Finite-State Verification for High Performance Computing

George S. Avrunin*
Department of Mathematics
and Statistics
University of Massachusetts
Amherst, MA 01003
avrunin@math.umass.edu

Stephen F. Siegel*
Department of Computer
Science
University of Massachusetts
Amherst, MA 01003
siegel@cs.umass.edu

Andrew R. Siegel
Mathematics and Computer
Science Division
Argonne National Laboratory
Argonne, IL 60439
siegela@mcs.anl.gov

1. INTRODUCTION

A glance at the list of the world's most powerful computing systems (`top500.org`) reveals that high performance computing has become practically synonymous with parallel computing. Yet parallel programs are notoriously difficult to get right. It is hard enough to verify that an ordinary sequential program computes what it is intended to compute, and parallelism introduces an entirely new layer of complexity. Moreover, parallel programs can behave nondeterministically, in the sense that they can produce different results when run with different numbers of processors, or when run on different platforms, and sometimes even when run twice on the same platform. Experience has shown that just to detect or reproduce these problems—let alone to pinpoint their causes and correct them—can be extremely time-consuming and labor-intensive.

The field of *finite-state verification* (FSV) offers a variety of methods for dealing with precisely these issues. But there are some significant differences between the HPC domain and those areas where FSV methods have been successfully applied in the past, so it is not obvious that FSV techniques can be useful in the HPC world.

In this paper, we discuss some preliminary results in applying FSV techniques to high performance parallel codes, with a particular emphasis on *scientific* programs that employ the widely-used *Message Passing Interface* (MPI) [7, 8]. These results suggest that such techniques may have significant potential for improving both the productivity of developers of parallel scientific programs and the quality of those programs. We also briefly sketch some of the research issues that must be addressed to achieve that potential.

2. SCIENTIFIC COMPUTATION AND MPI

Most parallel scientific programs rely on *message-passing* for inter-process communication. The basic ideas of this paradigm have been around since the late 1960s, and by the early 1990s, several different and incompatible message-passing systems were being used to develop significant applications. The desire for portability and a recognized standard led to the creation of MPI [7, 8], which defines the precise syntax and semantics for a library of functions for writing message-passing programs in a language such as C or Fortran. Since that time, a number of high-quality proprietary and open-source MPI implementations have become available on a wide variety of platforms, and MPI has become the *de facto* standard for parallel scientific software. Yet, while its adoption has made for the possibility of truly portable parallel programs, MPI alone cannot completely solve the problems that are inherent in the very nature of parallel computation.

Our experience in building scientific software (for example, with the ASCI FLASH project, the Argonne Petascale Computing Applications Group, and most recently the Applications Working Group of the BlueGene/L Consortium) has taught us that in real-world scientific applications a number of serious problems occur with surprising frequency: (1) deadlock, particularly when porting to new platforms or rerunning code with OS or system library upgrades; (2) results that are not independent of the number of processors used (in cases where there is reason to believe that they should be); (3) results that can not be reproduced on identical processor configurations; and (4) performance that is hampered by unnecessary operations at the application programmer level, such as redundant barriers or ghost-cell fills.

In all cases the possible causes are well understood: race conditions, implementation-specific MPI behavior (e.g., code which is not buffer-safe), and in some cases even bugs in the MPI implementation. However, tracking down the specific problem and getting the code to work or give believable results on the new machine can often be nearly impossible. We have seen many cases where bugs manifest themselves only once in every ten or more executions, or only on very large processor configurations that are difficult to access, or possibly only after very long and unpredictable integration times. Practically speaking, if we cannot find a way to reproduce a problem with regularity, there is little or nothing we can do with traditional tools and approaches.

Our alternative has been to reason informally about our parallel algorithms, deliberating questions such as "Is this algorithm buffer-safe?", "Is there a race condition?", or "Is there any way it can deadlock?". Unfortunately, this is much more difficult than it appears at first glance, and nearly impossible for relatively complex algorithms. We once spent

*Research partially supported by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement number DAAD190110564.

SE-HPCS'05, May 15, 2005, St. Louis, Missouri, USA
Copyright 2005 ACM 1-59593-117-1/05/0005 ...$5.00.

an entire meeting debating whether a single barrier could be removed. Although we finally agreed its removal was safe, a year later its absence turned out to be the cause of a race condition that had manifested itself on a new platform. It is not uncommon to spend several worker-months on portability issues of this type.

We have often been forced out of desperation to rewrite algorithms from scratch, in the blind hope that whatever was causing the problem will just "disappear." In a practical sense this has been successful. Clearly, though, it is not a reliable or adequate solution. What is desperately needed is a set of tools that can analyze the software and diagnose these types of problems *a priori* and independently of any execution of the code. This would result in tremendous increases in efficiency in the development of parallel scientific software, as well as in increased confidence in the correctness of that software.

3. FINITE-STATE VERIFICATION

The most widely used method for checking that a program meets its requirements is testing, i.e., executing the program on a set of inputs and examining the results. While testing is certainly a key part of any software development effort, it has a number of significant drawbacks. First, it is extremely expensive. The best estimates indicate that testing represents at least 50% of the cost of developing typical commercial software, and the percentage is certainly higher for safety- or mission-critical projects where correctness is vital. Second, even with the large amount of effort invested in testing, it is usually infeasible to test more than a tiny fraction of the inputs that a program will encounter in use. Thus, testing can reveal bugs, but it cannot show that the program behaves correctly on the inputs that are not tested. Finally, the behavior of concurrent programs, including most MPI programs, typically depends on the order in which events occur in different processes. This order depends in turn on the load on the processors, the latency of the communication network, and other such factors. A concurrent program may thus behave differently on different executions with the same inputs, so getting the correct result on a test execution does not even guarantee that the program will behave correctly on another execution with the same input.

Another approach is to construct a logical theory describing the program and to use automated theorem-proving tools such as ACL2 [6, 5] and PVS [10] to attempt to show that the program behaves correctly on all executions. While techniques based on this approach have achieved some notable successes, they typically use the automated theorem prover as a "proof-checker" at least as much as a "proof-finder," and require a great deal of effort and guidance by experts in mathematical logic. In practice, such techniques are far too expensive to apply routinely in software development and are reserved for small components of the most safety-critical systems.

The third main approach involves building a finite model that represents all possible executions of the program and using various algorithmic methods to determine whether particular requirements for the program hold in the model. For instance, the occurrence of deadlock during an execution of the program might be represented in the model by the reachability of certain states from the start state, and algorithmic methods for exploring the states of the model could determine whether or not deadlock was possible. These *finite-state verification* techniques are less powerful but much more automated than theorem-proving approaches, and, unlike testing, can give results about all possible executions of the program. (The term "model checking" is sometimes used to refer to this class of methods, although in its original technical meaning it refers to only a subset of them.)

There are two main drawbacks to FSV techniques. The first is that FSV really obtains results about a model of the program, rather than the program itself, and the results are therefore only as good as the model. Constructing good models that are small enough to be tractable can be very difficult, although recent research in automatic model extraction is producing useful tools that automatically create suitable models directly from source code. The second drawback is the *state-space explosion problem.* For concurrent programs, the number of states the program can reach is, in general, exponential in the number of processes, and these states must be represented in the model in some fashion. Indeed, almost all the questions one would want to answer about a concurrent program (e.g., does it deadlock, does a particular communication ever occur, etc.) are known to be at least *NP*-hard. Moreover, even toy concurrent programs can have 10^{100} reachable states, so that methods which naively examine each possible state are completely infeasible. There has, however, been an enormous amount of research on techniques to combat this problem, including ways to reduce the number of states that must be examined to check the property of interest, to manipulate sets of states at once, to make use of integer linear programming techniques, etc. (Of course, the complexity results make it clear that no single technique will work in all cases.) A number of tools have been developed that take advantage of these techniques. The range of applicability of FSV techniques is thus steadily increasing, and we believe it has reached the point where FSV techniques can be successfully applied to complex scientific parallel software systems.

4. EXPLORING THE APPLICABILITY OF FSV TO MPI PROGRAMS

The domain of high-performance MPI programs differs in significant ways from the domains to which FSV techniques have traditionally been applied. Moreover, many features specific to the MPI domain appear to pose deep challenges to finite-state approaches. For example, typical MPI programs manipulate enormous amounts of floating point data, which makes the construction of appropriate finite-state models particularly difficult. In addition, the MPI Standard itself leaves open many choices to the MPI implementation. In order to verify that an MPI program performs correctly under any legal implementation, all allowable behaviors of the implementation must somehow be represented in the model. This can be a significant source of state-explosion above and beyond the usual ones that arise in the verification of concurrent programs. A third issue is the nature of the properties that one wishes to verify. Freedom from deadlock is certainly a desirable property of an MPI program, and has also been studied extensively in the FSV literature in other domains. However, other properties that developers of scientific software would likely want to verify, such as the correctness of floating point calculations, are quite different from the kinds of properties that FSV techniques have typically been called upon to check.

These are just some of the reasons why it is not clear, *a priori*, that FSV techniques can be successfully employed in the MPI domain. Because of this, we have carried out several preliminary investigations of these issues, some of which we summarize below. The results of these investigations have led us to conclude that, in fact, FSV techniques have enormous potential to help solve many problems in the MPI domain, though significant work will be required to adapt the techniques to that domain.

4.1 A test case

Our initial exploration of the applicability of FSV to MPI programs involved a detailed study of a small scientific test program called Diffusion2d. Diffusion2d is a parallel program that simulates the evolution in time of a discretized function u defined on a 2-dimensional domain and governed by the Diffusion Equation. It is the "teacher's solution" to a programming project for a course in parallel programming (taught by A. Siegel at the University of Chicago). Though quite simple, Diffusion2d has many features common in scientific simulation programs: the physical domain is divided into a "grid" in which one processor is responsible for each section; each processor maintains a number of "ghost-cells" that mirror the contents of cells on neighboring processors; and there is a somewhat complicated routine to coordinate the writing of the data to disk, in order to ensure that the cells are written in the proper order.

We first formulated several properties that we expected to hold for all executions of Diffusion2d. In addition to freedom from deadlock, these included several claims concerning the order of occurrence of certain events during the execution of the program. For example, the value $u(x, n)$ of the evolving function u at a grid point x and time n depends on the values of $u(y, n-1)$ for certain grid points y. It is therefore essential to the correctness of the program that when $u(x, n)$ is calculated, the values used in the calculation are those of u at time $n-1$, and not, say, at time n or $n-2$. The programmer was confident that this was ensured because the program satisfied the following *lockstep* property: no $u(x, n)$ is calculated until all $u(y, n-1)$ (for all grid points y) have been.

We applied two FSV tools, Spin [4] and INCA [1, 12], that use quite different approaches. For both tools, we had to create the finite-state model of the program by hand, as there are currently no automated tools for creating models of MPI programs. This required a close reading of the code and discussions with its author. In creating the model, we dealt with the floating point problem by simply abstracting away the floating point data altogether, as we realized that the properties we intended to check did not depend in any way on the actual values of that data. Moreover, these models had to represent not only all possible behaviors of Diffusion2d, but also of the MPI implementation. For example, the MPI Standard states that for each `MPI_Send` statement encountered during execution, the implementation may choose whether to execute the send synchronously or asynchronously. In our Spin model, for instance, this was represented by a non-deterministic choice made each time a send takes place.

We were able to verify all the properties for a number of different configurations of Diffusion2d until we reached the lockstep property described above with a configuration involving a 1 × 4 grid, for which the tools spelled out an execution in which one processor had begun calculation of u at time 2, while another had not yet begun the calculation for time 1. Further exploration showed that the longer the grid, the further apart in time two processors could become. It turned out that this violation did not mean the program was incorrect, because all that was really required for correctness was that the calculation on neighboring processes had been completed, and this weaker property was indeed verified by the tools. Nevertheless, the violation of the stronger property was a surprise to the programmer, which highlights the fact that even simple parallel programs are extremely difficult to reason about informally.

The limitations due to state-explosion were already clear with this simple program. INCA could verify freedom from deadlock in configurations up to 8 × 8 under the assumption that all communication was synchronous. This was significantly larger than the largest grid Spin could handle (4 × 3), even under the same assumption. (As we will explain below, the verification under this assumption implies the general case.) The lockstep properties were checked in configurations up to 11 × 11 (under no additional assumptions). These grid sizes are still much smaller than those of real programs, but evidence from the application of FSV techniques to other kinds of software suggests that problems are usually exposed by verification of relatively small configurations. (This is quite different from the case with testing, where the small size may make it difficult to trigger particular pathological patterns of behavior. The key is that FSV takes into account all possible executions of the system.) A more detailed discussion of these results can be found in [15].

4.2 Theoretical results

Through our work on Diffusion2d it became clear that the asynchronous aspect of MPI message passing was going to be a major problem for FSV techniques. As mentioned above, the MPI implementation may choose between synchronous or asynchronous transmission of messages each time it encounters a send. This can be a major source of state explosion for two reasons: first, because of the need to model all the binary choices (synchronous vs. asynchronous), and second, because of the need to model all possible states of the message queues resulting from asynchronous sends. As an example of the impact this can have, in verifying freedom from deadlock for a Diffusion2d configuration with a 3 × 2 grid, Spin required 185 MB of memory. The same verification, when restricted to purely synchronous communication, required only 2 MB.

Unfortunately, it is not in general true that a property that holds on all synchronous executions of an MPI program must also hold for all executions of that program. However, it seemed reasonable to us that for *certain properties* and for *certain classes* of MPI programs, such a restriction might be justified. Theoretical results that justified such a restriction could be enormously valuable in reducing the work required to verify MPI programs, for the reasons given in the paragraph above, and so we undertook a theoretical investigation along these lines.

The results of that investigation [13, 14] revealed that there are in fact many ways to ameliorate (and sometimes to eliminate altogether) the state-explosion problem for certain classes of MPI programs and properties. The main assumption on the programs is that they contain no *wildcard re-*

ceives, i.e., no use of MPI_ANY_SOURCE or MPI_ANY_TAG. For this class of programs (which includes Diffusion2d), we showed, for example, that if the program is deadlock-free for all synchronous executions then it must necessarily be deadlock-free for all executions. This theorem confirms the intuition behind the practice, popular among MPI programmers, of checking a program for deadlocks by substituting synchronized sends for every standard send in the program code. It also justified our verification of freedom from deadlock for Diffusion2d under the assumption that all communication was synchronous.

Another result from our theoretical investigation answers a question about *determinism* for a large class of wildcard-free MPI programs (which again includes Diffusion2d). This theorem implies that any program in the class must always obtain the same result from a given input, no matter what choices are made by the MPI implementation. As we have noted, this is a very important property in the scientific computation domain. Indeed, our results about determinism have already been applied to a component from the FLASH project. The code in question implements a block redistribution algorithm for an adaptively refined mesh, which requires that blocks of data be redistributed periodically among the processes according to a complex communication pattern. The original redistribution routine, which was known to deadlock in some scenarios, was replaced with a routine written entirely in our restricted subset of MPI. Our results then made it relatively easy to establish freedom from deadlock for the new version.

Other theorems on wildcard-free MPI programs, such as one concerned with the impact of removing barriers, are discussed, with examples, in [14].

4.3 MPI-optimized verification

The theoretical results discussed above apply only to wildcard-free MPI programs. However, there are many parallel algorithms that depend in an integral way on the use of wildcards (e.g., a program employing a master-slave architecture in which the master uses a wildcard receive to receive a result from whatever slave has finished its task). Methods that could not be efficiently applied to such programs would have only a very limited usefulness.

In [11], we introduced an efficient method, called the Urgent algorithm, for verifying certain properties of MPI programs that may contain wildcard receives. The properties include freedom from deadlock, as well as any other assertion on the state of the program at termination.

The Urgent algorithm improves upon the methods described in §4.1 in two ways. First, it does not require explicitly modeling all possible synchronous vs. asynchronous choices made by the MPI implementation. Second, it uses a “partial order reduction” technique that has been tailor-made to take advantage of the MPI semantics to drastically reduce the number of states that need to be explored.

Interestingly, Spin also implements a partial order reduction strategy, but in all of the examples we have studied that strategy made no reduction at all in the number of states explored by Spin. The reason is that’s Spin’s *generic* algorithm knows nothing about the semantics of MPI, and so does not have the knowledge to take advantage of the MPI-specific features that the Urgent algorithm exploits. This provides yet another example of the need for FSV techniques that are “tailor-made” for MPI programs.

5. RESEARCH DIRECTIONS

In conclusion, there is now a growing body of evidence that suggests that FSV techniques can be used to answer a variety of important questions about MPI programs. However, significant modifications of the generic FSV algorithms must be studied and developed in order for those algorithms to be effective in the MPI domain. Furthermore, novel finite-state techniques will need to be explored in order to deal with the kinds of properties that are likely to be of interest to computational scientists.

While our initial explorations have shown the promise of applying FSV to scientific software, they have barely scratched the surface of what is possible. There are many more FSV tools and techniques to be explored, and much work to be done to adapt these to the scientific domain. We sketch here some directions for research that would help bring the benefits of FSV to developers of large-scale scientific software.

5.1 Model extraction and construction

In order to apply FSV techniques to a program, it is first necessary to construct an appropriate finite model of the program that captures all possible executions. Our work to date has been based on manually constructed models. Constructing these models requires a deep understanding of the system being modeled and the modeling notation being used, and is both labor-intensive and error-prone. So investigating the special problems of model extraction and construction for MPI-based codes is an important prerequisite for the application of FSV techniques. Some particular research problems include the following:

- **Exploration of different translations.** There are various ways to represent the MPI primitives in a finite-state model, and previous research has shown that the performance of an FSV tool can be extremely sensitive to the exact way in which the program code is translated into a model. Often, apparently small changes in the translation result in enormous improvements in tool performance. For each tool, we must therefore try various translations to determine which work best.

- **Development of an automatic MPI model extractor.** There has been a considerable amount of recent research into *automatic model extraction* for concurrent software, especially for programs written in languages like C and Java, and much of it should carry over to MPI programs.

- **Explore the use of abstraction and related techniques.** *Abstraction* is one of the strongest weapons in the fight against state-explosion. As an example, suppose we have two floating point variables `x` and `y` in a program. Simply to represent all possible pairs of values of these variables would require an astronomical number of states (2^{128} states, to be precise, assuming 64 bits are required to represent a floating point variable). In our model, we might instead replace them with variables that take values in the set {`POS`, `NEG`, `ZERO`} and thereby keep track only of whether the original variables were, respectively, positive, negative, or zero. To represent all possible pairs of values of the abstracted variables takes only 9 states. Of course, this model will contain less information than the original program, but *for a particular property* this might be all the information that is needed for verification. The art of abstraction is to find the *appropriate* abstractions for a given program and property. The abstractions must result in a model small enough to make verification tractable, but with enough information to allow the verification algorithm to arrive at a conclusive result. Finding appropriate abstractions will be particularly important for scientific programs, which generally deal with enormous amounts of data.

5.2 FSV techniques for MPI programs

As we have described above, the practical application of FSV techniques to significant MPI programs will require adaptation and development of existing FSV approaches. Some of the directions that we believe should be explored include:

- **Explore different FSV tools.** We have experimented with SPIN and INCA, but there are a number of other FSV tools based on different approaches. Moreover, the development of FSV tools continues rapidly, and only experimentation will tell, for example, how effective a new approach like the heuristic search implemented in HSF-SPIN[3] will be on scientific/MPI programs, or whether a particular tool is especially well-suited for checking a particular class of properties. It is therefore important to explore the application of a variety of existing and new FSV tools to MPI programs.

- **Extend the theoretical results.** The results of our theoretical investigation have already proven extremely useful. However, they currently cover only the basic blocking point-to-point MPI functions and the MPI collective functions. Extending these results to the non-blocking functions will be necessary if FSV tools are to be applied to realistic scientific programs.

- **Explore new and improved FSV techniques.** We expect that typical features of MPI-based scientific programs will allow us to take advantage of special-purpose FSV techniques, or optimizations of existing techniques. For instance, as described in the previous section, use of a particular subset of the MPI primitives allows the FSV tool to ignore buffering in checking some important properties like absence of deadlock, and the Urgent algorithm also takes advantage of particular features of the MPI communication constructs. It is important to look for additional FSV techniques that can take advantage of domain-specific features.

6. REFERENCES

[1] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6:97–123, January 1995.

[2] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology*, 13(4):359–430, 2004.

[3] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In M. B. Dwyer, editor, *Proceedings of the 8th Int. SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 57–79, Toronto, May 2001. Springer-Verlag.

[4] G. J. Holzmann. *The* SPIN *Model Checker.* Addison-Wesley, Boston, 2004.

[5] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[6] M. Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Trans. Softw. Eng.*, 23(4):203–213, 1997.

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface standard, version 1.1. `http://www.mpi-forum.org/docs/`, 1995.

[8] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/`, 1997.

[9] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

[10] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[11] S. F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005, Paris, January 17–19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 413–429, 2005.

[12] S. F. Siegel and G. S. Avrunin. Improving the precision of INCA by eliminating solutions with spurious cycles. *IEEE Trans. Softw. Eng.*, 28(2):115–128, 2002.

[13] S. F. Siegel and G. S. Avrunin. Modeling MPI programs for verification. Technical Report UM-CS-2004-75, Department of Computer Science, University of Massachusetts, 2004.

[14] S. F. Siegel and G. S. Avrunin. Modeling wildcard-free MPI programs for verification. To appear in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[15] S. F. Siegel and G. S. Avrunin. Verification of MPI-based software for scientific computation. In S. Graf and L. Mounier, editors, *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 286–303. Springer-Verlag, 2004.