Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs

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

Anastasia Mironova
School of Computing
University of Utah
Salt Lake City, UT 84112
mironova@sci.utah.edu

George S. Avrunin and Lori A. Clarke
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
{avrunin,clarke}@cs.umass.edu

ABSTRACT

We present a method to verify the correctness of parallel programs that perform complex numerical computations, including computations involving floating-point arithmetic. The 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.

Categories and Subject Descriptors

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

General Terms

Verification

Keywords

Finite state verification, numerical program, floating-point, model checking, concurrency, parallel programming, high performance computing, symbolic execution, MPI, Message Passing Interface, Spin

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, i.e., parallel programs that take as input a vector of (usually floating-point) numbers and produce as output another such vector. 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.

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

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 which makes it possible to easily express the model in the language of standard model checking tools, such as SPIN [10].

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 *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 parallel one, and it is already common for developers of scientific software to begin with a sequential version of the program or to construct one, for testing and other purposes. Moreover, we will see below that 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. Our method deals with this as follows. We use the model checker to explore all possible paths of the sequential program, and for each such path we record the *path condition pc*, the Boolean-valued symbolic expression on the input that must hold in order 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.

The method is described in detail in Section 2. 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 3. Section 4 discusses related work and Section 5 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) [16, 17]. (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 [20, 21]. 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.

Notice that the requirement that P_{par} and P_{seq} be equivalent implies, in particular, that each program be *deterministic*, i.e., that if given the same input twice, it will produce the same output. If either program fails to be deterministic, this will be caught and flagged as an error by our method.

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(a). 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 1(b), which is adapted from [8] 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 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.

```
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];
  }
```

(a) Sequential 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);
  }
}
```

(b) Parallel code

Figure 1: Matrix multiplication code excerpts

The model will use symbolic expressions in place of the floating-point values that arise in P_{seq}. (Integer values can also be modeled symbolically, though this is often not necessary.) A symbolic expression may be thought of as a tree-like structure in which the leaf nodes are either floating-point literals or symbolic constants. The symbolic constants are denoted $x_0, x_1, \ldots$ and correspond to the components of the input vector. To each non-leaf node in the tree is associated 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 from a given operator and one or two operands. We will use the usual infix notation to denote symbolic expressions, but we must keep in mind 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 we will see shortly that 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 [21] and [22]. 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 $(\texttt{X}, i)$, corresponding to the symbolic constant x_i, or $(\texttt{L}, \alpha)$, 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 every expression has a unique entry in the table, and so the expressions corresponding to two integers i and j are equal if and only if $i = j$.

The table that is constructed during the verification of the 2×2 matrix multiplication example is excerpted in Figure 2. 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.

In this case, 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 3), 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 [6]. (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 [11, 12]—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.

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	$(+, 23, 24)$	$(0.0 + x_2x_5) + x_3x_7$

Figure 2: Symbolic expression table for 2×2 matrix multiplication

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. We will see in Section 3.1 that 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 very inexpensive and provides sufficient precision for the examples we have looked at.

Each operation-preserving equivalence relation yields a different notion of program equivalence. We have identified three that we think are useful and have used in our implementation, though the same methods 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 had real arithmetic 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, [15] and the references cited there.

We mentioned above that 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.

```
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;
    }
}
```

Figure 3: Sequential Gaussian elimination code

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 3, 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.

```
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;
  }
}
```

Figure 4: Parallel Gaussian elimination code

In the parallel version (Figure 4), 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 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. To 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.)

We deal with this as follows. In M_{seq}, we model the floating-point variables symbolically, as before, but we also introduce an integer variable that gives the index in the symbolic table of the current path condition pc for the program. 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 3.1; 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 the Gaussian elimination example, 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. 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://laser.cs.umass.edu/~siegel/projects`.

3.1 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 `structs`, 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, in order 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 *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.

3.2 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 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).

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 [20], but we could not find an easy way to implement them in SPIN. Instead, we proved a result [23, Theorem 1] that justifies slightly weaker techniques, but 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.

In Table 1, we give data for the largest configuration of each program that we were able to verify. The columns 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.

3.2.1 *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 $n \leq 6$.

3.2.2 *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.

3.2.3 *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 [13]. 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 programs 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 $\mathbf{x}$ with itself, the code simply took the inner product of the two successive values. After correcting this error, we verified real equivalence (which is the best that can be hoped for, 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.

3.2.4 *monte*

Our fourth example is a parallel program taken from [8] 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

program name	equivalence type	parallel processes	sequential executions	symbolic expressions	input vector	output vector	path condition	states (10^3)	memory (MB)	time (s)
matmat	Herbrand	6	1	2202	200	100	0	4443	217	506
gauss	Herbrand	6	13327	247656	36	36	36	16114	801	3224
jacobi	real	17	4	8239	1333	36	3	6295	362	9846
monte	IEEE	9	4	1232	99	1	3	3112	279	738

Table 1: Experimental data

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 nondeterministic 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.

The second reduction technique exploited a *symmetry reduction theorem* [23, Theorem 2] that we proved for general parallel numerical programs. To see how this comes into play in this example, observe that in M_{par}, 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, and the model checker will be forced to explore all of them, leading to a rapid blowup in the number of states of M_{par}. This problem does *not*, however, arise for M_{seq}, because M_{seq} utilizes the random numbers in a fixed order. Moreover, for all executions of M_{seq}, both the output vector and the path condition turn out to be invariant under these permutations. The upshot of our symmetry reduction theorem is that in these circumstances, it suffices to explore only one of these permutations in M_{par}, rather than all of them.

Using these reductions, 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.

4. RELATED WORK

The idea of representing computations symbolically has a long history and has enjoyed many applications, including to testing and debugging (e.g., [2,3,9]). There has also been some work incorporating these ideas into model checking. For example, a component of the SLAM toolkit [1] 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.

Symbolic execution has also been used to improve the precision of Java PathFinder, in order to verify properties of Java programs that manipulate complex data structures and that may even contain unbounded loops [14, 18].

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 Elams, Tasiran, and Qadeer [5] 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.

There are a number of tools and techniques that can be used to estimate the error arising from floating-point computations in programs; see [15] for a description and comparison of some of these.

5. 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—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 pathological patterns of behavior. The difference is due to the fact that model checking takes into account all possible executions of the model.

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.

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. 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 must be computed 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 further removed 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, although it would be difficult to code some of the optimizations that we wanted to consider. We plan to explore these using the Bogor model checker [19], which is designed to allow easy customizations of its search strategy and other components. We intend to try to take advantage of this platform to explore a wide range of optimizations and extensions to our method.

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.

Acknowledgments

This material is based upon work supported by the National Science Foundation under awards CCF-0427071 and CCR-0205575 and by the U.S. Department of Defense/Army Research Office under awards DAAD19-01-1-0564 and DAAD-19-03-1-0133. We also wish to thank the Computing Research Association's CRA-W Distributed Mentor Program and the College of Natural Science and Mathematics at the University of Massachusetts for sponsoring the program and

funding, respectively, for 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.

6. REFERENCES

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M. B. Dwyer, editor, *Model Checking Software: 8th International SPIN Workshop, Toronto, Canada, May 19–20, 2001, Proceedings*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2001.

[2] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245. ACM Press, 1975.

[3] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[4] R. Cousot, editor. *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005, Paris, January 17–19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, 2005.

[5] T. Elmas, S. Tasiran, and S. Qadeer. 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*, pages 27–37, New York, NY, USA, 2005. ACM Press.

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

[7] S. Graf and L. Mounier, editors. *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*. Springer, 2004.

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

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

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

[11] IEEE. 754-1985 IEEE standard for binary floating-point arithmetic, 1985.

[12] IEEE. 854-1987 IEEE standard for radix-independent floating-point arithmetic, 1987.

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

[14] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *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*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[15] M. Martel. An overview of semantics for the validation of numerical programs. In Cousot [4], pages 59–77.

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

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

[18] C. S. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In Graf and Mounier [7], pages 164–181.

[19] Robby, M. B. Dwyer, and J. Hatcliff. 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*, pages 267–276, Helsinki, Finland, 2003. ACM Press.

[20] S. F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In Cousot [4], pages 413–429.

[21] S. F. Siegel and G. S. Avrunin. Verification of MPI-based software for scientific computation. In Graf and Mounier [7], pages 286–303.

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

[23] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. Technical Report UM-CS-2005-15, Department of Computer Science, University of Massachusetts, 2005.