GEM: Graphical Explorer of MPI Programs

Alan Humphrey,
Christopher Derrick,
and Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112
Email: {ahumphre,cderrick,ganesh}@cs.utah.edu

Beth Tibbitts
IBM Corporation
Eclipse Parallel Tools Platform
Lexington, KY 40511
Email: tibbitts@us.ibm.com

***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 In-situ Partial Order (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 now present a new tool called Graphical Explorer of MPI Programs (GEM) that overcomes this limitation. GEM is a plug-in architecture that greatly enhances the usability of ISP, and serves to bring ISP within reach of a wide array of programmers with its original release as part of the Eclipse Foundation's Parallel Tools Platform (PTP) Version 3.0 in December, 2009. GEM is now a part of the PTP End-User Runtime. 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. Here, we also describe the process and benefits of using GEM throughout the development cycle of our own test case, an MPI implementation of the A* search. We conclude with a summary of our future plans.**

***Keywords*-Dynamic Verification, Graphical User Interfaces, Dynamic Interleaving Reduction, Message Passing, MPI, Multi-core, Eclipse Parallel Tools Platform.**

I. 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 inter-process 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).

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] (summarized in § I-A) that incorporates the algorithms first introduced in [7, 8] is currently the only tool that can analyze large MPI programs while avoiding redundant interleavings and requiring no model building. While ISP was a significant step forward in the arena of dynamic verification tools for MPI, its usage was hindered by the absence of a widely usable and intuitive graphical user interface. This paper describes our contribution in this regard through a new tool called Graphical Explorer of MPI Programs (GEM). GEM is designed to serve as an Eclipse plug-in alongside the Parallel Tools Platform (PTP [9]), a rapidly evolving tool integration framework for parallel program development and analysis. GEM was initially released with PTP Version 3.0, and is now a part of the PTP End-User Runtime, included in successive PTP releases. The upcoming 4.0 version of PTP, scheduled to be released in June, 2010, will coincide with the release of Eclipse Helios.

GEM was initially created to provide a graphical interface for ISP via an integration within Eclipse, similar to the integration of ISP within Visual Studio [10]. This previous integration was not sufficient because Visual Studio runs on the proprietary Windows platforms, whereas the HPC community often prefers working with non-commercial open-source software; these are significant advantages GEM and PTP provide. GEM also offers many more features than our work in [10]. Given the growing use of PTP all over the world, we believe

Supported in part by Microsoft, NSF CNS-0509379, CCF-0811429, CCF-0903408. Portions of this material are supported by or based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under its Agreement No. HR0011-07-9-0002, the United States Department of Energy under Contract No. DE-FG02-06ER25752.

that ISP and GEM will help bring dynamic formal verification for MPI to every developer.

The rest of this section presents sufficient research background to appreciate our contributions. § II describes GEM in detail. § III provides details of how GEM handles a real-world verification task. § IV summarizes usage of GEM throughout the development cycle of our own MPI C application. § V describes our conclusions and our future plans.

A. Background on ISP

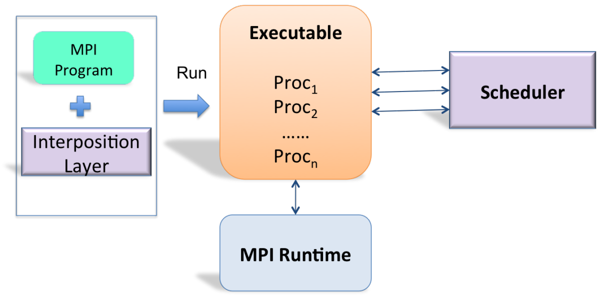


Fig. 1. Overview of ISP

There are many excellent MPI program debuggers, for instance TotalView [11], Umpire [12], Marmot [13], and Jitterbug [14]. Two unique features set ISP apart from all these tools: the ability to *determine relevant interleavings* (possible schedules with runtime behavior different from those already observed), and the ability 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 *matches before* that we introduced in [4], ISP can exert this fine degree of execution control. ordering rule

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

Fig. 2. MPI Example

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 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.

II. Highlights of GEM

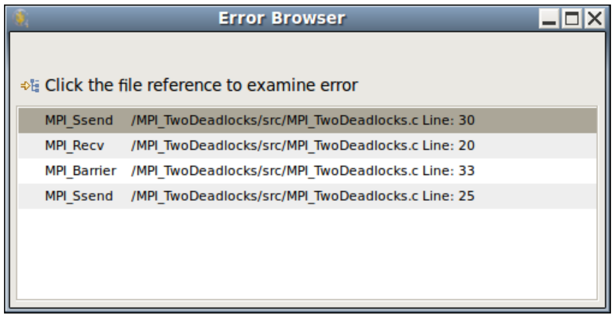
We begin with a description of what GEM offers followed by some of its design philosophies. GEM offers support for both Eclipse C/C++ Development Tools (CDT) Managed Build and Makefile projects, and is designed to accommodate

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 is ideally suited for new MPI programmers who will find it easy to follow the flow of the calls as they walk through their code. Most expert MPI programmers however, 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 4) clearly shows this ability. To make the Eclipse integration as seamless and effective as possible GEM also strongly adheres to the design conventions set forth by the Eclipse foundation, making it easily maintainable and extensible. GEM, combined with the plethora of parallel development tools PTP offers (e.g., scalable, parallel debugger and MPI job launch facilities), a developer can write, debug, formally verify and launch their MPI code on a cluster all from within the Eclipse IDE. More detail on PTP can be found at [9, 15].

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

A. GEM: External View

Basic Operation: Given a collection of files to analyze using ISP, GEM helps compile and link the files against the ISP profiler library (the interposition layer), and then invokes ISP's scheduler on the executable which creates a log file containing post-verification results. GEM then parses the log file and organizes its contents into efficient internal data structures. It then attempts to associate MPI calls with one another (*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. GEM includes a valuable ability to localize errors by allowing users to step through and display the state of processes involved in the error. When an error is flagged, the user is notified of the problem within the GEM Analyzer View and presented with the option to view the relevant source code within the Eclipse editor as shown in Figure 3.



Click the file reference to examine error

```
MPI_Ssend    /MPI_TwoDeadlocks/src/MPI_TwoDeadlocks.c Line: 30
MPI_Recv     /MPI_TwoDeadlocks/src/MPI_TwoDeadlocks.c Line: 20
MPI_Barrier  /MPI_TwoDeadlocks/src/MPI_TwoDeadlocks.c Line: 33
MPI_Ssend    /MPI_TwoDeadlocks/src/MPI_TwoDeadlocks.c Line: 25
```

Fig. 3. Deadlock Display by GEM

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 (Figure 4).

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. The entire collection of features described above and full Eclipse integration of GEM are shown in Figure 5.

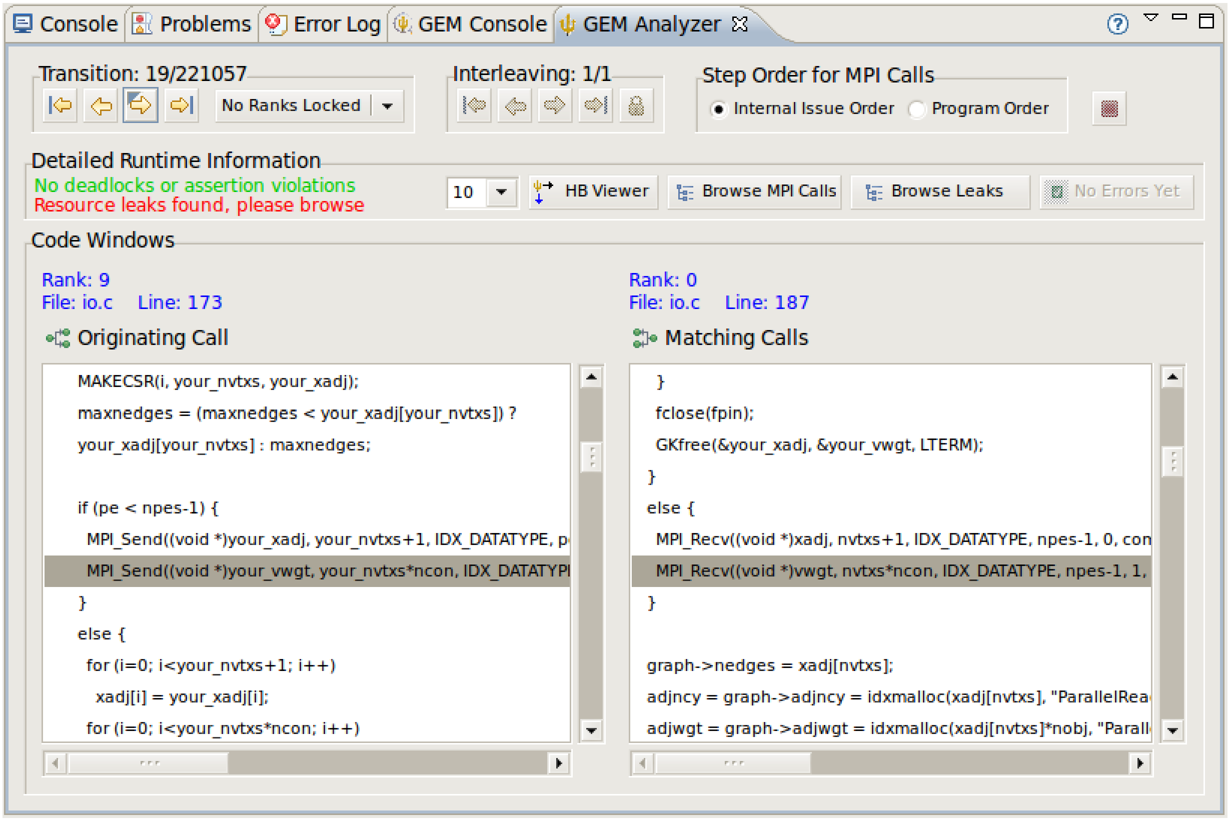
GEM Views: Each particular Eclipse development environment (e.g., C/C++, Java, Python) has its own perspective made up of a collection of individual Eclipse views (e.g., source editor, console). In addition to its own dedicated textual console view, GEM also provides the GEM Analyzer View, which serves four functions: (i) summarize verification results, (ii) help localize errors, (iii) allow the user to step through matching MPI calls, and (iv) link to the happens-before viewer.

Figure 4 depicts the GEM 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* on line 24 matching an *MPI_Recv* on line 18, and most interestingly (iii) that a resource leak was found.

At this point, from within the GEM Analyzer View, a user can click on the button "Browse Leaks" to obtain a shell window display indicating the source line containing the leak (an allocated but unfrequented MPI object). Clicking on a listed error takes the user to that exact line of source code within the Eclipse editor. The GEM Analyzer View clearly indicates whether a deadlock, assertion violation, or resource leak was found after a verification run (Figure 5). Future versions of ISP/GEM will also instrument C *malloc*s and track their corresponding $free$ operations.

Notice also the radio buttons *Step Order for MPI Call* offering two options: *Internal Issue Order* and *Program Order*. The *Rank Lock* feature is another option (borrowed from [10]) which shows whether the user is in the mode of stepping through MPI calls associated with one process (rank) or whether the stepping encompasses all processes.

Happens-Before Viewer: Happens-before is a distributed system concept introduced by Lamport in [16] 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 happens-before. A formal definition of the happens-before relation for MPI was presented for the first time in [4]. Understanding this relation has led to the creation of the happens-before-viewer which graphically shows how the various calls are related to each other. The paper [5] summarized how ISP's 'Java GUI' (as it was called then) presented this relation. In an effort to provide users with as many tools as possible GEM gives users the ability to view their current project through the happens-before viewer by clicking the button "HB Viewer" from the GEM Analyzer View (Figure 5).



[● Internal Issue Order] ○ Program Order

No deadlocks or assertion violations
Resource leaks found, please browse

Rank: 9
File: io.c   Line: 173

```
MAKECSR(i, your_nvtxs, your_xadj);
maxnedges = (maxnedges < your_xadj[your_nvtxs]) ?
your_xadj[your_nvtxs] : maxnedges;

if (pe < npes-1) {
  MPI_Send((void *)your_xadj, your_nvtxs+1, IDX_DATATYPE, p
  MPI_Send((void *)your_vwgt, your_nvtxs*ncon, IDX_DATATYPI
}
else {
  for (i=0; i<your_nvtxs+1; i++)
    xadj[i] = your_xadj[i];
  for (i=0; i<your_nvtxs*ncon; i++)
```

Rank: 0
File: io.c   Line: 187

```
  }
  fclose(fpin);
  GKfree(&your_xadj, &your_vwgt, LTERM);
}
else {
  MPI_Recv((void *)xadj, nvtxs+1, IDX_DATATYPE, npes-1, 0, con
  MPI_Recv((void *)vwgt, nvtxs*ncon, IDX_DATATYPE, npes-1, 1,
}

graph->nedges = xadj[nvtxs];
adjncy = graph->adjncy = idxmalloc(xadj[nvtxs], "ParallelRea
adjwgt = graph->adjwgt = idxmalloc(xadj[nvtxs]*nobj, "Parall
```

Fig. 4. GEM Analyzer View on ParMETIS

B. GEM: Internal Details

We now describe how GEM was architected and integrated within Eclipse. 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 share a common interface.

Described succinctly, the Eclipse Plugin Development Environment we used supports 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), (iii) *Standard Widget Toolkit* (SWT) offers the graphics and a set of widgets for UI design with layout strategies. (iv) *JFace* is a UI framework built on top of SWT to help manage images and fonts and to provide more complex viewer objects.

All Eclipse plug-ins are represented by a single instance of a plug-in class which extends AbstractUIPlugin. In our case, *GemPlugin* is the activator class for GEM, and is the source for shared information such as preferences, icons and images used. This class controls the life-cycle of GEM.

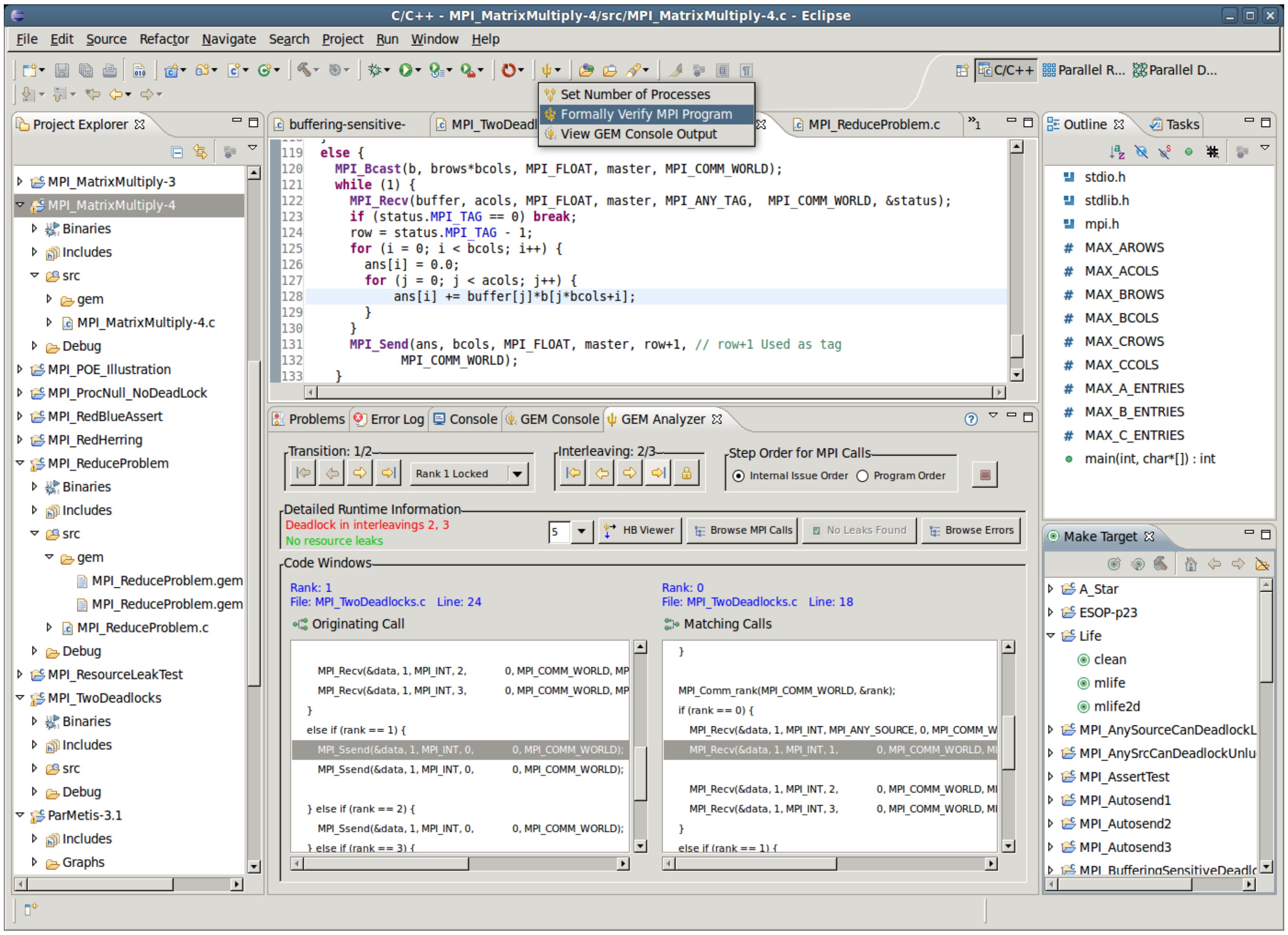
Basic Eclipse plug-in architecture uses a model that separates declaration from implementation. The shape of a contribution is declared with XML in a standard file(*plugin.xml*), and the implementation is in Java.

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

In contributing to these Eclipse extension points GEM has adhered to all expected interfaces. Actions defined by GEM are backed by classes implementing the IWorkBenchWindowActionDelegate interface. Commands defined by GEM are backed by classes which extend AbstractHandler. These classes all implement the behavior of a particular action or command defined in the XML descriptor file *plugin.xml*.

GEM is centered around the GEM Analyzer View, a collection of MPI runtime information and tools to debug and



```
  else {
    MPI_Bcast(b, brows*bcols, MPI_FLOAT, master, MPI_COMM_WORLD);
    while (1) {
      MPI_Recv(buffer, acols, MPI_FLOAT, master, MPI_ANY_TAG,  MPI_COMM_WORLD, &status);
      if (status.MPI_TAG == 0) break;
      row = status.MPI_TAG - 1;
      for (i = 0; i < bcols; i++) {
        ans[i] = 0.0;
        for (j = 0; j < acols; j++) {
            ans[i] += buffer[j]*b[j*bcols+i];
        }
      }
      MPI_Send(ans, bcols, MPI_FLOAT, master, row+1, // row+1 Used as tag
               MPI_COMM_WORLD);
  }
```

Deadlock in interleavings 2, 3
No resource leaks

Rank: 1
File: MPI_TwoDeadlocks.c   Line: 24

```
    MPI_Recv(&data, 1, MPI_INT, 2,         0, MPI_COMM_WORLD, MP
    MPI_Recv(&data, 1, MPI_INT, 3,         0, MPI_COMM_WORLD, MP
  }
  else if (rank == 1) {
    MPI_Ssend(&data, 1, MPI_INT, 0,         0, MPI_COMM_WORLD);
    MPI_Ssend(&data, 1, MPI_INT, 0,         0, MPI_COMM_WORLD);

  } else if (rank == 2) {
    MPI_Ssend(&data, 1, MPI_INT, 0,         0, MPI_COMM_WORLD);
  } else if (rank == 3) {
```

Rank: 0
File: MPI_TwoDeadlocks.c   Line: 18

```
  }

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_W
    MPI_Recv(&data, 1, MPI_INT, 1,          0, MPI_COMM_WORLD, MI

    MPI_Recv(&data, 1, MPI_INT, 2,          0, MPI_COMM_WORLD, MI
    MPI_Recv(&data, 1, MPI_INT, 3,          0, MPI_COMM_WORLD, MI
  }
  else if (rank == 1) {
```

Fig. 5.   GEM's Seamless Eclipse Integration

examine this information. To effectively create push-button dynamic formal verification and intuitive visual MPI runtime analysis, we've made heavy use of SWT and JFace for the UI component and rely on many cleverly crafted supporting classes, including a dedicated console (another view contributed by GEM) to accompany the GEM Analyzer View.

SWT provides a common set of widgets (buttons and menus), and strategies for layout and grouping of widgets. JFace provides viewer objects to display more complicated objects, as well as a mechanism for SWT widgets to respond to events such as selection or double clicks by registering a listener to a particular widget. It is essentially a thin layer on top of the underlying operating system's native widget set without any dependencies on Eclipse itself. All our views were created with SWT using FormLayout and GridLayout and Composites for grouping.

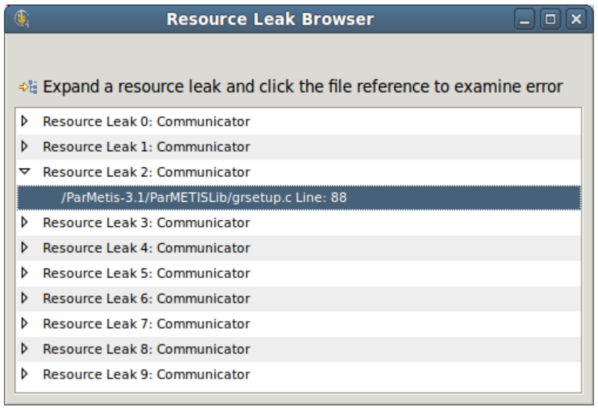
With an initial intention of donating GEM to the Eclipse Parallel Tools Platform (PTP), we used PTP-specific icons for our graphical resources. As GEM is now part of PTP, both plug-ins are bundled into a feature product which allows distribution with source code and license. All strings have also been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

III. Verifying ParMETIS using GEM

ParMETIS 3.1 [17] 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 [7] without GEM, the number of MPI calls recorded by the ISP scheduler exceeded 1.3 million.

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 an entire verification run to take


Resource Leak Browser

Expand a resource leak and click the file reference to examine error

```
Resource Leak 0: Communicator
Resource Leak 1: Communicator
Resource Leak 2: Communicator
    /ParMetis-3.1/ParMETISLib/grsetup.c Line: 88
Resource Leak 3: Communicator
Resource Leak 4: Communicator
Resource Leak 5: Communicator
Resource Leak 6: Communicator
Resource Leak 7: Communicator
Resource Leak 8: Communicator
Resource Leak 9: Communicator
```

Fig. 6. ParMETIS Communicator Leak

approximately 10 minutes. 9.75 minutes were required for ISP to do the verification and 15 seconds for GEM to parse the log file, load internal data structures and display the results. A 32 process verification of ParMETIS took 40 minutes and generated a log file that was 512MB.

With the C/C++ makefile support offered by GEM, only a few small modifications to the ParMETIS makefiles were required to verify it. Once the ParMETIS project built correctly and the ISP-profiled executable was produced, dynamic formal verification is accessed essentially the same way it is in GEM for any CDT Managed Build project. The only difference will be that we access the executable from context menus via the Eclipse Project Explorer View instead of the toolbar 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 discussed earlier (Figure 6). 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.

IV. GEM IN THE DEVELOPMENT CYCLE

To test the effectiveness of GEM and its ease of use throughout the development cycle of an MPI application, we created a parallel implementation of the A* search algorithm in MPI. The general idea behind our implementation is to have one process explore the maze long enough that the number of nodes in the priority queue is greater than or equal to the total number of processes involved. At this point each process receives a node and begins to work more or less independently trying to solve the maze starting from that node. Each process regularly informs the others what it has achieved so as to reduce unnecessary duplication of work. When a process realizes that it is impossible to solve the maze without using nodes that others have already reached in less steps it abandons its work and receives a new node from the root process to work on. This continues until one of the nodes successfully completes the maze at which point all processes know how many steps the maze was solved in and continue to work until they know they cannot solve the maze in less steps. At this point the optimal path has been found and is reported to the user.

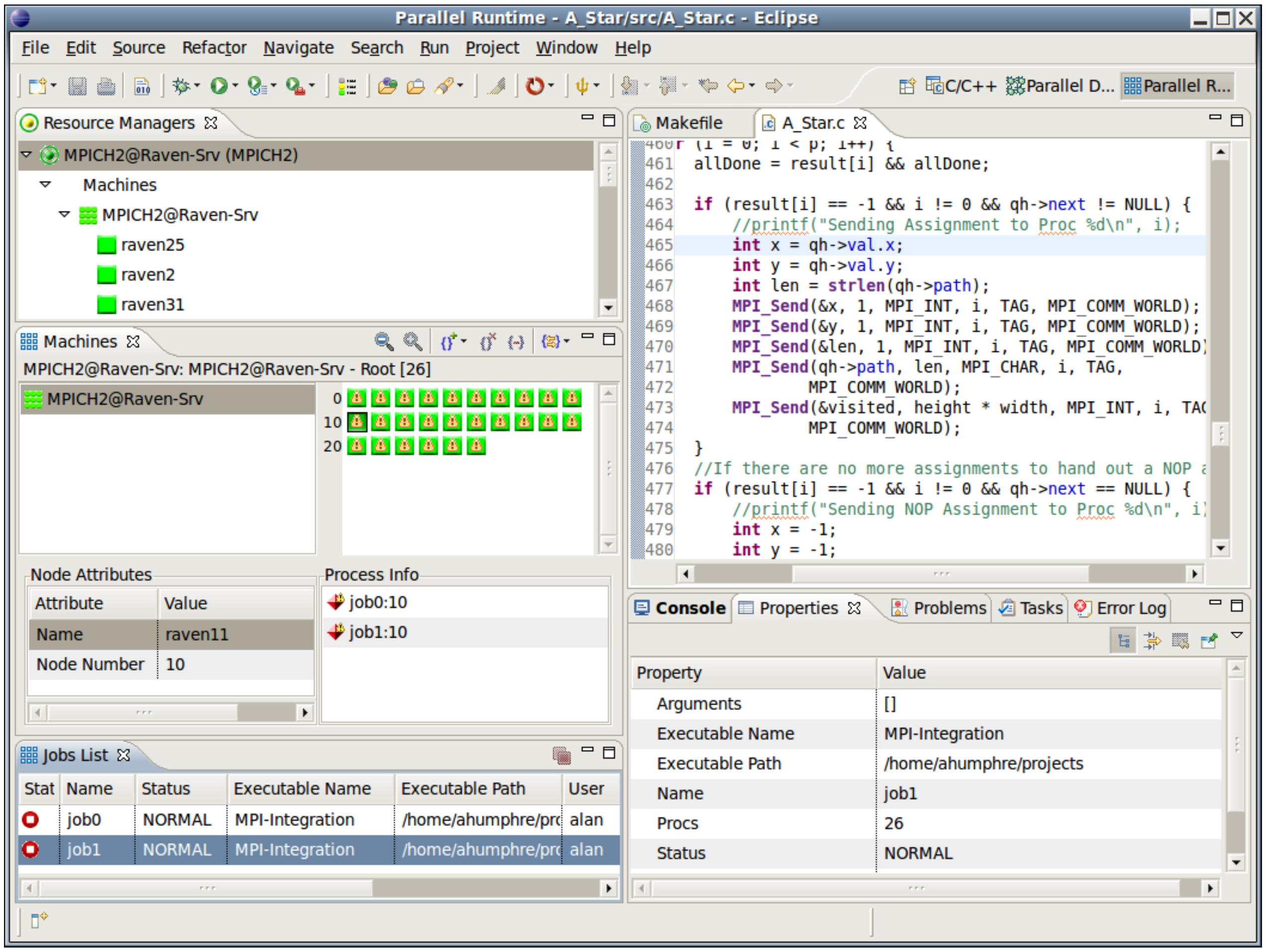
Our method of development was to incrementally add more functionality, and to test each addition by running the integrated push-button dynamic formal verification GEM offers and to graphically examine program behavior and MPI runtime characteristics. By using GEM in this way within the Eclipse development environment, our development and debugging time of the A* application was significantly reduced. In particular there were a number of deadlocks that were quickly uncovered and fixed thanks to the aid of GEM's intuitive runtime analysis reports and graphical displays.

Without the aid of GEM, it takes a great deal of time to locate the exact position of a deadlock and understand its cause, but with GEM a user is shown exactly where to find the deadlock without any effort on their part and with the aid of the runtime information provided it usually only takes a few moments to understand its cause. Here are some of the errors GEM aided us in correcting:

- Where root was receiving the search paths found by each process, we had copied the receive from another line of code and forgotten to change the sender. This led to a situation where root was trying to receive from root. When we ran the program we got a deadlock. GEM clearly reported which line this error occurred on.
- We altered the code and unintentionally created a situation where some of the processes never reached a collective call. Again running the code simply resulted in a deadlock and GEM told us that the error was a collective call which was not reached by certain processes and directed us to the exact line of that call. From there we made a simple change to the logic and the problem was fixed.
- We needed to send out assignments from the root process to all other processes and used a for loop to do so. Out of habit, we had the for loop start from zero. After seeing a deadlock we ran GEM and saw that the code was trying to send from process zero to process zero (since process zero was the root process). Once we realized the mistake we changed the for loop to start at one instead of zero.

Each of these errors was quickly solved thanks to features GEM provides, without which considerable time and effort would have been needed to track down the exact location and cause. Additionally GEM helped us discover, with the help of its optional irrelevant barrier detection, that one of the barriers we had placed in the code was not actually needed and could be removed.

As we had hoped, the powerful array of intuitive, graphical displays and tools GEM offers, combined with its seamless integration within Eclipse allowed convenient dynamic MPI formal verification throughout the development cycle or our MPI C application.


Parallel Runtime - A_Star/src/A_Star.c - Eclipse
File Edit Source Refactor Navigate Search Run Project Window Help
C/C++ Parallel D... Parallel R...
Resource Managers
MPICH2@Raven-Srv (MPICH2)
Machines
MPICH2@Raven-Srv
raven25
raven2
raven31
Machines
MPICH2@Raven-Srv: MPICH2@Raven-Srv - Root [26]
MPICH2@Raven-Srv
0
10
20
Makefile
A_Star.c
Console Properties Problems Tasks Error Log
Jobs List

```
for (i = 0; i < p; i++) {
allDone = result[i] && allDone;

if (result[i] == -1 && i != 0 && qh->next != NULL) {
    //printf("Sending Assignment to Proc %d\n", i);
    int x = qh->val.x;
    int y = qh->val.y;
    int len = strlen(qh->path);
    MPI_Send(&x, 1, MPI_INT, i, TAG, MPI_COMM_WORLD);
    MPI_Send(&y, 1, MPI_INT, i, TAG, MPI_COMM_WORLD);
    MPI_Send(&len, 1, MPI_INT, i, TAG, MPI_COMM_WORLD)
    MPI_Send(qh->path, len, MPI_CHAR, i, TAG,
            MPI_COMM_WORLD);
    MPI_Send(&visited, height * width, MPI_INT, i, TAC
            MPI_COMM_WORLD);
}
//If there are no more assignments to hand out a NOP a
if (result[i] == -1 && i != 0 && qh->next == NULL) {
    //printf("Sending NOP Assignment to Proc %d\n", i)
    int x = -1;
    int y = -1;
```

Node Attributes

Attribute	Value
Name	raven11
Node Number	10

Process Info

job0:10
job1:10

Stat	Name	Status	Executable Name	Executable Path	User
●	job0	NORMAL	MPI-Integration	/home/ahumphre/prc	alan
●	job1	NORMAL	MPI-Integration	/home/ahumphre/prc	alan

Property	Value
Arguments	[]
Executable Name	MPI-Integration
Executable Path	/home/ahumphre/projects
Name	job1
Procs	26
Status	NORMAL

Fig. 7. Eclipse PTP Parallel Runtime Perspective

We made use of many invaluable tools available in the Eclipse PTP as well. With an easily configured Parallel Launch Configuration and MPICH Resource Manager, we were able to launch the A* application on our remote cluster during the development cycle. Figure 7 illustrates a completed 26-process job submitted from within the PTP Parallel Runtime Perspective. PTP also offers extensive local and remote parallel debug services ([9, 15]).

V. 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 MPI Programs (GEM). Several interactive tutorials have been offered using the ISP/GEM combination [18, 19, 20]. The feedback we have received suggests that GEM has helped ISP become an intuitive productivity enhancing tool.

A number of avenues of further research are being pursued. First, we are working on a number of approaches to scale up ISP's verification 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 C/Fortran source codes that lie between MPI calls. At present, these source codes are simply executed without any scheduler interception, causing certain memory errors to be overlooked. For example, if an MPI non-blocking receive is immediately followed by a computational expression that uses the receive buffer without an intervening MPI wait/test, there would be a potential data race on the receive buffer. Once we instrument these expressions and add suitable facilities for detecting such errors, we plan to add to GEM control-flow views through these erroneous non-MPI source codes also.

Last but not least, it is important to be able to run dynamic verification at scale. Some bugs manifest only when a program is run at scale (*e.g.*, buffer overflows, array indices falling outside of allowed bounds, etc.) Unfortunately, ISP's approach does not allow the required degree of scalability. We have recently developed a scalable dynamic analysis tool called Distributed Analyzer for MPI (DAMPI, [21]) based on new distributed algorithms. We plan to develop a tight integration of DAMPI within GEM, including facilities for invoking DAMPI on a remote cluster (as illustrated in Figure 7) and displaying the verification results. Initial ideas in this regard were recently presented at [22].

Acknowledgements: We gratefully acknowledge the work done by the principal developers of ISP, namely Sarvani Vakkalanka, Anh Vo, and Michael DeLisi. We also acknowledge the contributions of Sriram Aananthakrishnan who created the happens-before viewer of ISP and Subodh Sharma who added the facility to detect functionally irrelevant barriers. Thanks also to Greg Watson of IBM for encouraging us to contribute GEM to Eclipse.

References

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

[2] "Test Results Comparing ISP, Marmot, and mpirun," http://www.cs.utah.edu/fv/ISP_Tests.

[3] S. F. Siegel, "Verifying Parallel Programs with MPI-Spin," in *PVM/MPI User's Group Meeting*, ser. LNCS, vol. 4757, 2007, pp. 13–14.

[4] S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby, "Reduced Execution Semantics of MPI: From Theory to Practice," in *FM*, 2009, pp. 724–740.

[5] S. Aananthakrishnan, M. Delisi, S. S. Vakkalanka, A. Vo, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "How Formal Dynamic Verification Tools Facilitate Novel Concurrency Visualizations," in *PVM/MPI*, ser. LNCS, vol. 5759, 2009, pp. 261–270.

[6] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp, "Implementing Efficient Dynamic Formal Verification Methods for MPI Programs," in *PVM/MPI*, ser. LNCS, vol. 5205, 2008, pp. 248–256.

[7] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Formal Verification of Practical MPI Programs," in *PPoPP*, 2009, pp. 261–269.

[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 *CAV*, 2008, pp. 66–79.

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

[10] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby, "Scheduling Considerations for Building Dynamic Verification Tools for MPI," in *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD), held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, July 20-21 2008, ACM 2008.

[11] "Total View Concurrency Tool," http://www.totalviewtech.com.

[12] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," in *SC*, 2000, article 51.

[13] B. Krammer, K. Bidmon, M. S. Mller, and M. M. Resch, "MARMOT: An MPI Analysis and Checking Tool," in *ParCo*, 2003.

[14] R. Vuduc, M. Schulz, D. Quinlan, B. de Supinski, and A. Sæbjörnsen, "Improving Distributed Memory Applications Testing by Message Perturbation," in *PADTAD*, 2006.

[15] "UPC and OpenMP Parallel Programming and Analysis in PTP with CDT," http://www.eclipsecon.org/2010/sessions/sessions?id=1428.

[16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[17] "ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering," http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[18] "Developing Scientific Applications using Eclipse and the Parallel Tools Platform," http://scyourway.nacse.org/conference/view/tut134.

[19] G. Gopalakrishnan, "Dynamic Verification of Message Passing and Threading," PPoPP Tutorial, 2010.

[20] G. Gopalakrishnan, R. M. Kirby, S. Vakkalanka, and A. Vo, "Practical Formal Verification of MPI and Thread Programs," PVM/MPI Invited full-day Tutorial, 2009.

[21] 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 *Supercomputing (SC)*, 2010, accepted in SC10. http://www.cs.utah.edu/fv/DAMPI/sc10.pdf.

[22] A. Vo, G. Gopalakrishnan, S. Vakkalanka, A. Humphrey, and C. Derrick, "Seamless Integration of Two Approaches to Dynamic Formal Verification of MPI Programs," PLDI Workshop 2010.