Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager

François Trahay, Alexandre Denis, Olivier Aumage, Raymond Namyst

Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager

François TRAHAY, Alexandre DENIS, Olivier AUMAGE, and Raymond NAMYST

INRIA, LaBRI, Université Bordeaux 1
351, cours de la Libération
F-33405 TALENCE, France
{trahay, denis, aumage, namyst}@labri.fr

Abstract. MPI applications may waste thousands of CPU cycles if they do not efficiently overlap communications and computation. In this paper, we present a generic and portable I/O manager that is able to make communication progress asynchronously using tasklets. It chooses automatically the most appropriate communication method, depending on the context: multi-threaded application or not, SMP machine or not. We have implemented and evaluated our I/O manager with Mad-MPI, our own MPI implementation, and compared it to other existing MPI implementations regarding the ability to efficiently overlap communication and computation.

Keywords: Polling, Interrupt, Thread, Scheduler, High-Speed Network

1   Introduction

Asynchronism is becoming ubiquitous in modern communication runtimes. This evolution is the combined result of multiple factors. *Firstly*, communication subsystems implement increasingly complex optimizations in order to make better use of networking hardware. As we have shown in [1], such optimizations require online analysis of the communication schemes and hence require the de-synchronization of the communication request submission from its processing. *Moreover*, providing rich functionality such as communication flow multiplexing or transparent multi-method, heterogeneous networking implies that the runtime system should again take an active part in-between the communication request submit and processing. And *finally*, overlapping communication with computation and being reactive actually do matter more now than it has ever done [2,3]. The latency of network transactions is in the order of magnitude of several thousands CPU cycles at least. Everything must therefore be done to avoid independent computations to be blocked by an ongoing network transaction. This is even more true with the increasingly dense SMP, multicore, SMT (also known as Intel's Hyperthreading) architectures where many computing units share a few NICs.

Since portability is one of the most important requirements for communication runtime systems, the usual approach to implement asynchronous

processing is to use threads (such as Posix threads). Popular commu-
nication runtimes indeed are starting to make use of threads internally
and also allow applications to be multithreaded as it can be seen with
both MPICH-2 [4], and Open MPI [5,6]. Low level communication li-
braries such as Quadrics' Elan [7] and Myricom's MX [8] also make use
of multithreading. Such an introduction of threads inside communication
subsystems is not going without troubles however. The fact that multi-
threading is still usually optional with these runtimes is symptomatic
of the difficulty to get the benefits of multithreading in the context of
networking without suffering from the potential drawbacks.

In this paper, we analyze the two fundamental approaches of integrating
multithreading and communications —interrupts and polling. We study
their respective benefits and their potential drawbacks, and we discuss
the importance of the cooperation between the asynchronous event man-
agement code and the thread scheduling code in order to avoid such dis-
advantages. We then introduce our proposal for symbiotically combining
both approaches inside a new generic network I/O event manager. The
paper is organized as follows. Section 2 exposes the problem of integrat-
ing threads and communications. Section 3 introduces our proposal for
a new asynchronous event management model and gives details about
our implementation. We evaluate this implementation in Section 4 and
Section 5 concludes and gives an insight of ongoing and future work.

2 Integrating threads and communication: the problems of network I/O events management

The detection of network I/O events can be achieved by two main strate-
gies. The most common approach consists in using the *active waiting*:
a polling function is called repeatedly until a network I/O event is de-
tected. The polling function is usually inexpensive, but repeating this
operation thousands of times may be prohibitive. The other method for
detecting communication events is the *passive waiting* which is based
on blocking calls. In that case, the NIC informs the operating system
that a network I/O event has occurred by using an interrupt, making
this method much more reactive than polling. However this operation
involves interrupt handlers and context switches which are rather costly.
The best method to use depends on the application, but in both cases,
some behaviors may lead to suboptimal performance. When using interrupt-
based methods, priority issues may occur: the thread that is waiting for
the communication event may be scheduled with some delay. This is the
case when, for example, it has been computing for a long period before it
blocks, lowering its priority. Moreover, the system has to support meth-
ods to detect the network I/O events. For instance, in a pure user-level
scheduler, interrupt-driven blocking calls are prohibited (unless a specific
OS extension like the *Scheduler Activations*[9] is used).

Using polling methods can also be problematic: if the system is over-
loaded (i.e. there are more running threads than available CPUs), the
polling thread may scarcely be scheduled, thus increasing the reaction
time. Moreover, some asynchronous communication operations need a

regular polling in order to progress. For instance, a *rendez-vous* requires a regular polling so that the preliminary phase makes progress. As it is shown in [2], some applications would significantly improve their execution time by efficiently overlapping communication and computation, which requires to poll communication events regularly.

3 An I/O manager model

To resolve these kinds of problems, we propose an I/O manager that provides the communication runtime systems with a network event detection service. Thus, communication libraries themselves become independent of the multithread issues and related hardware issues such as the number of CPUs. Thereby, they can focus their efforts on communication optimizations and other functionalities. By working closely with a specific thread scheduler, the I/O manager can be viewed as a progression engine able to schedule a communicating thread when needed or to dynamically adapt the polling frequency to maximize the reactivity/overhead ratio. The I/O manager handles both polling and interrupt-based methods, switching from one method to another depending on the context.

The implementation of our I/O manager called PIOMan (PM2 I/O Manager) relies on a two-level thread scheduler [10] which was slightly modified to interact with the I/O manager when necessary. The use of a two-level scheduler allows to precisely control thread scheduling at the user level, with almost no explicit (and expensive) interaction with the OS. This way, we can dynamically favour the scheduling of a thread requiring a high reactivity to communication events during a fixed period. PIOMan is available as three main versions: no-thread, mono (user-level threads) or SMP (user threads on top of kernel threads).

3.1 Overview of the I/O manager

The mechanism of our I/O manager is described through an example shown in Figure 1: the application first registers a callback function for each event type to detect. When the application starts a communication, it can submit the requests to poll (1) and wait for them or simply continue its computation. Periodically, the thread scheduler calls the I/O manager (2) in order to poll the network by calling the callback functions (3).

We propose to manage the communication events in a dedicated controller linked to the thread scheduler for several reasons. Firstly, centralizing avoids the concurrency issues encountered when several threads try to poll the same network. Since the I/O manager has a global view of the pending requests, it can poll each request one after another. Moreover, the manager has the opportunity to aggregate multiple requests. If several threads are waiting for messages on a single network interface, it can be interesting to aggregate these requests when polling.

Secondly, the thread scheduler has the opportunity to preempt a computing task and call the I/O manager in order to detect a potential network
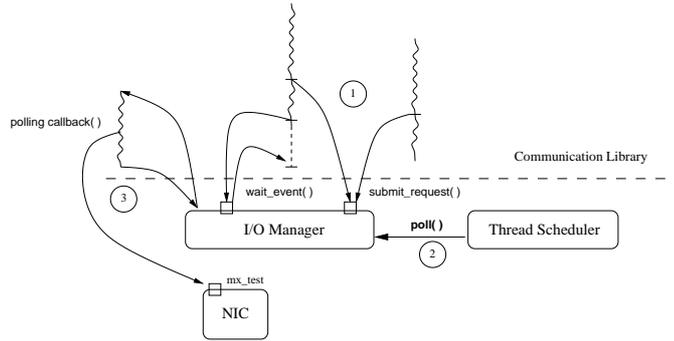
Fig. 1. Example of interaction between the I/O manager and the MPI library.
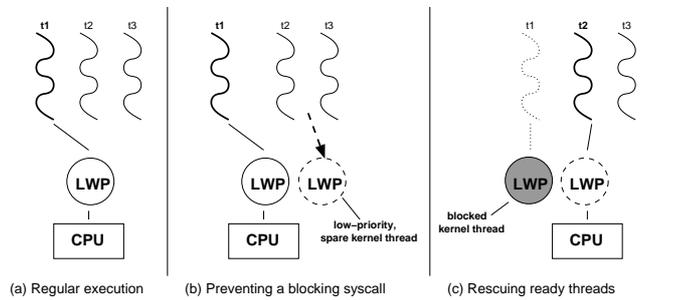

Fig. 2. Low priority, spare kernel-level threads are used to schedule remaining application threads in case a blocking syscall occurs during an I/O critical operation.

I/O completion and thus make the communication progress. This is useful when the application performs asynchronous operations that require some processing once the communication ends. For example, in a *rendezvous* protocol, the receiver has to post a receiving request to synchronize with the sender. Once both sides are synchronized, the transfer can start: one side receives the data that the other side sends. In that case, the progression offered by the I/O manager and the thread scheduler allows to completely overlap the communication with computation.

3.2 Passive waiting: interrupts

Passive monitoring through blocking system calls is tricky to implement in a two-level scheduler. Indeed, during regular execution of application threads, our scheduler binds exactly one kernel thread (also called LightWeight process – LWP) per processor (Fig. 2-a), so that the scheduling of threads can be entirely performed at the user-level. A blocking system call could therefore prevent a whole subset of user-level threads to run. To avoid this and keep reactivity low, we proceed as follows.

Before executing a (potentially blocking) I/O system call, the client thread first wakes up *a spare kernel thread* (Fig. 2-b) to shepherd the

remaining ready threads on the underlying processor. Since this kernel thread runs at a very low priority, it will not be scheduled until the previous kernel thread blocks. Thus, *if the system call completes without blocking*, the I/O client will continue its execution with a very high priority, as requested. At the end of the I/O section, the spare kernel thread simply returns to the "sleep" state. On the opposite, *if the call blocks*, the original kernel thread yields the CPU to the spare one (Fig. 2-c). Upon I/O completion, the NIC interrupt handler will wake up the original kernel thread that will, in turn, immediately continue the execution of the client thread. This way, the reactivity of the client thread is optimal.

Note that no modification to the underlying operating system is required, as opposed to solutions such as *Scheduler Activations* [9,11].

3.3 Active waiting: polling

In implementing active polling, our system carefully cooperates with the thread scheduler to avoid busy waiting and unnecessary context switches. Applications register new types of I/O events with some polling trigger(s) (at every context switch, after a period of time, when a CPU gets idle, etc.) The thread scheduler then invokes the I/O manager accordingly. However, these invocations occur in a restricted context with some classes of actions being prohibited (synchronization primitives, typically). Thus, they are similar to interrupt handlers within an operating system.

Most of the I/O manager code is consequently run *outside* the restricted context in the form of *tasklets* [12]. Tasklets have been introduced in operating systems to defer treatments that cannot be performed within an interrupt handler. They run as soon as possible (they have a very high priority) when the scheduler reaches a point where it is safe to run tasklets. They have additional properties. Firstly, tasklets of the same type run under mutual exclusion, which simplifies the I/O manager code and even makes it more efficient. Secondly, the execution of tasklets can be enforced on a particular processor, which allows to maximize cache affinity by running tasklets on the same processor as their client thread.

3.4 Handling of both interrupts and polling

Most of the network interfaces (MX/Myrinet, Infiniband Verbs, TCP sockets) provide both polling and interrupt-based functions to detect network I/O events. To ensure a good reactivity, our I/O manager uses one method or the other depending on the context: number of running threads and available CPUs. This kind of strategy has already been developed in Panda [13], but ours also takes into account the upper layer's preference: the communication library or the application has full knowledge of the request completion time. A smarter approach could also take into account the history of requests or their priorities. A similar method was developed in *polling watchdog*[14] but it required a specific kernel support.

Table 1. Benchmark program for MPI asynchronous progression.

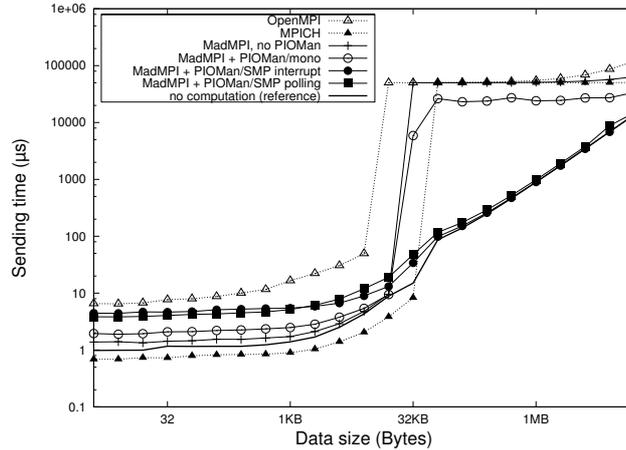| Sender	Receiver

`get_time(t1);`
`MPI_Send(...);`
`get_time(t2);`
	`MPI_IRecv(...);`
	`compute();`
	`    /* approx. 50ms. computation */`
	`MPI_Wait(...);`

Fig. 3. `MPI_Send` time with MX.

4 Evaluation

We have evaluated the implementation of our I/O manager using the NewMadeleine [1] communication library and its built-in MPI implementation called Mad-MPI. The point-to-point nonblocking posting (`isend`, `irecv`) and completion (`wait`, `test`) operations of Mad-MPI are directly mapped to the equivalent operations of NewMadeleine. We performed benchmarks that evaluate the MPI asynchronous operation progression in background (communication/computation overlap) and benchmarks that evaluate the overhead of PIOMan. All these experiments have been carried out on a set of two dual-core 1.8 GHz Opteron boxes interconnected through Myri-10G NICs with the MX1.2.1 driver providing a latency of $2.3\mu s$.

MPI asynchronous progression of communications. To evaluate the MPI asynchronous progression, we use the benchmark program listed on Table 1. This program attempts to overlap communication and computation on the receiver side. We record the time spent in sending and we compare the results to a reference obtained with Mad-MPI.
Figure 3 shows the sending time (time spent in `MPI_Send`) we measured over MX/Myrinet with Mad-MPI, OpenMPI 1.2.1, and MPICH/MX 1.2.7. We measured similar results over other network types (Infiniband and TCP). For small messages, all implementations show a sending time close

Table 2. PIOMan's average overhead.

	no thread	mono	SMP
polling	0.038 μs	0.085 μs	0.142 μs
interrupt	-	-	1.68 μs

to the network latency. For larger messages, when a *rendez-vous* is performed, we observe three different behaviors:

no asynchronous progress – OpenMPI and plain Mad-MPI do not support background progress of *rendez-vous* handshake. Therefore, the sender is blocked until the receiver reaches the `MPI_Wait`. MPICH makes the handshake progress thanks to the MX progression thread but in the current implementation, the notification of the transfer is not overlapped.

coarse grained interleaved progress – PIOMan/mono tasklets are scheduled upon timer interrupt, every 10 ms. We observe that the delay to complete the *rendez-vous* is now bounded by 10 ms instead of the full computation time.

full overlap – PIOMan/SMP is able to schedule tasklets on another LWP, thus we get a full overlap of communication and computation. We observe on the figure that the *rendez-vous* performance does not suffer from the computation on the receiver side.

We conclude that PIOMan is able to actually overlap MPI communication and computation while OpenMPI, MPICH, and plain Mad-MPI were not able to make communication progress asynchronously.

Overhead evaluation. We have evaluated the overhead of the I/O manager with empty polling and blocking functions. The results are shown in Table 2. The polling overhead differs from one version to the other. This is due to the cost of synchronization being different over each version. The interrupt overhead has only been evaluated on the SMP version since only this version implements the mechanism. We observe that the overhead is negligible for polling. On the other hand, the cost of blocking calls (interrupts) is quite high due to the awakening of the sleeping LWP and the communication between LWPs. However, interrupts are supposed to be used when the CPU is doing computation, where the delay would have been several order of magnitude higher without interrupts.

5   Conclusions and Future Work

Overlapping MPI communications and computation do matter if we do not want to waste thousands of CPU cycles. However, making communications progress efficiently is not so simple as adding a communication thread. In this paper, we have proposed a generic and portable communication events manager that is able to actually overlap communication and computation. This I/O manager is able to handle both active polling and interrupts and integrates gracefully with our multithreading scheduler.

We obtained effective communication/computation overlapping with our I/O manager, as opposed to other widespread MPIs.

In the near future, we plan to use PIOMan inside other MPI implementations such as MPICH-2 or communication frameworks like PadicoTM. We also intend to make a more efficient use of NUMA architectures by trying to execute polling tasklets on the most suitable CPU given the architecture topology.

References

1. Aumage, O., Brunet, E., Furmento, N., Namyst, R.: 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. Sancho, J.C., *et al.*: Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In: SC06, Tampa, FL, IEEE Computer Society (2006)
3. Doerfler, D., Brightwell, R.: Measuring MPI send and receive overhead and application availability in high performance network interfaces. In: Euro PVM/MPI. (2006) 331–338
4. ANL, MCS Division: MPICH-2 Home Page (2007) http://www.mcs.anl.gov/mpi/mpich/.
5. The Open MPI Project: Open MPI: Open Source High Performance Computing (2007) http://www.open-mpi.org/.
6. Graham, R.L., *et al.*: Open MPI: A high-performance, heterogeneous MPI. In: Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Barcelona, Spain (September 2006)
7. Quadrics Ltd.: Elan Programming Manual (2003) http://www.quadrics.com/.
8. Myricom Inc.: Myrinet EXpress (MX): A High Performance, Low-level, Message-Passing Interface for Myrinet (2003) http://www.myri.com/scs/.
9. Anderson, T.E., Bershad, B.N., Lazowska, E.D., Levy, H.M.: Scheduler activations: effective kernel support for the user-level management of parallelism. ACM Trans. Comput. Syst. **10**(1) (1992) 53–79
10. Runtime Team, LaBRI-Inria Futurs: Marcel: A POSIX-compliant thread library for hierarchical multiprocessor machines (2007) http://runtime.futurs.inria.fr/marcel/.
11. Danjean, V., Namyst, R., Russell, R.: Integrating kernel activations in a multithreaded runtime system on Linux. In: Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00). (2000)
12. Russel, P.: Unreliable guide to hacking the linux kernel (2000)
13. Langendoen, K., Romein, J., Bhoedjang, R., Bal, H.: Integrating polling, interrupts, and thread management. frontiers **00** (1996) 13
14. Maquelin, O., *et al.*: Polling watchdog: combining polling and interrupts for efficient message handling. In: ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture. (1996)