Implementing Efficient Dynamic Formal Verification Methods for MPI Programs*

Sarvani Vakkalanka[1], Michael DeLisi[1], Ganesh Gopalakrishnan[1], Robert M. Kirby[1], Rajeev Thakur[2], and William Gropp[3]

[1] School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA
[2] Math. and Comp. Sci. Div., Argonne Nat. Lab., Argonne, IL 60439, USA
[3] Dept. of Computer Sci., Univ. of Illinois, Urbana, Illinois, 61801, USA

Abstract. We examine the problem of verifying MPI programs for the absence of deadlocks and local assertion violations through dynamic (runtime) formal verification in which the processes of a given MPI program are executed under the control of an interleaving scheduler. The development of such an algorithm requires several challenges to be overcome in ensuring *full* coverage (as opposed to a testing tool that can miss bugs): (i) The algorithm must take into consideration MPI's out-of-order completion semantics (that is, MPI operations need not finish in the issue order), and (ii) the algorithm must ensure that MPI wildcard receive matches are considered in all possible ways. Our new algorithm is based on the following novel ideas. First, it rewrites wildcard receives to specific receives, one for *each* sender that can potentially match with the receive. It then recursively explores each case of the specific receives. The list of potential senders matching a receive is determined through a runtime algorithm that exploits MPI's operational ordering semantics. Our verification tool ISP that incorporates this algorithm efficiently verifies several programs and finds bugs in many cases where existing dynamic informal verification tools may miss them.

1 Introduction

With the increasing use of MPI for the distributed programming of virtually all high-performance computing clusters in the world, it is important that MPI programs be verified to be free of bugs. With the need to re-verify MPI programs after each optimization step, the process of verification must involve only modest computing resources and limit manual tedium. As MPI programs can contain many types of bugs, including deadlocks, assertion violations (*e.g.*, resource leaks caught by assertions that keep track of allocations and deallocations), and numerical inaccuracies, it is practically impossible for a single tool to guarantee the coverage of bugs in all these classes. Therefore, approaches that focus on a limited bug class and guarantee full coverage for that class are preferred.

In this paper, we present our C MPI program verification tool, named ISP, that incorporates a novel scheduling algorithm called POE (Partial Order reduction avoiding Elusive interleavings). ISP guarantees to detect *all* deadlocks and

* Supported in part by NSF CNS-00509379, and Microsoft HPC Institutes Program.

local assertion violations in MPI programs containing 24 of the most commonly used MPI functions. For these MPI programs, the ISP tool will explore close to the minimal number of interleavings. Furthermore, ISP does not require any modeling effort on part of users, allowing it to be easily re-run during program development. ISP enjoys the same ease of use as the dynamic verification tools Umpire [2], Marmot [3], ConTest [4], and Jitterbug [5] (to name a few). However, the POE algorithm offers the formal guarantee of finding *all* deadlocks. As shown by experiments on our web site [12], of the 69 Umpire tests, 30 contain deadlocks, and ISP detects all of them, while exploring a very small number of interleavings. In contrast, Marmot fails to find deadlocks in eight of these tests, despite being run multiple times. When these tests were run with MPICH2 repeatedly, the deadlock detection success was unpredictable. Tools that rely on perturbing schedules simply cannot guarantee coverage.

The crucial idea embodied in POE is the notion of exploring only *relevant* interleavings—a technique known in model checking as *partial order reduction* [6]. Without this idea, any exploration method for MPI will go out of hand. For instance, consider the short MPI program below that begins with two sends in P0 and P2, and a wildcard receive in P1. The total number of interleavings of all these MPI calls is 210.[4] However, to trigger `error1`, we need only consider the two interleavings corresponding to the sends matching the wildcard receive; more importantly, we must try *both* these interleavings, or else we can mask the bug. If the above-mentioned testing-oriented tools explore the space of all these 210 interleavings, they will find the error, but only in *one* of the *210 interleavings*. Thanks to partial order reduction, ISP will: (i) pick an arbitrary order for executing `P0`'s first send and `P1`'s first receive, (ii) pick an arbitrary order to execute `P2`'s first send and `P1`'s second receive, and then (iii) consider *both* the `Send` matches with the wildcard receive (shown by `*`).

```
P0: MPI_Send(to P1...); MPI_Send(to P1, data = 22);
P1: MPI_Recv(from P0...); MPI_Recv(from P2...);
    MPI_Recv(*, x); IF (x==22) THEN error1 ELSE MPI_Recv(*, x);
P2: MPI_Send(to P1...); MPI_Send(to P1, data = 33);
```

ISP has built-in knowledge of the commuting properties of MPI functions. For example, consider an MPI program in which `MPI_Barrier` is invoked by N processes. ISP would, in general, explore only one of the $N!$ ways in which to have invoked the barrier calls. In our implementation of the 24 MPI functions, the cases where alternate interleavings are to be explored include wildcard receives, `WAIT_ANY`, and `TEST_ANY`.

Overview of ISP's use of PMPI: We use the well-known PMPI mechanism, normally used for performance studies, to support runtime model checking in ISP. We introduce an extra process called the *verification scheduler*. ISP provides its own version of "MPI_f" for each MPI function f. Within MPI_f, we arrange for handshakes with the scheduler that realizes the POE algorithm. When the scheduler finally gives permission to fire f, we invoke PMPI_f from within our version of MPI_f. The MPI runtime only sees the PMPI_f calls.

[4] (7!)/((2!).(3!).(2!))

Related Work: A dynamic formal verification tool for reactive C programs, and including a partial order reduction algorithm, was first proposed by Godefroid [7]. In [8], Flanagan and Godefroid further extended this work to have a more efficient *dynamic partial order reduction* (DPOR) algorithm. In [1] we presented the first DPOR-based verification method for MPI programs that use one-sided communication. In [1], we suggested that DPOR can also be used to handle MPI programs that use the traditional MPI two-sided operations – blocking and non-blocking communication commands and collectives. In [9] (a two-page "tools paper"), we reported such an implementation, but later realized that our tool overlooks some aspects of MPI's out-of-order operation completion. It also proved incapable of controlling the MPI runtime to force the desired wildcard receive matches (see Section 2). POE overcomes both these limitations, and replaces DPOR – the former algorithm implemented within ISP. Exploiting MPI's semantics, POE employs a strategy of *lookahead computation* to discover how sends and receives in an MPI program can match. It then employs a strategy similar to *classical static partial order reduction*, as employed in tools such as SPIN [10] and Verisoft (see Section 2).

Roadmap: The remainder of this introduction presents in detail the three new ideas used in POE: *Forcing Wildcard Matches* (Section 1.1), *Handling Out-of-order Completion* (Section 1.2), and *Discovering Match Sets* (Section 1.3). Section 2 presents the POE algorithm in detail, focusing on sends, receives, and barriers. Section 2.2 describes how many additional MPI commands are smoothly handled by the extended POE algorithm implemented in ISP. We also discuss how the user interface of a Visual Studio integration of POE works: we strive to preserve the users' view of their MPI program, despite the fact that our POE algorithm changes the internal computation through dynamic rewriting. Section 3 presents experimental results and Section 4 concludes.

1.1 Forcing Wildcard Matches

Consider the example in Figure 1, with line 2 containing a wildcard receive. A match between the `Isend` on line 6 and `Irecv` on line 2 (wildcard) will enable `Recv` on line 3 to match with the `Isend` on line 9. However, if the `Isend` on line 9 were to match the `Irecv` on line 2, a *deadlock* would result, with `Recv` (line 3) no longer able to match `Isend` (line 6). Clearly, we cannot leave out this second option (process interleaving) during testing.

```
0 : // * means MPI_ANY_SOURCE
1 : if (rank == 0)
2 : { MPI_Irecv(&buff1, *, &req);
3 :   MPI_Recv (&buff2, from 2);
4 :   MPI_Wait (&req) }
5 : else if (rank == 1)
6 : { MPI_Isend(buff1, to 0, &req);
7 :   MPI_Wait (&req); }
8 : else if (rank == 2)
9 : { MPI_Isend(buff2, to 0, &req);
10:   MPI_Wait (&req); }
```

Fig. 1. *Relevant Interleavings and Elusive Matches during Dynamic Verification of MPI Programs*

The role of a dynamic verification tool for MPI is to determine, at runtime, the specific matches possible, and explore *all relevant* ones - that is, a representative of each equivalence class of equivalent interleavings. This method must be carried out at runtime: (i) the outcomes of control branches through `switch` statements

will be known only at runtime and (ii) the send/receive targets/sources, and other details (communicator, tag, etc.) may be values that are computed at runtime.[5]

We now explain briefly why DPOR does not work for MPI. Suppose a DPOR-based algorithm is able to determine that `Isend` (line 6) matched `Irecv` (line 2), and that `Isend` (line 9) is also a potential alternate match for this `Irecv`. According to the algorithm of [8], the dynamic verification scheduler must now somehow force this alternative match – say by firing the `Isend` (line 9) in real-time order before firing `Isend` (line 6). However, we know from MPI's semantics that the MPI runtime environments *do not guarantee that this alternative matching will occur*. We call these scenarios *(potentially) elusive matches*. Tricks such as inserting 'padding' delays that can perturb schedules may make elusive matches more likely, but still provide no guarantees. Therefore, we need an algorithm different from DPOR, and POE is our answer.

POE solves the problem of elusive matches *without requiring changes to the MPI library* and *without adding padding delays*. It *dynamically rewrites* wildcard receives into specific receives, one for each actual sender that it computes to be a *certain* match. In the context of the example in Figure 1, if we can force two recursive explorations, with `MPI_Irecv(buffer, from 1, &req);` and `MPI_Irecv(buffer, from 2, &req);` used successively in lieu of the existing line 2, we would have force-matched both the sends. The crucial fact is, of course, to *never* force-match with a send that is not going to be issued – this can cause a deadlock that does not exist. POE employs a strategy to discover all potential senders precisely, as outlined in Section 1.2, and Section 2.

1.2 Handling Out-of-order Completion

In MPI, (i) two `Isend`s targeting two different processes may finish out of order (with respect to issue order), while two `Isend`s targeting the same process must match in order. Likewise, (ii) two non-wildcard receives sourcing from the same source process must also match sends in order. Similarly, (iii) if the first receive or both receives are wildcards, even then they must match in issue order. As for waits and tests, (iv) they must not complete before their corresponding send/receive operations. Finally, (v) operations appearing *after* MPI barriers and MPI waits must not finish before the barrier or wait. *Notice that we did not say that operations before a barrier must finish before the barrier!* Section 2 will show that operations issued before a barrier can linger even after crossing the barrier.

1.3 Discovering Match Sets

POE employs an approach to bound the scope of search for locating potential matching sends for a wildcard receive. It relies on a formal notion of *fences* to determine when two operations issued by a dynamic verification scheduler through the PMPI layer will be carried out (i) by the MPI runtime, (ii) in that order. We are not saying that MPI has "fence instructions" akin to how CPUs have assembly instructions to order intra-core execution. However, there are still conceptually equivalent ordering points defined by the MPI semantics! Based on

[5] In this paper, we suppress details pertaining to communicators and tags.

a formulation of MPI fences, we can form *match sets* – sets of MPI operations that can be issued out of order by a dynamic verification scheduler. This is the idea of POE's *lookahead computation* alluded to earlier.

2 Basic POE Algorithm

Consider Figure 2. Note that although the `Isend` on line 8 is issued *after* the barrier on line 7, it is a potential match for the `Irecv(*)` on line 2. This is precisely because MPI's `Isend` can linger across a `Barrier`. The only ordering that MPI guarantees is that functions *after* a barrier will not be called until all functions before (and including) the barrier have been called on any process (rank). The following steps describe how the dynamic verification scheduler implementing the POE algorithm handles this example. Our POE scheduler will intercept every MPI operation `MPI_f` issued from every MPI process. It will often not issue these operations (through `PMPI_f`) immediately – but only make a note of it, and *later* issue them. Here is how POE will work on our example:

```
1: if (rank == 0)
2:  { MPI_Irecv (&buf0, *, &req);
3:    MPI_Barrier ();
4:    MPI_Wait (&req);
5:    MPI_Recv (&buf1, from 2); }
6:  else if (rank == 1)
7:  { MPI_Barrier ();
8:    MPI_Isend(buf1, to 0, &req);
9:    MPI_Wait (&req); }
10: else if (rank == 2)
11: { MPI_Isend(buf0, to
12:   MPI_Barrier ();
13:   MPI_Wait (&req); }
```

Fig. 2. *Ordering Semantics and Operation Lifetimes*

POE Algorithm:

- Collect `Irecv` (line 2), and do not issue.
- Collect `Barrier` (line 3), and do not issue.
- Since `Barrier` is a fence, do not collect anything more from rank 0; switch to rank 1.
- Collect `Barrier` (line 7), and do not issue; switch to rank 2.
- Collect `Isend` (line 11), and do not issue. Then collect `Barrier` (line 12), and do not issue.
- A fence has been reached in every rank. Now, form a *match set* in priority order, with the following priority order followed: barriers first, then non wildcard send/receives, and finally wildcard send/receives.
- In our current state, there is indeed a highest-priority match set formed by the barriers. *Now, POE sends these Barriers* into the MPI runtime through `PMPI_Barrier` calls.
- The next ordering points (fences) are attained at `Wait`.
- No match sets of non wildcard receives exist. Skip this priority order.
- At this point, we know the full list of senders that can match the wildcard receive.
- Dynamically rewrite `Irecv(*)` into `Irecv(1)` and `Irecv(2)`, in turn.
- Form the first match set of `Irecv(1)` and `Isend()` of line 8. Pursue this interleaving.
- Form the second match set of `Irecv(1)` and `Isend()` of line 11. Pursue this interleaving.

Note that for MPI programs with no wildcards, POE will examine the entire program under exactly one interleaving, thus highlighting the parsimonious search it embodies.

2.1 Semi-Formal Description of POE

The POE algorithm works by finding *match sets* of MPI operations and issuing them (possibly out-of-order) to the MPI runtime (using the PMPI versions of these operations). An MPI operation can essentially be in one of the two states: *issued* and *completed*. When an MPI operation is *issued*, it means that the MPI runtime is aware of the MPI operation. When an MPI operation is *completed*, it means that the operation has no presence in the MPI runtime. For example, when we say that an MPI receive operation is complete, we mean that a matching send has been found for that receive. For simplicity, we only deal with the following MPI operations in this section: `MPI_Barrier`, `MPI_Isend`, `MPI_Wait`, `MPI_Irecv`. We also assume that the operations have the same tag and that the communicator is `MPI_COMM_WORLD` for simplicity.

Since MPI semantics allow for nonblocking operations to linger across barriers, POE needs to emulate this out-of-order completion behavior of the MPI runtime. In addition, POE must also respect MPI's send and receive ordering guarantees. Therefore, rather than emulating the issue order of MPI operations, POE must emulate the *completion order* of MPI operations. Before going into more detail, we first define what we call *fence MPI operations*.

MPI Fence Operations: A *fence* is an MPI operation that must be completed before any following MPI operations from the same process can be issued. Any blocking MPI operation is a fence. `MPI_Barrier`, `MPI_Wait`, `MPI_Recv` are all fences.

POE executes all C statements in program order; however, it issues MPI operations to the MPI runtime only when they are guaranteed to complete. For example, an MPI receive (send) is issued only if a matching send (receive) is found. This is the idea of POE forming *match sets* as introduced in Section 1.3. In order to correctly emulate the out-of-order completion inherent within the MPI semantics (Section 1.2 presents it through examples; our web page [12] has details), POE builds a graph data structure of *completes-before* edges across MPI operations within the same process. We call these edges as *intra completes-before* (IntraCB) edges. In addition to IntraCB, POE also maintains a *conditional completes before* edge. This models how wildcards may *trump* non wildcards. For example, suppose an MPI process P0 has the code sequence `Recv(from 1); Recv(from *);` and MPI process P2 has code sequence `Send(to 0);`. Then this `Send` matches `R(*)` because the first offered match "`from 1`" requires a send from P1 which is not present. However, now if we consider the same P0 process, but a P1 process which is `Send(to 0);`, then this `Send` matches `R(from 1);`.

If there is an intra completes-before edge from i to j, then we call i as the *ancestor* of j. With these details, the POE algorithm proceeds exactly as illustrated on Page 5.

2.2 Implementing `WAIT_ANY` and `TEST_ANY`

ISP implements the POE algorithm that allows for executing MPI operations in an order different from the actual program order. Hence, when ISP traps an MPI request such as `MPI_Irecv(buffer, count, datatype, source, mpi_request)`, ISP stores the arguments for later issue. Let *op* be an MPI operation.

When *op* is one of `MPI_Wait`, `MPI_Waitall`, `MPI_Test`, or `MPI_Testall`, the out-of-order issuance does not cause any problems since the POE algorithm's intra-"completes-before" edges ensure that all ancestors, i.e, the `MPI_Isend`s and `MPI_Irecv`s corresponding to the requests are issued before *op* itself is actually issued. When *op* is one of `MPI_Testany` or `MPI_Waitany`, all `MPI_Irecv` and `MPI_Isend` ancestors of *op* are not necessarily issued before *op* itself is issued. Hence, when ISP invokes *op*, an error is thrown by the MPI runtime that the request structure is invalid (since the MPI runtime is not aware of the as yet unissued `MPI_Isend` or `MPI_Irecv` requests). In order to circumvent this problem, ISP issues *op* with `MPI_REQUEST_NULL` for those send and receive requests that are not yet issued and hence are ignored by the MPI runtime.

3 Experimental Results

We have experimented with all 69 Umpire [2] test cases, and in all 30 tests that have deadlocks, ISP finds the deadlocks, generating the fewest number of interleavings. We have also run ISP on the Monte-Carlo calculation of Pi, and the Game of Life example used in the EuroPVM/MPI 2007 Tutorial [11]. In all examples that do not employ wildcard receives, `WAIT_ANY`, or `TEST_ANY`, *ISP examines exactly one interleaving*. Some of these examples were instrumented to detect resource leaks, and in these cases, ISP guarantees the absence of resource leaks.

The restart time of the MPI system is clearly a dominant overhead. This price is being paid because as opposed to existing model checkers which maintain state hash-tables, we cannot easily maintain a hash-table of visited states including the state of the MPI program as well as the MPI run-time system. (Note: In resorting to re-execution, we are, in effect, banking on deterministic replay.) One very promising approach to eliminate restart overheads is the following. At `MPI_Finalize`, one can reasonably assume that the MPI run-time state is equivalent to the one just after `MPI_Init`, and therefore simply reset user state variables and transition each process to the label after `MPI_Init`. We are further looking into when it is appropriate to use this technique.

In most MPI programs, control flows are unaffected by most (data) variables. This allows us to eliminate those variables (and the associated computations) not contributing to control flows or the local assertions being checked. A preliminary implementation also exists for this algorithm. Also, a preliminary Visual Studio integration of ISP has also been implemented. A problem faced in this implementation was due to the fact that the actual run that occurs under ISP does not ever send wildcard receives into the MPI runtime. This problem was solved through a novel technique that (i) obtains trace information from ISP,

and (ii) mimics the dynamic rewriting of wildcard receives while making the Visual Studio debugger step through error traces. With this approach, the user's view of their program is preserved (more details on our web page [12]).

4 Concluding Remarks

We described our dynamic verification approach for MPI C programs that incorporates partial order reduction and dynamic rewriting based scheduling of MPI function call interleavings. ISP guarantees to detect *all* deadlocks and local assertion violations in C MPI programs that fall within ISP's range of supported commands (the commands and our verification results are documented on our website). MPI programs with additional calls may also be checked using ISP if they don't interfere with the commands currently supported (these commands will directly issue into the MPI runtime, without going through the PMPI mechanism). We detailed how we solved special problems posed by `WAIT_ANY` and `TEST_ANY`, and also how we reconcile a user-interface view with our dynamic rewriting process. We plan to release the full sources of ISP for experimentation, parallelize ISP itself using MPI, and make ISP widely available.

References

1. Salman Pervez, Robert Palmer, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Practical model checking method for verifying correctness of MPI programs. In *EuroPVM/MPI*, pages 344–353, 2007.
2. Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. *Proc. of SC2000*, pages 70–79, 2000.
3. Bettina Krammer and Michael M. Resch. Correctness checking of MPI one-sided communication using Marmot. EuroPVM/MPI, LNCS 4192, pages 105–114, 2006.
4. Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.
5. Richard Vuduc, Martin Schulz, Dan Quinlan, Bronis de Supinski, and Andreas Saebjornsen. Improved distributed memory applications testing by message perturbation. PADTAD - IV, 2006.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
7. Patrice Godefroid. Model checking for programming languages using Verisoft. *POPL*, pages 174–186, 1997.
8. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *POPL*, pages 110–121. ACM, 2005.
9. Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. In *Principles and Practices of Parallel Programming (PPoPP)*, 2008. 285-286.
10. Gerard J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2004.
11. William D. Gropp and Ewing Lusk. Using MPI-2: A Problem-based Approach, 2007. Tutorial.
12. `http://www.cs.utah.edu/formal_verification/europvm-mpi08/ISP`