Analyzing BlobFlow: A Case Study Using Model Checking to Verify Parallel Scientific Software

Stephen F. Siegel[1] and Louis F. Rossi[2]

[1] Verified Software Laboratory, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716, USA, siegel@cis.udel.edu
[2] Department of Mathematical Sciences, University of Delaware, Newark DE 19716, USA, rossi@math.udel.edu

Abstract. Model checking techniques are powerful tools for the analysis and verification of concurrent systems. This paper reports on a case study applying model checking techniques to a mature, MPI-based scientific program consisting of approximately 10K lines of code. The program, BlobFlow, implements a high order vortex method for solving the two-dimensional Navier-Stokes equations. Despite the complexity of the code, we verify properties including freedom from deadlock and the functional equivalence of sequential and parallel versions of the program. This has led to new insights into the technology that will be required to automate the modeling and verification process for complex scientific software.

1 Introduction

Over the past several years, there has been increasing interest in using *model checking* to debug and verify parallel scientific programs. Specific techniques vary, but usually involve three tasks: (1) an abstract model of the program is constructed, (2) correctness properties of the model are expressed in a formal language, and (3) automated algorithmic techniques are used to exhaustively explore all possible states of the model while checking that the properties hold. The model checking tool will either report that the properties always hold, or provide an explicit counterexample in the form of an execution trace of the model, greatly facilitating debugging.

This approach has several advantages over traditional testing and debugging techniques. The first is that it exhaustively explores all possible executions of the model, examining all allowable interleavings of statements from different processes, all choices available at a wildcard receive, and so on. These kinds of choices are difficult to explore at all by testing, let alone exhaustively.

One of the alleged shortcomings of the model checking approach is that it requires relatively small bounds on parameters such as the number of processes, the size of the input, and so on. This is not as serious a limitation as it first appears, because program defects almost always manifest themselves for relatively small values of these parameters. This point is often misunderstood by developers, who have seen many problems that only occur for very large process counts or inputs. A common example is an MPI program with the potential

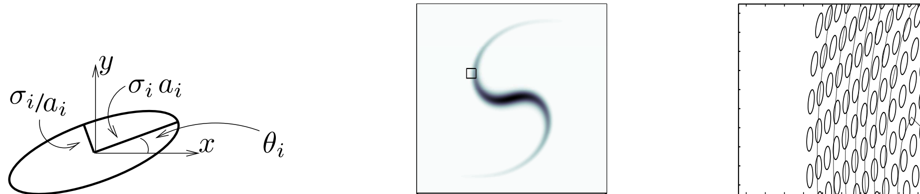

Fig. 1. Left: A schematic diagram of the geometry of an elliptical Gaussian basis function. Middle: a sample ECCSVM computation. Right: blowup of particles in the boxed area.

to deadlock: the program may run normally on small configurations if the MPI implementation chooses to buffer certain messages, but when the process count or memory requirements exceed some threshold, the implementation may choose to synchronize message delivery, revealing the deadlock. A model checker, however, will explore the possibility of forced synchronization in all states where it is permitted by the MPI Standard, and the failure will be detected in even the smallest configurations. Another example is an MPI program that may fail when the number of outstanding communication requests generated by the program exceeds some large bound. With model checking, the bound on the number of allowable requests is a parameter which can be given a small value, so the same failure will be detected for a small configuration of the program.

In fact, the ability of model checking to discover defects in small configurations is one of its most significant advantages over traditional debugging techniques. If one can only reproduce a failure using 1000 processes, and the resulting trace involves thousands of execution steps, analyzing the trace with a standard debugger can be difficult or impossible. The ability of model checkers to find small (even minimal) counterexamples is an immense advantage for debugging.

To date, most of the applications of model checking to scientific computing have involved only small example programs (e.g., [1–3]). The goal of the project reported on here was to determine whether model checking techniques could be successfully applied to a production code used in actual scientific research. The code we chose is L. Rossi's computational fluid dynamics program BlobFlow [4–6]. BlobFlow has been actively developed and used over the past 7 years to explore fluid flow phenomena and novel simulation algorithms. It consists of approximately 10K lines of C code and includes both a parallel (MPI-based) version and a sequential version. To the best of our knowledge, it is the largest parallel scientific application to have been successfully verified with model checking techniques.

The model checker used in this study is Mpi-Spin [7,8], an extension to the standard model checker Spin [9]. Mpi-Spin adds to Spin's input language a large number of primitives corresponding to the types, constants, and functions comprising MPI. It also incorporates a precise model of the semantics of the MPI operations, based on the MPI Standards. Though there are no tools to automatically translate MPI programs into the input language for Mpi-Spin,

```
 1 procedure ECCSVM(t_0, t_f, Δt: double; numElements: int; elements: Element[]) is
 2 |  vel : Vector[MaxElt];
 3 |  t ← t_0;
 4 |  while t ≤ t_f do
 5 |  |  for j ← 0 to numElements do computeVelocity(j, elements, vel);
 6 |  |  integrate(numElements, elements, vel);
 7 |  |  output(t, numElements, elements);
 8 |  |  t ← t + Δt;
 9 procedure computeVelocity(numElements: int; elements: Element[]; vel: out Vector[]) is
10 |  mpcoef : double[MpSize]; i, j, p : int;
11 |  for i ← 0 to MpSize − 1 do mpcoef[i] ← 0;
12 |  for i ← 0 to numElements − 1 do
13 |  |  for p ← 0 to PMax − 1 do
14 |  |  |  j ← f_1(i, p, elements);
15 |  |  |  mpcoef[j] ← mpcoef[j] + f_2(i, p, elements);
16 |  for i ← 0 to numElements − 1 do vel[i] ← f_3(i, elements, mpcoef);
```

Fig. 2. ECCSVM algorithm, sequential version

this language support makes manual translation much more straightforward. We used Mpi-Spin to verify two important properties of BlobFlow: (1) freedom from deadlock, and (2) the functional equivalence of the sequential and parallel versions.

2 ECCSVM and BlobFlow

The Elliptical Corrected Core Spreading Vortex Method (ECCSVM) [4, 5] is an algorithm for computing the motions of incompressible gases and liquids in two dimensions. The algorithm falls under the general category of *vortex methods* [10]. Vortex methods represent vorticity as a sum of localized, moving basis functions, referred to as *elements* or *blobs*. Each element is characterized by the position of its center and other parameters (Fig. 1).

The high-level structure of the ECCSVM algorithm is shown in Fig. 2. The algorithm takes as input the initial values of the elements as well as the initial and final times and the length of each time step. At each iteration, the velocities (*vel*) are computed from the current vorticity data. This is the most computationally expensive part of the algorithm, but is necessary because the *integration step*, which updates the positions and parameters of the elements, requires explicit knowledge of the velocities.

The velocity is computed by evaluating a Biot-Savart integral [11, §2.3] over the entire spatial domain of the fluid. This means that the computation of the velocity at one element center requires knowledge of the vorticity at every element. A naïve approach would thus require $O(N^2)$ operations, where N is the

```
1  procedure slave (elements : Element[]; mpcoef : double[]; packsize : int) is
2  |  packbuf : byte[PBSize]; databuf : double[DataSize × WorkSize];
3  |  indexbuf : int[WorkSize]; i, pos : int;
4  |  sendreq, recvreq : MPI_Request; status : MPI_Status;
5  |  MPI_RECV_INIT(indexbuf, WorkSize, MPI_INT, 0, MPI_ANY_TAG, recvreq);
6  |  MPI_SEND_INIT(packbuf, packsize, MPI_PACKED, 0, 0, sendreq);
7  |  while true do
8  |  |  MPI_START(recvreq);
9  |  |  MPI_WAIT(recvreq, status);
10 |  |  if status.tag = Done then break;
11 |  |  for i ← 0 to WorkSize − 1 do
12 |  |  |  if indexbuf[i] ≠ −1 then
   |  |  |  databuf[i] ← f3(indexbuf[i], elements, mpcoef);
13 |  |  MPI_WAIT(sendreq, MPI_STATUS_IGNORE);
14 |  |  pos ← 0;
15 |  |  MPI_PACK(indexbuf, WorkSize, MPI_INT, packbuf, packsize, pos);
16 |  |  MPI_PACK(databuf, DataSize × WorkSize,
   |  |       MPI_DOUBLE, packbuf, packsize, pos);
17 |  |  MPI_START(sendreq);
18 |  MPI_REQUEST_FREE(recvreq);
19 |  MPI_WAIT(sendreq, MPI_STATUS_IGNORE);
20 |  MPI_REQUEST_FREE(sendreq);
```

Fig. 3. Slave

number of elements, in order to compute the velocity field at all element centers. The ECCSVM uses a more technique known as the *fast multipole method* (FMM) to approximate the integral in $O(N)$ operations. The FMM requires a decomposition of the domain into near and far elements, and therefore computational resources for evaluating the velocity will vary substantially from element to element, so dynamic load balancing is crucial in the parallel version.

The parallel version of the algorithm changes only computeVelocity. The computation of the FMM coefficients is distributed equally among the processes and the results are summed onto each process with an MPI_ALLREDUCE. The computation of the velocities from the FMM coefficients (the invocation of f_3 on line 16), on the other hand, uses a variation of the master-slave pattern. The master sends to a slave a list of element indices and the slave performs the f_3 computations on the specified elements and packs and sends the results back to the master. The master unpacks the results and updates *vel* appropriately. When all the velocity calculations are complete, the master broadcasts *vel* to all processes. In the actual BlobFlow code, there are many variations on the standard patterns, however. For example, the master initially sends two tasks to every slave, persistent requests are used in sometimes complicated ways, there is somewhat complex packing and unpacking of data. The slave code (which is a small portion of the parallel computeVelocity code) is summarized in Fig. 3.

3 Verification

The first goal was to verify freedom from deadlock, and we constructed an MPI-SPIN model of the parallel version of BLOBFLOW for this purpose. We found that a naïve translation, in which each variable in the original program is mapped to a unique variable in the model, would never scale, due to the sheer number of variables in the program. The key idea to making a tractable model is *conservative abstraction*. Conservative abstraction is a process by which we leave out some information from the model, but we ensure that when there is a decision that requires that information, the model checker will explore all possible outcomes (and perhaps some that are not possible).

To be precise, if M is a model of a program P, then every execution e of P maps to an execution $f(e)$ of M. Many distinct program executions may map to the same model execution; the extent to which this happens is a measure of the *abstraction* of the model. There may also be model executions that are not in the image of f; these are called *spurious* executions. The model is *conservative* for a property π if the following is true: for all e, π holds for $f(e)$ iff π holds for e. If the model checker verifies that π holds on all executions of a conservative model, one can conclude π holds for P. If, on the other hand, the model checker produces a counterexample e', it is possible that π still holds for P because e' is spurious. Hence one must examine e', and if it is determined to be spurious, the model is *refined* by adding sufficient information to eliminate the spurious execution. One continues to refine the model in this way until either an actual counterexample is produced or the property is verified successfully [12].

The natural starting point is a very abstract but conservative model that kept just enough information to capture the rank and request arguments occurring in the MPI function calls. An excerpt of this model is shown in Fig. 4(a) and corresponds to the main slave loop of Fig. 3. The conditional statement of Fig. 3, line 10, which controls when the slave breaks out of the loop, has been replaced by a nondeterministic choice, which means that execution may or may not break out of the loop—both possibilities are explored at each iteration. For this model,

(a)
```
do :: MPI_Start(Pslave,&Pslave->recvreq);
      MPI_Wait(Pslave,&Pslave->recvreq, MPI_STATUS_IGNORE);
      if :: 1 -> break :: 1 fi;
      MPI_Wait(Pslave,&Pslave->sendreq, MPI_STATUS_IGNORE);
      MPI_Start(Pslave,&Pslave->sendreq) od
```

(b)
```
do :: MPI_Start(Pslave,&Pslave->recvreq);
      MPI_Wait(Pslave,&Pslave->recvreq, &Pslave->status);
      if :: status.tag == DONE -> break :: else fi;
      MPI_Wait(Pslave,&Pslave->sendreq, MPI_STATUS_IGNORE);
      MPI_Start(Pslave,&Pslave->sendreq) od
```

Fig. 4. (a) Abstract model of main slave loop. (b) Refined to include *status*.

```
do :: MPI_Start(Pslave, &Pslave->recvreq);
      MPI_Wait(Pslave, &Pslave->recvreq, &Pslave->status);
      if :: status.tag == DONE -> break :: else fi;
      do :: i < WorkSize -> c_code {
            if (Pslave->indexbuf[Pslave->i] != (uchar)(-1))
              Pslave->databuf[Pslave->i] =
                SYM_cons(f_3_id,
                SYM_cons(SYM_intConstant(Pslave->indexbuf[Pslave->i]),
                SYM_cons(Pslave->elements,
                SYM_cons(Pslave->mpcoef, SYM_NULL))));
            }; i++
         :: else -> i = 0; break od;
      MPI_Wait(Pslave, &Pslave->sendreq, MPI_STATUS_IGNORE);
      pos = 0;
      MPI_Pack(Pslave->indexbuf, WorkSize, MPI_BYTE, Pslave->packbuf,
        Pslave->packsize,&Pslave->pos);
      MPI_Pack(Pslave->databuf,DataSize*WorkSize,
        MPI_SYMBOLIC,Pslave->packbuf,Pslave->packsize,&Pslave->pos);
      MPI_Start(Pslave,&Pslave->sendreq) od
```

Fig. 5. Symbolic model of main slave loop

MPI-SPIN found a counterexample, and examination quickly revealed that it was spurious: one slave breaks out of the loop in its first iteration and the master ends up waiting forever for a result from that slave. We refined the model (manually) by adding the *status* variable (Fig. 4(b)). This provided sufficient precision to verify deadlock-freedom.

With the resulting model, which we call the *communication skeleton*, we were able to verify deadlock-freedom for up to 5 processes. (MPI-SPIN actually checks several other correctness properties automatically, such as (1) there are no allocated request objects for a process when MPI_FINALIZE is called; and (2) MPI_START is never invoked on an active request.) The execution time for the 5-process verification was 41 minutes on a Sun Ray with two dual-core 2.6 GHz AMD Opteron processors. The verification consumed 7.7 GB of RAM and explored 3.5×10^7 states. The numbers for the 4-process run were 45 seconds, 163 MB, 1.5×10^6 states; for 3 processes, 1 second, 56 MB, $62,798$ states.

Verifying the functional correctness of BLOBFLOW is a much harder problem. The goal is to verify that the sequential and parallel versions are *functionally equivalent*, i.e., they produce the same output on any given input. A method for this combining model checking and symbolic execution is described in [3] and we review it briefly here.

To perform this verification, we must model the program data *symbolically*: the input is represented as symbolic constants $X_1, X_2, \ldots$ and the output is represented as symbolic expressions in the X_i. Floating-point operations are replaced by corresponding symbolic operations, which simply build new symbolic expressions from their operands. These constructs are supported in MPI-SPIN

through a symbolic type and a set of symbolic operations. A model of this type is made for both the sequential and parallel programs. Mpi-Spin is then used to explore all possible executions of the models and verify that in each case, the symbolic expressions output by the two versions agree. Fig. 5 shows the symbolic model of the main slave loop, which refines the abstract model used in the communication skeleton.

Because floating-point arithmetic is only an approximation to real arithmetic, we must clarify what is meant by *functionally equivalent*. Two programs that are equivalent when the arithmetic operations are interpreted as taking place in the set of real numbers may not be equivalent if those operations are implemented using IEEE754 floating-point arithmetic. Two programs that are equivalent with IEEE754 arithmetic may not be equivalent if some other floating-point arithmetic is used. Different notions of equivalence may be appropriate in different circumstances. For example, a parallel program containing an MPI reduction operation using floating-point addition may obtain different results when run twice on the same input, because the sum may be computed in different orders. Such a program cannot be floating-point equivalent to any sequential program, but it may be real-equivalent to one. Mpi-Spin deals with this situation by offering the user a choice of three successively stronger equivalence relations: real, IEEE, and Herbrand, the last holding only if the two programs produce exactly the same symbolic expressions.

A straightforward application of this method used in previous work would not scale to BlobFlow. In the earlier work, the symbolic variables corresponded one-to-one with the floating-point variables in the original program; the sheer number of such variables in BlobFlow meant that the memory required to store one state of the model would be prohibitive.

Our solution to this problem involved two related ideas. The first is that groups of variables that tend to be manipulated together can often be represented by a single symbolic variable in the model. The second is that sections of code that are shared by the sequential and parallel versions can be represented by a single abstract symbolic operation. An example is the function f_3 that is invoked in both Fig. 2, line 16, and Fig. 3, line 12. Since we are only trying to prove the equivalence of the two versions, there is no need to know exactly what f_3 computes. Instead, we can just introduce a new symbolic operation for f_3, and use it wherever this function is invoked in the codes. We used both of these techniques extensively to design a reasonably small, conservative symbolic model.

Space does not permit us to describe the model in detail, but the source for the model and all other artifacts used in this study are available at `http://vsl.cis.udel.edu`. Using this model, we were able to verify functional equivalence for up to 4 processes. Since the parallel version uses a floating-point addition reduction operation (in the FMM computation), we used the real-equivalence mode. The 4-process run lasted 21 minutes, consumed 1.2 GB, explored 2.0×10^7 states, and generated 988 distinct symbolic expressions. The numbers for the 3-process run were 19 seconds, 96 MB, $691,837$ states, and 731 expressions.

4 Conclusion

In this paper, we have reported on our investigation using the model checker MPI-SPIN to verify correctness properties of a nontrivial parallel scientific program. We were able to verify generic concurrency properties, such as freedom from deadlock, for models of the program with up to 5 processes. We also verified the functional equivalence of the parallel and sequential versions of the program, for up to 4 processes and within specific bounds on certain parameters.

In carrying out this study, we have developed—at least in outline—a methodical way to construct the models. The method begins with a very abstract but conservative model that encodes only the data necessary to represent the rank and request arguments occurring in the MPI function calls of the program. The model is then progressively refined until sufficient precision is achieved to either verify or produce a valid counterexample to freedom from deadlock. This "communication skeleton" model is then augmented by representing the program data symbolically, and the resulting model is used to verify functional equivalence.

Several abstraction techniques proved useful in the construction of the symbolic model. The most important requires one to locate units of code common to the sequential and parallel versions of the program and abstract these segments using uninterpreted symbolic operations. This process is made easier if the sequential and parallel versions share code. It also requires partitioning the data of the programs and assigning a symbolic variable to each partition.

Further progress will require the development of a formal basis for these abstraction techniques. This would lay the groundwork for automating (at least in part) the model construction process. Indeed, there are clearly some established static analysis techniques that could be brought to bear on our method. These include standard techniques to estimate the set of variables read or written to by a program unit, and dependence analysis, which could be used to determine that certain variables must be incorporated into the models. These techniques, however, must be made aware of certain aspects of the semantics of the MPI functions used in the programs. In other contexts, the counterexample-driven refinement loop has been automated using theorem-proving techniques, and it is possible that similar techniques could be adopted for MPI-based programs.

Finally, more work is needed to address the state-explosion problem for master-slave style programs. In this study, we observed a very steep blowup in the number of states with the process count. The reason for this appears to be the combinatorial explosion inherent in the master-slave architecture, due to the large number of ways tasks can be partitioned among slaves and the differing orders in which results can be received by the master. Various reduction techniques have been devised for programs that avoid the nondeterministic constructs used in master-slave programs, such as MPI_ANY_SOURCE and MPI_WAITANY (e.g., [13]). These techniques allow model checking to scale effectively for many types of programs, such as standard discrete grid simulations. Other reduction strategies have been proposed to deal with wildcard receives (e.g., [14, 15]). These have not been incorporated into MPI-SPIN, but it is not clear they would make a substantial difference for master-slave programs in any

case. It appears that some fundamental algorithmic advance must be made in this area if model checking is to become practical for this important class of parallel scientific programs.

These are some of the many avenues of future research.

Acknowledgments. We are grateful to the U.S. National Science Foundation for funding under grant CCF-0733035 and to Samuel Moelius for assistance in constructing the Mpi-Spin models.

References

1. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In Graf, S., Mounier, L., eds.: Model Checking Software: 11th Intl. SPIN Workshop. Volume 2989 of LNCS., Springer (2004) 286–303
2. Palmer, R., Gopalakrishnan, G., Kirby, R.M.: Semantics driven partial-order reduction of MPI-based parallel programs. In: Parallel and Distributed Systems: Testing and Debugging (PADTAD V), London (July 2007)
3. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. Transactions on Software Engineering and Methodology **17**(2) (2008) Article No. 10, 1–34
4. Rossi, L.F.: Resurrecting core spreading methods: A new scheme that is both deterministic and convergent. SIAM J. Sci. Comp. **17**(2) (1996) 370–397
5. Rossi, L.F.: Achieving high-order convergence rates with deforming basis functions. SIAM J. Sci. Comput. **26**(3) (2005) 885–906
6. Rossi, L.F.: Evaluation of the Biot-Savart integral for deformable elliptical gaussian vortex elements. SIAM J. Sci. Comput. **28**(4) (2006) 1509–1532
7. Siegel, S.F.: The Mpi-Spin web page. `http://vsl.cis.udel.edu/mpi-spin` (2007)
8. Siegel, S.F.: Model checking nonblocking MPI programs. In Cook, B., Podelski, A., eds.: Verification, Model Checking, and Abstract Interpretation: 8th Intl. Conference (VMCAI 2007). Volume 4349 of LNCS. (2007) 44–58
9. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
10. Cottet, G.H., Koumoutsakos, P.D.: Vortex methods: Theory and Practice. Cambridge University Press (2000)
11. Saffman, P.G.: Vortex Dynamics. Cambridge University Press (1992)
12. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Model Checking of Software: 8th Intl. SPIN Workshop, Springer (2001) 103–122
13. Siegel, S.F., Avrunin, G.S.: Modeling wildcard-free MPI programs for verification. In: Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05), ACM Press (2005) 95–106
14. Siegel, S.F.: Efficient verification of halting properties for MPI programs with wildcard receives. In Cousot, R., ed.: Verification, Model Checking, and Abstract Interpretation: 6th Intl. Conference (VMCAI 2005). Volume 3385 of LNCS. (2005) 413–429
15. Vakkalanka, S., Gopalakrishnan, G., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Computer Aided Verification: 20th International Conference, CAV 2008, Princeton, USA, July 7–14, 2008, Proceedings. To appear.