Verification of MPI-based Computations

Stephen F. Siegel*

Verified Software Laboratory, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716, USA, siegel@cis.udel.edu

Mpi-Spin [1, 2] is an extension to the model checker Spin [3, 4] for verifying correctness of MPI-based parallel and distributed algorithms. It adds to Spin's input language a number of types, constants, and functions corresponding to primitives in the MPI Standard [5]. The semantics of these added primitives are defined using an abstract model of a generic MPI implementation which encodes all possible behaviors permitted by the Standard. Thus Mpi-Spin can verify that an algorithm will behave correctly on any (correct) MPI platform.

By default, Mpi-Spin will check for a number of standard defects in MPI-based algorithms: deadlocks, failure to wait for the completion of communication requests before finalization, attempts to access buffers used in active communication operations, out-of-order invocations of collective functions, and so on. It can also be used to verify the functional correctness of an algorithm—that an algorithm will always produce the correct result on any given input [6].

Functional correctness is specified by providing a sequential version of the algorithm which is presumed to be correct. The problem is then reduced to verifying that the parallel and sequential versions are functionally equivalent. This is accomplished using *symbolic execution* [7]: the inputs to the algorithms are modeled using symbolic constants X_i and the outputs are expressed as symbolic expressions in the X_i. Nondeterministic choice is used to model branches and the branch decision is recorded in a symbolic *path condition* variable `pc`. A simple decision procedure can detect that `pc` becomes unsatisfiable, at which point the search backtracks.

To compare the two versions, a composite model is constructed in which `pc` is initialized to *true*, the sequential model is executed, and then the parallel model is executed with the value of `pc` resulting from the sequential execution. Finally, an assertion is used to check that the symbolic results from the two versions coincide. A depth-first search of the state space of the composite model will explore all possible executions of the sequential version, each of which results in a path condition-symbolic output pair. For each of these pairs, all possible executions of the parallel version consistent with `pc` are explored, and the outputs are compared. Thus the path condition is used to "match up" executions of the two versions on the same inputs domains.

The notion of *equivalence* depends upon the semantics one associates to the numerical operations. Successively weaker notions of equivalence supported by Mpi-Spin are *Herbrand* (operations are treated as uninterpreted functions),

* Research supported by the University of Delaware and the U.S. National Science Foundation under Grants CCF-0733035 and CCF-0540948

```
Solution 27:
Path condition is conjunction
of the following:
  0. x8-x6*(x2/x0) == 0
  1. x5-x3*(x2/x0) == 0
  2. x7-x6*(x1/x0) == 0
  3. x4-x3*(x1/x0) == 0
  4. x0 != 0
Reduced row-echelon form:
   x0/x0         x1/x0 x2/x0
   x3-x3*(x0/x0) 0     0
   x6-x6*(x0/x0) 0     0
All parallel solutions
are Herbrand-equivalent
to the sequential solution.
```

```
Solution 27:
Path condition is conjunction
of the following:
  0. x8-x6*(x2/x0) == 0
  1. x5-x3*(x2/x0) == 0
  2. x7-x6*(x1/x0) == 0
  3. x4-x3*(x1/x0) == 0
  4. x0 != 0
Reduced row-echelon form:
   1 x1/x0 x2/x0
   0 0     0
   0 0     0
All parallel solutions
are real-equivalent
to the sequential solution.
```

Fig. 1. Gaussian elimination: MPI-SPIN output for 3×3 matrices (excerpt)

IEEE754 (corresponding the IEEE754 floating-point standard) and *Real* (values are treated as Real numbers). An excerpt of the output of MPI-SPIN on a Gaussian elimination algorithm is shown in Fig. 1. One of the path condition-output pairs resulting from the sequential version is displayed, followed by the conclusion that all possible executions of the parallel version consistent with that path condition produce the same result. Fig. 1(left) uses Herbrand arithmetic; Fig. 1(right) uses Real arithmetic. In this example, equivalence holds in both cases, though the output is simplified in the Real case.

Methods are currently being explored to improve the symbolic algorithm in various ways. The first is the use of abstraction to reduce the number of variables and computations in the composite model. An example dealing with a matrix-matrix multiplication algorithm (derived from [8]) is shown in Fig. 2. In this case, the sequential and parallel versions both make use of a vector-matrix multiplication helper function `vecmat`. To prove equivalence, it is not necessary to know anything about `vecmat`, so this function can be modeled using an uninterpreted symbolic function `VECMAT`. Furthermore, the state of all the variables comprising one row of input matrix A can be represented using a single symbolic variable; the entire matrix B can be represented using a single symbolic variable, and each row of C can be represented using one symbolic variable. Methods to automatically find these abstraction opportunities and perform the corresponding model transformations are being studied.

Matthew Dwyer has pointed out that the symbolic comparison algorithm lends itself naturally to parallelization. A single process could explore the sequential version and each path condition-output pair produced could be forked off to a separate process to explore the corresponding parallel executions. The processes exploring the parallel executions can run in parallel with no communication between them. This is because the path conditions are pairwise mutually exclusive, so it is not possible for the state spaces generated by two distinct

```
double a[N][L], b[L][M], c[N][M];
for (i=0; i<N; i++) vecmat(a[i], b, c[i]);
```

(a) specification: sequential code using vector-matrix multiplication function

```
double b[L][M], in[L], out[M];
while (1) {
  MPI_Recv(in, L, MPI_DOUBLE, 0, MPI_ANY_TAG, comm, &status);
  if (status.MPI_TAG == 0) break;
  vecmat(in, b, out);
  MPI_Send(out, M, MPI_DOUBLE, 0, status.MPI_TAG, comm);
}
```

(b) parallel "master-worker" implementation: excerpt of worker code

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \mapsto \begin{bmatrix} \tilde{a}_0 \\ \tilde{a}_1 \\ \tilde{a}_2 \end{bmatrix}, \quad \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \end{bmatrix} \mapsto \tilde{b}, \quad \begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \\ c_{20} & c_{21} \end{bmatrix} \mapsto \begin{bmatrix} \tilde{c}_0 \\ \tilde{c}_1 \\ \tilde{c}_2 \end{bmatrix} = \begin{bmatrix} \texttt{VECMAT}(\tilde{a}_0, \tilde{b}) \\ \texttt{VECMAT}(\tilde{a}_1, \tilde{b}) \\ \texttt{VECMAT}(\tilde{a}_2, \tilde{b}) \end{bmatrix}$$

(c) mapping of concrete variables to abstract variables

Fig. 2. Matrix-matrix multiplication: abstractions used to verify functional correctness

processes to intersect. A parallel version of the algorithm for grid or cluster environments is under development.

References

1. Siegel, S.F.: The Mpi-Spin web page. `http://vsl.cis.udel.edu/mpi-spin` (2008)
2. Siegel, S.F.: Model checking nonblocking MPI programs. In Cook, B., Podelski, A., eds.: Verification, Model Checking, and Abstract Interpretation: 8th Intl. Conference (VMCAI 2007). Volume 4349 of LNCS. (2007) 44–58
3. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
4. : "Spin—Formal Verification" web site. `http://spinroot.com` (2008)
5. Message Passing Interface Forum: MPI: A Message-Passing Interface standard, version 1.1. `http://www.mpi-forum.org/docs/` (1995)
6. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. Transactions on Software Engineering and Methodology **17**(2) (2008) Article No. 10, 1–34
7. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976) 385–394
8. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press (1999)