Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs

STEPHEN F. SIEGEL
University of Delaware
ANASTASIA MIRONOVA
University of Utah
and
GEORGE S. AVRUNIN and LORI A. CLARKE
University of Massachusetts

We present a method to verify the correctness of parallel programs that perform complex numerical computations, including computations involving floating-point arithmetic. This method requires that a sequential version of the program be provided, to serve as the specification for the parallel one. The key idea is to use model checking, together with symbolic execution, to establish the equivalence of the two programs. In this approach the path condition from symbolic execution of the sequential program is used to constrain the search through the parallel program. To handle floating-point operations, three different types of equivalence are supported. Several examples are presented, demonstrating the approach and actual errors that were found. Limitations and directions for future research are also described.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking, validation*

General Terms: Verification
Additional Key Words and Phrases: Finite-state verification, numerical program, floating-point, model checking, concurrency, parallel programming, high performance computing, symbolic execution, MPI, Message Passing Interface, Spin

This is a revised and extended version of a paper presented at the 2006 International Symposium on Software Testing and Analysis (ISSTA 2006). This research was partially supported by the National Science Foundation under awards CCF-0427071, CCR-0205575, and CCF-0541035, and by the U.S. Department of Defense/Army Research Office under awards DAAD19-01-1-0564 and DAAD19-03-1-0133. We also wish to thank the Computing Research Association's CRA-W Distributed Mentor Program and the College of Natural Sciences and Mathematics at the University of Massachusetts for sponsoring and funding Mironova during the summer of 2004. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation or the U.S. Department of Defense/Army Research Office.
Authors' addresses: S. Siegel, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716; email: siegel@cis.udel.edu; A. Mironova, Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT 84112; email: mironova@sci.utah.edu; G. Avrunin, L. Clarke, Department of Computer Science, University of Massachusetts, Amherst, MA 01003; email: {avrunin, clarke}@cs.umass.edu.

1. INTRODUCTION

In domains that require extensive computation, such as high-performance scientific computing, a program may be divided up among several processors working in parallel in order to reduce the overall execution time and increase the total amount of memory available to the program. The process of "parallelizing" a sequential program is notoriously difficult and error-prone. Attempts to automate this process have met with only limited success, and thus most parallel code is still written by hand. The developers of such programs expend an enormous amount of effort in testing, debugging, and a variety of ad hoc methods to convince themselves that their code is correct. Hence any techniques that can help establish the correctness of these programs or find bugs in them would be very useful.

In this paper we focus on parallel numerical programs. By a *numerical* program we mean a program whose primary function is to perform a numerical computation. For the purposes of this paper, we will think of a numerical program as taking a vector of (usually floating-point) numbers for input and producing another such vector as output. Examples include programs that implement matrix algorithms, simulate physical phenomena, or model the evolution of a system of differential equations. We are interested in techniques that can establish the correctness of a program of this type—i.e., prove that the program always produces the correct output for any input—or exhibit appropriate counterexamples if the program is not correct.

The usual method for accomplishing this—testing—has two significant drawbacks. In the first place, it is usually infeasible to test more than a tiny fraction of the inputs that a parallel numerical program will encounter in use. Thus, testing can reveal bugs, but, as is well-known, it cannot show that the program behaves correctly on the inputs that are not tested. Secondly, the behavior of concurrent programs, including most parallel numerical programs, typically depends on the order in which events occur in different processes. This order depends in turn on the load on the processors, the latency of the communication network, and other such factors. A parallel numerical program may thus behave differently on different executions with the same input vector, so getting the correct result on a test execution does not even guarantee that the program will behave correctly on another execution with the same input.

The method proposed here, which combines model checking with symbolic execution in a novel way, does not exhibit these two limitations: it can be used to show that a parallel numerical program produces the right result on any input vector, regardless of the particular way in which the events from the concurrent processes are interleaved.

In attempting to apply model checking techniques in this setting, two issues immediately present themselves. First, these techniques require the programmer to supply a finite-state model of the program being checked. But numerical programs typically deal with huge amounts of floating-point data, and the very nature of our problem dictates that we cannot just abstract this data away. Hence it is not obvious how to construct appropriate finite-state models of the programs without greatly exacerbating the state explosion problem. The second issue concerns the nature of the property we wish to check: the statement that the output produced

by the program is correct must be made precise and formulated in some way that is amenable to model checking tools.

We deal with the first issue by modeling computations in the programs *symbolically*. That is, in our model, the input is considered to be a vector of symbolic constants x_i, and the output is some vector of symbolic expressions in the x_i. The numerical operations in the program are replaced by appropriate symbolic operations in the model. Furthermore, each symbolic expression is represented by a single integer index into a table, which prevents the blowup of the size of the state vector and makes it possible to easily express the model in the language of standard model checking tools, such as SPIN [Holzmann 2004].

We deal with the second issue by requiring that the user provide a sequential version of the program to be verified, which will serve as a specification for the parallel one. The model checker will be used to show that the parallel and sequential programs are *functionally equivalent*, i.e., that they produce the same output on any given input. Of course, this means that our method only reduces the problem of verifying a parallel program to the problem of verifying a sequential one. However, most problems in this domain have a much simpler sequential solution than a parallel one, and it is already common practice for developers of scientific software to begin with a sequential version of the program or to construct one, for testing and other purposes. Moreover, as described below, our method provides information that can help verify the correctness of the sequential program as well.

Another issue that arises in this approach is the fact that most numerical programs contain branches on conditions that involve the input. Such programs may be thought of as producing a set of cases, each case consisting of a predicate on the input and the corresponding symbolic output vector. We use the model checker to explore all possible paths of the sequential program (up to some loop bound), and for each such path we record the *path condition pc*, the Boolean-valued symbolic expression on the input that must hold for that path to have been followed. The model of the parallel program is engineered to take as input not only the symbolic input vector, but the path condition *pc* as well. The model checker is then used to explore all possible paths of the parallel program that are consistent with *pc*. If, for every *pc*, the result produced by the parallel program always agrees with the result produced by the sequential one, the two programs must be equivalent (up to the specified loop bound).

The method is described in detail in Section 2. In Section 3, we give two reduction theorems for models of parallel numerical programs that help make the method of Section 2 usable with larger examples. Using SPIN, we have applied the method to four parallel numerical programs; we describe this experience and present some data that arose from it in Section 4. In Section 5, we discuss the problem of constructing models suitable for verification and an extension to SPIN intended to address some of the difficulties in extracting such models from code. Section 6 discusses related work, and Section 7 presents some conclusions and directions for future work.

2. METHODOLOGY

We consider a parallel numerical program P_{par} that consists of a fixed number of parallel processes. We write n for the number of parallel processes. We assume that

these processes have no shared memory and communicate only through message-passing functions such as those provided by the *Message Passing Interface* (MPI) [Message Passing Interface Forum 1995; 1997]. (Though much of what follows will apply equally to other communication systems, or even to shared memory systems, MPI has become the *de facto* standard for high performance computation, particularly in the domain of scientific computation.) We assume we are given a sequential program P_{seq}, which serves as the specification for P_{par}. We also assume that both P_{seq} and P_{par} terminate normally on every input, a property that can often be verified using more traditional model checking techniques [Siegel and Avrunin 2004; Siegel 2005]. In some cases, we may also have to impose a small upper bound on the number of iterations of certain loops in a program, to ensure that the model we build will not have an inordinately large (or even infinite) number of states.

Note that for P_{par} to be functionally equivalent to P_{seq} the output of each function must be a deterministic function of its input. Hence, if one of these programs can produce different outputs on two executions with the same input, we consider the functional equivalence property to be violated, regardless of the behavior of the other program. Our method will in fact detect this kind of violation, as well as the more typical violations in which both programs behave as functions but disagree on some input.

To simplify the presentation, we begin by explaining the method under the assumption that neither program contains branches on expressions involving variables that are modeled symbolically. After this we consider some numerical issues that arise from the fact that floating-point arithmetic is only an approximation to the arithmetic of the real numbers, and finally we describe the general approach, in which branches on symbolically modeled expressions are allowed.

2.1 A simple example

To illustrate the method, we consider the example of Figure 1. This sequential C code takes the product of an $N \times L$ matrix A and an $L \times M$ matrix B and stores the result in the $N \times M$ matrix C. We can consider this to be a numerical program for which the input vector consists of the $NL + LM$ entries for A and B, and the output vector consists of the NM entries of C at termination. There are many ways to parallelize P_{seq}, but we will consider the one shown in Figure 2, which is adapted from Gropp et al. [1999] and uses MPI functions for interprocess communication. Each process should be thought of as executing its own copy of this code, in its own local memory. A process may also obtain its *rank* (a unique integer between 0 and $n-1$) from the MPI infrastructure. For this code, which uses a master-slave approach to achieve automatic load-balancing, we assume that $N \geq n-1 \geq 1$, and that all three matrices are stored in the local memory of the process of rank 0 (the master). To compute the product, the master will distribute the work among the processes of positive rank (the slaves).

We assume that each slave process already has a copy of B in its local memory. The master begins by sending the first row of A to the first slave, the second row of A to the second slave, and so on, until the first $n-1$ rows of A have been handed out. A slave, after receiving a row vector of length L from the master, multiplies it by B, and sends back the resulting row vector of length M to the master. The

```
double A[N][L], B[L][M], C[N][M];
                 ⋮
int i,j,k;
for (i=0; i<N; i++)
  for (j=0; j<M; j++) {
    C[i][j] = 0.0;
    for (k=0; k<L; k++)
      C[i][j] += A[i][k]*B[k][j];
  }
```

Fig. 1. Sequential matrix multiplication code

```
int rank,nprocs,i,j,numsent,sender,row,anstype;
double buffer[L], ans[M];
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) { /* I am the master */
  numsent=0;
  for (i=0; i<nprocs-1; i++) {
    for (j=0; j<L; j++)
      buffer[j] = A[i][j];
    MPI_Send(buffer, L, MPI_DOUBLE, i+1, i+1, MPI_COMM_WORLD);
    numsent++;
  }
  for (i=0; i<N; i++) {
    MPI_Recv(ans, M, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
       &status);
    sender = status.MPI_SOURCE;
    anstype = status.MPI_TAG-1;
    for (j=0; j<M; j++)
      C[anstype][j] = ans[j];
    if (numsent<N) {
      for (j=0; j<L; j++)
        buffer[j] = A[numsent][j];
      MPI_Send(buffer, L, MPI_DOUBLE, sender, numsent+1, MPI_COMM_WORLD);
      numsent++;
    }
    else MPI_Send(buffer, 1, MPI_DOUBLE, sender, 0, MPI_COMM_WORLD);
  }
} else { /* I am a slave */
  while (1) {
    MPI_Recv(buffer, L, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    if (status.MPI_TAG==0) break;
    row = status.MPI_TAG-1;
    for (i=0; i<M; i++) {
      ans[i] = 0.0;
      for (j=0; j<L; j++)
        ans[i] += buffer[j]*B[j][i];
    }
    MPI_Send(ans, M, MPI_DOUBLE, 0, row+1, MPI_COMM_WORLD);
  }
}
```

Fig. 2. Parallel matrix multiplication code

master waits at a receive statement that will accept a message from any process (we will refer to a statement of this kind as a *wildcard receive*). After one or more messages have arrived, the master chooses one for reception, copies the row vector received into the appropriate row in C, sends the next row of A to the slave that had just returned the result, and returns to the wildcard receive. It continues in this way until all the rows of A have been handed out. After that point, whenever a slave sends in a result, the master sends back a termination message to that slave. After all results have come in, and the last termination message has been sent out, C should contain the product of A and B, and all processes should terminate normally.

The first step of our method is to create a finite-state model M_{seq} of P_{seq} in Promela, the input language for Spin. The model will use symbolic expressions in place of the floating-point values that arise in P_{seq}. (Symbolic expressions are also used in our method to represent certain integer and Boolean values, though one can often model these sorts of values in a more direct way.) A symbolic expression may be thought of as a tree-like structure in which the leaf nodes are either literals or symbolic constants. The symbolic constants are denoted $x_0, x_1, \ldots$ and correspond to the components of the input vector. Each non-leaf node in the tree is associated with a (unary or binary) operator, e.g., $+,-,*,/$, or any other arithmetic operator that occurs in the program.

Each numerical operation in the program involving a symbolically modeled variable is replaced by an operation on symbolic expressions in the model. The symbolic operation simply forms a new tree rooted at this operator with subtrees representing the one or two operands. We will use the usual infix notation to denote symbolic expressions, but note that no interpretation is given to the operations, and none of the usual rules of real arithmetic (associativity, commutativity, etc.) hold. For example, for the matrix multiplication program with $N = L = M = 2$, if the initial symbolic values for A and B are given by

$$A = \begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \end{pmatrix}, \quad B = \begin{pmatrix} x_4 & x_5 \\ x_6 & x_7 \end{pmatrix},$$

then the final value of $C[0][0]$ will be the symbolic expression $(0.0 + x_0x_4) + x_1x_6$, which does not equal the symbolic expression $x_0x_4 + x_1x_6$. The symbolic structure can be represented in a language such as Promela using standard data structures such as integer arrays, but as shown below this is not really necessary.

The next step of our method is to create a finite-state model M_{par} of P_{par}. To do this we use Spin processes to represent the processes of the parallel program and Spin channels to transfer messages between processes, using techniques such as those of Siegel and Avrunin [2004] and Siegel and Avrunin [2005]. The arithmetic operations are represented symbolically, just as in the sequential case. Finally, a *composite model* is formed, in which first M_{seq} is executed in its own Spin process, then M_{par} is executed using n additional Spin processes, and finally a series of Spin assertions are checked to verify that the final symbolic entries of the copy of C generated by M_{seq} agree with those generated by M_{par}. Spin is then used in verification mode to explore all possible paths of the composite model and to verify that the assertions are never violated. In the matrix multiplication example, there are many such paths, due to all the different possible orders in which the slaves can

return their results to the master.

Now, the method described above may work for small models, but it has a serious drawback. For a typical program, the size of the symbolic expressions—and therefore the size of the structure used to represent the state of the model—can quickly blow up. Like most model checking tools, SPIN stores the set of states it has encountered as it searches the state space of the model, and the amount of memory required to represent this set is usually the main barrier to a successful completion of the search. Since the memory required to represent the set is approximately the product of the number of states and the size of the structure used to represent a single state, the method we have proposed has little chance of scaling.

To ameliorate this problem, we use a form of *value numbering* to reduce the memory needed to represent a symbolic expression and use *subexpression sharing* to reduce the total number of expressions and facilitate expression comparison. Using this approach, the floating-point values in the original programs are represented by integer indices that refer to entries in a static *symbolic expression table*. (By *static*, we do not mean that the table never changes, but that it is shared by every state in the state space, just as a static variable in a Java class is shared by all instances of that class.) The table contains one entry for every expression (including every subexpression of every expression) that is encountered during the search of the state space of the composite model. An entry for a binary expression is a triple in which the first component is an operator code, the second component is an integer referring to an (earlier) entry in the table corresponding to the left operand, and the third component is an integer corresponding similarly to the right operand. The entry for a unary expression is similar but has only two components. An entry for a leaf expression has either the form $(\mathtt{X}, i)$, corresponding to the symbolic constant x_i, or $(\mathtt{L}, \alpha)$, where α is a floating-point number, corresponding to a literal value.

The table is initialized by entering the literal values 0 and 1, as these are needed by many models and by many of the routines in our symbolic manipulation package. Next, the symbolic constants for the input vector are entered into the table. Other entries to the table are made as needed as symbolic operations are performed during the search of the state space. The arithmetic operations are modeled by operations on integers that refer to entries in the table. The operation that performs addition, for example, takes two integers i and j, and first looks in the table to see if the triple $(+, i, j)$ has already been entered. If it has, the addition operation returns the index of that triple. If it has not, it appends that triple to the end of the table and returns the new index. This guarantees that each symbolic expression encountered during the search has a *unique* entry in the table. The practical consequence of this is that the equality of such expressions can be determined by simply comparing their indices in the table.

The table that is constructed during the verification of the 2×2 matrix multiplication example is excerpted in Figure 3. At the end of execution of M_{seq}, the table will have 26 entries. The variable $C[0][0]$ will be initialized to the value 0, the index of the expression 0.0 in the table, then set to the index 11, and finally set to 13, the index of the expression $(0.0 + x_0x_4) + x_1x_6$. Hence one of the assertions that will be checked is that, at the termination of any execution of M_{par}, the variable $C[0][0]$ in the master process will also be 13.

i	e_i	interpretation
0	$(\texttt{L}, 0.0)$	0.0
1	$(\texttt{L}, 1.0)$	1.0
2	$(\texttt{X}, 0)$	x_0
3	$(\texttt{X}, 1)$	x_1
⋮	⋮	⋮
9	$(\texttt{X}, 7)$	x_7
10	$(*, 2, 6)$	x_0x_4
11	$(+, 0, 10)$	$0.0 + x_0x_4$
12	$(*, 3, 8)$	x_1x_6
13	$(+, 11, 12)$	$(0.0 + x_0x_4) + x_1x_6$
14	$(*, 2, 7)$	x_0x_5
⋮	⋮	⋮
25	$(+, 23, 24)$	$(0.0 + x_2x_5) + x_3x_7$

Fig. 3. Symbolic expression table for 2×2 matrix multiplication

In this example, when the state space of M_{par} is explored, no new entries are ever made, because all of the expressions generated can already be found in the table. (In more complicated examples, however, the parallel program may also add new expressions.) In fact, for non-trivial sizes (see Section 4), SPIN can verify that the assertions are never violated, establishing the equivalence of the two programs.

2.2 Numerical Issues

Floating-point arithmetic is only an approximation to the arithmetic of real numbers, and many of the standard properties of the latter do not necessarily hold for the former [Goldberg 1991]. (The exact differences depend on which particular floating-point arithmetic one uses.) In the matrix multiplication example, the symbolic expressions computed by the sequential and parallel models are exactly the same, which guarantees that the programs being modeled will always produce the same results, no matter what arithmetic is used to execute the programs (assuming, of course, that the arithmetic functions are deterministic). There are cases, however, where two models may compute expressions that are not exactly the same, but which may be close enough for particular needs. For example, in most floating-point arithmetics—including all those that conform to the IEEE 754 or 854 standards [IEEE 1985; 1987]—the expressions $0 + f$ and f must always evaluate to the same floating-point value, for any floating-point expression f. Hence, if the symbolic results produced by the two models are the same "up to" the relation that identifies any symbolic expression e with the symbolic expression $0 + e$, we are still guaranteed that the two programs will produce the exact same floating-point results on any platform implementing IEEE arithmetic.

In general, let $\sim$ be an equivalence relation on the set $S(X)$ of symbolic expressions over a set of symbolic constants $X = \{x_1, x_2, \ldots\}$. We assume that $\sim$ is *operation-preserving*, i.e., that

$$e_1 \sim e_2 \wedge f_1 \sim f_2 \Rightarrow e_1 + f_1 \sim e_2 + f_2$$

holds for all $e_i, f_i \in S(X)$, and that similar statements hold for the other operators.

This means that each operation induces an operation on the set of equivalence classes $\bar{S}(X) \equiv S(X)/\sim$, and so all of the arithmetic and comparisons for equality in the models may be thought of as taking place in $\bar{S}(X)$.

Note that in $\bar{S}(X)$, it is no longer trivial to test for the equality of two elements. As shown in Section 4.1, our implementation deals with this by performing certain simplifications on an expression before it is entered into the symbolic table. This is not quite as strong as reducing the expression to a true normal form (i.e., to a unique representative of its equivalence class), but it is inexpensive and provides sufficient precision for the examples we have explored.

Each operation-preserving equivalence relation yields a different notion of program equivalence. We have identified three that we found useful in our implementation, though the same approach can certainly be used for other relations. The three relations are as follows:

—*Herbrand equivalence*: this is the strongest, and therefore most desirable, notion of equivalence. Two symbolic expressions are Herbrand equivalent if and only if they are exactly equal. As we have seen, two Herbrand equivalent programs will produce the same results, independently of the way in which the arithmetic operations are implemented.

—*IEEE equivalence*: this is a slightly weaker relation. There are a number of identities for real arithmetic that also hold for IEEE arithmetic, e.g, $x + y = y + x$, $xy = yx$, and $1x = x1 = x + 0 = 0 + x = x/1 = x$. Two elements of $S(X)$ are considered to be equivalent if one can be transformed to the other by a finite sequence of transformations corresponding to such identities. Two IEEE equivalent programs must produce the same output on any platform implementing IEEE arithmetic. Of course, they would also produce the same output if the arithmetic were exactly real arithmetic.

—*Real equivalence*: this is weaker still. Two elements of $S(X)$ are considered to be equivalent if one can be transformed to the other using any identities of real numbers, including those that do not hold for IEEE arithmetic, such as the associativity of addition or multiplication, and the distributive property. Two real equivalent programs would produce the same results if all computations were performed as real arithmetic, but they may produce different results when run on an actual computer, even one that implements IEEE arithmetic. The differences may be slight, but in some situations the error can mushroom and the two can differ greatly.

The sad truth is that real equivalence is often the best that we can hope for. This is because there are many common scenarios that rely on associativity or some other property that does not hold for IEEE arithmetic. For example, it is often the case that one needs to compute a sum of floating-point variables that reside in the local memory of different processes and return the result to every process. MPI provides a convenient way to do this: one just calls `MPI_Allreduce` with a parameter specifying that the *reduction operation* is to be floating-point addition. However, the MPI Standard states that the implementation may add the values in any order—the implementation is not even required to use the same order twice. Hence an MPI program making one call to `MPI_Allreduce` may produce

different results when run twice on the same input, even if the execution platform uses IEEE arithmetic. Because the MPI functions that perform reductions do not specify the order in which the arithmetic operations are applied, real equivalence is usually the best that can be achieved for programs that use these functions.

For programs that are real but not IEEE equivalent, difficult issues may arise in creating test oracles or in determining whether the error (the difference between the actual results and what the results would have been if real arithmetic had been used) falls within acceptable bounds. Such questions are simply beyond the scope of our method. Other investigations have attempted to deal precisely with floating-point errors, in different circumstances; see, for example, Martel [2005] and the references cited there.

As mentioned above, sometimes we might want to model integer variables symbolically. This requires a small modification to the above framework, in which we associate a type (either integer or floating-point) to each symbolic constant and, consequently, a type to each symbolic expression. The notion of Herbrand equivalence is unchanged, but for both IEEE and real equivalence we allow all the usual rules of integer arithmetic, including commutativity, associativity, and the distributive property for integer addition and multiplication.

2.3 The general case

The method used for the matrix multiplication example applies to any program with no branches on expressions that involve the symbolically modeled variables. We now drop this restriction. To illustrate the general case, we use the program in Figure 4, which implements the Gaussian elimination algorithm to transform an $N \times M$ matrix to its reduced row echelon form. The input vector for this program consists of the NM initial values of the matrix entries, and the output vector consists of the NM final values of those entries.

Recall that an important step in this algorithm is to locate, at each stage, a *pivot row*, i.e., a row at or below the current top row that contains a non-zero entry in the current column. This is accomplished in the sequential code by looping over the rows, starting at `top` and working down, looking for a non-zero entry. If none is found, the algorithm moves to the next column and loops over the rows again. This continues until the first non-zero entry is found, or until we fall off the bottom or the right side of the matrix. (For simplicity, we will ignore the question of choosing the nonzero entry of largest absolute value.)

In the parallel version (Figure 5), we assume that $n = N$ (where n is the number of parallel processes) and that the i^{th} row of the matrix is stored in the local memory of the process of rank i. The pivot row is determined in a very different way, using a call to `MPI_Allreduce` in which the reduction operation returns the minimum of the given values. Each process contributes an integer to this communication, according to the following rule: if its entry in position `col` is 0 or the rank of the process is less than `top`, the process contributes the integer n, else it contributes its rank. The call to `MPI_Allreduce` results in the minimum of all these contributions being stored in the variable `row` of each process. If, after this communication completes, `row` is less than n, then each process knows that the process of rank `row` will be used as the next pivot row and breaks out of the pivot-searching loop, else the search for a pivot continues. Additional communication is used to exchange the top and

```
double matrix[N][M];
                ⋮
int top,col,row,i,j;
double pivot,tmp;
for (top=col=0; top<N && col<M; top++, col++) {
  pivot = 0.0;
  for (; col<M; col++) {
    for (row=top; row<N; row++) {
      pivot = matrix[row][col];
      if (pivot!=0.0) break;
    }
    if (pivot!=0.0) break;
  }
  if (col>=M) break;
  if (row!=top)
    for (j=0; j<M; j++) {
      tmp = matrix[top][j];
      matrix[top][j] = matrix[row][j];
      matrix[row][j] = tmp;
    }
  for (j=col; j<M; j++) {
    matrix[top][j] /= pivot;
  }
  for (i=0; i<N; i++) {
    if (i!=top) {
      tmp = matrix[i][col];
      for (j=col; j<M; j++) {
        matrix[i][j] -= matrix[top][j]*tmp;
      }
    }
  }
}
```

Fig. 4. Sequential Gaussian elimination code

pivot rows and to broadcast the pivot row.

Because of the branch expressions that involve the floating-point input (e.g., `pivot!=0.0`), the sequential program can follow different paths, depending on the input. Consider, for example, the case where $N = M = 2$, and the matrix is initially $\begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \end{pmatrix}$. If $x_0 \neq 0$ and $x_3 - x_2(x_1/x_0) = 0$ then the program will follow a path resulting in the final value of $\begin{pmatrix} 1 & x_1/x_0 \\ 0 & 0 \end{pmatrix}$ (assuming IEEE arithmetic is used). If instead $x_0 \neq 0$ and $x_3 - x_2(x_1/x_0) \neq 0$, the final result is the identity matrix. In fact, in this 2×2 case, there are 7 possible paths through the sequential program. For each path there is an associated *path condition*, the predicate on the input vector that must hold in order for that path to be followed, and a resulting symbolic output vector. (Notice it is possible for two different paths to yield the same output: the path arising from the condition $x_0 = 0 \wedge x_2 \neq 0 \wedge x_1 \neq 0$ also yields the identity matrix.)

The path condition *pc* is represented symbolically in a way similar to that used

```
double matrix[M];
                ⋮
int top,col,row,j,rank,nprocs;
double pivot,tmp;
double toprow[M];
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
for (top=col=0; top<N && col<M; top++, col++) {
  for (; col < M; col++) {
    if (matrix[col]!=0.0 && rank>=top)
      MPI_Allreduce(&rank, &row, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    else
      MPI_Allreduce(&nprocs, &row, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    if (row<nprocs) break;
  }
  if (col>=M) break;
  if (row!=top) {
    if (rank==top)
      MPI_Sendrecv_replace(matrix, M, MPI_DOUBLE, row, 0, row, 0, MPI_COMM_WORLD,
        &status);
    else if (rank==row)
      MPI_Sendrecv_replace(matrix, M, MPI_DOUBLE, top, 0, top, 0, MPI_COMM_WORLD,
        &status);
  }
  if (rank==top) {
    pivot = matrix[col];
    for (j=col; j<M; j++) {
      matrix[j] /= pivot;
      toprow[j] = matrix[j];
    }
  }
  MPI_Bcast(toprow, M, MPI_DOUBLE, top, MPI_COMM_WORLD);
  if (rank!=top) {
    tmp = matrix[col];
    for (j=col; j<M; j++) {
      matrix[j] -= toprow[j]*tmp;
    }
  }
}
```

Fig. 5. Parallel Gaussian elimination code

for other variables. Specifically, we introduce into M_{seq} an integer variable that gives the index in the symbolic table of the current symbolic value of *pc*. This expression is Boolean-valued and can involve operators such as $<,>,=,\neq,\geq,\leq,\wedge,\vee$. Its initial value is the special symbolic expression **true**. At each point where there is a floating-point branch in the program, say on a condition e, the model calls a function $\phi(pc, e)$. This function returns one of three possible values: if it can determine that $pc \Rightarrow e$ it returns **true**; if it can determine that $pc \Rightarrow \neg e$, it returns **false**; and if it cannot determine either, it returns **unknown**. If the answer is **true** or **false**, the corresponding branch is taken, but if the answer is **unknown** then the model makes a non-deterministic choice between the true and false branches. In

this latter case, if the true branch is selected, the value of pc is updated by setting it to $pc \wedge e$, while if the false branch is selected, it is set to $pc \wedge \neg e$. Note that this process ensures that the disjunction of the path conditions is always **true**.

Recall that in the composite model, the execution of M_{par} begins just after M_{seq} terminates. Now, the branches in M_{par} will be dealt with in the same way as in M_{seq}, but—and this is the crucial point—M_{par} will use the same path condition variable that was used in M_{seq}. Hence execution of M_{par} begins with pc holding the final value computed by the sequential model. This means that, *assuming* $\phi(pc, e)$ *can be evaluated with sufficient precision*, the parallel model can only follow a path that is consistent with the one followed by the sequential model. Since the disjunction of the path conditions from the sequential program is **true**, however, all paths in the parallel program are considered. The precision with which $\phi(pc, e)$ can be evaluated depends, of course, on the power of the reasoning system that is being used, as discussed further below. Finally, the last step in the composite model is the sequence of assertions comparing the output vectors of the two models, just as before.

Now, when SPIN is used to check for assertion violations in the composite model, it will have to explore all possible paths through M_{seq}, and for each of these it will have determined a path condition-output vector pair $(pc, \mathbf{y})$. For each such pair, it will explore all possible paths of the parallel model that are consistent with pc, determine the parallel output $\mathbf{y}'$, and check the equivalence of $\mathbf{y}$ and $\mathbf{y}'$. If the assertions can never be violated, we can conclude that for any input vector, the two programs must produce equivalent results, assuming the arithmetic used in executing the programs obeys the identities of the designated equivalence relation.

Notice that the path condition pc produced by a sequential run does not necessarily specify all the branch conditions for M_{par}. In the Gaussian elimination code, for example, the sequential program breaks out of the loop that searches for a pivot as soon as the first non-zero entry is found. Hence the symbolic variables for the entries that are not examined remained unconstrained in pc. In the parallel code, on the other hand, each process examines its own entry to see if it is non-zero, and those processes that cannot make this determination based on pc must make a non-deterministic choice. The model checker explores all of these choices, and checks that each of them results in the same output vector $\mathbf{y}$.

The effectiveness of this approach depends heavily on the precision with which the $\phi(pc, e)$ are evaluated. If **unknown** is returned for a case where it can in fact be shown that $pc \Rightarrow e$ (or that $pc \Rightarrow \neg e$), it is possible that SPIN will explore *infeasible* paths through M_{seq}, or paths through M_{par} that are not consistent with the one followed by M_{seq}. In these cases the analysis might produce a spurious result, i.e., it might report that a violation has been found when one does not really exist. However, since the analysis is *conservative*, i.e., it only ignores a branch when it is certain that the branch cannot be taken, a positive result guarantees that the two programs are equivalent. The procedure used by our implementation to determine ϕ is described in Section 4.1; it is very lightweight but precise enough to yield a conclusive result in all of the examples we have studied.

A useful byproduct of this method is the set of pairs $(pc, \mathbf{y})$ produced by SPIN in analyzing M_{seq}. These can be used to establish the correctness of the sequential

program, although exactly how this is done would depend on the particular program. For Gaussian elimination, for instance, each of the matrices that corresponds to a $\mathbf{y}$ in one of the pairs can be checked, by a series of assertions, to satisfy the conditions of reduced row echelon form.

In summary, our method to compare a sequential and parallel version of a numerical program consists of the following steps: (1) build a Spin model M_{seq} of the sequential program in which the floating-point computations (and perhaps some integer computations) are represented symbolically, and in which branches are modeled using non-deterministic choices and a path condition variable; (2) in a similar way, build a Spin model M_{par} of the parallel program; (3) put these together to form a composite model, in which first M_{seq} is executed, then M_{par} (using the same path condition variable), and which ends with assertions stating that the outputs of M_{seq} and M_{par} agree; (4) use Spin to check that the assertions of the composite model can never be violated.

3. REDUCTION THEOREMS

Partial orders and related techniques play an important role in model checking, by reducing the number of states that need to be explored. Ideally, we would have liked to apply techniques that are optimized for models of MPI programs, such as those discussed in Siegel [2005], but we could not find an easy way to implement them in Spin. Instead, in Section 3.1 we give a result that justifies slightly weaker techniques, but these techniques can be easily incorporated into Spin models. One consequence of the theorem is that for models with no wildcard receives, we may instruct Spin to use only synchronous communication, and we may place the code for each process in an `atomic` block. Though this greatly restricts the ways in which events from the different processes can be interleaved, the theorem implies that Spin will still explore every possible terminal state of the model, which is all that is required for our purposes. For models with wildcard receives, we must use asynchronous communication, but we can still use `atomic` blocks, as long as every wildcard receive occurs either outside, or as the first statement of, an `atomic` block. In both cases, the reduction in the number of states explored can be dramatic.

Many parallel numerical programs utilize a structure in which identical worker processes send requests and results to a server. These requests can arrive in any order, and a model checker must explore all of them, leading to a rapid blowup in the number of states. For instance, in the parallel version of the Monte Carlo example discussed in Section 4, the worker processes can send their requests to the random number server in any order. Hence in one execution worker 1 may get the first block of random numbers and worker 2 the second block, while in another execution the situation could be reversed. In fact, any permutation of the block distribution can take place on each iteration of the main loop. In Section 3.2, we give a symmetry reduction theorem that can be applied when the output vectors and path conditions of the sequential version of the program are invariant under these permutations.

3.1 A partial order reduction theorem for MPI programs

To explore all possible terminal *states* of a program, it is usually not necessary to explore all possible *paths* through the program. That is because there are often

many *equivalent* paths, i.e., paths terminating in the same terminal state. This means that we may restrict the set of paths that we explore, as long as we are sure to keep at least one representative from each equivalence class. The theorem below justifies such a restriction for MPI programs.

To state the theorem, we adopt the formal model of Siegel [2005], which slightly extends the model of Siegel and Avrunin [2005] by adding the notion of a *synchronous global transition*. The reader is referred to Siegel [2005] for the complete details, but the basic idea is that a model essentially consists of an automaton for each process and a set of channels, each with a fixed sending and receiving process. The transitions of the automata may be labeled by *local*, *send*, or *receive* events. Each state in an automaton is either a *terminal state* (a state with no outgoing transitions), a *local-event state* (all transitions departing from that state are local), a *sending state* (there is only one departing transition and it is labeled by a send event), a *receiving state* (all the departing transitions are labeled by receive events), or a *send-receive state*. Send-receive states are used to model the semantics of the `MPI_SENDRECV` statement, which essentially forks two threads, one to execute a send and the other to execute a receive. A global state of the model gives a state in the automaton for each process and a (possibly empty) queue of messages for each channel. A global transition corresponds to either a single transition in one of the automata or a pair of send and receive transitions that execute synchronously; the latter is a *synchronous global transition*.

So let $\mathcal{M}$ be any model of an MPI program, and ν a *channel size vector*. Thus ν assigns to each channel c a value $\nu(c)$ which is either ∞ or a non-negative integer. Suppose that T is a finite trace of $\mathcal{M}$ from a global state σ_0 to a global state σ_f and that T is *ν-bounded*, i.e.,

$$\forall c \in \mathsf{Chan}\colon\ \mathsf{maxlen}_c(T) \leq \nu(c),$$

where $\mathsf{maxlen}_c(T)$ is the maximum length at any point in the execution of the trace T of the queue of messages that have been sent on channel c but not yet received. This means that the number of messages queued in any channel c never exceeds the bound $\nu(c)$. Suppose further that σ_f is *ν-halted*, i.e., there is no global transition departing from σ_f that does not cause the number of queued messages in some c to exceed $\nu(c)$. Another way of saying this is that there is no way to extend T to a ν-bounded trace of length $n+1$, where $n = |T|$.

We consider a game that involves two players, the *scheduler* and the *selector*, and constructs a new ν-bounded trace T'. The game consists of a sequence of stages, each of which extends the trace by one transition. So, at the beginning of stage i, the first $i-1$ transitions of T' have already been selected. Stage i proceeds as follows. First, if T' terminates in a ν-halted state, then the game *ends in deadlock*. Otherwise, the scheduler chooses a non-empty subset of enabled transitions, according to a certain rule that we describe below. The selector then chooses a specific transition from this subset and appends it to T'. The game ends if it deadlocks, or after n stages have completed, whichever occurs first. The selector wins the game if the final trace constructed terminates in σ_f, else the scheduler wins. The theorem states that there is a strategy for the selector so that the selector always wins.

We now describe the *scheduling rule* that constrains the choice made by the

scheduler. Say the current state is σ. Let E_σ denote the set of all ν-*enabled* global transitions at σ, i.e., the set of all global transitions enabled at σ that do not cause the number of queued messages for any channel c to exceed its bound $\nu(c)$. Suppose E_σ is non-empty. Then the scheduler must choose a non-empty subset F of E_σ that satisfies at least one of the following conditions: (1) $F = \{\tau\}$, where τ is not a local event transition, nor a receive transition emanating from a wildcard receive state (i.e., a state from which there are outgoing transitions labeled by receives on at least two distinct channels), nor a synchronous transition for which the associated receive transition emanates from a wildcard receive state, (2) F is the set of all transitions enabled in a single process that is at a local event state, or (3) $F = E_\sigma$. We call such a set F an *acceptable set at* σ, and the set of all acceptable sets at σ will be denoted C_σ.

To state the theorem precisely, we introduce the following notation. First, for any finite sequence $S = (x_1, \ldots, x_n)$ of elements of a set X, and any $x \in X$, we let $S \cdot x$ denote the sequence obtained by appending x to the end of S. Next, given any two global states σ and σ_f of $\mathcal{M}$, we define statements $\theta(\sigma, \sigma_f, n)$, for all $n \geq 0$, as follows: $\theta(\sigma, \sigma_f, 0)$ is the statement $\sigma = \sigma_f$, and for $n > 0$, $\theta(\sigma, \sigma_f, n)$ is the statement

$$E_\sigma \neq \emptyset \wedge \forall F \in C_\sigma \; \exists \tau \in F \colon \theta(\mathsf{des}(\tau), \sigma_f, n-1),$$

where τ is a transition from the state σ to the state $\mathsf{des}(\tau)$. This is a formal way of stating that, starting from σ, the selector can always force the trace to terminate at σ_f in n steps, no matter what moves are made by the scheduler.

THEOREM 1. *Let $\mathcal{M}$ be a model of an MPI program, ν a channel size vector, T a ν-bounded trace in $\mathcal{M}$ from a global state σ to a ν-halted global state σ_f. Then $\theta(\sigma, \sigma_f, |T|)$ holds.*

PROOF. The proof is by induction on $|T|$. If $|T| = 0$ then $\sigma = \sigma_f$, and since this is exactly the statement $\theta(\sigma, \sigma_f, 0)$, the theorem holds. So suppose $n > 0$, $T = (\tau_1, \ldots, \tau_n)$, and the theorem holds for any trace of length less than n. We cannot have $E_\sigma = \emptyset$ since τ_1 is ν-enabled at σ. Let $F \in C_\sigma$.

If $F = E_\sigma$ then let $\tau = \tau_1$. Then $(\tau_2, \ldots, \tau_n)$ is a ν-bounded trace from $\mathsf{des}(\tau)$ to σ_f of length $n-1$, and so by the inductive hypothesis, $\theta(\mathsf{des}(\tau), \sigma_f, n-1)$ holds. Hence $\theta(\sigma, \sigma_f, |T|)$ holds.

Suppose F is the set of all ν-enabled local event transitions from a single process p. Now there must exist an integer i such that $1 \leq i \leq n$ and τ_i is in process p. For if not, there would still be a local event transition in p enabled at σ_f, and σ_f would not be a ν-halted state. Let i be the least integer with this property. We claim that there is a ν-bounded trace

$$T' = (\tau_i', \tau_1', \ldots, \tau_{i-1}', \tau_{i+1}', \ldots, \tau_n')$$

from σ to σ_f with $\mathsf{label}(\tau_j') = \mathsf{label}(\tau_j)$ for all j. This is because we may move τ_i to the left one step at a time, applying Lemma 1 of Siegel [2004] at each step. Let $\tau = \tau_i'$, and argue as in the paragraph above that $\theta(\sigma, \sigma_f, |T|)$ holds.

Suppose F is a singleton set $\{\tau\}$, where τ is either a send, receive, or synchronous transition. Say τ is a receive in process p. Then, according to the scheduling rule, at σ, p must be in a receiving state for a sole channel c, and so $\mathsf{label}(\tau) = c?x$ for

some x that is already queued at σ. Now there must exist some i such that (1) $1 \leq i \leq n$, (2) τ_i is a receive in process p, and (3) there is at most one j such that $1 \leq j < i$ and τ_j belongs to process p, and, if there is such a j, then σ is a send-receive state and τ_j is the send emanating from that state. The reason for this is that if it were not the case, there would still be a receive enabled at σ_f, and σ_f would not be ν-halted. Now we argue as before to move τ_i to the left, using Lemma 2 of Siegel [2004] to move past the send τ_j if necessary. The only thing we must check is that the message received by τ_i was already queued at σ. However, since c is the sole receiving channel for p at σ, τ_i must also be a receive on c, and so we must have $\mathsf{label}(\tau_i) = c?x$, as required. Now we proceed to argue as in the paragraph above that $\theta(\sigma, \sigma_f, |T|)$ holds. The case where τ is a send is similar but easier since we do not have to deal with wildcards. If τ is a synchronous transition, then first decompose it into its send and receive parts, then move each all the way to the left, and then recompose them into a synchronous transition. □

We now examine some practical consequences of Theorem 1. Say we are using Spin to verify an assertion on the terminal states of $\mathcal{M}$. In creating a Spin model, we must specify a finite bound $\nu(c)$, for each channel c, when c is declared. By the *scheduling policy* we mean the mechanism of Spin that determines the set of all possible next transitions from a given state. The scheduling policy plays the role of the scheduler in our game. In its default mode, the scheduling policy returns all ν-enabled transitions. Hence in the default mode, Spin will explore all ν-halted states that are reachable by ν-bounded traces from the initial state.

Notice that, if there are terminal states that can only be reached by traces in which the number of queued messages for some c exceeds $\nu(c)$, these states will not be explored by Spin. In some cases, one may verify that there are no such states by using assertions to check that control never reaches a send statement for a channel when that channel is full. In other cases, this may not be possible, or the channel sizes required may be so large that the verification becomes infeasible. In such cases we may still use a less-than-satisfactory channel size and satisfy ourselves with a result that is not quite conservative. (Of course, if Spin finds an error, this is just as helpful as if we had used unbounded channels.) The situation is similar to the need that sometimes arises to place small bounds on the number of loop iterations, and is a problem that often arises with finite-state verification techniques. What we will see shortly, however, is that the reduction strategy we describe cannot make the problem any worse, i.e., if there exists a property violation within the specified bounds, it will still be found after applying the reduction.

Now, the scheduling policy used by Spin can be restricted with the careful use of `atomic` blocks. When control is inside an `atomic` block of a process, Spin's scheduling policy returns only the ν-enabled transitions from that process, assuming there is at least one. If there are none, then the process loses atomicity, and the scheduling policy returns the set of all ν-enabled transitions. A special rule is used for *rendezvous channels*, i.e., channels of size 0. A send on a rendezvous channel is not blocked precisely when the receiving process is in a position to receive the message; in this case, the Spin scheduling policy returns just the synchronous transition and control passes to the receiving process. If the receiving process happens to also be inside an `atomic` block, then atomic control passes directly

from the sender to the receiver.

Now we can use `atomic` statements in any way, as long as the resulting scheduling policy obeys the scheduling rule defined above. Let us say, for example, that we have inserted `atomic` statements in such a way that if a wildcard receive occurs within an `atomic` block B then it must be the first statement in B and B cannot be inside another `atomic` block. Assume also that $\nu(c) \geq 1$ for all c, so there are no rendezvous channels. These assumptions mean that the only state from which Spin's scheduling policy can select a wildcard receive is one in which no process has atomic control. But if no process has atomic control, Spin's scheduling policy will return the set of all ν-enabled transitions. Hence the scheduling policy satisfies the scheduling rule, and we are guaranteed that Spin will still explore all ν-halted states reachable from the initial state by ν-bounded traces.

Consider now the case where $\mathcal{M}$ has no wildcard receives, and let ν be any channel size vector. Say that we construct a Spin model of $\mathcal{M}$ in which we set all channel sizes to 0 and place the code of each process in a single `atomic` block. What is the resulting scheduling policy? In any state, it will return either (1) a singleton set consisting of a synchronous transition (that by assumption does not involve a wildcard), or (2) the set of all local transitions in a single process, or (3) the empty set, if the state is potentially halted (i.e., no synchronous or local event transition is enabled). If (3) occurs when the state is not terminal, Spin will report this as an improper end state (i.e., a deadlock). Hence if the search returns without ever reporting an improper end state, then the scheduling policy satisfies the scheduling rule, and we are guaranteed that the search has visited every ν-halted state of $\mathcal{M}$ reachable by a ν-bounded trace.

3.2 A symmetry reduction theorem for MPI programs

In this section we prove a theorem that has consequences for numerical programs in which the output vectors and path conditions exhibit symmetry in the symbolic constants. The theorem is expressed using the language of group theory and group actions [Rotman 1995].

Let n and m be non-negative integers, G a finite group, and X a G-set of cardinality n. Let

$$\mathcal{X} = \{(x_1, \ldots, x_n) \mid \{x_1, \ldots, x_n\} = X\},$$

which is a set of cardinality $n!$. Let Π and Y be G-sets, and let $\mathcal{Y} = Y^m$. The action of G on X induces an action of G on $\mathcal{X}$ by defining

$$g(x_1, \ldots, x_n) = (gx_1, \ldots, gx_n),$$

for $g \in G$. The action of G on Y induces a component-wise action on $\mathcal{Y}$ as well. We call the 6-tuple

$$\mathcal{C} = (G, X, \mathcal{X}, \Pi, Y, \mathcal{Y})$$

a *symbolic context*.

Let $\mathcal{C}$ be a symbolic context. Consider a pair of functions (C, f), where C assigns, to each $\mathbf{x} \in \mathcal{X}$ and $p \in \Pi$, a set $C(\mathbf{x}, p)$, and f assigns to each triple $(\mathbf{x}, p, c)$, where $\mathbf{x} \in \mathcal{X}$, $p \in \Pi$, and $c \in C(\mathbf{x}, p)$, an element $\mathbf{y} = f(\mathbf{x}, p, c) \in \mathcal{Y}$. Assume that for all

$g \in G$, $\mathbf{x} \in \mathcal{X}$, $p \in \Pi$, and $c \in C(\mathbf{x}, p)$, the following both hold:

$$C(g\mathbf{x}, gp) = C(\mathbf{x}, p) \tag{1}$$
$$gf(\mathbf{x}, p, c) = f(gx, gp, c). \tag{2}$$

Then we call (C, f) a *symbolic model* over $\mathcal{C}$.

THEOREM 2. *Let $\mathcal{C} = (G, X, \mathcal{X}, \Pi, Y, \mathcal{Y})$ be a symbolic context and let (C, f) be a symbolic model over $\mathcal{C}$. Suppose there are $\mathbf{x} \in \mathcal{X}$, $p \in \Pi$, and $\mathbf{y} \in \mathcal{Y}$ for which the following all hold:*

(1) $gp = p$ *for all* $g \in G$,
(2) $g\mathbf{y} = \mathbf{y}$ *for all* $g \in G$, *and*
(3) $\forall c \in C(\mathbf{x}, p)\ \exists g \in G\colon f(g\mathbf{x}, p, c) = \mathbf{y}$.

Then $f(\mathbf{x}, p, c) = \mathbf{y}$ *for all* $c \in C(\mathbf{x}, p)$.

PROOF. Given c, choose g to satisfy hypothesis 3. Then

$$\begin{aligned} f(\mathbf{x}, p, c) &= g^{-1} f(g\mathbf{x}, gp, c) \\ &= g^{-1} f(g\mathbf{x}, p, c) \\ &= g^{-1} y \\ &= \mathbf{y}. \quad \square \end{aligned}$$

Now we describe the application of this theorem to symbolic models of numerical programs. In this application, the set X is the set of symbolic constants, and $\mathcal{X}$ is the set of all possible input vectors to the model. The group G may be any subgroup of Σ_X, the group of all permutations of X. The set Π is the set of all Boolean-valued symbolic expressions in the symbolic constants, e.g., $(x_1 x_2)/x_3 \geq 0 \wedge x_2 \neq x_3$. A path condition, for example, is an element of Π. Notice that the action of G on X extends naturally to an action on Π in which G acts trivially on operators and literals. We may also take Π to be the set of all Boolean-valued symbolic expressions modulo an operation-preserving equivalence relation $\sim$, as long as $\sim$ is preserved by the action of G, i.e., $p \sim q \Rightarrow gp \sim gq$ for all $g \in G$. The examples of equivalence relations that we have considered in this paper all satisfy this property. The set Y is the set of all real-valued symbolic expressions in the symbolic constants, e.g., $(x_1 x_2)/x_3 + x_4$. Again, we may apply an appropriate equivalence relation. The set $\mathcal{Y}$ is the set of all symbolic output vectors.

The pair (C, f) represents our numerical program P. The set $C(\mathbf{x}, p)$ corresponds to the set of all executions of P on input $\mathbf{x}$ that are consistent with the path condition p. The element $\mathbf{y} = f(\mathbf{x}, p, c)$ represents the output of P when given input $\mathbf{x}$, a path condition p, and a particular execution c consistent with p. The assumptions (1) and (2) express what is essentially a functorial property in the symbolic constants: permuting the names of the input does not change the set of behaviors of the program, nor does it change the output produced by the program, except to permute the names of the symbolic constants in the output in the same way that they were permuted in the input.

The theorem may now be interpreted as follows: suppose we are given an input vector and a path condition p and output vector $\mathbf{y}$ such that both p and $\mathbf{y}$ are

invariant under the action of G. Then for each possible path through the model that is consistent with p, we may first permute the input, using the permutation induced by the action of any element of G, before computing the output produced by that path. If the output is always $\mathbf{y}$ then we may conclude the output would have been $\mathbf{y}$ even if we had not permuted the input.

4. EXPERIMENTS

In this section, we discuss our implementation of the method and our experience in applying it to four numerical programs. The source code for the implementation, the models, and all of the experimental results can be obtained at `http://www.cis.udel.edu/~siegel/projects`.

4.1 Implementation

The core of our implementation is a library of functions for manipulating symbolic expressions and maintaining the symbolic expression table. This library is written in C and is incorporated into our models by using the embedded C code facility of Spin. The entries in the table are C `struct`s and include fields for the index of the expression, an integer code representing the operator, pointers to the left and right subexpressions for binary expressions, etc. A hashtable is also used to find table entries quickly.

Three "levels" of each arithmetic operation are provided, corresponding to the three different equivalence relations discussed in Section 2.2. The level 0 operations correspond to Herbrand equivalence; these simply form a new expression from the operator and operands, check to see if the new expression already exists in the table, add it to the table if it does not, and return the index. The level 1 operations correspond to IEEE equivalence and do a little more work; the level 1 addition operation, for example, checks to see whether one of the operands is 0 (in which case it returns the other operand), or if one operand is the negative of the other (in which case it returns 0). The level 2 operations correspond to real equivalence. The level 2 addition operation, for example, exploits associativity and commutativity to reduce sums to a simplified form in which the parentheses are moved to the left as far as possible, the terms are ordered by increasing index, literal integer terms are combined into a single literal, and so on. Of course, when checking for IEEE equivalence, the level 2 addition and multiplication operations can also be used for all integer expressions.

Similar things could of course be done for the other level 2 operations (multiplication, subtraction, and so on), but in the examples we have studied so far this has not been necessary, and so at present these work exactly as the corresponding level 1 operations. In our experience, it seems that additive reduction operations, which are very common in parallel numerical programs, account for much of the difference between the exact symbolic expressions computed in the sequential and parallel models. Reductions over other operations, such as multiplication, seem to be much less common. In any case, the symbolic package is designed to make it easy to specify the symbolic operation used for a particular computation in the code and to add new versions of the symbolic operations to reduce expressions to other simplified forms, as the need arises.

The function ϕ, which attempts to determine whether the given path condition

Table I. Experimental data

	matmat	gauss	jacobi	monte
equivalence type	Herbrand	Herbrand	real	IEEE
parallel processes	6	6	17	9
sequential executions	1	13,327	4	4
symbolic expressions	2,202	247,656	8,239	1,232
size of input vector	200	26	1,333	99
size of output vector	100	36	36	1
size of path conditions	0	36	3	3
states ($\times 10^3$)	4,443	16,114	6,295	3,112
memory (MB)	217	801	362	279
time (seconds)	506	3,224	9,846	738

pc implies the given expression e (or $\neg e$), is implemented as follows. First, by construction, pc will always be a conjunction of smaller expressions of the form pc_i or $\neg pc_i$, where each pc_i arises by evaluating one of the conditional expressions in a branching statement. Our implementation of ϕ simply loops over i, looking for a pc_i which can be easily seen to imply e or $\neg e$. By "easily seen" we mean by using reasoning such as $x < y \Rightarrow x \neq y$, $x = y \Rightarrow \neg(x \neq y)$, and so on. If it finds such a pc_i, it returns **true** or **false**, as the case may be, otherwise it returns **unknown**. This lightweight procedure seems to be effective because the conditional expressions evaluated in the sequential and parallel programs tend to be quite similar.

All of the variables from the symbolic package are static, that is, they are not incorporated into the state vector by using, for example, the SPIN `c_track` function. Thus the only variables incorporated into the state vector are those corresponding to variables in the original programs and the path condition variable.

The type of equivalence that one wishes to verify (Herbrand, IEEE, or real) is controlled by a command-line argument specified when compiling the verifier (pan.c) generated by SPIN. This argument tells the symbolic package which level of symbolic operations it should use: level 0 for Herbrand equivalence, level 1 for IEEE equivalence, and level 2 for real equivalence. (When verifying IEEE equivalence, integer addition and multiplication operations use level 2, instead of level 1.) Finer control over the operations can be obtained by defining additional functions and calling them from the Promela model where desired.

4.2 The programs

For our preliminary study, we analyzed four scalable parallel numerical programs. We attempted to verify each of these using the method of this paper, scaling until SPIN exhausted the 800 MB of available memory or verification time exceeded 10,000 seconds. In each case, we began by trying to verify Herbrand equivalence. If an actual counterexample was found, we then tried to verify IEEE equivalence, and if an actual counterexample was found again we then tried Real equivalence. This approach guaranteed that we would verify the strongest type of functional equivalence that held for the program.

In what follows, we give a brief description of each program, we discuss certain issues that arose in verifying its correctness, and we explain what we were able to verify (or not verify).

In Table I, we give data for the largest configuration of each program that we were able to verify. The rows of the table give (1) the type of equivalence that was verified, (2) the number n of parallel processes, (3) the number of distinct sequential executions, (4) the number of expressions generated in the course of the verification, (5) the length of the input vector, i.e., the number of symbolic constants, (6) the length of the output vector, (7) the maximum number of terms in the path condition conjunction, (8) the number of states explored, (9) the amount of memory used by SPIN, and (10) the verification time. We used SPIN version 4.2.4 with options `-DCOLLAPSE -DSAFETY -DNOBOUNDCHECK` on a 2.2GHz Pentium 4 Linux box.

4.2.1 *matmat.* Our first example is the matrix multiplication program of Section 2.1, with $N = L = M = 2(n-1)$. As all the loops in the program code are already finite, it was not necessary to impose any bounds on them when constructing the models. We were able to verify that the sequential and parallel programs are Herbrand equivalent for $n \leq 6$.

4.2.2 *gauss.* Our second example is the Gaussian elimination program of Section 2.3, with $N = M = n$. We wrote both the sequential and parallel codes ourselves. Again, it was not necessary to impose any loop bounds when constructing the models. We were able to verify that the sequential and parallel programs are Herbrand equivalent for $n \leq 6$. We note, however, that in order to show that the sequential program really produces the reduced row echelon form, we needed IEEE arithmetic. This is because, for example, the use of Herbrand arithmetic results in matrix entries of the form x_0/x_0 where the definition of reduced row echelon form requires 1.0.

4.2.3 *jacobi.* Our third example implements a Jacobi iteration algorithm to solve a linear system of the form $A\mathbf{x} = \mathbf{b}$. Both the sequential and parallel versions are from the CD ROM accompanying the book by Karniadakis and Kirby [Karniadakis and Kirby II 2003]. (Both versions are in the file SCchapter7.cpp; the sequential version is the function `Jacobi` and the parallel version is the function `Jacobi_P`.) In this algorithm, the $N \times N$ matrix A and the column vector $\mathbf{b}$ of length N form the input, and the goal is to solve for the value of the column vector $\mathbf{x}$ of length N, which forms the output. We take $N = 2n+2$. The algorithm begins with an initial guess for $\mathbf{x}$ (the column vector in which every entry is 1.0), and then enters a loop in which the entries of $\mathbf{x}$ are adjusted at each iteration, based on the values of neighboring entries. The algorithm stops when the error term, which is computed as the inner product of the difference between two consecutive values of $\mathbf{x}$ with itself, falls below a given threshold ϵ, or when the number of iterations exceeds a fixed bound `MAXITS`.

In the parallel version, the data are partitioned so that each process contains a certain number of rows of A, $\mathbf{x}$, and $\mathbf{b}$. Communication is used to update the contents of *ghost-cells*, which mirror the boundary data on neighboring processes, and in a reduction operation used to compute the error term after each iteration. In our models, ϵ is treated as another symbolic constant, and we take `MAXITS` $= 3$. (Some constant bound must be specified for `MAXITS` if the model is to have a finite number of states.)

Our analysis quickly revealed that the results of the sequential and parallel pro-

grams could disagree for $n = 2$, even using real arithmetic. The source of the problem was a small mistake in the computation of the error in the sequential code: instead of taking the inner product of the difference between two successive values of `x` with itself, the code simply took the inner product of the two successive values. (We reported the error to the authors and it has been corrected in the second printing of the book.) After correcting the error, we verified real equivalence (which is the best one can expect, due to the floating-point reduction operation) for $n \leq 17$. While this example scaled significantly further than the others, it is also the only case in which time, rather than memory, proved to be the limiting factor. This appears to be due to the large amount of computation required to simplify expressions when using the level 2 operations.

4.2.4 *monte.* Our fourth example is a parallel program taken from Gropp et al. [1999] that implements a Monte Carlo algorithm to estimate π. (We wrote the sequential code.) The algorithm repeatedly chooses a point at random from a square with sides of length 2. If the distance from the point to the center exceeds 1.0, an integer variable `out`, initially 0, is incremented, else a variable `in` is incremented. The estimate for π is `4.0*in/(in+out)`. The algorithm stops when an error term falls below a fixed threshold ϵ, or `in+out` exceeds a fixed bound. In the parallel code, one process acts as a random number server, returning blocks of random numbers to the remaining "worker" processes. The worker processes use these blocks to determine a set of points and make their own local `in` and `out` calculations. The values of `in` and `out` are summed at the end of each iteration, using an integer reduction operation. At the end of the reduction, each process has the global sums, forms the estimate for π, computes the error, and decides whether to perform another iteration or terminate. In our models of these programs, we bounded the loops so that the number of points consumed by each worker could never exceed 4.

The random nature of this code presents an interesting challenge to our method. On the face of it, a program that depends in an essential way on the values returned by a random function can hardly be deterministic. We resolve this problem by considering the sequence of random numbers generated by the random function to be the *inputs* to the program. Hence our method can be used to verify that if the random number function were to generate the same sequence of values for the sequential and the parallel programs, the two programs must return the same estimate for π. This seems to us to be a natural extension of our notion of equivalence to numerical programs that use random numbers.

We used two additional reduction techniques for this example, which proved very effective. The first concerns the program statement

```
if (x*x+y*y<1.0) then in++ else out++;
```

which is used to determine whether a point $(\mathtt{x}, \mathtt{y})$ is within distance 1.0 of the center. If we were to follow our method strictly, each time this statement is executed in M_{seq} a non-deterministic choice would be made between the two alternatives. Since this statement is executed many times in the model, the number of sequential executions would blow up quickly. To avoid this problem, we made a simple program transformation. First, we defined a new operation `delta` which takes two floating-point arguments a and b, and returns the integer 1 if $a < b$ and 0 otherwise. The

statement above can then be replaced by

```
in += delta(x*x+y*y,1.0);
out += 1-delta(x*x+y*y,1.0);
```

which does not require a non-deterministic choice. The only change we had to make to the symbolic package was to add a level 0 operation for `delta`, i.e., we just treat `delta` as an uninterpreted function. With this modification, the symbolic output of M_{seq} will be a more complicated expression, involving many `delta`-subexpressions, but the number of executions of M_{seq} will be much smaller, which turns out to be a good tradeoff. Notice also what happens if we use IEEE arithmetic to compute the sum of `in` and `out`; since the symbolic package knows to use associativity and commutativity for integer expressions, the `delta` terms in the sum all cancel and the result is a single integer constant. This also reduces the number of states explored, since it allows the symbolic package to determine with precision when the sum exceeds the upper bound, rather than forcing it to make another non-deterministic choice.

Using these this transformation and the symmetry reduction theorem of Section 3, we were able to establish IEEE equivalence for $n \leq 9$. These reductions appear to be fairly general and should be useful with a wide variety of parallel numerical programs.

5. THE MODEL CONSTRUCTION PROBLEM AND MPI-SPIN

The Promela models for the programs described in Section 4.2 were all constructed by hand. As we have noted, the parallel versions of these programs use MPI, and so some way had to be found to represent this MPI communication using Promela constructs. Promela does not specifically support MPI, but it does provide some general abstractions that correspond to basic message-passing operations. In particular, there is a *channel* data structure with enqueue and dequeue operations that can be used to model the basic MPI point-to-point operations occurring in our (relatively simple) MPI programs. Collective operations such as `MPI_Allreduce` can be modeled using a "coordinator" process and additional channels. This is the modeling approach taken in previous work and used to construct the models for the experiments described in this paper. But while this approach may suffice for a "proof of concept" of the symbolic comparison method that is the main subject of this paper, it does have certain practical limitations. We discuss these limitations in detail in Section 5.1. Some of the limitations have been addressed in a new extension to Spin called Mpi-Spin [Siegel 2007a; 2007b], and in Section 5.2 we discuss some preliminary work using Mpi-Spin to implement the symbolic comparison method.

5.1 Limitations to the modeling approach

The programs we considered used only a small number of the MPI primitives—primarily the basic *blocking* point-to-point functions `MPI_Send` and `MPI_Recv`. The arguments for `MPI_Send` include (1) a pointer to the beginning of the buffer containing the data to be sent, (2) the number of data elements to send, (3) the type of the data elements, (4) the rank of the destination process, and (5) an integer tag. The arguments for `MPI_Recv` are similar.

In the models, much of the information contained in these parameters is abstracted away. The abstractions vary from model to model and even within a single model. For example, in the matrix multiplication example, where tags play a crucial role in the logic of the program, each message sent by the Master process in the model consists of two components: (1) a vector of length L consisting of integer indices of symbolic expressions and (2) a single byte representing the tag. The messages sent by the slaves also have a vector and a byte representing the tag, but the length of the vector is M. In the Gaussian elimination case, where the tags are not used in any significant way and so can be abstracted away altogether, a message used in point-to-point communication is just a vector of integer indices of symbolic expressions of length M, while a message sent as part of the `MPI_Allreduce` consists of a single byte.

The type of message that can be sent on a Spin channel, however, is fixed statically in the channel declaration and cannot be changed. One implication of this is that the channel declarations (and related code) must be tailor-made for each model. Similarly, the Promela code implementing collective operations may have to be re-designed for each case, depending on such things as the type of abstraction used to represent the data, the particular reduction operation, and so on. All of this limits the possibilities for code re-use from model to model, increasing the burden on the modeler and the likelihood of errors.

A second problem arises if one attempts to construct a model where a process sends one type of message to another process at one point and another type at another point. If the same Spin channel is to be used at both points, some sort of common abstraction must be found for the two message types, which may be difficult to achieve. If, on the other hand, distinct channels are used for the two message types, the relative order between messages in different channels will not be preserved, which may violate the MPI ordering semantics and result in a model that is not conservative.

A third problem with the channel-based approach is that it does not generalize to the more complex MPI operations. Of particular importance are the widely-used MPI *nonblocking* operations. These provide a precise way to specify how computation and communication can be carried out concurrently in an MPI program, a technique that is often given a large degree of credit for the high level of performance that scientific computing has achieved.

Nonblocking constructs can be used to specify that a communication task is to begin at one point in an MPI process and that the process should block at a subsequent point until that task has completed; computational code can be inserted between these two points to achieve the desired overlap. For example, one indicates that a send operation is to begin by *posting a send request* via a call to the nonblocking function `MPI_Isend`. (The "`I`" stands for "immediate" and indicates that this function returns immediately, rather than blocking.) The arguments for `MPI_Isend` are similar to those for `MPI_Send`, but in addition the nonblocking version returns a *handle* to a *request object*. A subsequent invocation of `MPI_Wait` on this request handle blocks until the send request has completed, i.e., until the data has been completely copied out of the send buffer and into either a system buffer or directly into the receive buffer of the receiving process. A nonblocking

receive operation works in an analogous manner, the corresponding request blocking until the data has been completely copied into the receive buffer. Note that nonblocking communication is strictly more general than blocking communication; `MPI_Send`, for example, is functionally equivalent to `MPI_Isend` followed immediately by `MPI_Wait`.

It is not at all clear how the simple channel-based abstraction can be modified to deal with nonblocking communication. Consider, for example, the case where a receive request is posted before any send request is posted. The posting of the receive request cannot be represented by pulling a message out of a queue, since the message does not yet exist. Yet somehow the request must be represented in the state, along with a reference to the variable(s) modeling the receive buffer. For if at some subsequent point a matching send request posts, the system must somehow pair the two requests and copy the appropriate data from the send variable(s) into the receive variable(s).

5.2 Mpi-Spin

Mpi-Spin is an extension to Spin for modeling MPI programs in a way that addresses the limitations discussed above. It adds to Spin's input language many functions, types, and constants corresponding to the MPI primitives, including all of the (standard mode) nonblocking MPI functions. The syntax of these functions is almost exactly the same as for the C bindings of the MPI functions. In particular, these functions accept arbitrary C pointers; they support various datatypes; they allow messages of any type to be sent at any time; and they support the most common reduction operations. No special effort is required on the part of the user to model the MPI infrastructure or primitives. All of this greatly reduces the effort required to construct models of MPI programs.

The basic idea behind the implementation of Mpi-Spin is to use a *communication record* structure to represent each outstanding communication request or buffered message in the system state. The fields for this C structure include a pointer to the send or receive buffer, the datatype, the ranks of the source and destination processes, and so on. The communication record values created in the course of the search of the state space are hashed and assigned unique ID numbers, just as in the case of the symbolic expressions, and it is these ID numbers that are assigned to variables in the Promela models. Instead of using multiple channels, a single global array of active communication records is maintained and modified by the various MPI functions. An additional MPI daemon process is used to model all possible behaviors of the MPI infrastructure and is responsible for such actions as pairing send and receive requests, buffering messages, and completing requests. All of these implementation details, however, are transparent to the user.

Mpi-Spin also introduces a type `MPI_Symbolic` for representing symbolic expressions, togther with a number of operations on that type. Hence Mpi-Spin provides all of the ingredients necessary for applying the symbolic comparison method to more complicated MPI programs, including those that use nonblocking communication.

To examine this potential, we considered another example from a popular MPI text [Snir et al. 1998]. This program is another variation on the Jacobi iteration algorithm, but takes as input an $(N + 2) \times (N + 2)$ matrix A ($N \geq 1$), which

is successively modified and also constitutes the output of the program. (The sequential version is Example 2.12; the parallel version we consider here is Example 2.27.) The values of the leftmost and rightmost columns and the top and bottom rows of A correspond to boundary conditions and are fixed, while the value of a cell in the interior $N \times N$ sub-matrix is updated on each iteration using a formula that is a function of the cell's left, right, upper, and lower neighbors. An additional $N \times N$ matrix B is used to temporarily hold the new values as they are computed.

In the parallel version, A and B are distributed by columns, and each process uses one or two additional columns for ghost cells. The body of the main loop of the parallel program consists of four steps. In the first step, the new values for the left and right columns of the local section of B are computed. In the second step, four communication requests are posted: two for sending those two columns of B to the left and right neighbors, and two for receiving the columns from the neighbors into the local ghost cells. In the third step, the new values for the interior columns of B are computed and then all the values of B are copied into A. The fourth step is a call to `MPI_Waitall` on an array consisting of the four request handles: this causes the calling process to block until all four requests have completed. Their completion guarantees that the ghost cells have been properly updated and that the data has been entirely copied out of the left and right columns, so that it is safe to proceed to the next loop iteration. Note that overlap between computation and communication is achieved by allowing the computation of the new interior values to take place concurrently with the ghost cell exchange.

Using Mpi-Spin, we quickly discovered a fault in the parallel version. The problem occurs whenever $N < 2n$. In that case, on at least one process, two send requests are posted using the same buffer—the single column of the local section of B—which is not allowed in MPI. In all other cases considered, Mpi-Spin was able to successfully verify the equivalence of the sequential and parallel versions. In the largest configuration which we verified successfully, $n = 6$, $N = 14$, and synchronous nonblocking communication was used; this resulted in 512,553 states explored, consumed 332 MB, and took 307 seconds.

As we have noted, MPI blocking communication is a special case of nonblocking communication and, in fact, blocking functions such as `MPI_Send` or `MPI_Recv` are implemented in Mpi-Spin by simply calling the corresponding nonblocking function and following this immediately with an `MPI_Wait`. It appears, however, that for programs that use exclusively blocking communication, the Mpi-Spin approach is not as efficient as the channel-based approach. For example, the verification of the Gaussian elimination example for $n = 5$ required 6.67 millions states and 1015 MB for Mpi-Spin; for our tailor-made channel-based model the numbers are 790,611 states and 35 MB. We could not scale this example to $n = 6$ using Mpi-Spin.

To address this shortcoming, we added a number of optimizations to Mpi-Spin for models that use exclusively blocking communication. The principal optimization is the use of channels instead of the communication record array. The messages sent on the channels are integer IDs of communication record values. In this way, the optimization maintains the full generality of the Mpi-Spin approach. To use this optimization, the user need only indicate with a flag that the model uses exclusively blocking communication; no changes to the model itself are required.

With the new optimizations in place, the Mpi-Spin models appear to scale comparably to our original tailor-made models. The Gaussian elimination example actually consumes slightly less states/memory using Mpi-Spin. For the matrix multiplication example, the Mpi-Spin model consumes approximately 20% more states/memory on the largest configuration. We plan to port the other two examples and to explore further optimizations in order to improve the performance of Mpi-Spin without sacrificing its usability and generality.

6. RELATED WORK

The idea of representing computations symbolically has a long history and has enjoyed many applications, including to testing and debugging (e.g., [Boyer et al. 1975; Clarke 1976; Hantler and King 1976]). Almost from the beginning, researchers have attempted to use symbolic execution techniques in the verification of parallel programs (e.g., [Brand and W. H. Joyner 1978]) but, as pointed out by Dillon [Dillon 1990], there are a number of significant obstacles that must be overcome. We will not attempt to review the large literature on symbolic execution or verification of parallel programs here, but rather briefly mention some of the work combining symbolic execution and model checking and explain how our approach differs from previous approaches.

Symbolic execution is a key part of the SLAM project. A component of the SLAM toolkit [Ball and Rajamani 2001] translates a C program into a program that operates solely on Boolean variables corresponding to predicates in the original program. A theorem prover is used in that process to determine the effect of each statement in the original program on the predicates. Another component uses symbolic execution to determine whether a path through the Boolean program corresponds to an actual execution of the original program. This is similar in spirit to our method, which translates a program into one which operates on symbolic expressions and uses a (very lightweight) form of theorem proving to determine branches and expression equivalence, but is intended to drive a counterexample-guided abstraction refinement approach.

Other researchers have used symbolic execution to improve the precision or speed of explicit state model checkers. For example, symbolic execution techniques have been combined with the search algorithms of Java PathFinder in order to verify properties of Java programs that manipulate complex data structures and that may even contain unbounded loops [Khurshid et al. 2003; Păsăreanu and Visser 2004]. Similarly, the ExpliSAT tool of Barner et al. [Barner et al. 2006] combines an explicit state vector with a symbolic CNF representation of part of the state. Recent work by Tomb, Brat, and Visser combine model checking with symbolic execution and a lightweight constraint solver to determine path feasibility and actual test data for paths in Java programs that violate language-specific properties [Tomb et al. 2007].

Our approach differs from this previous work in several ways: (1) in the way we use the path condition to filter out executions of the parallel program that are not consistent with a sequential execution, (2) in our emphasis on complex floating-point expressions, rather than on heap-allocated data and integer expressions, and (3) in our use of the value numbering scheme to represent the state space efficiently.

In another direction, the recent work of Elmas et al. [2005] also uses a non-

concurrent implementation as a specification for a concurrent program. That work, however, is directed at runtime verification of appropriate concurrent access to data structures and requires a special specification capturing an appropriate relaxation of atomicity. It does not consider the sort of parallel numerical programs we discuss here, for which construction of a sequential implementation is often a standard part of the development effort, and is not intended to determine the correctness of the numerical calculations implemented by the parallel program.

We also mention some work on model checking MPI programs. To the best of our knowledge, Matlin, Lusk, and McCune [Matlin et al. 2002] were the first to apply model checking techniques to an MPI problem, using SPIN to investigate a component of the MPI implementation MPICH. In [Siegel and Avrunin 2004; 2005] we showed how SPIN could be used to verify properties of simple MPI-based scientific programs; we also presented theorems for countering state-explosion for MPI programs that used only blocking functions and no wildcard receives. Some of these reduction results were generalized to deal with wildcard receives in [Siegel 2005] and nonblocking communication constructs in [Siegel and Avrunin 2007]. Pervez, Gopalakrishnan, Kirby, Thakur, and Gropp [Pervez et al. 2006] used SPIN to verify programs that use MPI's "one-sided" operations and found a subtle bug in one such program. In [Palmer et al. 2007], Palmer, Gopalakrishnan, and Kirby introduce a model for a somewhat different subset of MPI and show how dynamic partial order methods allow checking properties by considering only a subset of the possible traces. Pervez, Gopalakrishnan, Kirby, Palmer, Thakur, and Gropp [Pervez et al. 2007] have developed a model checking technique that works directly on source code, bypassing the model construction step. Using the modeling language TLA+, Palmer, Delisi, Gopalakrishnan, and Kirby [Palmer et al. 2007] have developed a formal description of a large portion of the MPI Standard and have integrated this into model checking tools. All of this work, however, is aimed at standard concurrency properties, rather than the correctness of the numerical calculation carried out by an MPI program. There are, however, a number of tools and techniques that can be used to estimate the error arising from floating-point computations in programs; see Martel [2005] for a description and comparison of some of these.

7. CONCLUSIONS AND FUTURE WORK

We have described a method that uses model checking techniques in combination with symbolic execution to verify the correctness of the calculations performed by parallel programs, including even complex floating-point calculations. We have successfully applied this method to four quite different examples, scaling to configurations of between 6 and 17 processes. While these numbers are much smaller than those that arise in practice, evidence from the application of model checking techniques with other kinds of software suggests that problems are usually exposed by verification of relatively small configurations. This is quite different from the case with testing, where the small size may make it difficult to trigger particular patterns of behavior. For example, a defect in an MPI-based program might only be revealed when a test case consumes enough memory to cause the MPI infrastructure to switch to a synchronous communication mode. In contrast, model checking would explore the possibility of forced synchronization whenever that possibility is

allowed by the MPI Standard. Hence the same defect could be uncovered by model checking a small configuration of the program.

The key idea of our method is to compare a sequential and a parallel program by using the path conditions arising from the sequential version as a filter when exploring the parallel version. This approach takes advantage of the fact that, since it is usually easier to construct a correct sequential numerical program, scientific software developers often start with a sequential version or develop one in tandem with the parallel version. There is, of course, a considerable amount of work on parallelizing compilers that can automatically construct a parallel implementation from sequential code (e.g, [Lim et al. 1999]). For the small examples on which we have tested our technique, it may be that such methods can produce correct parallel versions. The experience of developers of large parallel programs for scientific computing, however, indicates that automatic parallelization techniques cannot yet provide the performance of manual parallelization by skilled designers. Such manual parallelization, however, introduces the possibility of errors and it is these errors that our approach is designed to detect.

The approach does have several limitations. First, as it now stands, models of the sequential and parallel programs must be built by hand. This requires significant effort and a degree of skill on the part of the user. The ideal situation would be to have tools that automatically extract the models from source code, and indeed a great deal of research on this subject has been carried out, at least for other domains. As described in Section 5, we are exploring ways to adapt these techniques to MPI programs, though we expect to encounter some significant challenges when it comes to automatically creating models of programs with large amounts of floating-point data.

A second limitation is the need that sometimes arises to impose bounds on the number of loop iterations. Without this restriction, a model in which computations are performed symbolically might have an infinite number of states and would therefore not be amenable to standard model checking techniques.

A third limitation is the assumption that the computations performed in the sequential and parallel programs are carried out in a similar way, although the computations for the parallel program may be distributed in a complex manner. This assumption means that it is usually relatively inexpensive to determine if two symbolic expressions are equivalent or if one symbolic predicate implies another. The more distant the computations in the two programs become, the more powerful the symbolic manipulation must be in order to arrive at a conclusive result. If the computations performed by the two programs are very different, we might argue that the sequential program is not a good specification for the parallel one. Nevertheless, as we examine more complex programs, it is certainly possible that the kind of lightweight symbolic manipulation and theorem proving that we are currently using will no longer suffice. For this reason, we are exploring ways to integrate our approach with more sophisticated symbolic algebra and theorem proving tools.

Perhaps the greatest problem with model checking parallel programs is *state explosion*: the fact that the number of states of a program typically grows exponentially with the number of processes. A vast array of techniques has been developed to counteract this problem, and we have demonstrated that some of

these, such as partial order and symmetry reductions, can be adapted to work with our method. SPIN turned out to be an excellent platform for the rapid prototyping of our method. The MPI-SPIN extension allows for easy modeling of many features of MPI programs, including the nonblocking communication constructs, though further work is needed to make MPI-SPIN models as efficient for verification as the hand-coded ones. Furthermore, some of the techniques we want to explore for reducing the difficulty of verification may be difficult to implement in SPIN, and thus we also intend to explore other model checkers, such as Bogor [Robby et al. 2003], which is designed to allow easy customizations of its search strategy and other components. We also plan to look for additional theoretical results that can reduce the difficulty of verification.

Given these limitations, we certainly cannot claim that our method can be used to verify the correctness of every parallel numerical program. But we have shown that it works on some interesting, non-trivial examples and that when it is applicable, it seems to be a very effective approach for dealing with a very difficult problem. In addition, we expect to significantly increase the range of applicability of the method as we incorporate new and existing techniques from model checking, theorem proving, and symbolic algebra.

REFERENCES

BALL, T. AND RAJAMANI, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *Model Checking Software: 8th International SPIN Workshop, Toronto, Canada, May 19–20, 2001, Proceedings*, M. B. Dwyer, Ed. Lecture Notes in Computer Science, vol. 2057. Springer-Verlag, 103–122.

BARNER, S., EISNER, C., GLAZBERG, Z., KROENING, K., AND RABINOVITZ, I. 2006. ExpliSat: Guiding SAT-based software verification with explicit states. In *Hardware and Software, Verification and Testing*. Lecture Notes in Computer Science, vol. 4383. Springer-Verlag, 138–154.

BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*. ACM Press, 234–245.

BRAND, D. AND W. H. JOYNER, J. 1978. Verification of protocols using symbolic execution. *Comput. Networks 2*, 351–360.

CAPPELLO, F., HERAULT, T., AND DONGARRA, J., Eds. 2007. *EuroPVM/MPI*. Lecture Notes in Computer Science, vol. 4757. Springer-Verlag. To appear.

CLARKE, L. A. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering 2,* 3, 215–222.

COUSOT, R., Ed. 2005. *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005, Paris, January 17–19, 2005, Proceedings*. Lecture Notes in Computer Science, vol. 3385.

DILLON, L. K. 1990. Using symbolic execution for verification of ada tasking programs. *ACM Trans. Program. Lang. Syst. 12,* 4, 643–669.

ELMAS, T., TASIRAN, S., AND QADEER, S. 2005. VYRD: Verifying concurrent programs by runtime refinement-violation detection. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 27–37.

GOLDBERG, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23,* 1 (Mar.), 5–48.

GRAF, S. AND MOUNIER, L., Eds. 2004. *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings*. Lecture Notes in Computer Science, vol. 2989. Springer-Verlag.

Gropp, W., Lusk, E., and Skjellum, A. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press.

Hantler, S. L. and King, J. C. 1976. An introduction to proving the correctness of programs. *ACM Computing Surveys 8,* 3, 331–353.

Holzmann, G. J. 2004. *The* Spin *Model Checker*. Addison-Wesley.

IEEE. 1985. 754-1985 IEEE standard for binary floating-point arithmetic.

IEEE. 1987. 854-1987 IEEE standard for radix-independent floating-point arithmetic.

Karniadakis, G. E. and Kirby II, R. M. 2003. *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press.

Khurshid, S., Păsăreanu, C. S., and Visser, W. 2003. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, H. Garavel and J. Hatcliff, Eds. Lecture Notes in Computer Science, vol. 2619. Springer-Verlag, 553–568.

Lim, A. W., Cheong, G. I., and Lam, M. S. 1999. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*. ACM Press, New York, NY, USA, 228–237.

Martel, M. 2005. An overview of semantics for the validation of numerical programs. See Cousot [2005], 59–77.

Matlin, O. S., Lusk, E., and McCune, W. 2002. SPINning parallel systems software. In *Model Checking of Software: 9th Intl. SPIN Workshop*, D. Bosnacki and S. Leue, Eds. LNCS, vol. 2318. Springer-Verlag, 213–220.

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

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

Palmer, R., Delisi, M., Gopalakrishnan, G., and Kirby, R. M. 2007. An approach to formalization and analysis of message passing libraries. In *Proceedings of the 12th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS)*. LNCS. Springer. To appear.

Palmer, R., Gopalakrishnan, G., and Kirby, R. M. 2007. Semantics driven partial-order reduction of MPI-based parallel programs. ACM Press, New York, NY, USA, 43–53. Program Chair-Shmuel Ur and Program Chair-Eitan Farchi.

Pervez, S., Gopalakrishnan, G., Kirby, R. M., Palmer, R., Thakur, R., and Gropp, W. 2007. Practical model checking method for verifying correctness of MPI programs. See Cappello et al. [2007]. To appear.

Pervez, S., Gopalakrishnan, G., Kirby, R. M., Thakur, R., and Gropp, W. 2006. Formal verification of programs that use MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, B. Mohr, J. L. Träff, J. Worrington, and J. Dongarra, Eds. LNCS, vol. 4192. 30–39.

Păsăreanu, C. S. and Visser, W. 2004. Verification of Java programs using symbolic execution and invariant generation. See Graf and Mounier [2004], 164–181.

Robby, Dwyer, M. B., and Hatcliff, J. 2003. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, Helsinki, Finland, 267–276.

Rotman, J. J. 1995. *An Introduction to the Theory of Groups*, Fourth ed. Springer, New York.

Siegel, S. F. 2004. Efficient verification of halting properties for MPI programs with wildcard receives. Tech. Rep. UM-CS-2004-77, Department of Computer Science, University of Massachusetts.

Siegel, S. F. 2005. Efficient verification of halting properties for MPI programs with wildcard receives. See Cousot [2005], 413–429.

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

Siegel, S. F. 2007b. Model checking nonblocking MPI programs. In *Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14–16, 2007, Proceedings*, B. Cook and A. Podelski, Eds. Lecture Notes in Computer Science, vol. 4349. 44–58.

Siegel, S. F. and Avrunin, G. S. 2004. Verification of MPI-based software for scientific computation. See Graf and Mounier [2004], 286–303.

Siegel, S. F. and Avrunin, G. S. 2005. Modeling wildcard-free MPI programs for verification. In *Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming: PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA*. ACM Press, 95–106.

Siegel, S. F. and Avrunin, G. S. 2007. Verification of halting properties for MPI programs using nonblocking operations. See Cappello et al. [2007], 326–334. To appear.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. 1998. *MPI—The Complete Reference, Volume 1: The MPI Core*, Second ed. MIT Press.

Tomb, A., Brat, G., and Visser, W. 2007. Variably interprocedural program analysis for runtime error detection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA 2007)*. ACM, 97–107.