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 billions of processes/threads. Yet often it is only at scale that software defects first show themselves. When it is difficult or impossible to replicate problems on small scale platforms, development delays and resource costs are significant considerations.

Software mocks, in which reconfigurable components replace dependencies in an application component under test, are a powerful and versatile way to side-step expensive, complex, and/or otherwise impractical dependencies. We propose that mocking application dependencies, and MPI in particular, is an effective technique for testing and debugging exascale message-passing software using small-scale computing resources.

Categories and Subject Descriptors

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

General Terms

Reliability; Verification

Keywords

software verification; mock objects; MPI; exascale

1. INTRODUCTION

Verification and debugging are some of the most difficult challenges faced when developing software to be run on exascale systems. And while static analysis, relative debugging, and formal methods for development continue to make admirable progress[8], testing and debugging generally remain quite expensive both in terms of the consumption of dedicated 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 of processes/threads. However, real-world experience has shown that bugs are all-too-often first detected when extending 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.

SEHPCCSE '15, November 20, 2015, Austin, Texas, United States.

DOI: http://dx.doi.org/10.1145/2830168.2830173

1.1 Scenarios

We list here some representative scenarios in which traditional 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 footprint) of an algorithm diverges from prediction, profiling tools may be insufficient to completely identify and correct the problem (or could still involve considerable expense due to the need to investigate the performance under a variety of configurations.) Here we are considering only the performance of the *serial* portion of an algorithm, but in the context of a distributed implementation. Except in relatively trivial cases, rescaling the problem to enable study with more modest resources will unacceptably alter the balance of data structure sizes, cache usage, and computational load of the different phases of the algorithm.

Testing/debugging complex logic.
Although complex logic in a procedure frequently can be verified using modestly sized grids on a few or perhaps just one process, there remain important situations in which such reductions are exceedingly difficult. Consider the common situation in which an algorithm relies on complex data-

structures that are created and populated by some initialization layer of an application. Even if the routine to be tested is itself completely serial and thus has no *direct* dependency 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 initialization 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. Such a policy is especially valuable for MPI I/O procedures 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 buffers, greater complexity, and hot-spots in the communication fabric. Verification that an application correctly detects and handle these failures is difficult 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.

1.2 Goals

In this paper we propose a methodology that enables the development of unit tests that can be used to verify correct software behavior for scenarios such as those discussed above 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 sufficient 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 would eliminate *all* requirements for testing and debugging at scale, but rather that it has the potential to significantly reduce such needs. Also note that mocking is inherently unable to diagnose problems in the the implementation of the MPI layer itself except insofar as to aid in eliminating other possibilities.

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 intended 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 provide 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 database.

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 difficult 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 function (e.g., collectives). Nonetheless, MPI is *the* standardized interface for distributed-memory parallelism in high performance computing (HPC), and direct uses of MPI are pervasive throughout many important exascale applications. It is therefore impractical to insist that *all* HPC applications hide MPI within abstractions such as Charm++[7], Pebbles[10], and AM++[9]. Mocking can of course be equally useful in those contexts, and presumably considerably easier to implement.

Developers who are unfamiliar with the technique of mocking often initially confuse it with "stubbing". Stubbing is a technique to provide a *trivial* implementation of some interface for the purpose of aiding portability. E.g. many packages contain a stub of MPI which provide one-process behavior for the restricted subset of MPI used by the package. This can reduce the installation difficulty 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 synthetic values that are intended to probe other aspects of the procedure being tested. The technique is powerful but subtle and therefore requires some experience and thought to exploit.

To mock MPI the infrastructure must provide mechanisms for configuring the *apparent* behavior of nonexistent processes. Often this behavior is trivial: broadcasts and sends to nonexistent ranks can be ignored, barriers can be assumed to have been reached, and so on. Emulating the contribution of nonexistent ranks to reduction operations is not particularly difficult. The most complicated aspect is the emulation of messages sent from nonexistent ranks to the the process that is actually executing in the test configuration. The mocking layer must support arbitrary sequences

of procedure calls, each with a separately configurable set of outputs.

As with many other mocking scenarios, a potentially significant complication for the implementer of the mock service is the difficulty 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 difficult for the mock service to take appropriate actions without implementing a substantial 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 relatively simple global broadcasts and reductions. So the number of non-existent ranks that will have relevant *non-trivial* behavior to emulate is quite limited. Yet two important issues requiring more effort are: user-defined data types and complex sequences of data exchanges.

As a pedagogical example of our proposed methodology, consider a test in which we wish to "fool" the application into thinking that it is running in an environment with N_p processes on rank r. A prototype mock MPI implementation might look something like the following for `MPI_Comm_rank()` with an analogous implementation for `MPI_Comm_size()`:

```
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
```

A test for a procedure proc() might then configure the mock 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)
...
call proc(...)
< check results >
```

The `set()` methods store values for procedure *output* parameters 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 effort, though with far less complexity 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.

2.2 Available technologies

Mocking is generally easier to implement and use in an object-oriented context where dependency injection (DI) becomes 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], mockcpp[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 moment. 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 suitable templating capabilities comparable to those of C++, and therefore requires considerably more manual effort on the part of developers to instantiate mock objects. Regardless 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 the link-step with a mock-library. Further, to accommodate some use-cases in which user-defined state must be saved/compared/restored, users must be prepared to construct helper procedures that are passed into the mock library as part of the configuration step.

3. EXAMPLES

We consider here some highly-simplified examples that are representative of realistic difficulties encountered when testing complex parallel software.

3.1 Error trapping

Suppose we wish to test whether or not a procedure correctly traps a certain unsuccessful MPI return code within a given procedure. Arranging for MPI to routinely fail in that particular manner might be difficult 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 mock%set('MPI_iSend','ierr', MPI_ERR_TAG)
call proc_that_should_trap(...)
@assertExceptionRaised(...)
```

3.2 Complex data types

While standard approaches for manipulating procedure parameters suffice 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 buffer parameters. 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 being 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 elements 1 and 3 an array of type `UserDefined` in a call to `MPI_Recv`. If the user type is defined by:

```
type UserDefined
   real :: tau
```

```
   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 difficult or even impossible to address with the methodology advocated in this paper. This is because the observed incorrect behavior can be on a process that executing correct code, and processes with incorrect code may behave correctly 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 buffer:

```
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 buffer as well as a copy of the data during the call to `MPI_Isend()`, whereas `compare()` verifies that the buffer 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
```

4. CONCLUSION

Verification of software running on billions of cores is expected to be a serious barrier to scientific productivity on anticipated exascale platforms. Traditional approaches to testing and debugging on bleeding edge machines are expensive and present significant bottlenecks for productivity. Applying the methodology of software mocking in this environment may significantly improve the rate at which MPI applications can be developed, tested, and verified. By simulating the parallel context experienced by a single process within the application, developers can then routinely test and investigate code behavior on relatively modest computing 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, Office 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.

[6] M. Feathers. *Working Effectively with Legacy Software*. Prentice Hall. Pearson Education, Inc., Upper Saddle River, NJ, 2005.

[7] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *Parallel Programming Laboratory Technical Report #95-02*, 1994.

[8] I. Laguna, D. H. Ahn, B. R. de Supinski, T. Gamblin, G. L. Lee, M. Schulz, S. Bagchi, M. Kulkarni, B. Zhou, Z. Chen, and F. Qin. Debugging high-performance computing applications at massive scales. *Commun. ACM*, 58(9):72–81, Aug. 2015.

[9] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Am++: A generalized active message framework. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 401–410, New York, NY, USA, 2010. ACM.

[10] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active pebbles: Parallel programming for data-driven applications. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 235–244, New York, NY, USA, 2011. ACM.