A multithreaded communication engine for multicore architectures

François TRAHAY Elisabeth BRUNET Alexandre DENIS
Raymond NAMYST

INRIA, LABRI, Université Bordeaux 1
351 cours de la Libération
F-33405 TALENCE, FRANCE
{trahay,brunet,denis,namyst}@labri.fr

Abstract

The current trend in clusters leads towards an increase of the number of cores per node. As a result, an increasing number of parallel applications is mixing message passing and multithreading as an attempt to better match the underlying architecture's structure. This naturally raises the problem of designing efficient, multithreaded implementations of MPI. In this paper, we present the design of a multithreaded communication engine able to exploit idle cores to speed up communications in two ways: it can move CPU-intensive operations out of the critical path (e.g. PIO transfers offload), and is able to let rendezvous transfers progress asynchronously. We have implemented these methods in the PM2 software suite, evaluated their behavior in typical cases, and we have observed good performance results in overlapping communication and computation.

1 Introduction

The current biggest change that affects the architecture of conventionnal microprocessors is undoubtedly the increase of the number of cores per chip. While usual PC clusters still had few processors 5 years ago, it is now ordinary to have 8 or even 16 cores per machine, and the future ones will certainly hold hundreds of cores as the experimental 80-cores Intel processor foreshadows. In order to optimally exploit these new resources, it is necessary to change the way current cluster applications are designed. In particular, the "pure-MPI" approach, which consists in allocating one process per core, may not scale on massively multicore machines and requires costly mechanisms to keep the workload balanced among the whole set of cores. Most importantly, such an approach exhibits severe limitations in terms of fair and efficient use of the underlying network interface cards (NICs), as it entirely relies upon the network device driver for the scheduling and the multiplexing of the multiple communication flows

A lot of researchers have proposed hybrid solutions based on mixing multithreading and message passing, and many of them actually consist in using OPENMP+MPI [9, 11], where only one MPI process is created per node and comprised of several threads initiated by OPENMP. This leads to providing the runtime systems with a global view of the hardware utilization, and consequently to a better implementation of workload balancing, communication scheduling algorithms and congestion avoidance mechanisms.

This multiplication of communication sources implies that the underlying communication libraries must be able to handle concurrent communication requests. We showed [2] that the NEWMADELEINE communication engine was able to efficiently deal with multithreaded applications by applying various scheduling strategies – such as data aggregation – over the global communication flow.

In this paper, we present an extension of this communication engine that takes advantage of multithreading. The new design improves application's reactivity to communication events and augments parallelism of communication processing using generalized offloading. In Section 2, we present the design of an event-driven communication engine able to feed idle cores with communication tasks so as to speed up non-blocking transfers. In Section 3, we describe some highlights of the implementation. Section 4 gives a performance evaluation of our communication engine. Finally Section 5 draws a conclusion and gives ideas about future works.

2 Design of a multithreaded communication engine

Classical non-multithreaded communication engines such as OpenMPI [3], MPICH2 [1] or MVAPICH2 [4] do not fully take benefit from multicore architectures: com-

munication flows are processed independently with no centralized view. This is mainly due to the way requests are treated: if the application performs a non-blocking *send*, the communication processing (registering the request, submitting it to the network, etc.) is done sequentially by the communicating thread. Thus, it is hard to balance the treatment of different requests over the available cores. It is also difficult to ensure safety when several threads access simultaneously the communication library except through a library-wide scope mutex and most current MPI implementations do not support this level of thread-safety. Most thread-safe communication libraries actually only *support* the use of threads without *taking advantage* of multithreading.

2.1 An event-driven architecture

We have designed a multithreaded communication engine that takes advantage of the underlying multicore architecture. Unlike classical approaches that are mainly sequential, it uses an event-driven model that can natively spread unrelated communication treatments over different cores.

Communication processes are split into separate operations that can be performed anywhere on the machine. For instance, as depicted in Figure 1, sending data consists in (a) registering the request, (b) submitting the request to the network, (c) waiting for the request's completion. These operations are not required to be performed on the same core, and may be spread over the available CPUs depending on the load. This way, asynchronous *send* or *recv* operations can be performed in the background on idle cores: the asynchronous *send* actually only registers the request in a work list and generates an event. If a CPU is idle or running a low priority thread, the event is processed: the request is submitted to the network. Otherwise, the event will be processed later, when a core becomes idle or when the application reaches a *wait* operation. By doing this, communication operations are performed in the background and are overlapped with computations.

This *event-driven* model allows the communication engine to fully exploit multicore architectures by offloading asynchronous operations. Thus, many treatments can be performed on cores left idle by the application and thereby accelerate the communication processing. Applications that present irregular workload transparently benefit from such accelerations. The *event-driven* model also allows us to apply global optmizations strategies on communications flows.

Moreover, thread-safety can be efficiently implemented by using this *event-driven* model: instead of locking the whole communication processing with a mutex, it is possible to protect the processing of events separately. This way, each event is run under mutual exclusion and several threads can perform different operations at the same time. For instance, the contention problem when submitting a request to a NIC no longer occurs as the messages are submitted once at a time. As the communication processing runs for a very short period of time, the synchronization can be achieved by using light primitives such as spinlocks.

2.2 Offloading small messages submission

A classical problem when performing asynchronous communication is the CPU load required to send small packets. On most high performance networks, small packets are copied into a registered memory region, *i.e.* a memory region whose physical address is known from the network card, then they are sent through DMA, and very small packets (typically up to 128 bytes) are sent through PIO. In either case, the operation needs many CPU cycles, up to several dozens of microseconds, and may thus monopolize a core for a long time. Therefore, when an application submits a small packet to a non-multithreaded communication engine, the packet is actually submitted to the network by the application thread itself. Thus even a non-blocking *send* may take several dozens of microseconds to return.

In our event-driven model, a non-blocking *send* is split into the following operations: (a) registering the request, (b) submitting the request to the network interface. In this example, the second operation can be executed on any core. This way, the expensive data copy can be performed by an idle CPU, avoiding the communicating thread to wait for the end of this copy. If an idle core has completed the submission of the request when the application reaches the *wait* operation, the copy has actually overlapped the computation. Otherwise, the request submission is performed during the *wait* and the copy has only been delayed. Thus, the offload has no impact on regular computations.

Although this method may increase the latency (because of cache effects for instance), this load-balancing allows the communication engine to reduce the critical path and to accelerate the communication processing by overlapping asynchronous communication efficiently.

In order to offload efficiently asynchronous communication processing, it is important to avoid useless (and expensive) copies. Our model only proceeds to copy when it is inevitable: on the send path, the buffer's address is passed to the NIC that copies the buffer through DMA or PIO (depending on the buffer's length). The receive path can be more complicated: if an unexpected message arrives, it is copied into a buffer allocated especially for unexpected messages. When the corresponding receive request is posted, the message is detected and copied into the application's buffer. If the message is expected, the NIC directly copies the data to the buffer allocated by the application through DMA or PIO (depending on the message length).

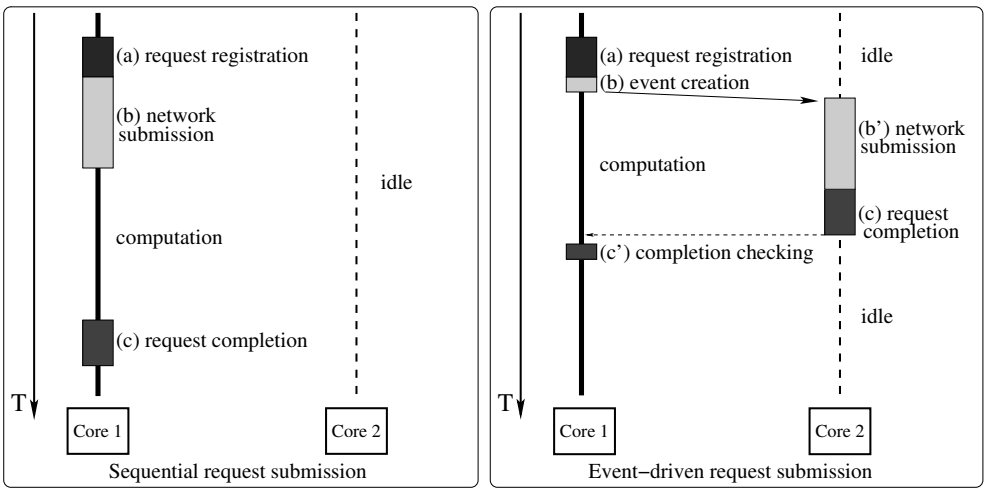

Figure 1. Asynchronous communication processing interleaved with computation.

By doing that, only necessary copies are performed and the processes to overlap are minimized.

2.3 Making *rendezvous* handshakes progress asynchronously

Another usual issue with asynchronous communication is the progression of *rendezvous* handshakes, which are typically used to implement zero-copy transfers of large messages (for instance Myrinet's MX driver uses a *rendezvous* protocol for message larger than 32kB). This zero-copy protocol consists in several packets on the wire: the sending side first sends a *rendezvous* handshake. When the receiving side is able to receive, it detects this *rendezvous* handshake, answers it and waits for the data. As soon as the sending side receives the answer, the data is sent. This way, the data is received directly on a buffer allocated by the application instead of a buffer allocated by the communication library and dedicated to unexpected messages.

A problem in this protocol is that it requires a high level of reactivity and thus the lack of progression of the handshake on one side may delay the transfer's beginning. We showed [10] that it was possible to guarantee this progression by performing a blocking system call on a dedicated thread, but this method suffers from a significant overhead.

By using an event-driven model, it is possible to split the communication process into the following operations: (a) registering the request, (b) sending/receiving the *rendezvous* request, (c) answering/waiting for the acknowledgement of the request, (d) sending/receiving the data. Again, these operations can be executed on any core. It is thus possible to make the *rendezvous* handshake progress on idle cores and to avoid the overhead due to the blocking call.

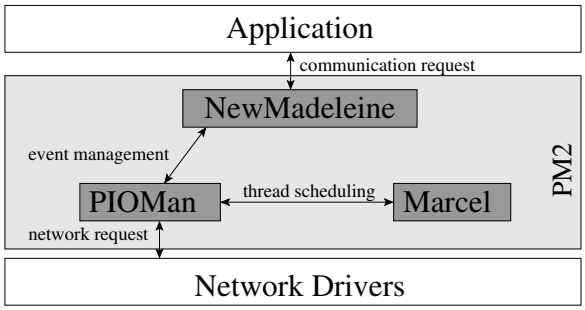

Figure 2. The PM2 software suite.

3 Highlights of the implementation

In this Section, we present the implementation of our multithreaded communication engine called PIOMAN. It has been implemented inside the PM2 software suite.

3.1 The PM2 software suite

The PM2 software suite [5] (see Figure 2) is composed of a communication library (NEWMADELEINE), a multithreading library (MARCEL), and an I/O event manager (PIOMAN).

NEWMADELEINE. Our communication library for high performance networks is called NEWMADELEINE [2]. It is available over MX/Myrinet, Verbs/InfiniBand, Elan/QsNet, and TCP/Ethernet. It aims at applying dynamic scheduling optimizations on multiple communication flows such as reordering, aggregation, multirail distribution, etc. Its running behavior is totally synchronized with the activity of

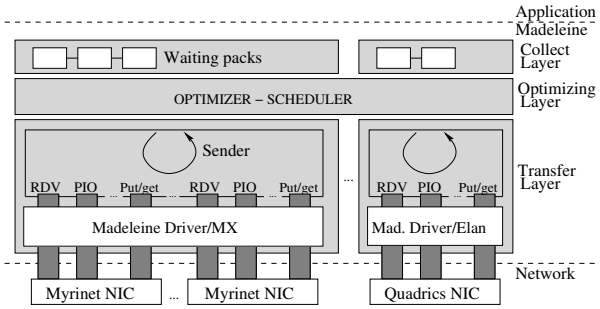

Figure 3. NewMadeleine architecture.

the underlying NICs, in contrast to more classical communication libraries which are totally driven by the application. NEWMADELEINE has a 3-layers architecture as depicted in Figure 3. The application enqueues packets into a list and immediately returns to computing. The scheduler is only activated when a NIC becomes idle in order to feed it.

MARCEL. MARCEL [6] features a two-level thread scheduler that achieves the performance of a user-level thread package while being able to exploit SMP machines. The architecture of MARCEL was carefully designed to support a high number of threads and to efficiently exploit hierarchical architectures. MARCEL extensively relies on the concept of *tasklets* [7]. Tasklets have been introduced in operating systems to defer treatments that cannot be performed within an interrupt handler. Tasklets have a very high priority, meaning that they are executed as soon as the scheduler reaches a point where it is safe to let them run.

PIOMAN. The communication engine of the PM2 software suite is PIOMAN [10]. It performs as an event detector. It aims at providing the other software components with a service that guarantees a predefined level of reactivity to I/O events. PIOMAN is able to balance the event detection processing over the whole machine and thus works closely with the MARCEL thread scheduler which provides information on the running threads and the available CPUs. This way, PIOMAN is able to choose the most appropriate method (polling or interrupt-based blocking call) depending on the context (number of computing threads, available CPUs, etc.) to ensure a high level of reactivity. PIOMAN uses the tasklet mechanism provided by MARCEL to execute detection methods on the most suitable CPUs. MARCEL also schedules PIOMAN on some triggers (CPU idleness, context switches, timer interrupts, etc.) so as to ensure a fast detection of communication events. PIOMAN is generic and is not bound to any particular network or communication library; however, in this paper we focus on the use of PIOMAN by NEWMADELEINE.

3.2 Integrating communication and computation

The detection of the completion of NEWMADELEINE's communication requests is performed by PIOMAN which provides an event detection service. It handles the multithreading issues in place of NEWMADELEINE and is in charge of balancing the events processing on the available cores. Thus, NEWMADELEINE can concentrate its efforts on communication optimizations and multiplexing without taking care of threading issues.

When PIOMAN is triggered by MARCEL, it executes callbacks provided by NEWMADELEINE that make the communication progress: poll the network interfaces, and submit new requests to the network if any is pending and the network is not busy. These callbacks are executed within tasklets in order to avoid simultaneous access to NEWMADELEINE data structures and to ensure a high level of reactivity. If a callback detects the completion of a request, it informs PIOMAN that unblocks the corresponding thread and asks MARCEL to schedule it. This way, communicating threads are ensured to be scheduled as soon as the communication event is detected.

The use of callbacks in PIOMAN makes it generic: the network-dependent code is supplied by the library using PIOMAN (namely: NEWMADELEINE), not by PIOMAN itself. NEWMADELEINE +PIOMAN already supports a large spectrum of network technologies: Myrinet, Infiniband, QsNet, and TCP.

As MARCEL schedules PIOMAN each time a core is idle, leaving a core idle will boil down to a busy waiting until PIOMAN wakes up a thread.

Offloading small message submission. We showed in Section 2.2 that transmitting small messages may take a long time and block the communicating thread for dozens of microseconds. In order to mask this delay when there are some idle cores in the system, we proceed as follows: when the application performs a non-blocking *send* for a small message, NEWMADELEINE only stores some metadata and the address of the buffer to be sent. The application then continues its computations. If a CPU becomes idle during this computation, MARCEL schedules PIOMAN that detects the pending *send* and submits the request to the network: the transfer (data copy, PIO, etc.) is performed on this idle CPU and when the application reaches the *wait* function, the message is already sent. This way, the copy is overlapped with the application computations. If the application reaches the *wait* function before the message has been submitted (every CPUs were busy), then the message is sent inside the *wait* function.

***Rendezvous* management.** When NEWMADELEINE has to perform a *rendezvous*, it submits the corresponding requests to PIOMAN in order to ensure the progression of the *rendezvous* handshake. PIOMAN then submits the requests to the network and waits the corresponding events. The detection of these events can be achieved by different methods depending on the context: if a CPU is idle, MARCEL schedules PIOMAN until a thread wakes up and acquires the CPU. Thus, PIOMAN can actively poll the network and eventually make the *rendezvous* handshake progress. When no CPU is idle, PIOMAN is obviously less intrusive and uses a blocking call on a specialized kernel thread [10] and thus detect the *rendezvous* handshake without interfering with the computing threads.

4 Evaluation

In this Section, we present the results obtained by comparing the original non-multithreaded NEWMADELEINE implementation with the PIOMAN-enabled NEWMADELEINE version. The first two programs try to overlap communication and computation for small and large messages. The last test consists in simulating an application working on a matrix that implies both computation and communication. The goal is to show that the overlap of communication and computation is possible in *"real-world"* applications such as convolution operations.

Our experiments have been carried out on a set of two dual quad-core 2.33 GHz XEON boxes with 4 GB of main memory running Linux version 2.6.22. Nodes are interconnected through MYRI-10G NICs with the MX 1.2.3 driver.

4.1 Small messages offloading

To evaluate the ability of our communication engine to overlap transfers of small messages (PIO+copy), we proceed as depicted in Figure 4: the sender performs an asynchronous send, computes during 20 μs and waits for the end of the communication. The receiver performs the same operations: it calls an asynchronous receive, computes during 20 μs and wait for the end of the communication. We measure the time required to perform all these operations which roughly corresponds to half the latency. The result is thus bounded by the computation time. The results we obtained are given in Figure 5.

The original NEWMADELEINE implementation does not overlap communication and computation at all: the time measured corresponds to $sum(communication, computation)$. The enhanced multithreaded version of NEWMADELEINE that uses PIOMAN overlaps the communication and the computation. Therefore, the measured time corresponds roughly to

```
get_time(t1);
nm_isend(len);
compute();
nm_swait();
get_time(t2);
```

Figure 4. Benchmark for small messages offloading and rendezvous progression.

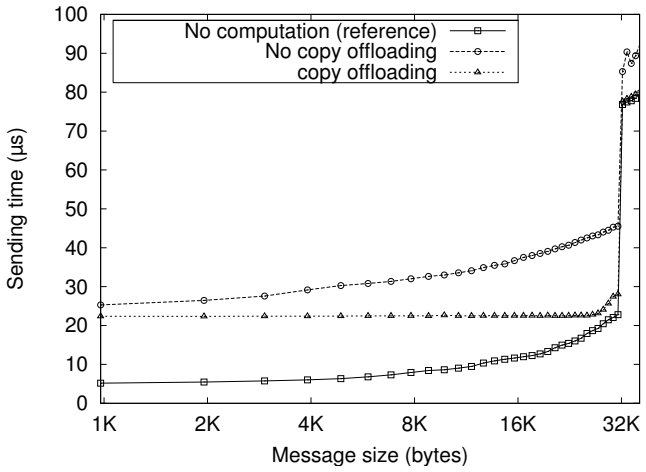

Figure 5. Small messages offloading results.

$max(communication, computation)$. When the communication time becomes equal to the computation time, we measure an overhead of 2 μs due to the communication between CPUs and the invocation of the tasklet that posts the request to the network interface.

4.2 *Rendezvous* handshake progression

We performed the same test for large data sizes, as depicted in Figure 4. This evaluates the progression of *rendezvous* handshakes and thus the overlap of communication and computation. We ran the same program with a computation time of 100 μs. The results are presented in Figure 6.

While the original NEWMADELEINE does not make the *rendezvous* handshake progress in background, and thus has a sending time of $sum(computation, communication)$, the multithreaded NEWMADELEINE version shows a full overlap of communication and computation: the measured time roughly corresponds to $max(computation, communication)$.

4.3 Real-world applications

Most MPI benchmarks only exploit a few features of the MPI API whereas some others are rarely treated. For instance, the quality of asynchronous communication meth-

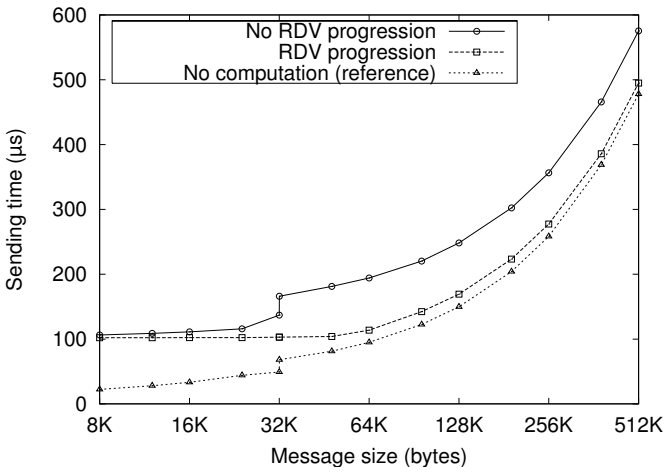

Figure 6. Offloading of rendezvous progression results.

	4 threads	16 threads
No offloading	441 μs	1183 μs
Offloading	382 μs	1031 μs
Speedup	14 %	13 %

Table 1. Impact of the number of threads on the communication offloading.

ods or the overlap of communication and computation are almost never evaluated. Yet, broad distributed *"real-world"* applications could benefit from efficient asynchronous communication methods. For instance, it was shown [8] that such applications could run significantly faster by using an MPI implementation that ensures a high level of communication and computations overlap. Other types of application can benefit from our multithreaded communication engine: applications that massively communicate through asynchronous methods should substantialy profit by the progression of communication provided by PIOMAN. Moreover, irregular applications that use asynchronous communication primitives should benefit from the copy offloading.

An adaptation of such applications to our communica-

```
get_time(t1);
compute1();
nm_isend();
compute2();
nm_swait();
nm_recv();
get_time(t2);
```

Figure 7. Benchmark application

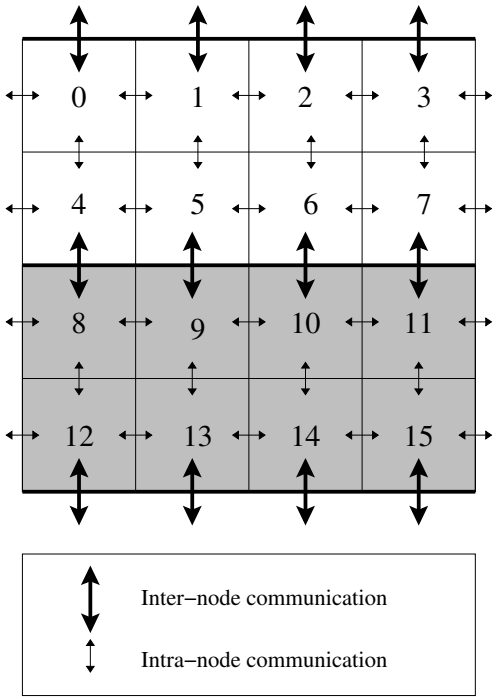

Figure 8. Communication scheme of the meta-application

tion engine is still in progress. In order to evaluate our implementation's behavior in *"real-world"* applications, we designed a meta-application that mimics the communication scheme of an application performing operations on a matrix such as convolutions. This program launches one MPI process per node of a cluster. Each process creates threads that compute a part of the matrix. The distribution of the threads on the matrix is shown in Figure 8. As the program performs a classical convolution-like operation, we proceed as depicted in Figure 7: each thread first computes its frontiers and send asynchronously the result to its neighbors. It then computes the remaining part of its domain and waits for its neighbors' results. This generates both intra-node and inter-node communication requests which are either submitted to the network (inter-node requests) or to a shared-memory channel. The message size is lower than the *rendezvous* threshold and thus we evaluate the effect of the copy offloading on the application execution time.

We measured the execution time for two configurations: the first one consists in distributing 4 threads over 2 nodes (*i.e.* 2 threads per node) whereas the second one (depicted in Figure 8) runs 16 threads over 2 nodes (*i.e.* 8 threads per node). The results we obtained are given on Table 1.

The original NEWMADELEINE exhibits an execution time of 441 μs when running 4 threads whereas the multi-

threaded version runs in 382 μs which represents a speedup of 14%. This speedup is due to the distribution of the threads over the nodes: each node has 8 cores which run 4 threads, leaving 4 idle cores. These idle cores actually keep on trying to offload the communication requests, avoiding a delay on the computing threads due to a buffer copy when performing an isend.

When running 16 threads, the original NEWMADELEINE shows an execution time of 1183 μs whereas the multithreaded version runs in 1031 μs which corresponds to a speedup of 13 %. The raise of the execution time between the 4-threads and the 16-threads can be explained by the size of the matrix which is 4 time bigger in this case. Due to the matrix size, the amount of data to be transfered increase and the communication time requiered grows. The speedup we get is roughly the same as the one we get with the 4-threads version. This is due to the ability of PIOMAN to fill the gap left by the thread scheduler when a thread waits for its neighbours' data: this gap is used to offload pending communication requests. When a thread waits for the end of a communication, there is a chance that the data transfer has been offloaded and thus the thread does not have to wait.

5 Conclusion and Future Work

The architecture of typical clusters nodes is evolving from single or dual CPU machines to multicore with 8, 16, or more cores. It radically changes the approach to communication management. With the move from single MPI applications to multithreaded MPI applications, communication management cannot ignore threads anymore.

This paper presents the design and implementation of a powerful event-driven communication engine that exploits multicore architectures to speed up communications within hybrid multithreaded applications. It is able to utilize efficiently idle cores to offload CPU-hungry PIO communication and to make *rendezvous* actually progress in background in an opportunistic way. We implemented this design inside the PIOMAN communication manager from the PM2 software suite. Our evaluation shows that it actually speeds up communication overlapping with computation by carefully exploiting underutilized resources.

This contribution opens the path to future works such as executing NEWMADELEINE optimization algorithms in background as PIOMAN events for adaptive packet scheduling optimization depending on the CPU availability. We still need some benchmarks of our multithreaded communication engine with real applications that actually mix multithreading and message passing. There are still investigations to be done on an adaptive strategy to choose whether to offload communication or not. Finally, we plan to integrate this multithreaded communication engine in MPICH2 through a collaborative work with Argonne National Laboratory.

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

References

[1] ANL, MCS Division. MPICH-2 Home Page, 2007. `http://www.mcs.anl.gov/mpi/mpich/`.

[2] 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*, March.

[3] 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.

[4] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. Design and implementation of high performance mvapich2: Mpi2 over infiniband. In *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, Singapore, May 2006.

[5] R. Namyst and J.-F. Mhaut. PM2: Parallel Multithreaded Machine; A computing environment for distributed architectures. 1995.

[6] Runtime Team, LaBRI-Inria Futurs. Marcel: A POSIX-compliant thread library for hierarchical multiprocessor machines, 2007. `http://runtime.futurs.inria.fr/marcel/`.

[7] P. Russel. Unreliable guide to hacking the linux kernel, 2000.

[8] J. C. Sancho and *et al.* Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC06*, Tampa, FL, 2006. IEEE Computer Society.

[9] L. Smith and M. Bull. Development of mixed mode mpi / openmp applications. *Scientific Programming*, 9(2-3/2001):83–98. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.

[10] 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.

[11] T. Viet, T. Yoshinaga, Y. Ogawa, B. Abderazek, and M. Sowa. Optimization for Hybrid MPI-OpenMP Programs on a Cluster of SMP PCs. *Japan-Tunisia Workshop on Computer Systems and Information Technology*, 2004.