Efficient Verification Solutions for Message Passing Systems

Subodh Sharma and Ganesh Gopalakrishnan
School of Computing, University of Utah, Salt Lake City, UT
Email: {svs,ganesh}@cs.utah.edu

***Abstract*—We examine the problem of automatically and efficiently verifying the absence of communication related bugs in message passing systems, specifically in programs written using Message Passing Interface (MPI) API and Multicore Communication API (MCAPI). A typical debugging or testing tool will fail to achieve this goal because they do not provide any guarantee of coverage of non-deterministic communication matches in a message passing program. While dynamic verification tools do provide such a guarantee, they are quickly rendered useless when an interleaving explosion is witnessed. The general problem is difficult to solve, though we propose that specialized techniques can be developed that can work on top of dynamic verification schedulers thus making them more efficient.**

In this work, we provide point solutions to deal with the interleaving explosion. Specifically, we present algorithms that accomplish the following tasks: (i) identifying irrelevant message passing operations (Barriers) in MPI programs that add to the verification complexity and degrade application's performance and (ii) reducing substantially the *relevant* set of interleavings using symmetry patterns; that needs to be explored for the detection of *refusal deadlocks* in MPI programs. We also share our experience dealing with non-determinism while verifying MCAPI programs which have a mixed use of shared memory and message passing programming primitives.

***Keywords*-MPI; Dynamic Verification; Partial Order Reduction**

I. Author Info

Author: Subodh Sharma
Advisor: Ganesh Gopalakrishnan
Number of years in PhD program:5

II. Introduction

Message passing systems are ubiquitous with there presence stretching across computing domains - starting from high performance computing all the way to embedded/heterogeneous multicore computing. While MPI [1] is a one such message passing library which is used to program most of today's supercomputers and clusters, MCAPI [2] is an API that is under development for writing programs that will run on low power heterogeneous embedded multicore systems. Although these concurrent libraries provide significant performance boost, they are also a source of worry. The extensive use of non-deterministic communication primitives (*e.g.*, MPI receives with `MPI_ANY_SRC`), the non-deterministic execution environments (*e.g.*, programs where communication is dependent on relative execution latency of participating threads), and unobserved inter-API interactions are the cause of hard to reproduce bugs(*Hiesenbugs*).

Typically the programs written using these APIs are tested by debuggers [3]–[5] and other perturbation based [6] testing tools. While there have been significant advances in the debugging and testing methods, they lack the fine control necessary to explore different schedules arising due to the non-determinism in an message passing program. In short, they fail to provide coverage guarantees.

Model checking tools can provide guaranteed coverage (MPI-SPIN [7]), however constructing models for these codes is often laborious and bug-prone. Recently created *formal dynamic verifiers* such as ISP [8], [9] and DAMPI [10] take an approach that integrates the best features of testing tools (ability to run directly on user applications) and model checking (coverage guarantees). They run the MPI program under the control of a verification scheduler, guarantee to detect all potential matches for non-deterministic (wildcard) receives, and explore each of these matches in different runs of the program. Thus, *they exhaustively explore and ensure full coverage of non-determinism.*

With such unrestrained coverage of non-determinism, the dynamic verifiers like ISP and DAMPI will have to grapple with the inevitable *interleaving explosion*. For instance, consider an MPI program with $n+1$ processes where each of the n processes sends a message to the $(n+1)^{th}$ process. The $(n+1)^{th}$ process posts n wildcard receive calls (say, in a loop). One can easily observe that even in such a simple setting, there will be $n!$ execution schedules. This is clearly unacceptable: all dynamic verifiers must, ideally, be equipped with approaches to detect when such exhaustive explorations are unnecessary, and then avoid them.

Problem Statement: The problem of pruning the interleaving space with multiple identical processes with *symmetric reasoning* is formally undecidable [11]. We do not attempt to provide another approximating solution to this problem. However, we do propose a specialized dynamic analysis method that will substantially reduce the number of interleavings while detecting *communication deadlocks*. We have implemented our initial work in this problem domain on top ISP. Our method is an improvement of ISP's algorithm called POE (partial order avoiding elusive interleavings). We call our method as MSPOE (macroscopic POE).

As a part of a larger effort of interleaving space reduction, we also propose another specialized algorithm to identify

irrelevant synchronizing MPI operations and remove them. Irrelevant MPI operations are those operations which when removed, do not alter the communication structure of the program. There are two-fold reasons for such operations to be removed: (i) they unnecessarily increase the number of MPI operations in the program, thus indirectly affecting the number of schedules a program can have; (ii) they increase the application running time and consequently increasing the verification time of these applications.

Rest of the paper is presented in the following fashion: Section III will briefly present our proposed solution for detecting *functionally irrelevant barriers* (FIB) in an MPI program followed by Section IV where we briefly discuss our MSPOE algorithm. We finally present our experiences with regard to threading and messaging non-determinism while constructing a model checker for embedded heterogeneous multicores in Section V. Finally, we and conclude with remarks on future work in Section VI.

III. Functionally Irrelevant Barriers

The importance of detecting irrelevant barriers comes from a number of perspectives. Many MPI users are known to employ collective barriers for "good measure"; they are unsure whether it is necessary. The authors of [12] narrate the example of an MPI program where a barrier was considered irrelevant and was proposed to be removed. A year later, they realized that its removal introduces a non-benign race condition. In [13], it is shown that barriers can consume a significant fraction of the total application time. Of course, users wanting to control performance by avoiding network or I/O contention may insert collective barriers. In this case, they are employing functionally irrelevant barriers for controlling the non-functional aspects of their program. The FIB algorithm [14] can help these users by checking that these barriers are indeed functionally irrelevant.

Motivating Example: Consider the example:

```
P0: Irecv(*, &handle); Barrier; Wait(&handle);
P1: Isend(to P0);  Barrier; ...rest of P1...
P2: ...some code...    Barrier; Isend(to P0);
```

Notice $Irecv$ is the asynchronous MPI receive and $Wait$ call is a blocking call to check the successful completion of the corresponding non-blocking request. Observe that there is no *Happens-Before* (HB) ordering between $Isends$ of $P1$ and $P2$. The $Irecv$ and $Isend$ from $P0$ and $P1$ can exist past barriers. Thus, removing the barriers will not create any new communication matchings for the $Irecv$ and therefore they are irrelevant.

Now consider an alternative example in which the $Wait$ in $P0$ is moved to be *before* its $Barrier$. Now, the collective barrier becomes **relevant**. This is because there would be a HB edge from $Wait$ to $Barrier$ as shown in Figure 1. Hence, $Barrier$ cannot be crossed until the $Irecv$ finishes. Therefore the $Isend$ from $P2$ cannot issue, and $Irecv$ must finish based on the $Isend$ from $P1$.

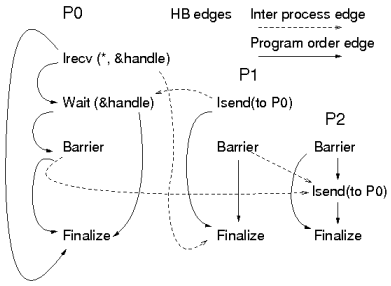

Figure 1. Example in Section III with Happens-Before edges

Results: The algorithm is implemented on top of ISP. Our web page [15] provides detailed results regarding FIB. To briefly summarize :

- **Monte-Carlo**: The code of Monte-Carlo, did not have any barrier calls. To acid-test our implementation, we deliberately inserted an irrelevant collective barrier, which our implementation flagged as such. The run times of the Fib algorithm are as follows: (i) with 4 processes, it explored 6 interleavings in 0.2 seconds, and with 5 processes, it explored 24 interleavings in 1.52 seconds. In both cases, the overhead due to FIB analysis was marginal ($< 1\%$).
- **2D Diffusion** This code had 2 irrelevant barriers which were caught by the tool. In fact, this example does not employ wildcard receives, and so all its barriers are irrelevant, and Fib finishes with one interleaving. The runtime of Fib on this example was less than a second. This reinforces that without wildcards we need only one interleaving.
- **Umpire test suite**: We ran our tool successfully on all the 69 tests that came along with Umpire tool [16]. Of the 36 tests that had barriers, all were flagged as **irrelevant**, with negligible runtimes.

IV. Symmetric MSPOE

Our method MSPOE [17] is implemented by augmenting the ISP tool and its POE. We first let POE compute the potential send matches for MPI non-deterministic receives as it currently does. The execution history, following the non-deterministic receive, is then examined by MSPOE. It chooses to include only some of these sends (called *relevant sends*) to match this non-deterministic receive for later explorations. These sends are the ones considered relevant to cause refusal deadlocks. In effect, instead of exploring all executions MSPOE explores *representative executions* sufficient to reveal refusal deadlock.

Observation: For an MPI program that does not decode data and has a refusal deadlock, it must either have an unequal number of sends and receives in some execution path, or must satisfy the following conditions: (i) it employs a process posting a wildcard receive and a specific receive and

			Interleavings		Time(sec)	
Benchmark	# of procs	Deadlocks?	ISP	ISP+sym-red	ISP	ISP+sym-red
Matrix Multiply	4	No	18	1	2.93	.16
Red-2D-Diffusion	4	No	90	1	29.4	.33
2D-Diffusion	4	No	>5000	1	>3600	3.34
Monte Carlo	6	No	120	1	24.86	.005
Integrate	4	No	81	31	19.46	7.4
Madre	4	No	>8000	1	>3600	.77

Table I

(ii) a previous wildcard receive consumes a send that was meant for the later occurring specific receive, thus orphaning the specific receive. MSPOE exploits this observation and computes relevant sends based on the occurrence of specific receives.

Motivating Example: In the example shown below there is a deadlock introduced by the use of the deterministic receive call.

P0: $S_{0,1}$(to P4);
P1: $S_{1,1}$(to P4);
P2: $S_{2,1}$(to P4);
P3: $S_{3,1}$(to P4);
P0: $R_{4,1}$(*);$R_{4,2}$(from P3);$R_{4,3}$(*);$R_{4,4}$(*);

Figure 2 shows that if $R_{4,1}$ were to match $S_{3,1}$ (rightmost transition from the initial node), the subsequent deterministic call ($R_{4,2}$) will be orphaned, thus creating a refusal deadlock.

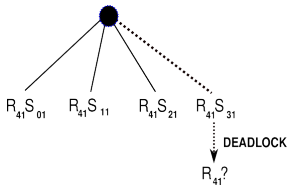

Figure 2. Possibilities after first R(*) match

ISP would explore all the matches starting from leftmost choice shown in Figure 2 and then moving right with every new run, generating four interleavings before finding the deadlock. MSPOE will, on the other hand, choose $S_{3,1}$ as the next relevant send to explore after any initial run. This guarantees that the deadlock will be detected in at most two interleavings.

Results: We implemented the MSPOE algorithm on top of ISP. All experiments were run on Intel Core Duo (2 Ghz) with 3 GB RAM. Table I on page 3 summarizes MSPOE results.

V. Multicore Checker (MCC)

Our main contribution to verify MCAPI applications is the MCC tool [18] that checks the correct use of the connection-less message passing constructs of MCAPI employing a reference implementation of the API. It is essential for a verification scheduler that provides guaranteed coverage to replay the program executions deterministically. This becomes a challenge in MCAPI space because of the following reasons: (i) an external thread scheduler (*e.g.* PThread) which makes it hard for a verification scheduler like MCC to get a control over the thread schedulings, (ii) presence of *only* wildcard receives, and (iii) presence of data dependent communication flow.

Consider the program shown below:

```
P0: S(to 2, 10); | P2: R(*, in x); R(*, in y) ;
P1: S(to 2, 20); |     if(x ==20)
                        assert ERROR;
```

If this example were an MPI program, ISP would *rewrite* the wildcard receives in to the deterministic receives after deciding on a specific match to explore in a program run. In the absence of deterministic receives in MCAPI, it was a challenge for MCC to enforce a deterministic match at the runtime. Consider the following situation: MCC decides to match `R(*, in x)` with send from $P0$. The scheduler signals the participating threads to issue these calls to the runtime. However, notice the thread scheduling is not under the control of MCC. Depending on a thread's execution latency, it is possible that `R(*, in x)` has not finished yet. In the meantime if MCC signals $P1$ to issue its send then we will have a communication race in the MCAPI runtime where sends from both $P1$ and $P2$ are competing to match $P2's$ first receive. Observe that despite MCC having decided a deterministic match, a race was witnessed at runtime. MCC handles this situation by introducing extra *fence* operations in the instruction stream to make sure the intended send-receive match *completes* before next matching pair of events are issued in to the runtime.

Thus, it is of paramount importance for any developer of a verification scheduler to obtain a good grip of the program's execution environment in order to explore and *re-play* the execution runs deterministically.

VI. Conclusions and Future Work

We have presented our initial proposal to combat non-determinism while verifying message passing systems. Our

initial results are inspiring. We obtained dramatic interleaving space reductions using our MSPOE algorithm; in some cases, from million interleavings to one. Our FIB method successfully detected multiple irrelevant barriers in several MPI benchmarks. We finally presented our experiences and lessons learnt while developing the MCC verifier for programs with mixed use of programming models. All of the above accomplishments are a part of our multifarious attack on problems in the message passing arena.

Future Work: We will extend our MSPOE algorithm to handle cases which have an obvious received data decoding that controls the ensuing communication flow of the program. To handle data decoding, we would require an MPI specific control flow graph (CFG) of the program. In [19], a CFG for MPI programs (*p-cfg*) is presented. However, *p-cfg* can handle only deterministic MPI programs. We intend to work on improving the *p-cfg* to handle non-deterministic MPI operations. Furthermore, we will develop flow-sensitive static analysis methods on top of the improved *p-cfg* to analyze conditional communication patterns. After analyzing the MPI programs with such patterns, the dynamic verification scheduler could be alerted to back-off from aggressive interleaving reduction optimizations for specific instance of wildcard receives.

We will also study the mixed use of programming models to program heterogeneous multicores and plan to device a stricter *Happens-Before* orderings in programs where inter-API interactions manifest. Such interactions, till date, are not properly semantically characterized. A poor shared memory interaction leading to a data-race may eventually cause a communication deadlock. Such scenarios will not only hard to debug but equally hard to reproduce.

REFERENCES

[1] "Message Passing Forum." [Online]. Available: http://www.mpi-forum.org/docs/

[2] "Multicore Association," http://www.muticore-association.com.

[3] B. Krammer, K. Bidmon, M. Mller, and M. Resch, "Marmot: An mpi analysis and checking tool," in *Parallel Computing - Software Technology, Algorithms, Architectures and Applications*, ser. Advances in Parallel Computing, F. P. G.R. Joubert, W.E. Nagel and W. Walter, Eds. North-Holland, 2004, vol. 13, pp. 493 – 500.

[4] "TotalView Concurrency Tool." [Online]. Available: http://www.totalviewtech.com

[5] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalatrace: Scalable compression and replay of communication traces for high-performance computing," *J. Parallel Distrib. Comput.*, vol. 69, pp. 696–710, August 2009.

[6] R. Vuduc, M. Schulz, D. Quinlan, B. de Supinski, and A. Sæbjørnsen, "Improving distributed memory applications testing by message perturbation," in *PADTAD '06: Proceeding of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*. ACM, 2006, pp. 27–36.

[7] S. F. Siegel and G. S. Avrunin, "Verification of mpi-based software for scientific computation," in *Model Checking Software: 11th International SPIN Workshop*. Springer-Verlag, 2004, pp. 286–303.

[8] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, "Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings," in *Computer Aided Verification (CAV 2008)*, 2008, pp. 66–79.

[9] 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 PPoPP*, ser. PPoPP. New York, NY, USA: ACM, 2009, pp. 261–270.

[10] 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," *SC Conference*, vol. 0, pp. 1–10, 2010.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[12] G. S. Avrunin, S. F. Siegel, and A. R. Siegel, "Finite-state verification for high performance computing," in *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, ser. SE-HPCS. New York, NY, USA: ACM, 2005, pp. 68–72.

[13] R. Rabenseifner, "Automatic profiling of mpi applications with hardware performance counters," in *Proceedings of the 6th EuroPVM/MPI*. London, UK: Springer-Verlag, 1999, pp. 35–42.

[14] S. Sharma, S. S. Vakkalanka, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp, "A formal approach to detect functionally irrelevant barriers in mpi programs," in *PVM/MPI*, 2008, pp. 265–273.

[15] S. Sharma, "FIBResults," 2008. [Online]. Available: http://www.cs.utah.edu/ svs/FIB_results/

[16] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of mpi applications with umpire," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '00. Washington, DC, USA: IEEE Computer Society, 2000.

[17] S. Sharma and G. Gopalakrishnan, "Scalable analysis for deadlock detection in mpi programs," *Submitted to ICS*, 2011.

[18] S. Sharma, G. Gopalakrishnan, E. Mercer, and J. Holt, "Mcc: A runtime verification tool for mcapi user applications," in *FMCAD*, 2009, pp. 41–44.

[19] G. Bronevetsky, "Communication-sensitive static dataflow for parallel message passing applications," in *CGO*, 2009, pp. 1–12.