Towards Deductive Verification of Message-Passing Parallel Programs

Ziqing Luo

Department of Computer and Information Sciences
University of Delaware
Newark, Delaware, USA
ziqing@udel.edu

Stephen. F. Siegel

Department of Computer and Information Sciences
University of Delaware
Newark, Delaware, USA
siegel@udel.edu

***Abstract*—Program verification techniques based on *deductive reasoning* can provide a very high level of assurance of correctness. These techniques are capable of proving correctness without placing artificial bounds on program parameters or on the sizes of inputs. While there are a number of mature frameworks for deductive verification of sequential programs, there is much less for parallel programs, and very little for message-passing. We propose a method for the deductive verification of message-passing programs that involves transforming the program into an annotated sequential program that can be verified with off-the-shelf deductive tools, such as Frama-C. The method can prove user-specified correctness properties without any bounds on the number of processes or other parameters. We illustrate this method on a toy example, and analyze its strengths and weaknesses.**

I. Introduction

Message-passing parallel programs have long been a mainstay of high performance computing. This situation is expected to continue for the foreseeable future; for example, a recent survey in the U.S. Exascale Computing Project found that a large majority of the software development efforts in that project are using or plan to use MPI [1]. Verification tools for message-passing parallelism are therefore of central importance to HPC.

A number of verification tools targeting MPI programs have been developed. These include model-checking-based tools such as CIVL [2], ISP [3], DAMPI [4], and MOPPER [5]. Yet these tools require (often small) bounds on the number of processes and on other parameters, such as the sizes of input data structures. Of course, if a bug is found within these bounds, the tool has served a useful purpose. But if the goal is to verify correctness of a program, such tools can only provide a limited kind of evidence.

Deductive verification approaches, in contrast, are able to prove properties of programs without such bounds. These approaches typically require more manual effort and skill from the user, but the field has matured considerably since its origins in the work of Floyd and Hoare. Today there are a number of robust frameworks for deductive verification — at least for sequential programs. The Frama-C static analysis platform [6], [7], for example, together with the WP plug-in [8], [9], has been used to prove correctness of a wide range of C programs. The Why3 verification platform [10], [11], which is often used as a back-end for Frama-C, has also been very successful. Many of the applications of these tools have been to problems of interest in HPC. These include the verification of a large collection of algorithms on basic data structures [12] and the programs in the Toccata "Gallery of Verified Programs" [13]. The gallery includes many programs with precise floating-point accuracy requirements, such as Kahan's algorithm for computing a discriminant [14], [15], verification of convergence and other numerical properties in a one-dimensional wave equation solver [16], and the verification of Strassen's matrix multiplication algorithm for arbitrary matrix sizes [17].

There is also a growing body of theoretical and practical work extending deductive approaches to concurrent programs. While most of the developments in this area apply to shared-variable (multi-threaded) concurrency, in recent years there have been investigations in deductive approaches for MPI programs. A notable example is the ParTypes tool, which uses "session types" and deductive reasoning to verify communication properties of MPI programs [18]–[20]. The user of ParTypes specifies a communication protocol in a special language, and the tool verifies that the program complies with the protocol. If the verification succeeds, the program must be communication-correct, and in particular, deadlock-free. While there are restrictions on the program and its use of MPI (e.g., it cannot use wildcard receives), the verification process does not require any bounds on the number of processes or other parameters.

In this paper, we propose a new method for constructing mechanized correctness proofs of message-passing programs. The basic idea is to view the message-passing program as a sequential program for an arbitrary

process—reflecting the "SPMD" style in which most MPI programs are written. Annotations and ghost state are added to this program in order to specify and prove a global invariant. The send and receive functions are replaced by stubs that permit arbitrary changes in state preserving the invariant. We call the resulting sequential program an *invariant-preserving projection* (IPP) of the original program, and show that the correctness of the projection implies the correctness of the original program. Deductive verification tools for sequential programs can then be applied to prove the projection.

This work is very preliminary, but we believe it will be of interest to the HPC correctness community because the method exhibits three nice qualities:

1) it can use existing verification platforms for sequential programs—taking advantage of the mature language processing, analysis, and verification condition generation capabilities they provide;
2) it can be applied to C code, and not just to modeling languages or process algebras, and
3) it can be used to verify functional correctness properties expressed in a rich contract language, in addition to deadlock-freedom and other generic safety properties.

The basic approach is described in Section II. Section III describes an application to a simple example, using Frama-C, WP, and Why3. Section IV deals with deadlock-freedom and termination. Additional related work is discussed in Section V. In Section VI, we assess the strengths and weaknesses of the method, and the challenges that must be overcome to apply it more broadly.

II. Construction of the Invariant-Preserving Projection

In this section, we describe how a message-passing program P is transformed into a sequential program that can be proved using existing deductive verifiers, such as Frama-C.

We assume P is written in C with simple *send* and *recv* primitives, and that global variables `nprocs` and `pid` hold the number of processes and the process ID (in $0..\texttt{nprocs}-1$), respectively. For simplicity, we assume there is some fixed channel size $d > 0$, i.e., for any two processes p and q, up to d messages sent from p to q can be buffered at any time.

An example of such a program is the *cyclic exchanger* of Figure 1. Each process in this program starts with an arbitrary value in `sbuf`. It sends this to its right neighbor, receives a value from its left neighbor, copies that value into `sbuf`, and repeats `nsteps` times. The channel size is assumed to be 1. At termination, variable `rbuf` will hold the value that was originally in `sbuf` on the process `nsteps` to the left (cyclically). Our goal

```
1 int rank, nprocs, nsteps;
2 double rbuf, sbuf;
3 #define  LEFT(pid) ((pid)>0 ? (pid)-1 : nprocs-1)
4 #define RIGHT(pid) ((pid)<nprocs-1 ? (pid)+1 : 0)
5 ...
6 void exchange() {
7     int t = 0;
8     while (t < nsteps) {
9         send(&sbuf, RIGHT(rank));
10        recv(&rbuf, LEFT(rank));
11        sbuf = rbuf;
12        t++;
13    }
14 }
```

Fig. 1. Cyclic exchanger: each process sends to its right and receives from its left.

is to prove that every process terminates with this result, for any $\texttt{nprocs} \geq 1$ and $\texttt{nsteps} \geq 1$.

The first step is decompose P into *blocks*. A block is just a contiguous sequence of statements in which no statement other than the first is a *send* or *recv*. These two communication statements can occur only as the first statement of a block.

Now consider the program P' in which each block is executed atomically, i.e., without interleaving of other processes. It is well-known (see, for example, Lipton [21]) that P and P' are equivalent in all important respects. In particular, the final states of P and P' are identical, and P is deadlock-free if and only P' is deadlock-free.

To see why these claims hold in our context, consider an execution of P, which is represented as some interleaved sequence of statements from different processes:

$$s_0, s_1, s_2, \ldots, s_n.$$

If s_i and s_{i+1} are statements from two different processes, and at least one does not involve communication, then the two statements commute, i.e., executing s_{i+1} followed by s_i results in the same state as executing s_i followed by s_{i+1}.

Now if s_0 is in process p, find the least $i > 0$ such that s_i is in process p. Because s_i commutes with all s_j $(1 \leq j < i)$, we can move s_i all the way to the left until it occurs just after s_0. We can proceed in this way until all of the statements in the block started by s_0 are contiguous. The next statement starts a new block, and we proceed in the same way to bring all statements in that block together. Continuing in this way, we eventually end up with a sequence which is an execution of P', and ends in the same final state as P.

The transformation process requires the formulation of a *core global invariant* ϕ. This is an assertion about the global state of the program that (1) is expected to hold in the initial state, and (2) should be preserved by each block. If these conditions hold, and all processes

terminate, then ϕ must hold in the final state. If ϕ implies the desired postcondition, the program is correct.

There is no method for constructing ϕ; rather, it requires a deep understanding of the algorithm and why it is correct. In the cyclic exchanger, for example, ϕ will include the following claim: if $0 \leq i <$ `nprocs` and the channel from process i to its right neighbor is not empty, then the message it contains is the original value of `sbuf` on the process that is $s-1$ steps to the left of i, where s is the number of *sends* executed by process i.

As can be seen from this example, the core global invariant may refer to parts of the state on all processes as well as the message channels. To enable this, one introduces *ghost state* into the program. These are variables occurring only in annotations and used only for verification. We add ghost variables to model the message channels. Additional ghost state may be added as needed. If there is a need to refer to a program variable `x`, then one must create a ghost array, say `x_g`, which has length `nprocs` and element type the type of `x`. An element `x[i]` in this array mirrors the value of the variable `x` on process i. At any point in the code where `x` is updated, similar ghost code updating `x_g[rank]` must be inserted. The exact ghost state added will depend on the particular problem and the part of the process-local state referenced in the global invariant. Details for the cyclic exchanger are given in Section III.

Next, we add stubs for the *send* and *recv* functions. These are function prototypes with contracts that specify the effects of these functions on the ghost code. The contract for *send* contains a precondition that states the send can only occur if the outgoing channel is not full. A postcondition specifies that when it returns, the given message has been added to the channel, and that no other state has changed. The contract for *recv* is similar.

Of course, it is not necessarily the case that a send or receive statement will be enabled as soon as control reaches such a statement. In general, a process may have to wait until actions from other processes enable the communication. If we attempted to prove the program as currently described, the proof would fail because the preconditions on the communication statements would not be met. This is as it should be—without further information, one cannot rule out the possibility that the program will deadlock.

For now, we will *assume* that every communication event is eventually enabled and focus on proving the functional correctness of the code, i.e., on partial correctness. In Section IV, we discuss how to prove deadlock-freedom and termination.

We must also encode the assumption that just before each block, other processes can change any part of the non-local state in any way that preserves ϕ. To deal with these issues, the IPP includes two more function stubs: *pre_send_interleave* and *pre_recv_interleave*. The contracts for each assume ϕ holds in the prestate and ensure ϕ holds in the poststate. Function *pre_send_interleave* also contains a postcondition stating that the sending channel is not full; *pre_recv_interleave* ensures the receive channel is not empty. The appropriate interleave function is called immediately before each communication operation.

Finally, the IPP includes an assertion that ϕ holds in the initial state, and a postcondition stating the desired correctness properties. The result is a closed, annotated sequential program that makes repeated calls to *interleave* functions which mix up the shared state in a way that preserves a global invariant. One is now free to use all the tools of deductive verification for sequential programs, including loop invariants, additional "process-local" ghost code, and axioms, to prove this program.

III. Example: Cyclic Exchanger

We now show how the method described above plays out for the cyclic exchanger. The annotations are expressed in ACSL, the specification language used by Frama-C. The entire IPP for the cyclic exchanger is given in Figure 2.

The channels are represented using ghost variables `chans` and `sizes`, both arrays of length `nprocs`. For $0 \leq i <$ `nprocs`, the integer `sizes[i]` is the number of buffered message in route from process i to process $(i+1)\%n$. We are assuming each such channel can hold at most 1 message; it is not hard to generalize this to any capacity. The variable `chans[i]` holds the actual data for the message in case `sizes[i]` $= 1$; if `sizes[i]` $= 0$, the contents of `chans[i]` is ignored. We have also simplified the `send` and `recv` functions by removing the source and destination parameters, since these are fixed.

A ghost array `sc` has been added to keep track of the "send count": `sc[i]` is the number of sends that have been executed by process i. Similarly, the array `rc` is the "receive count." These will be used to express the core invariant.

We use a logical function `oracle` to specify the expected contents of the send and receive buffers at each time step. The `oracle` function is uninterpreted, but constrained by formula `oracle_ax`. The expression `oracle(0,i)` represents the initial value of `sbuf` on process i and is unconstrained; for $t > 0$, `oracle(t,i)` is defined to be the value of `oracle` at time $t-1$ on the left neighbor of process i. This same technique can be used to define an oracle for many different numeric schemes, including stencil computations.

The contracts on the *interleave* functions require that ϕ hold in the prestate and ensure that ϕ hold in the post-

```
1  int rank, nprocs, nsteps;
2  double rbuf, sbuf;
3  #define LEFT(rank) ((rank) > 0 ? (rank) - 1 : nprocs - 1)
4  /*@ axiomatic OracleSpec {
5    @  logic double oracle(integer t, integer i);
6    @  axiom oracle_ax: \forall int t, i; t > 0 ==> oracle(t-1,LEFT(i)) == oracle(t,i);
7    @ } */
8  //@ ghost int * sc;       // send counters, one per proc
9  //@ ghost int * rc;       // recv counters, one per proc
10 //@ ghost double * chans; // sending channels, one per proc
11 //@ ghost int * sizes;    // sending channel sizes, one per proc
12 /*@ predicate inv0 = \valid(sc+(0..nprocs-1)) && \valid(rc+(0..nprocs-1))
13   @  && \valid(chans+(0..nprocs-1)) && \valid(sizes+(0..nprocs - 1)) &&
14   @  \separated(sc+(0..nprocs-1), rc+(0..nprocs-1), chans+(0..nprocs-1),
15   @     sizes+(0..nprocs-1));
16   @ predicate inv1 = \forall int i; 0<=i<nprocs ==> 0<=sizes[i]<=1;
17   @ predicate inv2 = \forall int i; 0<=i<nprocs ==> 0<=sc[i]<=nsteps;
18   @ predicate inv3 = \forall int i; 0<=i<nprocs ==> 0<=rc[i]<=nsteps;
19   @ predicate inv4 = \forall int i; 0<=i<nprocs ==> rc[i]==sc[LEFT(i)]-sizes[LEFT(i)];
20   @ predicate inv5 = \forall int i; 0<=i<nprocs ==> (sizes[i]==1 ==> chans[i]==oracle(sc[i]-1,i));
21   @ predicate inv6 = \forall int i; 0<=i<nprocs ==> (sc[i]-rc[i]==0 || sc[i]-rc[i]==1);*/
22 #define inv (0 <= rank < nprocs && inv0 && inv1 && inv2 && inv3 && inv4 && inv5 && inv6)
23 /*@ requires \valid(x) && sizes[rank] == 0 && 0 <= rank < nprocs;
24   @ assigns chans[rank], sizes[rank];
25   @ ensures chans[rank] == *x && sizes[rank] == 1; */
26 void send(double * x);
27 /*@ requires \valid(y) && sizes[LEFT(rank)] == 1 && 0 <= rank < nprocs;
28   @ assigns *y, sizes[LEFT(rank)];
29   @ ensures *y == chans[LEFT(rank)] && sizes[LEFT(rank)] == 0; */
30 void recv(double * y);
31 /*@ requires inv;
32   @ assigns sizes[0..nprocs-1], chans[0..nprocs-1], sc[0..nprocs-1], rc[0..nprocs-1];
33   @ ensures sc[rank]==\old(sc[rank]) && rc[rank]==\old(rc[rank]);
34   @ ensures sizes[LEFT(rank)]==1 && chans[rank]==\old(chans[rank]) && inv; */
35 void prerecv_interleave(void);
36 /*@ requires inv;
37   @ assigns sizes[0..nprocs-1], chans[0..nprocs-1], sc[0..nprocs-1], rc[0..nprocs-1];
38   @ ensures sc[rank]==\old(sc[rank]) && rc[rank]==\old(rc[rank]);
39   @ ensures sizes[rank]==0 && chans[rank]==\old(chans[rank]) && inv; */
40 void presend_interleave(void);
41 /*@ requires inv && sizes[rank]==0 && sbuf==oracle(0, rank);
42   @ requires 0<nsteps && sc[rank]==0 && rc[rank]==0;
43   @ assigns chans[0..nprocs-1], rbuf, sbuf, sizes[0..nprocs-1], rc[0..nprocs-1], sc[0..nprocs-1];
44   @ ensures rbuf == oracle(nsteps-1, LEFT(rank));*/
45 void exchange() {
46   int t = 0;
47   /*@ loop invariant inv && 0 <= t <= nsteps;
48     @ loop invariant t == 0 ==> sbuf == oracle(0, rank);
49     @ loop invariant t > 0 ==> (sbuf == oracle(t-1, LEFT(rank)) && sbuf == rbuf);
50     @ loop invariant t == sc[rank] && t == rc[rank];
51     @ loop assigns sbuf, t, rbuf, sc[0..nprocs-1], rc[0..nprocs-1];
52     @ loop assigns chans[0..nprocs-1], sizes[0..nprocs-1];
53     @ loop variant nsteps - t; */
54   while (t < nsteps) {
55     presend_interleave();
56     send(&sbuf);
57     /*@ ghost sc[rank]++;*/       /*@ assert A0: inv6; */
58     prerecv_interleave();
59     /*@ assert A1: inv4;*/        /*@ assert A2: inv5; */
60     recv(&rbuf);                  /*@ assert A3: inv5;*/
61     /*@ ghost rc[rank]++;*/
62     /*@ assert A4: inv4;*/        /*@ assert A5: inv6; */
63     sbuf = rbuf;
64     t++;
65   }
66 }
```

Fig. 2. Annotated code used to prove partial correctness of the cyclic exchanger. This program is proved using Frama-C+WP with Why3, and provers Alt-Ergo and CVC4.

state. Furthermore, these functions may modify any of the shared (ghost) variables in an arbitrary way—as long as in the resulting state ϕ holds. Hence the `assigns` clause includes all of the shared ghost variables. An exception is made for the ghost arrays `x_g` that mirror program variables — since other processes cannot access the `x` on "this" process, this procedure must ensure that `x_g[i] == \old(x_g[i])`.

The core invariant `inv` includes the following claims:

1) `rank` is in the range $0..\texttt{nprocs}-1$,
2) the various arrays are allocated and occupy disjoint memory regions,
3) the size (current number of buffered messages) of any channel is either 0 or 1,
4) all send and receive counts are bounded by `nsteps`,
5) the number of messages received by process i equals the number of messages sent by its left neighbor minus the number of those messages currently buffered,
6) if the outgoing channel from process i is not empty, the value of the buffered message is the value of the oracle at time step `sc[i]-1` in position i, and
7) on any process the send count equals the receive count, or is one greater than the receive count.

Loop invariants and intermediate assertions to assist the provers have also been added.

We used Frama-C+WP (Chlorine-20180501) to generate verification conditions for this program, and Why3 (v0.88.3) to discharge the VCs not automatically proved by WP. Using the Why3 IDE, we repeatedly applied the "Split" and "Inline" transformations to the VCs. We then invoked Alt-Ergo (v2.2.0) and CVC4 (v1.6) with a timeout of 6s and then invoked them again with a timeout of 100s. All VCs were discharged. The total time required to do this was approximately 113s. This experiment was done on a MacBook Air with a 1.6GHz i5 CPU and 8 GB RAM.

IV. Deadlock-freedom and Termination

A. Deadlock-freedom

A small extension to the method can be used to prove deadlock-freedom. We say a process is disabled at a state s, if that process is attempting to execute a *send* but the outgoing channel is full, or it is attempting to execute a *recv* and the incoming channel is empty. A deadlock occurs at a state s if at least one process has not terminated and every non-terminated process is disabled.

The program is deadlock-free if, at each interleave point in an execution, at least one process is enabled. We wish to express this as a predicate in ACSL. We form such a predicate as follows. First, we can number all communication statements in the program, starting from 0. Given an index j, one can construct a predicate $\theta_{i,j}$ that holds precisely when the j^{th} communication statement is enabled in process i. If communication statement j is a send in process i with destination expression d, then $\theta_{i,j}$ will state that the channel from i to d is not full. For a receive, $\theta_{i,j}$ states that the channel from the source to i is not full. Next, introduce a ghost array `pc[nprocs]` of integers, with the property that `pc[i]` is the ID number of a communication statement when control is just before that statement, otherwise `pc[i]` is -1. The predicate

$$\mathsf{EN} = \exists i \,.\, 0 \le i < \texttt{nprocs} \,\wedge\, \big(\texttt{pc}[i] = -1 \vee \bigvee_j (\texttt{pc}[i] = j \wedge \theta_{i,j})\big)$$

says that some process is enabled. (We should also include the case that every process has terminated, but we leave that out to simplify the discussion.)

In the cyclic exchanger, we let $\texttt{pc}[i] = \texttt{sc}[i] - \texttt{rc}[i]$. If $\texttt{pc}[i] = 0$ then process i is just before the *send*, else $\texttt{pc}[i] = 1$ and it is just before the *recv*. Then EN is equivalent to the following:

$$\exists i \,.\, 0 \le i < \texttt{nprocs} \wedge \big((\texttt{pc}[i] = 0 \wedge \texttt{sizes}[i] = 0) \vee (\texttt{pc}[i] = 1 \wedge \texttt{sizes}[\texttt{LEFT}(i)] = 1)\big).$$

To prove deadlock-freedom we could either add EN to the core invariant or simply prove the validity of $\texttt{inv} \rightarrow \mathsf{EN}$. We took the latter approach, inserting this formula as a lemma into the program. On our first attempt, the automated provers were unable to prove this lemma. However, when we placed a concrete upper bound on `nprocs` the lemma could be proved automatically by Z3. Increasing this upper bound increases the time it takes Z3 to do the proof. For an upper bound of 20, this took 47s. We also manually encoded the lemma in the input language of CVC4, (using arrays instead of pointers), and were able to prove it with an upper bound of 100 in 36s; for 200, it took 6 minutes and 13s. We have not yet found an automated prover that can prove this theorem without a concrete bound on `nprocs`. An informal proof of the theorem is given in Appendix A.

B. Termination

It is possible that a parallel program can be deadlock-free but not terminate, just as with a sequential program. However, if we have already established that the program is deadlock-free, then termination can be proved by purely process-local reasoning. If we can show that no process can execute an infinite number of statements, then we know that every execution is finite. If in addition the program is deadlock-free, then every process must terminate—otherwise there would be a finite execution in which some process did not terminate, i.e., a deadlock.

```
#define NPROCSB 20
#define LEFT(rank) ((rank) > 0 ? (rank) - 1 : nprocs - 1)
//@ ghost int sc[NPROCSB], rc[NPROCSB], sizes[NPROCSB];
int nprocs, nsteps;
/* note that inv0 and inv5 is not needed for proving the deadlock-freedom condition */
/*@ axiomatic FDL {
  @   predicate inv1 = \forall integer i; 0 <= i < nprocs ==> 0 <= sizes[i] <= 1;
  @   predicate inv2 = \forall integer i; 0 <= i < nprocs ==> 0 <= sc[i] <= nsteps;
  @   predicate inv3 = \forall integer i; 0 <= i < nprocs ==> 0 <= rc[i] <= nsteps;
  @   predicate inv4 = \forall integer i; 0 <= i < nprocs ==> rc[i]==sc[LEFT(i)]-sizes[LEFT(i)];
  @   predicate inv6 = \forall integer i; 0 <= i < nprocs ==> (sc[i]-rc[i]==0) || (sc[i]-rc[i]==1);
  @   predicate fdl = \exists integer i; 0 <= i < nprocs && ((sc[i]-rc[i]==0 && sizes[i]==0) ||
  @   (sc[i]-rc[i]==1 && sizes[LEFT(i)]==1));
  @   lemma bounded_free_of_deadlock : inv1 && inv2 && inv3 && inv4 && inv6 && 0<nprocs<=NPROCSB ==> fdl
  @ }*/
```

Fig. 3. ACSL lemma for proving deadlock-freedom in the cyclic exchanger, with `nprocs` ≤ 20.

To check that a process cannot execute infinitely, we can use Frama-C's techniques for proving termination. This mainly involves adding loop *variants*—integer expressions which are strictly decreasing and which imply the loop must exit once they are negative. This is quite straightforward in the example, and in fact in most HPC programs, which typically use `for` loops or other counters to guarantee that a program will terminate.

V. Related Work

Ashcroft [22] introduced the idea of using a global invariant to prove properties of parallel programs. Further theoretical groundwork includes the work of Owicki and Greis, which introduced an axiomatic proof system in the style of Hoare's logic [23]. A proof in that system consists of a proof for each process which satisfies a "non-interfering" property; other efforts along these lines include [24]–[26].

There are a growing number of deductive verification tools targeting concurrent systems. In addition to the tools mentioned in Section I, proof systems and tools for process algebras such as CSP have also been developed [27].

Many of these tools target shared-variable programs. VCC [28] uses an annotation language for multithreaded C programs and generates verification conditions for the Boogie platform. Other tools in this category include VerCors [29] and VeriFast [30]. These tools and languages deal with notions such as type invariants and object ownership, which are somewhat different from the issues that arise in message-passing.

There has been some research into verifying properties for MPI programs of arbitrary scale. Dataflow analyses on "parallel control flowgraphs" that generalize the traditional notions for sequential programs are one approach [31]–[34]. To the best of our knowledge this type of dataflow analysis has not been used to prove functional correctness properties of MPI programs. We have already discussed the ParTypes approach in Section I.

VI. Conclusion and Future Work

In this paper, we introduced a new method for verifying message-passing programs without bounding the number of processes or other parameters. In this method, one transforms the program into a sequential program representing a generic process (the IPP), and then applies deductive verification techniques to show that the IPP preserves a global invariant.

We demonstrated the method on a simple example, the *cyclic exchanger*. We manually transformed the program, and then used Frama-C/WP to verify the invariant. We also used Frama-C to try to verify three properties: the result computed by the program is correct (where the expected result was specified using a simple inductive formula); the program is free of deadlocks; and all processes terminate normally. For the deadlock property, the key formula could not be proved valid by any of the automated provers we used. However, these provers could validate the formula when placing an upper bound (as high as 200) on the number of processes; we also provide an informal proof in Appendix A. All other properties were fully proved.

While the example used in this paper is simple, it uses a general pattern that is common to many stencil computations. A 1-dimensional diffusion equation solver ("diffusion1d"), for example, might perform a cyclic ghost cell exchange at each time step that differs from the cyclic exchanger only in that the ghost cell exchange takes place in both directions. In addition, diffusion1d performs a nontrivial update at each time step. The update formula can be encoded in the oracle in a way that is similar to the oracle definition in the cyclic exchanger; see Figure 4.

For diffusion1d, since a process sends to its left and right neighbors, a single channel for each process will not do. Two channels per process could be used, or for greater generality, one could use a 2-dimensional array of channels with the entry at (i, j) a FIFO queue of buffered messages sent from process i to process j. The invariant for diffusion1d would specify the block

```
ghost double **a;
predicate oracle = \forall integer t, i; 0 < t && 1 <= i < (nx-1) ==>
  a[t][i] == a[t-1][i] + k * (a[t-1][i+1] + a[t-1][i-1] - 2 * a[t-1][i]);
```

Fig. 4. Oracle for a 1d-diffusion solver

distribution of the temperature array, and would relate the buffered messages with the oracle; see Figure 5.

With a little more work, these proof techniques could be applicable to higher-dimensional simulations (e.g., 2d- or 3d-diffusion), and a wide range of parallel stencil computations.

There are some ways this method could erroneously conclude that an incorrect program is correct. First, there could be a defect in Frama-C or the WP plugin, causing these tools to produce incorrect proof obligations (POs). Frama-C and WP are not formally verified programs; however, they are mature, widely-used tools that are actively developed and maintained. In this study, we also created several intentionally erroneous versions of the cyclic exchanger and re-ran Frama-C/WP on each. In each case, one or more POs could not be discharged, increasing our confidence in the soundness of these tools. One could also apply different deductive verification tools to confirm the Frama-C/WP results.

Second, a prover might erroneously conclude an invalid PO is valid. One can guard against being led astray by an errant prover by running several different provers on each PO. It is very unlikely that several would all make the same error.

Third, our transformation embeds a model of the MPI runtime in the sequential program. The model is specified by ghost state and contracts on prototypes for the message-passing functions. While this is relatively straightforward in this simple example, programs that use more complicated MPI routines will require correspondingly more complex models. If the model does not faithfully represent the MPI semantics, one could reach an erroneous conclusion. To move this approach forward will require the development of carefully crafted, general models of large parts of the MPI Standard. These models should be able to be used across different programs, rather than designing bespoke models for each problem.

In our experience, the main barrier to this approach is the ability to discharge the generated verification conditions. Even for the toy example used here, we had to repeatedly insert intermediate assertions before the provers could discharge all conditions. We also had to provide some manual guidance using the Why3 IDE to transform the formulas. As mentioned above, the validity of the formula which implies deadlock-freedom was never established by any automated tool (though it was proved for $\texttt{nprocs} \leq 200$). We were surprised that the correctness of such a simple program entails such complex theorems. We plan to provide these queries to SMT prover developers, with the hope they will guide future developments of those tools. It may also be the case that verification conditions arising from message-passing programs are often beyond the current capabilities of automated theorem proving, and interactive provers will be required.

ACKNOWLEDGMENT

Funding for this work was provided by the U.S. National Science Foundation under award CCF-1319571.

This material is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number DE-SC0012566. This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

[1] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. J. Naughton, III, H. P. Pritchard, M. Schulz, and G. R. Vallee, "A survey of MPI usage in the U.S. Exascale Computing Project," Oak Ridge National Lab., Tech. Rep. ORNL/SPR-2018/790, Jun. 2018. [Online]. Available: https://doi.org/10.2172/1462877

[2] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "CIVL: The Concurrency Intermediate Verification Language," in *SC15*, 2015. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807635

[3] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, "Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings," in *Computer Aided Verification*, ser. LNCS, vol. 5123, 2008, pp. 66–79. [Online]. Available: https://doi.org/10.1007/978-3-540-70545-1_9

[4] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for MPI programs," in *SC '10*. IEEE/ACM, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.7

```
\let left = LEFT(i); \let right = RIGHT(i);
  (sizes[i][left] == 1 ==> chans[i][left] == a[sc[i][left]][firsts[i]]) &&
  (sizes[i][right] == 1 ==> chans[i][right]
       == a[sc[i][right]][firsts[i] + nxls[i] - 1]);
```

Fig. 5. Part of the core global invariant for a 1d-diffusion solver

[5] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise predictive analysis for discovering communication deadlocks in MPI programs," in *FM 2014: Formal Methods*, ser. LNCS, vol. 8442. Springer, 2014, pp. 263–278. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06410-9_19

[6] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C — a software analysis perspective," in *Software Engineering and Formal Methods, SEFM 2012*, ser. LNCS. Springer, 2012, pp. 233–247. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33826-7_16

[7] "Frama-C," http://frama-c.com/, accessed Sep. 14, 2018.

[8] "Frama-C: WP plug-in," https://frama-c.com/wp.html, accessed Sep. 14, 2018.

[9] P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye, "WP plug-in manual: Frama-C Chlorine-20180501," http://frama-c.com/download/frama-c-wp-manual.pdf, 2018.

[10] J.-C. Filliâtre and A. Paskevich, "Why3: Where programs meet provers," in *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP'13)*, 2013, pp. 125–128. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37036-6_8

[11] "Why3: Where programs meet provers," http://why3.lri.fr, accessed Sep. 14, 2018.

[12] J. Burghardt and J. Gerlach, "ACSL by example: Towards a verified C standard library, version 17.2.0 for Frama-C 17 (Chlorine-20180502)," July, 2018. [Online]. Available: https://github.com/fraunhoferfokus/acsl-by-example

[13] "Toccata: Formally Verified Programs, Certified Tools and Numerical Computations," accessed Sep. 14, 2018. [Online]. Available: http://toccata.lri.fr/gallery/

[14] S. Boldo, "Kahan's algorithm for a correct discriminant computation at last formally proven," *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 220–225, Feb 2009. [Online]. Available: https://doi.org/10.1109/TC.2008.200

[15] ——, "Toccata: Accurate discriminant," accessed Nov. 13, 2017. [Online]. Available: http://toccata.lri.fr/gallery/discri.en.html

[16] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis, "Wave equation numerical resolution: A comprehensive mechanized proof of a C program," *Journal of Automated Reasoning*, vol. 50, no. 4, pp. 423–456, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10817-012-9255-4

[17] M. Clochard, L. Gondelman, and M. Pereira, "Toccata: solution to verifythis 2016 challenge 1," accessed Nov. 13, 2017. [Online]. Available: http://toccata.lri.fr/gallery/verifythis_2016_matrix_multiplication.en.html

[18] H. A. López, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida, "Protocol-based verification of message-passing parallel programs," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, J. Aldrich and P. Eugster, Eds. New York, NY, USA: ACM, 2015, pp. 280–298. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814302

[19] C. Santos, F. Martins, and V. T. Vasconcelos, "Deductive verification of parallel programs using Why3," in *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015.*, ser. EPTCS, S. Knight, I. Lanese, A. Lluch-Lafuente, and H. T. Vieira, Eds., vol. 189. New York, USA: ACM, 2015, pp. 128–142. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.189.11

[20] E. R. B. Marques, F. Martins, V. T. Vasconcelos, N. Ng, and N. Martins, "Towards deductive verification of MPI programs against session types," in *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013*, ser. EPTCS, vol. 137, 2013, pp. 103–113. [Online]. Available: https://doi.org/10.4204/EPTCS.137.9

[21] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717–721, 1975. [Online]. Available: http://doi.acm.org/10.1145/361227.361234

[22] E. A. Ashcroft, "Proving assertions about parallel programs," *J. Comput. Syst. Sci.*, vol. 10, no. 1, pp. 110–135, Feb. 1975. [Online]. Available: http://dx.doi.org/10.1016/S0022-0000(75)80018-3

[23] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Inf.*, vol. 6, pp. 319–340, 1976. [Online]. Available: http://doi.org/c9fkv7

[24] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, no. 2, pp. 125–143, Mar. 1977. [Online]. Available: http://dx.doi.org/10.1109/TSE.1977.229904

[25] ——, "The 'Hoare logic' of concurrent programs," *Acta Informatica*, vol. 14, pp. 21–37, 1980. [Online]. Available: http://dx.doi.org/10.1007/BF00289062

[26] L. Lamport and F. B. Schneider, "The "Hoare Logic" of CSP, and all that," *ACM Trans. Program. Lang. Syst.*, vol. 6, pp. 281–296, April 1984. [Online]. Available: http://doi.acm.org/10.1145/2993.357247

[27] Y. Isobe and M. Roggenbach, "CSP-Prover – a proof tool for the verification of scalable concurrent systems," *Journal of Computer Software*, vol. 25, no. 4, pp. 85–92, 2008. [Online]. Available: https://www.jstage.jst.go.jp/article/imt/5/1/5_1_32/_pdf

[28] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Springer Berlin / Heidelberg, 2009, vol. 5674, pp. 23–42. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03359-9_2

[29] A. Amighi, S. Blom, and M. Huisman, "VerCors: A layered approach to practical verification of concurrent software," in *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*. IEEE Computer Society, 2016, pp. 495–503. [Online]. Available: http://dx.doi.org/10.1109/PDP.2016.107

[30] B. Jacobs, "VeriFast," https://github.com/verifast/, accessed Nov. 13, 2017.

[31] D. R. Shires, L. L. Pollock, and S. Sprenkle, "Program flow graph construction for static analysis of MPI programs," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 – July 1, 1999, Las Vegas, Nevada, USA*, H. R. Arabnia, Ed. CSREA Press, 1999, pp. 1847–1853. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.2545&rep=rep1&type=pdf

[32] M. M. Strout, B. Kreaseck, and P. D. Hovland, "Data-flow analysis for MPI programs," in *Proceedings of the 2006 International Conference on Parallel Processing*, ser. ICPP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 175–184. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2006.32

[33] G. Bronevetsky, "Communication-sensitive static dataflow for parallel message-passing applications," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/CGO.2009.32

[34] S. Aananthakrishnan, G. Bronevetsky, and G. Gopalakrishnan, "Hybrid approach for data-flow analysis of MPI programs," in *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10–14, 2013*, A. D. Malony, M. Nemirovsky, and S. P. Midkiff, Eds. New York: ACM, 2013, pp. 455–456. [Online]. Available: http://doi.acm.org/10.1145/2464996.2467286

APPENDIX A
PROOF OF DEADLOCK FREEDOM

In this section we prove the validity of the deadlock-freedom formula `inv` $\rightarrow$ EN of Section IV.

Let $n =$ `nprocs`, $s_i =$ `sc[i]`, $r_i =$ `rc[i]`, and $z_i =$ `sizes[i]` $(0 \leq i < n)$. Then `inv` implies all of the following:

$$n \geq 1 \tag{1}$$

$$\forall i \,.\, 0 \leq s_i - r_i \leq 1 \tag{2}$$

$$\forall i \,.\, 0 \leq z_i \leq 1 \tag{3}$$

$$\forall i \,.\, s_{i-1} - r_i = z_{i-1} \tag{4}$$

The indexes are to be read modulo n; e.g., if $i = 0$ then $z_{i-1} = z_{n-1}$. We wish to show

$$\exists i . (s_i - r_i = 0 \wedge z_i = 0) \vee (s_i - r_i = 1 \wedge z_{i-1} = 1). \tag{5}$$

Summing (4) over all i, we obtain

$$\sum_i (s_i - r_i) = \sum_i s_i - \sum_i r_i = \sum_i z_i. \tag{6}$$

Suppose $s_i - r_i = 1$ for all i. By (6), $\sum_i z_i = n$, so by (3), $z_i = 1$ for all i. Hence any i will satisfy the second clause in (5).

Suppose $s_i - r_i = 0$ for all i. By (6), $\sum_i z_i = 0$, so by (3), $z_i = 0$ for all i. Hence any i will satisfy the first clause of (5).

So let us assume that neither $s_i - r_i = 0$ for all i, nor $s_i - r_i = 1$ for all i. Then $n \geq 2$, and there exists j for which $s_j - r_j = 0$ and $s_{j+1} - r_{j+1} = 1$. There are two cases: $z_j = 0$ or $z_j = 1$. If $z_j = 0$, then $i = j$ satisfies the first clause of (5). If $z_j = 1$, then $i = j + 1$ satisfies the second clause of (5).

APPENDIX B
ARTIFACT DESCRIPTION FOR REPRODUCIBILITY

In order to install the tools used in these experiments, first install the package manager OPAM (https://opam.ocaml.org). The exact versions of the tools used here can then be installed as follows:

```
opam pin add frama-c 20180501
opam pin add why3 0.88.3
opam pin add alt-ergo 2.2.0
```

It is possible that later versions of these tools will also work; to get the latest versions for which packages are available, use instead

```
opam install frama-c why3 alt-ergo
```

The automated theorem prover CVC4 can be obtained from http://cvc4.cs.stanford.edu/downloads/. We used the latest version at the time of publication, 1.6. The download site provides binaries for various platforms, as well as the source code and instructions for building from source. Alternatively, on OS X it can be installed using *Homebrew* with the following commands:

```
brew tap cvc4/cvc4
brew install cvc4/cvc4/cvc4
```

The automated theorem prover Z3 can be obtained from https://github.com/Z3Prover/z3/releases. We used the binaries of the latest version at the time of publication, 4.7.1. The download site provides binaries for various platforms, as well as the source code and instructions for building from source.

After all of these tools are installed, configure Why3 with the command

```
why3 config --detect
```

The experiments described in this paper can now be replayed using the artifacts at https://vsl.cis.udel.edu/correctness18. File `cyclic_exchanger.c` contains the annotated program shown in Fig. 2. The program can be proved as follows:

1) The command
   ```
   frama-c -wp -wp-prover why3ide
   cyclic_exchanger.c
   ```
   launches Frama-C/WP with the Why3 GUI. Figure 6 is a screenshot of the Why3 GUI. The "Theories/Goals" tree gives the verification obligations generated by Frama-C/WP.
2) Select File→Preferences. In order to achieve the performance reported in the paper, we set prover timeout to 10 seconds, memory to 4GB, and allowed at most 6 processes (which share the 4GB memory) to call provers in parallel. These parameters might require adjustment to obtain similar performance on different platforms.
3) Select the root goal, "exchange_Why3_ide.why", and click the "Alt-Ergo" and "CVC4" buttons. Most of the goals should be discharged by the two provers, but a few will remain unproved.
4) With the root goal still selected, click the "Inline" button three times. This repeatedly transforms the formulas by inlining predicate definitions, until no further inlining is possible.
5) Click the "Split" button once to break up conjunctive formulas into separate sub-goals.
6) Click the "Alt-Ergo" and "CVC4" buttons again. Several more goals should be proved.

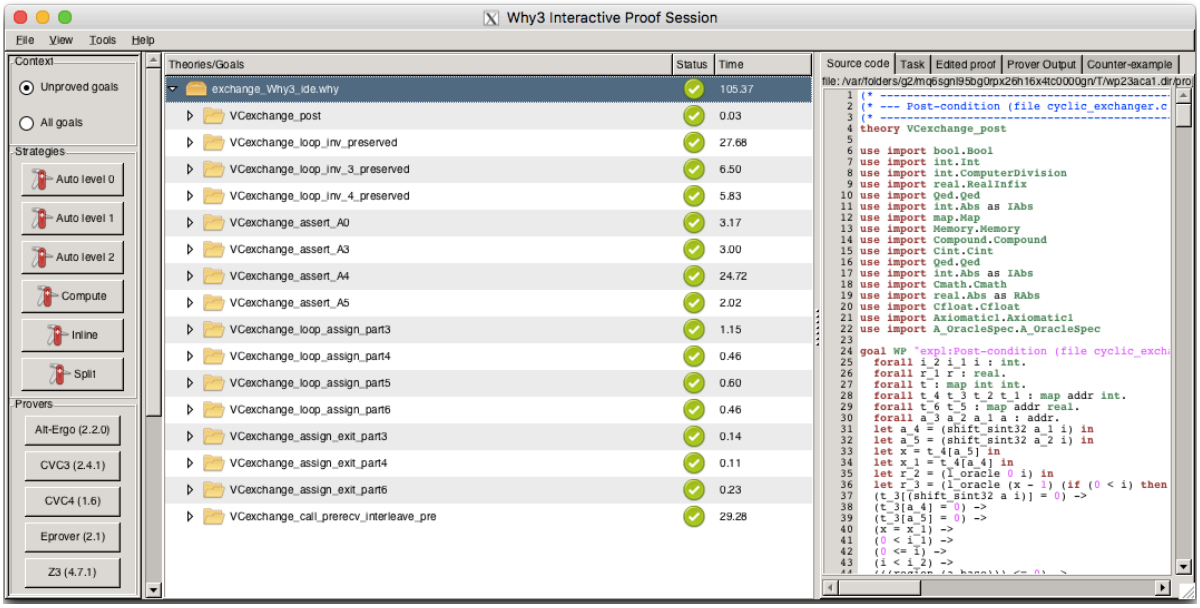

Fig. 6. Screenshot after using Why3 to discharge all the verification obligations for the cyclic exchanger

7) Set prover timeout to 50 seconds and allow at most 1 process to call provers sequentially. Click "Alt-Ergo" and "CVC4" again and all remaining goals should be proved.

The deadlock-freedom lemma is contained in file `fdl.c`, and can be proved using Frama-C/WP in a similar way. Only one proof obligation is generated. It was discharged by setting a timeout of 50 seconds, choosing 4GB memory, and then clicking the "Z3" button.

The deadlock-freedom lemma expressed directly in the native language of CVC4 is given in file `dl.cvc`. It can be proved quickly with the command

```
cvc4 --fmf-bound dl.cvc
```

This file uses an upper bound of 20 on `nprocs`, but it can be edited to use other upper bounds. As reported in Sec. IV, an upper bound of 200 required just over 6 minutes.

Note that the programs in the experiments can be verified with many different settings and SMT provers. The procedures we presented above are the most efficient we have found.