Proceedings of the First International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2014)
Porto, Portugal

Jesus Carretero, Javier Garcia Blas
Jorge Barbosa, Ricardo Morla
(Editors)

August 27-28, 2014

On the Performance Of the Thread-Multiple Support Level In Thread-Based MPI

Juan-Carlos Díaz-Martín
University of Extremadura, Spain
juancarl@unex.es

Juan-Antonio Rico-Gallego
University of Extremadura, Spain
jarico@unex.es

Abstract

Exascale systems are likely to have orders of magnitude less memory per core than current systems (though still large amounts of memory). As the amount of memory per core is dropping, going to thread-based models might be an unavoidable step towards the exascale milestone. AzequiaMPI is a thread-based open source full conformant implementation of MPI-1.3 for shared memory. We expose the techniques introduced in AzequiaMPI that, first, simplify the implementation and second, make the thread-based model to significantly improve the bandwidth of process-based implementations. Current version is also compliant with the MPI_THREAD_MULTIPLE thread-safety level, a feature of MPI-2.0 standard. The well known Thakur and Gropp MPI_THREAD_MULTIPLE tests show that both latency and bandwidth figures of AzequiaMPI significantly improve that of MPC-MPI, MPICH and Open MPI in an eight cores Intel Xeon E5620 Nehalem machine.

Keywords MPI performance, Thread-based MPI, MPI_THREAD_MULTIPLE, Multicore architectures

I. Introduction

MPI [4] is an industry de-facto parallel programming standard based on the message-passing paradigm. As new languages and alternatives as Unified Parallel C (UPC) and OpenMP are being proposed for supporting more efficiently multicore architectures, MPI keeps facing the challenge. The fact is that MPI is still used on shared memory due to its better portability and its good data locality due to data partitioning. An MPI application is composed by a set of independent program instances that communicate by message passing. Traditional mainstream MPI implementations as MPICH and OpenMPI build each instance as a full-fledged process. It entails some disadvantages in shared memory because message passing between two processes must go through a per-pair intermediate shared buffer, and copying degrades the communication efficiency.

Two trends drive the performance of current HPC clusters. One is the strong increase in the amount of memory-per-node, and the other the rising number of cores per processor. However, if the count of cores per machine is doubling approximately every 2 years, the DRAM DIMM capacity is just doubling about every 3 years [5]. This means that the memory capacity per core is expected to drop by 30 % every two years. As a result, exascale systems are likely to have orders of magnitude less memory per core than current systems. Against this backdrop, going to thread-based models might be an unavoidable step towards the exascale milestone [3], because it should reduce the application memory footprint [7]. Not only that, thread-based MPI improves the MPI performance in shared memory because enables optimised communication mechanisms, as the single copy. This paper is about this issue.

It is possible building the MPI instance as a thread. Thread-based MPI has been around for years, but the fact is that no thread-based MPI implementation has been widely adopted in practice. The reasons are not fundamental, but rather practical concerns imposed by the prevalence of the OS-level process. One of them is the issue of the global variables of a MPI rank, that get shared by them all under the thread-based case, giving place to faulty programs. Though privaticing them through program transformation techniques is a well studied requirement, and some partial efforts have been undertaken in this direction [[11], [10]], the fact is that no explicit tool is currently available, or at least provided by a main stream tool chain to transparently circunvent the problem. The other argumented weakness of thread-based MPI is the thread-safety of third-party software, either library code or device drivers. It is true that non thread-safe software is unusable by theaded MPI ranks, but the same happens to the threads created under the MPI_THREAD_MULTIPLE ability of the MPI-2 standard in a process-based implementation. All in all, a not thread-safe library should be labelled as quite limited software product nowadays, whether used in an MPI context or not.

AzequiaMPI [9] is thread-based but still an open source full conformant implementation of the MPI-1.3 standard. Its original version was targeted to embedded distributed signal processing platforms of DSP and FPGAs supporting a single address space, which forced us to implement MPI upon threads. This work presents performance evaluation of current AzequiaMPI, targeted to standard HPC multicore machines. The design is rooted on the lock-free queue structure used in MPICH2-Nemesis, but exploiting the advantage of sharing a common address space. Our tests show a relevant improvement against MPICH, Open MPI and MPC-MPI in an 8 cores Nehalem machine. The rest of the paper is as follows. Related work is presented in section 2. In section 3 we introduce the core design of the system and comparative performance figures. Section 4 presents the extensions to the desing that support the MPI_THREAD_MULTIPLE option in an efficient way. Section 5 concludes.

II. Related Work

Implementing a MPI node as a thread is not a new concept, but it has received very limited attention in the literature. Seminal work [2] discusses TOMPI (for Thread Only MPI), an early proof of concept prototype that implements just a handful of MPI primitives, not even in a conformant way.

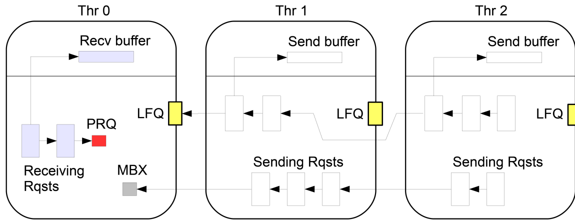

Figure 1: Requests and queues in AzequiaMPI.

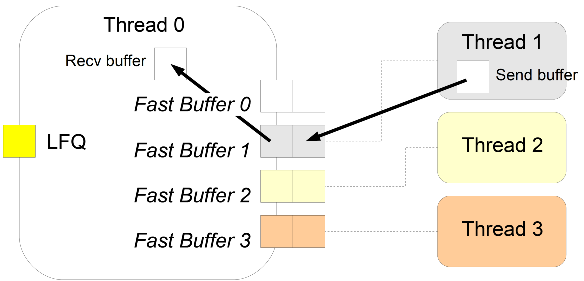

Figure 2: FastBuffers in a configuration of four threads.

TMPI (for threaded MPI) [11] is a more solid and deeper research on thread-based MPI. TMPI is also a partial implementation of MPI-1 (29 functions) addressed to clusters of multiprocessors. Authors claimed that, on the average, TMPI is 46% faster than MPICH, but note that the further Nemesis library made MPICH up to twice faster for short messages and shows an up to 1.4 factor of improvement for large messages. Unfortunately, TMPI only provides the source code of an early version based in mutexes.

MPI Actor (for MPI Accelerator) [6] is a middleware that maps MPI nodes to threads in multicore machines. It claims to patch any existing process-based MPI implementation, so that a MPI process runs as a thread in shared memory. Its big advantage would be avoiding the huge work of building a new fully conformant thread-based implementation. Unfortunately, MPI Actor source code is not publicly available. Anyway, the performance of AzequiaMPI overcomes by large that achieved by MPI Actor.

MultiProcessor Communications environment (MPC) [8], aims an efficient runtime system unifying the MPI, POSIX Threads and OpenMP. The key idea is to use user-level threads instead of processes to increase scheduling flexibility, to better control memory allocations and optimize scheduling of the communication flows with other nodes.The scheduler and memory allocator modules co-operate to preserve data locality, a crucial issue when dealing with NUMA nodes.

III. Thread-Single: Design and Performance

Fig. 1 shows the basic design of AzequiaMPI. Each MPI rank (a thread) has three queues, known as PRQ, MBX and LFQ. PRQ is the queue of pending receive requests, and MBX (for mailbox) is the queue of unexpectedly arrived send requests. Both are ordinary double-linked lists. Only its owner rank accesses to them. LFQ is a lock-free queue, the same used by MPICH2-Nemesis [1]. It allows a single receiver (its owner) and many senders. All of them access it without locking. To receive a message, the MPI rank r allocates a receiving request R from its per-thread pool. R is initialised so that a field points to the receive user buffer. Next, r explores MBX, looking for a send request S that matches R. On success, r performs the copy, updates S as satisfied, and finally dequeues and liberates R. If no matching happens, r enqueues R in its PRQ queue. If the receive primitive is blocking (MPI_Recv) r enters the progress routine, that basically polls LFQ for new events. If the primitive is non-blocking (MPI_Irecv) r returns.

To send a message, an MPI rank s allocates a sending request S from its per-thread pool. A field of S points to the send user buffer as Fig. 1 shows. Next, S is enqueued in the receiver lock-free queue LFQ. A blocking send primitive makes s entering the progress engine, that polls the state field of S until it is satisfied. Then s liberates S. A non-blocking send simply makes s to continue and defers the polling to subsequent test/wait invocations.

The progress routine basically polls LFQ for new events. When the invoking rank t dequeues a send request S from LFQ, t matches S against its PRQ. If a matching receiving request R is found, t dequeues and liberates R, makes the copy from the send buffer to the receive buffer and set S as satisfied. If a matching receive request is not found, S is enqueued in MBX.

MPICH2-Nemesis provides a mechanism called fastbox to accelerate small messages [1]. A fastbox is a small buffer associated to each pair of MPI ranks. It allows a sender to by-pass the LFQ by copying directly its message to the fastbox. After the copy is done, an integer flag acting as a turn is switched for reception. The receiver copies from the fastbox to its user buffer, switching the turn for sending. AzequiaMPI extends fastboxes to fastbuffers, as Fig. 2 illustrates. In AzequiaMPI we have observed in the respective left and right scenarios of Fig. 3 that a fastbuffer, with either one or two fastboxes, diminish the latency of small messages by half, and that a fastbuffer of two fastboxes multiplies the bandwidth of small messages by four.

Fastboxes, however, come with a price. They pose the issue of message ordering because introduce a second path between a sender and a receiver. If the receiver checks the fastbox before the LFQ or viceversa, it may receive the messages in the wrong order. This situation is handled so that every local message carries on a sequence number that is matched on reception, as it is done with source and tag. Messages up to 1 KB are delivered this way if the fast buffer is not full. Management of sequence numbers is tricky, but cheap. The true downside of fastBuffers is memory compsumtion, of order $O(Q^2)$ where Q is the number of cores per node. If the size of the fastBuffer is 2KB, the implementaton would allocate 2KB x 128 x 128 = 32 MB only to this resource on a machine of $Q = 128$ cores. To save memory AzequiaMPI creates fastbuffers on the fly, and only when really needed. Look at Fig. 2. The four fast buffers of rank 0 do not come into existence when the application starts up. Instead fastbuffer 1, for instance, is created by rank 1 when it tries to send its first small message to rank 0.

Fig. 3 shows the setup produced by the single-threaded latency and bandwidth tests from the well known Thakur and Gropp benchmark [12], in the eight cores of the Nehalem. The benchmark is included in the source distribution *gim.unex.es/azequiampi* of AzeqioaMPI. Latency is measured on a ping-pong setup, and likely bandwidth on a stream. Fig. 4 and Fig. 5 show the performance

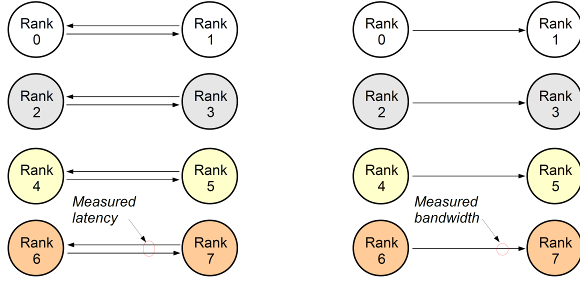

Figure 3: Thakur and Gropp single threaded tests. Experimental setup.

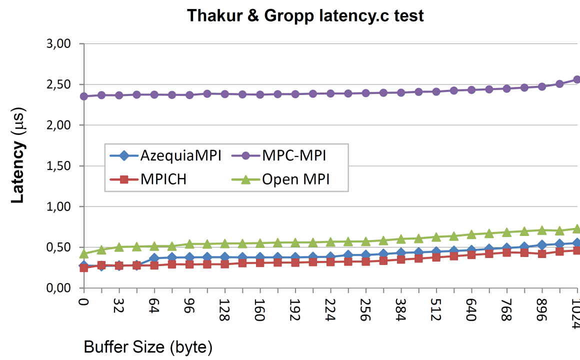

Figure 4: Latency measurements of four MPI implementations in the setup at the left side of Fig. 3.

achieved by this design, compared to that of MPICH, OpenMPI and MPC-MPI. AzequiaMPI threads, and MPICH and Open MPI processes are core bound in round-robin. It is currently not possible to bind tasks in MPC-MPI. The eight MPC-MPI threads, designed to run unbound in oversubscripted scenarios, have the eight cores available. Note that MPC-MPI is almost one order of magnitud slower in terms of latency than AzequiaMPI and MPICH. MPICH produces the best outcomes in this respect. AzequiaMPI shows the best bandwidth, partly due to an optimisation technique only possible when the send and receive user buffers share memory: both sender and receiver cooperate in the copy, so that, for instance, sender copies lower half of send buffer to lower half of receive buffer, and likely the receiver on the upper halves. We call this method "split copy" and it should be clear that it constitutes a significant advantage with respect to process based implementations. It seems that MPC-MPI does not currently exploit this technique.

IV. Thread-Multiple: Design and Performance

AzequiaMPI, being MPI-1.3 conformant, also supports the highest level of thread safety for user programs, MPI_THREAD_MULTIPLE, an MPI-2 feature that allows concurrent MPI calls from multiple threads. Turning thread-safe a MPI implementation is a problem of introducing the right set of mutexes around the critical message queues, what inevitably leads to a performance breakdown. Minimizing this impact is a challenging task. This section discusses the

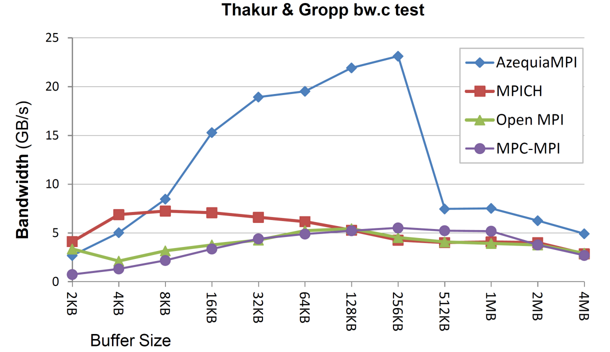

Figure 5: Bandwidth measurements of four MPI implementationsin the setup at the right side of Fig. 3.

MPI_THREAD_MULTIPLE design in AzequiaMPI.

The standard says that all the threads of a rank s can send a message to another rank r, and that any thread of rank r can handle the message, but a specific thread is not an addressable object. On other hand, the standard sets out that a pair of messages from the same rank but emitted by different threads are considered as concurrent events (even in the case that the messages had been emitted "physically" one after the other). This means that it is not possible to establish a temporal order relation between them and, as a result, they may be collected by the receiver rank in any order. Obviously, the messages sent by the same thread do keep the order. AzequiaMPI enforces such order using the simple concept of *flux*, which will be addressed below.

On the reception side, the standard literally states that "if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order." We understand that this statement frees the implementation from ordering receiving requests from different threads r_i of a rank r, and as a result, each thread r_i may set up its own PRQ receiving queue .

A flux is a sort of connexion, an object with a pair source-destination $[s_i, r]$ where s_i is the thread i of source rank s and r

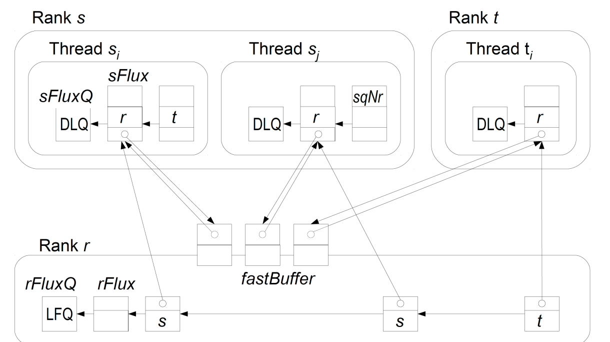

Figure 6: Design of fluxes. Any thread sending to a rank creates its private fastbuffer

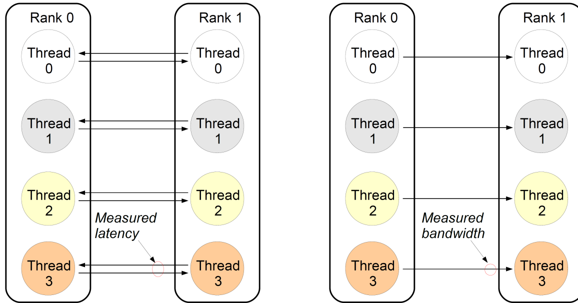

Figure 7: Thakur and Gropp multiple threaded tests. Experimental setup.

the destination rank. The source and destination extremes of a flux are internal objects of types $sFlux$ and $rFlux$ respectively. The first time the thread s_i invokes MPI_Send to send to a rank r, s_i allocates a $sFlux$ object, initialises it with the name $[s_i, r]$ and enqueues it in a private single-linked list. Next, at the destination extreme, s_i allocates the counterpart $rFlux$ object, initialises it with the handle of its corresponding $sFlux$ object and enqueues it in a lock-free queue of r named $rFluxQ$. Concurrent insertions in $rFluxQ$ are expected from any thread of any local rank. Every flux has an associated fastbuffer, a technique that improves the latency of the implementation (see Fig. 6). Fasbuffers introduce a second path from source to destination, what imposes a sequence number in the messages of a flux. The flux object keeps the current sequence number.

When a thread s_i invokes MPI_Send to send a message to rank r, it begins by looking up the $[s_i, r]$ object in its $sFluxQ$ queue. If r is not there then the flux object is created. As before mentioned, the first message of the flux under 1KB arranges the creation of the fastBuffer, whose handle is registered in $sFlux$. These small messages are sent by copying to the fastBuffer (Fig. 2) and then MPI_Send returns. Greater messages follow another path. A request object S is allocated per message, and enqueued in the LFQ of r, as shown in Fig. 1. Once enqueued, MPI_Send polls a flag of S. When the flag is set, the message has been received, then thread s_i frees S and returns. This process is done concurrently in a thread-safe way by all the s_i threads of process s without using any mutex.

When a thread r_i invokes MPI_Recv for a small message from rank s, it looks up all the fastbuffers associated to the threads of s. To this end r_i runs through the LFQ of $rFlux$ to access these fastbuffers. If a matching takes place the fastbuffer content is copied to the r_i user buffer and MPI_Recv returns. It can be thought that the $rFluxQ$ lock-free queue may be accessed for insertion while it is being explored. It can be shown, however that both operations are safe when they take place simultaneously, what avoids the introduction of a costly mutex to protect $rFluxQ$. For messages greater than 1KB, and also when no matching happens in the fastbuffer, MPI_Recv allocates a receiving request, inserts it in the private queue of receiving pending requests (PRQ) of r_i and enters the progress engine in an infinite loop.

Each iteration of the progress engine works in two stages. The first stage runs through the private PRQ queue of the invoking thread, let be t_j from rank t. For each request R found in PRQ the pair [s, keyTag] is obtained, where s is the desired source rank. Then

1. The set of fastBuffers from s (see Fig. 6) is probed against the pair [s, keyTag], as above discussed. If a matching happens R becomes satisfied and the progress engine returns.

2. If no matching takes place in the fastbuffers, the private unexpected queue (PMBX) of invoker t_i is probed against the pair [s, keyTag]. If a matching source request S is found, it is satistied, as well as R, and the progress engine returns.

3. If no matching takes place in PMBX, the global unexpected queue MBX is probed against the pair [s, keyTag]. MBX is a double-linked queue of owner t, concurrently read and written by all the t_i threads. It hence needs a mutex.

If no matching happens, the iteration of the progress engine enters the second stage. It consists of a loop of dequeue operations on the LFQ of rank t (see Fig. 1) until it becomes empty. This type of queue allows two or more concurrent enqueuers, but a single dequeuer, what imposes a mutex m to protect dequeueing. A thread t_j acquires m before a dequeue operation. Each dequeued request S is proved against all the pending requests of the private PRQ of t_j. If a matching of source and tag happens, but not of sequence

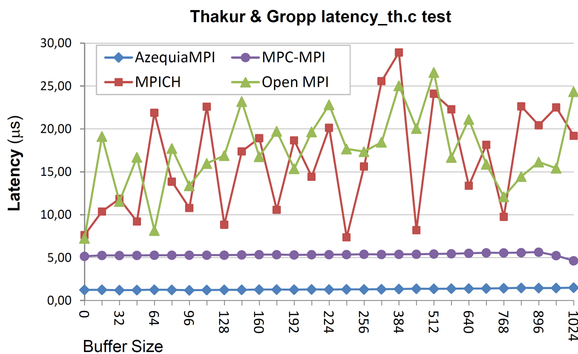

Figure 8: Latency measurements of four MPI implementations in the setup at the left side of Fig. 7.

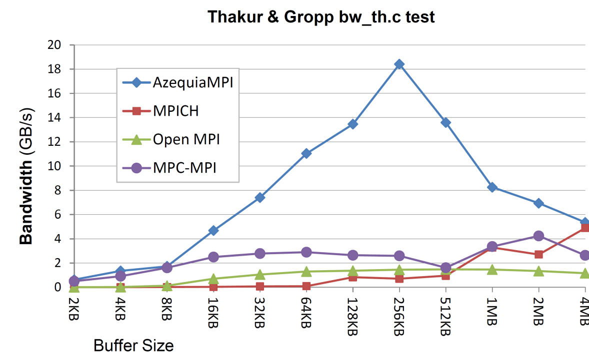

Figure 9: Bandwidth measurements of four MPI implementations in the setup at the right side of Fig. 7.

number, S is enqueued in the private unexpected queue PMBX of t_j; else is enqueued in the global unexpected queue MBX. The attained efficiency of this MPI THREAD MULTIPLE design has been measured with the Thakur and Gropp benchmark, and compared to that of MPICH, Open MPI and MPC-MPI. Fig. 7 shows the benchmark setup and Fig. 8 and Fig. 9 the obtained results. It can be appreciated that the AzequiaMPI figures considerably improve that of the rest of implementations.

V. Conclusions and further work

The thread-based design of AzequiaMPI and further optimizations based on its common address space makes it to outperform other MPI distributions in a significant manner. AzequiaMPI has shown that the thread-based approach opens opportunities to implement the MPI_THREAD_MULTIPLE thread safety level in a more efficient way than current popular MPI process-based libraries do. We aim to explore the implementation of the recent extensions of the MPI-3 standard to support shared memory at the ligth of these experiences in a thread-based framework.

Acknowledgment

The work presented in this paper has been partially supported by EU under the COST programme Action IC1305, 'Network for Sustainable Ultrascale Computing (NESUS)'.

References

[1] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 10–pp. IEEE, 2006.

[2] Erik Demaine. A threads-only mpi implementation for the development of parallel programs. In *In: Proceedings of the 11th International Symposium on High Performance Computing Systems*, pages 153–163, 1997.

[3] Jack Dongarra, Pete Beckman, and Terry Moore et al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[4] MPI Forum. Mpi: A message-passing interface standard, version 3.0., September 2012.

[5] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH Comput. Archit. News*, 37(3):267–278, June 2009.

[6] Zhiqiang Liu, Kaijun Ren, and Junqiang Song. Mpiactor - a multicore-architecture adaptive and thread-based mpi program accelerator. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, HPCC '10, pages 98–107, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Marc Perache, Patrick Carribault, and Hervé Jourdren. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *EuroPVM/MPI 2009*, pages 94–103, Helsinki, Finlande, September 2009. Springer-Verlag.

[8] Marc Pérache, Hervé Jourdren, and Raymond Namyst. Mpc: A unified parallel runtime for clusters of numa machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, pages 78–88, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Juan-Antonio Rico-Gallego and Juan-Carlos Díaz-Martín. Performance evaluation of thread-based mpi in shared memory. In Yiannis Cotronis, Anthony Danalis, DimitriosS. Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 337–338. Springer Berlin Heidelberg, 2011.

[10] Eduardo R. Rodrigues, Philippe O. A. Navaux, Jairo Panetta, and Celso L. Mendes. Preserving the original mpi semantics in a virtualized processor environment. *Sci. Comput. Program.*, 78(4):412–421, April 2013.

[11] Hong Tang, Kai Shen, and Tao Yang. Program transformation and runtime support for threaded mpi execution on shared-memory machines. *ACM Transactions on Programming Languages and Systems*, 22:673–700, 2000.

[12] Rajeev Thakur and William Gropp. Test suite for evaluating performance of multithreaded {MPI} communication. *Parallel Computing*, 35(12):608 – 617, 2009. Selected papers from the 14th European PVM/MPI Users Group Meeting.