Scalable Verification of MPI Programs

Anh Vo and Ganesh Gopalakrishnan
School of Computing, University of Utah, Salt Lake City, UT
{*avo,ganesh*}*@cs.utah.edu*

Abstract

Large message passing programs today are being deployed on clusters with hundreds, if not thousands of processors. Any programming bugs that happen will be very hard to debug and greatly affect productivity. Although there have been many tools aiming at helping developers debug MPI programs, many of them fail to catch bugs that are caused by non-determinism in MPI codes. In this work, we propose a distributed, scalable framework that can explore all relevant schedules of MPI programs to check for deadlocks, resource leaks, local assertion errors, and other common MPI bugs.

1. Author Info

Author: Anh Vo
Advisor: Ganesh Gopalakrishnan
Number of years in PhD program:3

2. Introduction

The Message Passing Interface (MPI) [7] library remains one of the most widely used APIs for implementing distributed message passing programs. Its projected usage in critical, future applications such as Petascale computing [6] makes it imperative that MPI programs be free of programming logic bugs. This is a very challenging task considering the size and complexity of optimized MPI programs.

In particular, performance optimizations often introduce many types of nondeterminism in the code. For example, the *MPI_Recv(MPI_ANY_SOURCE, MPI_ANY_TAG)* call that can potentially match a message from any sender in the same communication group (we will later refer to this as a *wildcard receive*) is often used for re-initiating more work on the first sender that finishes the previous item of work. A more general version of this call is the *MPI_Waitsome* call that waits for a subset of the previously issued communication requests to finish. These nondeterministic constructs potentially can result in MPI program bugs that manifest intermittently – the bane of debugging. Traditional MPI debugging tools such as Marmot [10] insert delays during repeated testing under the same input to perturb the MPI runtime scheduling. Experience indicates that this technique is often unreliable [1]. In order to detect all scheduling-related bugs, the framework under which MPI programs are debugged needs to have the ability to *determine* and *enforce* all *relevant* schedules (the concept of relevant scheduled will be explained in 4.1.2). ISP (In-Situ Partial Order) [17], [18], [20], [21], the current state of the art dynamic verifier for MPI programs, is currently the only known tool that has this ability.

However, the current framework of ISP does not provide good scalability when operating on large clusters, an environment where many large MPI applications are currently deployed. In addition, there are bugs that only manifest when the application scales up beyond certain threshold, and some bugs only manifest under a distributed environment. The next-generation framework will have to possess the following abilities: (i) detect and enforce all relevant schedules of MPI programs, (ii) work in distributed settings, and (iii) have good scalability. To this end, we have designed DMA (Distributed Message Passing Analyzer), a scalable distributed framework that can satisfy all the above requirements.

In the rest of this paper, we will provide an overview of the framework (figure 1 shows the proposed design of DMA), as well of the status of the work.

3. Related Work

In recent years, considerable effort has been spent on building efficient verification tools for MPI programs such as [10], [12], [19]. However, none of those tools offer the three basic abilities that we discussed earlier (able to detect and enforce relevant interleavings, distributed, and scalable). For example, tools such as Marmot have been shown in our experiments to miss very simple deadlocking scenarios, as shown in [1]. In [8], a scalable approach to detect deadlocks in MPI programs was proposed. Yet this approach relies on the deadlock actually happening in the current run to detect it. It does not have the ability to enforce different relevant scheduling to detect whether the deadlock could have happened in another schedule.

MPI-SPIN [15], [16], a model checker based on SPIN, can detect and exhaustively explores all schedules of MPI programs. However, MPI-SPIN requires the users to manually build a model for the program being verified, which is an impractical task, especially for non-computer scientists.

As mentioned earlier, ISP has the ability to determine and enforce all relevant schedules of MPI programs. However,

978-1-4244-6534-7/10/$26.00

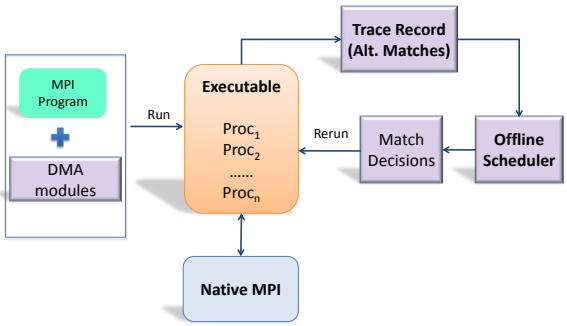

Figure 1. DMA Framework

the current structure of ISP does not provide good scalability. To the best of our knowledge, DMA is the only tool that has the potential to offer the same coverage as ISP does, at a much larger scale.

4. Research Overview

4.1. DMA Design

Figure 1 shows the proposed design of DMA. At a high level, DMA is designed as a series of modules, each operating as a PnMPI [14] module and is responsible for different functionalities of DMA. The modules are compiled as a dynamic shared library and are linked together with the MPI program. When the DMA-linked code executes, each process will detect alternative outcomes of the nondeterministic events that happened within the process and record them into some trace record. At the end of the initial run, an offline scheduler will analyze the trace record and create a schedule for the processes to follow upon restart. The processes are then restarted if necessary until all matches are explored. Subsequent runs by the processes might create new schedules that requires the scheduler to be reinvoked automatically.

4.1.1. PnMPI. DMA is designed to operate as a P^{N}MPI [14] module. P^{N}MPI extends the PMPI profiling interface to support multiple PMPI-based tools (ISP is an example of a PMPI-based tool). The advantages of using P^{N}MPI are as follows:

- P^{N}MPI allows the implementation of the DMA tool to be split up into multiple layers, with each layer addressing orthogonal issues (interleaving generation layer, resource leak tracking layer, deadlock detection layer, etc.).
- P^{N}MPI also eliminates the need to recompile the MPI target code every time changes are made to DMA

```
P0: MPI_Isend(to P1, data = 42); ...
P1: MPI_Irecv(*, x); if (x==42) then error1
else ...
P2: MPI_Isend(to P1, data = 21); ...
```

Figure 2. Simple MPI Example Illustrating Wildcard Receives

- Each layer or all of DMA layers can be turned off through a simple configuration file, which enables the developers quickly to choose which aspect of the program he wants to debug (or not debug at all)
- P^{N}MPI has very low overhead. For more details on P^{N}MPI overhead, please see [14]

4.1.2. Relevant Interleavings - What and Why. Consider the example in figure 2, in which the `MPI_Irecv` issued by `P1` can match either the `MPI_Isend` by `P0`, or the `MPI_Isend` by `P2`. Only the match by `P0` would result in an error. This example shows that different schedules of MPI programs do indeed matter, and only by discovering and examining these alternative schedules can we hope to have bug-free MPI programs.

However, if all possible schedules of an MPI programs are considered, a considerable degree of wasted testing happens. For example, assume that all processes in figure 2 issue an `MPI_Barrier` call before issuing the respective send/receive. A tool that is not able to pick up relevant MPI schedules will simply permute all different orders of issuing those barrier calls while in fact, it does not matter which barrier is issued first. This is why a scalable solution needs to consider only *relevant* schedules.

4.1.3. Detect and Enforce Relevant Interleavings. Unlike ISP, which chose a centralized approach to detection of alternative matchings (see [17] for more details), DMA opts for a totally distributed approach. Each process is responsible for recording all the alternative outcomes of the nondeterminism events that occur within the process itself. To accomplish this, each process needs to construct its *view of the world* through *piggybacking* (i.e., sending extra information in each MPI message), which itself is implemented as a P^{N}MPI module.

With the help of piggybacking, the distributed system as a whole will attempt to use piggyback data to construct a *causal line* C for each non-deterministic event e. Intuitively, C is an imaginary line that is formed by all the events that are provably (through the extra information obtained in piggybacking) caused by the non-deterministic event e. When a message m from `Q` is received by a process `P`, `P` will inspect the piggyback information and determine whether this message is *late* or not with respect to e (we shall skip the formal definition of *late* in the scope of this paper, but one can intuitively think of late as if the message could

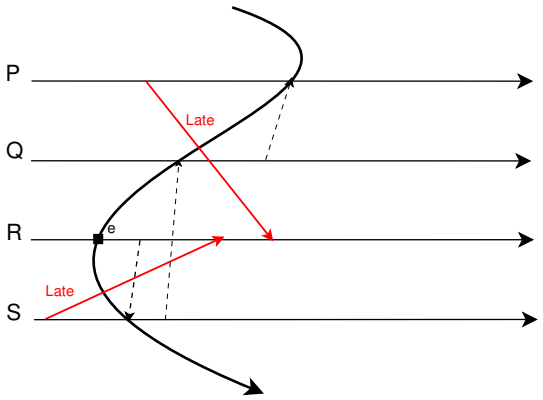

Figure 3. Causal Line and Late Messages

have been matched, but it arrived a bit late, so the slot has been taken by an earlier message). If it is considered late and it's eligible to match with e, it will be recorded as an alternative to e. Figure 3 helps visualize causal line and late messages. In the figure, the dashed arrows represents the messages carried piggyback information which is used to construct the causal line while the solid red arrows represent messages that cross the causal line (sent before and received after) and thus determined as late.

4.1.4. Completeness vs Performance. DMA in its current state is designed with two different protocols for handling late messages: one that uses a Lamport clock [11] (DMA-L) and one that uses a sparse representation of vector clocks [5], [13] (DMA-V) (the sparse representation itself is part of the design, but the details are omitted in this paper). Each protocol is designed to serve different sets of programs. DMA-L ignores the potential dependency between concurrent non-deterministic events, in which certain choices made by one event can effect the number of choices for the other event. On the other hand, DMA-V takes into account all the possible dependency between concurrent events. Obviously, DMA-V has a higher overhead than the DMA-L and also does not scale as well as DMA-L. However, based on previous experience working with ISP, many programs do not exhibit dependency between concurrent non-deterministic events.

5. Progress and Future Work

DMA was initially designed during my internship at Lawrence Livermore National Lab in summer 2009 under the mentorship of Dr. Bronis de Supinski. Since then, the designed has been continually evolved and has great potential. Initial experimental results conducted to evaluate the overhead of DMA are very promising. While ISP takes about 40 minutes to verify Parmetis [9] running with 32 processes in a workstation setting, the same verification run is completed under DMA in less than a minute (also in a workstation). Note that Parmetis does not make any non-deterministic MPI calls, the experiment is conducted purely in the interest of examining the overhead of the framework. The offline scheduler of DMA is still in the implementation phase and thus the experiments to measure the schedules exploration are yet to be carried out.

Future Work: When the implementation of DMA is finished, we plan on assessing its performance on some large MPI benchmarks such as the Sequoia [2] benchmark suite, MPI-Blast [3], and SpecMPI2007 benchmark suite [4]. The experiments will be carried out on the high performance clusters at Lawrence Livermore National Lab.

Following the completion of the implementation, our next step will be to investigate methods to address the *interleavings explosion* problem. Consider an MPI program that was designed based on a master-slave model in which the master posted a series of wildcard receives to respond to any request from the client. If there are N slaves, there will be N potential matches for each of these wildcard. If the master makes m of those wildcard calls, the total number of interleavings that we have to consider is N^m. For a modest program with 10 such wildcard receives, running with 32 processes in which 31 are slaves, the total number of interleavings will be about 819 trillion(31^{10})! Assuming that the program is coded such as the result of matching one wildcard receive does not affect the next set of matches (but the different matching of one wildcard matters), the whole program could have been verified in 310 interleavings. If the program is coded such as all the matches are completely independent, the whole program could have been verified in 1 interleaving. Recognizing such usage patterns in the program will significantly reduces the number of interleavings the analyzer has to explore. This will be our main focus during the next step of this work.

Acknowledgements: We would like to thank Sarvani Vakkalanka, the creator of ISP, which inspires the design of ISP. We also thank Bronis de Supinski, Martin Schulz, and Greg Bronevetski for their joint work on DMA.

References

[1] http://www.cs.utah.edu/formal_verification/ISP_Tests/.

[2] https://asc.llnl.gov/sequoia/benchmarks.

[3] http://www.mpiblast.org.

[4] http://www.spec.org/mpi.

[5] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *ACSC*, pages 56–66, 1988.

[6] A. Geist. Sustained Petascale: The next MPI challenge. Invited Talk at EuroPVM/MPI 2007.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[8] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller. A graph based approach for mpi deadlock detection. In *ICS 2009*.

[9] G. Karypis. METIS and ParMETIS. http://glaros.dtc.umn.edu/gkhome/views/metis.

[10] B. Krammer, K. Bidmon, M. S. Mller, and M. M. Resch. Marmot: An MPI analysis and checking tool. In *Parallel Computing 2003*, Sept. 2003.

[11] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15:93–100, 2003.

[13] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[14] M. Schulz and B. R. de Supinski. P^nMPI tools: a whole lot greater than the sum of their parts. In *SC*, page 30, 2007.

[15] S. F. Siegel and G. S. Avrunin. Verification of MPI-based software for scientific computation. In *SPIN* 2004.

[16] S. F. Siegel and L. F. Rossi. Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In *EuroPVM/MPI* 2008.

[17] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *CAV* 2009

[18] S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby. Reduced execution semantics of mpi: From theory to practice. In *FM*, pages 724–740, 2009.

[19] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing*, pages 70–79, 2000.

[20] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, , and R. Thakur. Formal verification of practical mpi programs. In *PPoPP*, pages 261–269, 2009.

[21] A. Vo, S. S. Vakkalanka, J. Williams, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Sound and efficient dynamic verification of mpi programs with probe non-determinism. In *EuroPVM/MPI*, pages 271–281, 2009.