

Practical Model-Checking Method for Verifying Correctness of MPI Programs

Salman Pervez¹, Ganesh Gopalakrishnan¹, Robert M. Kirby¹, Robert Palmer¹,
Rajeev Thakur², and William Gropp²

¹ School of Computing
University of Utah
Salt Lake City, UT 84112, USA

² Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA

Abstract. Formal program verification often requires creating a model of the program and running it through a model-checking tool. However, this model-creation step is itself error prone, tedious, and difficult for someone not familiar with formal verification. In this paper, we describe a tool for verifying correctness of MPI programs that does not require the creation of a model and instead works directly on the MPI program. Our tool uses the MPI profiling interface, PMPI, to trap MPI calls and hand over control of the MPI function execution to a scheduler. The scheduler verifies correctness of the program by executing all “relevant” interleavings of the program. The scheduler records an initial trace and replays its interleaving variants by using dynamic partial-order reduction. We describe the design and implementation of the tool and compare it with our previous work based on model checking.

1 Introduction

Parallel programs are notoriously difficult to debug, and MPI programs, particularly those that have intricate control flow or employ relatively new features such as one-sided communication, are no exception. Tools such as MARMOT [7], MPI-CHECK [8], Umpire [16], and Intel Message Checker [2] can detect many errors in MPI programs but do not guarantee that all interleavings of the processes in the program being tested have been systematically examined. While there are an exponential number of such interleavings, partial-order reduction [1, Chapter 10]—a class of methods belonging to the area known as model checking [1]—offers specific approaches to examine only *some* (usually a small fraction) of these interleavings and declare that the effect of examining all the interleavings has been achieved. Partial-order methods are commonly used to verify *models* of parallel programs. For MPI programs, this approach would require that programmers build, either manually or automatically, a *model* (description) of their protocol in a language such as Promela [6], MPI-SPIN [13], or Zing [9]. This model-creation step is known to be tedious and error prone.

We take the *in situ* approach to model checking, previously demonstrated in the context of many languages, including C programs in tools such as [5, 18] and Java programs in tools such as [17]. During in situ model checking, programs written in the target language (usually with some obvious simplifications such as data-range reduction) are directly model checked, without first creating a model. In this paper, we describe our tool that performs in situ model checking of MPI programs that use one-sided communication. We use a *dynamic* version of partial-order reduction (DPOR, [3]) to reduce the number of interleavings, thus being able to, in effect, exhaustively examine all traces of small (but intricate) MPI programs. We call our tool *in situ dynamic partial order*, or ISP. ISP handles many standard MPI communication functions, including `MPI_Barrier`, various flavors of `MPI_Send` and `MPI_Recv`, and some MPI one-sided functions. In this paper we focus on one-sided functions, partly because of the inherent intricacies of handling one-sided communication under in situ scheduling. The complex nature of MPI one-sided communication also forces us to use information specific to the underlying library, MPICH2 in this case. One restriction placed by MPICH2 is that for passive-target one-sided communication, the target process needs to be inside the MPI progress engine in order to process lock requests. We account for this restriction in ISP as described in Section 2. ISP can easily be extended to efficiently handle other MPI implementations.

```

0: MPI_Init
1: MPI_Win_lock
2: MPI_Accumulate
3: MPI_Win_unlock
4: MPI_Barrier
5: MPI_Finalize

```

Fig. 1. Simple MPI program

To motivate the ISP approach, consider the simple MPI program given in Figure 1, executed by two processes P0 and P1. Figure 2 shows how ISP examines two different interleavings of this program. ISP employs the MPI profiling interface, PMPI, to trap MPI calls and hand over control of the MPI function execution to a scheduling process. This *scheduler* can dictate the order in which each process makes MPI calls. We define the block of code starting from the beginning of an MPI call, going forward in the code path including C program statements, and ending at the beginning of the next MPI call to be a *transition* (in our current example, there are no intervening C program statements). We assume that no transition executes infinitely (MPI calls always complete and the intervening C statements have no infinite loops). We also assume that the MPI program is well formed in accordance with the MPI Standard 2.0. The errors detected by ISP are safety properties [1], including deadlocks, violations of `assert` statements placed by the user, and exceptions thrown at runtime.

Given these assumptions, at Step 1, ISP would find processes P0 and P1 to be runnable (**Options**). Assume that ISP randomly chooses P1, executing the instruction shown against P1.1, which is an `MPI_Win_lock`. At Step 2, P0 and P1 are both runnable again; ISP picks P1, executing `MPI_Accumulate`. Proceeding in this manner, we reach Step 4, where P1 executes `MPI_Barrier`. At this point, the only runnable process would be P0, forcing Steps 5 through 8. The execution of `MPI_Barrier` by P0 results in both processes becoming runnable once again.

Step No.	Proc. Options		First		Second	
			Inter-leaving	Trace due to First Interleaving	Inter-leaving	Trace due to Second Interleaving
1:	P0	P1	P1	P1.1: MPI_Win_lock	P1	P1.1: MPI_Win_lock
2:	P0	P1	P1	P1.2: MPI_Accumulate	P1	P1.2: MPI_Accumulate
3:	P0	P1	P1	P1.3: MPI_Win_unlock	P1	P1.3: MPI_Win_unlock
4:	P0	P1	P1	P1.4: MPI_Barrier	P1	P1.4: MPI_Barrier
5:	P0		P0	P0.1: MPI_Win_lock	P0	P0.1: MPI_Win_lock
6:	P0		P0	P0.2: MPI_Accumulate	P0	P0.2: MPI_Accumulate
7:	P0		P0	P0.3: MPI_Win_unlock	P0	P0.3: MPI_Win_unlock
8:	P0		P0	P0.4: MPI_Barrier	P0	P0.4: MPI_Barrier
9:	P0	P1	P1	P1.5: MPI_Finalize	P0	P0.5: MPI_Finalize
10:	P0		P0	P0.5: MPI_Finalize	P1	P1.5: MPI_Finalize

Fig. 2. Interleavings explored for the example in Figure 1

Now ISP picks P1, followed by P0, generating the first interleaving (the last two actions being `MPI_Finalize`).

A naïve implementation of ISP would now backtrack to the *decision point* at Step 9, picking P0 instead of P1, as shown by the second interleaving. We say this is “naïve” because we know that the order in which `MPI_Finalize` is invoked is immaterial. This is precisely what partial-order reduction does: it computes which actions are *commuting actions*, meaning that their interleavings do not produce any semantically observable changes in program outcome. Another commuting pair in the above program would be the two `MPI_Barrier` invocations. Under a naïve approach, an N -way barrier can generate all $N!$ interleavings of the order in which processes encounter the barriers; under partial-order reduction, we can simply generate one interleaving and claim complete coverage.

If our current example is run with MPICH2 with the `MPI_Win_lock` operation specifying an exclusive lock, the only actions that require interleaving are the `MPI_Win_unlock` calls within which shared variable updates take place. For a process accessing the MPI window remotely, only its `MPI_Win_unlock` call modifies the communication window, posting all the accumulated updates within that particular epoch. For this example, the ISP tool would generate two interleavings as opposed to 504 interleavings³ if we were to use only the in situ feature without DPOR. Since the theory of partial-order reduction is vast, we simply present our assumptions as a table of commuting MPI operations (Figure 6), citing past references [10] based on which such tables can be created. The table can be adjusted to correspond to any MPI implementation of choice, or even to suppress certain interleavings for quicker bug hunting. Also, as opposed to *static* partial-order reduction where the commuting nature of the two `MPI_Finalize` invocations would have been determined while going forward during the first interleaving, we instead follow the *dynamic* approach to partial-order reduction,

³ $2 \times (10!/(5!)^2)$

in which we fully generate the first interleaving and walk up the stack trace and mark places where interleavings can be added. Space does not permit a fuller description of DPOR; we note only that it exploits run-time information to effect better reduction (e.g., wildcard communication, as described in [10]).

2 Basics of Scheduling and In Situ Model Checking

In situ model checking lets a *scheduler* control the transitions of the given MPI program. The scheduler opens an array of communication channels (via TCP sockets) through which it receives *appeals* from each process. The pseudocode in Figure 3 captures how the MPI call of a process (generically called `Generic_Func`) is processed. Basically, the MPI function call is intercepted by the profiling library. It then conveys the process id (`pID`), the call type (`Generic_Func`), and the remaining arguments to the scheduler through the `sendToSocket` call. In reply, the scheduler provides either a “go-ahead” or a “loop” to the appealing processes. A loop signal indicates that the appealing process must make an `MPI_Iprobe` call, a side-effect-free mechanism that causes control to enter the MPI progress engine to process all queued-up events within it. `MPI_Iprobe` is needed with `MPICH2` in order to cause progress to occur on communication with other processes, because `MPICH2` does not use an asynchronous progress thread in its progress engine.⁴ When the appealing process receives “go-ahead,” it issues `PMPI_Generic_Func`, which then enters the MPI library.

```
MPI_Generic_Func(arg1, arg2...argN) {
  sendToSocket(pID, Generic_Func, arg1,...,argN);
  while(recvFromSocket(pID) != go-ahead)
    MPI_Iprobe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD...);
  return PMPI_Generic_Func(arg1, arg2...argN); }
```

Fig. 3. PMPI instrumentation pseudocode

issued from a remote (nontarget) process as a “no operation.” However, an `MPI_Win_lock` issued by the target process may cause a lock on the one-sided communication window to be acquired. Let us denote the `MPI_Win_lock` issued from a target as `MPI_Win_lock_T`, from a nontarget as `MPI_Win_lock_NT`, and use the altered names `Win_unlock_T` and `Win_unlock_NT` assuming the same conventions. In our framework, these functions are used to indicate when the “trapped control” comes to the `MPI_Generic_Func` associated with these calls. We also use the notation `PMPI_Win_lock_T` to indicate the PMPI call coming after the “trapped” `MPI_Win_lock` call issued by the target, and we similarly use the notations `PMPI_Win_lock_NT`, `PMPI_Win_unlock_T`, and `PMPI_Win_unlock_NT`. These PMPI calls signal the point at which the MPI system first knows that these MPI calls are being made. We now present some of the scheduling deci-

In situ model checking depends on the designer’s understanding of how a given MPI library handles each MPI call in terms of the latitude allowed in the MPI standard. For example, `MPICH2` treats a `MPI_Win_lock` operation

⁴ `MPI_Iprobe` does not have a version corresponding to `MPI_Generic_Func`; otherwise, it would cause an infinite loop when these `MPI_Iprobes` are trapped.

sions made by ISP. We explain these with the aid of Figure 1. We rely on the following conventions:

- Because of the assumptions made in Section 1 about *transitions*, we know that each time the scheduler will be handling up to N appeals of the form `sendToSocket(pID, Generic_Func, arg1, ..., argN)` (it would be N appeals unless some process has executed a blocking operation).
- The scheduler also keeps track of the lock state of each MPI one-sided window. We will use the terms *window locked* and *window unlocked*.

Consider P0 to be the owner of the window. We call the owner the *target* because that is where all decisions about locking and unlocking the one-sided MPI window are made. Consider the program in Figure 1 run using processes P0 and P1. Specifically, consider the scheduler actions with respect to the following interleaving:

- P0 does `MPI_Win_lock_T`. The scheduler records that the window is locked, and issues a go-ahead, permitting the `PMPI_Win_lock_T` call to be made.
- P1 does `MPI_Win_lock_NT`. The scheduler treats this as a ‘no op’ (reasons in Section 1) and gives the go-ahead, allowing P1 to make the `PMPI_Win_lock_NT` call.
- P1 does `MPI_Accumulate`. The scheduler gives the go-ahead, allowing P1 to make the `PMPI_Accumulate` call.
- P1 does `MPI_Win_unlock_NT`. Noting that the window is locked, the scheduler gives the go-ahead, allowing P1 to make the `PMPI_Win_unlock` call. It records that P1 is blocked.
- P0 does its `MPI_Accumulate`, receiving a go-ahead.
- P0 does its `MPI_Win_unlock_T`. Clearly, the scheduler must issue a go-ahead to P0, causing `PMPI_Win_unlock_T` to occur, thus freeing up the window. Note that P1 has already made its `PMPI_Win_unlock_NT` call. However the following race condition could occur: P0 could hurry through the MPI progress engine upon issuing `PMPI_Win_unlock_T`. Suppose P1’s lock request reaches P0 only after P0’s `PMPI_Win_unlock_T` call has returned. However, since the MPICH2 progress engine has no separate thread to grant locks, P1’s successful acquisition of the window is at the mercy of P0 entering the progress engine again, which happens when P0 executes its `PMPI_Barrier` call.
- For simplicity, our scheduler is implemented in so that it moves only one process at a time—in the current example, after we let go P1, we await P1 to make its next MPI command appeal before entertaining any other process.
- However, if we keep P0’s appeal in abeyance, the following deadlock might occur: The `PMPI_Win_unlock_NT` can cause an event to be placed in the target’s event queue. These events are processed only when the progress engine is entered. Since we have kept P0 in abeyance, however, the progress engine won’t be entered.
- Instead of keeping P0 in abeyance, we keep sending “loop” to P0, which causes the `IProbe`’s to be issued. This ensures that P1’s event will be processed, causing P1 to reach its next MPI command, at which point we can stop sending “loop” to P0.

```

1 S.add_last(<0...n-1>) /* randomly choose a proc to run at each depth */
2 backtrack.add_last(<n-1>)
3 done.add_last(<n-1>)
4 if(!fork()) execlp(MPI program) /* run the given MPI program */
5 make all server connections
6 while(backtrack.size() > 0) {
7   current choice = pick randomly from backtrack
8   get readable envelopes for all runnable processes
9   current envelope = envelope for current choice
10  servers[current choice] << goahead /* the chosen MPI process may execute */
11  update block/unblock info for all processes based on current envelope
12  if(chosen process executed MPI_Finalize) {
13    specify that current choice is DONE
14    close(servers[current choice])
15    decrement active procses }
16  if(active processes != 0) { /* backtrack shows no runnable procs. */
17    if(depth+1 >= backtrack.size()) {
18      rprocs = <all currently runnable processes>
19      if(rprocs.size() == 0) /* POSSIBLE DEADLOCK */
20        close all socket connections
21        report deadlock and print trace}
22      else { /* current interleaving can be explored further */
23        S.add_last(<all runnable procs>)
24        backtrack.add_last(<S.last.last>) /* randomly choose a proc */
25        done.add_last(<empty>) } }
26    depth++ }
27  else {
28    /* we have gone through one interleaving of the program. Remove all
29     choices from S as well as backtrack until the last decision point.
30     This is where we had more than 1 choice of MPI processes. */
31    while(backtrack.size() > 0 && backtrack.last.size() == 1) {
32      updateBacktrackInfo()
33      S.remove_last()
34      backtrack.remove_last()
35      done.remove_last() }
36    if(backtrack.size() > 0) { /* make sure search is not over */
37      remove most recent choice from backtrack at current depth
38      /* the next interleaving will be forced to take an alternate route */
39      reset checker state for next interleaving
40      if(!fork()) execlp(MPI program)
41      make all server connections
42      depth = 0
43      active procs = n } } }

```

Fig. 4. DPOR-based scheduling algorithm

3 In Situ Model Checking with Dynamic Partial-Order Reduction

The algorithm of Figure 4 shows how ISP exhaustively explores all *relevant* interleavings of the given MPI process as determined by DPOR. The first interleaving is chosen at random by following a standard depth-first search. It is then simply a matter of traversing up the stack, having DPOR identify points where adding interleavings might be useful, and carrying on the search.

The following data structures are used by ISP: **S** contains the set of processes that are able to run at each depth; **backtrack** contains the set of processes that are allowed to run at each depth; **done** contains the set of processes that have been explored at each depth; **servers** is the set of n server connections, one with

S			backtrack	First Interleaving	S'		backtrack'	S''		backtrack''
-----			-----	-----	-----		-----	-----		-----
P0	P1	P1	P1	P1.1: MPI_Win_lock	P0	P1	P1	P0	P1	P1
P0	P1	P1	P1	P1.2: MPI_Accumulate	P0	P1	P1	P0	P1	P1
P0	P1	P1	P1	P1.3: MPI_Win_unlock	P0	P1	P0 P1	P0	P1	P0
P0	P1	P1	P1	P1.4: MPI_Barrier	P0	P1	P1			
	P1	P0	P0	P0.1: MPI_Win_lock		P1	P0			
	P1	P0	P0	P0.2: MPI_Accumulate		P1	P0			
	P1	P0	P0	P0.3: MPI_Win_unlock						
	P1	P0	P0	P0.4: MPI_Barrier						
P0	P1	P1	P1	P1.5: MPI_Finalize						
	P1	P0	P0	P0.5: MPI_Finalize						

Fig. 5. DPOR algorithm applied to Example 1

each MPI process; and **active processes** is initialized to the number of MPI processes n . The most interesting is the *backtrack* set. Readers may view it as a set of sets that keeps track of meaningful interleavings at each depth. The sets S and *backtrack* are almost identical except that S contains the meaningless interleavings as well. Hence, a naïve implementation of ISP would simply discard the *backtrack* set and refer only to S . We now explain how this algorithm works by referring to the interleavings shown in Figure 5. Note that these interleavings correspond to the MPI program of Figure 1.

S is initialized so that its first element contains both P0 and P1. The first element of *backtrack* contains only P1, chosen at random; *depth* is initialized to 0. In lines 12–14 we choose a process randomly from *backtrack* at depth 0. Note that for the entire first interleaving, there will only be one possible choice at each depth. We then indicate to P1 that it may make its MPI call `MPI_Win_unlock`, by answering its appeal with a go-ahead token. ISP must now update its internal bookkeeping information. It does so in line 19 by noting which processes are blocked/unblocked as a result of executing the chosen MPI process. We have now reached a point where *backtrack* will indicate no possible choices in the next step. Line 29 is responsible for calculating the runnable processes so they can be added to the *backtrack* set. Again, both P0 and P1 will be added to S as runnable processes. The last step is to increment *depth* and continue our random depth-first-search algorithm.

The first significant digression from this pattern occurs when P0 calls `MPI_Finalize`. At this point, since both processes have called `MPI_Finalize`, we execute the *else* clause of line 45. The idea is to remove all the choices already made, so that in the next execution of the loop, a different interleaving can be explored. Thus, we remove the `MPI_Finalize` executed by P0. The `updateBacktrackInfo` function then is called on line 51. This function traverses up the set S and identifies any transitions that may need to be interleaved with the `MPI_Finalize` just removed. If any such transitions are identified, the corresponding MPI process is added to the *backtrack* set.

Following our DPOR assumptions, no change results to the *backtrack* set. We continue to remove choices until we reach the P0.3: `MPI_Win_unlock` call. This time, the function `updateBacktrackInfo` updates the *backtrack* set to look like *backtrack'* in Figure 5. This indicates that the `MPI_Win_unlock` functions of P0 and P1 must be interleaved in order to get a different, meaningful interleaving. We continue removing all choices that have already been taken until the *backtrack* set looks like *backtrack''*. At this point, we are ready to start our search from the beginning.

The DPOR-based algorithm of Figure 4 identifies all such meaningful interleavings and terminates the search either when it encounters a deadlock scenario on line 30 or the search is completed. The commuting MPI operations assumed by ISP are given in Figure 6.

<i>MPIFunctions</i>	<i>Dependence</i>
<code>MPI_Init</code>	None
<code>MPI_Send</code>	<code>MPI_Send</code> , <code>MPI_Ssend</code> , <code>MPI_Recv</code>
<code>MPI_Ssend</code>	<code>MPI_Send</code> , <code>MPI_Ssend</code> , <code>MPI_Recv</code>
<code>MPI_Recv</code>	<code>MPI_Send</code> , <code>MPI_Ssend</code>
<code>MPI_Barrier</code>	None
<code>MPI_Win_lock</code>	None
<code>MPI_Win_unlock</code>	<code>MPI_Win_unlock</code>
<code>MPI_Win_free</code>	None
<code>MPI_Finalize</code>	None

4 Case Study: Byte-Range Locking

Fig. 6. Supported MPI functions

Our work in [11] described how we model checked the byte-range-locking protocol presented in [15]. This uncovered a subtle but crucial deadlock bug that had gone unnoticed during testing. With the help of ISP, we successfully caught this bug in the source code of this protocol. We note that no modeling effort and no changes to the source code were required. The results are presented in Figure 7. ISP has been tested on other smaller protocols and has worked as expected. It can be viewed as an exhaustive testing facility that gives the *effect* of examining all interleavings of small but intricate MPI programs.

<i>Program</i>	<i>#procs</i>	<i>interleavings w/o DPOR</i>	<i>interleavings with DPOR</i>
<code>byterange</code>			
reduced depth	2	2289	119
<code>byterange</code>			
full depth	2	-	1522

Fig. 7. Experimental results

allowed us to reduce the search depth and find the bug more quickly. By enabling DPOR within ISP, however, we were able to reproduce the deadlock scenario without having to reduce the search depth. While a hand-written model of the same protocol using SPIN could find the same bug without partial-order reduction [11], with ISP we have eliminated the nontrivial task of modeling MPI programs in Promela. The ISP approach is especially beneficial if the interven-

The most striking feature of these results is that ISP was unable to find this bug without using DPOR. The search algorithm was aborted after it did not finish within 24 hours. Our knowledge of the algorithm

ing C statements between MPI calls cannot easily be modeled in Promela, the actual MPI library in use cannot faithfully be modeled, or the error is triggered by a bug in the MPI library.

All our experiments consisted of test programs up to a depth of 20. In other words, they each had fewer than 20 MPI function invocations. Model checking such programs can take anywhere from half an hour to an hour on a single 1 GHz processor with 1 GB of memory.

5 Related Work and Conclusions

Model checking has been used for verifying MPI programs by Siegel et al. in [13, 14]. The closest related work to ours is [18], where *distributed* in situ model checking for Pthreads programs has been presented.

In our experience with the byte-range-locking algorithm, the initial program presented in [15] exhibited no discernible bugs, even with conventional testing. However, porting the same program to a laptop caused deadlock. This prompted us to model the protocol in Promela, revealing the bugs reported in [11]. This paper thus comes full circle, and shows that the same bugs can be detected at the C program level *without* model extraction.

Clearly, restarting ISP from `MPI_Init` in order to explore each new interleaving requires a huge overhead, even with partial-order reduction dramatically reducing the number of interleavings. To reduce the overhead, we plan to explore three ideas: (i) divide the program using MPI barriers, and interleave only the code between two subsequent barriers, cutting down the extent of interleavings and also helping to localize errors; (ii) use MPI checkpointing systems (e.g., [4]) to see whether we can checkpoint intermediate states and restart from there as opposed to restarting the search from `MPI_Init`; and (iii) use distributed ISP. We hope to research these topics in the context of ISP.

We would also like to make ISP compatible with all MPI library implementations, not just MPICH2. One issue is that the MPI standard gives too much freedom to implementors. For example, the blocking semantics of `MPI_Send` and some of the one-sided functions are far from being well defined. However, with ISP it is possible to force the underlying MPI library to follow a stricter interpretation of the MPI standard. This approach will allow ISP to be used with any MPI library that conforms to the MPI standard. The code of ISP is available at [12].

Acknowledgments

This work was supported by NSF award CNS-0509379, by the Microsoft HPC Institutes program, and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

1. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
2. Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *SE-HPCS '05*, pages 78–82, 2005.
3. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.
4. Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. In *ICPP*, August 2006.
5. P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186, 1997.
6. Gerard J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.
7. Bettina Krammer and Michael M. Resch. Correctness checking of MPI one-sided communication using MARMOT. In *EuroPVM/MPI 2006*, pages 105–114, September 2006. LNCS 4192.
8. G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15:93–100, 2003.
9. Robert Palmer, Steve Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby. Gauss: A framework for verifying scientific computing software. In *Workshop on Software Model Checking*, 2005. ENTCS 953.
10. Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby. Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In *PADTAD*, 2007.
11. Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Formal verification of programs that use MPI one-sided communication. In *EuroPVM/MPI*, pages 30–39, 2006.
12. Preliminary release of the ISP software at http://www.cs.utah.edu/formal_verification/isp.tar.gz.
13. Stephen F. Siegel. Model checking nonblocking MPI programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2007.
14. Stephen F. Siegel and George S. Avrunin. Verification of MPI-based software for scientific computation. In *SPIN Workshop*, pages 286–303, April 2004.
15. Rajeev Thakur, Robert Ross, and Robert Latham. Implementing byte-range locks using MPI one-sided communication. In *EuroPVM/MPI*, pages 120–129, September 2005.
16. Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Proc. of SC2000*, pages 70–79, 2000.
17. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE*, September 2000.
18. Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Workshop on Model Checking Software (SPIN 2007)*, July 2007.