How Formal Dynamic Verification Tools Facilitate Novel Concurrency Visualizations *

Sriram Aananthakrishnan[1], Michael DeLisi[1], Sarvani Vakkalanka[1], Anh Vo[1], Ganesh Gopalakrishnan[1], Robert M. Kirby[1], and Rajeev Thakur[2]

[1] School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA
[2] Math. and Comp. Sci. Div., Argonne Nat. Lab., Argonne, IL 60439, USA

Abstract. With the exploding scale of concurrency, presenting valuable pieces of information collected by formal methods tools intuitively and graphically can greatly enhance concurrent system debugging. Traditional MPI program debuggers present trace views of MPI program executions. Such views are redundant, often containing equivalent traces that permute independent MPI calls. In our ISP formal dynamic verifier for MPI programs, we present a collection of alternate views made possible by the use of formal dynamic verification. Some of ISP's views help pinpoint errors, some facilitate discerning errors by eliminating redundancy, while others help understand the program better by displaying concurrent even orderings that must be respected by *all* MPI implementations, in the form of *completes-before* graphs. In this paper, we describe ISP's GUI capabilities in all these areas which are currently supported by a portable Java based GUI, a Microsoft Visual Studio GUI, and an Eclipse based GUI whose development is in progress.

1 Introduction

Program debugging tools and their graphical user interfaces must strive to meet three goals: (i) locate errors in the tool's range reliably, and display the errors intuitively, (ii) eliminate redundant work in locating the errors, and correspondingly avoid presenting redundant work, (iii) depict items of interest so that users gather deep insights about their programs and the APIs/libraries they use. These three goals are even more important to meet for concurrent program debuggers, because concurrent program executions are often far less intuitive than sequential program executions. This paper is on a family of new graphical user interfaces (GUI) that we have equipped our previously reported In-Situ Partial order (ISP, [1–4]) tool with, as our first attempt to meet the above goals. Information is presented by the ISP tool through a portable Java based GUI and an optional Microsoft Visual Studio GUI, and an Eclipse GUI (development in progress) that will integrate and extend these views.

Being a formal verification tool, ISP guarantees to locate deadlocks, C assertion violations, resource leaks, and many invalid uses of MPI calls for a chosen

* Supported in part by NSF CNS-00509379, Microsoft HPC Institutes Program, and the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

application and its test harness. Being a dynamic verification tool, ISP does all its verification by "cleverly" running the MPI application on a standard computer (say, a laptop). The main idea (cleverness) in ISP is in the fact that it explores only *certain* process schedules, after which it proclaims the program to be free from these classes of errors. A concurrent program with five processes, with each process taking part in five sequential steps has potentially 10 billion interleavings (schedules).[3] However, it can be easily seen that large groups of interleavings are totally equivalent: either they do not trigger a bug, or they all do. For example, an interleaving in which an `MPI_send` is posted before its matching non-deterministic `MPI_Recv` is equivalent to the one in which things are posted oppositely (these are *independent actions* [5]). A conventional testing tool does not take advantage of this independence, and may present all the interleavings as traces, when in fact the essence of all interleavings might have been summarized by picking *any one* of the interleavings. By avoiding this exponential interleaving exploration, ISP can both run far more efficiently and also present only the *relevant* interleavings.

ISP determines what is a relevant interleaving at runtime. The main idea can be explained using the example of an MPI program containing one wildcard receive (`P0:: recv(from *)`) and three potentially matching MPI sends (`P1::send(to P0)`, `P2::send(to P0)`, and `P3::send(to P0)`). ISP will, at runtime, (i) determine all MPI sends that can match the wildcard receive (say only the sends from P1 and P2 can match P0's receive), (ii) dynamically rewrite the wildcard receive to the two specific receives, and (iii) replay the entire program for each such match (*i.e.*, issue `P0:: recv(from P1)`, run the whole program, restart all MPI processes, and run again, now issuing `P0:: recv(from P2)`). In this manner, ISP generates relevant alternate interleavings only if non-deterministic actions are present. For programs that are heavily non-deterministic, ISP will explore a large number of interleavings. However, in practice, most MPI programs are deterministic. For ParMETIS, a 14k LOC program, ISP generated exactly one interleaving [2]; the number of possible interleavings is astronomical.

The results produced by any tool (including ISP) cannot be trusted if the underlying MPI library does not conform to the MPI standard, if the user does not model a sufficient number of processes, or the user drives the application with limited ranges of data. More than these well-known restrictions of any tool, there are many important, but often overlooked violations of assumptions possible with MPI debugging tools. First, the MPI standard provides considerable latitude for conforming implementations, while we run the ISP tool on *specific* MPI platforms. Here, one of the most important features in the ISP tool is the notion of *completes-before* (CB). Briefly, ISP intercepts user's MPI program actions using the PMPI mechanism, and even deliberately delays and/or issues these actions into the MPI runtime. It does this in its quest to discover the full extent of non-determinism (*e.g.*, all sends matching a wildcard receive). ISP relies on the completes-before graph to achieve this goal. Therefore, even if run on a

[3] The mathematical derivation to calculate the number of riffle shuffles of five decks of five cards applies here.

fixed platform that has a specialized implementation of the MPI standard, ISP gives the effect of running on any other MPI-compliant platform. We define *completes-before* to be the partial order of MPI action occurrences that *must be guaranteed by all standards-compliant MPI libraries.* After encountering many non-intuitive (but correct) MPI examples, we realized that the completes-before graph can help users understand such examples. The display of completes-before is an important part of ISP's GUI.

Since MPI is a complex and rich API, it is important for MPI program verification tools to display helpful information about MPI operation scheduling. With the growing use of API functions with complex non-blocking and/or out of order completion semantics in concurrent programming, concurrency debugging tools that explicate API behavior can prove to be invaluable to their users. We later give an example concerning MPI probes that illustrates this issue.

2 Background, Related Work

Frameworks such as the Eclipse Parallel Tools Platform (PTP, [6]), Microsoft's Visual Studio, and TotalView are popular among MPI programmers. Shaeli [7] has explored MPI program trace visualizations. These tools do not have ideas similar to ours. The tool CHESS [8, 9] from Microsoft Research focuses on thread verification. It includes many useful features such as displaying schedule strings that lead to an error, and pre-emption control techniques that help users root-cause errors. Because of the differences between threading and message passing, these ideas are not directly applicable to MPI.

A Jumpshot (or other [10]) view of an MPI program execution driven by simple test harnesses often shows the temporal clustering and recurrence of MPI communications. Such tests (and hence visualizations) rarely involve the elusive 'corner case' interleavings that lead to bugs. A designer may try to 'jiggle' the schedule by inserting random delay statements. While often effective for thread programs, this rarely has the intended effect with MPI programs for several reasons, the main reason being that in most MPI programs, executions fall into a handful of equivalence classes. Thus, most schedule perturbations cause only the order of independent actions to swap. In addition, random delay statements add delays that affect the computational speed even where things do not matter.

Consider a simple example of an MPI program execution where three MPI commands are posted – one being a wildcard receive and the other two being two competing sends that try to match this receive. After all these commands are posted, the external world has *no* control over which send matches the wildcard receive. Inserting noisemakers into MPI library implementation codes is infeasible in most cases (*e.g.*, commercial MPI libraries). Even if arranged through PMPI, there is only so much MPI runtime control one can obtain. In other words, a designer is left with ineffective control over permuting MPI non-deterministic choices. We need a systematic search over all such *dependency inducing* situations. Dynamic partial order reduction (DPOR, [11]) offers that route. Our brand of MPI-specific DPOR is Partial Order avoiding Elusive interleavings (POE) [1]

– the workhorse behind our ISP tool [1–4]. This paper is on equipping ISP – that has powerful core algorithms – with a commensurately powerful set of user-interface facilities. With this combination, users are not flooded in their visual displays with unnecessary (equivalent) schedule variations – a major hindrance when learning MPI and/or when debugging a large-scale MPI program.

High-performance libraries such as MPI have many subtle features that routinely confound even MPI experts. We have given the following "quiz" to many MPI experts and found them most often failing the test. The quiz is: *Can the* `MPI_Irecv` *of P2 be matched by the* `MPI_Isend(to P2)` *issued by P1?*[4].

```
P0: MPI_Isend(to P2, &h)   ; MPI_Barrier()       ; MPI_Wait(&h);...
P1: MPI_Barrier()          ; MPI_Isend(to P2, &h); MPI_Wait(&h);...
P2: MPI_Irecv(from *, &h)  ; MPI_Barrier()       ; MPI_Wait(&h);...
```

To summarize, today's MPI debugging tools have the following deficiencies:

- They flood the graphical view with relevant (few) and irrelevant (most) variations of executions, unable to highlight which is which. This causes cognitive overload.
- They are not equipped with means to determine *relevant* schedules (nothing like a DPOR algorithm for instance), as well as means to *enforce* these schedules during dynamic verification. This causes bugs to be missed. A revealing display is the short programs listed in [12] where such omissions can be starkly seen.
- They do not educate MPI users concerning the subtleties of the API. They do not inform MPI users to anticipate portability bugs – where speed-paths change. (In the above quiz, in some platforms, P0's send may be the one found matching the wildcard receive most of the time; in a new platform, P1's send may be the one.)

POE Recap: Here, we summarize how POE handles our 'quiz' above: ISP will (i) trap the MPI operations using PMPI, (ii) delay issuing the `Isend` from P0 and `Irecv` from P2, (iii) come to `MPI_Barrier` which is now issued into the MPI runtime, and (iv) now find that `MPI_Isend` of P1 is also eligible to match with P2's `Irecv`. At this point, ISP dynamically rewrites `MPI_Irecv(from *)` into `MPI_Irecv(from P0)` and `MPI_Irecv(from P1)`, and does the following. It first issues the set {`P2:MPI_Irecv(from P0)`, `P0:MPI_Isend(to P2)`} and finishes the rest of the program according to POE. Then, in another restart and replay of the entire program, it will issue the set {`P2:MPI_Irecv(from P1)`, `P1:MPI_Isend(to P2)`} and finishes the rest of the program according to POE. Changing the order in which the MPI operations from P0 and P1 are processed is justified because of their *independence* [5] – an assumption met by all MPI libraries we have come across. In fact, what POE is trying is to simulate all possible orderings and platform conditions under which the executions may happen. Notice that ISP does not generate different interleavings corresponding to the posting order of P0' and P2's first MPI calls.

[4] The sly answer is: *"Yes, otherwise we would not be asking this question!"*

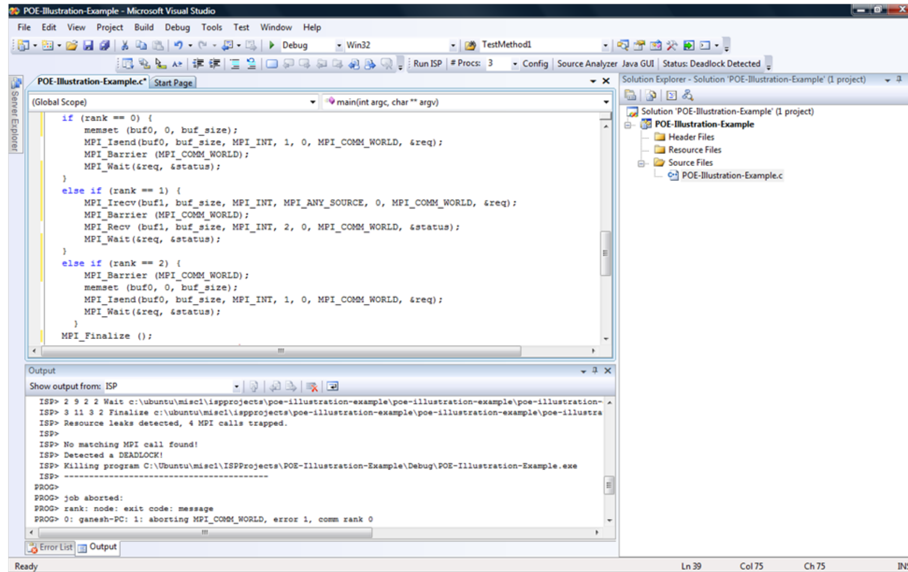

Fig. 1: Visual Studio Plug-in of ISP

Our Contributions: ISP has been released [13] for free download, runs on Windows, MAC OS/X, and Linux, handles several MPI libraries, and has been very effective in finding subtle bugs in large programs [2]. ISP has been used to verify the following large-scale programs, including: (1) the 14k LOC ParMETIS hypergraph partitioner (for two processes, we can verify ParMETIS in under five seconds on a laptop), (2) MADRE [14], (3) the MPI-BLAST program for genome sequencing [15], (4) the ADLB program [16], and (5) the Inter Radiosity Solver (IRS) from LLNL [17]. ISP has also handled most of the examples from Pacheco's widely used MPI book [18], thus making it an ideal tool for learning MPI. We now summarize features of ISP's GUI.

• All displays of ISP's user interface are arranged through a detailed trace file that is post-verification processed. The size of the trace file is small because of POE that generates and displays only the relevant interleavings.

• For alternate wildcard matches, ISP uses intuitive graphics (dotted lines for alternative matches). This way, one can visualize non-deterministic choices without "display explosion".

• ISP has Visual Studio (VS) integration as well as a Java GUI. Its Eclipse integration is in progress. The VS interface makes ISP truly a push-button model-checker of actual MPI codes – a user-friendly debugger look and feel.

• The Visual Studio interface displays all communication matches, opening windows dynamically based on the number of matches to be shown.

• It allows the user to cut into the underlying Visual Studio debugger at any selected highlight point (*e.g.* communication match or deadlock). This allows a smooth transition into conventional debugging when needed (a facility that we hope to develop further during ISP's integration into the Eclipse PTP.

• The fact that many collectives do not possess the barrier semantics comes out elegantly through the CB graph.
• Mouse-driven tool tips reveal the details of the underlying MPI commands.

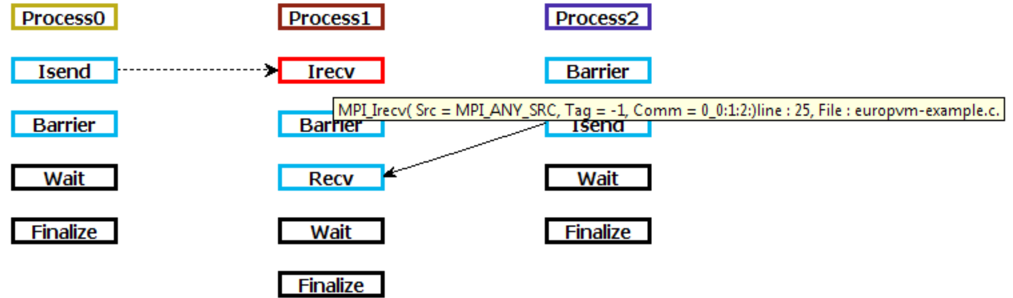

Fig. 2: Non-deadlocking Interleaving

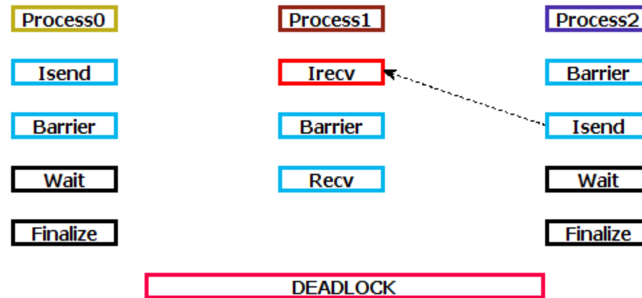

Fig. 3: Deadlocking Interleaving

3 Basics of ISP's User Interface

Consider the MPI example:

```
P0: Isend(to 1, &h)  ; Barrier()      ; Wait(&h)    ; ...
P1: Irecv(from *, &h); Barrier()      ; Recv(from 2); Wait(&h); ...
P2: Barrier()        ; Isend(to 1, &h); Wait(&h)    ; ...
```

Here, we highlight the MPI communication sources and targets and the communication handles. A user wanting to formally verify this program under ISP can launch the intuitive VS Plug-in of ISP, displaying the figure shown in Figure 1.

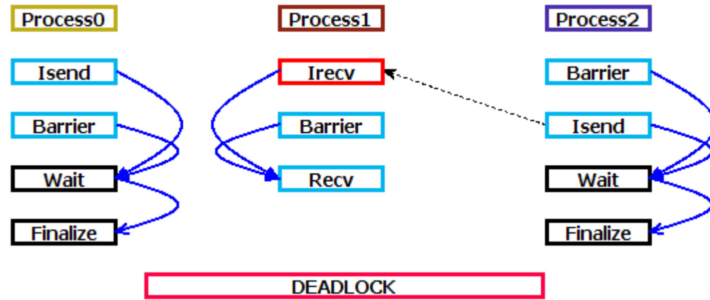

Fig. 4: IntraCB for Deadlocking Interleaving

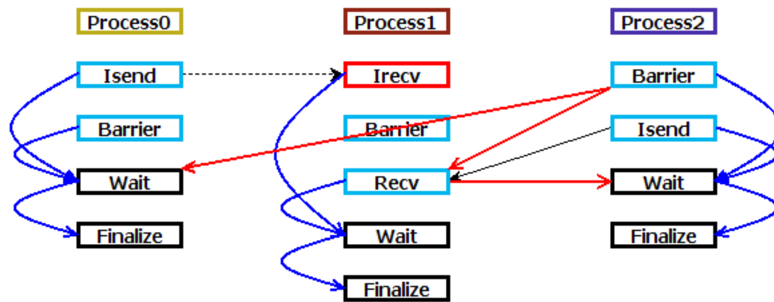

Fig. 5: IntraCB and InterCB Graphs

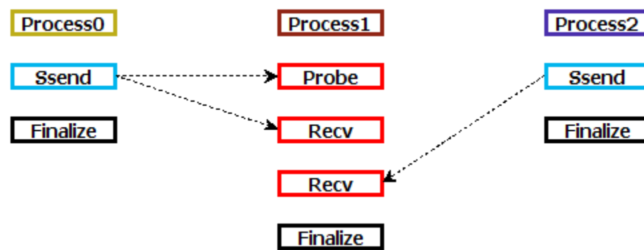

Fig. 6: Iprobe Match for Interleaving 1

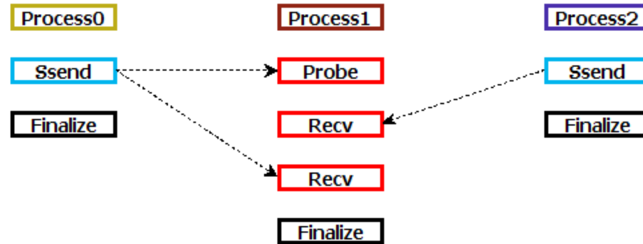

Fig. 7: Iprobe Match for Interleaving 2

Next, the user runs ISP whose POE algorithm will recognize that the `Irecv` of P1 has two potential send matches. (Note: there are many VS GUI views in this sequel; we refrain from presenting them owing to the lack of clarity on paper.) It runs two interleavings, finding a deadlock in one interleaving. The user may then single-step the execution trace. When `MPI_Barrier` is encountered, the display adjusts itself, showing all processes involved in the barrier. Soon the user reaches the deadlock point, at which time the user invokes the transition navigator, obtaining a tree display of the transitions invoked. Those transitions that are not matched yet are highlighted in red color in this display. At this point, the user may cut into Visual Studio for more incisive debugging.

Now, the user may be interested in finding the root cause of the bug. They display the communication matches which shows the two interleavings that this example has. The wildcard receive – source of non-determinism – is shown in red. The interleaving in which P0's Isend matches this receive (Figure 2) does not lead to a deadlock. The tool tips show the details of each icon (MPI command, file/line). The interleaving in which P2's Isend matches this receive (Figure 3) does lead to a deadlock.

The user further wants to drill down and locate the bug. They display the full completes-before graph with respect to a chosen group of nodes. Completes-before (CB) has two components: *intraCB* which shows the process-local part of CB, and *interCB* which shows how the CB graph builds across processes. They may merely display the *intra* completes-before relation (Figure 4) that shows the completes-before graph corresponding to the deadlocking interleaving. The completes-before graph shows that an `Isend` issued before an `MPI_Barrier` need not complete before the barrier; it can complete afterwards! This tells the user that both wildcard matches are possible. The CB graph also tells the user how different platforms may process the MPI commands out of order. They may choose to display both intraCB and interCB as in Figure 5. This tour should tell the reader that ISP's MPI-specific GUIs are able to say a lot within a small amount of space. *We also wish to point out the conceptual depth of these displays that can help a beginner learn MPI programming reliably.*

How Completes-Before (CB) is determined: The MPI standard requires non-overtaking in the sense that two MPI messages send from Pi and Pj arrive in that order (per tag and communicator). Thus, if there is a Send(to P1) followed by another Send(to P1) in an MPI program, these commands must *complete* in order (of course they are always issued in program order). On the other hand, if a Send(to P1) is followed by a Send(to P2) in an MPI process, these sends may finish out of order. Imagine the first one sending a large message while the second one sends a short message; in this case, it would be inefficient to wait for the first send to finish. It also is no violation of non-overtaking to allow the second send to finish. The full definition of the CB relation appears in [1]. Our formalization of CB allows us to achieve two objectives at once: (i) schedule actions within POE so that the maximal extent of wildcard receive non-determinism is discovered (the same idea is also followed for wildcard probes); (ii) it also displays to the MPI user how each platform may reorder the commands issued from *the very same* MPI process. Of course, the display of relevant interleavings by ISP avoids exploding the view with a display of interleaving variations of commands issued from *different* processes.
Probes De-Mystified An adaptation of a Fortran example [19] concerning probes is as follows:

```
P0: Ssend(to 1); ...
P1: Probe(from *); Recv(from *); Recv(from *); ...
P2: Ssend(to 1); ...
```

In the above example, either P0's Ssend(to 1) or P2's Ssend(to 1) can enable P1's Probe(from *). In a regular MPI platform, if P0's Ssend(to 1) enables the P2's Probe(from *), one can expect P0's Ssend(to 1) to match the Recv command of P1 following the Probe. If this program is ported and run on a different platform, P2's Ssend(to 1) may match the same Recv command. From [19], a designer will likely recall that suppose a Probe matches a certain send, then the following Recv command *need not match the same send*! What if the expert forgets this fact and is baffled by the inundation of traces shown by a conventional debugger? ISP would, on the other hand produce a display shown in Figures 6 and 7. This display very clearly shows which send the probe matched, and which send that the following receive actually matched.

4 Conclusions

Formal verification tools that display concurrency concepts are useful for concurrent program understanding and incisive debugging. With the multi-core and peta-scale revolutions looming, such efforts are long overdue. Almost all concurrency debugging runs into exponential variations in executions. In addition to interleaving reduction through partial order reduction, a vast array of "exponential compression" methods await to be exploited sufficiently in the area of MPI programming: process symmetries and data-space symmetries are two examples.

The current approach recommended with ISP (and most concurrency tools, for that matter) is to downscale a problem before such a tool is used. This is not sufficient for many bug classes, including resource bugs, and much work remains in this regard.

References

1. Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*. Springer, 2008.
2. 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.
3. Subodh Sharma, Sarvani Vakkalanka, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. A formal approach to detect functionally irrelevant barriers in MPI programs. In *EuroPVM/MPI*, pages 265–273, 2008. LNCS 5205.
4. Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Implementing efficient dynamic formal verification methods for mpi programs. In *EuroPVM/MPI*, pages 248–256, 2008. LNCS 5205.
5. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.
6. `http://www.eclipse.org/ptp`.
7. Basile Schaeli, Ali Al-Shabibi , and Roger D. Hersch. Visual debugging of mpi applications. In *EuroPVM/MPI*, 2008. LNCS 5205.
8. Madan Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.
9. `http://research.microsoft.com/CHESS`.
10. Richard Vuduc, Martin Schulz, Dan Quinlan, Bronis de Supinski, and Andreas Saebjornsen. Improved distributed memory applications testing by message perturbation. In *PADTAD*, 2006.
11. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05*, pages 110–121, 2005.
12. Michael DeLisi. `http://www.cs.utah.edu/formal_verification/ISP_Tests`.
13. Michael DeLisi. `http://www.cs.utah.edu/formal_verification/ISP-release`.
14. Stephen F. Siegel and Andrew R. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. In *EuroPVM/MPI*, pages 218–226, 2008.
15. `http://www.mpiblast.org`.
16. Rusty Lusk, Steve Pieper, Ralph Butler, and Anthony Chan. Asynchronous dynamic load balancing. `http://unedf.org/content/talks/Lusk-ADLB.pdf`.
17. The IRS Benchmark Code. `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/`.
18. Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.
19. A note on the probe command. `http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/node50.htm`.