**Testing and Debugging Exascale Applications by Mocking MPI**

|  |  |  |
| --- | --- | --- |
| Thomas Clune  NASA Goddard Space  Flight Center  Greenbelt, MD 20771  thomas.l.clune@nasa.gov | Hal Finkel  Leadership Computing  Facility  Argonne National  Laboratory  9700 South Cass Avenue Building 240  Argonne, IL 60439  hfinkel@anl.gov | Michael Rilee Rilee Systems  Technologies LLC and  NASA GSFC  PO Box 5532  Derwood, MD 20855 michael.l.rilee- 1@nasa.gov |

**ABSTRACT**

Debugging and code verification present daunting challenges for message-passing parallel applications that employ bil-lions of processes/threads. Yet often it is only at scale that software defects first show themselves. When it is di�cult or impossible to replicate problems on small scale platforms, development delays and resource costs are significant consid-erations.

Software mocks, in which reconfigurable components re-place dependencies in an application component under test, are a powerful and versatile way to side-step expensive, com-plex, and/or otherwise impractical dependencies. We pro-pose that mocking application dependencies, and MPI in particular, is an e↵ective technique for testing and debug-ging exascale message-passing software using small-scale com-puting resources.

and formal methods for development continue to make ad-mirable progress[8], testing and debugging generally remain quite expensive both in terms of the consumption of dedi-cated computing resources and in terms of wasted developer time due to delays in availability of such large resources.[4] In an ideal world, all software defects could be exhibited using modestly sized computational domains and small numbers However, real-world experience has of processes/threads.

shown that bugs are all-too-often first detected when ex-tending an application to larger domains and/or computing platforms. Further, even once a defect has been detected and isolated, the creation of a small-scale reproducer can require precise understanding of the nature of the problem to preserve the salient characteristics. Thus, problems must often be largely resolved at-scale before a proper small-scale reproducer can be crafted.

|  |  |  |
| --- | --- | --- |
| **Categories and Subject Descriptors** | **1.1** | **Scenarios** |

D.2.4 [**Software/Program Verification**]: Model check-ing—*Assertion checkers*; D.2.5 [**Testing and Debugging**]: Distributed debugging —*Error handling and recovery*

**General Terms**

Reliability; Verification

We list here some representative scenarios in which tra-ditional approaches to testing and/or debugging of exascale applications would appear to require the use of large-scale computing resources:

*Serial algorithmic performance.*

When the measured performance (speed or memory foot-

|  |  |
| --- | --- |
| **Keywords** | print) of an algorithm diverges from prediction, profiling tools may be insu�cient to completely identify and correct |

software verification; mock objects; MPI; exascale

**1.**  **INTRODUCTION**

Verification and debugging are some of the most di�cult

challenges faced when developing software to be run on exas-

cale systems. And while static analysis, relative debugging,

Permissiontomakedigitalorhardcopiesofallorpartofthisworkfor personalorclassroomuseisgrantedwithoutfeeprovidedthatcopiesare notmadeordistributedforprofitorcommercialadvantageandthatcopies bearthisnoticeandthefullcitationonthefirstpage.Tocopyotherwise,to republish,topostonserversortoredistributetolists,requirespriorspecific permission and/or a fee. Request permissions from Permissions@acm.org. SEHPCCSE ’15, November 20, 2015, Austin, Texas, United States.

Copyright 2015 ACM ISBN 978-1-4503-4012-0/15/11...$15.00 DOI: http://dx.doi.org/10.1145/2830168.2830173

5

structures that are created and populated by some initial-ization layer of an application. Even if the routine to be tested is itself completely serial and thus has no *direct* de-pendency on MPI, there can still be an induced dependency in the corresponding tests due to MPI procedure calls in the initialization logic for the application. Configuring the ini-tialization to produce realistic large-scale data structures to drive the control logic on a small number of processes is not possible in many cases and the developer must then choose between enhancing the initialization to support testing and creating a custom variant. As a trivial example consider the case where the size of the local data structure is a nonlinear function of the grid size.

*Error handling/trapping.*

Consider an application that attempts to trap and fail gracefully when MPI procedures return with an error code.

**2.**  **APPROACH**   
 In software engineering parlance, a “mock” is an interface, or collection of interfaces, that can be used to replace a dependency within a software system[6]. Mocks are not in-tended to be fully functional, but rather to facilitate testing in the presence of complex and/or expensive dependencies. Mocks can be used to verify that correct data is sent to an external dependency and can be configured to produce predictable return values from an external dependency. A canonical example of the use of mocks is for tests of software that modifies large/important databases. In addition to the large overhead for connecting to a real shared database, tests would undesirably modify values in the database and could not rely on existing values in the database. Mocks pro-vide an appropriate sandbox to ensure that the procedures which interact with the database are correctly implemented without the cost and risk of working directly with the main

Such a policy is especially valuable for MPI I/O procedures database.

due to inherent uncertainties in the state of the file system. MPI errors are more frequent at high process counts due in part to the total number of procedure calls, but also due to larger bu↵ers, greater complexity, and hot-spots in the communication fabric. Verification that an application cor-rectly detects and handle these failures is di�cult as small use cases may rarely or possibly never generate the necessary conditions.

*Deadlock and Race Conditions?.*

As with explicit failure signals from MPI, race conditions become increasingly likely as applications scale to larger numbers of processors. Ensuring that race conditions are correctly guarded against, or debugging a procedure in which a race condition is suspected, therefore generally involves running a large scale scenario.

**2.1**  **Mocking MPI**   
 Mocking MPI, at a high level, allows a test that only uses a few processes (perhaps just one) to act as though they are part of a much larger group. This capability is somewhat di�cult to create as MPI makes its callers explicitly aware of the size of each communicator (group of processes), and requires that each group must be complete in order to func-tion (e.g., collectives). Nonetheless, MPI is *the* standardized interface for distributed-memory parallelism in high perfor-mance computing (HPC), and direct uses of MPI are per-vasive throughout many important exascale applications. It is therefore impractical to insist that *all* HPC applications hide MPI within abstractions such as Charm++[7], Peb-bles[10], and AM++[9]. Mocking can of course be equally useful in those contexts, and presumably considerably easier

|  |  |  |
| --- | --- | --- |
| **1.2** | **Goals** | to implement. |
| Developers who are unfamiliar with the technique of mock- |

In this paper we propose a methodology that enables the development of unit tests that can be used to verify correct

ing often initially confuse it with “stubbing”. Stubbing is a technique to provide a *trivial* implementation of some in-

|  |  |  |  |
| --- | --- | --- | --- |
| software behavior for scenarios such as those discussed above | terface for the purpose of aiding portability. | E.g. | many |

while utilizing only a single process executing on a single node (i.e., in the extreme case, one could attempt to verify many aspects of an exascale application on a simple laptop with su�cient memory.) In particular we expect to be able to verify software characteristics such as:

*•* serial performance and memory consumption*•* correct loop bounds  
*•* correct topology of neighbor processes  
*•* existence/size of messages from other processes*•* error handling / fault tolerance  
*•* race conditions and deadlock

Note that we are not suggesting that this methodology

packages contain a stub of MPI which provide one-process behavior for the restricted subset of MPI used by the pack-age. This can reduce the installation di�culty for new users and/or permit deployment of an application in environments that lack MPI or restrict its use. Whereas stubbing provides a trivial but technically correct implementation for some subset of MPI, mocking attempts to *emulate* the behavior of many MPI processes while providing essentially none of the actual functionality. Each test configures the mock layer such that calls to MPI processes return predetermined syn-thetic values that are intended to probe other aspects of the procedure being tested. The technique is powerful but sub-tle and therefore requires some experience and thought to exploit.

To mock MPI the infrastructure must provide mechanisms for configuring the *apparent* behavior of nonexistent pro-cesses. Often this behavior is trivial: broadcasts and sends to nonexistent ranks can be ignored, barriers can be assumed

would eliminate *all* requirements for testing and debugging to have been reached, and so on. Emulating the contri-

at scale, but rather that it has the potential to significantly bution of nonexistent ranks to reduction operations is not

reduce such needs. Also note that mocking is inherently particularly di�cult. The most complicated aspect is the

unable to diagnose problems in the the implementation of the MPI layer itself except insofar as to aid in eliminating other possibilities.

6

of procedure calls, each with a separately configurable set of Mocking is generally easier to implement and use in an

outputs. object-oriented context where dependency injection (DI) be-

As with many other mocking scenarios, a potentially sig-nificant complication for the implementer of the mock ser-vice is the di�culty in manipulating private data structures defined within the code under test (e.g., if the code involves a complicated data structure for which an MPI data type has been constructed, it can be di�cult for the mock service to take appropriate actions without implementing a substan-tial subset of MPI’s type management system.) Allowing the test implementer to use the existing definitions of these entities is essential to creating a practical solution.

The good news is that in many MPI applications, any given process primarily communicates with only a small number of neighbors with the major exception being rela-tively simple global broadcasts and reductions. So the num-ber of non-existent ranks that will have relevant *non-trivial* behavior to emulate is quite limited. Yet two important is-sues requiring more e↵ort are: user-defined data types and complex sequences of data exchanges.

comes a powerful technique. DI exposes dependencies within a system under test enabling them to be readily replaced with configurable mock behaviors. Indeed, several mocking frameworks (e.g. Google Mock[1], Hippo Mocks[2], mock-cpp[3]) exist for C++ and could be more-or-less directly applied to mock MPI. These frameworks generally combine preprocessor directives and templating to allow developers to instantiate mock objects with a relatively modest amount of source code. They provide flexible means for configuring the outputs and specifying the expected input parameters to a sequence of calls to the layer being mocked.

The situation for Fortran is a bit more bleak at the mo-ment. We (Clune and Rilee) have made some initial steps to introduce mocking capabilities within pFUnit[5], a unit testing framework for Fortran with MPI. While Fortran now supports object-oriented programming, it still lacks suit-able templating capabilities comparable to those of C++, and therefore requires considerably more manual e↵ort on

|  |  |  |
| --- | --- | --- |
| As a pedagogical example of our proposed methodology, | the part of developers to instantiate mock objects. | Re- |

consider a test in which we wish to “fool” the application into thinking that it is running in an environment with *Np* processes on rank *r*. A prototype mock MPI implementation might look something like the following for MPI\_Comm\_rank()

gardless of this, the standard interfaces for MPI, except for the now-deprecated C++ interfaces, are procedural. This means that rather than using mock-objects for dependency injection, we must resort to reconfiguring an application at

|  |  |  |
| --- | --- | --- |
| with an analogous implementation for MPI\_Comm\_size(): | the link-step with a mock-library. | Further, to accommo- |

subroutine MPI\_Comm\_rank(comm, rank, ierr) use MockMPI, only: mock   
integer, intent(in) :: comm   
integer, intent(out) :: rank, ierr

call mock%verify(’MPI\_Comm\_rank’, ’comm’, comm) call mock%get(’MPI\_Comm\_rank’, ’rank’, rank) call mock%get(’MPI\_Comm\_rank’, ’ierr’, ierr) end subroutine MPI\_Comm\_rank

date some use-cases in which user-defined state must be saved/compared/restored, users must be prepared to con-struct helper procedures that are passed into the mock li-brary as part of the configuration step.

**3.**  **EXAMPLES**   
 We consider here some highly-simplified examples that are representative of realistic di�culties encountered when test-ing complex parallel software.

|  |  |  |
| --- | --- | --- |
| A test for a procedure proc() might then configure the mock | **3.1** | **Error trapping** |

to behave as rank 4 of a 10 process execution in a manner like:

use MockMPI, only: mock   
call mock%set(’MPI\_Comm\_size’,’npes’, 10) call mock%set(’MPI\_Comm\_rank’,’rank’, 4)

Suppose we wish to test whether or not a procedure cor-rectly traps a certain unsuccessful MPI return code within a given procedure. Arranging for MPI to routinely fail in that particular manner might be di�cult or even impossible, but we can configure the mock to do so as part of such a test:

... use MockMPI, only: mock, MPI\_ERR\_TAG

call proc(...) call mock%set(’MPI\_iSend’,’ierr’, MPI\_ERR\_TAG)

< check results >

The set() methods store values for procedure *output* pa-rameters for later retrieval by the get() methods. The verify() method can detect whether an *input* parameter matches previously set expectations, if any.

Note that this style of testing can also be very useful in non-exascale contexts. Also note that we are not advocating this particular implementation approach for mock MPI; it is only meant to be suggestive, easily understood, and fit within the 2-column format.

A complete implementation of a mock layer for MPI would be a require a significant e↵ort, though with far less com-plexity than the implementation of MPI itself. However, just as with MPI, many applications would only require a relatively modest subset of interfaces to be supported for practical use.

call proc\_that\_should\_trap(...) @assertExceptionRaised(...)

**3.2**  **Complex data types**   
 While standard approaches for manipulating procedure parameters su�ce in cases where the types are fixed (e.g.,’rank’, ’recvcounts’, etc.), a generic implementation of a mock MPI will be unable to *directly* manipulate bu↵er pa-rameters. Instead, users will likely be required to create and pass to the mock a set of small auxiliary procedures that save/compare/restore values for the types actually be-ing used. (Possibly the cretaion of these could be semi-automated through some sort of templating preprocessor.) Suppose we wish to test a procedure which receives ele-ments 1 and 3 an array of type UserDefined in a call to MPI\_Recv. If the user type is defined by:

|  |  |  |
| --- | --- | --- |
| **2.2** | **Available technologies** | type UserDefined |
| real :: tau |

7

integer :: n   
end type UserDefined

then synthetic values can be specified in a test:

subroutine test\_complicated()   
type (UserDefined) :: x(4)   
x(1) = UserDefined(2.5, 5) !synthetic   
x(3) = UserDefined(3.5, 8) !synthetic   
call mock%set(’MPI\_Recv’, ’rbuf’, custom\_set, x) call complicated\_procedure(...)   
...

end subroutine test\_complicated

The aux. procedure custom\_set() is defined as:

subroutine custom\_set(addr, rbuf)   
! sets elements 1 and 3 of rbuf   
 type (c\_ptr) :: addr   
 type (UserDefined) :: rbuf(4)   
 type (UserDefined), pointer :: saved\_buf call c\_f\_pointer(addr, saved\_buf, [4]) rbuf(1) = save\_buf(1)   
 rbuf(3) = save\_buf(3)   
end subroutine custom\_set

**3.3**  **Race condition**   
 At first glance, race conditions would appear to be dif-ficult or even impossible to address with the methodology advocated in this paper. This is because the observed in-correct behavior can be on a process that executing correct code, and processes with incorrect code may behave cor-rectly with regard to the values of its data. However, with a modest amount of ingenuity, it is possible to configure the MPI mock to detect such code defects.

In this example we demonstrate a test that is meant to ensure that a call to MPI\_Wait() is executed after a call to MPI\_Isend() and priort to any local modification of that

**4.**  **CONCLUSION**   
 Verification of software running on billions of cores is ex-pected to be a serious barrier to scientific productivity on anticipated exascale platforms. Traditional approaches to testing and debugging on bleeding edge machines are ex-pensive and present significant bottlenecks for productivity. Applying the methodology of software mocking in this en-vironment may significantly improve the rate at which MPI applications can be developed, tested, and verified. By sim-ulating the parallel context experienced by a single process within the application, developers can then routinely test and investigate code behavior on relatively modest comput-ing resourced during routine software development.

**5.**  **REFERENCES**   
[1] Google c++ mocking framework.

"http://code.google.com/p/googlemock". Accessed: 2015-0815.

[2] Home — hippo mocks project — assembla.

<https://www.assembla.com/wiki/show/hippomocks>. Accessed: 2015-0815.

[3] mockcpp — a c/c++ mock framework.

"http://code.google.com/p/mockcpp". Accessed: 2015-08-15, also:   
https://bitbucket.org/godsme/mockcpp.

[4] Tools for exascale computing: Challenges and   
strategies. U.S. Department of Energy, O�ce of   
Science "http://science.energy.gov/~/media/ascr/ pdf/research/cs/Exascale%20Workshop/Exascale\_ Tools\_Workshop\_Report.pdf", 2011.

[5] T. Clune and M. Rilee. pFUnit 3.0 - a unit testing framework for parallel fortran software.

<http://sourceforge.net/projects/pfunit>, 2014. Accessed: 2014-07-08.

bu↵er: [6] M. Feathers. *Working E↵ectively with Legacy*

type (my\_type) :: sbuf   
call mock%set(’MPI\_Isend’, ’sbuf’, sbuf, capture) call mock%expect\_call(’MPI\_Wait’, compare)   
call code\_with\_race(sbuf)   
call mock%verify\_all()

The auxiliary procedure capture() saves the address of the passed bu↵er as well as a copy of the data during the call to MPI\_Isend(), whereas compare() verifies that the bu↵er still contains the same values during MPI\_Wait().

module custom\_mod   
 type (C\_PTR), save :: save\_addr   
 type (my\_type), save :: sbuf\_save   
contains   
 subroutine capture(sbuf\_addr)   
 type (C\_PTR), intent(in) :: sbuf\_addr type (my\_type), pointer :: sbuf   
 save\_addr = sbuf\_addr   
 call c\_f\_pointer(sbuf\_addr, sbuf) sbuf\_save = sbuf   
 end subroutine capture   
 subroutine compare()   
 type (my\_type), pointer :: sbuf   
 call c\_f\_pointer(sbuf\_addr, sbuf) @assertEqual(sbuf\_save, sbuf)   
 end subroutine compare   
end module custom\_mod

8