinet MX. We obtained similar results with Infiniband.

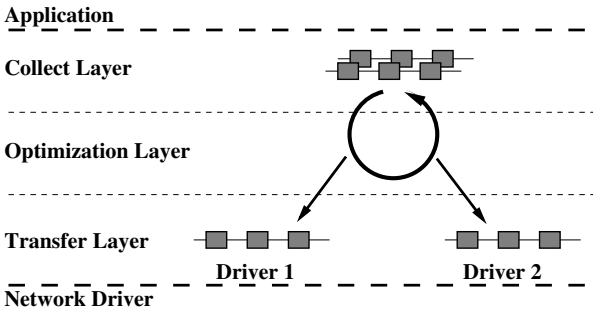

Figure 1. Architecture of NEWMADELEINE

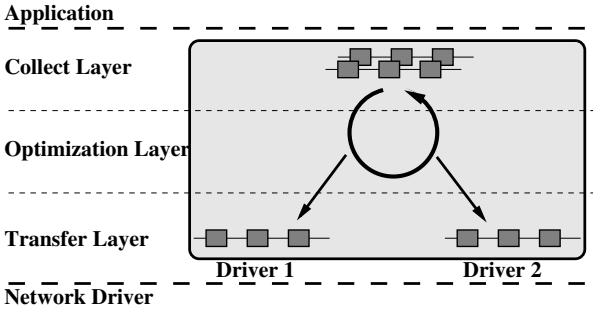

Figure 2. Coarse-grain locking in NEWMADELEINE

3 Impact of thread-safety

In this Section, we study various mechanisms required when designing a fully thread-safe communication library. We also evaluate the benefits and performance impact of such techniques.

3.1 Coarse-grain locking

The easiest way to protect a communication library from concurrent accesses consists in using a coarse-grain locking mechanism. Each time a thread accesses the library, a mutex is held (see Figure 2, the gray box is the scope of the lock). When the thread returns from the library, the mutex is released. The mutex is also released before entering a blocking section in order to avoid deadlocks. This method permits to support concurrent accesses without paying a too large overhead: each access to the library causes only one locking operation. As the mutex is held for a very short period (a few microseconds at most), we use *spinlocks* to implement this coarse-grain locking mechanism within NEWMADELEINE. For such very short critical sections, *spinlocks* are more efficient than plain mutex. If another thread already holds the lock when a thread tries to acquire it, the thread waits actively until the lock is released, with no context switch.

In order to evaluate the overhead of this locking mechanism, we performed a pingpong test. The results we obtained are depicted on Figure 3. The use of

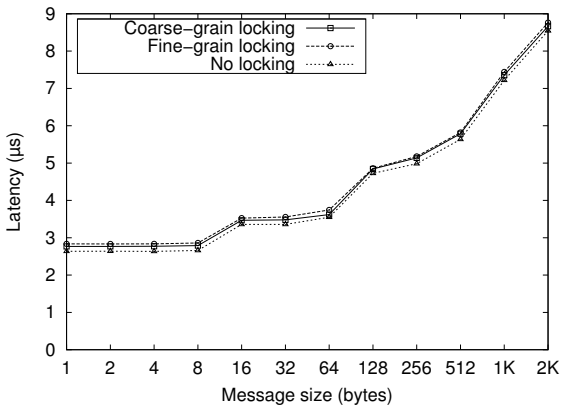

Figure 3. Impact of locking on latency

a library-wide lock ("coarse-grain locking" on the Figure) implies a constant overhead of 140 ns that do not impact bandwidth. The spinlock is held and released twice (once for submitting the message to the collect layer, once to transmit it through the network), each acquire/release cycle costs 70 ns.

The global lock ensures thread-safety with a limited impact on latency but suffers from a lack of parallelism: as soon as a thread enters the library, the communication actions (polling, message submission, etc.) are limited to the one performed by this thread. Thus communication processing is serialized when several threads communicate. On Figure 5, we can see the results we obtained for a concurrent pingpong test: two threads perform pingpong tests concurrently. We observe that the latency for each thread roughly corresponds to twice the single-thread latency. This is due to the mutual exclusion between the two threads that have to wait each other to access the communication library.

3.2 Fine-grain locking

In order to get an efficient thread-safe communication library, it is necessary to allow threads to process communication flows in parallel. Similar actions should still be performed under mutual exclusion (e.g. polling a thread-unsafe network) but unrelated processes should be parallelizible: it should be possible to send a message over a network while receiving data from another NIC. Polling in parallel over different networks should also be permitted.

To allow such parallel actions, locking has to get finer and critical sections has to be identified more precisely. In NEWMADELEINE shared data structures are limited to two sets of lists:

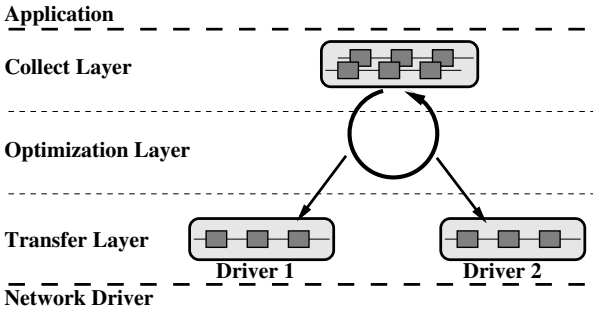

Figure 4. Fine grain locking in NEWMADELEINE

- The lists of packets to schedule in the collect layer (one list per peer). This list is accessed by the application (through `nm_isend`, etc.) and by the optimization layer. A synchronization issue may thus appear here if the application adds a new packet to send (or to receive) while NEWMADELEINE's optimization layer remove a packet (*i.e.* a new optimized packet is provided to the driver)

- The lists of packets to send through the network in the transfer layer (one list per driver). These lists are accessed by the optimization layer when a new optimized packet is submitted to the transfert layer. A NEWMADELEINE driver accesses its list when the corresponding NIC becomes idle: it then removes an entry from the list and transmit the packet to the NIC driver. Accesses to each of these lists have to be performed under mutual exclusion in order to avoid both the optimization layer and the transfert layer to modify a list simultaneously.

Implementing a fine-grain locking mechanism thus boils down to use separate locks for each list as shown on Figure 4. The lists in the collect layer are protcted through a global lock as the packet scheduler needs to iterate over those lists to generate a packet to send.

To evaluate the overhead of such a mechanism, we performed a classical pingpong test. The results we obtained are shown on Figure 3. We measured that the fine-grain locking introduces a constant overhead of 230 ns with no impact on bandwidth. This overhead is higher than the coarse-grain locking (140 ns) because the number of locks is higher in the critical path.

Figure 5 shows the results we obtained for the concurrent pingpong test described in Section 3.1. Fine-grain locking performs better than coarse-grain locking as the communication library can be accessed simultaneously by several threads. The latency obtained by

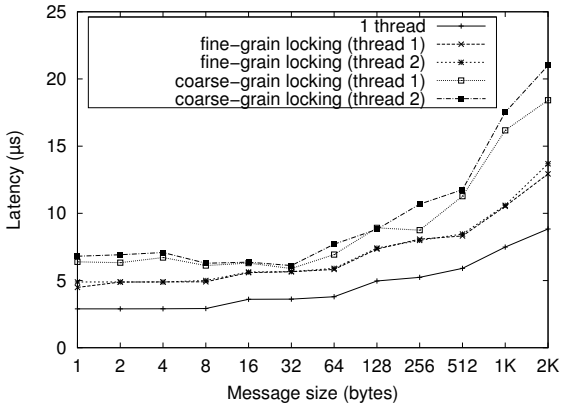

Figure 5. Two threads perform concurrently pingpong programs

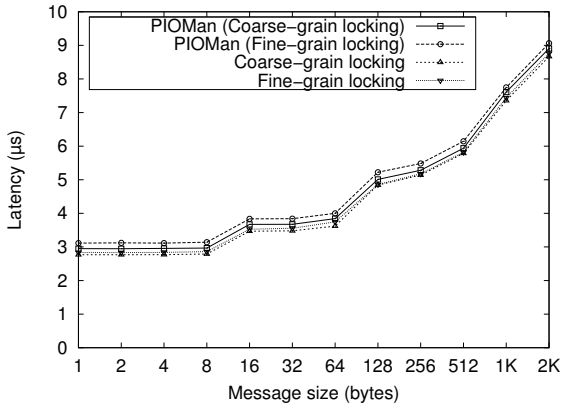

Figure 6. Impact of PIOMan on latency

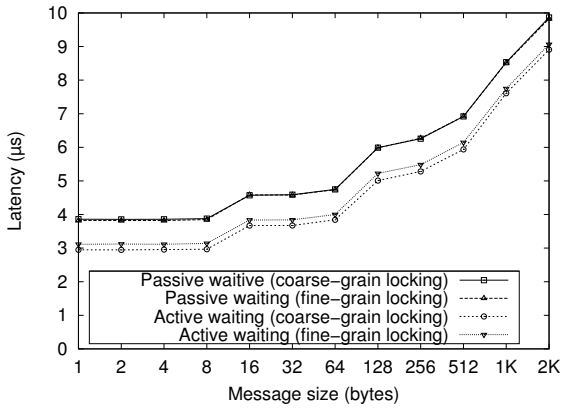

Figure 7. Impact of semaphores on latency

each thread is though higher than the latency obtained when using only one thread. This can be explained by the more intensive use of the NIC and by the contention when accessing the different locks.

3.3 Busy waiting v.s. passive waiting

In order to design an efficient thread-safe communication library, it is necessary to understand its behavior in a multi-threaded context. A classical problem when mixing threads and communication is the implementation of waiting functions (e.g. `MPI_Wait`). Most regular communication libraries implement waiting function as busy waiting: when a thread waits for the end of a communication, it keeps polling until the corresponding network request succeeds.

Although this behavior is extremely efficient in a single-threaded environment, it can be problematic when multiple threads perform this operation in parallel. In this latter case, polling is done concurrently and thus contention may decrease the polling frequency for each thread. Another problem here is that performing the same operation on several CPUs simultaneously wastes CPUs that could be used to schedule the application threads.

In the thread scheduler's point of view, waiting threads should wait on blocking primitives (semaphores, conditions, etc.) in order to let other threads be scheduled. This behavior permits to decrease the overal execution time for applications that intensively use threads and implementing waiting functions as busy waiting may lead to an inefficient use of processors: for instance, on a 4-core machine, dedicating one core to communication leads to up to 25 % decrease of the computation power.

But if waiting functions are based on classical blocking primitives, no polling will be performed while a thread is blocked. The blocking primitives thus have to be modified. In NEWMADELEINE, this is implemented by the PIOMAN progression engine that is called from the thread scheduler when a thread is about to block on a semaphore. This optimization requires modifications of the thread scheduler in order to add a few hooks at key points (CPU idleness, context switches, timer interrupts, etc.). These hooks are used to call PIOMAN so as to poll the networks.

To evaluate the impact of using PIOMAN to poll the network, we performed the same latency test as in Section 3.1. The results depicted on Figure 6 show an overhead of 200 ns due to the management of PIOMAN internal lists as well as locking.

The use of blocking primitives also introduces an overhead since it implies expensive context switches.

Figure 7 shows the results of the latency test when waiting functions are implemented with semaphores. It appears that the impact of the context switches on latency is rather high (750 ns).

It is thus important to avoid these context switches when possible. A solution consists in using a *fixed spin* algorithm [7] that mixes active and passive waiting: when a thread is about to block on a semaphore, it first polls for a short duration (for instance 5 μs) and then enters the semaphore. By doing this, the context switch is avoided if the expected event occurs within 5 μs. If this event happens later, the context switch occurs but its cost is amortized as it then represents a small percentage of the total communication cost.

4 Multi-threading the communication engine

In the previous Section, we have shown how to ensure thread-safety in a communication library, and we have measured its performance overhead. In this Section, we study the impact of using idle cores to process communication flows. Indeed, the development of multicore chips and the increase of the number of cores per node make it important to *take advantage* of multi-threading instead of only *supporting* it.

We showed that using fine-grain locking within a communication library has a limited overhead, so we only consider this type of locking mechanism here.

4.1 First step: using idle cores to make communication progress

The increase of the number of cores per node may lead to *"holes"* in the scheduling: as the number of thread increases, the need for both intra-node and inter-node synchronization makes threads wait for incoming messages or mutex release. These holes can be used to make communication progress in the background. We showed [11] that *rendezvous* handshakes can be managed by idle cores, allowing to overlap computation and communication of large messages.

This technique requires to poll the network on a core while the application computes on another core. To evaluate the overhead of defering the polling to a separate core, we performed a pingpong test that binds the main thread to a CPU.

The results we obtained for this test are depicted in Figure 8. The application thread is bound to CPU 0, thus polling on this CPU leads to better performance. Polling on a CPU that shares a L2-cache (CPU 1) implies an overhead of 400 ns and polling on a CPU that does not share a cache with CPU 0 (CPU 2 or CPU 3) leads to an overhead of 1.2 μs. These extra costs are due to communication between cores and cache misses.

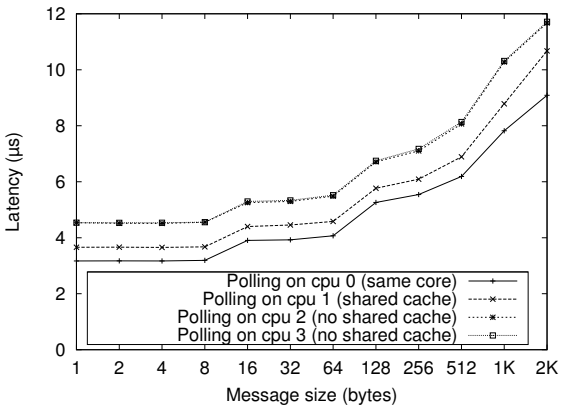

Figure 8. Impact of cache affinity on a quad-core chip

The same test has been carried out on a set of dual quad-core Xeon machines and the results are similar: polling on a CPU that shares a cache (CPU 1) costs 400 ns, polling on the same chip but on a separate cache (CPU 2 or CPU 3) costs 2.3 μs and polling on another chip (CPU 4 to CPU 7) costs 3.1 μs.

4.2 Second step: using idle cores to perform cpu-intensive tasks

In addition to the progression of communication in the background, idle cores can be exploited to perform time-consuming operations such as message submission to the network [10]. This way, communication of small messages and computation are overlapped. In our previous papers, PIOMAN relied extensively on tasklets [12] to offload communication processing. Tasklets are well-suited for such mechanisms as it offers a convenient way to defer a treatment. It is though possible to offload communication processing without using tasklets: while a core is idle, MARCEL invokes PIOMAN that can detect that a message needs to be submitted to a network. This requires some precautions since usual locking mechanisms cannot be used in this context.

In order to evaluate these two techniques of message submission offloading, we performed an overlapping test that consists in a pingpong using non-blocking communication primitives. A 10 μs computing phase is inserted between the message submission (*i.e.* `nm_isend`) and the message waiting (*i.e.* `nm_wait`). The results depicted on Figure 9 show

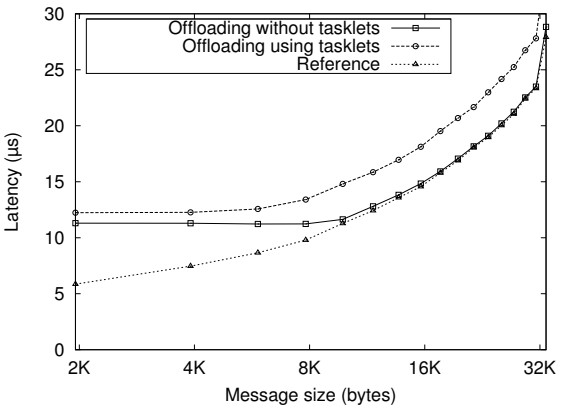

Figure 9. Impact of tasklets on defered message submission

that offloading message submission with tasklet introduces an overhead of 2 μs whereas using idle cores to transmit the data (without tasklets) costs 400 ns. This latter overhead corresponds to the cost of defering polling on an idle core as explained in Section 4.1. The overhead of tasklets seems to be due to the complex locking mechanism involved when a tasklet is invoked.

5 Related work

Although processor development is clearly heading to a massive use of multicore processors, the issue of multi-threading in communication library has received little attention in the literature. Most MPI implementations do not fully support the use of threads. MiMPI [5] is thread-safe and is able to use internal threads to make communication progress. Though, this implementation is only available for TCP and performs badly for small messages. USFMPI [4] is a MPI-1.2 implementation for TCP and Myrinet GM that supports multi-threading. It also uses threads to make *rendezvous* handshakes progress in the background. MPICH-Madeleine [2] is a thread-safe MPI implementation that uses internal threads to make asynchronous communication progress. Several approaches have been studied [3] to build a thread-safe MPI implementation, and implemented in MPICH2, making it fully thread-safe. OpenMPI [6] provides the `MPI_THREAD_MULTIPLE` thread-safety level and can use a progression thread for TCP, but these options are advertised as lightly tested and are unpublished as far as we know. Works on multi-threading are not limited to MPI implementations and some low-level communication libraries such as Myrinet MX [8] are able to run threads in order to make communication progress in the background.

6 Conlusion and future work

We have studied the impact of thread-safety when implementing a generic communication library. We have shown that ensuring basic thread-safety has an overhead close to zero on latency and very few modifications need to be applied to the implementation, even though the identification of critical sections can be tedious. Locking mechanisms required to do this have a limited impact on raw performance and permits to achieve good results when the application uses threads.

The impact of introducing multi-threading inside the communication library itself has also been studied. We showed that exploiting idle cores for background communication progression or to offload CPU-intensive tasks is useful for applications able to overlap communication and computation. However, these features have a strong impact on raw network latency. In particular, despite the convenience of tasklets to defer communication treatments to idle cores, such mechanism must be used carefully as it implies a significant overhead. Moreover, it appears that using idle cores to process communication flows requires to take thread locality and cache effects into account.

This contribution opens the path to future works such as using internal multi-threading in the ongoing port of MPICH2-Nemesis over the NEWMADELEINE-PIOMAN software stack. This will allow us to benchmark our multi-threaded communication library with real applications that mix multi-threading and message passing.

Acknowledgements. This work has been funded by the project "LEGO" (ANR-CICG05-11) from the French National Agency for Research (ANR).

References

[1] O. Aumage, E. Brunet, N. Furmento, and R. 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*.

[2] O. Aumage, G. Mercier, and R. Namyst. MPICH-Madeleine: a True Multi-Protocol MPI for High-Performance Networks. In *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, San Francisco, 2001. IEEE.

[3] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward Efficient Support for Multithreaded MPI Communication. In *Recent Advances in Parallel*

Virtual Machine and Message Passing Interface: 15th European PVM/MPI Users' Group Meeting, 2008.

[4] S. Caglar, G. Benson, Q. Huang, and C. Chu. USFMPI: A Multi-threaded Implementation of MPI for Linux Clusters. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2003.

[5] F. Garcia, A. Calderón, and J. Carretero. MiMPI: A Multithread-Safe Implementation of MPI. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 1999.

[6] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A Flexible High Performance MPI. In *The 6th Annual International Conference on Parallel Processing and Applied Mathematics*, 2005.

[7] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. *ACM SIGOPS Operating Systems Review*, 25(5):41–55, 1991.

[8] Myricom Inc. Myrinet EXpress (MX): A High Performance, Low-level, Message-Passing Interface for Myrinet, 2003. `http://www.myri.com/scs/`.

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

[10] F. Trahay, E. Brunet, A. Denis, and R. Namyst. A Multithreaded Communication Engine for Multicore Architectures. In *CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, Apr. 2008. IEEE.

[11] F. Trahay, A. Denis, O. Aumage, and R. Namyst. Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager. In *EuroPVM/MPI*. Springer, 2007.

[12] M. Wilcox. I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers. In *Linux.conf.au*, Perth, Australia, January 2003. The University of Western Australia.