Automatic Formal Verification of MPI-Based Parallel Programs

Stephen F. Siegel
Verified Software Laboratory, Department of Computer and Information Sciences, University of Delaware
siegel@cis.udel.edu

Timothy K. Zirkel
Verified Software Laboratory, Department of Computer and Information Sciences, University of Delaware
zirkeltk@udel.edu

Abstract

The Toolkit for Accurate Scientific Software (TASS) is a suite of tools for the formal verification of MPI-based parallel programs used in computational science. TASS can verify various safety properties as well as compare two programs for functional equivalence. The TASS front end takes an integer $n \geq 1$ and a C/MPI program, and constructs an abstract model of the program with n processes. Procedures, structs, (multi-dimensional) arrays, heap-allocated data, pointers, and pointer arithmetic are all representable in a TASS model. The model is then explored using symbolic execution and explicit state space enumeration. A number of techniques are used to reduce the time and memory consumed. A variety of realistic MPI programs have been verified with TASS, including Jacobi iteration and manager-worker type programs, and some subtle defects have been discovered. TASS is written in Java and is available from `http://vsl.cis.udel.edu/tass` under the Gnu Public License.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods

General Terms Verification

Keywords Symbolic execution, MPI, message-passing, debugging, verification

1. Motivation. Developing correct, portable MPI-based parallel programs is difficult for several reasons. First, traditional testing methods can only sample a small portion of a program's parameter and input space. Second, there are many sources of nondeterminism arising from MPI, e.g., how statements from different processes are interleaved in time, the buffering policy of the MPI implementation, and the use of "wildcard" receives. Often, a program may appear to run correctly on one machine, but when ported to a new platform which resolves the nondeterminism differently, a defect is revealed. These defects may manifest themselves as deadlocks or as results which are incorrect or differ unexpectedly from execution to execution. Such defects are often difficult to identify and isolate.

2. Symbolic Execution. TASS uses *symbolic execution* [3, 4] to deal with the issue of the input space. The user may insert pragmas into the C source to identify input variables, e.g.,

```
#pragma TASS input { N > 1 && N <= 10 }
int N;
```

(This pragma is accompanied by a predicate which restricts the input space.) TASS initially assigns a unique *symbolic constant* $(X_1, X_2, \ldots)$ to each such variable. As TASS "executes" the program, operations on these values result in more complex symbolic expressions.

A branch is treated as a nondeterministic choice, and a boolean-valued *path condition* variable *pc* is used to record the choices made. For example, if the *true* branch is chosen at a branch on condition `a>0`, when the value of `a` is the symbolic expression $X_1 + 1$ and the value of *pc* is $X_2 < 0$, then the value of *pc* becomes $X_2 < 0 \wedge X_1 + 1 > 0$. Automated theorem proving techniques are used to determine if *pc* becomes unsatisfiable, which indicates the current path is infeasible and need not be explored. TASS dispatches most of the proofs itself, but when it cannot, it invokes CVC3 [1]. If CVC3 cannot produce a definitive answer, TASS logs a *possible* violation. TASS analysis is *safe*: if it says a property holds, the property holds (within the specified bounds). Spurious error reports are possible, but rare, in our experience.

The ability to reason about the infinite input space is one of the main advantages over traditional testing and dynamic verifiers such as ISP [10], which analyze executions for one input vector at a time.

3. State enumeration. A *symbolic state* assigns a symbolic expression to each variable in the program, including *pc*. It represents a set of concrete states: any assignment of concrete values to symbolic constants that satisfies *pc* results in a concrete state in the set.

TASS uses explicit state enumeration techniques to deal with the nondeterminism arising from MPI (as well as from symbolic execution branches described above). An explicit representation of the state is used, and the set of all reachable states is explored with a depth-first search. Typically, the user places bounds on certain variables to ensure the state space is finite (or reasonably small).

A number of techniques are used to reduce the memory footprint. The state is represented hierarchically: at the highest level there are components for the input and output variables, the state of each process, and the set of buffered messages. A process state consists of the state of each global variable and a call stack. The call stack is a sequence of frames. Each frame has the value of all local variables and the current location within a function body. The lifecycle of a state component has three phases: mutable, immutable, and canonic (flyweighted). These design choices facilitate maximal sharing between states on the stack or seen set, while still allowing fast modifications when a component is not shared.

4. Properties checked. TASS checks a number of safety properties as it explores the state space: absence of (absolute or potential) deadlocks, buffer overflows (from pointer arithmetic, array indexing, etc.), reading uninitialized variables, division by zero, memory leaks, and assertion violations; the type and size of a message received with `MPI_Recv` are compatible with the receive buffer/type; proper use of `malloc` and `free`, `MPI_Init`, `MPI_Finalize`, and so on. In all cases, violations are logged and the search continues

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
ACM 978-1-4503-0119-0/11/02.

program	bounds	nprocs	time (s)	states
adder	$n \leq 200$	30	443	411044
diffusion	$n_x \leq 100 \wedge n_t \leq 2$	50	4653	5788608
matmat	$l \leq 2 \wedge m \leq 2 \wedge n \leq 10$	7	13157	151752013
matmat	$l \leq 2 \wedge m \leq 2 \wedge n \leq 10$	11	242	10543295
integrate	$intervals = 12 \wedge tol = 0.01$	6	3788	3511123
laplace	$n_x \leq 4 \wedge n_y \leq 12 \wedge n_t \leq 30$	5	30911	3384271
laplace	$n_x \leq 4 \wedge n_y \leq 30 \wedge n_t \leq 3$	20	1787	875577

Figure 1. TASS performance verifying functional equivalence

after adjustments to the path condition, in order to report as many errors as possible from a single search. If no errors are reported then it is guaranteed that all the safety properties hold on all executions with the given processor count and within the specified bounds.

5. Simplification. Two symbolic states are *equivalent* if they represent the same set of concrete states. The ability to recognize equivalence can greatly reduce the search space. This can often be accomplished by transforming symbolic expressions into a canonical form. For example, TASS transforms every real-valued expression into a quotient p/q, where p and q are polynomials in *primitive* expressions. (A *primitive expression* is any non-concrete expression that is not the result of `+`, `-`, `*`, or `/`; symbolic constants and array-read and array-write expressions are examples.)

TASS performs other equivalence-preserving state transformations. For example, each equation occurring as a clause in *pc* may be considered a linear expression $\sum_i a_i x_i$, where a_i is a concrete real and x_i is a monomial in the primitives. Gaussian elimination is performed on the matrix of coefficients resulting from these equations to determine if any monomials can be reduced to concrete values. For example if *pc* is $X_1 + X_2 = 3 \wedge X_1 - X_2 = 1$, then 2 will be substituted for X_1 wherever X_1 occurs in the state, 1 will be substituted for X_2, and *pc* will become *true*.

6. Reduction. Exploring every possible interleaving is not necessary to verify the properties in our list. For example, for programs which restrict their use of MPI to a certain safe subset (which excludes `MPI_ANY_SOURCE`), it can be shown that it is safe to consider a single interleaving [6, 7]. TASS uses the "urgent" partial order reduction algorithm, an optimization which is safe even in the presence of `MPI_ANY_SOURCE` [5]. For programs in the safe set, it still explores only a single interleaving; otherwise, whenever some process is at an any-source receive, some subset of the possible interleavings must be explored. This can reduce the number of states explored dramatically.

7. Comparative symbolic execution. Pragmas can be inserted into the source code to identify output as well as input variables. Together, these annotations specify an input set X and output set Y, and the program may be considered a function $f\colon X \to Y$. Two programs with the same X and Y may be compared to see if they define the same function. This is extremely useful in computational science, where developers often start with a simple sequential version of an algorithm, and transform it by hand by adding layers of optimizations and parallelism before arriving at the production code. TASS gives the developers the ability to check that the original and final codes are functionally equivalent. The technique used is *comparative symbolic execution* [9]. TASS constructs a model in which the two programs read the same input and run concurrently, and checks the assertion that the outputs agree at termination. (KLEE [2] is another symbolic execution tool which has been used to verify functional equivalence in some cases; however it does not apply to parallel programs.)

8. Collective assertions. Assertions are an important tool for developing reliable sequential programs. However, assertions within a single process are unable to capture important properties of the interactions between different processes. To remedy this, TASS gives users the ability to use *collective assertions* [8]. A single collective assertion comprises a set of locations in each process and an expression on the global state. The semantics are defined as follows: whenever control in a process reaches one of the locations, a "snapshot" of the local state of that process is sent to a coordinator; once a snapshot has been received from each process, the expression is evaluated on the global state formed by uniting the snapshots.

9. Collective loop invariants. The infinite-state problem arising from loops can be surmounted in some cases using an invariant, which the user can associate to a loop using another pragma. The invariant is a form of collective assertion which is not only checked by TASS, but is used to simplify the state of a process after each loop iteration. This results in a possible loss of precision (i.e., possible spurious error reports) but is still safe. This technique applies to loops that cut across multiple processes, and even to comparative symbolic execution.

10. Experiments. We have applied TASS to small but non-trivial MPI programs. Examples include (1) a program to sum the elements of a block-distributed array, (2) a block-distributed solver for the 1d-diffusion equation, (3) a manager-worker matrix multiplication routine, (4) a 1d adaptive mesh numerical integrator, and (5) a column-distributed 2d-Laplace solver using Jacobi iteration. Results for verifying the equivalence of various configurations of these with the corresponding sequential/specification program are shown in Fig. 1. More extensive case studies, and further extensions and optimizations, are in progress.

Acknowledgments

Supported by NSF grants CCF-0953210 and CCF-0733035 and the University of Delaware Research Foundation.

References

[1] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer.

[2] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[3] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *TACAS 2003*, volume 2619 of *LNCS*, pages 553–568.

[4] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[5] S. F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In R. Cousot, editor, *VMCAI 2005*, volume 3385 of *LNCS*, pages 413–429.

[6] S. F. Siegel and G. S. Avrunin. Modeling wildcard-free MPI programs for verification. In *PPoPP '05*, pages 95–106. ACM, 2005.

[7] S. F. Siegel and G. S. Avrunin. Verification of halting properties for MPI programs using nonblocking operations. In F. Cappello, T. Hérault, and J. Dongarra, editors, *Euro PVM/MPI 2007*, volume 4757 of *LNCS*, pages 326–334. Springer, 2007.

[8] S. F. Siegel and T. K. Zirkel. Collective assertions. In R. Jhala and D. Schmidt, editors, *VMCAI 2011*, volume 6538 of *LNCS*, pages 387–402.

[9] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM TOSEM*, 17(2):Article 10, 1–34, 2008.

[10] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal verification of practical MPI programs. In *PPoPP 2009*, pages 261–270. ACM.