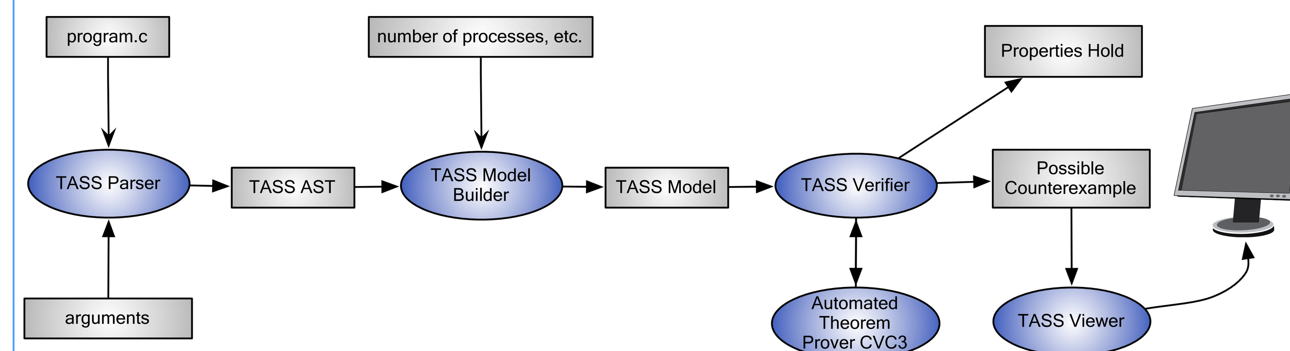
Automatic Formal Verification of MPI-Based Parallel Programs

Stephen F. Siegel, Timothy K. Zirkel, Yi Wei
The Verified Software Laboratory, Department of Computer and Information Sciences, University of Delaware, Newark DE 19716

1. Abstract

The Toolkit for Accurate Scientific Software (TASS) is a suite of tools for the formal verification of MPI-based parallel programs used in computational science. TASS can verify various safety properties as well as compare two programs for functional equivalence.

Developing correct, portable MPI-based parallel programs is difficult for several reasons. First, traditional testing methods can only sample a small portion of a program's parameter and input space. Second, there are many sources of nondeterminism arising from MPI, e.g., how statements from different processes are interleaved in time, the buffering policy of the MPI implementation, and the use of "wildcard" receives. Often, a program may appear to run correctly on one machine, but when ported to a new platform which resolves the nondeterminism differently, a defect is revealed. These defects may manifest themselves as deadlocks or as results which are incorrect or differ unexpectedly from execution to execution. Such defects are often difficult to identify and isolate.



The TASS front end takes an integer $n \geq 1$ and a C/MPI program, and constructs an abstract model of the program with n processes. Procedures, structs, (multi-dimensional) arrays, heap-allocated data, pointers, and pointer arithmetic are all representable in a TASS model. The model is then explored using symbolic execution and explicit state space enumeration. A number of techniques are used to reduce the time and memory consumed. A variety of realistic MPI programs have been verified with TASS, including Jacobi iteration and manager-worker type programs, and some subtle defects have been discovered. TASS is written in Java and is available from http://vsl.cis.udel.edu/tass under the Gnu Public License.

2. Symbolic Execution

TASS uses **symbolic execution** [1,2] to reason about all possible inputs to a program. The user may insert pragmas into the C source to identify input variables and specify assumptions restricting input values, e.g.,

```
#pragma TASS input {N > 1 && N <= 10}
int N;
```

Arguments to main are treated as inputs and pre-processor object-like macros can also be declared as inputs. Each such variable is initially assigned a unique *symbolic constant* (*X1, X2, ...*). As TASS "executes" the program, operations on these values result in more complex symbolic expressions.

Branches are treated as nondeterministic choices, and a boolean-valued *path condition* (*pc*) records the choices made. If *pc* becomes unsatisfiable, it means a path is not feasible. TASS attempts to determine whether *pc* is infeasible, and invokes the automated theorem prover CVC3 [3] if it cannot.

3. Safety Properties Verified

TASS can verify a variety of safety properties in a single C/MPI program. These include:

- Absence of (absolute or potential) deadlocks
- Absence of buffer overflows from pointer arithmetic, array indexing, etc.
- Absence of reading uninitialized variables
- Absence of division by zero
- Absence of memory leaks
- Proper use of `malloc` and `free`
- Absence of assertion violations
- Type and size of a message received with `MPI_Recv` are compatible with the receive buffer/type
- Proper use of `MPI_Init` and `MPI_Finalize`

4. Comparative Symbolic Execution

TASS pragmas can identify output as well as input variables. These annotations specify an input set *X* and an output set *Y*. Thus the program can be considered a function $f: X \rightarrow Y$. When two programs have the same input and output sets, TASS can check that they compute the same functions. In TASS, the first program of a comparison is referred to as the **specification**, and the second as the **implementation**.

```
#pragma TASS input int
#define B 10
#pragma TASS input {n>=0 && n<=B} int
#define n 10
#pragma TASS input
double a[n];
#pragma TASS output
double sum;

void main() {
    double result = 0.0;
    int i;

    for (i=0; i<n; i++) result += a[i];
    sum = result;
}
```

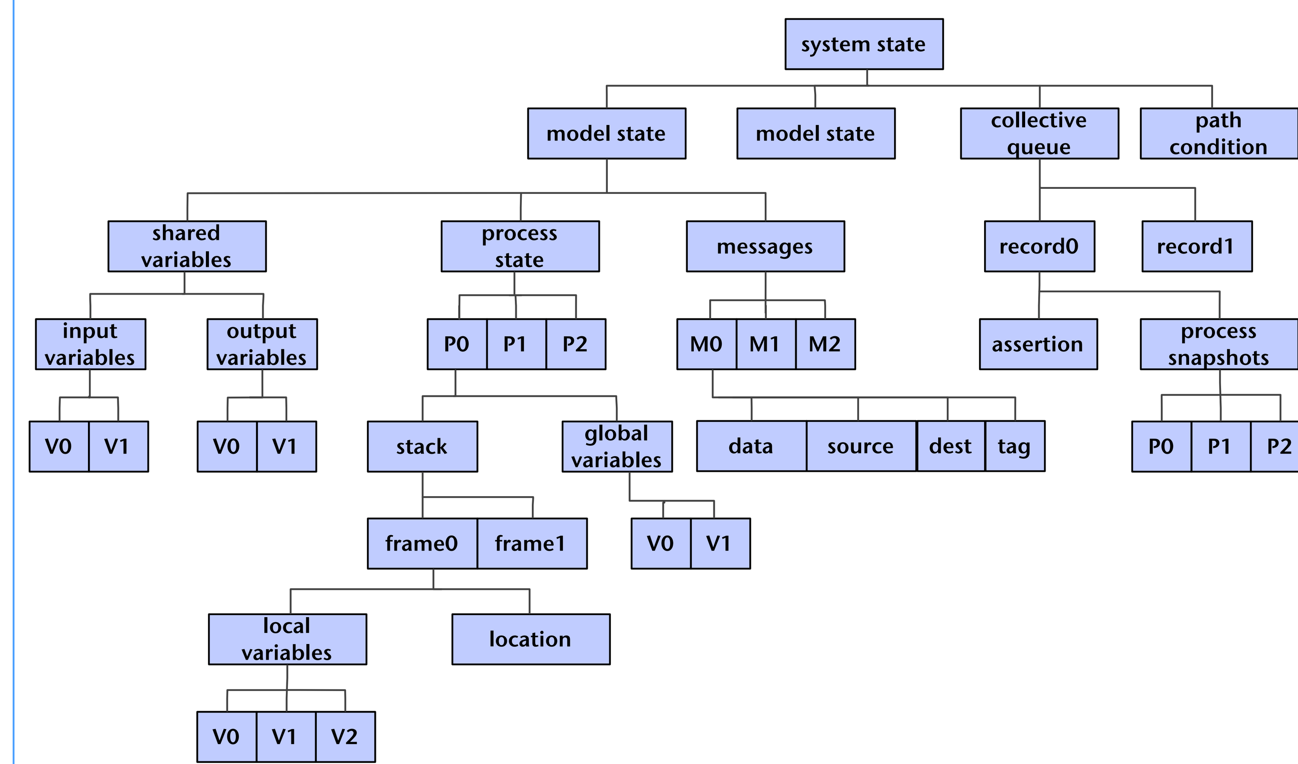
```
        ⋮
double computeGlobalSum() {
    double result = localSum, buffer;
    int i;

    for (i=1; i<nprocs; i++) {
        MPI_Recv(&buffer, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        result += buffer;
    }
    return result;
}

void main(int argc, char **argv) {
    int first, afterLast, i;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    first = n*myrank/nprocs;
    afterLast = n*(myrank+1)/nprocs;
    for (i=first; i<afterLast; i++) localSum += a[i];
    if (myrank == 0)
        sum = computeGlobalSum();
    else
        MPI_Send(&localSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Finalize();
}
```

The programs above compute the sum of elements in an array in two different ways. TASS will verify that for all array lengths up to the given bound B and for all possible real number array entries, the result will be the same for both programs using **comparative symbolic execution** [4]. A *composite model* is constructed in which the two programs run concurrently. The same symbolic constants are used to initialize the input variables in the two programs, and a single path condition variable is used. At each terminal state, an assertion comparing the output variables in the two programs is checked. If the assertion holds on all executions, the programs are functionally equivalent.

5. States

A **symbolic state** assigns a symbolic expression to each variable in the program, including *pc*. It represents a **set** of concrete states: any assignment of concrete values to symbolic constants that satisfies *pc* results in a concrete state in the set.

TASS uses explicit state enumeration techniques to deal with the nondeterminism arising from MPI (as well as from symbolic execution branches). An explicit representation of the state is used, and the set of all reachable states is explored with a depth-first search. Typically, the user places bounds on certain variables to ensure the state space is finite (or reasonably small).

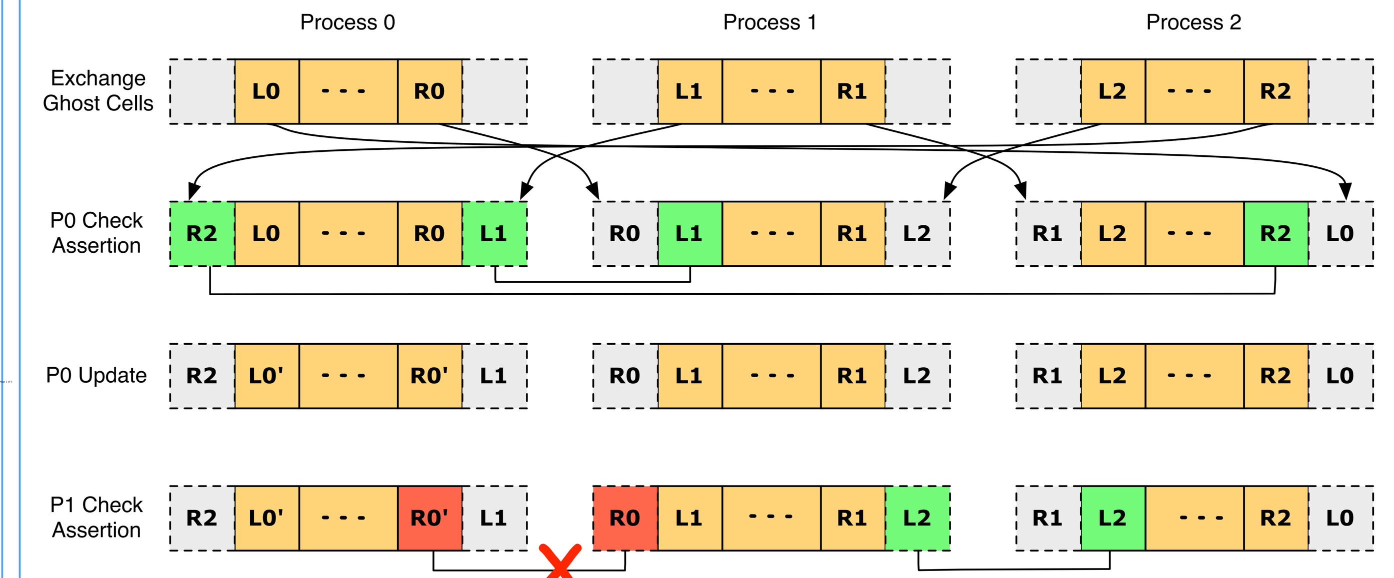


Two symbolic states are *equivalent* if they represent the same set of concrete states. The ability to recognize equivalence can greatly reduce the search space. This can often be accomplished by transforming symbolic expressions into a canonical form. For example, TASS transforms every real-valued expression into a quotient *p*/*q*, where *p* and *q* are polynomials in *primitive* expressions. A *primitive expression* is any non-concrete expression that is not the result of +, -, *, or /; symbolic constants and array-read and array-write expressions are examples.

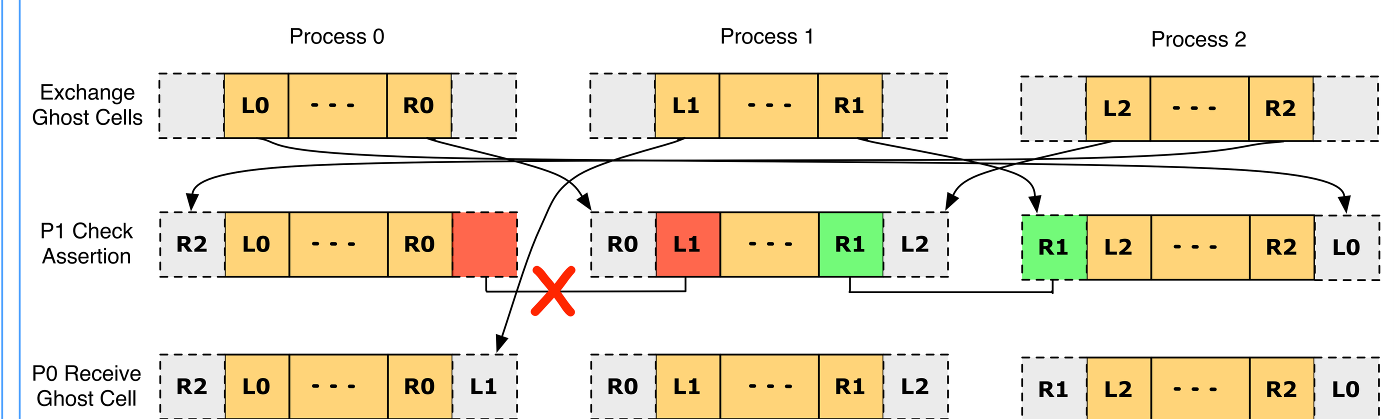
6. Collective Assertions

TASS provides a powerful generalization of standard assertions called **collective assertions** [5]. Collective assertions allow for specifying relations among variables in different processes. They work analogously to MPI's collective operations, but do not add any synchronization or change the semantics of the program in any way.

A collective assertion is not simply an assertion on the global state of the program. To see why, consider a block-distributed 1d diffusion solver with `NPROCS` processes. In order to update cells on the border of a block, processes maintain *ghost cells* that have the value of their right neighbor's leftmost cell and left neighbor's rightmost cell. After ghost cells are exchanged, it is desirable to assert that they are correct. Each process will repeatedly exchange ghost cells, check an assertion, and update cells. There are two views of this assertion. The first is "My ghost cells are correct." Below we show one possible execution.

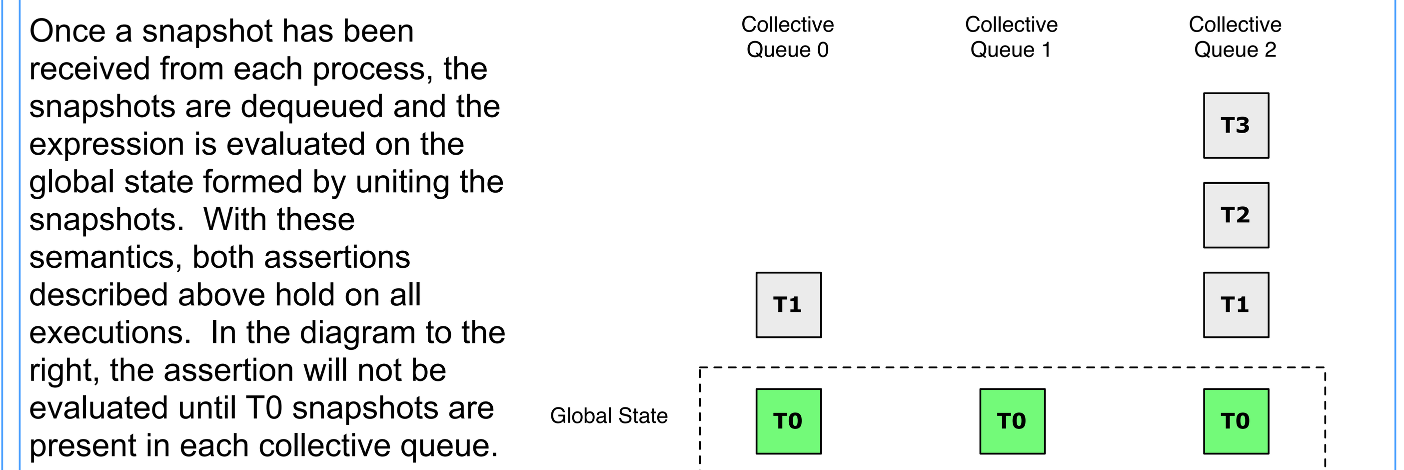


When Process 1 checks its assertions, one fails because Process 0 has already updated. But there is nothing wrong with this execution – the problem lies in interpreting the assertion as a predicate on the global state. The dual view of the assertion is "My neighbor's ghost cells are correct." Since messages may be buffered, this view is subject to a similar failure. In this exchange, the message from Process 1 to Process 0 is buffered and not immediately received.



This example demonstrates why the assertion semantics must be defined in a way that is appropriate for message-passing programs. Collective assertions are one such generalization. A single collective assertion comprises a set of locations in each process and an expression on the global state. The semantics are defined as follows: whenever control in a process reaches one of the locations, a "snapshot" of the local state of that process is saved in a queue.

Once a snapshot has been received from each process, the snapshots are dequeued and the expression is evaluated on the global state formed by uniting the snapshots. With these semantics, both assertions described above hold on all executions. In the diagram to the right, the assertion will not be evaluated until T0 snapshots are present in each collective queue.



TASS uses the keyword `joint` to indicate a collective assertion that is used only when comparing two programs. An implementation can refer to variables in the specification using the `spec.var` syntax. Any program can refer to a variable in a different process using the `PROC[pid].var` syntax, where `pid` is any integer valued expression. The code below shows the TASS pragmas for the main loop of parallel 1d diffusion. The identifiers `GHOSTS` and `COMPARE` indicate the assertion to which the pragma belongs. One assertion is created for each identifier.

```
for (time = 1; time <= nsteps; time++) {
    exchange ghost cells();
#pragma TASS collective assert GHOSTS u[nxl] == PROC[right].u[0] \
      && u[1] == PROC[left].u[PROC[left].nxl+1];
    update();
#pragma TASS joint assert COMPARE forall {int i | 1<=i && i<=nxl} \
      u[i]==spec.u[first+i-1];
}
```

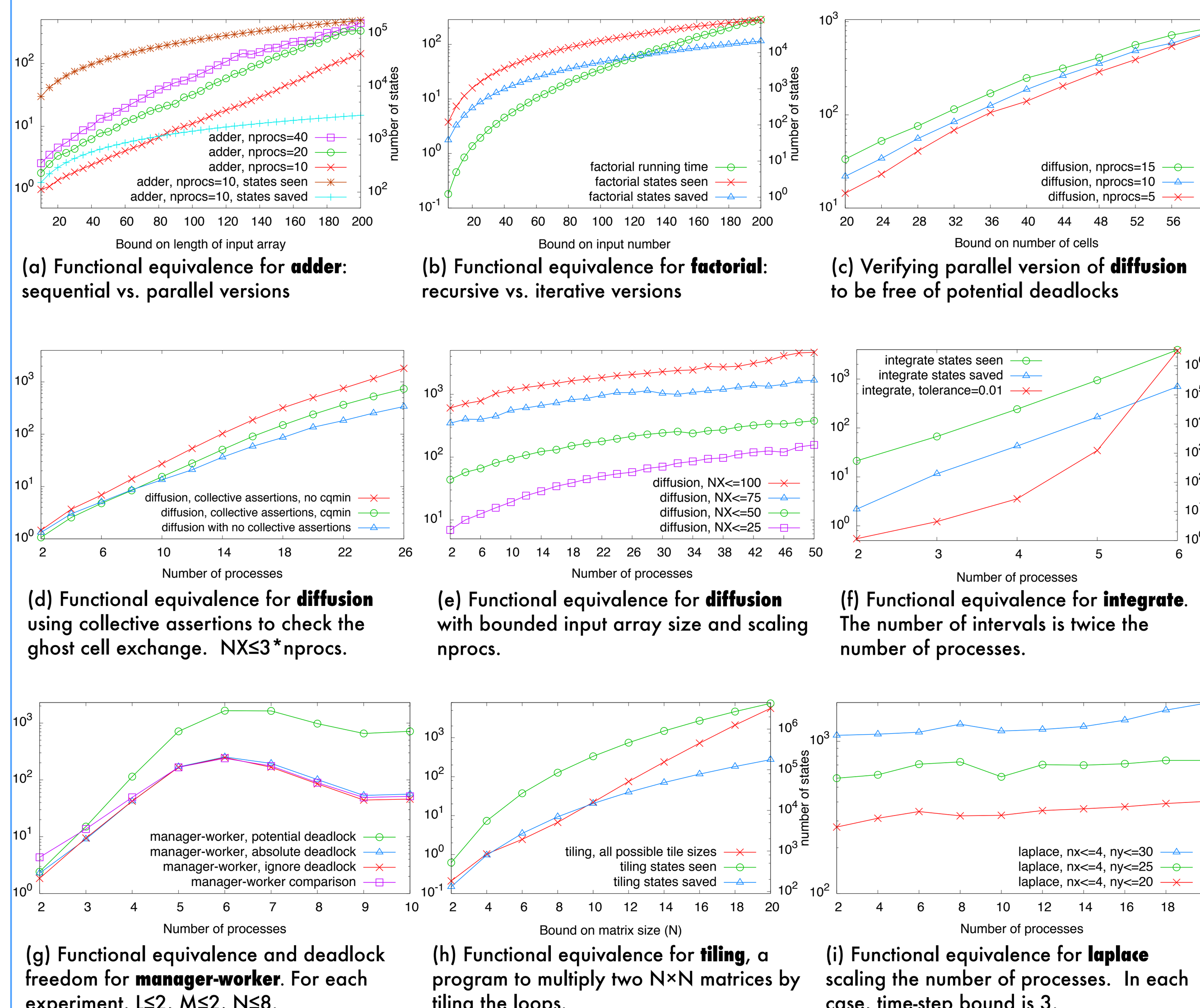
The corresponding code in the sequential version is:

```
for (time = 1; time <= nsteps; time++) {
    update();
#pragma TASS joint assert COMPARE true;
}
```

With these pragmas, TASS can verify both correctness of the ghost cell exchange and equivalence of cell values after each iteration.

7. Performance

These graphs describe resource use by TASS when analyzing a variety of C programs. In each case the left y-axis is time in seconds.



(a) Functional equivalence for **adder**: sequential vs. parallel versions
(b) Functional equivalence for **factorial**: recursive vs. iterative versions
(c) Verifying parallel version of **diffusion** to be free of potential deadlocks
(d) Functional equivalence for **diffusion** using collective assertions to check the ghost cell exchange. NX≤3*nprocs.
(e) Functional equivalence for **diffusion** with bounded input array size and scaling nprocs.
(f) Functional equivalence for **integrate**. The number of intervals is twice the number of processes.
(g) Functional equivalence and deadlock freedom for **manager-worker**. For each experiment, L≤2, M≤2, N≤8.
(h) Functional equivalence for **tiling**, a program to multiply two N×N matrices by tiling the loops.
(i) Functional equivalence for **laplace** scaling the number of processes. In each case, time-step bound is 3.

8. Future Work

Several enhancement to TASS are planned. First, support for more C libraries is in progress. In particular, the C `math`, `float`, and `stdio` libraries are short term targets. Additionally, we plan to expand the MPI support to handle non-blocking MPI calls. Work is under way to add support for verifying the numerical accuracy of computations. The only currently supported input language is C. We plan to extend TASS to handle other languages such as C++ and Fortran. Finally, we eventually plan to support other high-performance computing APIs such as OpenMP, OpenCL, and CUDA.

Supported by the National Science Foundation under Grants CCF-0733035 and CCF-0953210 and by the University of Delaware Research Foundation

9. References

1. King, J.C.: Symbolic execution and program testing. Comm. ACM **19**(7) (1976) 385–394
2. Khursid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. TACAS 2003. LNCS **2619**. 553–568
3. Barrett, C., Tinelli, C.: CVC3. CAV 2007. LNCS **4590**. 298–302
4. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM TOSEM **17**(2) (2008) Article 10, 1–34
5. Siegel, S.F., Zirkel, T.K.: Collective assertions. VMCAI 2011. LNCS **6538**. 387–402
6. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, version 2.2, September 4, 2009. http://www.mpi-forum.org/docs/

Verified Software Laboratory   http://vsl.cis.udel.edu   Contact: siegel@cis.udel.edu