GEM: Graphical Explorer for MPI Programs

Alan Humphrey, Christopher Derrick,
Ganesh Gopalakrishnan and Beth Tibbits

UUCS-09-007

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

December 28, 2009

Abstract

Formal dynamic verification can complement MPI program testing by detecting hard-to-find concurrency bugs. In previous work, we described our dynamic verifier called ISP that can parsimoniously search the execution space of an MPI program while detecting important classes of bugs. One major limitation of ISP, when used by itself, is the lack of a powerful and widely usable graphical front-end. We present a new tool called Graphical Explorer of Message Passing (GEM) that overcomes this limitation. GEM is a plug-in architecture that greatly enhances the usability of ISP, and may help bring ISP within reach of a wide array of programmers, given its imminent release as part of the Eclipse Foundation Parallel Tools Platform (PTP) Version 3.0. This paper describes GEM's features, its architecture, and usage experience summary of the ISP/GEM combination. Recently, we applied this combination on a widely used parallel hypergraph partitioner. Even with modest amounts of computational resources, the ISP/GEM combination finished quickly, and intuitively displayed a previously unknown resource leak in this code-base.

1 Introduction

Over the past two decades, high performance computing (HPC) has evolved from the domain of the expert programmer to become an everyday approach used by engineers and researchers. A majority of these parallel programs employ the message passing interface (MPI [1]) library for interprocess communications and for invoking collective operations such as *barriers* and *reductions*. MPI continues to enjoy a dominant position in HPC, and has been ported to run on virtually every parallel machine available today. Given the extensive presence of MPI, it is imperative that highly effective debugging tools be created for MPI programs. Today, there are an impressive array of tools available for debugging MPI programs. These tools tend to provide extensive facilities for stepping through process executions and graphically visualizing executions. Unfortunately, these tools only provide *ad hoc* techniques for process interleaving (schedule) generation, and as a result, many interleavings are not considered. In practice, these omitted interleavings are known to harbor bugs [2]. Considering all interleavings is not an option because there are an astronomical number of them (*e.g.*, over 10 billion for a five-process MPI program where each process performs merely five MPI calls).

Formal verification methods can help parsimoniously search the execution space of an MPI program while detecting important classes of errors. It is essential that a practical formal verification tool for MPI programs directly accept user source codes, and not rely upon hand-built models of the code, as needed by all other formal tools (*e.g.*, [3]). Obtaining such models is next to impossible in practice, considering the difficulty of modeling the C/MPI semantics and the rapidity with which programs are changed during optimization cycles. Our tool ISP [4, 5, 6, 7] (summarized in § 1.1) is currently the only such tool.

Previously, the usage of ISP was hindered by the absence of a widely usable and intuitive graphical user interface. This paper describes our contribution in this regard of a tool called *Graphical Explorer of Message passing* (GEM). GEM borrows many ideas from our own past work in this area, namely the integration of ISP within Visual Studio [8]. However, our past work was insufficiently general. Besides, Visual Studio runs on proprietary Windows platforms, whereas the HPC community often prefers working with non-commercial software. Most relevant to this paper is the fact that GEM is designed to serve as an Eclipse plugin alongside the *Parallel Tools Platform (PTP)* (PTP [9]), a rapidly evolving tool integration framework for parallel program analysis. In fact, GEM is being released along with PTP Version 3.0 – the latest PTP release that is imminent. Given the growing use of PTP all over the world, we believe that ISP and GEM will help bring dynamic formal verification for MPI to every designer.

The rest of this section presents sufficient research background to appreciate our contributions. § 2 describes GEM in detail. § 3 provides details of how GEM handles a real-world verification task. § 4 describes our conclusions and our future plans.

1.1 Background on ISP

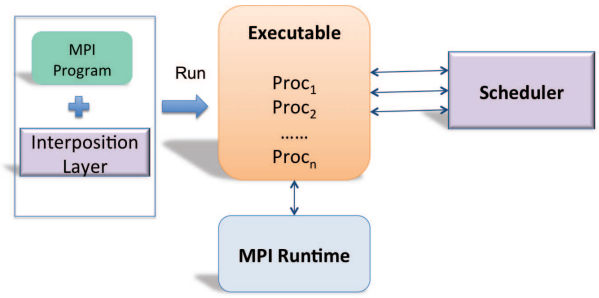

Figure 1. Overview of ISP

There are many excellent MPI program debuggers, for instance TotalView [10], Umpire [11], Marmot [12], and Jitterbug [13]. Two unique features set ISP apart from all these tools: the ability to *determining relevant interleavings*, and the abilitity to *enforce interleavings*. For an illustration of these concepts, consider the example in Figure 2 (for brevity, we do not show the *Wait* calls associated with the non-blocking *Isend* and *Irecv* calls). Considering the overall magnitude of the verification problem, we believe that a verification tool must not spend effort varying the order in which the constituent *Barrier* calls of a matching set of MPI barriers are issued to the MPI runtime. Likewise, unless the MPI library itself is in error, there is nothing much to be gained by posting deterministic sends and receives in different orders (there are millions of such calls issued in an MPI program). As far as we know, none of the alternative tools exploit these options. ISP's focus is away from such permutations of deterministic matches, and toward discovering the maximal degree of non-determinism (*i.e.*, discovering relevant interleavings).

For further illustration of these ideas, consider Figure 2 again. As shown, matching P_2's *Isend* with P_1's *Irecv* leads to 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

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

matches before that we introduced in [4], 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 twice over: once where P_0's *Isend* is considered, and the second time where P_2's *Isend* 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 *Isend* 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).

It is important to emphasize that while the internal issue order computed by ISP appears to be an extremely skewed schedule, it can actually occur on an MPI platform. Even though ISP executes the given MPI program on a specific machine using a specific MPI library, it forces this skewed schedule to occur by delaying non-deterministic non-blocking operations. For example, by delaying *Irecv*, ISP is able to discover the match with respect to the *Isend* of P_2. *The possibility of considering P_0's Isend is not lost by so delaying.* In this way, ISP can verify a program for portability even though it is running the program on a specific platform where the natural schedule would perhaps always prefer P_0's *Isend*. ISP's ability to maximize the latent non-determinism at run time and then verifying over all the possibilities gives it the ability to issue verification guarantees.

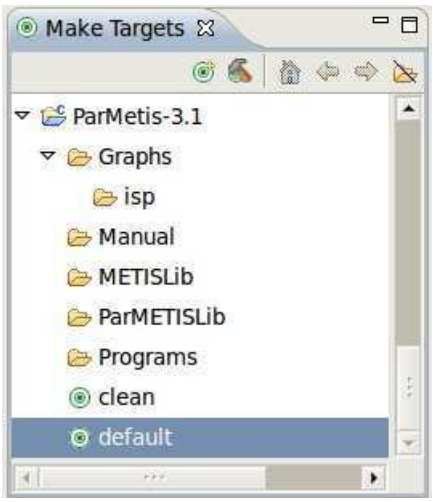

Figure 3. CDT Make Targets View

2 Highlights of GEM

We begin with the design philosophies of GEM followed by a description of its views. GEM is designed to accommodate MPI programmers with different levels of training. As one example, even though ISP internally carries out dynamic reordering and instruction rewriting, GEM has the ability to present verification results as if the matches happened according to program order. This view is ideally suited for new MPI programmers. However, most expert MPI programmers wish to see what a tool does internally (to debug inexplicable behaviors). We therefore also provide the ability to view instructions in the *internal execution order*. Figure 5 clearly shows this ability. GEM also strongly adheres to many of the conventions set forth by the Eclipse foundation. This will help GEM serve as a cockpit from which a designer can seamlessly invoke performance measurement tools (that are being integrated into PTP) and correctness tools (ISP being our focus). We also provide the flexibility of using either CDT (Eclipse C/C++ Development Tools) Managed Build or Makefile projects as shown in Figure 3.

Finally, we provide an extensive help contribution with GEM. We now describe the external view of GEM (§ 2.1) and its internal architecture (§ 2.2).

2.1 GEM: External View

Basic Operation: Given a collection of files to analyze using ISP, GEM helps compile and links the files against the ISP profiler, and then invokes ISP's scheduler on the executable creating a log file containing post-verification results. GEM then parses the log file and organizes its contents. It then attempts to associate MPI calls with one an-

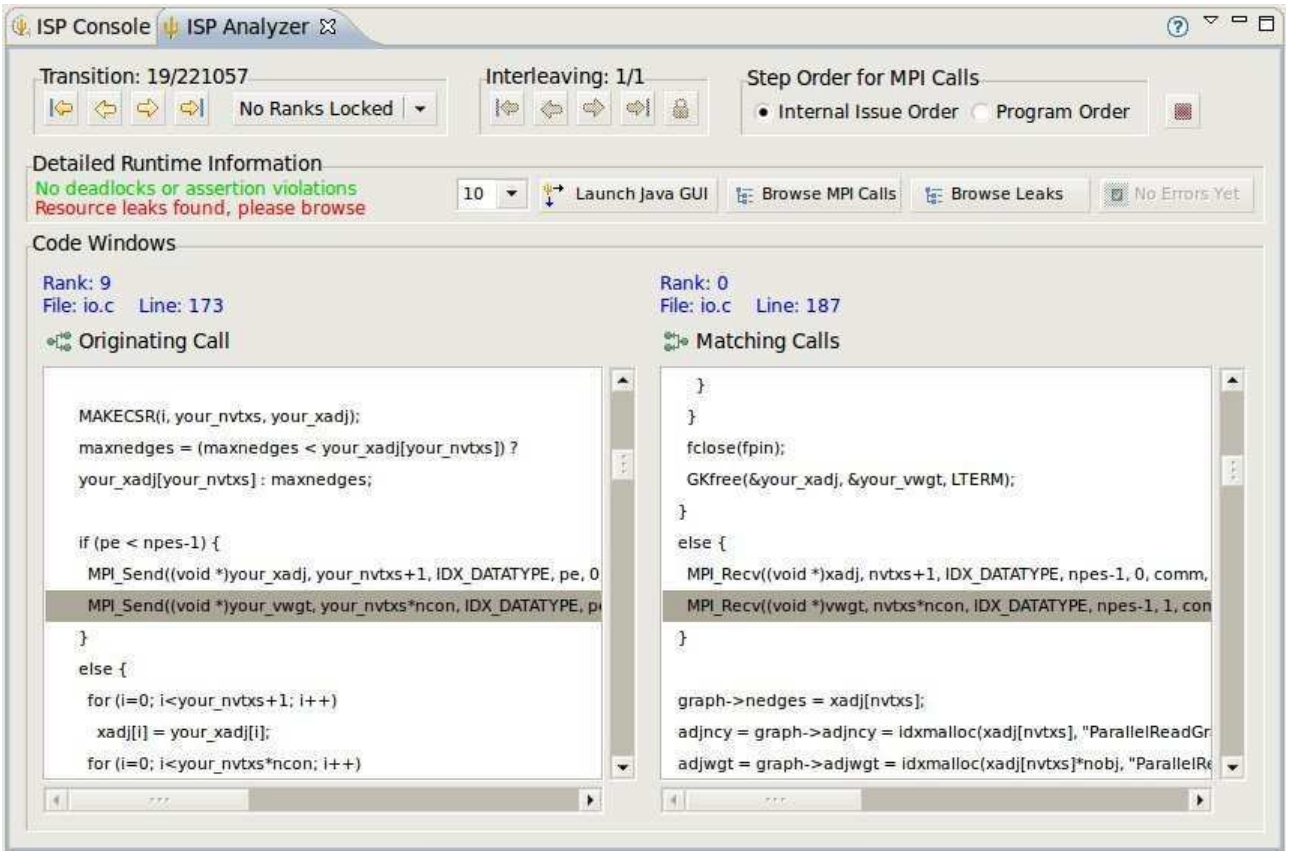

Figure 5. Analyzer View on ParMETIS

other (*e.g.*, sends need to be associated with their corresponding receives). Any call that fails to associate in this manner is flagged as a deadlock. As shown in Figure 4 GEM includes a valuable ability to localize errors by allowing users to step through and display the states of processes involved in the error. As mentioned earlier, GEM also allows users to view the execution results according to the program order or according to ISP's internal execution order. GEM displays MPI point to point operations by listing the send and the receive actions in separate windows. Collective operations such as barriers and reduction operations are listed showing detailed information on one of the calls in one window and listing the remaining calls in summary form in another window.

GEM Views: In addition to the usual textual console view, GEM also provides an analyzer view that serves three functions: (i) summarize verification results, (ii) link to the matches-before viewer, and (iii) allow the user to step through matching MPI calls. Figure 5 depicts the analyzer view obtained by running a 10-process version of ParMETIS through GEM, clearly showing these facts: (i) that 221,057 MPI calls were processed, (ii) that the nineteenth transition is an *MPI_Send* and its matching *MPI_Recv* which are shown along with information on which files they occur in, and most interestingly (iii) that a resource leak was found. At this point, a user can click on the button "Browse Leaks" to obtain a GUI display indicating which exact source line contains the leak. Notice also the radio buttons *Step Order for MPI Calls* offering two options: *Internal Issue Order* and *Program Order*. The *No Ranks Locked* is another option (borrowed from [8]) which shows whether the user is in the mode of stepping through one process (rank) or whether the stepping encompasses all processes. The analyzer view indicates whether a deadlock, assertion violation, or resource leak was found. The button Browse Leaks lists all leaks and opens an editor to help investigate the leak further. Currently, ISP keeps track of MPI object leaks (communicators, type objects, and requests). Future versions of ISP/GEM will also instrument C *mallocs* and track their corresponding *free* operations.

Matches-Before Viewer: Happens-before is a distributed system concept introduced by Lamport in [14] to keep track of time in a distributed system on the basis of event *causalities*. In MPI programs, the salient 'happenings' are message matches; for this reason, we call this relation matches-before. A formal definition of the matches-before relation

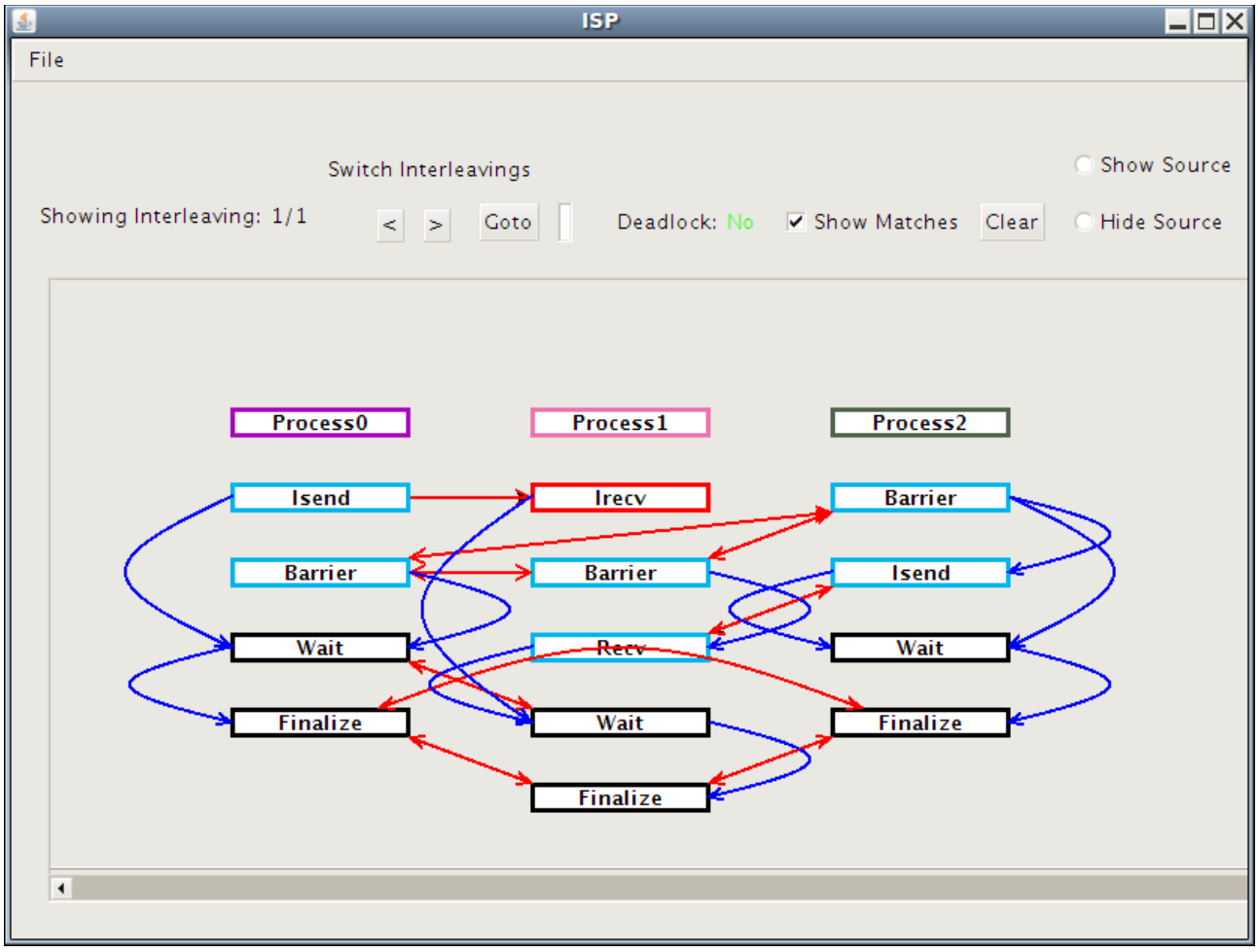

Figure 6. Matches-Before Viewer of ISP

for MPI was presented for the first time in [4]. The paper [6] summarized how ISP's 'Java GUI' (as it was called then) presented this relation. GEM incorporates this unique view as its happens-before viewer facility presented in Figure 6. In this example, we can see that under *Process*0, we have an *Isend* followed by a *Barrier* MPI call. However, there is no arrow between this *Isend* and the *Barrier* – thus clearly showing that these MPI commands may match out of program order (as was explained in § 1.1). The other details included in this view are the following: (i) which commands are non-deterministic (in this example, *Process*1's *Irecv* can match with both *Isend*s, and hence this *Irecv* is colored red, and (ii) all possible matches. In summary, the matches-before viewer informs MPI programmers how their code can execute on any platform compliant MPI runtime.

2.2 GEM: Internal Details

We now describe how GEM was architected. The first thing to keep in mind is that Eclipse is not a single tool with a few small add ons, but rather a small kernel with a collection of extension points, or places to tie into and extend the architecture. These extension points all differ in purpose but all share a common interface. Described succinctly, Eclipse is an extensible platform essentially consisting of three layers. (i) *Eclipse Platform* which offers common programming-language-neutral infrastructure; (ii) *Java Development Tools (JDT)*, which adds a rich, full-featured Java IDE to the Eclipse Platform; and (iii) *Plug-In Development Environment (PDE)* which extends the JDT with plug-in development support. The Eclipse platform itself consists of several components separated into two primary categories: (i) *Core*, which is a runtime component that defines plug-in infrastructure, and provides a workspace to manage projects; (ii) *User Interface (UI)* that provides a *Workbench*

to define the Eclipse UI (e.g. editors, views, perspectives), the *Standard Widget Toolkit* (SWT) to provide the graphics and a set of widgets for UI design with layout strategies to group collections of widgets, and *JFace* which is a UI framework built on top of SWT to help manage images and fonts and to provide more complex viewer objects. The creation of GEM and its help plug-in relies upon the following Eclipse extension points:

- Popup Menus: org.eclipse.ui.popupMenus
- Toolbar Buttons (menus): org.eclipse.ui.menus
- Commands: org.eclipse.ui.commands
- Handlers: org.eclipse.ui.handlers
- Key Bindings: org.eclipse.ui.bindings
- Views: org.eclipse.ui.views
- Preferences: org.eclipse.core.runtime.preferences
- Preference Pages: org.eclipse.ui.preferencePages
- Help: org.eclipse.help.toc

With an initial intention of donating GEM to the Eclipse Parallel Tools Platform (PTP), we used PTP-specific icons for our graphical resources. For the help plug-in, we used PTP style sheets. As GEM will now be part of the 3.0 release of PTP, both plug-ins are bundled into a feature product which allows distribution with source code and license. All strings have been externalized for internationalization. Our hope in distributing our work along with source code under the Eclipse Public License is that the community would be able to contribute to and extend GEM in the future.

3 Verifying ParMETIS using GEM

ParMETIS 3.1 [15] is a parallel graph partitioning and sparse matrix ordering library that finds wide use. Verifying ParMETIS makes for an excellent study due to the inherent complexities involved with verifying and analyzing the runtime results of a project of such size. Some routines provided by ParMETIS have more than 12,000 lines of code between themselves and their helper functions, and involve an enormous number of MPI calls. In past tests of ISP [5] without GEM, the number of MPI calls recorded by the ISP scheduler exceeded 1.3 million.

Figure 4. Deadlock Display by GEM

The test machine used for this particular case study using GEM was an HP Pavilion laptop running Ubuntu 8.10, with 4GB RAM and an Intel Core2Duo T-9300 CPU running at 2.5 GHz. To get a feel for the runtime complexities and realistic range of use for GEM, we began verifying with two processes and gradually increased this number. At 10 processes, we found GEM to take 10 minutes for a verification run. A 32 process verification of ParMETIS took 40 minutes and generated a log file that was 512MB. We feel that beyond 10-12 processes, verifying a project of this size is perhaps best suited for a cluster.

The Makefile support provided by GEM calls for only a few small modifications to the ParMETIS Makefiles (Figure 3). Once the ParMETIS project builds correctly and the ISP profiled executable is produced, we can access dynamic formal verification using GEM essentially the same way we would for any CDT Managed Build project. The only difference will be that we access the executable from context menus via the Project Explorer View instead of the toolbarbar icon.

Thanks to ISP's scheduling algorithm, only *one* schedule was explored. All of our tests verified that the ParMETIS code was free from deadlocks and local assertion violations. However, our tests using GEM discovered a communicator leak in the ParMETIS code, as already discussed (Figure 5). These types of results are instantly recognizable within the GEM Analyzer view. This particular result is further proof of the effectiveness of graphical debugging tools for parallel application development. With one click in the shell window provided by GEM, the user can navigate to the source line where the communicator was allocated.

4 Conclusions and Future Plans

In this paper, we summarized how the usability of our dynamic verifier for MPI programs, namely ISP, has been vastly enhanced by the design of the Graphical Explorer of Message Passing (GEM). Several interactive tutorials have been offered using the ISP/GEM combination (the most recent being a 15-minute slot in the IBM PTP tutorial during Supercomputing 2009). We found that the availability of GEM made what was a powerful but intimidating tool (namely ISP) into a pleasant, intuitive, productivity enhancing tool.

A number of avenues of further research remain. First, we are working on a number of approaches to scale up ISP's

search algorithms. Second, we are in the process of adding many more default checks into ISP, and correspondingly enhancing the error viewing facilities in GEM. Third, we plan to instrument the salient aspects of the C code that lie between MPI calls (at present, these C codes are simply executed without any scheduler interception). We find that given the importance of mixed programming, we will run into the problem of deterministic replay should this C space behavior harbor thread non-determinism and/or races. Once we are able to instrument and replay the thread-space behaviors, we plan to enhance GEM's display capabilities to include these behaviors as well.

Acknowledgements: We gratefully acknowledge the work done by the creators of ISP, notably Sarvani Vakkalanka, Anh Vo, and Michael DeLisi. We also acknowledge the contributions of Sriram Aananthakrishnan who created the matches-before viewer of ISP. Thanks also to Greg Watson of IBM for encouraging us to contribute GEM to Eclipse.

References

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

[2] Test results comparing isp, marmot, and mpirun. `http://www.cs.utah.edu/fv/ISP_Tests`.

[3] Stephen F. Siegel. Verifying parallel programs with MPI-SPIN. In Franck Cappello, Thomas Hrault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer, 2007.

[4] Sarvani Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, and Robert M. Kirby. Reduced execution semantics of mpi: From theory to practice. In *FM 2009*, pages 724–740, November 2009.

[5] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, , and Rajeev Thakur. Formal verification of practical mpi programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 261–269, 2009.

[6] Sriram Aananthakrishnan, Michael Delisi, Sarvani S. Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. How formal dynamic verification tools facilitate novel concurrency visualizations. In *EuroPVM/MPI*, pages 261–270, 2009.

[7] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In *Computer Aided Verification (CAV 2008)*, pages 66–79, 2008.

[8] Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Scheduling considerations for building dynamic verification tools for MPI. In *Parallel and Distributed Systems - Testing and Debugging (PADTAD-VI)*, Seattle, WA, July 2008.

[9] The Eclipse Parallel Tools Platform. `http://www.eclipse.org/ptp`.

[10] TotalView concurrency tool. `http://www.totalviewtech.com`.

[11] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*. IEEE Computer Society, 2000. Article 51.

[12] Bettina Krammer, Katrin Bidmon, Matthias S. Mller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In *Parallel Computing 2003*, September 2003.

[13] Richard Vuduc, Martin Schulz, Dan Quinlan, Bronis de Supinski, and Andreas Sæbjörnsen. 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.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[15] ParMETIS - Parallel graph partitioning and fill-reducing matrix ordering. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.