Minimizing MPI Resource Contention in Multithreaded Multicore Environments

David Goodell,* Pavan Balaji,* Darius Buntinas,* Gábor Dózsa,† William Gropp,‡
Sameer Kumar,† Bronis R. de Supinski,§ Rajeev Thakur*

*Argonne National Laboratory, Argonne, IL 60439, USA, {goodell,balaji,buntinas,thakur}@mcs.anl.gov
†IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA, {gdozsa,sameerk}@us.ibm.com
‡University of Illinois, Urbana, IL 61801, USA, wgropp@illinois.edu
§Lawrence Livermore National Laboratory, Livermore, CA 94551, USA, bronis@llnl.gov

***Abstract*—With the ever-increasing numbers of cores per node in high-performance computing systems, a growing number of applications are using threads to exploit shared memory within a node and MPI across nodes. This hybrid programming model needs efficient support for multithreaded MPI communication. In this paper, we describe the optimization of one aspect of a multithreaded MPI implementation: concurrent accesses from multiple threads to various MPI objects, such as communicators, datatypes, and requests. The semantics of the creation, usage, and destruction of these objects implies, but does not strictly require, the use of reference counting to prevent memory leaks and premature object destruction. We demonstrate how a naïve multithreaded implementation of MPI object management via reference counting incurs a significant performance penalty. We then detail two solutions that we have implemented in MPICH2 to mitigate this problem almost entirely, including one based on a novel garbage collection scheme. In our performance experiments, this new scheme improved the multithreaded messaging rate by up to 31% over the naïve reference counting method.**

I. Introduction

Although MPI is the dominant programming model for large-scale systems, most researchers expect some hybrid programming model that uses threads within MPI processes to emerge as a result of the trend towards ever wider multicore nodes. Initially, this hybrid model may conform to a relatively simple MPI usage model in which only one thread invokes MPI functions (i.e., the `MPI_THREAD_FUNNELED` level of thread support). However, we anticipate that applications will eventually invoke MPI functions from several threads concurrently (i.e., the `MPI_THREAD_MULTIPLE` level of thread support). Thus, MPI implementations must evolve to provide efficient messaging from multiple threads.

Several barriers exist to providing this support. Some barriers involve basic issues for threaded programming such as ensuring that the code is reentrant and restricting access to certain low-level routines to one thread at a time. Other issues arise from MPI semantics. We explore one such issue: reference counting of MPI opaque objects.

In this paper we review the object lifetime management requirements imposed on MPI implementations by the MPI standard and illustrate the typical implementation based on reference counting. We make several contributions:

- Discussion of the limitations of this naïve solution;
- Design of two solutions, including one based on a novel garbage collection scheme;
- The implementation of these solutions in MPICH2;
- A simple experimental study to capture the performance upper bound for any reference counting mechanism within MPICH2 based on an implementation that performs no reference counting;
- Performance results for our reference counting solutions.

Overall, our results demonstrate that the solutions improve performance over the naïve implementation by up to 31% on our test platform. Furthermore, our algorithm is scalable in the number of threads and objects, whereas the naïve implementation is fundamentally nonscalable.

The rest of this paper is organized as follows. Section II discusses the MPI semantics that suggest reference counting. We then discuss details of MPICH2 relevant to any reference counting implementation in Section III. We present the naïve implementation in Section IV and our advanced solutions in Section V. In Section VI we discuss our performance results.

II. MPI Object Semantics

The MPI standard [1] specifies several object types, including `MPI_Request`, `MPI_Datatype`, and `MPI_Comm`. The user references and manipulates these opaque types only through handle values and function calls. Each object type has associated functions that create, manipulate, and free instances of the objects. For example, an `MPI_Datatype` can be created with `MPI_Type_vector`, used in a call to `MPI_Send`, and scheduled for destruction by `MPI_Type_free`.

Unless otherwise stated, we use the terms *MPI object* or *object* to imply opaque objects accessed via handles, and not directly manipulated objects such as `MPI_Status`. To minimize confusion with the standard C language function `free` and the "cost" definition of the word "free," we refer to MPI functions such as `MPI_Comm_free` as *user-release functions* and their invocations as *user releases*.

Most user releases are straightforward invocations of functions that include the word "free," although user releases of `MPI_Request` objects are more complex. While a call to `MPI_Request_free` serves as the user release of a

```
MPI_Type_vector( ..., &tv );
MPI_Type_commit( &tv );
MPI_Comm_dup( ..., &comm );
MPI_Irecv( buf,1,tv,0,1,comm, &req );
MPI_Comm_free( &comm );
MPI_Type_free( &tv );
 ... arbitrarily long computation
MPI_Wait( &req, &status );
```

Fig. 1. Valid Releasing of MPI Objects While They Are Still Needed

persistent request, the completion of other requests also serves as their user releases. Technically, correct MPI programs perform user releases of all objects, including through completions of requests, although many applications choose not to release objects with lifetimes through the invocations of `MPI_Finalize`, similar to programs that do not free all heap memory prior to exiting.

MPI semantics allows user releases before the lifetime of the object is completed. For example, users may invoke `MPI_Request_free` for a nonblocking send request before the send is guaranteed to have completed. Similarly, users can invoke `MPI_Comm_free` on a communicator and `MPI_Type_free` on a datatype before all requests that use those objects are completed. Thus, MPI semantics imply *internal* references for MPI implementations. In particular, each request typically implies internal references to a communicator and a datatype.

Figure 1 illustrates a valid MPI program that creates a datatype and communicator, initiates a receive using `MPI_Irecv`, and then performs user releases of the datatype and communicator. In this example, the datatype and communicator must be retained internally until the `MPI_Wait` on the request returned by the `MPI_Irecv` completes. An MPI implementation must retain these objects until all internal references are no longer needed, at which point it should reclaim the resources associated with the objects.

Failure to reclaim the resources associated with objects results in an implementation that will use all available memory prior to the completion of many long running applications. As described in Section IV, the most obvious solution uses a reference count mechanism. Each use increments the reference count, and each user release decrements it. In multithreaded environments, some mechanism must ensure that the increments and decrements are atomic.

Performing reference count updates nonatomically leads to errors. MPICH2 thread tests failed about 50% of the time with reference counts updated with ++ and −−, instead of with either lockfree equivalents or in a special critical section. However, our results presented in Section VI demonstrate that these naïve mechanisms to ensure correct reference counting significantly reduce performance compared to the single thread implementation. Thus, we need a solution that correctly and *efficiently* manages the object semantics in the presence of multiple threads. This paper explores such solutions.

III. MPICH2 Internals Background

MPICH2 [2] uses a custom memory allocator and integer-valued object handles to manage opaque MPI objects. The custom memory allocator provides several advantages, including enhanced performance and memory safety. By allocating objects via direct and indirect blocks, in a fashion similar to a UNIX file system inode, MPICH2 can assign an index value to each object. This object storage index is used as one value in a bit-field tuple of *(storage-area, mpi-type, storage-index)* that composes the object's handle value.[1] The *mpi-type* field indicates the MPI object type, such as `MPI_Comm`, `MPI_Datatype` or `MPI_Request`, to which the handle refers. The *storage-area* field indicates whether the underlying object is predefined and possibly stored specially, in a direct block array or in an array accessible via an indirect block.

In addition to this common object allocation layer used for all MPI objects, MPICH2 provides another small optimization above this layer for `MPI_Request` objects [3]. In general, an MPI implementation must allocate a request object for each communication operation such as `MPI_Send` or `MPI_Recv`. Similarly to high-performance multithreaded memory allocators, MPICH2 uses a thread-local pool of request objects to minimize object allocation mutex contention at high message rates. When the pool is empty and an object must be allocated, the thread acquires the allocation critical section and allocates a batch of requests to fill the pool. When a particular thread must free a request object, it returns the request to the thread-local pool. This approach provides performance benefits by amortizing the cost of acquiring the mutex over many requests (typically 128). It also provides improved cache locality for request objects because it prevents one thread from easily obtaining another thread's LIFO allocated request.

IV. Naïve Management of MPI Object Lifetimes

Naïvely, we can manage MPI object lifetimes through a simple reference counting scheme, well known in many contexts [4], [5], [6] since at least 1960 [7]. This section specifically describes the original MPICH2 implementation. Our discussions with other MPI implementers indicate that minor variations on this scheme are currently the most commonly used approach.

All MPI objects contain an integer reference count field that is appropriately aligned for atomic access on the current architecture. MPICH2 initializes this field to 1 when the object is allocated. At this point, the calling function has the sole reference to the new object. We subsequently adjust the reference count as follows:

1) Every time a function or object adds a new reference to an object, we increment its reference count atomically via a macro.
2) If a reference changes ownership from one function or object to another, the reference count remains unchanged.

[1] Predefined MPI datatypes such as `MPI_INT` are handled slightly differently in order to encode type size as an additional optimization.

3) Upon a user release or completion of an internal reference, we use an atomic macro to decrement the object's reference count and test whether it is now zero, in which case we reclaim the resources associated with the object.

The initial reference (e.g., resulting from routines like `MPI_Comm_create`) is actually the application's reference. Any other code in the object creation path that must retain a reference to the object after the application may have freed the object must add a reference of its own.

Let us examine this approach in the context of the code from Figure 1. After `MPI_Type_vector` and `MPI_Comm_dup`, the reference counts of both the datatype and the communicator are 1 for the sole references held by the user. The `MPI_Irecv` call creates the `MPI_Request` with a reference count of 1 and increments the reference counts of the datatype and communicator to 2. The additional references are internal references held by the request. The subsequent user release operations reduce both reference counts to 1. All reference counts remain unchanged during the application's computation phase. The `MPI_Wait` call releases the remaining reference on each request, datatype, and communicator. As each reference count drops to zero, the corresponding object is deallocated.

We can provide atomicity of reference count updates via a single global mutex, a mutex used for all reference counts, a mutex for reference counts by object type, a mutex for each specific object, or hardware-dependent atomic instructions. We usually prefer hardware-dependent atomic instructions, when available, because they provide fine granularity while consuming little or no additional space in the object. Most systems implement their system mutexes via these atomic instructions; and, thus, using those mutexes introduces additional overhead. One notable exception to this case is the dedicated lock-box hardware provided on IBM's Blue Gene/P, which provides comparable mutex performance without the usual `lwarx`/`stwcx` atomic instructions.

This approach provides several benefits. First, it is straightforward: a programmer must follow rules that are easy to remember and to apply to new code. Second, it does not require special language support or a runtime system to implement it. Virtually all MPI implementations are written in the C language, which does not perform garbage collection. Software does exist [8], [9] to add garbage collection support in C, but these packages often incur unacceptable space performance, experience "embarrassing pauses," and consume limited thread resources. These characteristics are generally undesirable but are unacceptable in high-performance computing contexts. Third, the scheme works with a mixture of pointers and integer handles, such as MPICH2 uses. Fourth, it always reclaims memory as soon as possible; objects do not linger beyond their final usage.

This approach has two minor downsides. First, it requires more programmer effort and is more error prone than true garbage collection such as provided by the Java programming language and runtime environment [10]. Second, it provides no protection against object leaks caused by circular references, although they do not occur in MPICH2 in practice.

However, the naïve reference counting approach incurs a significant performance cost. `MPI_Request` objects typically hold references to `MPI_Comm` and `MPI_Datatype` objects. These request objects are transient and result in frequent reference count updates to communicator and datatype objects. If multiple threads make communication calls at high frequency using shared communicators and types, the contention for the shared object reference counts can greatly degrade performance. Worse, this pattern typically scales poorly as the number of threads accessing the same shared objects increases. Depending on a number of architectural factors such as the cache coherency protocol, the exact set and fairness of atomic assembly instructions available, and thread scheduling policies, performance can fall off rapidly as thread counts increase.

V. Two Improved Solutions

To mitigate the aforementioned performance problems with the naïve solution, we developed two superior solutions.

A. Suppression of Predefined Object Reference Counts

We derive our first improved reference counting solution from two observations. First, predefined MPI objects such as `MPI_COMM_WORLD` are neither explicitly allocated nor deallocated by MPI programs. The lifetime of these objects always spans from the call to `MPI_Init` to the call to `MPI_Finalize`. Second, many MPI programs use only predefined communicators and datatypes.

Thus, a simple solution for a broad set of benchmarks and applications is to suppress reference counting completely on predefined communicator and datatype objects (such as `MPI_COMM_WORLD` and `MPI_INT`). MPICH2 request objects reference only these two object types, although we can easily apply this technique to other classes of objects. Reference counting still occurs for other object types as in the naïve solution and for all user-derived objects.

We easily implemented this suppressed reference counting in MPICH2. Because the handle value encodes both the *mpi-type* and *storage-area*, we can test for predefined `MPI_Comm` and `MPI_Datatype` objects with just a few inexpensive integer shifting and masking operations. Other MPI implementations that use pointers as handle values require only a dereference to access an `is_predefined` Boolean field in the object.

B. Reference Counting with Garbage Collection Hybridization

A more comprehensive solution to the MPI object life cycle problem involves a limited form of garbage collection for heavily contended object classes. We refer to these objects with potential for heavy contention as *garbage collection-managed objects (GCMOs)*. Under this scheme, when references to GCMOs are acquired by a *transient object*, such as an `MPI_Request`, a pointer to the GCMO is stored but the reference count in the referent object is not manipulated. When the transient object is deallocated, the pointer field to the referent GCMO is cleared. GCMOs still follow the

naïve reference counting rules outlined in Section IV for nontransient references, such as when one datatype is derived from another datatype, with one key exception: reference counts for freshly allocated objects start with a value of 2 instead of 1. The extra reference is logically held by the garbage collector.

Example: We explain the behavior of our garbage collection scheme with the help of an example: its effect on the code in Figure 1. After `MPI_Type_vector` and `MPI_Comm_dup`, the reference count of both `tv` and `comm` is 2. One reference is held by the user, and the other reference is held internally by the garbage collector. The `MPI_Irecv` call does not alter the reference count of either object, although it does create the `MPI_Request` object, `req`. This request is still reference counted in the naïve fashion, starting with one user reference and one or more internal references. During the user releases on the next two lines, the reference counts for `comm` and then `tv` both fall to 1.

At this point the application enters an arbitrarily long computation phase. Both `comm` and `tv` remain allocated during this period, a necessary behavior because of the pending nonblocking request. The `MPI_Wait` call releases both the user reference and all internal references to the request object, deallocating the request prior to the function's return. The communicator and datatype, however, remain live with a reference count of 1 that is held exclusively by the garbage collector.

At some later point in the program's execution (such as in another datatype or communicator creation function, or in `MPI_Finalize`), the MPI library will initiate a garbage collection cycle. Upon examining the set of outstanding requests and eligible datatypes/communicators, the collector will determine that `tv` and `comm` are both safe to reclaim and will deallocate them.

The following algorithm describes our approach more formally.

Algorithm: Let T be the set of all allocated transient objects that will serve as our root set. At allocation, GCMOs are placed into the live set of GCMOs, G_ℓ. When the reference count of a GCMO, $g_1 \in G_\ell$, falls to 1, only the garbage collector and some (possibly empty) set of transient objects, $T_1 \subseteq T$, hold references to g_1. At this point, g_1 is moved from G_ℓ to G_e, the set of GCMOs that are potentially eligible for reclamation.

The garbage collection algorithm is based on the traditional mark-and-sweep approach [11] and is detailed in Figure 2. We chose mark-and-sweep because it is simple to understand and to implement; in particular it does not suffer from the incompatibilities with the C language and high-performance computing environments that are typical of copying (sometimes called *scavenging*) collectors [8].

Several conditions can trigger the start of a garbage collection cycle. The application's call to `MPI_Finalize` is one such condition. Other conditions are a matter of policy decision; however, object creation functions such as `MPI_Comm_create` and `MPI_Type_vector` make excellent choices due to the existing required interaction with the object allocator. This also provides such functions the opportunity to reclaim memory if an insufficient amount is available to allocate new objects due to objects in G_e that would be successfully reclaimed by a garbage collection cycle.

```
let G_e ← the set of GCMOs referenced solely by the
    collector;
let T ← the set of all allocated transient objects (e.g.
    MPI_Requests);

enter ALLOCATION critical section;
/* assumes |G_e| available in O(1) time */
if |G_e| < 0 then
    /* no GCMOs can be freed by this
       cycle                                  */
    return;
end
/* initially assume all GCMOs in G_e are
    not live                                  */
for g in G_e do
    mark_not_live(g);
end
/* mark phase - scan transient objects
    and mark GCMOs                            */
for t in T do
    /* t may hold multiple refs, e.g.
       MPI_Comm/Datatype/Win                  */
    for g in t do
        mark_live(g);
    end
end
/* sweep phase - GCMOs that are not
    marked live can be reclaimed              */
for g in G_e do
    if ¬ is_live(g) then
        deallocate(g);
    end
end
exit ALLOCATION critical section;
```

Fig. 2. The Garbage Collection Algorithm

A thread initiates a garbage collection cycle by acquiring the mutex that protects object allocation. This approach prevents interference between garbage collection and object allocation. It also enables scanning of any relevant internal data structures used in the object allocator.

Discussion: This hybrid approach of reference counting plus garbage collection appears to be a nearly optimal solution to the MPI object lifetime problem. It eliminates virtually all of the contention present in the naïve approach, fully and correctly implements the semantics of MPI objects, and is relatively simple to implement in an existing reference counted code base. The only added cost is a certain amount of additional memory that is technically eligible to be returned

to the object allocation pool but is instead kept live by the garbage collector. This amount will vary significantly with application behavior, implementation decisions, and tunable collection policy parameter values.

Unlike many garbage collection schemes, this approach requires no additional threads or other asynchronous mechanisms such as signal handlers. This feature is particularly important in current HPC environments where thread counts may be constrained and context switches can be extremely disruptive to application performance [12].

In our implementation the set G_e is maintained as a singly linked list of objects, but this data structure choice is not specifically required by the algorithm. Similarly, T may be implicitly maintained by directly inspecting memory pools or more explicitly via a linked list data structure.[2]

Assuming that both T and G_e are maintained via $\mathcal{O}(n)$ traversal data structures, such as linked lists, then the cost of a garbage collection cycle is bounded by $\mathcal{O}(|G_e| + |T|)$. If the number of GCMOs referenced by each $t \in T$ is not fixed, then the upper bound becomes $\mathcal{O}(|G_e| + r_{\text{avg}}|T|)$, where r_{avg} is the average number of references held in each transient object. Note that G_ℓ is not present in either of these bounds. The algorithm is sensitive only to the number of outstanding transient objects and the number of GCMOs eligible for reclamation. Objects kept live by nontransient references have no impact on cycle time.

Clearly, if there are no eligible GCMOs ($|G_e| = 0$), then a cycle cannot free any objects. If $|G_e|$ can be evaluated in $\mathcal{O}(1)$ time, such as when tracked by a simple counter, then the marking phase may be easily skipped and the bound becomes piecewise:

$$\begin{cases} \mathcal{O}(|G_e| + |T|), & \text{if } |G_e| > 0 \\ \mathcal{O}(1), & \text{if } |G_e| = 0 \end{cases}$$

The question of when to initiate garbage collection cycles involves another implementation decision. `MPI_Finalize` is a necessary time to collect garbage objects, as it is guaranteed to be the last call to the MPI library (modulo calls like `MPI_Finalized`) in a valid MPI program. All communication is guaranteed to be complete by the end of this routine, so all MPI objects may be reclaimed at this point. In our implementation, MPI object creation functions such as `MPI_Comm_create` also trigger garbage collection cycles when $|G_e|$ grows larger than some threshold. We choose this threshold statically, but could easily calculate a dynamic threshold instead. Another possible collection point is the corresponding user-release function such as `MPI_Comm_free`. The primary advantage of collection at this point is to minimize $|G_e|$; however, this reduces the number of user-release operations over which the collection cycle cost is amortized. We did not specifically experiment with this collection point.

[2]Implementation note: one must ensure that $\forall t \in T$, references to GCMOs stored in t must be cleared upon allocation or deallocation when objects are recycled. Otherwise leaks may occur because of a false reference. If objects are not directly recycled, such as from `malloc`, then the GCMO reference fields must be reset upon allocation to ensure correctness.

Implementations must maintain T (and the value of $|T|$ if used to make GC policy decisions) in a manner that does not incur overhead, similar to the reference counting overhead we are working to avoid. For example, keeping track of $|T|$ by atomically incrementing or decrementing a shared counter on every modification of T would incur a severe performance penalty. As mentioned in Section III, MPICH2 currently uses a thread-local storage optimization [3] to manage request allocation. This optimization eliminates virtually all contention from request allocation.

VI. Performance Experiments

In this section, we first review the current state of multithreaded MPI benchmarking and explain our "Neighbor Message Rate Benchmark" that we use in our evaluation. We then present performance results on an IBM Blue Gene/P system. These results demonstrate that the our hybrid reference counting garbage collection approach attains substantial performance across the range of thread counts available on this system.

A. Benchmark Selection

The average number of cores per node in HPC systems is on the rise, providing greater potential for multithreading than ever before. Despite MPI's dominance as a programming model, however, few application benchmarks or real applications utilize the `MPI_THREAD_MULTIPLE` level of support. The reason is primarily the causality dilemma of high performance `MPI_THREAD_MULTIPLE` support. Applications do not utilize `MPI_THREAD_MULTIPLE` because MPI implementations do not often support it well (in terms of correctness and performance). However, MPI implementations do not support it well not only because implementing such support is difficult but also because applications do not use it. Hence, no canonical benchmark or application currently is suitable for examining the impact of our work. The *NAS Parallel Benchmarks (NPB)* [13] are commonly used to demonstrate application impact for message-passing programs; however, they are not multithreaded. The *NAS Parallel Benchmarks Multi-Zone (NPB-MZ)* [14] variants do use MPI+threads (via OpenMP), but they only use the `MPI_THREAD_FUNNELED` level of thread safety.

Several microbenchmarks are available to quantify MPI multithreaded performance at the `MPI_THREAD_MULTIPLE` level. The *OSU Micro-Benchmarks (OMB)* [15] include a multithreaded ping-pong benchmark, *osu_latency_mt*. Its description is as follows:

> The multi-threaded latency test performs a ping-pong test with a single sender process and multiple threads on the receiving process. In this test the sending process sends a message of a given data size to the receiver and waits for a reply from the receiver process. The receiving process has a variable number of receiving threads (set by default to 2), where each thread calls `MPI_Recv` and upon receiving a message sends back a response of equal size. Many

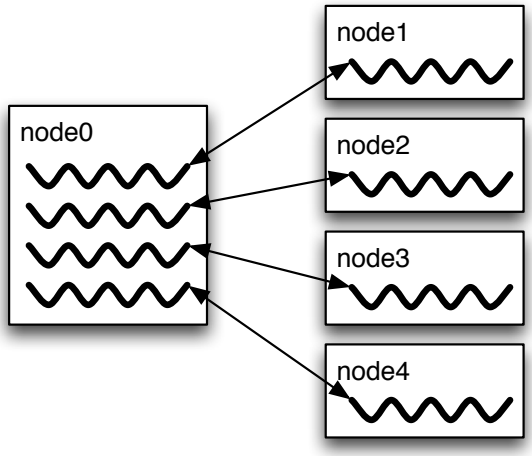

Fig. 3. Neighbor Message Rate Benchmark Communication Pattern

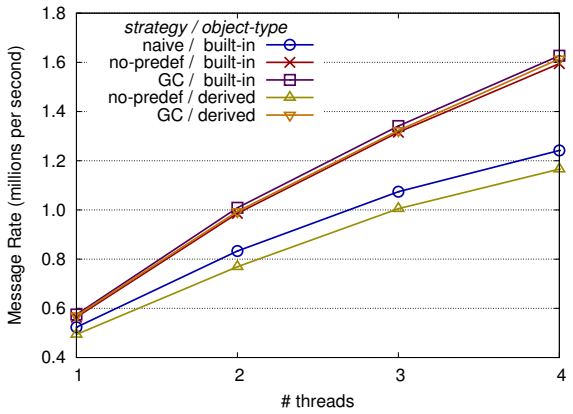

Fig. 4. Multithreaded Message Rate, Zero-Byte

iterations are performed and the average one-way latency numbers are reported.

While a valid measurement that occasionally may provide interesting data, this test primarily measures receive-side performance, including the degree of overall parallelism available in the entire networking stack. If a given platform does not provide parallelism between two processes at the operating system or network level, then overhead reductions in the MPI stack will be negligible when measured with this benchmark. Obtaining overall and network-level parallelism on a variety of platforms is beyond the scope of this paper.

None of these benchmark options satisfactorily captures the performance impact of the MPI object lifetime problem. Therefore we derive our own benchmark based on message rate measurements to quantify the impact of our optimizations.

B. The Neighbor Message Rate Benchmark

We posit that overall high-performance support for `MPI_THREAD_MULTIPLE` will be achieved by eliminating the numerous loosely related or unrelated implementation problems imposed by the intersection of the MPI Standard and architectural concerns. Many of these are still open problems, such as a providing a high-performance multithreaded receive queue implementation. The MPI object lifetime problem discussed in this paper is one such problem for which we provide a solution. Any one of these problems has the potential to completely serialize `MPI_THREAD_MULTIPLE` performance, hiding the impact of solutions to other problems.

To this end we present a minor variation on our threaded message rate benchmark [3] that sends and receives messages bidirectionally in order to provide the most parallel and efficient baseline possible. This benchmark measures the aggregate message rate for N threads in a single MPI process, each sending to and receiving from a corresponding peer process on a separate node. That is, if N threads are communicating on *node0*, then there will be N other nodes: *node1* through *nodeN*. This communication pattern is illustrated in Figure 3.

On our experimental platform, an IBM Blue Gene/P system, the baseline threaded message rate performance is best when access to network hardware is not serialized. The most straightforward way to achieve this situation is for each thread to communicate with a separate neighbor node. We dub this the "Neighbor Message Rate Benchmark." For the Blue Gene/P architecture, N is limited to 4 threads per node.

Each iteration of the test involves each thread posting 12 nonblocking receives and 12 nonblocking sends from/to the peer thread followed by a call to `MPI_Waitall` to complete the requests. Each thread executes 10 warm-up iterations before timing 10,000 more iterations. All messages sent and received in this paper are zero bytes long in order to minimize the impact of data transfer times on the measurements. The benchmark reports the total number of messages sent divided by the elapsed time in seconds. This value is not intended to measure application-like performance; it is primarily useful for differential performance comparisons between different implementation techniques on the same platform and avoids some of the well-known microbenchmarking pitfalls [16], [17].

C. Blue Gene/P Performance Evaluation

Figure 4 shows the message rates achieved on an IBM Blue Gene/P system at Argonne National Laboratory when using the various strategies described in Sections IV and V. Garbage collection clearly outperforms the other strategies by a substantial margin, sending and receiving approximately 1.6 million messages per second with four threads. All strategies fail to achieve optimal speedup because developing an MPI implementation that scales linearly with the number of threads is still an open problem. The absolute numbers do not matter as much as the performance relative to the naïve approach, shown in Figure 5. With four threads the garbage collection scheme outperforms the naïve approach by approximately 31%.

All strategies perform at least slightly worse when derived objects (from `MPI_Comm_dup` and `MPI_Type_vector`) are used. The MPI implementation must perform a small

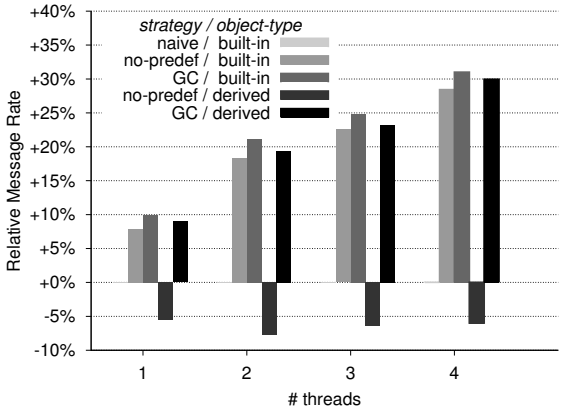

Fig. 5. Message Rate Relative to the Naïve Approach

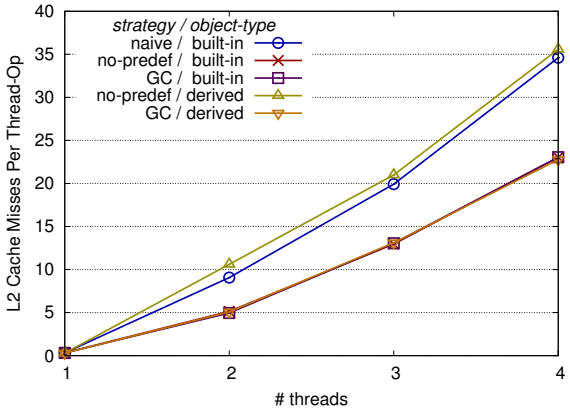

Fig. 6. Average L2 Cache Misses per Communication Operation per Thread

amount of additional work to use and maintain user-derived objects when compared with predefined objects such as `MPI_INT` or `MPI_COMM_WORLD`. We note, however, that the performance of our garbage collection method when user-derived objects are used is nearly indistinguishable from the performance it attains with predefined objects.

Figure 5 clearly illustrates the increasing performance gap between the naïve approach and our improved techniques. In the case of garbage collection versus the naïve approach, this gap grows from 10% at one thread to 31% at four threads. The lack of scalability in the naïve approach is primarily a consequence of the increasing cache coherency contention, demonstrated in Figure 6. This figure plots the average number of L2 cache misses per communication operation (`MPI_Send` or `MPI_Recv`) per thread when running the message rate benchmark. With four threads the L2 cache miss counts for the naïve approach are approximately 1.5 times greater than for the garbage collection case.

VII. Related Work

We previously considered MPI implementation thread safety requirements [18], briefly touching on MPI reference counting semantics. We also proposed a small suite of benchmarks for evaluating overall threaded MPI performance [19]. While those benchmarks provide useful ways to quantify overall threaded MPI implementation performance, they do not provide a "torture test" for threaded MPI performance as our neighbor message rate benchmark does. Many other experimental multithreaded MPI implementations exist [20], [21], [22], [23], [24].

Valois introduced *lockfree reference counting (LFRC)* [4] in order to deal with the ABA problem[3] in his lockfree data structure algorithms. While we are not specifically dealing with the ABA problem here, LFRC provides the basis for the naïve implementation discussed in Section IV. Detlefs et al. further focused on LFRC as a memory reclamation technique for lockfree data structures [6].

Hart et al. provide an excellent comparison [25] of the LFRC, *hazard-pointer-based reclamation*, *epoch-based reclamation*, and *quiescent-state-based reclamation* memory reclamation schemes. Modulo LFRC, these approaches are not directly comparable with our techniques because they deal primarily with lockfree data structure manipulations, especially for read-mostly workloads in an operating system context. However, we might be able to use one or more of these techniques in concert with our scheme and known concurrent garbage collection algorithms [26] in order to provide a lockfree or nearly lockfree version of the algorithm from Figure 2.

Treumann [27] recently proposed adding a facility to the MPI standard that would allow users to assert certain program behavior for an MPI implementation. Such a mechanism could conceivably be used by the user to promise the implementation that premature user releases (such as shown in Figure 1) would not occur in a given program. Such a facility might permit implementations to deallocate MPI objects immediately at user release time without maintaining reference counts or performing garbage collection. However, users would have to update application code to use the new facility, which is clearly not preferable to the seamlessly available performance improvements discussed in Section V.

VIII. Conclusions and Future Work

We have presented the first in-depth discussion and analysis of MPI object lifetime management issues. The naïve solution to this problem was then detailed along with two superior solutions. One improved solution is a novel hybrid of the naïve reference counting solution and garbage collection. This garbage collector achieves nearly optimal performance by entirely removing operations from the messaging performance critical path. We presented several experiments that illustrate the performance benefits of our algorithm. In particular, a message rate benchmark demonstrated that our approach improved

[3]The ABA problem occurs when using the compare-and-swap (CAS) atomic operation to manipulate pointers. If a pointer changes values from A to B and then back to A again between the time that pointer is read and then manipulated with CAS, the CAS will succeed, often corrupting the data structure. A particularly nasty variant occurs when objects are recycled: A is supplanted by B, A's referent object is freed and then reallocated immediately by another thread, and then the "new" A object supplants B.

performance of support for `MPI_THREAD_MULTIPLE` by 31% with four threads.

Treumann's assertions [27] deserve study as another possible solution to the MPI object lifetime problem. We intend to examine them in the future.

Our approach is susceptible to deadlock in the presence of thread failures or cancellations. In the future we intend to develop a lockfree or nearly lockfree version of our approach, possibly by using one or more of the techniques described by Hart et al. [25].

We also intend to study policy decisions and implementation issues in the garbage collection algorithm presented in Section V. Data structure choices and threshold selection both have the potential to affect garbage collection cycle times substantially. Moreover, we intend to port the implementation of our algorithm to other platforms in order to measure the performance impact of different architectural concerns such as cache coherency protocol, memory hierarchy design, and number of cores.

ACKNOWLEDGMENTS

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357 and Award DE-FG02-08ER25835, and by the National Science Foundation Grant #0702182. Portions of this work were performed under the auspices of the U.S. Department of Energy, supported by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

REFERENCES

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 2.2," September 2009, http://www.mpi-forum.org/docs/docs.html.

[2] "MPICH2," http://www.mcs.anl.gov/mpi/mpich2.

[3] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Fine-grained multithreading support for hybrid threaded MPI programming," *International Journal of High Performance Computing Applications*, vol. 24, no. 1, pp. 49–57, 2010.

[4] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. ACM New York, 1995, pp. 214–222.

[5] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl*, 3rd ed. O'Reilly Media, Inc., July 2000, ch. 8, pp. 266–267.

[6] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. J. Steele, "Lock-free reference counting," *Distributed Computing*, vol. 15, no. 4, pp. 255–271, December 2002.

[7] G. E. Collins, "A method for overlapping and erasure of lists," *Communications of the ACM*, vol. 3, no. 12, pp. 655–657, December 1960.

[8] H. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software – Practice and Experience*, vol. 18, no. 9, pp. 807–820, 1988.

[9] "Boehm-Demers-Weiser garbage collector," http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[10] J. Gosling and H. McGilton, "The Java language environment," Sun Microsystems Computer Company, May 1995.

[11] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[12] P. H. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, no. 1, pp. 3–16, 2008.

[13] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," NASA Ames Research Center, Moffett Field, CA, NAS Technical Report NAS-95-020, 1995.

[14] R. Van der Wijngaart and H. Jin, "NAS Parallel Benchmarks, multi-zone versions," NASA Ames Research Center, Moffett Field, CA, NAS Technical Report NAS-03-010, 2003.

[15] "OSU Micro-Benchmarks (OMB)," http://mvapich.cse.ohio-state.edu/benchmarks/.

[16] W. Gropp and E. Lusk, "Reproducible measurements of MPI performance characteristics," in *Recent advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users Group Meeting*, ser. Lecture Notes in Computer Science, vol. 1697. Springer, 1999, pp. 11–18.

[17] K. D. Underwood, "Challenges and issues in benchmarking MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, ser. Lecture Notes in Computer Science, vol. 4192. Springer, 2006, pp. 339–346.

[18] W. Gropp and R. Thakur, "Thread safety in an MPI implementation: Requirements and analysis," *Parallel Computing*, vol. 33, no. 9, pp. 595–604, September 2007.

[19] R. Thakur and W. Gropp, "Test suite for evaluating performance of multithreaded MPI communication," *Parallel Computing*, vol. 35, no. 12, pp. 608–617, December 2009.

[20] E. D. Demaine, "A threads-only MPI implementation for the development of parallel programs," in *Proceedings of the 11th International Symposium on High Performance Computing Systems*, July 1997, pp. 153–163.

[21] F. García, A. Calderón, and J. Carretero, "MiMPI: A multithread-safe implementation of MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*. Lecture Notes in Computer Science 1697, Springer, Sep. 1999, pp. 207–214.

[22] H. Tang and T. Yang, "Optimizing threaded MPI execution on SMP clusters," in *Proceedings of the 15th ACM International Conference on Supercomputing*, Jun. 2001, pp. 381–392.

[23] T. Plachetka, "(Quasi-) thread-safe PVM and (quasi-) thread-safe MPI without active polling," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting*. Lecture Notes in Computer Science 2474, Springer, Sep. 2002, pp. 296–305.

[24] S. G. Caglar, G. D. Benson, Q. Huang, and C.-W. Chu, "USFMPI: A multi-threaded implementation of MPI for Linux clusters," in *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2003.

[25] T. Hart, P. McKenney, A. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *Journal of Parallel and Distributed Computing*, vol. 67, no. 12, pp. 1270–1285, 2007.

[26] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: An exercise in cooperation," *Communications of the ACM*, vol. 21, no. 11, pp. 966–975, November 1978.

[27] R. Treumann, "Another pre-preposal*[sic]* for MPI 2.2 or 3.0," http://lists.mpi-forum.org/mpi-forum/2008/04/0076.php, April 2008.