Sound and Efficient Dynamic Verification of MPI Programs with Probe Non-Determinism [⋆]

Anh Vo[1], Sarvani Vakkalanka[1], Jason Williams[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. We consider the problem of verifying MPI programs that use `MPI_Probe` and `MPI_Iprobe`. Conventional testing tools, known to be inadequate in general, are even more so for testing MPI programs containing MPI probes. A few reasons are: (i) use of the `MPI_ANY_SOURCE` argument can make MPI probes non-deterministic, allowing them to match multiple senders, (ii) an `MPI_Recv` that follows an MPI probe need not match the `MPI_Send` that was successfully probed, and (iii) simply re-running the MPI program, even with schedule perturbations, is insufficient to bring out all behaviors of an MPI program using probes. We develop several key insights that help develop an elegant solution: prioritizing MPI processes during dynamic verification, handling non-determinism, and safe handling of probe loops. These solutions are incorporated into a new version of our dynamic verification tool ISP. ISP is now able to efficiently and soundly verify larger MPI examples, including MPI-BLAST and ADLB.

1 Introduction

The correctness of MPI programs is of paramount importance, especially considering the growing cost of conducting large-scale simulations, and the losses (opportunity costs and unreliable results) due to errant or crashing simulations. Conventional testing oriented approaches are woefully inadequate for finding bugs in MPI programs. To appreciate the magnitude of the problem, consider an MPI program with five processes where each process can make five MPI calls (called '*Generic MPI Example*'). Such a program has over 10 billion potential interleavings (schedules)! Unaware of which interleavings induce control flow variations down the execution path leading to bugs, testing tools often pursue a 'best effort' randomization of the interleavings. Unfortunately, given the exponentially growing execution space, these techniques do not attain adequate coverage of the *relevant interleavings* [1].

⋆ Supported in part by Microsoft, NSF CNS-0509379, CCF-0811429, 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.

While static analysis and model-based verification (*e.g.*, modeling programs in dialects (*e.g.* [17]) of SPIN [13]) are two widely followed *formal* approaches, the former generate false alarms causing a designer to spend weeks confirming whether a reported bug is genuine [12], while the latter requires designers to re-express an MPI program in an alternate notation. Thus, both these approaches prove impractical for realistic MPI programs, especially when also faced with the need to hand-model the semantics of the MPI calls and the surrounding C code, repeating all this for *each* program tuning step. In contrast, our approach – dynamic formal verification – runs the program after replacing the native scheduler with a *formal* verification scheduler. Pioneered by Godefroid [11] and later improved in the dynamic partial order (DPOR) approach [10], dynamic verification enjoys a growing presence, in tools such as CHESS [2], MODIST [23], CREST [8], Bandera [9], Java PathFinder [3], and Inspect [24].

Our dynamic verification approach for MPI programs based on a customized DPOR algorithm called Partial Order avoiding Elusive interleavings (POE) [16, 20–22] explores all relevant interleavings parsimoniously. Consider a special case of the *Generic MPI Example* with exactly one wild-card (`MPI_ANY_SOURCE`) receive "Recv(from *)" in process P0 which is matched by sends Send1(to P0) and Send2(to P0) issued by processes P1 and P2. Let all other MPI calls be point-to-point MPI calls. POE will explore just *two* interleavings, one where Send1(to P0) matches Recv(from *) and the other where Send2(to P0) matches Recv(from *). because these cases can affect verification outcome by conveying different values to the wildcard receive. Moreover, POE *enforces* these matches by dynamically rewriting Recv(from *) into Recv(from P1) and Recv(from P2) over two successive replays of the program. Permuting the issue order of point-to-point calls is pointless for finding safety violations. ISP has been used to verify a number of real-world MPI programs (*e.g.*, ParMETIS [14], MADRE [18], and the Implicit Radiation Solver or IRS [7]) for the absence of deadlocks, resource leaks, communication races, and assertion violations [21, 22]. ISP has been released [4] to run on Linux, Mac OS/X, and Windows, supporting the MPI libraries MPICH2, Open-MPI, and Microsoft MPI, supporting > 60 MPI 2.1 functions.

`MPI_Probe` detects the presence of a receivable message, while `MPI_Iprobe` call is its non-blocking counterpart. Here, we present the addition of `MPI_Probe` and `MPI_Iprobe` to ISP, so far deliberately postponed due to their highly non-deterministic behavior. Several key insights finally helped us design a sound and efficient algorithm, now allowing us to verify two large examples – MPI-BLAST and ADLB – that make heavy use of `Probe`s.

Related Work: Pioneering work on formal methods for MPI began with MPI-SPIN [17], a SPIN-based model checker that can be used to verify MPI programs for deadlocks and safety errors. Its need for hand-built verification models was already mentioned. ISP is believed to be the only dynamic formal verification tool for MPI. Non-MPI dynamic formal verification tools (*e.g.*, [2, 23]) have no awareness of the MPI semantics – hence inapplicable for MPI.

2 Dynamic Interleaving Reduction with MPI Probes

The reason why handling `Probe` and `Iprobe` is an involved process is captured by the following three examples. (*Note:* Unless otherwise indicated, we assume that `Iprobe` is called in a polling-loop until its flag is set to true – the most common usage of Iprobes. We also represent `MPI_ANY_SOURCE` by a *. For simplicity, we will ignore the data payloads, tags, and statuses of these messages and only focus on the destination/source of the sends and receives. *End Note:*)

Example-1:

```
P0: Isend (to P1); Ssend(to P2); Wait ();
P1: Iprobe (from *, &status); Recv (from status.MPI_SOURCE), Recv (from *)
P2: Recv(from P0), Ssend (to P1);
```

If this example is naively run on a regular MPI platform, one may find that P0's `Isend(to P1)` enables P1's `Iprobe` to exit its loop. This causes `P1`'s receive to actually receive P0's `Isend(to P1)` itself (the message that was probed). Thereafter, P0's Ssend will match P2's Recv, and then P2's Ssend will match P1's Recv. If this program is ported and run on another machine, one may find a different outcome: (i) while P0's `Isend` is active, P0's `Ssend` can post. (ii) this causes P2's `Recv` to fire, followed by P2's `Ssend`. (iii) Now we can have both P0's `Isend` and P2's `Ssend` become candidates for P1's `Iprobe` match (because this is a wildcard `Iprobe`, where * shows `MPI_ANY_SOURCE`).

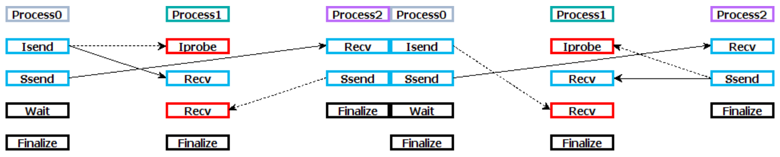

Fig. 1: Different Possible Matching of Iprobes

Formal tools such as ISP must discover the most general set of such executions possible on any platform and find bugs in them. It must also discover these behaviors by parsimoniously exploring interleavings. Our new algorithms are designed with these goals in mind; in particular, the insight that allows us to handle Example-1 is as follows:

Recognize the priority level at which a dynamic verifier must fire probes. In other words, by delaying the wildcard `Iprobe`, *we can allow both match possibilities the chance of being discovered.* This approach is, fortunately, a fundamental part of POE in handling wildcard receives *[21]. Our new algorithm now simply incorporates this approach for wildcard probes also. Also when multiple probe matches are found, our new algorithm also performs dynamic rewriting, as was done in ISP for wildcard receives, as explained in our Generic MPI Example.*

An important issue that arises when dealing with `Iprobe` is that it is not *a priori* known how many times `MPI_Iprobe` will be called from within the probe loop until it returns true, even after a send operation that directs the probing process has been launched. To overcome this non-deterministic behavior, our scheduler employs the following approach:

When eligible sends are found for `Iprobe`*s, the scheduler of ISP must busy-wait on the* `Iprobe`*s until they return true.* Since matching sends are already posted, the busy-wait loops are guaranteed to terminate. However, by doing this, we do not consider the case where matching sends have been initiated, but the `Iprobe` call can still non-deterministically return false. For our verification purposes, we assume that the false code path contains only harmless statements. A complete verification solution might require other analysis methods such as static analysis, for which we are also actively exploring.

There are other subtleties associated with probes; they are illustrated through the next two examples:

Example-2:

```
P0: Isend(to P1); Ssend(to P2); Wait();
P1: Iprobe(from *, &status); Ssend(to P2);
    Iprobe(from *, &status); Recv (status.MPI_SOURCE); Recv(from *);
P2: Recv(from P0); Recv (from P1); Ssend(to P1);
```

Despite similarities with Example-1, this example contains two `Iprobe`s that look very much the same, yet have very different possible matching sends. Here, the first `Iprobe` in P1, having to complete before the `Ssend (to P2)` in P1, has only one possible matching send (namely `Isend(to P1)` issued by P0. After the `Ssend(to P2)` of P1 is matched with P2's `Recv(from P1)`, the situation is now played out similarly to the previous example. This allows the next `Iprobe` to see two possible matching sends, namely P0's `Isend(to P1)` and P2's `Ssend(to P1)`. ISP has to consider both these matches in two different interleavings.

The high level observations from this example are the following.

Recognize the local completion semantics *of* `Iprobe`*s.* `Iprobe` *calls have to be completed before any other following MPI calls in the same process. The* delayed execution *of ISP has to take this into account.*

Example-3:

```
P0: Recv(from P1); Send (to P1);
P1: Iprobe(from P0, &flag, &status);
    if (flag == false) Send(to P0) else Recv (from P0);
```

From this example, it is straightforward to see that busy waiting on `Iprobe` inside the scheduler is not always safe (the singly threaded scheduler can go into an infinite loop, as no send has been posted). Busy-waiting is safe only when matching sends for `Iprobe` are already determined. This example describes the simplification of a pattern that is found repeatedly in many large MPI programs.

Our new algorithm handles such situations as follows: The `Iprobe` is delayed. *Recognize that busy-waiting for* `IProbe`*s to return is not always safe. The* `Iprobe` *calls should be allowed to return, with the flag set to false, when the execution cannot progress anymore and a matching send has not yet been found.*

3 ISP Overview

At a high level, ISP works by intercepting the MPI calls made by the target program and making decisions on when to send these MPI calls to the MPI library. This is accomplished by the two main components of ISP: the Profiler and the Scheduler.

The Interposition Layer: The interception of MPI calls is accomplished by compiling the ISP interposition layer together with the target program source code. The interposition layer makes use of the profiling mechanism PMPI. It provides its own version of *MPI_f* for each corresponding MPI function f. Within each of these MPI_f, the profiler communicates with the scheduler using TCP sockets[3] to send information about the MPI call that the process wishes to make. It will then wait for the scheduler to make a decision on whether to send the MPI call to the MPI library or to postpone it until later. When permission to fire f is granted from the scheduler, the corresponding *PMPI_f* will be issued to the MPI run-time. Since all MPI libraries come with functions such as *PMPI_f* for every MPI function *f*, this approach provides a portable and light-weight instrumentation mechanism for MPI programs being verified.

The ISP Scheduler: The scheduler is where our main scheduling algorithm, namely POE (Partial Order avoiding Elusive interleavings) is carried out. The scheduler meets the following objectives: G1: discovers the maximal set of sends that can match a wildcard receive (viewed across all MPI-standard compliant MPI libraries); G2: accurately models the semantics of the global operations (such as barriers) of MPI. In MPI, not all MPI operations issued by a process complete in that order, and a proper modeling of this out-of-order completion semantics is essential in order to meet goals G1 and G2. For example, two `MPI_Isend` commands issued in succession by an MPI process P1 to the same target process (say P2) are forced to match in order, whereas if these `MPI_Isend`s are targeted to two *different* MPI processes, then they *may match contrary to the issue order*. As another example, any operation following an `MPI_Barrier` must complete only after the barrier has completed, while an operation issued *before* the barrier may linger across the barrier, and actually complete *after* the barrier!

Section 3 provides an ISP overview and summarizes the POE algorithm. Section 4 begins an elaboration of our main contributions, namely support for `Probe`s and `Iprobe`s.

Main Steps of the POE Algorithm: The POE algorithm works as follows. There are two classes of statements to be executed: (i) those statements of the embedding programming language (C, C++, and Fortran) that do not invoke MPI commands and (ii) the MPI function calls. The embedding statements in an MPI process are local in the sense that they have no interactions with those of another process. Hence, under POE, they are executed in program order. When an MPI call f is encountered, the scheduler records it in its state; however, it does not (necessarily) issue this call into the MPI run-time. (Note: When we say that the scheduler issues/executes MPI call f, we mean that the scheduler grants permission to the process to issue the corresponding PMPI_f call to the MPI run-time). This process continues until the scheduler arrives at a *fence*,

[3] When running within a local machine, ISP uses unix sockets to reduce communication overhead.

where a fence is defined as an MPI operation that cannot complete after any other MPI operation following it. Note that both `MPI_Probe` and `MPI_Iprobe` are considered fences. The list of such fences include `MPI_Wait`, `MPI_Barrier`, *etc.*, and are formally defined in [21]. When all MPI processes are at their individual fences, the full extent of all senders that can match a wildcard receive becomes known, and dynamic rewriting can be performed with respect to these senders. The collection of sends and matching receives can then be issued. For details, please see [21].

Completes-Before Ordering: The Completes-Before (CB) ordering accurately captures when two MPI operations x and y issued from the same process in program order are guaranteed to complete in that order. For example, if an MPI process P1 issues an `MPI_Isend` that ships a large message to P2 and then issues `MPI_Isend` that ships a small message to P3, it is possible for the second `MPI_Isend` to complete first. A summary of the completes-before order of MPI is as follows: (i) **Send Order**: Two `Isend`s sending data to the same destination complete in issue order. (ii) **Receive Order**: Two `Irecv`s receiving data from the same source complete in issue order. (iii) **Wildcard Receive Order**: If a wildcard `Irecv` is followed by another `Irecv` (wildcard or not), the issue order is respected by the completion order. (iv) **Wait Order**: A `Wait` and another MPI operation following it complete in issue order. For a formal description of the CB relation, please see [21].

4 Implementation of Probe and Iprobe

Early issue of sends: As mentioned before, ISP does not allow the processes to issue MPI calls into the MPI runtime system until all processes reach a fence point (otherwise ISP cannot control from which send a wildcard receive chooses to receive). Consequently, all receives as well as non-blocking sends are delayed. *Note:* `MPI_Sends` are also treated as non-blocking, because they are usually buffered if the message size does not exceed the eager send limit *End Note:*. This *delayed issue* approach poses a major obstacle for implementing `Probe`s and `Iprobe`s: the returned status of a probe call cannot be determined. This problem can be solved by one of these two approaches:

- Having ISP manually manipulate the returned status, which would also require the trapping of `MPI_Get_count`.
- Issue the sends after the scheduler finishes collecting the envelope of the sends. This *early issue* approach for sends is considered safe because, unlike receives which can have non-deterministic behavior under the presence of wildcards, sends are always deterministic. This allows the MPI library to automatically take care of writing the returned status.

Although both methods are technically sound, we opted to go with the latter approach since trapping more MPI calls incurs higher overhead for the scheduler. However, the chosen approach also poses several difficulties, one of which can be described by the following example: In the following code, assume that `data` is large enough to exceed the eager send limit:

```
P0: Send (data, to P1); Probe(from P1, &status); Recv (from P1);
P1: Send (data, to P0); Probe(from P0, &status); Recv (from P0);
```

When the verification is run under no buffering mode[4], the program is obviously deadlocked, and ISP detects the deadlock easily since the program is instructed to execute all `MPI_Send`s as `MPI_Ssends` . The situation gets complicated when the program is verified in buffering mode, under which ISP will consider all `MPI_Send`s as non-blocking. However, because the size of data exceeds the eager send limit of the MPI library, the early issued `MPI_Send`s block within the processes without the scheduler being aware of its states. In order to address this problem, *all `MPI_Sends` are converted to MPI_Isend following by an MPI_Wait.* The `MPI_Send`s are issued early as `MPI_Isend`s and they are completed by `MPI_Wait`s when the scheduler finds a matching receive.

Basic POE algorithm with Probe: Consider Figure 2, which is a slightly modified version of Example-1 mentioned earlier. We will use this example to illustrate the basic steps of POE, with the *italicized text* indicating new changes (*i.e.* the new contributions of this work) made to POE to handle `Probe`s and `Iprobe`s.

```
1: if (rank == 0) {
2:   MPI_Isend(to P1,&req); MPI_Barrier();
3:   MPI_Ssend(to P2); MPI_Wait(&req);}
4: else if (rank == 1) {
4:   MPI_Barrier(); while (!flag) MPI_Iprobe(from *, &flag, &status);
5:   MPI_Recv(from status.MPI_SOURCE); MPI_Recv(from *);}
6: else if (rank == 2) {
7:   MPI_Irecv(from 0,&req); MPI_Barrier();
8:   MPI_Wait(&req); MPI_Ssend(to P1);}
```

Fig. 2: *Ordering Semantics and Operation Lifetimes*

- Collect `Isend` of line 2, *let the process issue it. Mark it as "issued", but still not "matched"*
- Collect `Barrier` (line 2), and do not issue.
- `Barrier` is a fence, stop collecting from rank 0; switch to rank 1.
- Collect `Barrier` (line 4), and do not issue; switch to rank 2.
- Collect `Irecv` (line 7), and do not issue. Then collect `Barrier` (line 7), and do not issue.
- A fence has been reached in every rank. Now, form a match set in priority order, with the following priority order followed: barriers first, then non wildcard sends/receives/*probes*, and finally wildcard sends/receives/*probes*.
- In our current state, there is indeed a highest-priority match set formed by the barriers. Now, POE sends these `Barrier`s into the MPI runtime through `PMPI_Barrier` calls.

[4] ISP can verify MPI programs under full buffering or no buffering. With the full buffering mode, all `MPI_Send`s are buffered. With the no buffering mode, all `MPI_Send`s are treated as `MPI_Ssend`s

- The next ordering points (fences) are attained at `Ssend` (line 3), `Iprobe` (line 4), `Wait` (line 8). Note that the `Ssend` is already issued by the process due to our *early issue of send* implementation (Apparently that process is now blocking).
- The non-wildcard match-set of `Ssend` (line 3) and `Irecv` (line 7) are both issued.
- Now we have reached the ordering point of `Wait` (line 3), `Iprobe` (line 4), and `Ssend` (line 8). No other higher priority match sets exists, we can now find out all the potential senders that can match the `Iprobe`.
- Dynamically rewrite `Iprobe(*)` into `Irecv(P0)` and `Iprobe(P2)`, in two different executions.
- Form the first match set of `Iprobe(from P0)` and `Isend(to P1)` of line 2. *Mark them both as matched. Only issue the `Iprobe` because the `Isend` was already issued earlier. In addition, the scheduler needs to ensure that it should not collect anymore calls from P0 until a receive is posted for the Isend of line 2.* i.e.*, the fence point for P0 is still in place.* Since a matching send is already issued earlier, P0 will *busy-wait* on the `Iprobe` of line 8. Thus, the program loop should execute only once!
- Form the second match set of `Iprobe(from P2)` and `Ssend (to P1)` of line 8. Pursue this interleaving though re-execution of the MPI program.

Soundness: Verification under ISP without probes is argued sound in [21]. The addition of probes preserves all our earlier assumptions. The engineering of efficient support for probes is our main contribution here; the elementary semantics of our engineering are similar to how wildcard receives were handled earlier, and hence are sound.

5 Experimental Results

We have tested our new algorithm against all the aforementioned examples (the original algorithm was tested against various benchmarks and in many cases found deadlocks missed by conventional tools [1]). In all our tests, ISP verifies the program with the *correct* number of interleavings, *i.e.*, all the interleavings explored by ISP are relevant. We also verified the subtle example found in the MPI Standard (Example 3.18 [6]), where a `Recv` might end up not receiving what the process probed (because the `Recv` was used to receive from `MPI_ANY_SOURCE`, not the source returned by `Probe` through the status).

Consider Figure 3. Note that because the `Recv` of lines 9 and 11 are called as wildcard receives and do not use the status returned by `Probe`, the program becomes incorrect. ISP correctly sees that there are four possible matchings (two for the first Probe, two for the first Recv). Here is a summary of what happens:

- All fence points reached at `Send` of line 2, line 4, and `Probe` of line 7.
- Two possible matchings for `Probe(*)`, one with `Send` of line 2, the other with `Send` of line 4.
- Pursue the 1st interleaving: rewrite `Probe(*)` into `Probe(P0)`.
- P2 reaches fence point again at `Recv(*)` of line 9. Both P0 and P1 still at previous fence points.

```
1: if (rank == 0)
2:    MPI_Send (data1, MPI_INT, to P2);
3: else if (rank == 1)
4:    MPI_Send (data2, MPI_DOUBLE, to P2);
5: else if (rank == 2) {
6:    for (i = 0; i < 2; i++) {
7:       MPI_Probe(from *, &status);
8:       if (status.MPI_SOURCE == 0)
9:          MPI_Recv(data1, MPI_INT, from *);
10:      else
11:         MPI_Recv(data2, MPI_DOUBLE, from *);}}
```

Fig. 3: *Pseudocode of pattern found in Example 3.18 in MPI 1.1 Standard*

- Two possible matchings for `Recv(*)`, one with `Send` of line 2, the other with `Send` of line 4.
- Pursue the 1st interleaving: rewrite `Recv(*)` into `Recv(P0)`
- No extra interleaving is required for the `Probe` and `Recv` of the second iteration of the loop (only one matching send is possible).
- When the execution finishes, pursue second interleaving: rewrite `Probe(*)` of line 7 into `Probe(P1)`, `Recv(*)` of line 9 into `Recv(P0)`.
- Third interleaving: rewrite `Probe(*)` of line 7 into `Probe(P0)`, the `Recv(*)` of line 9 into `Recv(P1)`.
- Fourth interleaving: rewrite `Probe(*)` of line 7 into `Probe(P1)`, the `Recv(*)` of line 9 into `Recv(P1)`.

We have also tested ISP with MPI-Blast[5] and ADLB[15], two large and non-trivial MPI programs (about 70K lines of code for MPI-Blast and 8K lines of code for ADLB). Both of which have extensive usage of wildcard `Probe` and `Iprobe`. In both cases, the new algorithm successfully verified both programs for freedom from deadlocks and assertion violations. The verification was done for a small number of processes. We also observed that the number of interleavings grew very fast for high number of processes due to the high numbers of wildcard probes.

6 Conclusions

We described our new algorithm to formally verify MPI programs for deadlocks, resource leaks, and assertion violations under the presence of `MPI_Probe` and `MPI_Iprobe` calls. The algorithm is implemented in our tool ISP. To the best of our knowledge, ISP is the *only* dynamic verification tool for MPI that verifies large and non-trivial MPI programs. The new algorithmic enhancements enable ISP to handle a wider range of MPI applications.

Earlier, we mentioned that one drawback of the current implementation of `Probe` is the large number of interleavings generated when the number of wildcard calls is high. This has recently been addressed by more intelligently handling MPI dependencies (future publication). We are also exploring other techniques that downscale an MPI program while preserving the targeted bugs.

References

1. `http://www.cs.utah.edu/formal_verification/ISP_Tests/`.
2. `http://research.microsoft.com/en-us/projects/chess/`.
3. `http://javapathfinder.sourceforge.net/`.
4. `http://www.cs.utah.edu/formal_verification/ISP-release/`.
5. `http://www.mpiblast.org`.
6. Mpi standard 1.1. `http://www.mpi-forum.org/docs/mpi-11.ps`.
7. The IRS Benchmark Code. `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/`.
8. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, Univ. of California, Berkeley, Sep 2008.
9. M. Dwyer, J. Hatcliff, and D. Schmidt. Bandera : Tools for automated reasoning about software system behavior. In *ERCIM News, 36*, Jan. 1999.
10. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *POPL 2005.*
11. P. Godefroid, B. Hanmer, and L. Jagadeesan. Systematic software testing using VeriSoft: An analysis of the 4ess heart-beat monitor. *Bell Labs Technical Journal*, 3(2), April-June 1998.
12. P. Godefroind and N. Nagappan. Concurrency at microsoft - an exploratory survey, *EC2 2008.*
13. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
14. G. Karypis. METIS and ParMETIS. http://glaros.dtc.umn.edu/gkhome/views/metis.
15. R. Lusk, S. Pieper, R. Butler, and A. Chan. Asynchronous dynamic load balancing. `unedf.org/content/talks/Lusk-ADLB.pdf`.
16. S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. A formal approach to detect functionally irrelevant barriers in MPI programs. *EuroPVM/MPI 2008. LNCS 5205, pp. 265-273.*
17. S. F. Siegel and G. S. Avrunin. Verification of MPI-based software for scientific computation. *SPIN 2004.*
18. S. F. Siegel and A. R. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. *EuroPVM/MPI 2008.*
19. S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby. Scheduling considerations for building dynamic verification tools for MPI. *PADTAD-VI 2008.*
20. S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. Implementing efficient dynamic formal verification methods for MPI programs. *EuroPVM/MPI 2008. LNCS 5205, pp. 248-256.*
21. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. *CAV 2008. LNCS 5123, pp. 66-79.*
22. A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, , and R. Thakur. Formal verification of practical mpi programs. *PPoPP 2009, pp 261-269.*
23. J. Yang, et al. MODIST: Transparent Model Checking of Unmodified Distributed System. *NSDI 09.* To appear.
24. Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. *SPIN 2008. LNCS 5156, pp 288-305.*