Preprint
UCRL-JC-138650

Dynamic Software Testing of MPI Applications with Umpire

J.S. Vetter and B.R. de Supinski

This article was submitted to
SC2000, Dallas, Texas, November 4-10, 2000

July 24, 2000

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

Approved for public release; further dissemination unlimited

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (423) 576-8401
http://apollo.osti.gov/bridge/

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161
http://www.ntis.gov/

OR

Lawrence Livermore National Laboratory
Technical Information Department's Digital Library
http://www.llnl.gov/tid/Library.html

Dynamic Software Testing of MPI Applications with Umpire

Jeffrey S. Vetter Bronis R. de Supinski
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California, USA 94551
{vetter3,bronis}@llnl.gov

Abstract

As evidenced by the popularity of MPI (Message Passing Interface), message passing is an effective programming technique for managing coarse-grained concurrency on distributed computers. Unfortunately, debugging message-passing applications can be difficult. Software complexity, data races, and scheduling dependencies can make programming errors challenging to locate with manual, interactive debugging techniques. This article describes Umpire, a new tool for detecting programming errors at runtime in message passing applications. Umpire monitors the MPI operations of an application by interposing itself between the application and the MPI runtime system using the MPI profiling layer. It, then, checks its MPI behavior for specific errors. Our initial collection of programming errors includes deadlock detection, mismatched collective operations, and resource exhaustion. We present an evaluation that demonstrates the effectiveness of this approach.

1 Introduction

Message passing serves as an effective programming technique for exploiting coarse-grained concurrency on distributed computers as evidenced by the popularity of the Message Passing Interface (MPI) [7, 19]. Unfortunately, debugging message-passing applications can be difficult. In fact, some accounts of HPC software development report that debugging and testing can consume almost 50% of application development time [10, 16]. Software complexity, data races, and scheduling dependencies can make simple programming errors very difficult to locate with manual debugging techniques. Worse, few debugging tools are even targeted to MPI abstractions. Of those tools that provide MPI support, they generally force users to analyze their software iteratively, discovering errors by interactively probing message envelopes and queues using low-level commands. In reality, users employ a spectrum of manual techniques to infer explanations for MPI programming errors, basing their conclusions on debugger interaction, explicit modifications to the source code, message tracing, and visualizations. This interactive analysis is time-consuming, error-prone, and complicated, especially if the user must analyze a large number of messages, or if the messages are non-deterministic. This unfortunate situation forces users to design applications conservatively and to avoid advanced MPI operations and optimizations.

0-7803-9802-5/2000/$10.00 © 2000 IEEE.
This work was performed under the auspices of the U.S. Dept. of Energy by University of California LLNL under contract W-7405-Eng-48.
LLNL Document Number UCRL-JC-138650.
Appears in Proc. SC 2000 (electronic publication).

1.1 Key Insights and Contributions

Quite simply, if users could automatically test for common errors in their application's MPI behavior, they could use this test information to expedite their application software development. To address this issue, we have developed Umpire, an innovative tool that dynamically analyzes any MPI application for typical MPI programming errors. Examples of these errors include resource exhaustion and configuration-dependent buffer deadlock. Umpire performs this analysis on unmodified application codes at runtime by means of the MPI profiling layer. By interposing Umpire between the application and the MPI runtime system, we maintain portability while recording sufficient information about every MPI operation issued by the application to make reasonable judgements about the application's behavior.

Our overall goal with Umpire is to make users more productive by systematically detecting programming problems before the user is forced to manually debug their application. More importantly, as users expose new MPI programming problems, we can add them to Umpire's suite of verification algorithms.

1.2 Related Work

The research community has documented an extensive set of complex programming challenges associated with multithreaded and distributed applications. To address these challenges, researchers have proposed solutions [1-5, 8, 9, 12-15, 20] to help users manage this complexity during the software development. In contrast to interactive debugging with a contemporary multiprocessor debugger, such as

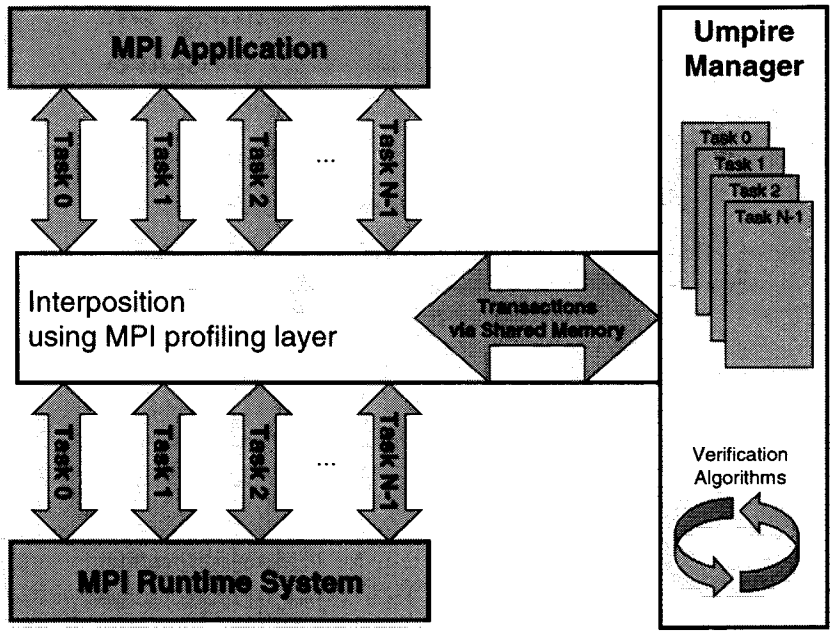

Figure 1: Umpire Architecture.

Etnus' TotalView, most of these tools rely on some level of automation to drive error detection. They automatically analyze applications for errors by using additional semantic knowledge about a well-defined abstraction within the application, which, in turn, allows the tool to test any sequence of operations for errors. This research, as a result, underscored the usefulness of such techniques and warranted the development of successful commercial programming tools including both dynamic analysis tools such as Rational's Purify [17], KAI's Assure [11], Compaq's VisualThreads (also Eraser [18]), Compaq's Atom [6], and static analysis tools, such as APR's FORGE, FLINT (FORTRAN LINT), and Sun's Locklint [21]. Despite MPI's popularity, few automated debugging and testing tools exist for MPI. We know of no other work that automatically tests an MPI application for erroneous behavior.

Dynamic tools strike a balance between knowledge of the application and . Statically reasoning about program semantics is challenging.

1.3 Paper Organization

The balance of this paper discusses these issues in more detail. First, in Section 2, we present an overview of our system design. Next, in Section 3, we detail specific programming errors that our system highlights. Section 6, then, provides an evaluation of Umpire. Finally, Section 7 concludes this paper.

2 System Overview

As Figure 1 illustrates, our prototype tool, Umpire, exploits the MPI profiling layer to capture information about the execution of an MPI application. Originally designed for collecting performance information, the profiling layer provides sufficient access to the interactions between the application and the MPI runtime for Umpire to reason about the application's MPI operations while remaining portable across a wide variety of MPI implementations. Although we could possibly increase the number of verification checks by melding the profiling layer information with the MPI runtime internals, it would also bind Umpire to specific MPI implementations. As we explore in Section 3, writing verification algorithms for the MPI specification using the profiling layer is a challenging effort per se. Customizing each of these algorithms to exploit the internals of popular MPI implementations would be prohibitively expensive.

```
1064 MPI_Isend pre
        world_rank = 0
        seq_number = 117
        pc = 100004a4
        count = 128
        datatype = 8
        dest = 1
        tag = 31
        comm = 0
        chksum = 961937199
        arequest = 804397860
        request = <unprintable>
```

Figure 2: MPI_Isend operation record for Umpire.

2.1 Architecture

Umpire decomposes into two basic components in Figure 1: the collection system and the manager. As we mentioned earlier, Umpire gathers information about an application's MPI behavior through the profiling layer. As each MPI task invokes a MPI library routine, Umpire captures information about the MPI call including the parameter list and possibly, derived information, such as a buffer check sum.

The information included in Umpire transactions varies with individual MPI calls. In many cases, a transaction includes the values of all parameters for the MPI call. Some transactions log additional information that might include the addresses of certain parameters or a checksum, computed from a parameter's value. All Umpire transactions include the return program counter (PC) of each MPI call. This PC allows the mapping of specific errors to their locations within the application source code.

With this information in hand, Umpire performs two functions: a local check and a global check. The local check, if necessary, verifies the MPI operation using task local information. One example of a local check is the check sum on non-blocking send buffers. Because Umpire does not store the entire send buffer, it needs to be able to reconcile initiation and completion of non-blocking operations. For instance, when Umpire encounters a MPI_Wait, it needs to locate the matching operation, say MPI_Isend, and perform the check sum calculation using the message envelope from the matching MPI_Isend. This type of check can only be performed at the task local level.

The second function performed by Umpire for a call is a global check. After the local check, Umpire sends this MPI operation information to the Umpire Manager with a transaction via a shared memory buffer. The most important verifications performed by Umpire occur at the global level; the Umpire Manager completes these checks.

2.2 Manager

The Umpire Manager, which is a thread in task 0, collects these transactions. The Manager processes the transaction by checking it with a verification algorithm, storing it for future reference, or both. It discards the transaction record once it determines that the MPI call cannot lead to any unsafe or erroneous conditions. Each MPI operation may involve one or more verification algorithms; we discuss these algorithms in Section 3.

Internally, the manager has several data structures that record information about the ongoing global state of the MPI application. Two major data structures are the MPI operations queues and the resource registry. The manager has one MPI operation queue for each MPI task. Each queue is a chronologically ordered list of MPI operations for each task. The manager deletes operations from each queue once it determines that the operation cannot lead to an error in the application. The other important data structure within the Umpire Manager is the resource registry. This registry records a description of MPI resources, such as communicators and derived types. This data structure allows Umpire to track usage of these resources and to map upcoming MPI operations appropriately.

During MPI_Init, Umpire sets up the shared memory communication buffers between each task and the manager. Also, task 0 spawns a kernel level thread to execute the Umpire manager. The manager operates asynchronously, communicating with all MPI tasks via the shared memory when necessary.

```
While any MPI task remains do
    Get pending operation info from
        a task via shared memory
        buffer.
    Verify operation data
        consistency and checks on
        parameters.
    Update Manager data structures
        including operation queues
        and resource registry.
    Verify global integrity by
        generating a dependency
        graph of operations in the
        operation queues.
    Return result to task i.
```

Figure 3: Manager's control loop.

Figure 3 lists the Umpire Manager's control loop. As each task executes MPI calls, it notifies the manager of the impending call by placing a record of the operation in the shared memory buffer. The manager takes this operation from the buffer, and performs a series of checks on the record for consistency. Next, the manager updates it internal data structures with information from the operation. If the operation can contribute to a dependency cycle, it records the operation in the corresponding operation queue. If the operation creates or modifies a resource, the manager updates the resource registry. After updating the data structures, the manager carries out a series of correctness checks on the global state of the MPI application. The most important of these correctness checks is a deadlock detection algorithm that generates a dependency graph on the operations in the manager's operations queues. Section 4.1 provides details on this deadlock detection algorithm. Finally, the manager transmits a result to the task that notified it of the operation, which was blocked while the manager performed its verification. A more aggressive strategy would allow the tasks to proceed while the manager verified correctness concurrently; however, in some cases, this strategy could cause the MPI application to exit prematurely, possibly before the manager has uncovered the error.

2.3 Key Design Decisions

Two key design decisions contribute to Umpire's reasonable performance. First, Umpire's interposition layer uses shared memory to ship transactions to the manager. Although this decision limits the scalability of Umpire, it drastically reduces the latencies in Umpire communication from milliseconds to microseconds. We are considering the design of a distributed memory version of Umpire. Yet the negative performance and scalability implications of sending messages to and processing messages with a centralized manager will most likely force a migration of the current design to one that uses a distributed algorithm for deadlock detection.

Second, the interposition imposed by Umpire is flexible; different MPI calls have different transaction semantics that are tailored to the call's contribution to the verification algorithms. In some cases, Umpire must generate two transactions for each MPI call: one immediately before the call to the MPI runtime system and one immediately after the call returns. For many MPI calls, Umpire does not

require at least one of these transactions. Most MPI calls that cannot lead to unsafe or erroneous programs are simply not registered with the manager.

3 Programming Errors in MPI

We focus our evaluation on the message-passing interface (MPI) [7, 19] because MPI serves as an important foundation for a large group of applications, and because MPI is a modular library that has both well-defined syntax and semantics. Concisely, MPI provides a wide variety of communication operations including blocking and non-blocking sends and receives; blocking collective operations, such as broadcast and global reductions; and data type creation and manipulation functions. Currently, we focus our attention on an important subset of heavily-used MPI operations and their respective programming errors. For example, currently, Umpire does not support MPI_Cancel and MPI_Waitany. Our strategy is applicable to other MPI programming errors; we are expanding Umpire to detect a comprehensive range of unsafe or erroneous MPI conditions as identified by the MPI standard.

Now, to demonstrate the capabilities of our system, we outline the following representative programming errors and explain Umpire's verification algorithms for each error. These errors include configuration-dependent deadlock, mismatched collective operations, errant writes to send buffers, and resource tracking errors (including leaks). In the following section, our examples show straightforward sequences of MPI operations that create these errors. In real applications, these errors rarely occur in such concise sequences. Often, they occur in different subroutine calls and they are almost always hidden within control flow constructs.

4 Deadlock

There are many ways to create a deadlock in an MPI program. Although most deadlocks are self-evident, some deadlocks manifest themselves in apparently unrelated error conditions. Worse, some MPI deadlocks depend on configuration parameters, such as MPI message buffer space. The MPI standard states that "*programs with these configuration dependent deadlocks are valid, although they are unsafe; they should be avoided in portable programs.*" Users may develop an application on a platform with one set of configuration parameters, but the application deadlocks when it is executed with a different set of configuration parameters or on another platform.

A considerable amount of related research addresses deadlocks and race conditions in message passing applications. Clearly, a developer has a straightforward manifestation of an application that deadlocks. However, MPI presents special implementation challenges. Although straightforward deadlocks remain a challenge, MPI's eager message protocols can introduce a configuration dependent deadlock and its collective operations can create a communicator topology deadlock.

4.1 Deadlock Detection

Two basic methods exist for deadlock detection in distributed systems: detecting cycles in *wait-for* graphs prior to execution, and using timeouts to break the deadlock after it occurs. Umpire uses *wait-for* graphs, although it can gracefully terminate a deadlocked application using timeouts.

The Umpire Manager tracks blocking MPI communication calls, including collective operations, communicator management routines, and completions of non-blocking requests using its set of MPI operation queues for each task. Using this information, Umpire constructs a *wait-for* graph from the ordered operations in each queue. Umpire's recursive deadlock detector attempts to construct a cycle of dependencies from the blocking operations in these queues. Umpire discards transaction records for these calls when it determines that the operations cannot lead to a deadlock.

Realistically, since deadlocks can involve constructs that are not part of MPI, our tool can also time-out after a user-specified period. The timeout period is configurable, so a user can adjust the threshold to accommodate their platform and application. When Umpire times out, it reports the contents of the blocking request queues, as well as other information about the current state of each MPI task.

4.1.1 Blocking MPI Calls

MPI's blocking operations play a significant role in MPI application deadlocks. Unlike non-blocking operations, blocking operations will not return to the application until MPI can allow the application to reuse resources specified in the call.

4.1.2 Non-blocking MPI Calls

Non-blocking calls, in contrast, may return to the application before the operation started by the call completes and before the user is allowed to reuse resources specified in the call. Typically, MPI associates a request object with an operation when it is started with a non-blocking call. The operation completion becomes visible to the user by a completed request, signified by a successful MPI_Test or MPI_Wait operation.

4.1.3 Collective Calls

Although MPI's collective operations are not necessarily synchronizing, all members of the process group must execute collective calls over the same communicator in the same order. For collective operations, matching calls must be at the queue heads of all tasks in the call's communicator. Figure 5 shows the queues that correspond to one possible execution of the mismatched collective operations. In this example, tasks 0-2 have complete their broadcasts and call the barrier, before task 3 informs the manager that it has called the barrier. When the manager processes the transaction for task 3's barrier, it detects the cycle in the *wait-for* graph, indicating a deadlock. At this point, Umpire generates a report and aborts the program,

reporting the sequence of MPI calls causing the deadlock. Since unsafe programs are valid, we are adding a mechanism for users to specifically enable and disable individual verification algorithms using MPI_Pcontrol. MPI 1 does not support non-blocking collective communication operations.

4.1.4 Wildcards

For message receive operations, MPI provides wildcards for both the source and the tag in the message envelope. These values, MPI_ANY_SOURCE and MPI_ANY_TAG indicate that any source and any tag, respectively, are acceptable for the incoming message envelope.

4.2 Deadlock Algorithm

The deadlock algorithm attempts to find a sequence of dependencies in the operation queues that create a cycle. Using a recursive function, Umpire begins with the oldest element in the head of the queue that changed most recently. It searches for the first blocking operation in the queue. Umpire, then, determines if the operation can complete. To accomplish this, Umpire might probe the queues of other tasks. If the operation can complete, then Umpire updates all related queues, and returns. If the target queue has a blocking operation that prohibits the completion of the current operation, then Umpire follows the dependencies generated by that blocking operation. If the queue does not contain any blocking operations or a matching operation, then it simply returns because it can make no determination about the current operation. If at any time Umpire must follow a dependency to the task that initiated the dependency search, then it declares that a deadlock exists and aborts the application.

4.3 Configuration Dependent Buffer Deadlock

Configuration dependent deadlocks can arise from complex code interactions; however, Figure 6 illustrates a simple configuration dependent deadlock---a common MPI programming error. This example executes properly when the MPI configuration parameter for the eager send limit is set slightly above 60,000 bytes. However, when the limit drops below this threshold, the MPI application deadlocks. Unwittingly, the user has introduced a configuration-dependent deadlock into their MPI application. Given the large number of configurable parameters on current MPI runtime systems and the ease with which these parameters can change, it is important to expose these errors consistently on any platform.

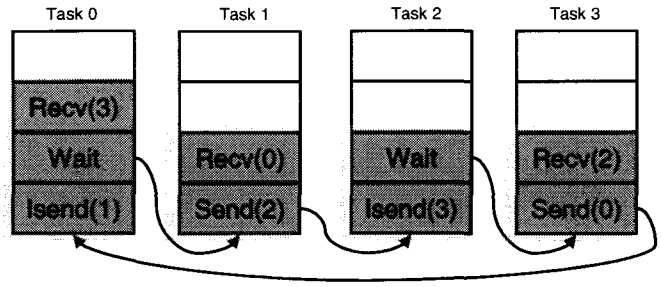

Figure 4: Dependency cycle for example of configuration dependent buffer deadlock.

Figure 4 illustrates a cycle in the dependency graph for a configuration-dependent buffer deadlock example. Only blocking operations can contribute to this cycle; hence, the initiation of non-blocking operations, such as MPI_Isend, cannot contribute to a deadlock cycle. MPI's eager message protocol allows send routines to complete if their message size is relatively small. In this example, the sends in tasks 0, 1, and 3 complete, which allows each task to post its receive. When the manager posts the MPI_Wait to the operation queue for Task 2, the manager detects a possible deadlock because it has completed a cycle in the dependency graph.

In this example the cycle does not manifest itself until this MPI_Wait operation, a blocking operation, is posted, so the corresponding receives is in the queue, even though a dependency graph clearly reveals a cycle.

4.4 Mismatched Collective Operations

Figure 7 shows mismatched collective operations. Although not immediately obvious, this programming error is another common type of MPI deadlock. The MPI standard requires that programs invoke collective communications---convenient features of MPI that support communication across all tasks or within derived communicators---so that deadlock will not occur, whether or not the particular MPI implementation forces a collective synchronization. This error frequently occurs during the development of MPI codes, where barriers and broadcasts can occur within subroutine calls. In many cases, the MPI runtime system manifests such deadlocks through cryptic messages. In one MPI implementation that does not distinguish between internal messages for separate collective operations, our example in Figure 7 generates a "message too large" error which results when an underlying message send of the broadcast matches a message receive of the barrier.

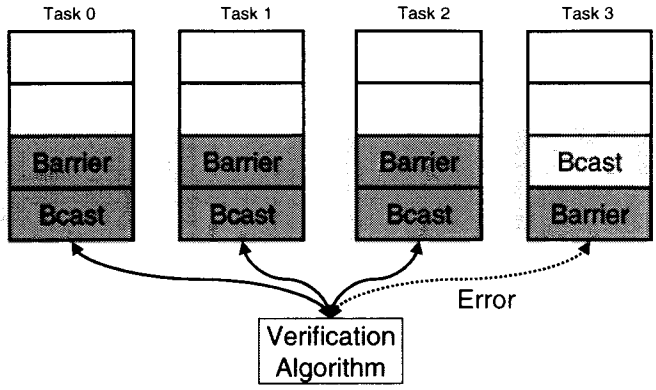

Figure 5: Umpire's deadlock detection queues for 4 tasks within a single communicator.

Detecting where violations of these ordering requirements for collective operations occur is difficult using traditional debugging techniques. For example, with most MPI implementations, if the size of the broadcast message is below the eager send limit, tasks 0, 1 and 2 complete the broadcast operation of Figure 7 and continue into the barrier operation. Thus, all of the tasks are in the barrier when the error becomes apparent. Umpire verifies the sequence of these MPI operations and detects mismatched collective operations automatically. It also uses its PC information to report the exact locations where they occur in the code.

5 Resource Tracking Errors

Another common set of problems in MPI applications result from resource tracking errors. MPI has several features that must allocate underlying resources to satisfy an application's requests. In particular, when applications create opaque objects (derived types, derived communicators, groups, key values, request objects, error handlers, and user-defined reduction operations), the MPI implementation may allocate memory to store internal bookkeeping information. Applications can exhaust memory if they create opaque objects repeatedly without releasing them. Further, some MPI implementations use fixed-size tables to manage these resources, which results in strict limits on the number of these objects that can exist concurrently. Presently, Umpire tracks and automatically reports "leaks" of three commonly used MPI opaque objects: derived data types, requests, and communicators.

5.1 Basic Resource Leaks

Leaks of MPI opaque objects can occur in several different ways. Figure 8 shows a straightforward data type leak, in which the only record of the opaque object handle is overwritten by another handle. As illustrated in Figure 9, another error occurs when applications finalize MPI without freeing opaque objects. Depending on the MPI implementation, this type of resource tracking error may not have any negative effect on the program. However, it can reduce portability since some implementations significantly restrict the number of derived communicators - most notably, implementations that target SMP clusters with on-node shared memory communication.

5.2 Lost Requests

Lost requests, an extremely important example of "leaked" resources in MPI, occur when an application overwrites a request handle for a non-blocking MPI operation. These violations eventually result in a missed completion operation for a non-blocking operation. Figure 10 shows an example sequence of MPI operations that lead to a lost request. Because the second MPI_Irecv overwrites the request handle to the first MPI_Irecv, no matching completion confirms the end of the first MPI_Irecv. Note that the MPI application can execute to completion without such a matching completion. However, calculations depending on data transmitted for such an operation are likely to be incorrect, and, worse, non-deterministic.

5.3 Identification of Resource Tracking Errors

Umpire tracks each type of opaque object separately and reports all leaked objects at MPI_Finalize. This report identifies the PC and task that allocated the object. The MPI standard allows assignment and comparison of opaque object handles. Figure 12 shows a leak-free example that is very similar to the code that creates a lost request. Although the code may be dangerous, it is correct and illustrates a difficulty in determining when opaque objects are leaked. Umpire compares both the address of the request and the request handle itself. Umpire does not misidentify the code in Figure 12 as having a leak since we track opaque handles even after they are apparently lost. Thus, we remove the request from the list of leaked objects when we process the second wait.

5.4 Errant Changes to Send Buffers

Another error that can result from non-blocking communication operations is errant changes to send buffers. As Figure 11 demonstrates, this error occurs when a send buffer changes between send initiation and send completion. This type of error is particularly difficult to locate because, depending on the MPI implementation, this error is non-deterministic. That is, the errant write to the send buffer can occur before or after the data in buf has been copied by the MPI runtime.

Umpire guards against this type of error by calculating a checksum on the send buffer at initiation and then recalculating that checksum at completion. If the checksums differ, then Umpire records this discrepancy and issues a warning to the user.

Task 0	Task 1

```
int dsize = 60000;
...
MPI_Send (&data, dsize, MPI_CHAR, 1, tag, comm);
MPI_Recv (&data, dsize, MPI_CHAR, 1, tag, comm, &status);
...
```

```
int dsize = 60000;
...
MPI_Send (&data, dsize, MPI_CHAR, 0, tag, comm);
MPI_Recv (&data, dsize, MPI_CHAR, 0, tag, comm, &status);
...
```

Figure 6: Configuration-dependent deadlock example.

Tasks 0, 1 & 2	Task 3
... MPI_Bcast (buf0, buf_size, MPI_INTEGER, 0, comm); MPI_Barrier (comm); ...	... MPI_Barrier (comm); MPI_Bcast (buf0, buf_size, MPI_INTEGER, 0, comm); ...

Figure 7: Mismatched collective operations example.

```
MPI_Type_contiguous (128, MPI_INTEGER, &newtype);
MPI_Type_vector (3, 1, 2, MPI_DOUBLE, &newtype);
```

Figure 8: Derived type leak.

```
MPI_Comm_split (MPI_COMM_WORLD, color, key, &newcomm);
...
MPI_Finalize ();
```

Figure 9: Communicator leak.

```
MPI_Request req;
MPI_Status status;
...
MPI_Irecv (buf0, buf_size, MPI_INTEGER, 0, tag1, comm, &req);
MPI_Irecv (buf1, buf_size, MPI_INTEGER, 0, tag2, comm, &req);
...
MPI_Wait (&req, &status);
...
MPI_Finalize();
```

Figure 10: Lost request example.

```
MPI_Request req1;
MPI_Status status;
...
MPI_Isend (buf, buf_size, MPI_INTEGER, 0, tag1, comm, &req1);
...
buf[0]=1234;
...
MPI_Wait (&req1, &status);
...
```

Figure 11: Example of errant write to send buffer.

```
MPI_Request req, req2;
MPI_Status status;
...
MPI_Irecv (buf0, buf_size, MPI_INTEGER, 0, tag1, comm, &req);
req2 = req;
MPI_Irecv (buf1, buf_size, MPI_INTEGER, 0, tag2, comm, &req);
...
MPI_Wait (&req, &status);
MPI_Wait (&req2, &status);
...
MPI_Finalize();
```

Figure 12: Request assignment example.

6 EVALUATION

We have tested a number of MPI applications with our operational prototype, including a suite of MPI simple test cases to calibrate Umpire for each of the programming errors. In order to demonstrate the real value of Umpire, we tested several widely-used applications, including publicly available benchmarks and codes. These codes included the DOE ASCI benchmark suite, benchmarks from the NAS

Parallel Benchmark suite, FFTW, QCDMPI, PSPASES, and ParaMetis. Not surprisingly, Umpire did not expose any major MPI errors in many of these mature codes, but these codes did stress Umpire's internal structure.

With this result, we turned our testing attention to other less mature codes. We uncovered several problems in the message passing components of these codes that, to our knowledge, were previously unnoticed. For instance, in one 3D 27-point stencil code, Umpire located a buffer deadlock in the initialization code. The majority of this application's message passing activity was completely correct; however, this initialization code was hidden in several layers of abstraction. As a result, all of the MPI tasks called MPI_Send before they called the matching MPI_Recv. Each message is 508 bytes in size, and because the MPI eager send limit is usually much larger at tens of thousands of bytes, users could rest assured that they would not suffer a deadlock. In this case, the application design elected to ignore this deadlock situation, but simple changes to the size of the initialization structures or a drastic increase in the number of MPI tasks, could cause a buffer deadlock.

6.1 Effectiveness

Umpire does not prove that the application is free from errors; it exposes MPI application programming errors through software testing. We believe that Umpire will prove most useful during the early phases of application development and porting. During our design process for Umpire, we ranked performance as a secondary concern. Since our tool is focussed on application correctness, it will slow down applications; users should make no inferences about the performance of their code while they are testing with Umpire. Our only concern with respect to performance is that testing with Umpire should not require excessive runtimes relative to the normal runtime of the application. From our testing of several MPI applications, it is clear we have met this goal. Even with 4 tasks on the IBM SP2, Umpire causes a slowdown of approximately 2.

7 Conclusions

We have presented Umpire, an innovative tool for dynamically detecting MPI programming errors. By using the MPI profiling layer to interpose Umpire between the MPI application and the MPI runtime system, we retain a level of portability while capturing sufficient information about the application's MPI activity to reason about erroneous MPI usage. Umpire's shared memory design allows the user to test their application in a reasonable amount of time. Umpire's initial collection of programming errors includes deadlock detection, mismatched collective operations, and resource exhaustion. We have used Umpire to find errors in a several MPI applications.

7.1 Future Work

As mentioned earlier, we are considering the design of a distributed memory version of Umpire, so that users will not be restricted to executing their MPI application on one SMP. Yet the negative performance and scalability implications of sending messages to and processing messages with a centralized manager will most likely force a migration of the current design to one that uses a distributed algorithm for deadlock detection.

We continue documenting MPI programming errors and designing verification algorithms for Umpire. These errors include mismatches among tasks in derived type maps.

Another interesting proposal is to use Umpire to verify implementations of MPI.

References

[1] Z. Aral and I. Gertner, "High-Level Debugging in Parasight," *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24:151-62, 1989.

[2] P.C. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," *ACM Trans. Computer Systems*, 13(1):1-31, 1995.

[3] M.E. Crovella and T.J. LeBlanc, "Performance debugging using parallel performance predicates," *SIGPLAN Notices (ACM/ONR Workshop on Parallel and Distributed Debugging)*, 28, no.12:140-50, 1993.

[4] J. Cuny, G. Forman et al., "The Ariadne debugger: scalable application of event-based abstraction," *SIGPLAN Notices (ACM/ONR Workshop on Parallel and Distributed Debugging)*, 28, no.12:85-95, 1993.

[5] A. Dinning and E. Schonberg, "Detecting Access Anomalies in Programs with Critical Sections," Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, 1991, pp. 85-96.

[6] A. Eustace and A. Srivastava, "ATOM: a flexible interface for building high performance program analysis tools," Proc. 1995 USENIX Technical Conf., 1995, pp. 303-14.

[7] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, 2nd ed. Cambridge, MA: MIT Press, 1999.

[8] D.P. Helmbold, C.E. McDowell, and J.-Z. Wang, "Determining Possible Event Orders by Analyzing Sequential Traces," *IEEE Trans. Parallel and Distributed Systems*, 4(7):827-40, 1993.

[9] R. Hood, K. Kennedy, and J. Mellor-Crummey, "Parallel program debugging with on-the-fly anomaly detection," Proc. Supercomputing'90, 1990, pp. 74-81.

[10] HPCC, "HPCC 1998 Blue Book. (Computing, Information, and Communications: Technologies for the 21st Century)," Computing, Information, and Communications (CIC) R&D Subcommittee of the National Science and Technology Council's

Committee on Computing, Information, and Communications (CCIC) 1998.

[11] Kuck.and.Associates.Inc., *KAI Assure*, http://www.kai.com/assure-all, 2000.

[12] D.C. Marinescu, H.J. Siegel et al., "Models for Monitoring and Debugging Tools for Parallel and Distributed Software," *Jour. Parallel and Distributed Computing*, 9:171-84, 1990.

[13] J. Mellor-Crummey, "Compile-time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs," *SIGPLAN Notices (ACM/ONR Workshop on Parallel and Distributed Debugging)*:129-39, 1993.

[14] B.P. Miller and J.-D. Choi, "Breakpoints and Halting in Distributed Programs," Proc. Eighth Int'l Conf. Distributed Computing Systems, 1988, pp. 316-23.

[15] R.H.B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, 1993, pp. 1-11.

[16] W. Pfeiffer, S. Hotovy et al., "JNNIE: The Joint NSF-NASA Initiative on Evaluation," NSF-NASA 1995.

[17] Rational-Corporation, *Rational Purify for UNIX*, http://www.rational.com/products/purify_unix, 2000.

[18] S. Savage, M. Burrows et al., "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Computer Systems*, 15(4):391-411, 1997.

[19] M. Snir, S. Otto et al., Eds., *MPI--the complete reference*, 2nd ed. Cambridge, MA: MIT Press, 1998.

[20] M. Spezialetti and R. Gupta, "Exploiting program semantics for efficient instrumentation of distributed event recognitions," Proc. 13th Symp. Reliable Distributed Systems, 1994, pp. 181-90.

[21] SunSoft, "Locklint User's Guide," SunSoft, Manual 1994.