

# Seamless Integration of Two Approaches to Dynamic Formal Verification of MPI Programs

Anh Vo, Ganesh Gopalakrishnan, Sarvani Vakkalanka, Alan Humphrey and Christopher Derrick

School of Computing, University of Utah, UT, USA  
 {avo,ganesh,sarvani,ahumphre,cderrick}@cs.utah.edu

## Abstract

We have built two tools for dynamically verifying MPI programs – one called ISP and the other called DMA. Both these tools are aimed at formally analyzing the executions of an MPI programs by running executions, analyzing the actual MPI operation dependencies that manifest, and rerunning executions to cover the dependency space. ISP implements an MPI-specific dynamic partial order reduction algorithm called POE while DMA implements a new distributed algorithm based on logical clocks. While ISP is able to scale up to tens of processes on medium-sized applications and offer more precise coverage of the dependency space, DMA is designed to scale up to thousands of processes on large applications, offering reduced (yet formally characterizable) coverage. In this position paper, we briefly describe ISP and DMA and explain the need to maintain both these tools – yet seamlessly integrate them within a common integration platform, with our current choice being Eclipse Parallel Tools Platform (PTP).

## 1. Introduction

Anyone who writes any program – concurrent or not – has to test it. Focussing on message passing MPI [1] based *deterministic* programs, it is well known that they offer a very high degree of isolation between processes, with all process actions being independent across processes. For such programs, conventional testing is quite effective, as running one schedule is equivalent to running any other schedule. However, in programs with non-deterministic MPI calls, it is known how skewed the matches can be – *e.g.*, see [2] where we demonstrated unacceptable bug omission rates on the Umpire test suite when employing conventional testing methods. For such programs, issue-time modulation methods that depend on inserting non-deterministic sleep durations (*e.g.*, [3, 4]) are not particularly effective because altering the issue time of MPI calls is often not going to change the way in which racing MPI sends find matches with MPI non-deterministic receives deep inside the MPI runtime. These delays also un-necessarily slow down the entire MPI program. Last but not least, nothing formal can be stated when a delay-based testing method finds no violations – there could still be serious bugs left!

Beginning with [5, 6] we demonstrated a new method for *active testing* of MPI programs that is tantamount to the formal process

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]... \$5.00

of *partial order reduction* that computes *persistent sets* [7, 8] at every state. This technique has been implemented as a tool called ISP that has been widely demonstrated. Recently, we have built the Graphical Explorer of Message passing (GEM) tool [9] that is now part of the Parallel Tools Platform [10] end user runtime.

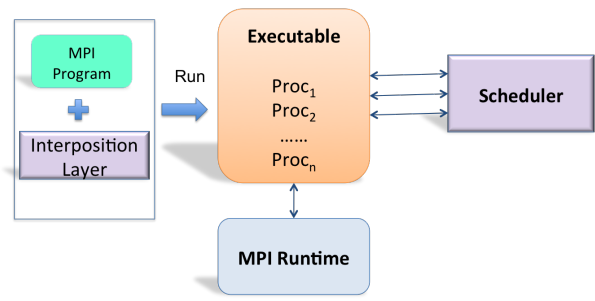


Figure 1. Overview of ISP

$P_0$	$P_1$	$P_2$
$Isend(to : 1, 22);$	$Irecv(from : *, x)$	$Barrier;$
$Barrier;$	$Barrier;$	$Isend(to : 1, 33);$
	$if(x == 33)bug;$	

Figure 2. MPI Example

Given an example such as in Figure 2, the ISP tool (Figure 1) seeks to discover all non-deterministic matches. If  $P_2$ 's  $Isend$  can match  $P_1$ 's  $Irecv$ , we have a bug – but can this match occur? The answer is yes: first, let  $P_0$ 's  $Isend$  and  $P_1$ 's  $Irecv$  be issued; then the execution is allowed to cross the  $Barrier$  calls; after that,  $P_2$ 's  $Isend$  can be issued. At this point, the MPI runtime faces a non-deterministic choice of matching either  $Isend$ . Notice that this particular execution sequence can be obtained only if the  $Barrier$  calls are allowed to match *before* the  $Irecv$  matches. Existing MPI testing tools cannot exert such fine control over MPI executions. Thanks to the theory of *happens before* that we introduced in [11, 8], ISP can exert this fine degree of execution control.

In more detail, by interposing a scheduler (Figure 1), ISP is able to safely reorder, at runtime, MPI calls issued by the program. In our present example, ISP's scheduler (i) *intercepts* all MPI calls coming to it in program order, (ii) dynamically reorders the calls going into the MPI runtime (ISP's scheduler sends *Barriers* first; this is correct according to the MPI semantics), and (iii) at that point discovers the non-determinism.

Once ISP determines that two matches must be considered, it re-executes (replays from the beginning) the program in Figure 2

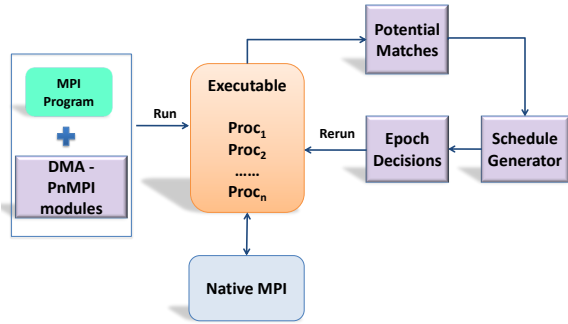


Figure 3. DMA Framework

twice over: once where  $P_0$ 's *Irecv* is considered, and the second time where  $P_2$ 's *Irecv* is considered. But in order to ensure that these matches do occur, ISP must dynamically rewrite *Irecv(from : \*)* into *Irecv(from : 0)* and *Irecv(from : 2)* in these replays. If we did not so determinize the *Irecv*s, but instead issued *Irecv(from : \*)* into the MPI runtime, such a call may match *Irecv* from another process, say  $P_3$ . In summary, (i) ISP achieves discovers the maximal extent of non-determinism through dynamic MPI call reordering, (ii) it achieves scheduling control of relevant interleavings by dynamic instruction rewriting. While pursuing relevant interleavings, ISP detects the following error conditions: (i) deadlocks, (ii) resource leaks (e.g., MPI object leaks), and (iii) violations of C assertions placed in the code. ISP re-runs the code through all the relevant interleavings. For the given MPI program operating under the given input data set, ISP guarantees to find all deadlocks, resource leaks, and violations of local assertions (e.g., C `assert` calls placed in the code).

## 2. Scalable Dynamic Verification

It is clear that ISP's approach will scale only as much as its centralized scheduler will allow. This has lead us to totally re-design the determination of happens-before by developing a decentralized algorithm that uses logical clocks (Lamport clocks or Vector clocks). The architecture of this tool is shown in Figure 3.

DMA works as follows:

- All processes maintain "time" or "causality" through *Lamport Clocks* (which are a frequently used optimization in lieu of the more precise but expensive *Vector Clocks*).
- Each MPI call is trapped (using P<sup>N</sup>MPI [12] instrumentation) at the node where the call originates. For deterministic receive operations, the local process updates its own Lamport clock; for deterministic sends, it sends the latest Lamport clock along with the message payload using *piggy-back messages*. All Lamport clock exchanges occur through *piggy-back messages*.
- Each *non-deterministic* receive advances the Lamport clock of the local process. During execution, this receive will have matched one of the MPI sends targeting this process (else we would have caught a deadlock and reported it). However, each send that does not match this receive but impinges on the issuing process is analyzed to see if it is *causally concurrent* (computed using Lamport clocks). If so, it is recorded as a *Potential Match* in a file.

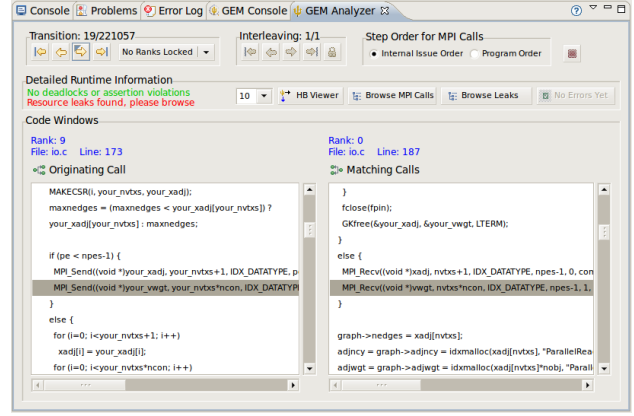


Figure 4. Analyzer View on ParMETIS

- At the end of the initial execution, DMA's scheduler computes the *Epoch Decisions* file which has the information to force alternate matches. Now, DMA's scheduler proceeds to carry out a depth-first walk over all Epoch Decisions (replay alternative matches at the last step; then at the penultimate step; and so on till the Epoch Decisions are exhausted).

While our work on DMA is less than six months old, it has already shown considerable promise in terms of large-scale dynamic verification. We are also beginning to understand the tradeoffs between DMA and ISP. While DMA can handle many every-day examples, it cannot for instance yet handle the example in Figure 2 because of the fact that when  $P_1$ 's *Barrier* is encountered, the *Wait of Irecv* of  $P_2$  would have made  $P_1$ 's clock go up. However *Barrier* calls for taking the global maximum of the Lamport clock values across processes. Thus in effect, the increased clock of  $P_1$  incorrectly participates in the global max operation which is then broadcast to all processes. Scalable solutions to this (and other situations) are under consideration. We also know of other situations where instead of Lamport clocks, Vector clocks must be employed for more precise verification.

## 3. Integration of ISP and DMA within GEM

GEM was created to be a graphical interface that provides a means for users to easily launch ISP and then display the results of the analysis in a straightforward and intuitive manner. A brief summary of the results is provided in the section titled Detailed Runtime Information (Figure 4 shows GEM performing an analysis). This is where errors are brought to the user's attention.

To examine an error more closely a user need only press the Examine Errors Button (grayed out in Figure 4 and with the text changed to No Errors Yet because the file being analyzed has no MPI Errors) to have a popup window launched that describes each of the errors and which opens the editor with the offending line highlighted in a view when one of these errors is double clicked. Also in this section there are buttons that let a user Browse Leaks (which operates much the same as Browse Errors, but displays resource leaks in place of errors) and Browse MPI Calls (which shows a tree representation of every call made).

The other sections in the analyzer are designed to let the user step through the code to see the runtime behavior. The section marked Interleaving lets a user change which interleaving (aka schedule) is selected and the section marked Transition lets a user change which call in that interleaving is being displayed. The large section at the bottom titled Code Windows shows the current call

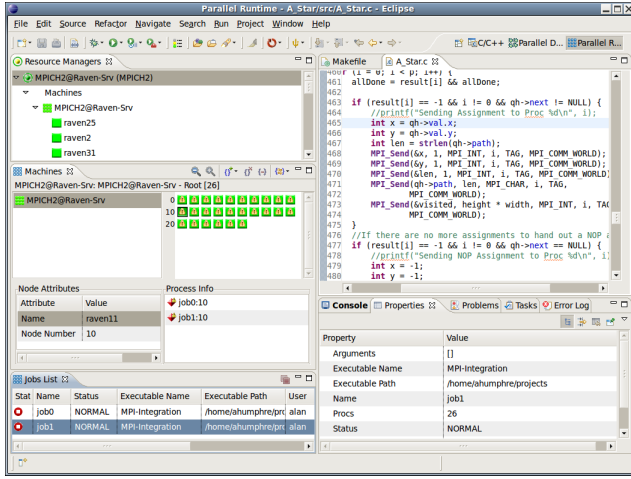


Figure 5. Eclipse PTP Parallel Runtime Perspective

in the left half and every matching call in the right half (in Figure 4 there is only one match so detailed information is provided, if the call is a collective operation and is matched by calls found in distinct locations of the code a list is provided and by clicking on an element of the list the editor is opened with the selected call highlighted and in view).

GEM has many other features that due to size limitation cannot be discussed further. Suffice it to say that this integration framework provides very intuitive feedback to MPI programmers about the formal verification results returned by ISP and those who desire further details are encouraged to view our technical report [9].

We are in the processes of integrating DMA into GEM in much the same way that ISP already is, this will allow the strengths of both tools to be used in mutually complementary ways without making the verification process more complicated. Another exciting aspect of this integration rises from the fact that since we can schedule jobs on cluster machines using the Eclipse PTP Parallel Runtime Perspective shown in Figure 5, a designer will have one cockpit to drive formal dynamic verification tools from (Figure 6 shows this "cockpit"). This opens up possibilities for studying how tools that effect meaningful coverage/scalability tradeoffs can be synergistically combined – and perhaps also combined with other PTP based tools (such as for performance evaluation and conventional testing methods). For example, designers may be able to generate scenarios of interest by running ISP, and then dispatch them for in-depth analysis at scale using DMA. This would represent a nice combination of approaches where the expensive (and harder to use) cluster machine is not an essential prerequisite to have access to before verification at scale can be conducted. Other techniques such as [13] may also be integrated into this tool framework.

Last but not least with "cloud computing" resources likely to become widely available, one can also imagine scenarios where we partition the state space of the models being verified into parts that can be separately examined on the cloud. Again, the mindset here is to use production-quality cluster supercomputers only for actual long-lasting high-performance simulation runs, while using other machine types such as the cloud as debugging servers.

**Acknowledgements:** The authors like to thank Bronis R. de Supinski, Martin Schulz, Greg Bronevetsky and Beth R. Tibbitts for ideas and encouragement.

## References

[1] "MPI 2.1 Standard," MPI Standard 2.1, <http://www.mpi-forum.org/docs/>.

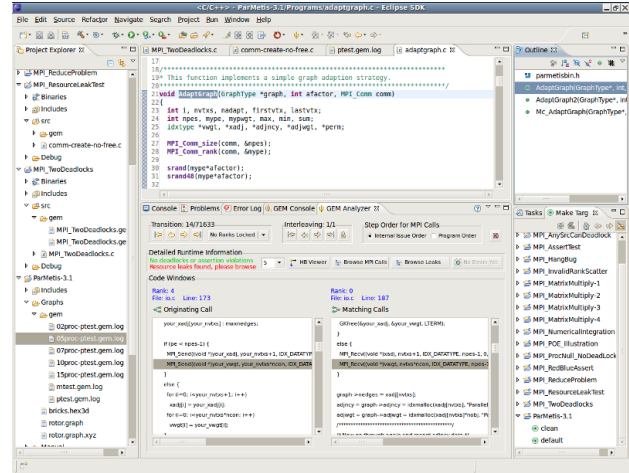


Figure 6. Shows how GEM is integrated into the rest of Eclipse

[2] "Test results comparing isp, marmot, and mpirun," [http://www.cs.utah.edu/fv/ISP\\_Tests](http://www.cs.utah.edu/fv/ISP_Tests).

[3] S. Coptly and S. Ur, "Toward automatic concurrent debugging via minimal program mutant generation with aspectj," *Electr. Notes Theor. Comput. Sci.*, vol. 174, no. 9, pp. 151–165, 2007.

[4] R. Vuduc, M. Schulz, D. Quinlan, B. de Supinski, and A. Sæbjørnsen, "Improving distributed memory applications testing by message perturbation," in *Proc. 4th Parallel and Distributed Testing and Debugging (PADTAD) Workshop, at the International Symposium on Software Testing and Analysis*, Portland, ME, USA, July 2006.

[5] 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.

[6] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Formal verification of practical mpi programs," in *Principles and Practices of Parallel Programming (PPOPP)*, 2009, pp. 261–269.

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

[8] S. Vakkalanka, *Efficient Dynamic Verification Algorithms for MPI Applications*. PhD Dissertation, 2010, <http://www.cs.utah.edu/~sarvani/dissertation.html>.

[9] "Gem - isp eclipse plugin," <http://www.cs.utah.edu/formal-verification/ISP-Eclipse>.

[10] "The Eclipse Parallel Tools Platform," <http://www.eclipse.org/ptp>.

[11] S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby, "Reduced execution semantics of mpi: From theory to practice," in *FM 2009*, Nov. 2009, pp. 724–740.

[12] M. Schulz and B. R. de Supinski, "P<sup>n</sup> mpi tools: a whole lot greater than the sum of their parts," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, ISBN:978-1-59593-764-3.

[13] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, "A graph based approach for MPI deadlock detection," in *ICS 2009*, pp. 296–305.