A Case Study in the Use of Model Checking to Verify Parallel Scientific Software

Stephen F. Siegel
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA
siegel@cis.udel.edu

Samuel E. Moelius III
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA
moelius@cis.udel.edu

Louis F. Rossi
Department of Mathematical Sciences
University of Delaware
Newark, DE 19716, USA
rossi@math.udel.edu

ABSTRACT

We report on a case study using the model checking tool Mpi-Spin to verify a nontrivial, deployed, MPI-based scientific program. The program, BlobFlow/ECCSVM, is an open source code implementing a high order vortex method for solving the Navier-Stokes equations in two dimensions. Over the past several years, it has been used for a variety of applications, including jets, coherent vortex structures, and fluid-structure interactions. Despite the size and complexity of the code, we are able to verify not only data-independent properties such as freedom from deadlock, but also the functional equivalence of the sequential and parallel versions of the program, under certain restrictions. Our "lessons learned" include new insights into the technology that will be required to automate the modeling and verification process for complex scientific software.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; I.1.0 [**Symbolic and Algebraic Manipulation**]: General; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Verification

Keywords

MPI, Message Passing Interface, parallel programming, formal methods, analysis, finite-state verification, model checking, deadlock, concurrent systems, Spin, ECCSVM, vortex methods, fluid flow

1. OVERVIEW

1.1 Introduction

The revolutionary advances made by scientific computing in the last decade have depended in large part on the computational power provided by parallelism. But parallelism has also contributed to the increasing complexity of scientific software, making the programs more difficult to develop, test, and debug. The result of this has been not only an increasing burden on developers but also a decrease in the level of confidence in the correctness of the programs.

One of the reasons parallel programs are more difficult to develop than comparable sequential programs is the *nondeterminism* introduced by parallelism. By this we mean any factor that may affect program execution but is not specified by the program's source code. For parallel programs based on the Message Passing Interface (MPI), such factors include (1) the way that statements from different processes are interleaved in time, (2) the choices made by the MPI infrastructure in selecting a message for reception at an MPI_ANY_SOURCE receive, (3) the choices made in selecting a completed request at an MPI_WAIT_ANY statement, and (4) the choice of whether to buffer a message emanating from a standard-mode send or force such a message to be received synchronously. The existence of such factors means that a program that executes correctly on a given test input may operate incorrectly when run a second time with the same test input. This is why traditional testing and debugging techniques alone are often inadequate for parallel programs.

A number of approaches have been proposed to deal with these problems. Among these are techniques based on *model checking* [3], which generally involve three tasks: (1) the construction of a finite-state formal model of the program, typically expressed in the language of a particular model-checking tool, (2) the formulation of one or more correctness properties for the model, and (3) the use of automated algorithmic techniques to exhaustively explore all possible states of the model and verify that the properties hold on *all* executions. These techniques are particularly attractive because in theory they can not only establish the correctness of a parallel program but can automatically produce a trace demonstrating how the program may fail if the program is not correct. Thus model checking has the potential to both reduce development effort and increase confidence in the correctness of the final programs.

Of course, model checking is not a panacea. In the first place, it operates on a mathematical model of the program instead of on the program itself. If the model does not faithfully correspond to the program, then the conclusions reached by the model checker may be invalid. A second

Technical Report UD-CIS-2007-343, November 12, 2007

The first and second authors were supported by the National Science Foundation under Grant No. CCF-0541035.

problem is that, in order to keep the number of states tractable, one usually analyzes small configurations of the program, in terms of the number of processes, the size of the input, and so on. Testing exhibits neither limitation.

On the other hand, much progress has been made (in other domains) on automating the process of model construction. (See, for example, the Bandera tool set for multi-threaded Java programs [4].) Indeed, one of the goals of this study is to gain insight into the problem of automatic model construction for MPI-based programs. As for the second problem, experience has shown that defects in programs almost always manifest themselves in small configurations. In fact, defects that may never be revealed by testing small configurations are often detected by model checking. A typical example is an MPI program which deadlocks only if the MPI implementation forces some message to be delivered synchronously. The implementation might choose to buffer the message whenever sufficient resources are available, and so the defect might not be revealed by testing until a very large message size or large number of processes is reached. With model checking, however, the possibility of synchronization will be explored at every possible state, and so the defect could be detected using a small configuration. There is also a growing body of techniques for reducing the number of states that need to be explored by a model checker, allowing model checking techniques to scale to increasingly larger configurations. The wide adoption of model checking in the hardware industry and its growing relevance in the software industry testify to this progress.

Scientific software, however, poses significant challenges to model checking techniques. Among the many issues are (1) the reliance of scientific software on enormous quantities of floating-point data, (2) the use of complex MPI communication constructs, and (3) the problem of constructing appropriate finite-state models of scientific programs. Finding ways to deal with these issues without exacerbating the state explosion problem appears to be difficult. Consequently, most of the research has been concerned with specific techniques to overcome these obstacles. The actual applications have been to relatively simple programs, such as a solver for the 2-dimensional *heat* (or *diffusion*) *equation* [14, 24], a simple Jacobi iteration algorithm for solving a linear system of equations [25], a Gaussian elimination program, a matrix multiplication program, and a Monte Carlo simulation for estimating π [26]. Each of these programs is either an example from a programming course or textbook or was written specifically as an example for verification. Clearly, if the range of applicability of model checking technology is to expand to include production codes used in actual scientific research, then more realistic examples must be studied.

In this paper we report on a case study using the model checker Mpi-Spin [22, 23] to verify L. Rossi's computational fluid dynamics program *BlobFlow* [16, 17]. BlobFlow has been actively developed and used over the past 7 years to explore fluid flow phenomena and novel simulation algorithms (e.g., [17–19]). It consists of approximately 10K lines of C code and includes both a parallel (MPI-based) version and a sequential (MPI-free) version. The program is sufficiently complex to pose a more realistic challenge to current techniques, yet not so large that it lies hopelessly outside their bounds. By attempting to verify BlobFlow, we hoped to (1) ascertain how well current model checking techniques scale to realistic codes, (2) learn how those techniques might be modified or extended to improve their performance and range of applicability, and (3) gain insight into entirely new techniques that may prove useful.

1.2 Methodology

For our model checker, we used Mpi-Spin, an open-source extension to the model checker Spin [10] for verifying MPI-based parallel programs. It adds to Spin's input language (Promela) many of the commonly-used MPI primitives, including all of those used in BlobFlow, greatly facilitating the model construction process. Details on its implementation can be found in [23]. The distribution available from the Mpi-Spin web page [22] also includes a detailed user's manual and numerous examples.

One of the foremost obstacles to the use of model checking is the need to first construct a finite-state model of the program one wishes to verify. As is the case with most model checking tools, the input language of Mpi-Spin, MPI-Promela, differs in significant ways from programming languages such as C. For example, there is no floating-point type, pointer type, nor any mechanism for dynamically allocating memory. Indeed, such constructs are extremely difficult to encode directly without leading to an explosion in the number of states of the model. The number of possible states of a single 64-bit floating point value, for example, is 2^{64}, so any approach which requires the exploration of all such states is clearly intractable. One of the main thrusts in model checking research has therefore involved ways to *abstract* such constructs, the goal being to incorporate just enough information to prove the desired correctness properties, and no more. While there has been some progress in automating this process in other domains, there is very little in the published literature on abstractions for MPI-based programs, and the model construction process remains at this point as much an art as a science. In verifying BlobFlow, we therefore hoped to discover methodical techniques to aid the model construction process, including abstractions appropriate for MPI-based scientific programs.

Recall that model checking techniques require as input one or more correctness properties. We began our investigation of BlobFlow by attempting to verify properties of the "communication skeleton" of the program, i.e., properties which do not depend on the data manipulated by the program. These properties are *generic* in the sense that they are expected to hold for any correct MPI-based program, and include properties such as freedom from deadlock and the proper deallocation of communication request objects. The model we developed for checking these properties abstracts away most of the data manipulated by the program, but faithfully represents the flow of control and invocation of MPI functions in the program.

Of course, it is still possible for the generic communication properties to hold but for the parallel program to be erroneous. In fact, what it means for a scientific program to be *correct* is a many-layered question, and may involve issues such as the numerical stability of the algorithms used, concordance with empirical data, and so on. Clearly, it makes sense to separate these concerns. The particular concern of interest to us is the question of the correctness of the *parallelization* of the program, i.e., the correctness of the parallel program under the assumption that the sequential version is correct. To be precise, the next property we attempted to establish for BlobFlow was the *functional equivalence* of

the sequential and parallel versions of the program, i.e., the claim that for any given input, the output produced by the two versions is the same.

There are two reasons why this approach to verifying the correctness of parallelization is valuable. First, since sequential programs are generally deterministic, they are much more amenable to standard testing and debugging techniques, and indeed a well-known body of effective techniques and tools already exists for such programs. The second reason is practical: it is already standard practice for developers of scientific software to first develop a sequential implementation of an algorithm before implementing a parallel version (as is indeed the case with BlobFlow). Hence the approach utilizes artifacts that generally already exist.

To verify the functional equivalence of the sequential and parallel versions of BlobFlow, we used the method described in [26]. This method extends model checking with symbolic execution in a way that allows one to compare the output of two programs on any input. The method requires a considerably more complex model in which the floating-point data manipulated by the programs are represented using symbolic structures.

Initially we planned on limiting our analysis to the current development version of BlobFlow, making no modifications to the source code. However, as we looked at the models we had built, we could not help noticing ways in which the code could be improved. We decided to experiment with some of these modifications in a way that incorporated the model checking techniques we had developed. For each proposed modification, we adhered to the following protocol: first, (1) make the modification in the models; then (2) re-run Mpi-Spin on those models to check that the properties still hold; next (3) make the modifications in the source; and finally (4) run the modified code on the BlobFlow test suite. We found this process to be one of the most pleasurable phases of the project: while it took a good deal of effort to construct the original models, the effort required to modify the models was trivial, and the rewards, immediate.

The remainder of this paper is organized as follows. In Sec. 2 we give a brief overview of the ECCSVM algorithm and its sequential and parallel implementation in BlobFlow. In Sec. 3 we describe the method we used to construct the communication skeleton model and results we obtained by analyzing that model with Mpi-Spin. Sec. 4 deals with the problem of functional equivalence and the symbolic model used to verify that property. Our experience modifying BlobFlow is described in Sec. 5. In Sec. 6, we conclude with some "lessons learned" from our experience, including avenues for future research.

The source code for the models and all related artifacts can be downloaded from `http://vsl.cis.udel.edu/`.

2. ECCSVM AND BLOBFLOW

2.1 Overview

The motions of gasses and liquids fall under the broad category of fluid dynamics, which plays a central role in many important physical processes and industrial applications. Understanding the motions of fluids is central to understanding processes as great as weather patterns and ocean currents, and as minute as the flow of blood through capillaries or the motion of ink droplets propelled by ink jet printers.

The Elliptical Corrected Core Spreading Vortex Method (ECCSVM) [16, 17] is an algorithm for computing the motions of incompressible gasses and liquids in two dimensions. It is best suited for flows with high Reynolds numbers. The *Reynolds number* of a fluid flow is a dimensionless ratio Re between inertial and viscous forces. The motion of the air exiting a leaf blower is an example of a high Reynolds number flow, while that of honey dripping from a spoon is a low Reynolds number flow.

The algorithm falls under the general category of *vortex methods* [5, 21]. These methods offer particular advantages in efficiency, accuracy and adaptivity for flow fields dominated by isolated regions of vorticity. Vortex methods are distinguished by the facts that (1) they require little or no tailoring to flow geometries and domains and (2) they are *naturally adaptive*. This last term means that computational resources are automatically concentrated in regions of the domain requiring resolution, and nowhere else.

2.2 Vortex methods

Fluid motion can be divided into two broad categories, steady and unsteady. A steady flow is one in which the velocity field at every point in space does not change with time. Unsteady flows, in which the flow velocities may vary in time, are far more difficult to calculate. Vortex methods have proven to be very effective in both two-dimensional and three dimensional unsteady flow problems, but in this paper, we will focus on two spatial dimensions. While some fluid flow algorithms solve partial differential equations for fluid pressure (p) and velocity ($\mathbf{u}$), vortex methods solve an equivalent system for *vorticity* ($\nabla \times \mathbf{u}$), the local angular momentum in the fluid (see [2, 5, 15, 21] for historical applications and general references).

Vortex methods rely upon the fact that velocity and vorticity encode equivalent information. To solve an unsteady flow problem in two dimensions with pressure and velocity, one must solve for the two velocity components plus the pressure over the entire domain at each discretized moment in time. Thus, there are three pieces of information that must be determined at each point in space/time. On the other hand, in two dimensions, the $\mathbf{i}$ and $\mathbf{j}$ components of vorticity are zero so we keep track of only the $\mathbf{k}$ component of the vorticity, denoted ω:

$$\omega = (\nabla \times \mathbf{u}) \cdot \mathbf{k} = \frac{\partial u_2}{\partial x} - \frac{\partial u_1}{\partial y}. \tag{1}$$

The fluid pressure plays no role at all in the evolution of the vorticity field, so rather than determining three pieces of information at every point in space and time, we only need to determine one, the value of ω.

One may compute ω from $\mathbf{u}$ using (1). To go the other way, one first evaluates the *Biot-Savart* integral [20, §2.3] over the entire domain to obtain the *stream function* ψ:

$$\psi(\mathbf{x}) = \frac{1}{4\pi} \iint \log(|\mathbf{x} - \mathbf{x}'|^2)\omega(\mathbf{x}')\, d\mathbf{x}', \tag{2}$$

where the domain of integration is the region occupied by the fluid and $\mathbf{x} = (x, y)$ is any point in that region. The velocity field is then obtained by

$$u_1 = \frac{\partial \psi}{\partial y}, \quad u_2 = -\frac{\partial \psi}{\partial x}. \tag{3}$$

One way to understand the relationship between ω and ψ is by substituting (3) into (1) to obtain $\nabla^2 \psi = -\omega$. The solu-

tion of this elliptic PDE can be written as the convolution (2) [11, 27].

In two dimensions, the *vorticity equation* takes the form

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} \cdot \nabla)\omega = \frac{1}{\mathrm{Re}} \nabla^2 \omega. \tag{4}$$

The left hand side of (4) is called a material derivative, the time rate of change in a reference frame that is moving with the fluid. If it were not for the nonlinear term $(\mathbf{u} \cdot \nabla)\omega$, (4) would be a simple diffusion equation with diffusivity 1/Re. This nonlinear term is a rich source of interesting physical phenomena, such as flow instabilities and turbulence, but the term also makes the equation especially difficult to treat numerically. As we shall see later, the moving computational elements in vortex methods offer a particularly elegant and effective solution. The terms on the left hand side of (4) represent *inertial* forces, while the right hand side represents *diffusive* forces. The Reynolds number, Re, is a dimensionless ratio relating the two effects. Viewed this way, (4) is very similar to the heat equation and describes vorticity as a quantity that moves with the fluid and diffuses.

Vortex methods are schemes that represent the fluid vorticity as a linear combination of localized, moving basis functions. This approach contrasts sharply with more traditional schemes, such as finite difference methods (which compute values on a fixed grid) or spectral methods (which use global, stationary basis functions). In the vortex methods literature, these moving basis functions are alternatively referred to as *computational elements* or *blobs*. If ω is the exact vorticity, the representation $\widehat{\omega}$ takes the form

$$\widehat{\omega}(\mathbf{x}) = \sum_{i=1}^{N} \gamma_i \phi(\mathbf{x} - \mathbf{x}_i, \sigma_i), \tag{5}$$

where γ_i is the *strength*, $\mathbf{x}_i$ is the *centroid*, and σ_i is the *core width* of the i^{th} computational element. For example, if one chose to use Gaussian computational elements, the expression would be

$$\widehat{\omega}(\mathbf{x}) = \sum_{i=1}^{N} \frac{\gamma_i}{4\pi\sigma_i^2} e^{-\frac{|\mathbf{x}-\mathbf{x}_i|^2}{4\sigma_i^2}}. \tag{6}$$

The goal is to replace the partial differential equation (4) with a system of ordinary differential equations for $\mathbf{x}_i$ and σ_i. To understand how moving computational elements can solve (4) effectively, we can substitute (6) into (4) and examine the contributions from each of the three constitutive terms:

$$\frac{\partial \widehat{\omega}}{\partial t} = \sum_{i=1}^{N} \frac{\gamma_i}{4\pi\sigma_i^2} e^{-\frac{|\mathbf{x}-\mathbf{x}_i|^2}{4\sigma_i^2}} \times \left\{ \left(\frac{d(\sigma_i^2)}{dt} \right) \left[-\frac{1}{\sigma_i^2} + \frac{|\mathbf{x}-\mathbf{x}_i|^2}{4\sigma_i^4} \right] + \left(\frac{d\mathbf{x}_i}{dt} \right) \cdot \frac{\mathbf{x}-\mathbf{x}_i}{2\sigma_i^2} \right\}, \tag{7a}$$

$$(\mathbf{u} \cdot \nabla)\widehat{\omega} = \sum_{i=1}^{N} \frac{\gamma_i}{4\pi\sigma_i^2} e^{-\frac{|\mathbf{x}-\mathbf{x}_i|^2}{4\sigma_i^2}} \left(-\mathbf{u} \cdot \frac{\mathbf{x}-\mathbf{x}_i}{2\sigma_i^2} \right), \tag{7b}$$

$$\nabla^2 \widehat{\omega} = \sum_{i=1}^{N} \frac{\gamma_i}{4\pi\sigma_i^2} e^{-\frac{|\mathbf{x}-\mathbf{x}_i|^2}{4\sigma_i^2}} \left(-\frac{1}{\sigma_i^2} + \frac{|\mathbf{x}-\mathbf{x}_i|^2}{4\sigma_i^2} \right). \tag{7c}$$

Notice that the computational elements decay exponentially as one moves away from the centroid, so these expressions are sums of localized effects centered at the centroids ($\mathbf{x}_i$'s). If we plug our approximations (7) into the exact expression (4), we find that setting

$$\frac{d(\sigma_i^2)}{dt} = \frac{1}{\mathrm{Re}} \tag{8}$$

will cancel out the leading order terms involving $(\mathbf{x} - \mathbf{x}_i)^0$ (only found in (7a) and (7c)). As a bonus, this automatically eliminates terms involving $|\mathbf{x}-\mathbf{x}_i|^2$. Collecting the next term involving $(\mathbf{x}-\mathbf{x}_i)$ (only found in (7a) and (7b)), we see that we cannot quite make terms cancel out perfectly because $\mathbf{u}$ varies in space and the centroid velocity takes a single value. The best that can be done is to let

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{u}(\mathbf{x}_i). \tag{9}$$

The fundamental source of error in a vortex method is attributed to the slight difference between $\mathbf{u}(\mathbf{x})$ and $\mathbf{u}(\mathbf{x}_i)$ across the support of the computational element, and it is for this reason that the method converges to the exact solution as the core width (σ_i) diminishes to zero.

Thus, (9) and (8) comprise a vortex method for solving (4). The general stategy for solving (4) follows these steps:

1. the values for ω at the initial time $t = t_0$ are given. The desired initial conditions are transformed into computational element parameters γ_i, $\mathbf{x}_i$, σ_i and perhaps other shape parameters as well.

2. the Biot-Savart integral is evaluated to compute the velocity field $\mathbf{u}$ at each centroid $\mathbf{x}_i$ at time t.

3. solve (9) by moving the centroids $\mathbf{x}_i$ and capture diffusion by an appropriate method such as core spreading (8).

4. replace t with $t + \Delta t$ and go to Step 2.

Completing Step 2 requires a calculation of the Biot-Savart integral for the particular ϕ one is using. The Biot-Savart integral (2) for Gaussian computational elements (6) will yield the velocity field

$$\mathbf{u}(\mathbf{x}) = \sum_{i=1}^{N} \frac{\gamma_i}{2\pi} \frac{1}{|\mathbf{x}-\mathbf{x}_i|^2} \begin{bmatrix} -(y-y_i) \\ x - x_i \end{bmatrix} e^{-\frac{|\mathbf{x}-\mathbf{x}_i|^2}{4\sigma_i^2}}. \tag{10}$$

The vast majority of computational effort in vortex methods is expended in step 2 evaluating the velocity field. Examining (2), we see that the velocity field evaluation at one point requires knowing the vorticity over the whole domain. In the Gaussian example (10), we see that knowing the velocity field at one point will require a summation over all N elements. If we were to rely solely upon (10), we must perform $O(N^2)$ operations to find the velocity field at all N centroids. In the next section, we shall see we can improve the velocity evaluation considerably, but this will not change the fact that step 2 consumes most of the computational resources in a vortex calculation.

There are many other forms of vortex methods, but they all follow a strategy similar to Steps 1-4 above. Investigators use different shaped basis functions for reasons including improving spatial accuracy and conforming to special geometries. Considerable research has focused on capturing the diffusive term in (4) when Re is large but finite, resulting in a number of schemes, including random walks [2],

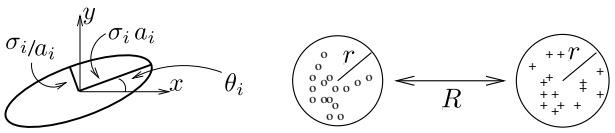

Figure 1: Left: A schematic diagram of the geometry of an elliptical Gaussian basis function. Right: Separation of scales into near and far fields

particle redistribution [6, 7, 13] and core spreading [8, 12]. ECCSVM uses core spreading (7c), a technique where diffusion is captured by having individual basis functions spread. (See [16] for an explanation of technical problems associated with these techniques and how they can be resolved.) In all vortex methods, there is no geometry built into the computational elements, so one can add or remove computational elements without having to reconfigure the data structure as is the case with most traditional schemes. Finally, all vortex methods use moving computational elements (9) to capture the convective term which is the most challenging feature in models of fluid motion.

Methods with moving computational elements are also effective when solving the *convection-diffusion equation*,

$$\frac{\partial \rho}{\partial t} + (\mathbf{u} \cdot \nabla)\rho = \frac{1}{\mathrm{Pe}} \nabla^2 \rho, \tag{11}$$

where ρ is any passive scalar quantity and $\mathbf{u}$ is a known velocity field. A passive scalar is a quantity transported by a fluid that does not interact with it. For instance, dye in water is a passive scalar. Similar to the vorticity equation the Péclet number is a dimensionless ratio of convective to diffusive effects. While structurally similar to (4), the key difference is that the field ρ and the known velocity field are decoupled. If one wishes to test the accuracy or other aspects of a vortex method, one can solve (11) by providing an explicit velocity field and skipping the Biot-Savart integral (Step 2). The representation for ρ is identical to that of $\widehat{\omega}$ in (5), and the evolution equations for $\mathbf{x}_i$ and σ_i would be the same if one were using Gaussian basis functions.

2.3 Algorithm

The ECCSVM is both more accurate and more complex than the simple method described above, but it has the same basic ingredients. The most important difference is that the ECCSVM uses *deforming* basis functions instead of *rigid* basis functions. This feature, carefully applied, yields a vortex method for high Reynolds number viscous flows with an unusually high order spatial accuracy. Spatial accuracy describes how the computed solution converges to the exact solution as a function of core size, σ. As one uses smaller, more refined elements, one expects that the total error, the difference between the computed solution and the exact solution, should decrease as some power of σ. An axisymmetric core spreading method achieves an accuracy of $O(\sigma^2)$ but ECCSVM achieves a convergence rate of $O(\sigma^4)$. A rigorous analysis and validation of ECCSVM is explained in [17]. The deforming basis functions are *elliptical Gaussians* so that each blob is described not only by a strength (γ_i), position ($\mathbf{x}_i$), and width (σ_i), but also an *aspect ratio* (a_i^2) and

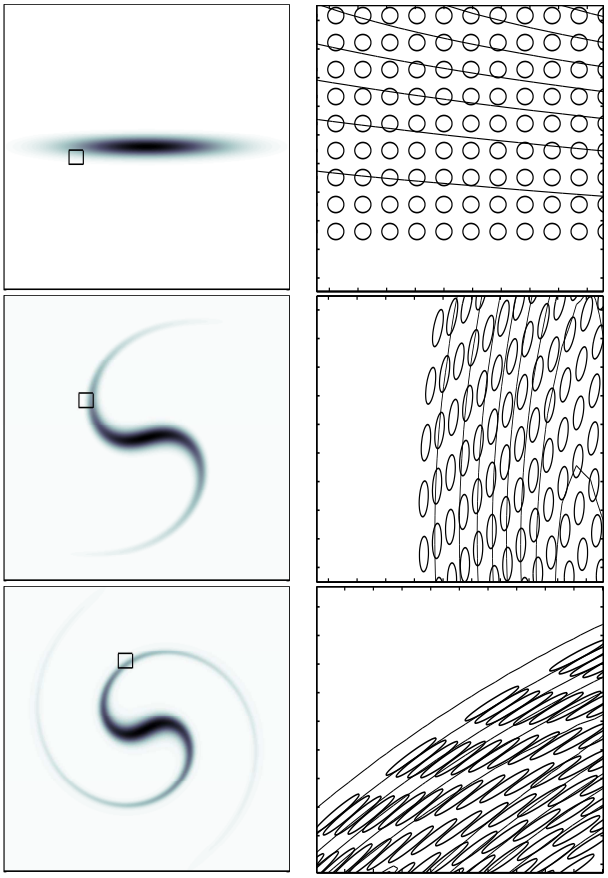

Figure 2: A sample ECCSVM computation.

an *orientation* (θ_i) as given in the expression

$$\phi(\mathbf{x}; \mathbf{x}_i, \sigma_i, a_i, \theta_i) = \frac{1}{4\pi\sigma_i^2} \exp\left(-\frac{|A_{\theta_i, a_i}(\mathbf{x} - \mathbf{x}_i)|^2}{4\sigma_i^2} \right), \tag{12}$$

where

$$A_{\theta,a} = \begin{bmatrix} \cos\theta/a & \sin\theta/a \\ -a\sin\theta & a\cos\theta \end{bmatrix}.$$

Fig. 1 (left) is a schematic diagram of the computational element showing a single contour of an elliptical Gaussian. The centroid and shape parameters evolve in time as a function of the velocity field and its spatial derivatives [17,18] to achieve significantly greater accuracy.

In Fig. 2, we present a sample computation for the mixing of a *passive scalar* in a fluid (11). In this example, we have imposed a counterclockwise differentially rotating velocity field where the fluid rotates fastest in the center. The state of the calculation is shown at three times. The initial state $t = 0$ is shown in the first row. The value of ρ at each point is indicated by the intensity of the gray-scale. An area of detail is indicated by the small black box. The box will move with the flow so that we can observe how the method represents ρ over time. Initially, a thin streak of dye is placed horizontally in the center of the domain, and ρ diffuses slowly ($Pe = 10^5$) as it is stretched by the differential rotation. The second row is the state of the system at approximately one turnover time in the center ($t = 30$). Since

the flow rotates differentially, the box has not yet completed a full revolution, and the tips of the streak have traveled only 5/8 of a revolution. The third row displays the system after the center turns over twice ($t = 60$), and the tips have traveled 5/4 of a revolution. The field ρ is shown in the left column. The computational element positions in the small box at left are represented at the right with orientation θ and semi-major and minor axes of $\sigma a/5$ and $\sigma/(5a)$. Thin lines represent contours of ρ corresponding to the field values shown as gray-scale intensities at the left. Notice that no computational resources are expended where there is no ρ and that ECCSVM naturally adapts to the changing flow geometry by moving and deforming the computational elements.

Another significant improvement over the basic method is the use of the *fast multipole method* (FMM) to compute the Biot-Savart integral (2). A naive approach would take $O(N^2)$ operations to compute the velocity field. The fast multipole method relies upon a separation of scales into near and far fields to reduce the computational complexity to $O(N)$. In Fig. 1 (right), the +'s and o's represent computational elements, formed into two distinct sets separated by a distance R. If there are m +'s and n o's, a direct velocity evaluation requires $(m+n)^2$ calculations. However, it is possible to approximate the velocity field induced by each of the +'s as a Laurent series relative to the center of the right cluster. The series coefficients are computed once and are used many times to estimate far field influences at different computational element centroids. This series will converge as long as we evaluate the series outside of the right cluster, which is guaranteed for all the elements in the left cluster. Combining the velocity field influences of all of the +'s is just a matter of summing the coefficients of the individual Laurent series. Thus, the influence of the the m +'s in the right cluster on any o in the left cluster can be performed by summing $p_{\max}$ terms of a single Laurent series where $p_{\max}$ controls the accuracy of the far-field FMM approximation. The influence of the other o's must be evaluated directly. Without multipole summation, the number of operations is $(m+n)^2 = m^2 + n^2 + 2mn$. With multipole summation, we use Laurent series to evaluate the influence of vorticity in the far field requiring $(m+n)p_{\max}$ operations and direct interactions for the near field requiring $m^2 + n^2$ operations. Therefore, we replace an $O(mn)$ operation with a $O(m+n)$ operation. The last component of the FMM is the construction of a nested structure of clusters which allows one to choose the smallest cluster size to be arbitrarily small and still make use of the multipole coefficients over large areas [9]. The original FMM was limited to singular computational elements, but these algorithm has been extended to work with elliptical Gaussian basis functions and other regular elements [19] with minimal additional requirements.

The high-level structure of the ECCSVM algorithm is shown in Fig. 3. (Like the other algorithms discussed in this paper, this algorithm is expressed in pseudocode which omits many details in the actual BlobFlow code, but nevertheless captures the essential features of the code.) 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 (variable *vel*) must be computed and stored from the current vorticity data; this is the most computationally expensive part of the algorithm

```
 1 procedure ECCSVM (t_0, t_f, Δt : in double;
       numElements : in int; elements : in Element[]) is
 2 |   vel : Vector[MaxElt];
 3 |   t ← t_0;
 4 |   while t ≤ t_f do
 5 |   |   computeVelocity(numElements, elements, vel);
 6 |   |   integrate(numElements, elements, vel);
 7 |   |   output(t, numElements, elements);
 8 |   |   t ← t + Δt;
 9 |   end
10 end procedure

11 procedure computeVelocity(numElements : in int;
       elements : in Element[]; vel : out Vector[]) is
12 |   mpcoef : double[MpSize];
13 |   i, j, p : int;
14 |   for i ← 0 to MpSize − 1 do mpcoef[i] ← 0;
15 |   for i ← 0 to numElements − 1 do
16 |   |   for p ← 0 to PMax − 1 do
17 |   |   |   j ← f_1(i, p, elements);
18 |   |   |   mpcoef[j] ← mpcoef[j] + f_2(i, p, elements);
19 |   |   end
20 |   end
21 |   for i ← 0 to numElements − 1 do
   |   |   vel[i] ← f_3(i, elements, mpcoef);
22 end procedure
```

Figure 3: ECCSVM algorithm, sequential version

and is discussed in more detail below. Even though vorticity and velocity encode equivalent information, the *integration step* of the algorithm requires explicit knowledge of the velocities in order to update the positions of the element centers. After that, the positions and other parameters of each element are updated in the integration step. Elements may also be created or destroyed in the integration step, so *numElements* may be modified. Finally, the algorithm outputs the new values of the elements and proceeds to the next iteration.

The computation of the velocity field using the FMM is outlined in the computeVelocity procedure of Fig. 3. It uses local array *mpcoef* to store the summed Laurent coefficients which encode the far field influences. The computation of *mpcoef* is accomplished by looping over all elements (i) and the terms in the series (p). For each i and p, an index into *mpcoef* and a contribution to be added to the entry at that index are computed in lines 17–18. The details are not important, other than to understand that both values are functions of only i, p, and *elements*, these functions being represented by the symbols f_1 and f_2, respectively. The function f_1 associates each element with a particular family of nested clusters. The velocities are updated using the *mpcoef* data for far field influences and direct interactions for near field interactions. Again, the details of this computation have been abstracted by the function symbol f_3. (One detail that must be checked is that for each i, the code represented by f_i is deterministic, and cannot loop forever or throw an exception.)

2.4 Parallelization

The ECCSVM algorithm achieves high spatial resolution with a much smaller memory footprint than those of other

methods, such as finite-difference schemes on grids, because they are naturally adaptive. Furthermore, fewer high order elements are needed to perform a calculation requiring many low order elements. For vorticity computations, 99.9% of the time in a typical execution is spent computing the velocity field. If we consider the velocity field to be divided into direct (near-field) and multipole (far-field) interactions, the direct interactions consume 80% of the computational resources while the multipole calculations consume 20%. Since the memory requirements are relatively low and the computational requirements per element are high, the parallel, MPI-based version of BlobFlow adopts a strategy in which every process has a complete copy of all the data, while the work required to compute the velocity field is divided up among the processes.

We now describe the parallel version of ECCSVM in detail. Each MPI process is assumed to start with the same initial data as that provided to the sequential version. The main (Fig. 3, lines 1–10) and integrate functions are unchanged. In fact, all differences between the two versions occur in the implementation of computeVelocity, so we turn our attention to that function.

Fig. 4 presents a pseudocode description of the parallel version of computeVelocity. The procedure is broken down into several subprocedures, each of which has its own local variables.

As in the sequential version, the first task is to compute the coefficients for the multipole summation. This is carried out in the procedure multipole. The work is distributed by having each process take a (roughly) equally-sized slice of the *elements* array and compute the contributions from the elements in its slice. The codes for computing the index into *mpcoef* and the contribution to the corresponding entry are exactly the same as in the sequential version and are denoted using the same symbols, f_1 and f_2. After each process has performed this work, the *mpcoef* arrays are added across all processes using an MPI_ALLREDUCE.

This is followed by the computation of the velocity field. While the computation of the velocity at any point is independent of that computation at any other point, an approach that simply assigns an equal number of elements to each process in this phase will suffer from load imbalance. The problem is that each velocity calculation involves a sum of near and far field contributions, and the time required for the near field computation is proportional to the square of the number of elements in the near field. This number varies in an unpredictable way from cluster to cluster and from one time step to the next. For this reason, a variation on the master-slave approach is taken.

Each process of positive rank plays the role of slave by calling the procedure slave. A single task for a slave consists of computing the velocity for WorkSize elements. The slave receives such a task from the master as a list of element indices. MPI *persistent requests* are instantiated in lines 31 and 32 for repeatedly receiving element index lists into indexbuf and for sending back velocity data in packbuf. The slave then enters its main loop. The receive request is started in line 34 and the slave blocks at line 35 until that request has completed. The tag of the incoming message is then examined. If the tag equals the constant value Done, indicating that there is no more work to be done, the slave breaks out of the main loop, waits for the send request (started in the previous iteration) to complete, deallocates the two request objects, and returns. Otherwise, the slave loops through the indices received. The value -1 is used to indicate an entry in the index array that should be ignored. (This is needed as WorkSize may not divide evenly the total number of tasks to be performed.) If the index is not -1, it is a valid index into the *vel* and *elements* arrays and the slave computes the velocity for that elements using the same code as in the sequential version (indicated by symbol f_3). At this point, the slave waits for the prior send request to complete and then uses MPI_PACK to incorporate the integer and floating-point values into a single buffer packbuf in a platform-independent way. (An MPI *derived datatype* could also be used for this purpose.) The persistent send request on *packbuf* is then started, and the next iteration begins.

The master (line 50) is the process of rank 0 and distributes tasks to the slaves. The integer variable *job* keeps track of the total number of tasks distributed so far. A task is prepared by invoking sendTask (line 78).

The master begins by sending two tasks to every slave (line 55) because an earlier version of the code was used on a cluster of workstations with a high-latency network, and profiling revealed that a significant amount of time was wasted by slaves waiting for incoming tasks. The problem was solved by modifying the pattern so that for each slave, at any time there is always one task being processed by the slave and one task "in transit" to the slave. Modifications of this type, which are targeted towards improving performance (perhaps for a particular platform) but which also may increase the complexity of the code, are quite common in high-performance scientific computing.

After sending out the initial tasks, the master initializes one persistent receive request for each slave (line 58). The receive buffer used for the slave of rank i is *packbufs*[i]. All of the requests are started in line 59, and then the master enters its main loop.

The first step in the main loop (lines 60–66) is to block until at least one receive request has completed. At that point, one of these requests is selected, the size (in bytes) of the incoming message is determined and stored in *msgSize*, and the message is processed by calling processPackBuf (line 88). This unpacks the list of integer indices and the floating point velocity data into *indexbuf* and *databuf*, respectively, and then inserts each velocity datum into the proper place in the *vel* array. Next, a new task is prepared for the slave which just returned this result, the receive request for that slave is restarted, the task is sent out to the slave, and another iteration of the main loop begins.

Control exits the main loop when all tasks have been sent out. At that point there are two outstanding tasks for each slave (since each slave was initially sent two). These are received and processed in lines 67–74, after which the persistent receive requests are deallocated and a Done message is sent to each slave.

3. DATA-INDEPENDENT PROPERTIES

The data-independent properties are *generic* in the sense that they should hold for any correct MPI program, and are checked automatically by Mpi-Spin. They include (1) freedom from deadlock, (2) for each process, MPI_INIT and MPI_FINALIZE are each called exactly once; MPI_INIT is called before any other MPI function and no MPI function is called after MPI_FINALIZE, (3) when a process p calls MPI_FINALIZE, there are no allocated request objects for

```
1  procedure computeVelocity (numElements : in int;
       elements : in Element[]; vel : out Vector[]) is
2      mpcoef : double[MpSize]; packsize, membersize : int;
3      multipole(numElements, elements, mpcoef);
4      MPI_PACK_SIZE(WorkSize, MPI_INT, membersize);
5      packsize ← membersize;
6      MPI_PACK_SIZE(DataSize × WorkSize, MPI_DOUBLE,
           membersize);
7      packsize ← packsize + membersize;
8      if rank = 0 then master(numElements, packsize, vel)
       else slave(elements, mpcoef, packsize);
9      MPI_BCAST(vel, numElements × DataSize,
           MPI_DOUBLE, 0);
10 end procedure

11 procedure multipole (numElements : in int;
       elements : in Element[]; mpcoef : out double[]) is
12     mpbuf : double[MpSize]; i, j, p, start, end : int;
13     for i ← 0 to MpSize − 1 do mpcoef[i] ← 0;
14     start ← rank × (numElements/nprocs);
15     end ← (rank + 1) × (numElements/nprocs);
16     if rank = nprocs − 1 then end ← numElements;
17     for i ← start to end − 1 do
18         for p ← 0 to PMax − 1 do
19             j ← f1(i, p, elements);
20             mpcoef[j] ← mpcoef[j] + f2(i, p, elements);
21         end
22     end
23     MPI_ALLREDUCE(mpcoef, mpbuf, MpSize,
           MPI_DOUBLE, MPI_SUM);
24     for i ← 0 to MpSize − 1 do mpcoef[i] ← mpbuf[i];
25 end procedure

26 procedure slave (elements : in Element[];
       mpcoef : in double[]; packsize : in int) is
27     packbuf : byte[PBSize];
28     databuf : double[DataSize × WorkSize];
29     indexbuf : int[WorkSize]; i, pos : int;
30     sendreq, recvreq : MPI_Request; status : MPI_Status;
31     MPI_RECV_INIT(indexbuf, WorkSize, MPI_INT, 0,
           MPI_ANY_TAG, recvreq);
32     MPI_SEND_INIT(packbuf, packsize, MPI_PACKED,
           0, 0, sendreq);
33     while true do
34         MPI_START(recvreq);
35         MPI_WAIT(recvreq, status);
36         if status.tag = Done then break;
37         for i ← 0 to WorkSize − 1 do
38             if indexbuf[i] ≠ −1 then
               databuf[i] ← f3(indexbuf[i], elements, mpcoef);
39         end
40         MPI_WAIT(sendreq, MPI_STATUS_IGNORE);
41         pos ← 0;
42         MPI_PACK(indexbuf, WorkSize, MPI_INT,
               packbuf, packsize, pos);
43         MPI_PACK(databuf, DataSize × WorkSize,
               MPI_DOUBLE, packbuf, packsize, pos);
44         MPI_START(sendreq);
45     end
46     MPI_REQUEST_FREE(recvreq);
47     MPI_WAIT(sendreq, MPI_STATUS_IGNORE);
48     MPI_REQUEST_FREE(sendreq);
49 end procedure

50 procedure master (numElements, packsize : in int;
       vel : out Vector[]) is
51     packbufs[nprocs][PBSize] : byte;
52     req : MPI_Request[nprocs]; status : MPI_Status;
53     i, job, proc, msgsize : int;
54     job ← 0;
55     for i ← 1 to 2 do
56         for proc ← 1 to nprocs − 1 do
               sendTask(proc, numElements, job);
57     end
58     for proc ← 1 to nprocs − 1 do
           MPI_RECV_INIT(packbufs[proc], packsize, MPI_PACKED,
               proc, MPI_ANY_TAG, req[proc − 1]);
59     MPI_START_ALL(nprocs − 1, req);
60     while job < numElements do
61         MPI_WAIT_ANY(nprocs − 1, req, proc, status);
62         MPI_GET_COUNT(status, MPI_PACKED, msgsize);
63         processPackBuf(packbufs[proc], msgsize, vel);
64         MPI_START(req[proc − 1]);
65         sendTask(proc, numElements, job);
66     end
67     for i ← 1 to 2 do
68         for proc ← 1 to nprocs − 1 do
69             MPI_WAIT(req[proc − 1], status);
70             MPI_GET_COUNT(status, MPI_PACKED, msgsize);
71             processPackBuf(packbufs[proc], msgsize, vel);
72             if i = 1 then MPI_START(req[proc − 1]);
73         end
74     end
75     for proc ← 1 to nprocs − 1 do
           MPI_REQUEST_FREE(req[proc − 1]);
76     for proc ← 1 to nprocs − 1 do
           MPI_SEND(0, 0, MPI_INT, proc, Done);
77 end procedure

78 procedure sendTask (proc, numElements : in int;
       job : in out int) is
79     indexbuf : int[WorkSize];
80     i : int;
81     for i ← 0 to WorkSize − 1 do
82         if job < numElements then indexbuf[i] ← job;
83         else indexbuf[i] ← −1;
84         job ← job + 1;
85     end
86     MPI_SEND(indexbuf, WorkSize, MPI_INT, proc, Work);
87 end procedure

88 procedure processPackBuf (packbuf : in byte[];
       msgsize : in int; vel : out Vector[]) is
89     pos, index, i : int;
90     indexbuf : int[WorkSize];
91     databuf : double[DataSize × WorkSize];
92     pos ← 0;
93     MPI_UNPACK(packbuf, msgsize, pos,
           indexbuf, WorkSize, MPI_INT);
94     MPI_UNPACK(packbuf, msgsize, pos,
           databuf, WorkSize × DataSize, MPI_DOUBLE);
95     for i ← 0 to WorkSize − 1 do
96         index ← indexbuf[i];
97         if index ≠ −1 then vel[index] ← databuf[i];
98     end
99 end procedure
```

Figure 4: procedure computeVelocity, parallel version

p and there are no buffered messages or active communication requests which have p as their destination, and (4) MPI_START is only invoked on inactive persistent requests (or on MPI_REQUEST_NULL). To verify these properties, we constructed a model we refer to as the *communication skeleton*, as it represents all the MPI communication calls in the original program but removes the content of most of the messages and abstracts away the floating-point computations of the program.

The approach we took to the construction of the communication skeleton is known as *counterexample-guided abstraction refinement* [1]. We say that a model is *conservative* if it encodes all possible executions of the program and perhaps also some additional executions which could never occur with the program. These additional or *spurious* executions usually result from the use of abstractions in the model. The advantage of using a conservative model is that if one proves a property holds on all executions of the model then the property is guaranteed to hold on all executions of the program. The strategy for constructing such a model can be summarized as follows:

1. start with a very abstract but conservative model M of the program P
2. run the model checker on M
3. if the model checker reports the property holds for M then terminate with the conclusion that the property holds for P
4. otherwise, examine the counterexample and determine whether it is spurious
5. if the counterexample is not spurious, terminate with the conclusion that the property does not hold for P
6. otherwise, *refine* M by adding sufficient detail to eliminate the spurious counterexample, and go to step 2.

To form our initial abstract model, we began by using the Doxygen tool [28] to construct the call graph of the program. We then identified all nodes in the graph corresponding to MPI communication functions (i.e., sends, receives, waits, and so on, but not packing and unpacking functions). We then focused our attention on the subgraph consisting of only those nodes u such that (1) u is reachable from the entry node, and (2) u is backwards reachable from an MPI communication node. For each node in the subgraph, we formed a corresponding "inline" function stub in the model.

To determine which variables to incorporate in the model, we began by including (1) all variables occurring in a rank argument of an MPI function, e.g., the source or destination argument for a send or receive operation, and (2) any variable occurring in a request handle argument of an MPI function. For example, the variable *proc* in the master as well as the array of requests *req* are modeled because of their occurrence in the first argument of the MPI_START on line 64 of Fig. 4. These variables were declared to be of an appropriate integer type and an array of MPI_Request, respectively.

We then proceeded to translate the statements in the modeled functions as follows. Statements that could directly affect the flow of control through the model or could modify the value of one of the modeled variables were incorporated into the model; all other statements were ignored.

```
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
```

(a) very abstract model

```
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
```

(b) refined to include *status*

Figure 5: Abstract models of main slave loop

This would invariably incorporate expressions that involved non-modeled variables. In these cases, either the unmodeled variables were abstracted out of the expression or the expression was replaced with a nondeterministic choice. In either case, this was done in a way that ensured the resulting model was conservative. For example, line 86 of Fig. 4 is translated as

```
MPI_Send(NULL, 0, MPI_POINT, proc, Work)
```

where `MPI_POINT` is a 0-byte type specific to Mpi-Spin. As another example, consider the loop of lines 33–45 of Fig. 4. In our first model, this was translated as in Fig. 5(a). Note that the inner loop (lines 37–39) and the packing statements are ignored, as these do not involve any modeled variables. Line 36, on the other hand, involves the variable *status*, which is not modeled, but can affect the flow of control by breaking out of the outer loop. The line is replaced by a nondeterministic choice in the model.

Not surprisingly, the initial model generates a spurious counterexample: e.g., in the initial model described above, one slave might break out of the loop in its first iteration and another could loop forever, leading to deadlocks (and other property violations). Of course, all of these counterexamples are spurious, as they could not occur in the actual program. Hence in the refinement step, we added the variable *status*, which controls precisely when a slave breaks out of the loop. The MPI-Promela code for the refined model is given in Fig. 5(b).

In the end, the following variables were incorporated into the skeleton: *sendreq* (line 30), *recvreq* (line 30), *status* (line 30), *req* (line 52), i (line 53), and *proc* (line 53).

Two variables that were *not* incorporated into the skeleton were *job* (in the master) and *numElements*. The test involving these variables on line 60 of Fig. 4 is replaced with a nondeterministic choice, a conservative abstraction. The fact that we could establish deadlock-freedom without *numElements* was a pleasant surprise, since it implies the following. For each configuration of the model for which we verified freedom from deadlock, the property holds *regardless of the value of numElements*.

The variable t from the ECCSVM algorithm of Fig. 3 is

```
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
```

Figure 6: Symbolic model of main slave loop

also omitted, though the test involving t can*not* simply be replaced with a nondeterministic choice. For if it were, then the resulting model would permit executions in which the processes executed the main loop different numbers of times, leading to deadlock. Since t_0, t_f, and Δt are all constant and identical on each process, it is clear that every process must execute the main loop the same number of times, so these deadlocking executions are spurious. In order to refine this model we let one process (the master) make the nondeterministic choice on whether or not to break out of the loop, and had this choice (represented as a 0 or 1) incorporated into the MPI_ALLREDUCE on line 23 of Fig. 4. In this way, we eliminated the spurious executions in which processes could loop different numbers of times without changing any other aspect of the model.

With this model, we were able to verify all of the data-independent properties for the configurations shown in Fig. 7(a). For each configuration, we give the number of states explored by Mpi-Spin, the amount of memory (in bytes) and time (in seconds) consumed. The option `-dl` was used; this tells Mpi-Spin that for each send, the possibility of synchronization should be explored and is necessary for an exhaustive search for deadlock. All runs were performed on a machine with two Dual-Core AMD Opteron Processors (each running at 2.613 GHz) with 64 GB of RAM.

4. FUNCTIONAL CORRECTNESS

The method for verifying the functional equivalence of a sequential and parallel program is described in [26] and we review it very briefly here. The basic idea is to represent the data *symbolically*: the input is represented as symbolic

nprocs	Time	Memory (MB)	States
2	<0:01	53	2389
3	0:01	56	62798
4	0:45	163	1.5×10^6
5	40:47	7708	3.5×10^7

(a) Results for the communication skeleton model

nprocs	Time	Memory (MB)	States	Sym. Ex.
2	<0:01	54	20473	425
3	0:19	96	691837	731
4	20:34	1177	2.0×10^7	988

(b) Results for the symbolic model

Figure 7: Verification statistics

constants $X_1, X_2, \ldots$ and the output is represented as symbolic expressions in the X_i. Floating-point operations are replaced by appropriate symbolic operations. A model of this type is made for both the sequential and parallel programs. The two models are then joined in a *composite model* in which the sequential model is executed first, then the parallel model is executed, and finally a sequence of assertions are checked to verify that the symbolic expressions output by the two models agree. Complexities arise when the programs contain branches on expressions involving the symbolic variables. These are modeled using nondeterministic choice and a *path condition*, i.e., a symbolic predicate recording the condition on the input that must hold in order for execution to follow a particular path. These constructs are supported in Mpi-Spin through the introduction of a symbolic type, a set of symbolic operations, and functions that determine whether a symbolic predicate is valid.

To construct these models for BlobFlow, we began with the communication skeleton model of Sec. 3 and augmented it according to the following scheme:

Step 1: locate maximal MPI-free segments of the code common to the sequential and parallel versions. Each such segment must perform a deterministic transformation of the program state and must execute in finite time, without throwing exceptions. As a first (abstract) approximation, each segment may be represented as an uninterpreted symbolic operation on the state. These are what we have denoted $f_1, f_2, \ldots$ above. Since the goal is to prove the equivalence of the two versions, it is generally not necessary to model the details of what goes on within such a segment.

Step 2: partition the state. That is, for each version, take the set of all variables in the program and decide which variables to include in the model. Call the set of chosen variables S. Then express S as a disjoint union $S = \cup_i S_i$ and for each i associate to S_i a model variable x_i. We will discuss below some guidelines for forming the partition.

Step 3: choose abstractions. That is, choose a type for each model variable x_i and decide on the operations that can be performed on it. For example, if S_i consists of a single variable of integer type, it can be represented as an integer variable in the model in a straightforward way. If S_i contains several variables of different types or variables that are not strongly related, it can be represented with an "uninterpreted" symbolic type, i.e., one with no associated

```
typedef struct {
  double x, y, strength, dx, dy;
  int    order;
} blob_external;
typedef struct {
  double sin2, cos2, sincos, costh, sinth, du11,
         du12, du21, u_xx, u_xy, u_yy, v_xx;
  int    refinecnt, nint;
} blobparms;
typedef struct {
  blob_external blob0, blob1, blob2, blob3, blob4;
} metablob;
extern metablob  mblob[MaxElt];
extern blobparms tmpparms[MaxElt];
```

Figure 8: Source code excerpt: *vel* corresponds to fields `dx` and `dy` of field `blob0` of `mblob`, and fields `du11`, `du12`, `du21`, `u_xx`, `u_xy`, `u_yy`, and `v_xx` of `tmpparms`

operations. If S_i consists of all the entries in some array of floating-point numbers, one could represent S_i using a single symbolic variable. Arrays of floating-point numbers of a given length admit operations such as element-wise addition and these operations can be represented symbolically in the same way that ordinary scalar operations are.

Step 4: translate the common segments. For each code segment identified in Step 1, determine (a) an upper bound In on the set of partitions that are read in that segment, and (b) an upper bound Out on the set of partitions that are modified. I.e., if there is any execution of the program in which a program variable x in partition S_i is read, then S_i must be included in In; similarly for Out. Say that the model variables corresponding to the partitions in In are $u_1, \ldots, u_m$ and the model variables corresponding to the partitions in Out are $v_1, \ldots, v_n$. Let $f_1, \ldots f_n$ be n distinct uninterpreted function symbols and translate the code segment as the simultaneous assignment

$$\textbf{forall } i \in \{1, \ldots, n\} : v_i \leftarrow (f_i\ u_1\ ...\ u_m).$$

The structure on the right is a symbolic expression with operator f_i and the m arguments $u_1, \ldots, u_m$.

Step 5: translate the remaining statements. For the statements not in common segments, use a more precise representation of the state transformation. For example, standard integer operations may be used for integer model variables, symbolic operations for the expressions involving symbolic variables, and so on. In all cases, the translation should be conservative. As in the communication skeleton model, nondeterministic choice may be necessary.

Having followed this scheme for the sequential and parallel symbolic models, we formed the composite model and entered the counterexample-guided refinement loop described in Sec. 3. The refinement strategies we used included (1) adding variables to the modeled set, (2) refining the variable partitions or the code segments, and (3) using a more precise representation of certain modeled variables. For example, if a spurious counterexample arose from using a single symbolic variable to represent an array, it was sometimes necessary to represent the array directly as an array of symbolic variables.

The way that the variables are partitioned can be quite complex. For example, our final symbolic model, which corresponds closely to the pseudocodes of Figs. 3 and 4, includes an array *vel* of symbolics. But the actual program variables represented by *vel* are spread out over various fields of two different arrays of C structs, as illustrated in Fig. 8. The reason for grouping these variables together is that they tend to be grouped together in the code. For example, in the actual code, the broadcast corresponding to that of line 9 of Fig. 4 actually packages together and sends the contents of those variables; these variables are all modified in the code corresponding to f_3; they are all read (but not modified) in the code corresponding to $\mathsf{integrate}$; none of them is read or modified in the code corresponding to f_1 or f_2, and so on. In general, we tended to group together variables that are treated the same way in the MPI communication constructs and the common blocks.

The common code segments we identified in forming the symbolic model include $\mathsf{integrate}$, f_1, f_2, and f_3, each of which has already been reduced to an abstract symbol in the pseudocodes. Another common block consists of lines 16–19 of Fig. 3, which are identical to lines 18–21 of Fig. 4. Identifying this common block led us to model the array *mpcoef* using a single symbolic variable admitting symbolic array operations. To see how this was done, notice that the operation performed by the common code may be represented as

$$mpcoef \leftarrow mpcoef + g(i, elements). \tag{13}$$

Here, $g(i, elements)$ corresponds to the array

$$\sum_{p=0}^{\mathsf{PMax}-1} \mathsf{scalarToArray}(f_1(i, p, elements), f_2(i, p, elements)),$$

where $\mathsf{scalarToArray}(k, x)$ is the array in which the k^{th} element is x and all other elements are 0. In the model, however, g is simply another uninterpreted symbolic operation and (13) is used to represent the common block in both the sequential and parallel models. It is not hard to see that this abstraction is sufficiently precise to establish the equivalence of the sequential and parallel versions of the computation of *mpcoef*.

The remaining data in the pseudocode is modeled as follows. The array *elements* is modeled by a single, uninterpreted symbolic variable; *vel* is modeled by an array of uninterpreted symbolics; *indexbuf* and the byte arrays *packbuf* and *packbufs* are modeled exactly as they are in the pseudocode; *databuf* is modeled as an array of symbolic reals. The other integer variables (including *job* and *numElements*) are also modeled directly as integers; in particular the integer variable *numElements* is updated at each iteration using a nondeterministic choice. Constant values had to be specified for $\mathsf{WorkSize}$ and MaxElt, and an explicit upper bound was imposed on the number of iterations of the outermost loop.

An excerpt of the symbolic model, corresponding to the same main slave loop illustrated in Fig. 5, is given in Fig. 6.

Results for verification of functional equivalence can be found in Fig 7(b). For these runs, we used $\mathsf{WorkSize} = 2$ and $\mathsf{MaxElt} = 4s + 1$, where s is the number of slaves. The number of outermost loop iterations was bounded by 3. The symbolic mode used was "real," allowing Mpi-Spin to use all properties of real numbers to simplify symbolic expressions,

```
1  start ← 1;
2  while true do
3  |  MPI_START(recvreq);
4  |  MPI_WAIT(recvreq, status);
5  |  if status.tag = Done then
6  |  |  MPI_WAIT(sendreq, MPI_STATUS_IGNORE);
7  |  |  break;
8  |  end
9  |  for i ← 0 to WorkSize − 1 do
10 |  |  if start = 0 then
   |  |  |  MPI_WAIT(sendreq, MPI_STATUS_IGNORE);
11 |  |  else start ← 0;
12 |  |  if indexbuf[i] ≠ −1 then  databuf[i] ← ···;
13 |  end
14 |  pos ← 0;
15 |  MPI_PACK(indexbuf, ..., packbuf, packsize, pos);
16 |  MPI_PACK(databuf, ..., packbuf, packsize, pos);
17 |  MPI_START(sendreq);
18 end
19 MPI_REQUEST_FREE(recvreq);
20 MPI_REQUEST_FREE(sendreq);
```

Figure 9: An early version of the slave main loop

including associativity and commutativity of multiplication and addition. The right-most column gives the total number of distinct symbolic expressions created in the course of the search.

5. MODIFICATIONS

While constructing the models, we noticed several places where improvements could be made to the parallel version of the code. In most cases, these improvements were minor. For example, the call to MPI_ALLREDUCE on line 23 of Fig. 4 was previously a call to MPI_REDUCE followed by a call to MPI_BCAST. However, some such improvements were rather subtle, and required careful scrutiny of the code.

Making these improvements taught us the following. First, since a model abstracts away many superfluous details of the code, it is often easier to see areas for improvement in the model than in the code itself. Second, Mpi-Spin makes it very easy to experiment with such improvements. As described in Sec. 1.2, we followed a protocol in which all modifications were first made and verified in the model before altering the code. This approach allowed us to quickly verify that an improvement did not "break" any of the correctness properties we had established.

We now describe one instance where Mpi-Spin prevented us from introducing a defect into the code.

The original version of the slave main loop is summarized in Fig. 9. Notice that it features an odd test in lines 10–11. (We believe this to be an artifact of an earlier version of the code.) In particular, the code performs multiple wait calls on the same request handle. Since the handle is set to MPI_REQUEST_NULL after the return of the first such call, the subsequent calls, while not strictly incorrect (they are essentially treated as "no-ops"), are certainly unnecessary.

To simplify the code, we removed the variable *start*, along with lines 1, 10, and 11, and we replaced lines 5–8 with

```
5  MPI_WAIT(sendreq, MPI_STATUS_IGNORE);
6  if status.tag = Done then break;
```

We then verified the new versions of the two models and determined that all correctness properties still held.

Recall that the main advantage of using MPI nonblocking operations is to maximize overlap of computation and communication. The modified code just described can therefore be improved by delaying the wait until the point just before *packbuf* (the buffer associated with *sendreq*) is reused on lines 14–16 of Fig. 9. We therefore moved the call to MPI_WAIT to this point. However, when we ran Mpi-Spin on this new version of the communication skeleton, it reported an error when *sendreq* was freed before the request had completed. (While the MPI Standard allows this in certain situations, Mpi-Spin takes the stricter view that persistent requests should always be completed before being freed. In our case, this defect would certainly lead to failure if the slave deallocated the buffer and returned before the request had completed.) We fixed the problem by inserting a call to MPI_WAIT between the two calls to MPI_REQUEST_FREE on lines 19 and 20 of Fig. 9, resulting in what now appears in Fig. 4. Rerunning Mpi-Spin on our models with this change indicated that the problem was then fixed.

6. CONCLUSION

In this paper, we have reported on our experience using the model checker Mpi-Spin to verify correctness properties of a nontrivial parallel scientific program. We were able to verify generic data-independent 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 precisely 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 first requires one to locate segments of code common to the sequential and parallel versions of the program and abstract these segments using uninterpreted symbolic operations. The second involves partitioning the data of the programs and assigning a symbolic variable to each partition. The partitions can be quite complex, involving entire arrays or various fields of different data structures.

What is clearly required to move the field forward is 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 impact analysis, which could be used to determine that certain variables must be incorporated into the models. In other contexts, the counter-example-driven refinement loop has been automated using theorem-proving techniques, and it is possible that similar techniques could be adopted for MPI-based programs. These are some of the many avenues of future research.

7. REFERENCES

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[2] A. J. Chorin. Numerical study of slightly viscous flow. *J. Fluid Mech.*, 57:785–796, 1973.

[3] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, 1999.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE '00: Proc. of the 22nd Intl. Conference on Software Engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[5] G.-H. Cottet and P. D. Koumoutsakos. *Vortex methods: Theory and practice*. Cambridge University Press, Cambridge; New York, 2000.

[6] G. H. Cottet and S. Mas-Gallic. Particle methods to solve transport diffusion equations - Part II: the Navier-Stokes system. *Numer. Math.*, 57:805, 1990.

[7] A. Gharakhani. A higher order vorticity redistribution method for 3-D diffusion in free space. Technical Report SAND2000-2505, Sandia National Laboratories, Albuquerque, NM 87185, Oct 2000.

[8] C. Greengard. The core spreading vortex method approximates the wrong equation. *J. Comp. Phys.*, 61:345–348, 1985.

[9] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.*, 73:325–348, 1987.

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

[11] F. John. *Partial differential equations*. Springer-Verlag, 4th edition, 1982.

[12] A. Leonard. Vortex methods for flow simulation. *J. Comp. Phys.*, 37:289–335, 1980.

[13] S. Mas-Gallic. Deterministic particle methods: Diffusion and boundary conditions. In C. Greengarde, editor, *Lectures in Applied Mathematics: Vortex Methods and Vortex Dynamics*, pages 433–465. American Mathematical Society, 1991.

[14] R. Palmer, G. Gopalakrishnan, and R. M. Kirby. Semantics driven partial-order reduction of MPI-based parallel programs. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD V)*, London, July 2007.

[15] L. Rosenhead. The point vortex approximation of a vortex sheet. *Proceedings of the Royal Society of London Series A*, 134:170–192, 1931.

[16] L. F. Rossi. Resurrecting core spreading methods: A new scheme that is both deterministic and convergent. *SIAM J. Sci. Comp.*, 17(2):370–397, 1996.

[17] L. F. Rossi. Achieving high-order convergence rates with deforming basis functions. *SIAM J. Sci. Comput.*, 26(3):885–906, 2005.

[18] L. F. Rossi. A comparative study of lagrangian methods using axisymmetric and deforming blobs. *SIAM J. Sci. Comput.*, 27(4):1168–1180, 2006.

[19] L. F. Rossi. Evaluation of the Biot-Savart integral for deformable elliptical gaussian vortex elements. *SIAM J. Sci. Comput.*, 28(4):1509–1532, 2006.

[20] P. G. Saffman. *Vortex Dynamics*. Cambridge University Press, 1992.

[21] J. A. Sethian. A brief overview of vortex methods. In K. Gustafson and J. A. Sethian, editors, *Vortex Methods and Vortex Motion*. SIAM Frontiers in Applied Mathematics, 1990.

[22] S. F. Siegel. The Mpi-Spin web page. `http://vsl.cis.udel.edu/mpi-spin`, 2007.

[23] S. F. Siegel. Model checking nonblocking MPI programs. In B. Cook and A. Podelski, editors, *Verification, Model Checking, and Abstract Interpretation: 8th Intl. Conference (VMCAI 2007)*, volume 4349 of *LNCS*, pages 44–58, 2007.

[24] 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 Intl. SPIN Workshop*, volume 2989 of *LNCS*, pages 286–303. Springer-Verlag, 2004.

[25] S. F. Siegel and G. S. Avrunin. Modeling wildcard-free MPI programs for verification. In *Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*, pages 95–106. ACM Press, 2005.

[26] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In L. L. Pollock and M. Pezzé, editors, *Proceedings of the ACM SIGSOFT Intl. Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 157–168. ACM, 2006.

[27] I. Stakgold. *Green's fnctions and boundary value problems*. Wiley-Interscience, 1979.

[28] D. van Heesch. Doxygen. `http://www.doxygen.org`, 2007.