Efficient MPI Support for Advanced Hybrid Programming Models

Torsten Hoefler[1⋆], Greg Bronevetsky[2], Brian Barrett[3], Bronis R. de Supinski[2], and Andrew Lumsdaine[4]

[1] University of Illinois at Urbana-Champaign, Urbana, IL, USA,
htor@illinois.edu
[2] Lawrence Livermore National Laboratory, Center for Applied Scientific Computing, Livermore, CA, USA
{bronevetsky1,bronis}@llnl.gov
[3] Sandia National Laboratories, Albuquerque, NM, USA
bwbarre@sandia.gov
[4] Indiana University, Open Systems Lab, Bloomington, IN, USA,
lums@cs.indiana.edu

Abstract. The number of multithreaded Message Passing Interface (MPI) implementations and applications is increasing rapidly. We discuss how multithreaded applications can receive messages of unknown size. As is well known, combining MPI_Probe/MPI_Recv is not thread-safe, but many assume that trivial workarounds exist. We discuss those workarounds and show how they fail in practice by either limiting the available parallelism unnecessarily, consuming resources in a non-scalable way, or promoting global deadlocks. In this light, we propose two fundamentally different efficient approaches to enable thread-safe messaging in MPI-2.2: fine-grained locking and matching outside of MPI. Our approaches provide thread-safe probe and receive functionality, but both have deficiencies, including performance limitations and programming complexity, that could be avoided if MPI would offer a thread-safe (stateless) interface to MPI_Probe. We propose such an extension for the upcoming MPI-3 standard, provide a reference implementation, and demonstrate significant performance benefits.

1 Introduction

Current processor trends are leading to an abundance of clusters composed of multi-core nodes. While the Message Passing Interface (MPI [1]) remains a viable programming model to use all processors in these systems, multi-core systems naturally lead to increased use of shared memory programming models based on threading. Hybrid MPI/threaded programs can decrease the surface to volume ratio between MPI processes, which can result in more efficient use of the interconnection network [2]. Thus, these hybrid programs are becoming increasingly common [3]. As a result, it is critical for MPI to support the model well.

MPI-2 includes a mechanism to request a level of thread support. Previously, most hybrid programs could conform to the MPI_THREAD_FUNNELED level. With the increase in hybrid programs, applications that use shared memory task parallelism and,

⋆ The first author performed most of this work at Indiana University.

thus, require MPI_THREAD_MULTIPLE support, are more likely. This trend not only motivates the implementation of that support [4] but also an examination of how well the MPI standard supports those programs. We find that the support is generally sufficient [5] although one glaring weakness exists: The semantics of probing for messages (e.g., in order to receive messages of unknown size) does not interact properly with realistic uses in threaded programs.

In this work, we discuss the issue of receiving messages of unknown size in multithreaded MPI programs. We explain the problem and show why obvious approaches to its solution are not feasible. We then discuss two elaborate techniques that would work with MPI-2.2. Despite the complex implementation of such techniques, which could be done in a library, we show that all proposed solutions limit performance significantly. Finally, we discuss an addition to the MPI standard that would enable the desired functionality. We describe a reference implementation, discuss issues in the context of hardware-optimized implementations, and present benchmark results which show the benefits of this approach.

2 Multithreaded MPI Messaging

We discuss several options for MPI version 2.2 to receive messages of unknown size in multithreaded environments. Unknown size messages in MPI are received with the sequence of *probe* (determine the size), *malloc* (reserve buffer), and *receive* (receive message). We investigate the issue of *false matching*, in which two threads perform a probe, malloc and a subsequent receive *concurrently*. Two actions happen concurrently if they happen completely independently (e.g., without synchronization or code flow dependencies) so that they could interleave in any way. Assume that two threads, A and B, perform a probe, malloc, and receive, denoted by A_p, A_m, A_r and B_p, B_m, B_r respectively. If those calls happen concurrently, then they could interleave as the series: $A_p, B_p, B_m, B_r, A_m, A_r$ that leads to incorrectly matching a message in thread B that was probed in thread A. We show that simple workarounds either limit parallelism unnecessarily or require structural changes to the application. Therefore, we propose two more sophisticated approaches and advocate for extensions or changes to the MPI standard to improve support for probing in threaded environments.

Separating threads with tags or communicators False matching could be avoided by using different virtual channels to address each thread in each process. A virtual channel in MPI is uniquely identified by the tuple (c, s, τ) (communicator, source, tag) on the receiver side and (c, r, τ) (communicator, receiver, tag) on the sender side. False matching can be avoided by using different tags (or communicators) for each thread. However, one would need $t \cdot p$ communicators (or tags) in order to address all threads in an MPI universe with p processes, each with t threads. This mechanism is not scalable (binds $\Omega(p)$ resources) and not flexible enough for many applications. For example, a multithreaded master in a master/worker implementation can no longer use automatic load-balancing in which any idle thread probes and receives the next message to arrive. Similarly, a reentrant library that calls MPI with a variable (not predetermined) number of threads cannot use tag-based thread-addressing. Thus, such thread-addressing schemes seem unsuitable for most applications.

2.1 A Fine-grained Locking Mechanism

Clearly, with MPI's matching semantics, coarse-grained locking (e.g., protecting the access to probe/malloc/recv at the communicator) overly limits parallelism. For exam-

ple, a probe/receive pair with tag=4 and src=5 does not conflict with a probe/receive pair with tag=5 and src=5. However, another probe/receive pair with tag=4, src=5 would conflict with the first pair. Thus, we could lock each possible (communicator, source, tag) tuple separately. In the following, we assume that each lock is associated with a specific communicator and we limit the discussion to (source, tag) pairs.

One could arrange locks for (source, tag) pairs in a two-dimensional matrix. However, storing a $\max(source) \cdot \max(tag)$ matrix in main memory is infeasible. A sparse matrix representation with a hash table or map [(source, tag) $\rightarrow$ lock] seems much more efficient.

We show a simple locking strategy that minimizes the critical region with a non-blocking receive in Listing 1.1. However, this strategy does not cover wildcard receives.

```
lock map(src,tag)
probe(src, tag, comm, stat)
buf = malloc(get_count(stat)*sizeof(datatype))
irecv(buf, get_count(stat), datatype, src, tag, comm, req)
unlock map(src,tag)
wait(req)
```

Listing 1.1. Simple (limited) receive locking protocol.

Probe/receive pairs with wildcards must be performed mutually exclusively within a set of channels. Thus, if a wildcard is used, we must lock a full row or column of the matrix. If both fields are wildcards, we must lock the whole matrix. As a result, we consider four (source, tag) cases in order to implement a fine-grain locking strategy: (1) (int,int), (2) (any_src,int), (3) (int,any_tag), and (4) (any_source,any_tag). We denote any_src or any_tag with an asterisk (*) in the following. In order to support each case fully, we need a sparse two-dimensional (src, tag) and thread-safe data structure with the following operations:

(un)lock(x,y) acquires/releases (x,y)
(un)lock(x,*) acquires/releases all entries on src x
(un)lock(*,y) acquires/releases all entries on tag y
(un)lock(*,*) acquires/releases the whole matrix

Our sparse two-dimensional locking protocol differentiates among these four cases, using three levels of locks: A two-dimensional map of locks for all points (source, tag), two one-dimensional maps of locks for each source and tag line, and one lock for the whole matrix. It uses lists of held locks per (source, tag) pair, for each source and each tag and for the whole matrix. Listing 1.2 shows a possible algorithm that implements a sparse two-dimensional locking structure. The code shown in Listing 1.2 is a critical region that is protected with locks itself!

```
if (source != MPI_ANY_SOURCE and tag != MPI_ANY_TAG)
  check if either whole matrix, source, tag, (source, tag) is locked
  if (nothing is locked)
    lock (source, tag) and increase usage count of source, tag, matrix
if (source != MPI_ANY_SOURCE and tag == MPI_ANY_TAG)
  check if either whole matrix or source is locked
  check if any_source or some tag for source is in use
  if (nothing is locked/used)
    lock source and increase usage count of source and matrix
```

```
if (source == MPI_ANY_SOURCE and tag != MPI_ANY_TAG)
  check if either whole matrix or tag is locked
  check if any_tag or if some source for tag is in use
  if (nothing is locked/used)
    lock tag and increase usage count of tag and matrix
if (source == MPI_ANY_SOURCE and tag == MPI_ANY_TAG)
  check if whole matrix is locked or in use
  if (nothing is locked/used)
    lock matrix
```

Listing 1.2. Function to lock the `2d_sparse_map`. Unlock is equivalent.

However, while this local locking scheme ensures correct and parallel message reception, it can unexpectedly influence global synchronization. For example, rank 0 sends two messages to rank 1 in which sending of the second message depends on a reply to the first message. The first message has tag 1, and the second message has tag 2. The receiver, rank 1, has two threads A and B. Thread A receives from channel (0, 2) and thread B from channel (0, any_tag). Thread A sends the needed reply after the message is received. We show pseudo-code for rank 0 in Listing 1.3 and for rank 1 in Listing 1.4.

```
A:
  send(..., 1, 1, comm)
  recv(..., 1, 1, comm)
  send(..., 1, 2, comm)
  ...
```

Listing 1.3. Rank 0

```
A:
  probe/recv(0, 2, comm)
B:
  probe/recv(0, ANY_TAG, comm)
  send(..., 0, 1, comm)
```

Listing 1.4. Rank 1

This program must terminate in a correct MPI implementation that supports MPI_THREAD_MULTIPLE. However, if A locks (0, 2) first and enters MPI_Probe then B cannot lock (0, any_tag). Thus, ranks 0 and 1 cannot proceed and the presented algorithm can cause spurious deadlocks.

In general, a receive with an explicit (integer) source and tag can block ones with wildcards, for example, receiving on channel (0, 1) blocks receives on (any_src, 1), (0, any_tag), and (any_src, any_tag). Thus, wildcard probes and receives must dominate more specific ones, which requires that MPI_Probe has not yet been called for the more specific one. Since MPI calls cannot be aborted, we must poll with multiple probes/receives. Only the most general probe/receive (any_src, any_tag) is allowed to block. We can implement the required polling with the same two-dimensional locking scheme to enable maximum concurrency. Listing 1.5 shows the polling (nonblocking) algorithm.

```
while(!stat)
  lock 2d_sparse_map(src,tag) /* see previous listing */
  iprobe(src, tag, comm, stat)
  if(stat)
    buf = malloc(get_count(stat)*sizeof(datatype))
    irecv(buf, get_count(stat), dtatype, src, tag, comm, req)
  unlock 2d_sparse_map(src,tag)
  if(stat) wait(req)
```

Listing 1.5. Polling receive locking protocol.

We note that requiring polling is a fundamental problem that prevents an efficient implementation of many multithreaded implementations.

Most parallel MPI applications only use a subset of the possible parameter combinations during a program run. For example, an application might not use any_src or any_tag at all, which enables the use of the simple locking scheme described in Listing 1.1. Other applications might use any_tag in all probes and receives, and enable a much simpler, one-dimensional locking of source (even though this limits possible parallelism).

Table 1 lists all combinations and possible optimizations. An x in the column any_src or any_tag means that any_src or any_tag is used during the program run. An x in "direct" indicates that at least one call does not use any_src and any_tag. For

Scenario	any_src	any_tag	Specific	Strategy
1	-	-	x	simple 2d, blocking
2	-	x	-	simple 1d, blocking
3	-	x	x	2d lock, polling
4	x	-	-	simple 1d, blocking
5	x	-	x	2d lock, polling
6	x	x	-	2d lock, polling
7	x	x	x	2d lock, polling

Table 1. Possible parameter combinations.

example, under scenario 4, all calls use any_src as an argument and thus a simple one-dimensional locking scheme can be used. Scenario 7, the most general one under which a program run could use all combinations, requires the polling scheme (Listing 1.5). Different scenarios can be defined for each communicator. Thus, performing all calls with various wildcards on distinct communicators simplifies locking requirements but might lead to other problems as discussed in the introduction.

Further, although most applications only use a subset of the possible parameter combinations, which allows for a specialized implementation, a library-based solution must provide the general implementation. Similarly, an implementation of language bindings such as MPI.NET [6] must assume the general case (scenario 7) so using the fine grained locking approach likely entails a high cost.

2.2 Matching Outside of MPI

If polling is infeasible, we can instead perform MPI source/tag matching outside of the MPI library in order to provide correct threaded semantics for MPI_Probe. This solution uses a helper thread that repeatedly calls MPI_Probe with any_src and any_tag. When MPI_Probe returns, the thread allocates a message buffer, into which it then receives the probed message with MPI_Irecv. The associated MPI_Request is stored in a data structure for use when an application thread issues a matching receive operation. This data structure is similar to the two-dimensional locking structure from Listing 1.2. For each (source, tag) pair (including wildcards) it maintains the count of threads that are waiting to receive a messages with that pair as well as two lists of messages. The first list tracks "expected" messages – newly arrived messages that match this (source, tag) pair and have been matched to waiting threads but not yet been picked up by those threads. The second list tracks "unexpected" messages – newly arrived messages for which a receiver thread has not yet been identified. Since all such messages match four

different (source, tag) pairs (including wildcards), each is placed into four such lists, one for every pair. The data structure maintains a lock and a condition variable for each pair to synchronize access to the count and the message lists.

This method can also take advantages of the previously described matching lock mechanism. However, it requires implementation of the complete matching semantics in a thread-safe way (including thread synchronization) on top of MPI and introduces additional buffering, which is clearly suboptimal. The implementation would also require eager and rendezvous protocols for performance reasons and would also lose potential optimizations such as matching in hardware. Thus, such an implementation is highly undesirable from a user's perspective.

3 Extending the MPI Standard: Matched Probe

We have discussed issues with tread-safe matching in MPI and pointed at a problem in the specification. We have shown that all simple workarounds are either infeasible or incorrect (deadlock). Although our two mechanisms support correct semantics of MPI_Probe in threaded environments, they are nontrivial and limit either performance or concurrency significantly in the general case (any_src and any_tag possible). Thus, a general library implementation, as is required for new language bindings, cannot limit those scenarios and must pay the cost of general support. Further, both mechanisms duplicate work that an MPI implementation performs internally and limit hardware offload capabilities.

For these reasons, we must modify the MPI standard to eliminate the need to use these mechanisms, which would entail deprecating the existing probe operations. One possible solution would replace those operations with thread-safe versions that return a request that the application can later complete (in the original thread or not, but under application control) [7]. While in MPI-2.2, matching is done in probe and then again in receive, we decouple matching and receiving. We propose to add two new calls, mprobe and mrecv (and their nonblocking versions) to the MPI standard. We sketch the proposal here; a detailed version is available elsewhere [7]. The proposed *mprobe* returns a **message handle** that identifies a message (which is then unavailable in any other matching context). The proposed *mrecv* can then receive such a matched message. Listing 1.6 shows an example for thread-safe matching with a matched probe.

```
MPI_Message msg; MPI_Status status;
/* Match a message */
MPI_Mprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &msg, &status);

/* Allocate memory to receive the message */
int count; MPI_get_count(&status, MPI_BYTE, &count);
char* buffer = malloc(count);

/* Receive this message. */
MPI_Mrecv(buffer, count, MPI_BYTE, &msg, MPI_STATUS_IGNORE);
```

Listing 1.6. Matched probe example.

This mechanism reduces the user burden and minimizes the total number of locks required. We also enable efficient hardware matching and eager protocols. We discuss an implementation and possible issues in the following.

3.1 A Reference Implementation of Matched Probe

The matched probe proposal has been implemented as a proof of concept using Open MPI. Open MPI provides two mechanisms for message matching: One in which matching occurs inside the MPI library (used with network APIs such as Open Fabrics, TCP, and shared memory) and one in which matching occurs either in hardware or in a lower-level library (used with network APIs such as Myrinet/MX and Portals). The implementation of matched probe presented in this paper is based on MPI-level message matching. Issues with hardware level matching are discussed in Section 3.2.

The matched probe implementation does not significantly change the message matching and progression state machine of Open MPI. It adds an exit state from message matching (MPROBE in addition to PROBE and RECV), and adds an entry point back into the state machine. Open MPI tracks all unexpected messages (those that the matched probe operation can impact) as a linked list of message fragment structures, which includes source and tag. Communicators are separate channels and use separate lists. The list of unexpected messages is walked in an identical fashion for a probe and a receive. However, the fragment is removed from the unexpected message list and processed in the receive case.

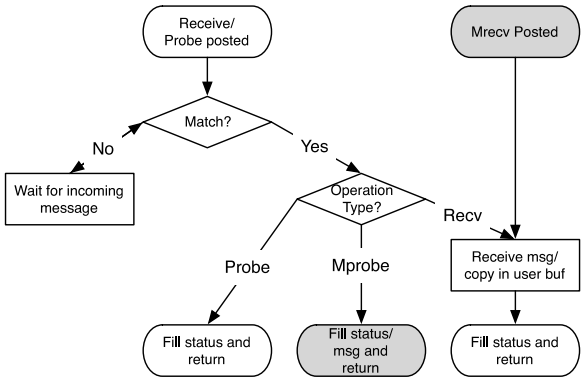


Fig. 1. High-level state diagram of MPI receive matching.

In the case of matched probe, the message fragment is removed from the unexpected message list (similar to a receive). It is then stored in the MPI_Message structure returned to the user. When the user calls MPI_Mrecv or MPI_Imrecv, the message fragment is retrieved from the MPI_Message structure that the user provided and the normal receive state machine is started from the point right after message matching.

3.2 Low-level Message Matching

The previously described implementation of Matched Probe for MPI-level matching, while straightforward, will not work if the lower level communication API provides message matching (such as Portals on the Cray XT line, Myrinet/MX, TPorts on Quadrics, and PSM on Qlogic). In these cases, the message matching state engine is not exposed to the MPI implementation, and may be executed on NIC hardware. In these cases, we must extend the interface of the lower-level API to support Matched Probe, likely with an implementation of similar complexity to the Open MPI implementation. Likewise,

firmware based hardware matching (TPorts and Accelerated Portals), adding entry points out-of and back in-to the firmware state machine should be straightforward.

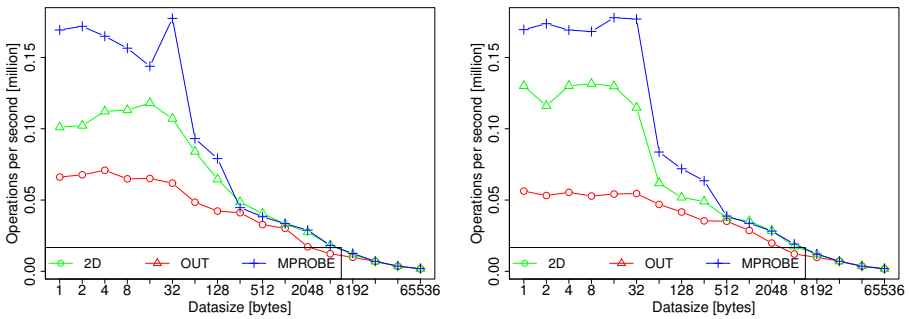
Hardware assisted matching presents a more complicated situation. Hardware designs would require modifications to support a matched probe. In addition, carrying the extra state to restart the state machine for a partially matched message could be cumbersome in hardware. However, since these designs are not in use, such designs have no bearing on the practical cost of this MPI extension. Thus, adoption of our extension requires a trade off between the benefits of making future designs of this type compatible our extension.

4 Performance Evaluation

We use two benchmarks that assess the performance and concurrency of the different mechanisms for thread-safe message reception. Both benchmarks and the two-dimensional locking (Section 2.1) are integrated in the publicly available Netgauge tool [8]. The benchmarks were run on Sif at Indiana University. Sif consists of Xeon L5320 1.86 GHz CPUs with a total of 8 processing cores per node running Linux 2.6.18 connected with Myrinet 10G. We used Open MPI revision 22973[5] using the TCP transport layer, configured with `--enable-mpi-thread-multiple`.

4.1 Receive Message Rate

Our first benchmark compares the message receive rate at a multithreaded receiving process with two-dimensional locking (2D, cf. Section 2.1) and matching outside MPI (OUT, cf. Section 2.2) for MPI-2.2 and the new matched probe (MPROBE, cf. Section 3) mechanism. In this test, 8 processes send to process 0, which uses 8 threads to receive the messages. Each process i sends its messages with tag i and each thread j either receives messages from process $j+1$ or any_src, with tag $j+1$ or any_tag.



(a) directed (b) any,any

Fig. 2. Message rate of different options for Open MPI on Sif.

Figure 2 shows the different message rates achieved by the two locking schemes and wrong matching with 8 processes sending to 8 threads on process 0. Figure 2(a) shows results for *directed* (i.e., neither any_src nor any_tag) and Figure 2(b) shows *any* (any_src and any_tag). The OUT and 2D implementations exploit knowledge of which wildcard pattern (any,any or directed) to expect (cf. Table 1). Both figures show

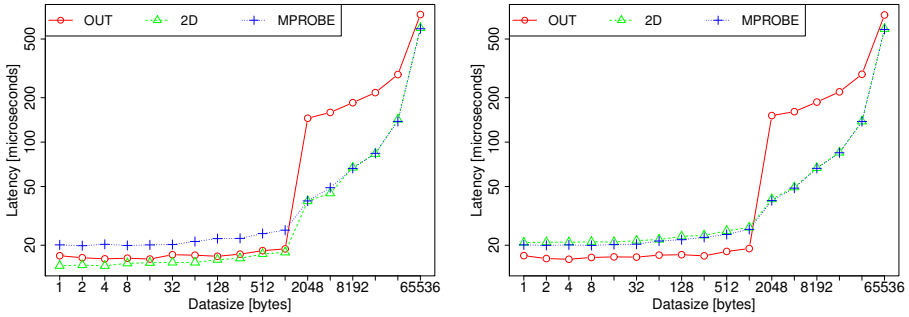
[5] available at: http://svn.open-mpi.org/svn/ompi/tmp-public/bwb-mprobe

significant performance differences between the approaches for small message-sizes. The rate of larger messages is bandwidth-bound and thus similar for all approaches. The two-dimensional locking scheme is faster than the matching outside of MPI, which must copy each message. However, our matched probe implementation outperforms both approaches and achieves the highest message rates.

4.2 Threaded Roundtrip Time

Our threaded roundtrip time (RTT) benchmark measures the time to transmit n messages between two processes with t threads each. It is thus somewhat similar to the overhead benchmark proposed by Thakur et al. [9]. Process 0 synchronizes its t threads with `pthread_barrier_wait` before each thread $j \in \{0..t-1\}$ sends n messages with tag j to process 1. The t threads at process 1 receive and send n messages from/to process 0 and each thread in process 0 receives n messages. The receives either use a specific tag $j \in \{0..t-1\}$ or any_tag and a specific source $s \in \{0, 1\}$ or any_src.

Figure 3 shows the latency overhead of the different locking schemes. For the



(a) any,any (b) directed

Fig. 3. Latency of different options for Open MPI on Sif.

any,any case in Figure 3(a), the current implementation of MProbe results in higher latency than both the 2D locking and matching outside MPI schemes. Latency increases mainly due to 2d-locking and outside MPI locking only using a single lock (cf. Table 1) based on the knowledge that only any,any receives are used while the matched probe implementation in MPI must handle the general case. As an aside, this example demonstrates the potential of additional info objects in MPI in which users could specify such constraints.

Figure 3(b) shows the latencies for the directed case (using integer tag and src values). For small messages, Mprobe is faster than 2-d locking due to the explicit removal from the queue (it only needs to be locked once). The outside MPI version is even faster for small messages because it receives the messages immediately and the copy overhead is low. However, for large messages, the copy overheads are dominating.

In Open MPI itself, there are two sources of unnecessary latency in the current MProbe implementation that we could remove with further development: creation of an additional request structure and an additional call to the progress engine. The extra request structure results in a small overhead, approximately 10 ns. The significant latency hit is from the additional progress engine calls, which could probably be mitigated through additional optimization. The issue is exacerbated by the high cost of entering Open MPI's progress engine when multithreaded support is enabled.

5 Summary and Conclusions

In this paper we describe the problem of receiving messages of unknown size in threaded environments. We show that often assumed simple solutions to the problem either introduce significant overheads or may lead to spurious deadlocks. We propose two advanced protocols to solve the problem in MPI-2.2. However, both protocols add various overheads to the critical paths. We then propose an extension of the MPI-3 standard that solves the matching problems. We show a reference implementation in Open MPI and discuss issues that might arise in hardware implementations.

Our performance analysis shows the benefits with regard to the message rates of the matched probe approach over the other protocols. We also analyze latencies for a multithreaded ping-pong benchmark. This analysis demonstrates that protocols on top of MPI can take advantage of special domain knowledge (only any,any calls), which serves as another good example for adding user assertions to the MPI standard.

Acknowledgments The authors thank all members of the MPI Forum that were involved in the discussions about matched probes, Douglas Gregor (Apple), and the anonymous reviewers. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-434306).

References

1. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4th 2009) http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.
2. Itakura, K., Uno, A., Yokokawa, M., Ishihara, T., Kaneda, Y.: Scalability of Hybrid Programming for a CFD Code on the Earth Simulator. Parallel Comput. **30**(12) (2004) 1329–1343
3. Rabenseifner, R.: Hybrid Parallel Programming on HPC Platforms. In: Proc. of the Fifth European Workshop on OpenMP, EWOMP'03, Aachen, Germany (2003)
4. Gropp, W.D., Thakur, R.: Issues in Developing a Thread-Safe MPI Implementation. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Proceedings. (2006)
5. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Thakur, R.: Toward Efficient Support for Multithreaded MPI Communication. In: European PVM/MPI Users' Group Meeting. (2008) 120–129
6. Gregor, D., Lumsdaine, A.: Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure. In: Proceedings of PPoPP 2008, New York, NY, USA (February 2008) 133–142
7. Gregor, D., Hoefler, T., Barrett, B., Lumsdaine, A.: Fixing Probe for Multi-Threaded MPI Applications (Revision 4). Technical report, Indiana University (January 2009)
8. Hoefler, T., Mehlan, T., Lumsdaine, A., Rehm, W.: Netgauge: A Network Performance Measurement Framework. In: High Performance Computing and Communications, HPCC. Volume 4782. (9 2007) 659–671
9. Thakur, R., Gropp, W.: Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE. In: European PVM/MPI User's Group Meeting, Proceedings. (2007)