A scalable and generic task scheduling system for communication libraries

François Trahay, Alexandre Denis

A scalable and generic task scheduling system for communication libraries

Francois TRAHAY #1, Alexandre DENIS *2

University Bordeaux 1/LaBRI,
351 cours de la Liberation F-33405 Talence, France
1 francois.trahay@labri.fr

* INRIA Bordeaux – Sud-Ouest/LaBRI,
351 cours de la Liberation F-33405 Talence, France
2 alexandre.denis@labri.fr

***Abstract*—Since the advent of multi-core processors, the physionomy of typical clusters has dramatically evolved. This new massively multi-core era is a major change in architecture, causing the evolution of programming models towards hybrid MPI+threads, therefore requiring new features at low-level. Modern communication subsystems now have to deal with multithreading: the impact of thread-safety, the contention on network interfaces or the consequence of data locality on performance have to be studied carefully.**

In this paper, we present PIOMan, a scalable and generic lightweight task scheduling system for communication libraries. It is designed to ensure concurrent progression of multiple tasks of a communication library (polling, offload, multi-rail) through the use of multiple cores, while preserving locality to avoid contention and allow a scalability to a large number of cores and threads. We have implemented the model, evaluated its performance, and compared it to state of the art solutions regarding overhead, scalability, and communication and computation overlap.

I. INTRODUCTION

Cluster architecture has dramatically evolved since the introduction of multicore technology a few years ago. Where typical clusters used to be comprised of dual-processor nodes not so long ago, 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. For example Intel announces the 8-core *Nehalem-EX* for late 2009. An 8-way motherboard with such processors will lead to 64 cores per node. It marks the advent of the *massively multicore* era.

Thus, the approach to program clusters has to evolve. The classical "pure MPI" model, with one process per core, 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, this hybrid solution introduces a mix of threads and communication which is not straightforward at low level.

Mixing threads and networking requires some precautions in communication libraries so as to avoid race conditions when threads access concurrently the library [1]. Moreover, we have shown that the communication library itself may benefit from multi-threading [2], for example to make non-blocking communication actually progress in background and overlap with computation.

In this paper, we study the scalability, with regard to the number of cores and threads, of such mechanisms used when mixing threads and network communication, and we propose a scalable communication engine.

The remaining of this paper is composed as follows. Section II analyzes the interactions between communication and threads which serve as a motivation to our work. Section III presents our design for a scalable and generic communication engine. Section IV gives some outline of our implementation. Section V presents a performance evaluation of our implementation. Finally, Section VI draws a conclusion of this study and shows directions for further work.

II. MOTIVATION AND ANALYSIS

With the increase of the number of cores per nodes in clusters, the scalability of high performance runtime systems that mix communication and threads becomes a major concern. Such hybrid solutions become widespread due to the use of multicore chips in clusters, but these new paradigms introduce serious problematics to runtime developpers as processors are getting massively multicore. The increase of the number of communication flows leads to more pressure on communication libraries as well as on hardware resources (NICs, memory bus, etc.)

A. Concurrent communication

Supporting concurrent accesses to a communication library is usually achieved with a low amount of work. A global mutex is used to prevent concurrent accesses that would otherwise cause data corruption. As hybrid approaches that mix MPI and threads become widespread, applications now need the `MPI_THREAD_MULTIPLE` thread-safety level. This level of

thread-safety is provided by some modern MPI implementations such as OPENMPI [3] or INTEL MPI. However, handling several concurrent accesses efficiently requires more precise approaches in order to avoid competition. The pressure on data structures highly depends on the granularity of critical sections (the portions of codes that have to be protected by a lock). Fine-grain locking techniques [1] or lock-free algorithms [4] have to be applied to communication libraries in order to achieve good concurrency

The increasing number of cores also implies concurrency issues at the hardware level: the communication flows as well as the total amount of communication increase and network interface cards (NICs) are intensively used. Multirail clusters permit to reduce the pressure on NICs by extending the cumulated bandwidth [5]. Even within multirail clusters, the number of NICs did not grow up as quickly as the number of cores, so each NIC still has to be shared by several cores. Communication libraries thus have to be able to multiplex communication flows on top of the multirail network. Multiplexing multiple communication flows paves the way to complex optimizations such as packet aggregation or messages reordering. Viewing multiple messages as buffers to send may help: messages can be grouped into pools of packets that have to be sent to the same destination. Optimizations are then applied to each pool and the packets of data transmitted to the NICs are unrelated from the application messages as it is illustrated on Figure 1. Having a global view of communication flows also permits to arbitrate access to the NICs. We showed that, instead of passing application messages to the NICs directly, buffering packets and applying optimizations improve throughput and avoid NICs saturation [6].

The current development of multicore chips as well as NUMA architectures also implies locality issues. The placement of tasks influences the performance of network on NUMA machines [7]. Even on SMP architectures, the network performance depends on cache effects [1]. It is thus important to take locality into account when designing a modern communication library.

B. Overlapping communication and computation

Multicore chips are an opportunity to make communication progress in the background, and thus to overlap communication and computation. Such optimization permits to hide communication and thus to reduce the global execution time [8]. Using progression threads to make *rendezvous* handshake progress is a widely used technique to achieve overlap. Actually, communication libraries such as Myrinet MX [9] and some MPI implementations such as OPENMPI ensure the progression of communication in background through this mechanism.

RDMA-enabled NICs can also be utlized to make communication and computation overlap. For instance, RDMA-Read methods allow to cut one message in the *rendezvous* handshake and permit to seamlessly overlap communication and application computation [10] on the sender side. However this technique only works on a given set of hardware that implements RDMA and only allows to overlap at the sender side.

The increase of the number of cores per nodes in cluster leads to thread scheduling side effects: more cores implies longer intra-node synchronization. These synchronization issues ofte leave holes in thread scheduling. We showed [2] that it is possible to exploit these holes to make the communication library progress. *Rendezvous* handshake or packet submission to the NICs may be offloaded to idle cores so as to perform these operations in the background and thus to allow communication and computation to overlap.

A similar approach consists in offloading a part of the protocol processing onto specialized NICs [11]. Such a technique could be used to handle the *rendezvous* handshake at the hardware level, allowing to overlap communication and computation. However, the need for specialized hardware makes this technique highly non-portable.

In a general manner, the current evolution of clusters architecture is an opportunity for runtime libraries to take advantage of multithreading instead of only supporting it. We thus have to use idle cores to process communication treatments while taking locality into account. Designing a system that permits to execute tasks (such as polling a network, replying to a *rendezvous* handshake, etc.) and to preserve data locality would allow communication libraries to exploit efficiently multicore machines.

III. DESIGN OF A SCALABLE LIGHT TASK SCHEDULING SYSTEM

In this section, we present the design of our scalable and generic task scheduling system for communication libraries.

The increase of the number of concurrent communication flows leads to a complexification of communication libraries processing as explained in Section II. We propose to offload the handling of network events and of network submission to a specialized task manager. The communication library becomes independent of the multithreading issues and can focus its efforts on the optimization of data flows or other functionnalities.

The task manager provides the communication library with a light task scheduling service: when the communication library has to execute a task (polling a network, submitting a packet to the network, etc.), it delegates it to the task manager. A task consists in running a function with a given parameter. A CPU set is attached to the task so as to avoid unwanted cores to execute it. As some treatments need to be performed repeatedly (polling a network for example), an option is also added to a task.

In order to execute tasks when a core is idle, hooks are inserted in the thread scheduler. This way, the task manager is called during scheduling holes. This permits to execute tasks during CPU idleness. However, if threads use the CPUs constantly without blocking, the task manager may not be called. This would result in deadlocks since polling is not performed anymore. Adding hooks at other keypoints of the

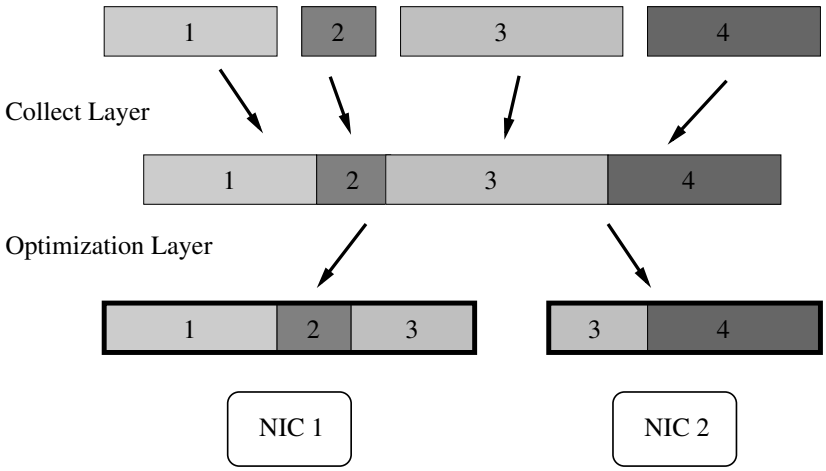


Fig. 1. Multiplexing messages is an opportunity for cross-flow optimizations

thread scheduling such as timer interrupt or context switches permits to ensure a progression of communication.

Since the communication library offloads the detection of network events, the corresponding tasks have to be stored by the task manager. A naive solution consists in maintaining a global list of tasks to be executed. Since this list may be modified concurrently, it has to be protected either through a mutex or through a lock-free mechanism. However, this big-lock technique is likely not to scale up, which is a critical issue in the current multicore era. Many thread may modify the list by executing tasks or by adding new tasks. The contention on the mutex implies a prohibitive overhead. Storing all the tasks in a single list also prevents from preserving tasks affinity.

A. Hierarchical queues

In order to provide a scalable task management, it is mandatory to use a hierarchy of task lists that can be viewed as a tree of tasks [12]. This hierarchy is directly mapped onto the machine architecture as depicted on Figure 2. For a given task, the list in which it is inserted expresses the scheduling area: a task in a *Per-Core Queue* is only executed by the corresponding core; if it is inserted in a *Per-Cache Queue*, any core that shares the corresponding cache may process this task. The same tree structure is applied to the whole machine and the resulting tree may be composed of *Per-Core Queues*, *Per-Cache Queues*, *Per-Chip Queues*, *Per-NUMA Node Queues* and a *Global Queue*, depending on the machine architecture. Each of these lists has to be protected against concurrent access. Lock-free algorithms or mutexes can be used to implement this.

When the communication library submits a new task, it provides the task manager with a CPU set that expresses the list of CPUs that are allowed to execute the task. This CPU set is examinated to find the corresponding task queue and the task is inserted in this list.

When the thread scheduler reaches a keypoint, it calls the TASK_SCHEDULE function. This function executes Algorithm 1: it runs the tasks stored in the *Per-Core Queue* that corresponds to the local CPU. When all the tasks from the list have been executed, the upper queue is selected and its tasks are processed.

Algorithm 1 Task_Schedule

```
for Queue = Per_Core_Queue to Global_Queue do
  Task ⇐ Get_Task(queue)
  while Task ≠ NULL do
    run(Task)
    if OptionRepeatIsSet(Task) then
      Enqueue(Queue, Task)
    end if
  end while
end for
```

The Algorithm 2 is used for selecting a task from a list. The content of the queue is first evaluated without holding the mutex in order to avoid unnecessary contention. If the queue is not empty, its mutex is acquired and the list state is checked once again. This technique permits to avoid race conditions with a minimal overhead since the mutex is only held when the list contains tasks.

Algorithm 2 Get_Task(Queue)

```
Result ⇐ NULL
if notempty(Queue) then
  LOCK(Queue)
  if notempty(Queue) then
    Result ⇐ dequeue(queue)
  end if
  UNLOCK(Queue)
end if
return Result
```

Using hierarchical queues permits to preserve the tasks affinity since each task may be processed on a CPU specified by the communication library. The hierarchical lists also

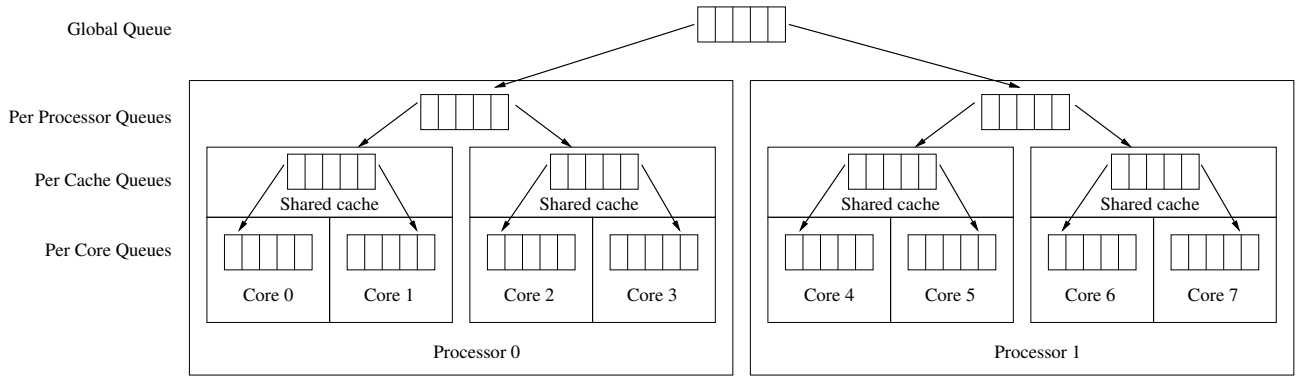


Fig. 2. Hierarchical lists mapped to a machine topology

reduce contention on queues mutexes since the locks are acquired locally most of the time.

IV. Implementation

In this Section, we present the implementation of our scalable task manager. It has been implemented in the PIOMan I/O Manager [13]. The mechanism of task offloading is exploited by the NewMadeleine communication library [6].

A. PIOMan

PIOMan is the progress engine of the PM2 software suite. 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. It works closely with the Marcel thread scheduler which provides information on the running threads and the available CPUs. Marcel schedules PIOMan on some triggers (CPU idleness, context switches, timer interrupts, etc.) so as to ensure a fast detection of communication events.

Since Marcel uses a hierarchical structure to depict the machine architecture, PIOMan inserts task queues into Marcel topology levels. In order to protect the list from concurrent access, PIOMan uses spinlocks. Indeed, a thread that modifies a list enters the corresponding critical section for a very short period, less than the time required to perform a context switch. Using a classical mutex or a semaphore to ensure the list consistency would imply a risk of costly context switches. On the contrary, using spinlocks to ensure the mutual exclusion garantees a fast access to the list.

When the communication library submits a new task, it first initialize the task structure and fill the CPU set according to the task affinity. As the task is submitted, the topology node that corresponds to the CPU set is determined and the associated queue is chosen. The list mutex is hold and the task is enqueued into the list.

When the thread scheduler reaches a key point – that is when a CPU is idle, when a context switch occurs or during timer interrupts – PIOMan is called so that tasks can be executed. PIOMan first processes local tasks (*i.e.* tasks from the *Per-Core Queue* that corresponds to the current CPU) and scans upper queues until it reaches the *Global Queue*. Due to Algorithm 2 empty lists do not require to be locked, reducing contention. When the processing of a repetitive task ends, the task is re-enqueued into the same list.

B. NewMadeleine

Our communication library for high performance networks is called NewMadeleine [6]. 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.

The multithreading subsystem as well as the handling of networks events of NewMadeleine is based on PIOMan task manager. NewMadeleine uses tasks to poll networks for incoming or outcoming messages. These tasks are repetitive: it is considered completed once the corresponding network polling succeeds. In order to maintain polling affinity, the CPU set attached to these tasks contains the cores that share a cache with the current CPU.

NewMadeleine also exploits tasks to submit new requests to the network. In a previous paper [2] we showed that this permits to overlap communication and computation even for small messages. The submission of requests to a network works as follows: the state of each core is evaluated in order to find an idle core that could process the task. If such a core is found, a task is created and the nearest idle core is specified in the CPU set. This way, the communication may overlap the computation while minimizing the cache effects as well as the contention on the list lock. If all the cores are busy, the task is created in the *global queue*. When a core becomes available – because a thread enters a blocking section or because a thread waits for the end of the communication – the task is processed and the communication may overlap the computation. If the submission of the request to the network does not succeed immediately (*i.e.* the request completion needs to be polled), a polling task is submitted to PIOMan. It is to be noticed that the task structure does not require an allocation since it is included in the *packet wrapper* structure that NewMadeleine uses to represent the data to transmit to the network.

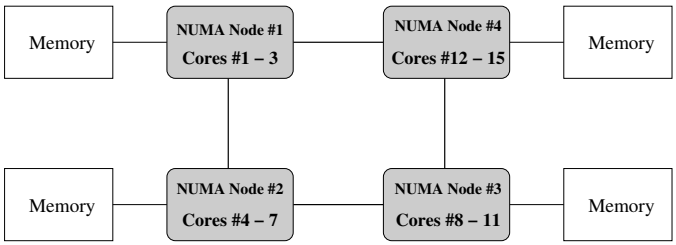


Fig. 3. Topology of Kwak

As an extension, NEWMADELEINE could use the task mechanism in other cases. For now, optimizations on communication flows are computed when a NIC becomes idle and the computation have to be done quickly in order to minimize the optimization overhead. NEWMADELEINE could create a task that pre-computes all the available optimization scores. As soon as a NIC becomes idle, the best optimization could be applied to communication flows without paying the optimization overlap. Idle cores could also be used to exploit efficiently slow networks or grid configurations: tasks could be created to apply data filters such as data compression, encryption or encoding/decoding.

The combinaison of PIOMAN tasks and NEWMADELEINE fine-grain locking permits to process communication operations in parallel. For instance, NEWMADELEINE is able to poll several networks concurrently or to poll a network while submitting a new request to it. The use of idle cores to execute communication tasks allows to take advantage of scheduling holes to make communication progress in the background transparently. Overlapping communication and computation is then achieved automatically.

V. EVALUATION

In this Section, we evaluate the performance and scalability of PIOMAN. The raw performance of task scheduling is evaluated before comparing our implementation to other MPI implementations on scalability and overlapping benchmarks.

A. Scalability of the tasks scheduler

We have first benchmarked our task scheduler itself, without any networking involved. In this benchmark, we measure the time spent to create an empty task (with no computation), to schedule it, and to notice its completion. We have measured the performance of every queue in the hierarchy. In all cases, the task is submitted by core #0. We have performed benchmarks on two different machines.

The first machine, `borderline`, is a 4-socket dual-core AMD Opteron 8218 (total: 8 cores). This CPU model does not feature L3 cache, thus sibling cores on a chip do not share cache, but they share physical memory banks. Results of the test for this machine are shown in Table I.

The second machine, `kwak`, is a 4-socket quad-core AMD Opteron 8347HE (total: 16 cores). This CPU model features a L3 cache, shared between the 4 cores of each chip. The machine is composed of 4 NUMA nodes. Its NUMA topology is depicted on Figure 3 Results of the test for this machine are shown in Table II.

We measured that the raw cost of submitting a task and schedulling it locally on core #0 is around $700\,ns$ for both machines, which we take as a reference in the following. The measured overhead compared to this reference depends on the level of the queue where the task is submitted. In the topology of both machines, we distinguish 3 levels:

- level 1, per-core queues: the performance is roughly constant and the overhead is negligible. The overhead is comprised only of inter-core or inter-CPU NUMA communication. Core #0 and its siblings (core #1 on `borderline`, cores #1, #2, and #3 on `kwak`) are faster than the others since processing is done locally, either in memory or in L3 cache, whereas every other core gets roughly the same overhead corresponding to an inter-CPU NUMA communication. This overhead is roughly $100\,ns$ on `borderline` and $1\,\mu s$ on `kwak`. Core #0 presents a slight overhead (approximatively 25 ns) compared to cores #3-7 since this core both creates tasks and executes them. There no contention in this case since the cores are polling locally in their own queue.
- level 2, per-chip queues: at this level, the queue is shared between cores that share a memory bank, and L3 cache depending on the CPU model. On `borderline`, we observe a total time around $1.1\,\mu s$ for task scheduling, which makes an overhead of roughly $300\,ns$ compared to the reference performance. On `kwak`, the total time is around $1.9\,\mu s$ for task scheduling on NUMA node #1, which makes an overhead of roughly 1.2 μs compared to the reference performance. Other NUMA nodes exhibit an overhead of roughly 200 ns compared to the reference performance. The performance of NUMA node #3 remains unexplained. We assume this high overhead is due to a race condition, but this has to be investigated. The overhead is explained by the contention to acquire the spinlock of the queue. The contention is still low since the competition takes place between cores that share memory. We observe that tasks are equally processed by each cores within a NUMA node: the Per-Chip Queue being shared by four cores, each of them executes roughly 25 % of the submitted tasks.
- level 3, Global Queue: at this level, the queue is shared by all cores. The measured overhead is around 13.5 μs on `kwak` and 4 μs on `borderline`. Such a high overhead is caused by hard contention to acquire the spinlock that protects the queue. This overhead actually depends on the number of cores that try to schedule tasks. The overhead appears to grow quickly with the number of cores.
  The distribution of tasks execution across the cores shows it is unbalanced: most of the tasks are executed by cores located on NUMA node #2. This can be explained by the use of spinlocks on this NUMA machine: when the spinlock is released, the cores located on the same NUMA node notice it quickly while other cores have to

core	#0	#1	#2	#3	#4	#5	#6	#7
per-core queues	770	788	839	818	846	858	858	1819
per-chip queues, 2 cores	1114		1059		1157		1199	
global queue (8 cores)	4720							

Time given in nanoseconds.

TABLE I
MICRO-BENCHMARK OF TASK SCHEDULING ON A 4-WAY DUAL-CORE

core	#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15
per-core queues	723	697	697	697	1777	1787	1776	1777	1777	1867	1866	1867	1747	1737	1737	1787
per-chip queues, 4 cores	1905				2037				2046				5216			
global queue, 16 cores	13585															

Time given in nanoseconds.

TABLE II
MICRO-BENCHMARK OF TASK SCHEDULING ON A 4-WAY QUAD-CORE

wait the notification to their NUMA node.

It is clear from these measures that placing tasks in Per-Core or Per-Chip queues that ensure locality reduces contention and scales very well. On the other hand, it appears that a Global Queue shared between all cores does not scale very well with the number of cores.

B. Impact of threads on latency

In order to evaluate the impact of threads on network latency, we use the multi-threaded latency test that is included in the OSU Micro Benchmark suite [14]. This benchmark performs ping-pong test with a single sender and multiple receiver threads. The sending process sends a 4-bytes message to the receiver and waits for a reply. Each receiving thread calls `MPI_Recv` and sends back a 4-bytes reply. This operation is performed many times and the average one-way latency numbers are reported.

This test was conducted on the BORDERLINE cluster in Bordeaux. This cluster nodes are composed of 4-socket dual-core AMD Opteron 8218. The OS is Linux 2.6.22 and each box is equipped with one Myricom Myri-10G NIC (with the MX 1.2.7 driver) and one ConnectX Infiniband NIC (MT25408, with the OFED 1.2-c driver). The results were obtained using Infiniband.

The multi-threaded latency test was conducted for MAD-MPI (the MPI interface to NEWMADELEINE that uses PIOMAN), MVAPICH2 1.2p1 and OPENMPI 1.3.1. However, dispite the thread-safety parameter passed to OPENMPI during its configuration, this MPI implementation couldn't run this test as segmentation faults occured.

Figure 4 shows the performance comparisons between MAD-MPI and MVAPICH. The average latency observed for MVAPICH highly depends on the number of receiving threads. This may be due to the concurrency between the threads that wait for incoming messages and keep polling the network.

On the other hand, the latency observed for NEWMADELEINE remains almost constant as the number of receiving threads grows, even when this number exceeds the number of CPUs. This can be explained by the global management of communication flows: receiving threads wait their data using a blocking condition. The receiving of data is performed by idle threads. The concurrency while polling is thus very limited.

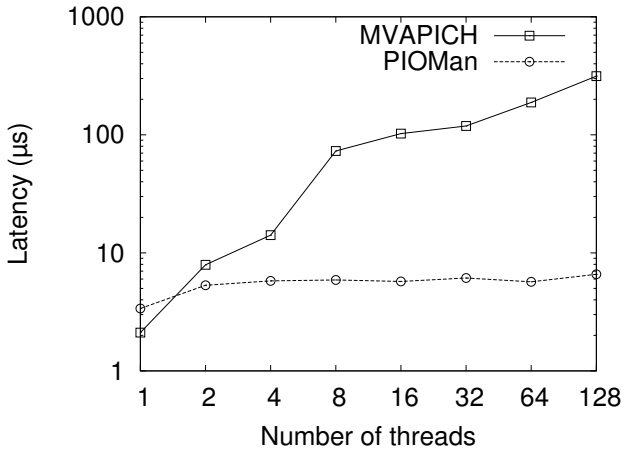


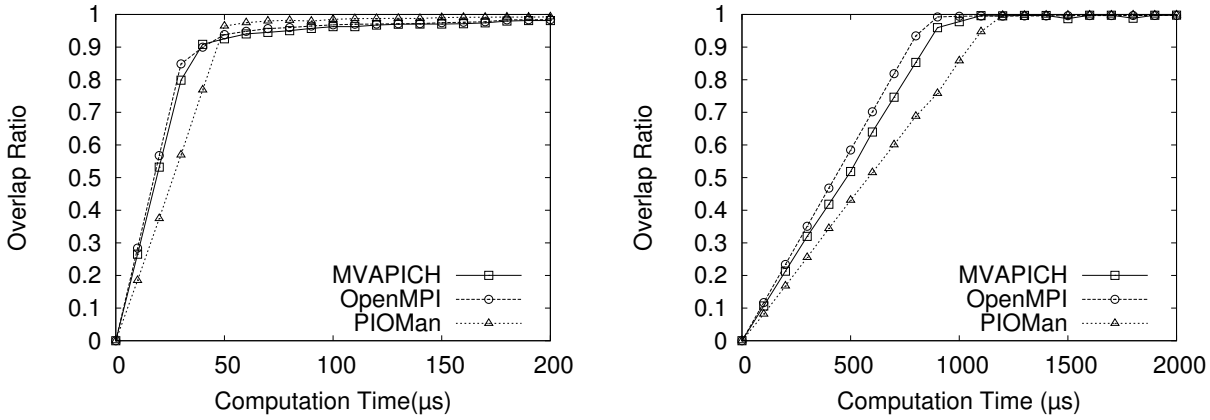
Fig. 4. Multi-threaded latency test on BORDERLINE

C. Overlapping of communication and computation

The progression of communication in the background has been evaluated using the micro benchmark proposed in [15]. This program consists in performing a non-blocking communication operation, computing for a certain duration and wait for the completion of the communication. The overlap ratio that we measure is defined as:

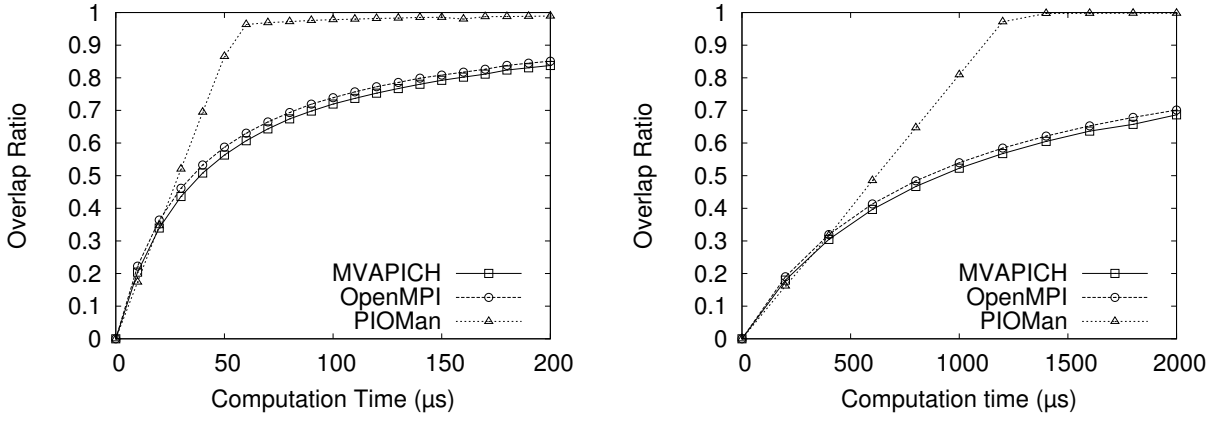
$$Overlap = \frac{T_{comp}}{T_{total}}$$

Where T_{comp} corresponds to the computation duration and T_{total} corresponds to the total duration of the communication and the computation (*i.e.* the time spent between `MPI_Isend`/`MPI_Irecv` and `MPI_Wait`). As a result, a high overlap ratio means that communication is overlapped efficiently.



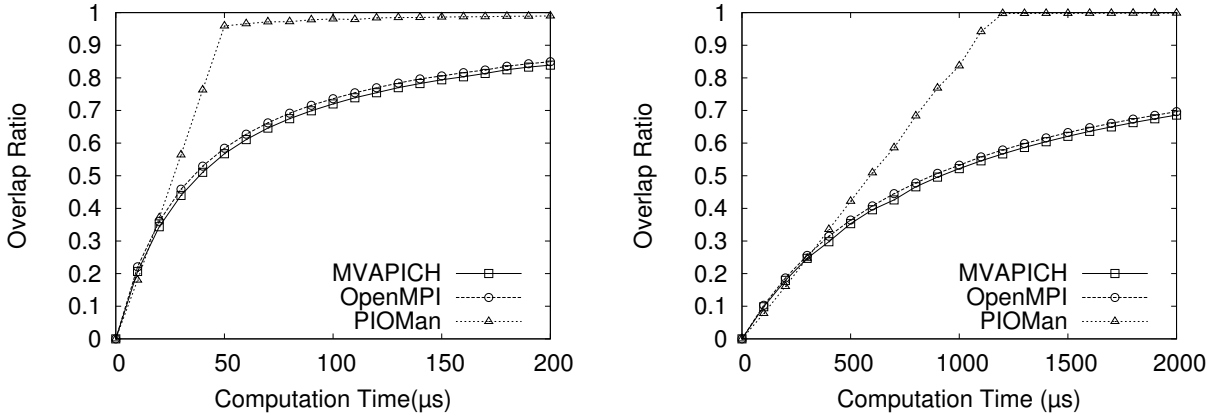
(a) Send 32 KB (b) Send 1 MB

Fig. 5. Overlap Performance (computation on sender side)



(a) Recv 32 KB (b) Recv 1 MB

Fig. 6. Overlap Performance (computation on receiver side)



(a) Send/recv 32 KB (b) Send/Recv 1 MB

Fig. 7. Overlap Performance (computation on both sides)

We have conducted this benchmark on the `Borderline` cluster using Infiniband. We compare MAD-MPI, MVAPICH2 1.2p1 and OPENMPI 1.3.1. Figures 5, 6, and 7 show the results we obtained for 32 KB and 1 MB message size.

OPENMPI and MVAPICH have the same behavior: the communication and the computation is overlapped only at the sender side. The use of a RDMA-Read technique permits to avoid a part of the *rendezvous* [10]. However, the communication and the computation are not overlapped when the receiver process computes between the non-blocking operation and the corresponding waiting MPI call.

MAD-MPI is able to overlap communication and computation in either cases: when a MPI non-blocking call is performed, a task is created in order to detect the message completion. The application can then continue its computation and the *rendezvous* handshake is performed in the background.

VI. CONCLUSION

The massive use of multicore processors in modern clusters leads toward an increase of the number of cores per node. This evolution raises scalability issues as communication libraries have to support more and more concurrent communication flows. Designing an efficient modern communication library requires precautions in order to limit the impact of thread-safety, the contention on network interfaces or the consequence of data locality on performance.

In this paper, we have proposed the design and implementation of a task offloading mechanism for communication libraries. Our task manager works closely with the thread scheduler in order to process communication tasks on idle cores. This way, CPU-hungry PIO communication or *rendezvous* handshakes can be offloaded, overlapping communication and computation transparently. The use of hierarchical data structures permits to schedule tasks in a scalable manner: contention on task queues is reduced and data affinity is preserved.

We have implemented this design in the PIOMAN I/O manager and the NEWMADELEINE communication library makes an extensive use of this mechanism to process communication flows in the background. The performance results we conducted demonstrate the scalability of the task mechanism and performance comparisions with state of the art solutions show that PIOMAN is able to overlap communication and computation even on the receiver side.

In the short term, we plan to study the opportunity to use lock-free algorithms to reduce contention on task queues and to decrease the overhead of the task mechanism. The possibility to use preemptive tasks – that is, tasks that can be executed immediately, even on a distant CPU where a thread is computing – will also be investigated. We also plan to integrate the task mechanism in an I/O library. In the long term, the goal is to provide a generic framework able to optimize both communication and I/O in a scalable way.

REFERENCES

[1] F. Trahay, E. Brunet, and A. Denis, "An Analysis of the impact of multi-threading on communication performance," in *CAC 2009: The 9th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2009*. Rome, Italy: IEEE Computer Society Press, May 2009.

[2] 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: IEEE, April 2008. [Online]. Available: http://hal.inria.fr/inria-00224999

[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] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward Efficient Support for Multithreaded MPI Communication," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2008, pp. 120–129.

[5] S. Coll, E. Frachtenberg, F. Petrini, A. Hoisie, and L. Gurvits, "Using multirail networks in high-performance clusters," *Cluster Computing, 2001. Proceedings. 2001 IEEE International Conference on*, pp. 15–24, 2001.

[6] 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*, Long Beach, California, USA, March 2007, also available as LaBRI Report 1421-07 and INRIA RR-6085. [Online]. Available: http://hal.inria.fr/inria-00127356

[7] S. Moreaud and B. Goglin, "Impact of numa effects on high-speed networking with multi-opteron machines," in *The 19th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2007)*, Cambridge, Massachussetts, November 2007. [Online]. Available: http://hal.inria.fr/inria-00175747

[8] J. Sancho, K. Barker, D. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM New York, NY, USA, 2006.

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

[10] S. Sur, H. Jin, L. Chai, and D. Panda, "RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM New York, NY, USA, 2006, pp. 32–39.

[11] A. Maccabe, W. Zhu, J. Otto, and R. Riesen, "Experience in offloading protocol processing to a programmable NIC," in *2002 IEEE International Conference on Cluster Computing, 2002. Proceedings*, 2002, pp. 67–74.

[12] S. Dandamudi and S. Cheng, "Performance impact of run queue organization and synchronization on large-scale NUMA multiprocessor systems," *Journal of Systems Architecture*, vol. 43, no. 6, pp. 491–512, 1997.

[13] 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*, ser. Lecture Notes in Computer Science, F. Cappello, T. Herault, and J. Dongarra, Eds., vol. Recent Advances in Parallel Virtual Machine and Message Passing Interface, no. 4757. Springer, 2007, pp. 170–177. [Online]. Available: http://hal.inria.fr/inria-00177167

[14] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. Panda, "Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics," in *Supercomputing, 2003 ACM/IEEE Conference*, 2003, pp. 58–58.

[15] A. Shet, P. Sadayappan, D. Bernholdt, J. Nieplocha, and V. Tipparaju, "A framework for characterizing overlap of communication and computation in parallel applications," *Cluster Computing*, vol. 11, no. 1, pp. 75–90, 2008.