

ISP: A Tool for Model Checking MPI Programs ^{*}

Sarvani Vakkalanka Subodh Sharma Ganesh Gopalakrishnan Robert M. Kirby

School of Computing, University of Utah
{sarvani, sv, ganesh, kirby}@cs.utah.edu

Categories and Subject Descriptors F. Theory of computation [F.3 Logics and Meanings of Programs]: F.3.1 Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords MPI, Formal Verification, Model Checking, Dynamic Partial Order Reduction

1. Introduction

A significant body of parallel programs are written using MPI [1]. Bugs in MPI programs arise due to many reasons, for example: (i) during manual optimizations that turn blocking sends/receives into their non-blocking counterparts, (ii) in the context of using wild-card communications (and the resulting non-determinism), and (iii) in programs that use one-sided communication [2].

Programmers are acutely aware of the need to test parallel programs over all their interleavings. They also know that this is impossible, thus confining testing to interleavings that are guessed to be adequate – very often incorrectly, as it turns out. Model checking [4] has helped debug many concurrent systems by relying on an *exhaustive* search of all possible interleavings – albeit on a suitably abstracted system model. For example, SPIN [5] recently won ACM’s prestigious software award, and has been used to verify many real-world protocols. For model checking to be successful in practice: (i) Designers must have modestly expensive techniques to create *models* of the protocols to be verified. (ii) They must be able to examine a fraction of the interleavings in a concurrent system, and be able to claim that the remaining interleavings are equivalent and hence need not be examined, using a suitable state space reduction algorithm [4, 5].

This paper presents a tool called ISP (in-situ partial order) which aspires to grow into a practical MPI model checker. In the only other active research effort on developing a model checker (called MPL_SPIN) for MPI programs [6], the following approach is taken: (i) they model MPI library functions in Promela (SPIN’s modeling language), and uses SPIN for verification; (ii) they use the C extension features of SPIN to model advanced features such as non-blocking communication commands. (iii) they require users to manually express their intended MPI C program in MPL_SPIN

^{*} Supported in part by NSF award CNS-0509379 and Microsoft HPC Institutes.

(a macro-based extension of Promela). The drawbacks of this approach are several: users are forced to express their parallel algorithms in Promela as well as C, and maintain consistency of these representations across the program life-cycle. In addition, non-MPI library functions have to be manually broken down into Promela. ISP, on the other hand, *directly* checks MPI C programs. There is no extra work involved in handling non-MPI library functions. It employs run-time model checking methods [11] based on dynamic partial order reduction (DPOR) [12]. Our first paper on ISP [3] showed how well DPOR could potentially work on simple examples, and also confirmed all the above advantages of ISP over MPL_SPIN. In this paper, we provide an overall performance assessment of ISP. We describe our new implementation that, in addition to facilitating easy experimentation, also relies upon the formal semantics of MPI (described in [8]) more faithfully in formulating the partial order reduction algorithm. We implement more MPI primitives, including wild-card and many collectives. We present improvements possible due to a search optimization that avoids re-initializing the MPI system through an MPI_Init for each interleaving examined. Last but not least, we suggest ways to combine static analysis and ISP-based model checking.

In [9], the authors improve the coverage of MPI testing methods by forcing many more message interleavings to occur. Yet, the key property of our ISP approach is that it does not merely increase the likelihood of generating more message interleavings; it *guarantees* to cover every relevant interleaving. The work in [10] is very similar to ISP except, instead of partial order reduction, they employ iterative context bounding. They show the effectiveness of their approach on many real-world benchmarks.

2. Why Dynamic Dependence?

Partial order reduction attempts to eliminate redundant traces generated through commuting actions. Consider two C statements $a[j]++$ and $a[k]--$ that are concurrently enabled in two threads. These actions commute if $(j \neq k)$ – a fact best known at run-time (static analysis can, in general, not determine such information accurately). This *dependence information* is exploited by DPOR at run-time, thus helping to eliminate needless interleavings. What about the commuting behavior of MPI primitives such as wild-card communications? It turns out, as we show in [8], that treating them requires a DPOR approach. This is because the senders that can match a wild-card receive statement are known only at run-time. If this information is not accurately determined, then either a conservative approach has to be employed or soundness may be compromised due to an overlooked dependency.

3. How ISP Works

Currently ISP checks for deadlocks and local assertion violations (soon to be extended to check for resource leaks) in MPI programs that read all their inputs at the beginning. For every MPI function (e.g. MPI_Send), ISP provides a replacement function (a modest,

one-time effort). When invoked, these replacement functions first consults a central scheduler through TCP sockets. If the scheduler gives permission, the replacement function invokes `PMPI_Send` (provided in the profiling layer of most MPI implementations, and having the same functionality as `MPI_Send`). This allows the scheduler to march the processes of a given MPI program according to one arbitrary interleaving, till all processes hit `MPI_Finalize`. ISP examines the resulting trace of actions, and records, at each of its choice points, whether a different process could have been selected. Such alternative choices are deemed necessary based on the dynamic dependence between actions in the current trace (see [8] for details). If, at choice point i , a different process p_1 is deemed necessary to have been run, ISP (which is implemented using “stateless search” [5]) re-executes the entire MPI program till it comes to choice point i , and picks p_1 to run. Our studies in [3] show that the DPOR algorithm in ISP can considerably reduce the number of interleavings than without DPOR.

4. Results and Assessment

We ran three simple examples: (i) the parallel trapezoidal rule computation (Trap), (ii) the control skeleton of the Monte Carlo evaluation of Pi (MC), and (iii) A byte-range locking protocol using one-sided MPI communication (BR). The byte-range protocol was actually a realistic protocol [2]. These codes as well as additional details (including ISP’s code with documentation) are available from our website¹.

- For Trap, a deliberately introduced deadlock was found instantaneously. Following correction, the program was exhaustively searched over 33 total interleavings, taking 8.94 seconds. Of this time, 8.44 seconds were spent in restarting the MPI system.
- For MC, three deadlocks (which were not evident upon casual inspection) were automatically detected (detailed on our website). After correcting the code, ISP ran over 3,427 interleavings, taking 15.52 minutes, of which 15.077 minutes were spent restarting the MPI system.
- For BR, one deadlock was found after exploring 62 interleavings. After correction, the code ran over 11,000 interleavings, and the run was killed (pending further improvements to ISP before we exhaust the execution space of BR).

5. Improvements to ISP

Eliminate Restart Overhead: The restart time of the MPI system is clearly a dominant overhead. This price is being paid because as opposed to existing model checkers which maintain state hash-tables, we cannot easily maintain a hash-table of visited states including the state of the MPI program as well as the MPI run-time system. (Note: In resorting to re-execution, we are, in effect, banking on deterministic replay.) One approach eliminate restart overheads – introducing control back-edges – seems to pay-off handsomely, and works as follows. At `MPI_Finalize`, one can reasonably assume that the MPI run-time state is equivalent to the one just after `MPI_Init`, and therefore simply reset user state variables and transition each process to the label after `MPI_Init` (similar to the approach in CHES). Preliminary experiments show that the execution time of MC reduces from 15.52 minutes to 63 seconds, finding the same deadlocks. For Trap, the time reduces from 8.94 seconds to 0.347 seconds.

Strength Reduction: In most MPI programs, control flows are unaffected by most (data) variables. This allows us to drop those variables (and the associated computations) not contributing to control flows or the local assertions being checked. This work is in progress.

Barrier Insertion: We are investigating how and whether users can insert a pair of `MPI_Barrier` instructions, and request ISP to perform interleavings only within their confines. These barriers can, later, be moved to other parts of the MPI program. This may help users localize model checking, following their own insights into their program.

Loop Peeling: Since most MPI programs have loops from which MPI operations are invoked, it seems attractive/necessary to (through static analysis) peel out some of the iterations and have them alone be trapped and interleaved by ISP. The other iterations of the MPI program loops must still carry out these communications – however, without being trapped by ISP. We are investigating ways in which this method can be implemented without unduly complicating ISP’s scheduler.

Distribute Search: ISP itself follows a highly parallelizable algorithm. The alternative interleavings may well be explored by other nodes of a large cluster. Our implementation of a dynamic partial order reduction algorithm for PThreads [7] (a different effort with no code in common) shows that with a suitably designed heuristic, linear speed-up on large clusters is a possibility.

6. Concluding Remarks

We present a practical model checking method that can detect deadlocks, local assertion violations, and many forms of resource leaks in MPI programs, without requiring users to hand-model their code as required in [5, 6].

The authors thank Salman Pervez for the initial implementation of ISP and Robert Palmer for laying down the foundations for our work.

References

- [1] The Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. <http://www.mpi-forum.org/>.
- [2] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp, Formal Verification of Programs that use MPI One-sided Communication, *EuroPVM/MPI*, 30–39, 2006.
- [3] Salman Pervez, Robert Palmer, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp, Practical Model Checking Method for Verifying Correctness of MPI Programs, *EuroPVM/MPI*, 344–353, 2007.
- [4] E.M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.
- [5] G. J. Holzmann, *The SPIN Model Checker*, Addison-Wesley, 2004.
- [6] Stephen F. Siegel, *Model Checking Nonblocking MPI Programs*, In *VMCAI*, 44–58, LNCS 4349, 2007.
- [7] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby, Distributed Dynamic Partial Order Reduction based Verification of Threaded Software, In *SPIN*, 58–75, LNCS 4595, 2007.
- [8] Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby, Semantics Driven Dynamic Partial-Order Reduction of MPI-based Parallel Programs, In *Parallel and Distributed Systems - Testing and Debugging PADTAD-V*, London, UK, July 2007.
- [9] Richard Vuduc, Martin Schulz, Dan Quinlan, Bronis de Supinski, and Andreas Saebjornsen, Improved Distributed Memory Applications Testing By Message Perturbation, *Parallel and Distributed Systems: Testing and Debugging (PADTAD - IV)* 2006.
- [10] Madan Musuvathi and Shaz Qadeer, Iterative context bounding for systematic testing of multithreaded programs, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 446–455, 2007.
- [11] Patrice Godefroid, *Model Checking for Programming Languages using VeriSoft*, In *POPL*, 174–186, 1997.
- [12] Cormac Flanagan and Patrice Godefroid, Dynamic partial-order reduction for model checking software, In *POPL*, 110–121, 2005.

¹ http://www.cs.utah.edu/formal_verification/ppopp08-isp/.