Collective Assertions

Stephen F. Siegel and Timothy K. Zirkel*

Verified Software Laboratory, Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, USA
{siegel,zirkeltk}@udel.edu
http://vsl.cis.udel.edu

Abstract. We introduce the notion of *collective assertions* for message-passing-based parallel programs with distributed memory, such as those written using the Message Passing Interface. 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 sent to a coordinator; once a snapshot has been received from each process, the expression is evaluated on the global state formed by uniting the snapshots. We have extended the Toolkit for Accurate Scientific Software (TASS), a verifier based on symbolic execution and explicit state enumeration, to check that collective assertions hold on all possible executions of a C/MPI program. We give several examples of such programs, show that many properties of them are naturally expressed as collective assertions, and use TASS to verify or refute these.

1 Introduction

Assertions are an important tool for developing reliable sequential programs. They are used to specify correct behavior, reveal faults, and isolate defects. They can be checked at runtime, or verified statically using a number of different techniques. Most importantly, they are easy to use, since they do not require a developer to learn much beyond the expression syntax of the programming language.

In this paper, we focus on distributed memory, message-passing, multiprocess programs, such as those expressed using the Message Passing Interface (MPI, [10]). This includes most programs used in high-performance scientific computing. In this domain, the success of assertions is less clear. This is not surprising: the most difficult and subtle defects in such programs involve the interaction of multiple processes, yet the programming languages and libraries used by developers support only assertions local to a single process.

There has been research to develop *global assertions* in distributed programs, i.e., assertions that may refer to the global state. While these might prove useful in certain cases, it is our view that they are not appropriate for expressing many of the properties that arise naturally in the domain of interest. We now give an example illustrating the problem.

* Supported by the U.S. National Science Foundation grants CCF-0733035 and CCF-0953210, and the University of Delaware Research Foundation.

R. Jhala and D. Schmidt (Eds.): VMCAI 2011, LNCS 6538, pp. 387–402, 2011.
© Springer-Verlag Berlin Heidelberg 2011

```
int first = PID*NX/NPROCS;
int nxl = (PID+1)*NX/NPROCS - first;
int left = (PID+NPROCS-1)%NPROCS;
int right = (PID+1)%NPROCS;
int time = 0;
float u[nxl+2];

/* ... initialize u ... */
for (time=1; time<=nsteps; time++) {
  send(u[1],     left);
  recv(u[nxl+1], right);
  send(u[nxl],   right);
  recv(u[0],     left);
  assert PROC[right].u[0] == u[nxl] &&
   PROC[left].u[PROC[left].nxl+1] == u[1];
  /* ... local update of u ... */
}
```

	Proc 0	Proc 1	Proc 2
1.		send left	
2.			send left
3.		recv right	
4.	send left		
5.			recv right
6.			send right
7.		send right	
8.			recv left
9.			**assert**

Fig. 1. Left: block-distributed 1d diffusion solver with global assertion. Right: an execution fragment in which the assertion, if interpreted literally, fails.

The numerical solution of the "diffusion equation" is a standard example used in many parallel programming texts. In one dimension, the standard solution manipulates a one-dimensional array `u` of length `NX` of real values. The algorithm iterates through discrete time steps. At each time step, the value of each cell in `u` is updated using a formula that is a function of the current value of that cell and those of its left and right neighbors. Assume a cyclic domain, so there is no need for special handling at the boundary. A typical (distributed-memory) parallel version of the algorithm is outlined in C-like pseudocode in Fig. 1 (left). Each of the `NPROCS` processes executes a copy of this code. The processes may still exhibit different behaviors because each has a unique ID number (`PID`) between 0 and `NPROCS` $-$ 1, inclusive.

The original array is block-distributed so that each process "owns" a contiguous slice of length `nxl` of the original array. Assume `NX` $\geq$ `NPROCS`, so that `nxl` ≥ 1 on each process. To update the cell on the left boundary of its slice, a process needs the value of the right-most cell of its left neighbor; a similar issue holds at the right boundary. This problem is managed by having each process maintain a left and right *ghost cell*: the left ghost cell to mirror the value of the left neighbor's right-most (ordinary) cell, and a similar cell for the right neighbor. To incorporate the ghost cells, `u` is given length `nxl` $+ 2$, with the ghost cells in positions 0 and `nxl` $+ 1$. At each time step, the processes update their ghost cells using the communication operations shown. To simplify the discussion, assume asynchronous communication with unbounded channels, i.e., a message sent by a `send` operation can always be buffered, so the routine shown will not deadlock. After the exchange completes, a process updates its cells in the usual way, and proceeds to the next time step.

Suppose we wish to assert the correctness of the ghost cell exchange. From the point of view of one process, this might be stated informally as "after the ghost cell exchange, the value of my right neighbor's left ghost cell agrees with the value of my right-most non-ghost cell," with a similar statement for the left neighbor. To express this requires some notation to refer to variables in other processes. We write `PROC`$[i]$.`x` to denote a variable `x` in the process with PID i, where i is an integer-valued expression. Using this notation, the desired assertion appears in the pseudocode. As with other statements, one assertion appears in each process, though the asserted expressions differ because each process has its own `PID`.

Let us assume these assertions are given the obvious semantics: that when control reaches an assertion, the expression is evaluated in the current global state, and the assertion is violated if it evaluates to *false*. (Leave aside for now the question of how this could be implemented.) It is not hard to see the assertion can fail on many executions, such as the one in Fig. 1 (right). In that execution fragment, process 2 reaches the assertion before its right neighbor, process 0, has received any message. When the assertion is evaluated, the value of `u[0]` on process 0 will not necessarily agree with that of `u[nxl]` on process 2. But there is nothing wrong with this execution—it is the assertion that is broken.

How can we fix the problem? We might alter the semantics by restricting the set of executions on which the assertion should be evaluated. For example, we could declare that the assertion imposes a barrier, and is not evaluated until all processes reach it. There are two problems with this approach. First, it does not capture the intuitive property that the developer has in mind, which applies to all executions, not just the subset which synchronize at the assertion. Hence it leaves a large number of legal executions untested, and as we shall see in Sec. 4 (in the *wildcard_gather* example) there are natural assertions that hold for all executions in which the assertion is treated as a barrier, but fail on other executions. Second, if the assertions are to be used for runtime checks, as they usually are, the additional forced synchronization would be unacceptable from a performance view. Surely one desirable quality of any assertion system should be that the assertions not change the set of possible behaviors of the program. Similar comments apply to other possible restrictions, such as forcing the communication to be synchronous, or forcing all processes to move in lockstep.

A more precise informal statement of the desired property might be "for all $i \geq 1$, the value of my right-most non-ghost cell at time step i after the exchange agrees with the value of my right neighbor's left ghost cell whenever it is in time step i just after the exchange." But it is not clear how this could be made precise using standard assertions, at least without performing complex modifications to the program (e.g., by adding and maintaining history variables).

The solution proposed in this paper is the *collective assertion*. Like a global assertion, a collective assertion may refer to the state of several processes, but unlike a global assertion, the process states may have existed at (very) different times in the execution. A collective assertion is specified by placing an assertion statement in each process. When control in that process reaches the assertion,

a snapshot of the process state is taken. Once every process has reached its assertion, these snapshots are composed into a single global state (ignoring the buffered messages), in which all the asserted expressions are evaluated. Only at this point—just after the last process reaches the assertion—is the assertion determined to have passed or failed. With this semantics, the ghost cell assertion in our example will hold on every execution—and never for vacuous reasons.

A collective assertion is analogous to the *collective operations* of MPI [10, Chap. 5]. These are communication primitives which involve a set of processes, and include barrier, broadcast, gather, reduction, and other operations. Like our collective assertion, an MPI collective operation requires a statement in each process; each process contributes something to the operation upon executing that statement; and the operation as a whole does not complete until every process has made its contribution. Unlike MPI's collectives, however, our assertions never impose any additional synchronization, since we want the program to exhibit the same set of behaviors whether the assertions are turned on or off.

In this paper, we present formal definitions of *collective assertions* and their semantics. We have realized these notions in our Toolkit for Accurate Scientific Software [11]. TASS uses symbolic execution and explicit state enumeration techniques [7, 6] to verify that safety properties of C/MPI programs hold for all possible executions within user-specified bounds. We discuss how we have implemented collective assertion verification in TASS, report on experiments designed to gauge the cost of verifying the assertions, give several additional examples of properties that are naturally expressed using them, and report on our success using TASS on those examples. These include examples where TASS is used to verify the equivalence of two programs. We conclude with a discussion of related work, and various ways the techniques may be extended in the future.

2 Model

For a formal description, we need a model. The model we use is based on [1, Def. 2.31]. For sets X and Y, let $\mathsf{Func}(X, Y)$ denote the set of all functions from X to Y. For $f \in \mathsf{Func}(X, Y)$, $x \in X$, $y \in Y$, $f[x := y]$ denotes the function which is the same as f, except possibly at x, where it returns y. Let X^* denote the set of finite sequences of elements of X and $\mathsf{len}(\sigma)$ the length of a sequence σ.

Let $n \geq 1$ and $\mathsf{Var}_1, \ldots, \mathsf{Var}_n$ be n mutually disjoint sets of *variables*. Let Val be a set of *values*. Let Chan be a set of *channels*, each with a *capacity* $\mathsf{cap}(c) \in \{1, 2, \ldots\}$. Let $V \subseteq \mathsf{Var} \stackrel{\text{def}}{=} \bigcup_i \mathsf{Var}_i$, and $\mathsf{Eval}(V) \stackrel{\text{def}}{=} \mathsf{Func}(V, \mathsf{Val})$. The set of *communication actions* is $\mathsf{Comm} \stackrel{\text{def}}{=} \{c!e, c?x \mid c \in \mathsf{Chan}, x \in \mathsf{Var}, e \in \mathsf{Expr}\}$, where Expr denotes the set of expressions over Var. The exact syntax of expressions is not important, but we assume all expressions are side-effect free.

Definition 1. *A* **program graph** *over* (V, Chan) *is a tuple*

$$\mathsf{PG} = (\mathsf{Loc}, \mathsf{Act}, \mathsf{Effect}, \hookrightarrow, \mathsf{Loc}_0, g_0), \tag{1}$$

where $\mathsf{Cond}(V)$ *denotes the set of boolean-valued expressions over* V *and*

1. Loc *is a set of* locations *and* Act *is a set of* actions,
2. $\mathsf{Effect}\colon \mathsf{Act} \times \mathsf{Eval}(V) \to \mathsf{Eval}(V)$ *is the* effect function,
3. $\hookrightarrow \subseteq \mathsf{Loc} \times \mathsf{Cond}(V) \times (\mathsf{Act} \cup \mathsf{Comm}) \times \mathsf{Loc}$ *is the* conditional transition relation,
4. $\mathsf{Loc}_0 \subseteq \mathsf{Loc}$ *is a set of* initial locations, $g_0 \in \mathsf{Cond}(V)$ *is the* initial condition.

We write $l \overset{g:\alpha}{\hookrightarrow} l'$ *to denote that* $\langle l, g, \alpha, l' \rangle \in \hookrightarrow$.

Definition 2. *A* **channel system** CS *over* $(\mathsf{Var}, \mathsf{Chan})$ *is a tuple* $[\mathsf{PG}_1 | \cdots | \mathsf{PG}_n]$, *where for each* i, $\mathsf{PG}_i = (\mathsf{Loc}_i, \mathsf{Act}_i, \mathsf{Effect}_i, \hookrightarrow_i, \mathsf{Loc}_{i,0}, g_{i,0})$ *is a program graph over* $(\mathsf{Var}_i, \mathsf{Chan})$.

Semantics. We assume an expression semantics is given; this specifies how to extend any $\eta \in \mathsf{Eval}(\mathsf{Var})$ to some $\tilde{\eta} \in \mathsf{Eval}(\mathsf{Expr})$. The elements of $\mathsf{Eval}(\mathsf{Chan}) \overset{\text{def}}{=} \mathsf{Func}(\mathsf{Chan}, \mathsf{Val}^*)$ are *channel evaluations*. Let ξ_0 denote the channel evaluation that maps every channel to the empty sequence. The sets of *(global) states* and *initial states* are defined by

$$\mathsf{State} = \mathsf{Loc}_1 \times \cdots \times \mathsf{Loc}_n \times \mathsf{Eval}(\mathsf{Var}) \times \mathsf{Eval}(\mathsf{Chan})$$
$$I = \{\langle l_1, \ldots, l_n, \eta, \xi_0 \rangle \in \mathsf{State} \mid \forall 0 < i \leq n. (l_i \in \mathsf{Loc}_{0,i} \wedge \eta \models g_{0,i})\}.$$

A *transition* is a triple $\langle s, \alpha, s' \rangle$, where $s = \langle l_1, \ldots, l_i, \ldots, l_n, \eta, \xi \rangle, s' \in \mathsf{State}$, $\alpha \in \mathsf{Act} \overset{\text{def}}{=} \bigcup_i \mathsf{Act}_i \cup \{\tau\}$, and one of the following holds:

1. for some i, $\alpha \in \mathsf{Act}_i$, $l_i \overset{g:\alpha}{\hookrightarrow} l_i'$, $\eta \models g$, and $s' = \langle l_1, \ldots, l_i', \ldots, l_n, \mathsf{Effect}(\alpha, \eta), \xi \rangle$,
2. $\alpha = \tau$ and for some i, $l_i \overset{g:c?x}{\hookrightarrow} l_i'$, $\mathsf{len}(\xi(c)) = k > 0$, $\xi(c) = v_1 \cdots v_k$, $\eta \models g$, and $s' = \langle l_1, \ldots, l_i', \ldots, l_n, \eta[x := v_1], \xi[c := v_2 \cdots v_k] \rangle$, or
3. $\alpha = \tau$ and for some i, $l_i \overset{g:c!e}{\hookrightarrow} l_i'$, $\mathsf{len}(\xi(c)) = k < \mathsf{cap}(c)$, $\xi(c) = v_1 \cdots v_k$, $\eta \models g$, and $s' = \langle l_1, \ldots, l_i', \ldots, l_n, \eta, \xi[c := v_1 v_2 \cdots v_k \tilde{\eta}(e)] \rangle$.

We write $s \overset{\alpha}{\to} s'$ to denote that $\langle s, \alpha, s' \rangle$ is a transition. We say s is *terminal* if there is no transition departing from s, i.e., none of the form $s \overset{\alpha}{\to} s'$.

An *execution fragment* is a (finite or infinite) sequence $\rho = s_0 \overset{\alpha_0}{\to} s_1 \overset{\alpha_1}{\to} \cdots$. We say ρ is *initial* if $s_0 \in I$. The *length* of ρ is the number of transitions.

3 Collective Assertions

Definition 3. *Let* CS *be a channel system and* $\mathsf{Loc} \overset{def}{=} \bigcup_{1 \leq i \leq n} \mathsf{Loc}_i$. *A* collective assertion σ *is a function from a subset* $\mathsf{dom}(\sigma)$ *of* Loc *to* $\mathsf{Cond}(\mathsf{Var})$.

Hence σ involves some set of locations, and to each such location it associates a global expression. Note the locations may be in several processes, and there may be several locations in the same process. Although it is not required by the definition, we will see that if σ is to have any chance of holding, $\mathsf{dom}(\sigma)$ must have at least one location in each process.

Let Σ be a set of collective assertions such that, for any location l, $l \in \mathsf{dom}(\sigma)$ for at most one $\sigma \in \Sigma$. Let $\mathsf{ALoc} = \{l \in \mathsf{Loc} \mid \exists \sigma \in \Sigma. l \in \mathsf{dom}(\sigma)\}$. The

elements of ALoc are the *assertion locations*. Given $l \in \mathsf{ALoc}$, let $\mathsf{assert}(l)$ denote the unique $\sigma \in \Sigma$ such that $l \in \mathsf{dom}(\sigma)$.

Let $\rho = s_0 \overset{\alpha_0}{\to} s_1 \overset{\alpha_1}{\to} \cdots \overset{\alpha_{m-1}}{\to} s_m$ be a finite execution fragment. We now define what it means for Σ to hold on ρ. The definition requires that every process encounter the same assertions in the same order. (This is also the case with MPI's collective operations.) Write $s_j = \langle l_{j,1}, \ldots, l_{j,n}, \eta_j, \xi_j \rangle$ $(0 \le j \le m)$.

We define a sequence $\Lambda_i^j \in (\mathsf{Eval}(\mathsf{Var}_i) \times \mathsf{Loc}_i)^*$ for each $1 \le i \le n$ and $1 \le j \le m$. This sequence contains the accumulated snapshots from process i after reaching state s_j in ρ. The definition is by induction on j. For $j = 0$,

$$\Lambda_i^0(s) = \begin{cases} \langle \eta_0|_{\mathsf{Var}_i}, l_{0,i} \rangle & \text{if } l_{0,i} \in \mathsf{ALoc} \\ \text{the empty sequence} & \text{otherwise} \end{cases} \qquad (1 \le i \le n). \tag{2}$$

Hence $\Lambda_i^0(s)$ is a sequence of length 1 or 0, depending on whether $l_{0,i}$ is an assertion location. Note $\eta_0|_{\mathsf{Var}_i}$ denotes the restriction of η_0 to Var_i, and captures the values of the local variables for PG_i.

Now assume $1 \le j \le m$ and we have defined Λ_i^{j-1} for all i. The transition α_{j-1} lies in some process PG_i. If $l_{j,i} \in \mathsf{ALoc}$ then $\Lambda_i^j = \Lambda_i^{j-1}.\langle \eta_j|_{\mathsf{Var}_i}, l_{j,i} \rangle$. Otherwise, $\Lambda_i^j = \Lambda_i^{j-1}$.

Let $\Lambda_i = \Lambda_i^m$; this is the final sequence of snapshots at the end of ρ.

Note that given $\chi_i \in \mathsf{Eval}(\mathsf{Var}_i)$ for each $1 \le i \le n$, we can form their union $\chi = \bigcup_i \chi_i \in \mathsf{Eval}(\mathsf{Var})$: for any $v \in \mathsf{Var}$, $\chi(v) = \chi_i(v)$, where $v \in \mathsf{Var}_i$. This is the "composite" global state formed from the local snapshots.

Let $k \ge 1$. If $\mathsf{len}(\Lambda_i) \ge k$, write $\Lambda_i[k] = \langle \eta_i^k, l_i^k \rangle$ for the k^{th} element of Λ_i. We say that Σ *holds at the* k^{th} *occurrence in* ρ if

1. $\forall 1 \le i, j \le n.((\mathsf{len}(\Lambda_i) \ge k \wedge \mathsf{len}(\Lambda_j) \ge k) \Rightarrow \mathsf{assert}(l_i^k) = \mathsf{assert}(l_j^k))$, and
2. $(\forall 1 \le i \le n.\mathsf{len}(\Lambda_i) \ge k) \Rightarrow \bigcup_{1 \le i \le n} \eta_i^k \models \bigwedge_{i=1}^n \sigma(l_i^k)$.

The first constraint says all processes which have reached their k^{th} assertion location agree on the k^{th} assertion to be checked. The second says that if all processes have reached the k^{th} assertion, the asserted expressions hold at the composite state.

Definition 4. *Let Σ, ρ, and the Λ_i be defined as above. The collective assertion set Σ* holds on *ρ if (i) for all $k \ge 1$, Σ holds at the k^{th} occurrence in ρ, and (ii) if s_m is a terminal state then all Λ_i have the same length. We say* CS satisfies *Σ if Σ holds on every finite initial execution fragment of* CS.

Note there are three ways in which Σ can fail to hold for ρ: (1) the collective assertions are encountered in different orders by two processes, (2) the final state is terminal, yet there remain assertions entered by some processes but not by others, or (3) all processes have entered an assertion but the asserted expression fails to hold in the composite state.

```
1  procedure check(⟨s, Λ1, ..., Λn⟩ ∈ AState \ {sV}): AState is
2    if ∀j.len(Λj) = 0 then return ⟨s, Λ1, ..., Λn⟩;
3    choose j such that len(Λj) > 0 and let σ = assert(peek(Λj));
4    if ∃i.(len(Λi) > 0 ∧ assert(peek(Λi)) ≠ σ) then return sV; /*out of order*/
5    if ∃j.len(Λj) = 0 then return ⟨s, Λ1, ..., Λn⟩;
6    foreach j ∈ {1...., n} do Λ'j, ⟨ηj, lj⟩ ← dequeue(Λj);
7    if ⋃1≤j≤n ηj ⊭ ⋀_{j=1}^{n} σ(lj) then return sV;  /* assertion failure */
8    return ⟨s, Λ'1, ..., Λ'n⟩;
9  procedure next(⟨s, Λ1, ..., Λn⟩ ∈ AState \ {sV}, s -α-> s'): AState is
10   let i be the index of the process to which α belongs;
11   let ⟨l1, ..., ln, η, ξ⟩ = s;
12   if li ∈ ALoc then return check⟨s', Λ1, ..., enqueue(Λi, ⟨η|Vari, li⟩), ..., Λn⟩;
13   return ⟨s', Λ1, ..., Λn⟩;
```

Fig. 2. Next state function for ATS. Function **dequeue** returns both the left-most element and the sequence obtained by removing that element; **peek** just returns the leftmost element; **enqueue** returns the sequence obtained by appending the given element on the right. Also, $\mathsf{assert}(\langle \eta, l \rangle) = \mathsf{assert}(l)$.

3.1 The Extended Transition System

We now define an extended transition system ATS which can be used to determine whether a channel system satisfies a collective assertion set Σ. The approach mirrors the definition above, with one adjustment. Instead of waiting until termination to check the assertions, an assertion is checked as soon as there is at least one snapshot queued from each process. From a practical point of view, this allows a violation to be reported earlier—as soon as the last process reaches the assertion. Also, as soon as an assertion is checked, the snapshots can be dequeued, reducing the memory required to store the snapshots.

The states in ATS model the snapshot queues and include a terminal *violation state* s_V:

$$\mathsf{AState} \stackrel{\text{def}}{=} \big(\mathsf{State} \times (\mathsf{Eval}(\mathsf{Var}_1) \times \mathsf{Loc}_1)^* \times \cdots \times (\mathsf{Eval}(\mathsf{Var}_n) \times \mathsf{Loc}_n)^*\big) \cup \{s_V\}.$$

The initial states are all states of the form $\mathsf{check}\langle s, \Lambda_1^0(s), \ldots, \Lambda_n^0(s)\rangle$, where $s \in I$ and $\Lambda_i^0(s)$ is as in (2). The function check, defined in Fig. 2, first checks that no out-of-order violation has occurred. Then, if there is at least one entry in each queue, it dequeues one snapshot from each and checks the assertion at the resulting composite state. It returns the violation state if any check fails.

Given any $\tilde{s} = \langle s, \ldots \rangle \in \mathsf{AState}$ and any transition $s \xrightarrow{\alpha} s'$, there is a transition $\tilde{s} \xrightarrow{\alpha} \mathsf{next}(\tilde{s}, s \xrightarrow{\alpha} s')$ in ATS, where the function next is defined in Fig. 2.

Define a predicate Φ on AState as follows: for $\tilde{s} \in \mathsf{AState}$,

$$\Phi(\tilde{s}) \;\Leftrightarrow\; (\tilde{s} \text{ is terminal} \Rightarrow (\tilde{s} \neq s_V \wedge \forall i.\mathsf{len}(\Lambda_i) = 0)), \tag{3}$$

where $\tilde{s} = \langle s, \Lambda_1, \ldots, \Lambda_n \rangle$ if $\tilde{s} \in \mathsf{AState} \setminus \{s_V\}$. We claim

Theorem 1. CS *satisfies* Σ *if and only if* Φ *holds at every state reachable from an initial state in* ATS.

Proof. Let A be the set of all finite execution fragments ρ of CS such that Σ does not hold on ρ but Σ does hold on any proper prefix of ρ. Let B be the set of all finite execution fragments in ATS that terminate in a state at which Φ does not hold. We claim there is a surjective map from A to B. This will complete the proof, since it implies $A = \emptyset \Leftrightarrow B = \emptyset$.

Given $\rho = s_0 \overset{\alpha_0}{\to} \cdots \overset{\alpha_{m-1}}{\to} s_m \in A$, let $\tilde{\rho}$ be the execution fragment in ATS $\tilde{s}_0 \overset{\alpha_0}{\to} \cdots \overset{\alpha_{m-1}}{\to} \tilde{s}_m$, where $\tilde{s}_0$ is the initial state corresponding to s_0 and $\tilde{s}_{j+1} = \mathsf{next}(\tilde{s}_j, s_j \overset{\alpha_j}{\to} s_{j+1})$ for $0 \le j \le m-1$. There are two possibilities: either (i) there exists k such that Σ does not hold at the k^{th} occurrence in ρ, or (ii) Σ holds at the k^{th} occurrence in ρ for all k, but s_m is a terminal state and $\exists 1 \le i, j \le n.\mathsf{len}(\Lambda_i) > \mathsf{len}(\Lambda_j)$. In the first case, $\tilde{s}_m = s_V$, so $\tilde{\rho} \in B$. In the second case, $\tilde{s}_m = \langle s_m, \Lambda'_1, \ldots, \Lambda'_n \rangle$ is terminal and $\Lambda'_i > 0$, since, whenever elements are removed from the queues (line 6 of Fig. 2), one element is removed from each queue. Hence $\tilde{\rho} \in B$ in this case as well. Conversely, given any $\tilde{\rho} \in B$, $\tilde{\rho}$ ends at either s_V or a terminal state with a non-empty snapshot queue. In the first case it must arise in the above way from a $\rho \in A$ satisfying (i); in the second from a ρ satisfying (ii). □

Hence the validity of a set of collective assertions is an invariant property for the extended transition system. It can therefore be verified using standard techniques, such as depth-first search of the reachable states.

4 Implementation and Experiments

In this section we discuss how we have implemented collective assertion verification in TASS, describe the results of some preliminary experiments carried out with that implementation, and give several more examples of properties that are naturally expressed using collective assertions.

4.1 Overview of TASS

TASS takes as input a C/MPI program and an integer $n \ge 1$, and verifies a number of safety properties of an n-process instance of the program. Absence of deadlock, buffer overflows, memory leaks, and (ordinary) assertion violations are some of the properties checked. Currently only standard-mode blocking MPI functions are supported, but this is being expanded to a much larger subset of MPI. Not every C language feature is supported at this time, but many of the most commonly-used features are, including multi-dimensional arrays, structs, pointers and pointer arithmetic, and dynamically allocated data.

TASS uses symbolic execution to reason about all possible inputs to the program. The program inputs are represented as symbolic constants; operations result in symbolic expressions in those symbolic constants. An additional boolean-valued path condition variable is used to keep track of the guards that had to

evaluate to *true* in order for the current path to have been executed. Automated theorem proving techniques are used to determine if the path condition becomes unsatisfiable, whether an array index is out of bounds, and so on. TASS has its own internal support for simplifying symbolic expressions, placing them into a canonical formal, and dispatching some of the proof obligations; for those it cannot dispatch itself, it uses CVC3 [2].

TASS allows the user to specify an arbitrary initial condition. In practice this is used to place bounds on certain inputs (array sizes, loop iterations) in order to make the number of states tractable. Without such bounds it is possible that TASS will never return because the state space is infinite. In general, TASS is applied to configurations that are smaller than those the program would typically encounter in use, but within the specified bounds it performs an exhaustive exploration of all possible program behaviors and inputs.

Partial order reduction [5] is a family of techniques for reducing the number of states that need to be explored in order to verify a class of properties. TASS uses the MPI-specific POR technique, the *urgent algorithm* [12]. Given a state s, the urgent algorithm can conclude that it is safe to explore only those transitions enabled in a single process, rather than all enabled transitions. We say such a process is "urgent" at s. If more than one process is urgent, a heuristic is used to select one of them. (By default, TASS chooses the first urgent process it finds.) Any such heuristic is sound, i.e., if a violation exists within the specified bounds, one will be found. But the number of states explored can differ greatly with the heuristic, in ways that are usually difficult to predict.

TASS can also use *comparative symbolic execution* [13] to verify that two programs are functionally equivalent (i.e., "input-output" equivalent). The basic idea is to construct a model in which the two programs run sequentially—one after the other—and at the end the outputs are compared. The state space of this model is then exhaustively explored. This technique is particularly useful in computational science, for comparing a complex parallel version of a program (the "implementation") to a simple, trusted sequential version ("specification").

4.2 Collective Assertion Specification and Verification in TASS

The assertions are specified as pragmas inserted into the code at the desired locations. Fig. 3 gives examples of the pragmas used in the 1d-diffusion code. We have already discussed the '`PROC`' notation for referring to a variable in another process. We have also added support for existential and universal quantifiers.

An identifier (`GHOSTS` or `COMPARE`) gives a name to the collective assertion of which the pragma is a part. One collective assertion $\sigma = \sigma(\texttt{id})$ is created for each identifier `id`, and σ consists of all location-expression pairs specified by a pragma with identifier `id`. The location is the point in the source code immediately following the pragma.

Collective assertions can also be used when comparing two programs, particularly to specify the expected correspondence between variables at various control points. The semantics are easily defined: if one program has n processes, and

```
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];
}
```

(a) main loop in parallel version (implementation)

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

(b) main loop in sequential version (specification)

Fig. 3. TASS collective assertion pragmas in 1d-diffusion code

the other m, we can consider the two programs together to be a single program with $n + m$ processes running concurrently. All of the definitions of Sections 2 and 3 apply; the urgent POR scheme will automatically avoid exploring the unnecessary interleavings resulting from this combination. An advantage of this approach over the standard comparative method is that if a discrepancy is detected, it can be reported immediately, rather than waiting until both programs have terminated. It is also helpful in isolating the source of a fault.

The keyword `joint` can be used in place of `collective` to indicate a collective assertion that is to be used when comparing two programs. These pragmas will be ignored when verifying either program individually. Typically, the asserted expression in the specification will be vacuous ("`true`") so the pragma in the specification is used only to label a location. A reasonable policy is to forbid references to implementations within the specification, while implementations can and should refer to the specification.

A joint assertion is used in Fig. 3. It asserts the values of the non-ghost cells held by this process agree with the corresponding values in the specification. This is checked at the end of each time step, rather than only at termination.

TASS verifies collective assertions by performing a depth-first search of the transition system ATS defined in Section 3. To implement this, we added local state queues to the state, and a special handler is executed whenever a process reaches a collective location. One implementation detail concerns the way the local state snapshots are stored in each state. In TASS, these process states are immutable objects which are created using the Flyweight design pattern. Hence the queues contain only references to the process states, each of which was created at an earlier point in the execution. If the entire process state data were instead duplicated in each state, the memory required to store the states would be prohibitively large.

nprocs	Number of States Normal	CQMin
2	4886	3496
3	10537	7492
4	18624	13188
5	29363	20728
6	42970	30256
7	59661	41916
8	79652	55852
9	103159	72208
10	130398	91128
11	161585	112756
12	196936	137236
13	236667	164712
14	280994	195328
15	330133	229228

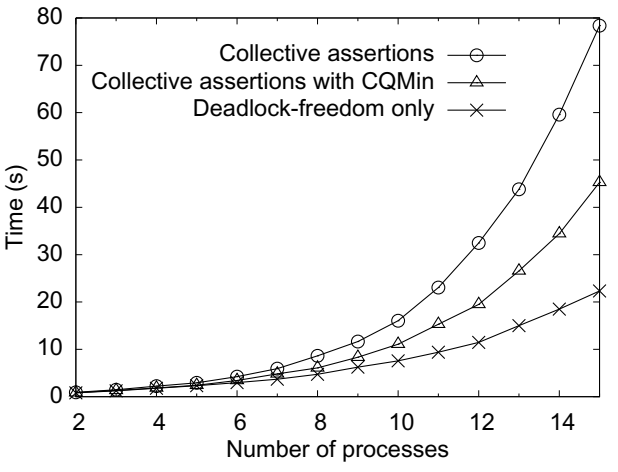

Fig. 4. Verification of diffusion1d, comparing parallel and sequential versions. `NX` is bounded by 3*`nprocs`, `NSTEPS` by 2. For each value of `nprocs`, three experiments were run: (a) verifying only absence of potential deadlock, (b) verifying that and the collective assertions, and (c) same as (b) but using the queue-minimizing heuristic (CQMin).

4.3 Scaling Experiment

After successfully verifying the two collective assertions in small configurations of the 1d-diffusion code, we decided to scale this example to gauge the cost of verifying the assertions. We scaled the number of processes n from 2 to 15 and for each n conducted three experiments. All three involve using TASS to compare the sequential and parallel versions, as well as verify absence of deadlocks.

In the first experiment, used as a baseline, we turned the assertion checking off (so only the deadlock property was checked). In the second experiment, we checked the collective assertion and the deadlock property using the default heuristic for the urgent POR scheme. In the third experiment, we checked the assertion and the deadlock property, but used a novel *collective queue minimization* heuristic. This heuristic selects, among all "urgent" processes, a process with the shortest local state queue. The idea is to try to keep the processes as "close together" as possible. In some cases, such as this diffusion example, this heuristic essentially imposes a barrier at the collective assertion point. In other cases, it will not impose such a barrier, and it would not be safe to do so (see example *wildcard_gather* below). As described in Section 4.1, in all cases the heuristic is safe, i.e., a violation will be found if one exists.

In each case, we recorded the verification time and the number of states explored. All were run on a 2.8GHz quad-core Intel i7 iMac with 16GB RAM.

The results are shown in Fig. 4. Note that there is a substantial cost to collective assertion checking in terms of time: up to 4× at the worst case. Nevertheless, even the maximum time of 80 seconds is not unreasonable. There is no difference in the number of states between the baseline and the first collective assertion

experiment because (1) both use the same POR algorithm and (2) in this example, the state space is a tree, so there is never a case where the introduction of the snapshots can cause a pair of states that matched in the first case to fail to match in the second. The queue minimization heuristic led to a reduction in the number of states and cut the runtime almost in half.

4.4 Further Examples

We examined the examples distributed with TASS for other opportunities where collective assertions could express useful properties. We have found such opportunities in virtually every case and report on a representative sample here. We then used TASS to verify these assertions.

```
for (i=0; i<N; i++) {
  x = x + a[i];
#pragma TASS joint assert C true;
}
m = x/N;
#pragma TASS joint assert M true;
```

```
for (i=1; i<=N; i++) {
  x = ((i-1)*x + a[i-1])/i;
#pragma TASS joint assert C x*i==spec.x;
}
m = x;
#pragma TASS joint assert M m == spec.m;
```

Fig. 5. Two ways to compute the mean of an array of floating-point numbers

Mean. This is an example where the assertion is more complicated that just "$x = y$". Fig. 5 shows two different ways to compute the mean of an array of N floating-point numbers. The first corresponds exactly to the standard definition: it sums the elements and divides the result by N. The second computes a "running mean": after the i^{th} iteration the value of `x` should be the mean of the first i numbers. The assertion relating these is `x*i==spec.x`, which is checked at the end of each loop iteration. A joint assertion at the end claims the final results should agree, which indeed holds. (This relies on the fact that TASS interprets arithmetic to take place over the mathematical reals instead of floating-point numbers.) When verified with N bounded above by 10, TASS explored 132 states and took 0.3 seconds.

Wildcard_gather. This example demonstrates that it is not always safe to verify a collective assertion by imposing a barrier at the assertion. The program of Fig. 6 is composed of a root process (rank 0) and $n-1$ worker processes. Each worker sends one number to the root, which in this case happens to be the worker's `PID`. The root receives these in whatever order they arrive (using a wildcard receive) and inserts the number into an array at the position which is the rank of the sending process. The routine is called twice in a row.

The collective assertion is invoked once by each process in each call. It asserts that the value stored in the root's array agrees with the value sent. The code is incorrect: a race condition may occur if one worker sends, then proceeds to the second call and sends again; meanwhile, the root is still in the first invocation but receives this second message at the wildcard receive. This results in the entry

```
int myrank, nprocs; double *a, x;

void root() {
  MPI_Status status; int i;
  a = (double*)malloc(nprocs*sizeof(double));
  a[0] = 0.0;
  for (i=1; i<nprocs; i++) {
    MPI_Recv(&x, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    a[status.MPI_SOURCE] = x;
  }
#pragma TASS collective assert C true;
  free(a);
}

void worker() {
  x = PID;
  MPI_Send(&x, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
#pragma TASS collective assert C x==PROC[0].a[myrank];
}

void main() {
  int argc; char **argv; int i;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  for (i=0; i<2; i++) if (PID == 0) root(); else worker();
  MPI_Finalize();
}
```

Fig. 6. *Wildcard_gather*: a gather routine that uses a wildcard at root. One number from each non-root process is sent to root, which inserts it into an array. The routine has a race condition revealed by a violation to collective assertion `C`.

for that process getting set again (in this case, to the same value), while the message from some other process remains unreceived in the first invocation. The array entry for that process will still contain the initial undefined value when the root reaches the collective assertion point and the snapshot is taken, leading to a violation when the assertion expression is evaluated.

If a barrier is placed at the collective assertion point, the race condition disappears: no worker can proceed into the second iteration until the root has received all $n-1$ messages. Hence a system that only checked executions that synchronize at assertion points would fail to detect the race condition.

For 10 processes, the error is detected by TASS after exploring 1011 states. The execution time, which includes the time to write the violating trace to disc, was 0.5 seconds.

Laplace2d. This is the most complex example we studied. It is a numerical solution to the 2d-Laplace equation. The algorithm works on a row-distributed 2d-grid with fixed boundary values. The update formula depends on a cell's

upper, lower, right, and left neighbors. Rather than iterate for a fixed number of time steps, as in the diffusion case, the program iterates until a certain convergence criterion is reached (or a bound on the maximum number of iterations is reached, whichever occurs first). The convergence criterion is that the L_2 norm between two consecutive approximate solution falls below a given threshold ϵ. Again, we used TASS to compare a sequential and parallel version. In the parallel version, each process computes the contribution to the error resulting from the rows it owns. These local errors are summed at the end of each iteration to obtain the global error, which is then returned to every process. (This is implemented using a single `MPI_Allgather` collective operation). The joint assertion we formulated claims that at each iteration, the global error in each process in the parallel program agrees with the global error in the sequential program on the corresponding iteration. We successfully verified this assertion for various small configurations. For 6 processes, the dimensions of the grid bounded above by 4×12, and the number of iterations bounded by 3 (inclusive), TASS completed in 18.0 seconds, exploring 49,419 states.

5 Related Work

Composing local snapshots to form a global state of a distributed system is not a new idea. Chandy and Lamport used this notion in their algorithm for computing a globally consistent state [4]. This algorithm has been extended in many directions; see for example [9] and the references cited there. The goal of this line of work is to construct a state that is "close to" an actual global state occurring in the execution, where "close to" preserves some specified class of predicates, such as deadlock or termination. Simmons shows how these notions can be extended to efficiently check a global assertion in a single process by using earlier snapshots, as long as no actions could have taken place in the other processes to impact the evaluation of the global expression [14, 15]. Much of the hard work is in figuring out when a process should take its snapshot in order to be consistent with other processes and not affect the property of interest.

Our focus is different: we are not interested in finding a globally consistent set of snapshots or any notion of "close to." Instead, the user decides exactly where the snapshots are to be taken in each process by explicitly placing collective assertion statements at appropriate points. In fact, the selection of these locations is an important part of the specification of the desired behavior. We thus do not have to worry about the many difficult issues that arise in the global assertion approach. (Nor are we concerned with capturing a consistent view of the channel states, a major source of complexity in the earlier work.) Also, for an analysis tool such as TASS, the mechanics of gathering the snapshots is trivial, since TASS itself is not a distributed program.

The basic approach underlying TASS, which combines symbolic execution with a search of the reachable states of a parallel program, was introduced in [6]. There are many other approaches to debugging and verifying parallel programs that may also apply to the kinds of problems discussed here. Kovacs et al. have

developed a debugger that can check linear temporal logic formulas on a particular trace as the user guides the execution [8]. The ISP verifier is a dynamic model checker that uses a modified runtime system to explore all relevant behaviors of an MPI program to verify certain safety properties [17]. Both of these approaches, however, only operate on a given concrete input, in contrast to the symbolic execution approach taken by TASS.

There are many other methods for verifying functional equivalence of *sequential* programs; see for example [16] for an approach that can handle complex loop transformations. KLEE is another symbolic execution tool that has been used to check equivalence of sequential programs [3].

6 Conclusion

Collective assertions appear to be a natural way to express many correctness properties of distributed multiprocess programs. We have explored how to verify these assertions in small instances of the programs using TASS, but many questions remain. Can runtime checking of collective assertions be implemented effectively at full scale? To do so would require careful reasoning to determine what part of a process's state is required to evaluate an expression, as sending an entire process snapshot is not feasible for realistic programs.

When multiple collective assertions occur in a program, our semantics requires that each process encounter these assertions in the same order. This may not be the most appropriate choice for all problems. For example, the user might want to specify that two collective assertions are completely independent of each other, and can be encountered in different orders by two processes. A flexible approach would allow the users to specify distinct "communication universes" for this end. Like an MPI *communicator*, each such universe would have associated to it some subset of processes. Each collective assertion would then specify the universe to which it belongs. The assertions within one universe would be required to occur in the same order for every process in that universe's process set. For assertions in two different universes, there would be no such constraint.

We have extended C's expression language to include references to other processes and first-order quantifiers. What other operations would be useful? A sum operator is an obvious candidate, and undoubtedly many array operations could help specify important properties.

The semantics for collective assertions for infinite executions is weak. Would a more useful definition require every entry in a local state queue to eventually be consumed? It would be interesting to see if collective assertions can be applied usefully to reactive programs.

Finally, the general idea of making collective analogs of sequential constructs could have other applications. We have recently begun using TASS to explore "collective loop invariants." In many cases, these enable TASS to verify properties of a parallel program with unbounded loops.

References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
2. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
3. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (2008)
4. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. 3(1), 63–75 (1985)
5. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
6. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
7. King, J.C.: Symbolic execution and program testing. Comm. ACM 19(7), 385–394 (1976)
8. Kovács, J., Kusper, G., Lovas, R., Schreiner, W.: Integrating temporal assertions into a parallel debugger. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 113–120. Springer, Heidelberg (2002)
9. Kshemkalyani, A.D.: Fast and message-efficient global snapshot algorithms for large-scale distributed systems. IEEE Transactions on Parallel and Distributed Systems 21, 1281–1289 (2010)
10. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, version 2.2, September 4 (2009), `http://www.mpi-forum.org/docs/`
11. Siegel, S.F., et al.: The Toolkit for Accurate Scientific Software web page (2010), `http://vsl.cis.udel.edu/tass`
12. Siegel, S.F.: Efficient verification of halting properties for MPI programs with wildcard receives. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 413–429. Springer, Heidelberg (2005)
13. 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), Article 10, 1–34 (2008)
14. Simmons, S., Kearns, P.: A causal assert statement for distributed systems. In: Hamza, M.H. (ed.) Parallel and Distributed Computing and Systems, pp. 495–498. IASTED/ACTA Press (1995)
15. Simmons, S.J.: Causal distributed assert statements. Ph.D. thesis. The College of William and Mary, director-Kearns, Phil (1999)
16. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 599–613. Springer, Heidelberg (2009)
17. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 261–270. ACM, New York (2009)