MAAPED: Predictive Dynamic Analysis Tool for MPI Applications

Subodh Sharma
University of Oxford

Ganesh Gopalakrishnan
University of Utah

Greg Bronevetsky
Lawrence Livermore National Labs

***Abstract*—Formal dynamic analysis of MPI programs is critically important since conventional testing tools for message passing programs do not cover the space of possible non-deterministic communication matches, thus may miss bugs in the unexamined execution scenarios. While modern dynamic verification techniques guarantee the coverage of non-deterministic communication matches, they do so indiscriminately, inviting exponential interleaving explosion. Though the general problem is difficult to solve, we show that a specialized dynamic analysis method can be developed for dramatically reducing the number of interleavings when looking for certain safety properties such as *deadlocks*. Our MAAPED (Messaging Application Analysis with Predictive Error Discovery) tool collects a single program trace and predicts deadlock presence in other (unexplored) traces of an MPI program for the same input. MAAPED hinges on initially computing the potential alternate matches for non-deterministic communication operations and then analyzes such matches which may lead to a deadlock. The results collected are encouraging.**

I. Introduction

We examine the problem of *efficiently* testing Message Passing Interface (MPI) programs for presence/absence of deadlocks. The reason for choosing this problem space is that for a large class of MPI programs, the number of execution schedules is often exponential large, making the verification problem of detecting deadlocks prohibitively expensive. There has been a considerable body of work in formal analysis of MPI applications. These solutions can be broadly classified under: model-checking (MPI-SPIN [1]), testing based (MARMOT [2], UMPIRE [3]), and formal dynamic analysis (ISP [4], [5], DAMPI [6]). We base our work on formal dynamic analysis tools (such as ISP and DAMPI) since we believe that the dynamic analysis tools incorporate the best features of model-checking (coverage) and testing methodologies.

Formal dynamic analysis tools perform a direct-code model checking and stateless replay of the MPI programs over the non-deterministic execution space. These tools employ dynamic partial order reduction (DPOR [7]) technique tailored to MPI as explained in [4]. While it is true that such tools achieve a high degree of execution space reduction by not permuting the order in which *deterministic* MPI actions are issued, they have no choice but to exhaustively examine non-deterministic receives with their potential send matches. The sheer number of options to explore makes these tools an expensive choice.

Problem Statement: The problem we formulate and solve in this paper is the following: Suppose we are given an MPI program which issues wildcard receive or non-deterministic probe calls. We assume that the program has only communication non-determinism: *i.e.*, the code between message passing calls is assumed to be sequential, and is written in a *deterministic* manner (*e.g.*, in C/C++) to carry out computations. We also assume that the choice of which send matches a wildcard receive does not affect subsequent communications in the program. Note that when a particular wildcard receive-send match affects the subsequent control flow of the program, we must explore all potential matches of that receive call since distinct matches may result different executions. However, in programs where above assumptions hold, we would witness that all the MPI calls from the application are issued in a single trace. Provided these assumptions are satisfied, we now want to answer the following question: given a single execution trace of the program, can we analyze this trace and decide (i) either that all subsequent execution scenarios are deadlock free, or (ii) that there exists one alternative execution scenario that *will* deadlock? We have observed that in a large class of SPMD styled MPI programs, the above mentioned assumptions hold. Since, we would log all the MPI calls from the program in a single trace, we can successfully infer all the potential matches of every non-deterministic receive operation. This inference enables us to solve the problem that we posed above.

II. MAAPED Overview

MAAPED tool currently discovers deadlocks in MPI programs. The workflow of the tool is depicted in Figure 1. A trace of the program is obtained through the PMPI layer working along with a ISP type scheduler. Using this trace, the MAAPED *match generator* component computes an over-approximate set of matching receive calls for each of the send operation. Then MAAPED computes the dependencies among the communication events in the trace such as: (i) program ordering among events from the same process (ii) fence orderings enforced by barrier and other blocking operations. Such dependencies are computed by the *dependency constructor* component. Using this dependency information among events, we refine the set of over-approximate match-graph relation. The dependency information is constructively used to eliminate the match possibilities that can never manifest in reality. The refined match-graph is then fed to the deadlock analysis component which predicts if a deadlock is present in some alternate interleaving where a particular send call is left unmatched. Consider the example shown below:

```
P1: R(*);R(*); R(3);
```

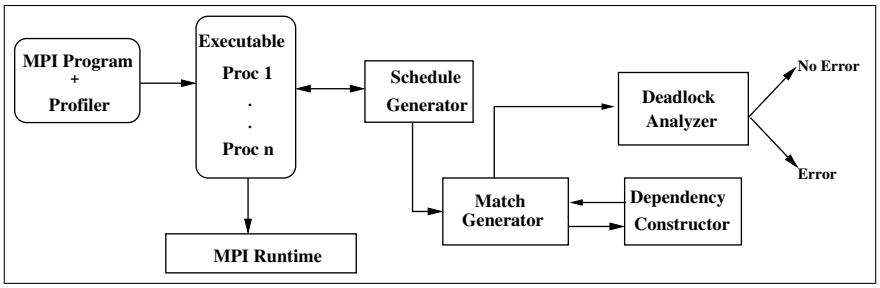

Fig. 1. Workflow of MAAPED

```
P2: S(1);
P3: S(1); S(1);
```

In this example R(*) is a wildcard receive and R(3) is a receive from P3. Similarly S(1) is a send to P1. There is a deadlock in the code, which will manifest only when the first two sends of P3 match with the first two wildcard receives from P1. Suppose the first execution completed successfully without exposing the deadlock (*i.e.*, when send from P2 matched with one of the two wildcard receives from P1). The MAAPED tool will first construct the complete matching possibilities for each send, for instance, P2's send can match P1's first two receives. The dependency constructor will subsequently construct the dependency that P3's first send should match before P3's second send. This dependency will eliminate the matching of P3's second send with P1's first receive. After refining the match-graph, the deadlock analyzer will analyze different matching possibilites and will infer that under certain matching R(3) can get *orphaned* (*i.e.*, when both sends from P3 are matched with the wildcard receives from P1). In this manner MAAPED predicts the deadlock presence without re-running the program. MAAPED showed encouraging results when compared with ISP. The experiments are documented in Table I. The results illustrate that even though MAAPED can

TABLE I
MAAPED VS ISP RESULTS

			Interleaving		T(s)
Test	Procs	Dl? [a]	ISP	M'PED [b]	M'PED
DTG-deadlock	5	Yes	3 †	1 ✓	0.009
Integrate_mw	8	No	> 3500	1	1.669
Matrix Multiply	8	No	120	1	4.564
Gaussian Elimination	8	No	$> 20,000$	1	2.68
Floyd Warshall	8	No	$> 20,000$	1	9.14

[a] Deadlock [b] MAAPED † ISP misses the deadlock under optimized run
✓ MAAPED discovers the deadlock

produce false-alarms, none were found on our benchmarks. MAAPED detected deadlocks with better timing as compared to ISP. The time taken by ISP is not shown, however, one can easily estimate the time taken by ISP to be approximately equal to the product of total number of interleavings explored by ISP and the time taken by MAAPED for a single run.

Future Work: MAAPED tool at the moment cannot handle programs where the communication related control flow is dependent on a prior wildcard receive match. Further, it cannot handle programs where communication involves decoding of the MPI status object (typically used in dynamic load balancing). We term such type of communication pattern by *reply-channel* based communication. We plan to extend the tool to be able to handle programs that fall under such a class. Further, we would also work on proving that all deadlocks (without ommissions) are discovered by the tool without producing any false alarms for a fixed input.

III. CONCLUSION

We present a dynamic analysis tool that discovers latent MPI deadlocks in a predictable and efficient manner as opposed to other dynamic verifiers in the domain and our initial results appear to be encouraging.

ACKNOWLEDGMENT

The research was supported by the grant NSF OCI 1148127. We also thank Sriram Aananthkrishnan for his feedback and discussions related to this topic.

REFERENCES

[1] S. F. Siegel and G. S. Avrunin, "Verification of MPI-based software for scientific computation," in *International SPIN Workshop on Model Checking Software (SPIN)*, 2004, pp. 286–303.

[2] B. Krammer and M. M. Resch, "Correctness checking of MPI one-sided communication using Marmot," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI), LNCS 4192*, 2006, pp. 105–114.

[3] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of MPI applications with Umpire," in *SC*, 2000.

[4] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, "Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings," in *Proceedings of the 20th International Conference on Computer Aided Verification*, ser. CAV '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 66–79. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70545-1_9

[5] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Formal verification of practical MPI programs," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 261–270. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504214

[6] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for MPI programs," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*. ACM, 2010. [Online]. Available: http://www.cs.utah.edu/fv/DAMPI/sc10.pdf

[7] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software." in *POPL*, J. Palsberg and M. Abadi, Eds. ACM, 2005, pp. 110–121.