DafnyMPI: A Dafny Library for Verifying Message-Passing Concurrent Programs

ALEKSANDR FEDCHIN, Tufts University, USA and American University of Central Asia, Kyrgyzstan
ANTERO MEJR, Tufts University, USA
HARI SUNDAR, Tufts University, USA
JEFFREY S. FOSTER, Tufts University, USA

The Message Passing Interface (MPI) is widely used in parallel, high-performance programming, yet writing bug-free software that uses MPI remains difficult. We introduce DafnyMPI, a novel, scalable approach to formally verifying MPI software. DafnyMPI allows proving deadlock freedom, termination, and functional equivalence with simpler sequential implementations. In contrast to existing specialized frameworks, DafnyMPI avoids custom concurrency logics and instead relies on Dafny, a verification-ready programming language used for sequential programs, extending it with concurrent reasoning abilities. DafnyMPI is implemented as a library that enables safe MPI programming by requiring users to specify the communication topology upfront and to verify that calls to communication primitives such as `MPI_ISEND` and `MPI_WAIT` meet their preconditions. We formalize DafnyMPI using a core calculus and prove that the preconditions suffice to guarantee deadlock freedom. Functional equivalence is proved via rely-guarantee reasoning over message payloads and a system that guarantees safe use of read and write buffers. Termination and the absence of runtime errors are proved using standard Dafny techniques. To further demonstrate the applicability of DafnyMPI, we verify numerical solutions to three canonical partial differential equations. We believe DafnyMPI demonstrates how to make formal verification viable for a broader class of programs and provides proof engineers with additional tools for software verification of parallel and concurrent systems.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Model checking**; • **Theory of computation** → **Program verification**.

Additional Key Words and Phrases: Message passing, deadlocks, liveness, rely-guarantee, Dafny, MPI

ACM Reference Format:
Aleksandr Fedchin, Antero Mejr, Hari Sundar, and Jeffrey S. Foster. 2026. DafnyMPI: A Dafny Library for Verifying Message-Passing Concurrent Programs. *Proc. ACM Program. Lang.* 10, POPL, Article XXX (January 2026), 27 pages. https://doi.org/TBD

Preprint. To appear in the *Proceedings of the ACM on Programming Languages (PACMPL), POPL 2026.*

1 Introduction

Computational science relies heavily on parallelism to achieve high performance and enable large-scale simulation, forecasting, and decision-making. The correctness of parallel scientific software is critical, yet there are few practical formal verification techniques for such systems. Model checking and dynamic verification approaches [23, 26, 31, 32, 34] can suffer from the state explosion problem

Authors' Contact Information: Aleksandr Fedchin, aleksandr.fedchin@tufts.edu, Tufts University, Medford, Massachusetts, USA and American University of Central Asia, Bishkek, Kyrgyzstan; Antero Mejr, antero.mejr@tufts.edu, Tufts University, Medford, Massachusetts, USA; Hari Sundar, hari.sundar@tufts.edu, Tufts University, Medford, Massachusetts, USA; Jeffrey S. Foster, jfoster@cs.tufts.edu, Tufts University, Medford, Massachusetts, USA.

ACM 2475-1421/2026/1-ARTXXX
https://doi.org/TBD

and typically require assuming a predetermined input and number of processes. Proof assistants, e.g., those using concurrent separation logic [13, 16], can perform arbitrarily complex reasoning, but verification of even small programs requires substantial effort by a team of specialists. Existing approaches for solver-aided languages [12, 21] typically emphasize generality by supporting shared-memory concurrency with dynamic thread creation, but this generality can make it difficult to reason about the functional behavior of programs. As a result, prior work typically focuses on relatively simple synchronization patterns, e.g., assuming that all receive operations are blocking and all send operations are not [21].

To address these limitations, we present a new, scalable approach to verifying programs that use the Message Passing Interface (MPI) [25]. Our approach supports verification in the context of blocking, non-blocking, and collective communication behaviors. This is crucial in the MPI setting, where multiple messages may transfer concurrently and asynchronously relative to the initiating process and large messages can cause a send operation to nondeterministically block when buffering is unavailable. We are able to support these behaviors at the cost of both requiring the user to specify the communication topology upfront and relying on the fact that MPI disallows dynamic process creation and assumes no shared memory.

We implement our approach as a library called DafnyMPI, for the Dafny programming language [19, 20]. Dafny is commonly used for the verification of sequential programs [4, 5] but it lacks built-in concurrency support. By integrating DafnyMPI as a library, we show that our approach can be easily integrated with an existing verification framework. This also allows us to preserve all the features already available in Dafny, making DafnyMPI more accessible to software engineers with no prior experience verifying parallel software. Specifically, Dafny already enforces termination and prevents such runtime errors as division by zero. Dafny automates much of the proof process, and the user guides verification by writing pre- and postconditions and invariants. DafnyMPI builds on top of this system by presenting an API that the program can invoke for inter-process communication. Communication correctness, including deadlock freedom, is guaranteed by DafnyMPI if the program satisfies the preconditions on DafnyMPI method calls, and the postconditions allow reasoning about the return values. (Section 2 introduces Dafny and DafnyMPI and demonstrates their use to reason about a parallelized numerical implementation of the linear convection partial differential equation.)

To prove that the specification of DafnyMPI guarantees deadlock freedom, we introduce a core calculus that closely models the behavior of key MPI primitives: barriers and non-blocking send and receive operations. An ordering property on message tags ensures that a process blocked on the message with the lowest tag is eventually unblocked by a matching command from its communication partner. Crucially, our operational semantics models the nondeterministic behavior of MPI send operations: such operations may either block until the receiving process initiates the communication on its end or, if the MPI runtime has enough memory available, the message may be transferred to an internal buffer and the sender may be allowed to proceed. (Section 3 formally defines the core calculus and its operational semantics and outlines the proof of deadlock freedom.)

A user of DafnyMPI may not only want to ensure their program is correct but also to reason about its output. Since concurrent code is error-prone, it is often useful to prove that the parallel version produces the same result as a simpler sequential one. DafnyMPI makes this possible by requiring users to specify in advance the expected results of each inter-process communication operation and by enforcing that the read and write buffers are used safely. Reasoning about functional equivalence between two manually written programs (one parallel and one sequential) can then proceed by transitivity: the user writes a functional specification and proves that both implementations satisfy it. (Section 4 describes how we can prove functional equivalence with DafnyMPI.)

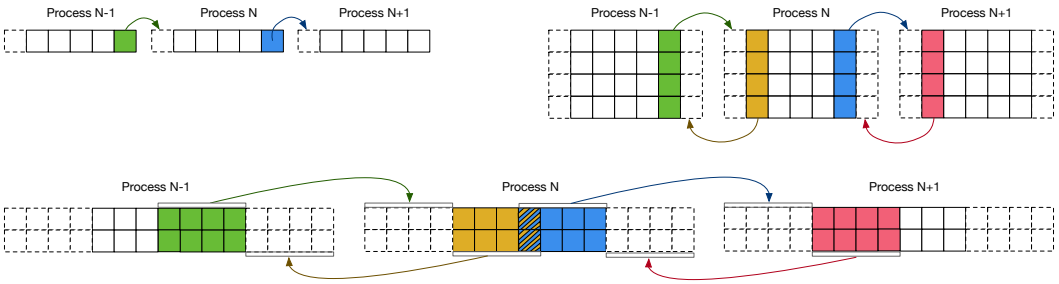

Fig. 1. Structure of MPI communication for the Linear Convection (top left), Poisson (top right, transposed), and Heat Diffusion (bottom, transposed) PDE solver implementations.

To apply DafnyMPI to real-world programs, we use standard Dafny techniques to prove termination and address engineering issues ranging from practical MPI constraints, such as tag overflow, to proof engineering challenges, such as brittleness. Additionally, we specify the behavior of several useful higher-level collective communication operations, which our core calculus abstracts away as decorated barriers. Overall, after dealing with all of the above, DafnyMPI consists of around 2000 lines of Dafny code, of which under 400 lines form the trust base that models our core calculus. (Section 5 discusses the structure of DafnyMPI and associated engineering considerations.)

To demonstrate the practical utility of DafnyMPI, we use it to verify numerical solutions for three partial differential equations (PDEs): Linear Convection, Heat Diffusion, and Poisson. We discuss the proof engineering effort required to verify each benchmark and show that the increased complexity of the parallel versions is mirrored by a higher percentage of ghost (non-executable) code. Finally, we compile the verified Dafny code to Python and demonstrate that the resulting programs exhibit the expected runtime behavior under parallel execution—specifically, that increasing the number of processes leads to faster computation. (Section 6 describes these experiments.)

In summary, the main contributions of this paper are as follows:

- We introduce a technique for verifying concurrent MPI programs and establishing deadlock freedom, termination, and functional equivalence with a sequential implementation.
- We implement this technique as DafnyMPI, a library for the Dafny programming language.
- We verify three MPI programs to evaluate the practical benefits of the approach. We show that the parallel implementations do exhibit better runtime performance than their sequential counterparts.

DafnyMPI and the associated benchmarks are available at https://doi.org/10.5281/zenodo.17102521

In summary, by verifying MPI programs in a language designed for sequential reasoning, we aim to push the boundaries of scalable verification and support the development of reliable software.

2 Overview

MPI is best suited for applications with well-defined communication topologies, where parallelism serves to optimize computation. One of the key uses of MPI is in partial differential equation (PDE) solvers, where each process is responsible for computing a portion of the solution. Figure 1 shows the structure of MPI communication for three PDE solvers that we use as our benchmarks. All three compute the solution iteratively, and each process must exchange data with its neighbors, as indicated by arrows in the figure. In this section, we focus on the linear convection equation (top left in Figure 1), which is the simplest of the three because each process only sends one value and only receives one value. The remaining two benchmarks are discussed in detail in Section 6. Here,

we begin by outlining the mathematics behind the PDE to motivate concurrency and then present the corresponding code and discuss our verification methodology.

2.1 Example: Linear Convection Partial Differential Equation

The linear convection equation models a wave moving at a constant speed. More formally, it describes a quantity $u(x,t)$ that varies in space and time, and it states that the change over time ($\frac{\partial u}{\partial t}$) is proportional to the slope ($\frac{\partial u}{\partial x}$):

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{1}$$

To solve the equation, one must be able to find the value of u at a given time step t and point x given the initial state u_0 [29]. A numerical solution must approximate the partial derivatives in terms of small discrete steps in time (Δt) and space (Δx). One approach, known as the *upwind* scheme, is to use the forward difference to approximate the temporal derivative and the backward difference to approximate the spatial derivative:

$$\frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} + c\frac{u(x,t) - u(x-\Delta x,t)}{\Delta x} = 0 \tag{2}$$

Solving Equation 2 for $u(x,t+\Delta t)$, we get:

$$u(x,t+\Delta t) = u(x,t) - c\frac{\Delta t}{\Delta x}(u(x,t) - u(x-\Delta x,t)) \tag{3}$$

In other words, the numerical solution iteratively computes $u(x,t+\Delta t)$ for all points in the spatial domain of u. Note that this computation is local—the value at a point x at time $t+\Delta t$ only depends on the values at points x and $x-\Delta x$ at time t. The locality of the computation means it can be easily parallelized across several processes, where each process is responsible for computing a portion of the solution. However, the processes must also exchange information at each time step to compute the boundary values: if point $x-\Delta x$ is outside its allotted portion of its spatial domain, the process responsible for computing $u(x,t+\Delta t)$ must receive the value $u(x-\Delta x,t)$ from its neighbor. This exchange of boundary values is shown with arrows between processes in Figure 1 (top left). The Message Passing Interface is ideal for facilitating such structured communication.

2.2 Solving PDEs with the Message Passing Interface

The Python code on the left side of Figure 2 implements Equation 3 using MPI. As is standard in MPI, the code is executed by launching it simultaneously across many CPU cores. Each process begins by obtaining its rank and the total number of running processes, also known as the size (lines 1–3). Next, each process computes the portion of the initial state it is responsible for (lines 5–8). The initial conditions can vary; for simplicity, we do not further specify the corresponding function `init` here. The processes with adjacent ranks are responsible for adjacent portions of the spatial domain. We will say that process A is to the left of process B if it is responsible for the portion of the spatial domain immediately to B's left on the x-axis.

Each process then simulates the equation on `u` for `nt` time steps (lines 12–32). At each time step, the process updates its portion of `u` according to Equation 3, which is implemented by function `upwind` in the code (lines 9–10). Each process performs the computation locally for all points it is responsible for (lines 15–16) except for the leftmost boundary value (`u[0]` in the code), which depends on the information from the neighboring process. The leftmost process 0 only needs to communicate the boundary value to the process on the right (lines 18–19), while the rightmost process $N-1$ only needs to receive one from the process on the left. (lines 21–22). Every other process both receives a boundary value from the process on the left (lines 28–29) and sends one to the process on the right (lines 25–27). Note that for efficiency, the code uses the *immediate*

send command (`Isend`), which allows the two operations to run concurrently—the immediate send operation is initialized before the receive but is not guaranteed to complete until after the `wait` call on line 31. Notice also that each message is tagged with a unique integer derived from the time step and rank. We use these tags as a key part of the verification, as discussed below. The other arguments shared by all send and receive operations are the rank of the destination or source process, respectively, and the buffer—passed by reference—from which the message is read (for sends) or to which it is written (for receives).

A round of communication completes with each process calling `Barrier` (line 32), which synchronizes all processes before the next round of communication. After `nt` iterations, the `Gather` command collects all segments of `u` into a single array `out` managed by process 0 (lines 33–35). Process 0 may then handle the output as needed, e.g., by saving it to disk or displaying a plot.

2.3 DafnyMPI

The goal of this paper is to develop a methodology for proving that MPI programs such as the one in Figure 2 are correct, by which we mean that they must terminate successfully with a result that is either identical to that of a matching sequential implementation or equivalent up to rounding differences caused by the non-associative nature of floating-point arithmetic and similar machine-level approximations. We treat sequential implementations as reference implementations since they are typically easier to write and are therefore less likely to contain bugs. They are also simpler to debug. Full program correctness, then, is contingent on the following properties:

- *Deadlock Freedom.* Any process waiting on a send (receive) is eventually unblocked by the matching receive (send), and any process waiting at a barrier eventually passes that barrier. We present a formal proof of deadlock freedom in Section 3.
- *Functional Equivalence.* The MPI program must compute the same result as a corresponding sequential implementation, which is manually written by the programmer and acts as a reference implementation. In our benchmarks, each process computes a portion of the final solution, and all such portions are then merged and returned by the root process (rank zero). DafnyMPI allows reasoning about the output of each process because it enforces the safe use of read and write buffers and ensures that all messages conform to the user-provided specification. We discuss these properties and the methodology for proving functional equivalence in Section 4.
- *Termination.* Every process will eventually terminate. Aside from deadlock freedom, which we list separately, termination requires the absence of infinite loops, endless recursion, and runtime errors such as division by zero. Dafny enforces all these properties by default, and so we use the standard techniques for proving them. Section 5 discusses termination and related proof-engineering considerations such as proof brittleness.

Dafny. We choose to prove these properties in Dafny, a verification-aware language that enforces termination and the absence of runtime errors, including division by zero, out-of-bounds array access, and null pointer dereference. Dafny automates much of the proof process by applying weakest-precondition calculus to generate verification conditions, which are then discharged to an SMT solver. The user of Dafny may write pre- and postconditions and invariants to guide the proof process and to enforce additional properties. In particular, functional equivalence between a pair of methods can be established transitively by requiring both to satisfy the same functional specification in their postconditions.

Thus, given a Python program, we manually translate it into an equivalent Dafny program. We then parallelize the computation by rewriting it with DafnyMPI and prove that the resulting version

```
c = MPI.COMM_WORLD
rank = c.Get_rank()
N    = c.Get_size()
D = MPI.DOUBLE
nt, dx = 25, 0.05
# u is the portion of the domain
# that the process is responsible for
u = init(rank, nx=40, dx=dx, size=N)
def upwind(u, u_1, c=1, dt=.025):
  return u - c * dt / dx * (u - u_1)

for n in range(nt):
  un = u.copy()
  bt = n * (N - 1)  # base tag
  for i in range(1, len(u)):
    u[i] = upwind(un[i], un[i - 1])
  if rank = 0:
    c.Send([un[-1:], D],
           dest=rank+1, tag=bt)
  elif rank = N - 1:
    c.Recv([u[0:1], D],
           src=rank-1, tag=bt+rank-1)
    u[0] = upwind(un[0], u[0])
  else:
    r = c.Isend([un[-1:], D],
                dest=rank+1,
                tag=bt+rank)
    c.Recv([u[0:1], D],
           src=rank-1, tag=bt+rank-1)
    u[0] = upwind(un[0], u[0])
    r.wait()
  c.Barrier()
out = zeros(nx, dtype='d') \
      if rank = 0 else u
c.Gather(u, out, root=0)
```

```
c := new MPI.World(
  N,    // size / # of processes
  Msg,  // (tag, N) ↦ ...,
  Src,  // (tag, N) ↦ tag % (N - 1)
  Dest, // (tag, N) ↦ tag % (N - 1) + 1,
  Clct, // (id,  N) ↦ if id < nt
  // then Barrier(tag=id * (N - 1))
  // else Gather(tag=nt * (N - 1)...)
  LastClct); // (N) ↦ nt + 1
tag, clct := -1, 0
```

```
assert rank-1 = Src(bt, N)
 ∧ rank = Dest(bt, N)
 ∧ NoBetween(tag, bt+rank-1)
 ∧ bt+rank-1 > tag
 ∧ CanWrite(u[0:1])
 ∧ Clct(clct, N).tag ≤ bt+rank-1
 ∧ bt+rank-1 < Clct(clct + 1, N).tag...
tag := bt+rank-1
assume u[0:1] = Msg(bt+rank-1, N)...
```

```
assert rank = Src(bt, N)
 ∧ rank+1 = Dest(bt, N)
 ∧ un[-1:] = Msg(bt, N)
 ∧ CanRead(un[-1:])
 ∧ NoBetween(tag, bt+rank)
 ∧ bt+rank > tag
 ∧ Clct(clct, N).tag ≤ bt+rank
 ∧ bt+rank < Clct(clct + 1, N).tag...
tag := bt+rank
```

```
assert Clct(clct + 1).tag > tag
 ∧ NoBetween(tag, Clct(clct + 1).tag)...
clct := clct + 1
assume clct=LastClct(N) ⟹ Done()...
```

Fig. 2. Example Python MPI program (left) and corresponding Dafny proof obligations (right, simplified). Proof obligations that deal with collective operations are grayed out to highlight the point-to-point communication.

is functionally equivalent to the original. After verification, both Dafny programs can be compiled back into Python for execution.

The key challenge with this approach is that Dafny is a single-threaded language, with no built-in notion of parallelism or concurrency. Thus, DafnyMPI is designed with the meta-property that, if the proof obligations of its MPI primitives are met, then the parallel execution of the program is guaranteed to satisfy the properties listed above.

DafnyMPI proof obligations for point-to-point communication. The right side of Figure 2 shows a selection of the DafnyMPI proof obligations, written as Dafny code, for our example, which we discuss next. Here and in Section 3, we focus on deadlock freedom, as that is the most intricate property to prove. In the figure, the correspondence between source code and proof obligations is shown by color. In one case, a program statement corresponds to multiple proof obligations, which is shown with appropriately colored stripes. We gray out proof obligations that deal with collective operations to keep the focus on the point-to-point communication for now; we return to collective operations later in this section.

At the start of the program, corresponding to line 1 on the left, the code creates a new `MPI.World` instance that must include programmer-supplied specifications of the overall communication pattern of the program. Specifically, the programmer provides the *size* `N`, the total number of processes—which, as we do here, can be left unconstrained to show the proof holds for any number of processes greater than one—and a list of functions describing the program's use of point-to-point messages and collective operations. The first three functions, `Msg`, `Src`, and `Dest`, describe point-to-point messages, mapping message tags and the size to the contents of the message, the message source, and the message destination, respectively. For our example, we omit the message contents for conciseness, and the source and destination are computed from the tag and the size. The next two functions, `Clct` and `LastClct`, describe the program's collective operations, including barriers. `LastClct` specifies the total number of collective operations in the program, which, in the general case, is a function of the number of processes. In our example, there are always exactly `nt` barriers—one per loop iteration (lines 12–32)—and one `Gather` call on line 35, so `LastClct` is constant and is set to `nt+1`. The `Clct` function maps each collective operation ID to its description, including the operation type (e.g., `Barrier` or `Gather`) and the lowest tag—across all processes—of any point-to-point message that immediately follows it. This allows us to order collective operations, which otherwise cannot be assigned tags, relative to point-to-point messages. Finally, the variable `tag` is initialized to -1 to track the tag of the last completed message, and `clct` is initialized to 0 to track the ID of the next upcoming collective operation.

Next, the code corresponding to the two `Recv` calls on lines 21–22 and 28–29 asserts that, in order, the source and destination of the message match the tag, no message was skipped (the `NoBetween(a,b)` predicate holds if and only if there is no tag between `a` and `b` for which the current process is the sender or the receiver), the current tag is greater than the one previously used, the destination buffer (`u[0:1]`) can be written to, and the tag falls between the previous and the next collective operations (in our case, barriers). Once the command executes, the process's `tag` counter is updated to the message's tag. Finally, the snippet assumes (the rely part of rely–guarantee reasoning) that `recv` returns the message `Msg(bt+rank-1,N)` corresponding to the tag in the specification.

The code corresponding to the `Send` call on lines 18–19 similarly begins by asserting that the source, destination, and contents of the message match the tag (the guarantee part of rely–guarantee reasoning), and that the source buffer is free to be read from. Next, the code enforces the same tag ordering properties as for the `Recv` call. After the send completes, the process's `tag` counter is incremented. Whenever a send is non-blocking, such as on lines 25–31, these different conditions are enforced at different points along the program's execution. The ordering properties on the tag are enforced at the corresponding `wait` call (line 31) that completes the send, while the rest of the conditions are checked immediately when the point-to-point communication is initialized (lines 25–27). Additional pre- and postconditions, not shown in the figure, ensure that the process does not send a duplicate message and that the send buffer is not written to between the calls to `ISend` and `wait`.

To understand why the conditions listed above prevent deadlock, suppose, for a contradiction, that deadlock does occur and that, among all the blocked processes, process P is blocked on an

MPI call with the lowest tag. The matching process for that message must not have initiated the communication with P on its own end yet, since otherwise P would eventually unblock. Therefore, the matching process must not have reached the point in the program at which it initiates communication with P. At the same time, there is a contradiction, because the matching process cannot be blocked at this point in the program, since then it would be the process blocked on the message with the lowest tag. Therefore, the matching process must be running and so there is no deadlock. (Section 3 formalizes this argument.)

Collective operations. The final block of code on the right side of Figure 2 deals with collective operations and corresponds to the `Barrier` call on line 32 and the `Gather` call on line 35. The code asserts that no barrier or collective operation was skipped and that the barrier is called only after any message that must precede it. The ordering of collective operations relative to point-to-point messages is additionally enforced on every call to `Recv`, `Send`, or `wait`, as shown by the grayed-out proof obligations in the relevant code snippets.

At each call to `Barrier` or `Gather`, the ID of the next collective operation is incremented. When this ID reaches `LastClct(N)`, Dafny can assume `Done()`, a special condition indicating that the program has completed its execution. The user of the DafnyMPI library must have a postcondition on the main method that checks that `Done()` is true. In our example, this condition is only fulfilled after the call to `Gather` on line 35. This call has additional pre- and postconditions attached to it (not shown in the figure) that ensure the inputs conform to the specification provided by the user, e.g., that `u` holds part of the output corresponding to the process's `rank`.

Together, these conditions guarantee deadlock freedom and allow the user to prove that the variable `out` will, for the zeroth process, contain the solution to the PDE after `nt` time steps. The next section formalizes this approach for a core calculus where we prove that these conditions are sufficient.

3 Formalizing DafnyMPI

In this section, we formalize DafnyMPI on a core calculus and prove that for a given program, if DafnyMPI's requirements are satisfied, then that program will be deadlock-free when run in parallel. Our core calculus adds several primitives from the MPI 1.0 standard [25] to a language of commands. As far as we are aware, later MPI standards maintain the semantics of these primitives, so our approach would still apply. Moreover, in our experience, most MPI programs are written in a way that does not require the more advanced features of later standards, such as MPI 2.0 or MPI 3.0, which introduce additional features such as one-sided communication and dynamic process management.

DafnyMPI directly encodes the following four MPI functions:

- `MPI_IRECV` (*immediate receive* or *non-blocking receive*) requests to receive a message with a given tag from a given source. The operation returns immediately, and the message may then be received in the background. The operation returns an `MPI_REQUEST` object that can be waited on to ensure that the message has been received.
- `MPI_ISEND` (*immediate send* or *non-blocking send*) requests to send a message with a given tag to a given destination. The operation returns immediately with an `MPI_REQUEST` object that can be waited on to ensure the send has been completed. Note that completion is local in this context: it does not guarantee that the message is received at the destination, only that the sender can reuse the buffers associated with the operation.
- `MPI_WAIT` (*wait*), called on an `MPI_REQUEST` object, blocks a process until the corresponding communication has been completed.
- `MPI_BARRIER` (*barrier*) blocks until all processes also reach a barrier.

In Section 5, we discuss how we can extend the core calculus to support MPI functions that are available in DafnyMPI, using syntactic sugar or as simple variations of the initial four.

- `MPI_RECV` (*blocking receive*) can be encoded as an `MPI_IRECV` followed immediately by the relevant `MPI_WAIT`.
- `MPI_SEND` (*blocking send*) can be encoded similarly with `MPI_ISEND` and `MPI_WAIT`.
- `MPI_GATHER` must be called by all processes, after which the data associated with each process is transferred to the designated root process and stored in a single array. We encode this operation as a decorated barrier call, requiring that all processes are synced before exchanging information in this way.
- `MPI_ALLREDUCE` must be called by all processes, after which the data submitted by each process is folded using a binary operation such as summation or logical and. The result is then communicated back to all processes. This operation is encoded as a decorated barrier similarly to `MPI_GATHER`.

DafnyMPI does not currently support MPI *communicators*, which group processes together so that, e.g., only a group needs to pass a barrier, or *wildcards*, which allow one process to exchange a message with an arbitrary process associated with the same communicator. We leave support for these features to future work. Instead, DafnyMPI programs use the single default `MPI_COMM_WORLD` communicator on all operations.

3.1 The Core Calculus

Figure 3 shows the syntax of our core calculus, which is stratified into expressions and commands. As our proof focuses on interprocess communication, expressions e are limited to integers n, variables x, comparisons $e = e$, and basic arithmetic operations $e + e$, $e * e$, e **div** e, $-e$. Because there is no shared memory, each process has its own variable environment, and there is no way for one process to affect the evaluation of an expression by another process. However, an expression may not read a variable that is being written by a background receive call (see Section 3.3).

Commands c include four communication primitives corresponding to the MPI functions listed above: **irecv** $e\ x$, non-blocking receive of a message with tag e, where the message is written to x; **isend** $e\ x$, non-blocking send of a message with tag e, where the message is read from x; **wait** e, waiting for completion of the send or receive with tag e; and **barrier**, which blocks until all processes reach a barrier. In all cases, tags are integers. The remaining commands—**skip**, **if**, **while**, **set**, and sequencing—have the standard semantics, where zero is considered false and non-zero integers are considered true.

Our semantics models the state of a single process as a tuple $\langle c, \sigma, t, b\rangle$, where c is the command to be executed, σ is the variable environment mapping variables to integers, t is the largest tag previously used in a **wait** call, and b is the number of barriers the process has passed so far. Then the global program state S, encompassing all processes, is a tuple $(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)$, where $\mathcal{P}$ is a set of process states, and $\mathcal{B}_r$, $\mathcal{B}_s$, and $\mathcal{B}_m$ are the global receive, send, and message buffers, respectively. Throughout this section, we will also use $N = |\mathcal{P}|$ to refer to the number of processes.

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid e = e \mid e + e \mid -e \\
 & \mid & e * e \mid e \textbf{ div } e \\
c & ::= & \textbf{irecv } e\ x \mid \textbf{isend } e\ x \\
 & \mid & \textbf{wait } e \mid \textbf{barrier} \\
 & \mid & \textbf{skip} \mid \textbf{if } e\ c\ c \\
 & \mid & \textbf{while } e\ c \mid \textbf{set } x\ e \mid c\ \textbf{;}\ c
\end{array}
$$

Fig. 3. Core calculus syntax.

The receive and send buffers are maps from message tags to variable names, indicating the destination (for receives) or source (for sends) of the message within the process environment. The message buffer is a map from message tags to message payload, which is the value that would eventually be copied from the source variable in the sender's environment to the destination

variable in the receiver's environment. This intermediate payload buffer is necessary because MPI allows a send operation to complete before the corresponding receive is called, in which case the message contents must be buffered internally, while the sender is allowed to reuse the source variable. It is sufficient to model the buffers globally because the programmer is required to provide functions that uniquely identify the sender and receiver for each message tag as part of the program's specification. Section 3.2 goes through a concrete example of how the buffers are used.

Our operational semantics is nondeterministic and may take a step in any process that is not blocked. Thus, we treat the running processes as a set and use the set notation for convenience. In particular, we write $\langle c, \sigma, t, b\rangle \cup \mathcal{P}$ (omitting curly braces around the singleton set) to select one process $\langle c, \sigma, t, b\rangle$ with the remaining processes as $\mathcal{P}$, and we write $\mathcal{P}_1 \cup \mathcal{P}_2 \cup \ldots$ to nondeterministically split the set of processes into subsets. We store the process rank by binding an integer to `rank` in σ. As a shorthand, we use a subscript on the process state to indicate rank, i.e., $\langle c, \sigma, t, b\rangle_i$ indicates a state where $\sigma(\texttt{rank}) = i$. We omit the subscript when the rank is irrelevant.

A state reduction is a sequence of states $S_0, S_1, \ldots, S_k$ such that each preceding state transitions to the next one according to the operational semantics (Section 3.2). As another shorthand, we sometimes refer to the state components of the ith process during the kth reduction step by subscripting with i, k, e.g., $\sigma_{2,0}$ refers to the variable environment of the process with rank 2 in the initial state S_0.

DafnyMPI programs are *single instruction, multiple data* (SIMD), with the same command c executed on all nodes. The initial state of the execution of command c on N nodes is given by:

$$S_0 = (\bigcup_{0 \le i < N} \langle c;\ \textbf{barrier}, \{\texttt{rank} \mapsto i, \texttt{size} \mapsto N\}, -1, 0\rangle, \emptyset, \emptyset, \emptyset)$$

Notice the processes are initialized with variables `rank` and `size` to let the process know its rank and the total number N of processes running. Each process is also initialized with the exact same sequence of commands to execute, where the last command must be a **barrier**.

We verify a given program with respect to a user-provided specification, where, as in Section 2, the user must describe the overall communication pattern of the program ahead of time. Specifically, the user provides five functions below, which we will refer to throughout this section and which together describe all MPI operations. Note that all five functions are parameterized only by process-independent quantities, so any two processes will always compute the same values:

- Sender and Receiver are total functions that map a message tag and the number of processes to the rank of the process that sends or receives a message with that tag. In our running example in Figure 2, these correspond to functions `Src` and `Dest`, respectively.
- Message (`Msg` in Figure 2) is a total function mapping a message tag and the number of processes to the message payload, which in our core calculus semantics may only be an integer.
- BarrierCount (`LastClct` in Figure 2) maps the number of processes to the total number of barriers a process must pass before terminating.
- Barrier is a total function that maps the rank of a barrier and the number of processes to a message tag, indicating that any message with a smaller tag must be sent and received before the process reaches the barrier. This function roughly corresponds to function `Clct` in Figure 2, although DafnyMPI supports collective operations not explicitly modeled by the core calculus, as we describe in Section 5.

3.2 Operational Semantics

This section presents the small-step operational semantics of our core calculus. The goal is to model the behavior of message-passing programs as closely as possible to the MPI standard, so

that the proof of deadlock freedom that follows applies in all possible situations. One design choice that distinguishes our semantics from much of prior work is that we allow send operations to block nondeterministically. In prior work (e.g., [21]), send operations are typically assumed to be non-blocking, which simplifies reasoning but does not capture the scenario, easily triggered by large message payloads, in which the MPI implementation temporarily runs out of internal buffer space. To model this behavior, our semantics allows a message to be transferred from the sender to the shared message buffer $\mathcal{B}_m$ in one of two ways: (i) eagerly, before the sender reaches a **wait** call (representing a normal immediate send), or (ii) lazily, only when the receiver executes its own **wait** (representing an immediate send that blocks). The remainder of this section describes these rules.

Figure 4 gives the small-step operational semantics, divided into two groups: (a) local rules that touch a single process, and (b) global rules that also manipulate shared buffers or several processes at once. Our semantics also includes big-step expression evaluation, $\langle e, \sigma \rangle \Downarrow v$, meaning evaluating expression e in variable environment σ yields value v. As expression evaluation is entirely standard, we omit these rules.

The local rules, shown in Figure 4a, define the single stepping relation $\rightsquigarrow$ on process states $\langle c, \sigma, t, b \rangle$ meaning taking one step in the process whose state is on the left yields the state on the right. IfTrue and IfFalse cover the two cases for conditionals, and While expands one step of the iteration. Finally, Set updates the variable environment appropriately, and SeqSkip removes a **skip** at the start of a sequence. A process whose command is a lone **skip** is considered to have terminated, and thus there is no rule to evaluate it further.

The global rules, shown in Figure 4b, define the single stepping relation $\longrightarrow$ on program states $(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)$. Importantly, these rules omit several premises that would typically be checked dynamically but instead are guaranteed to hold by the Dafny verifier. For example, the Send rule does not verify that the sender's rank matches the tag's designated sender, because this is checked as part of using DafnyMPI. We formalize these preconditions as *axioms* in Section 3.3, below.

The first two reduction rules are administrative: Proc lifts a single-process reduction to the reduction of the entire program state, and SeqStep takes a step in the first command of a sequence. Notice that both rules use set notation to pick a process nondeterministically.

Most of the remaining rules model send operations, and the main challenge lies in their nondeterministic behavior. Depending on the implementation and message size, the MPI runtime may copy the payload into an internal buffer, allowing the sender's subsequent **wait** to complete immediately. Alternatively, the process may block until a matching receive is called. Our operational semantics aims to mimic both of these behaviors.

We use the example in Figure 5 to illustrate the communication rules as we describe them. The figure shows two of the possible partial reduction sequences of a simple program. In the initial state (State 0 in the figure), the process with rank 0 is poised to initiate a non-blocking send for a message with tag t, and the process with rank 1 is poised to initiate a non-blocking receive for the same message. Either command may fire first, leading to State 1 or State 6, respectively.

Rule Send adds a request to send x as a message under the appropriate tag t'. The request is registered by placing x in the send buffer $\mathcal{B}_s$. Similarly, Recv adds a request to receive x with tag t'. As described above, these operations are non-blocking. In our example, we label the state transitions with S (send) and R (receive), and we see the appropriate updates to $\mathcal{B}_s$ and $\mathcal{B}_r$, respectively.

Continuing along the top of the figure, the transition from State 1 to State 2 illustrates TransferNoWait, which, given a sender process that is not waiting or at a barrier (recall c_i indicates the command of the ith process), copies the contents of the to-be-sent message from x to $\mathcal{B}_m$ with the appropriate tag t'. In the example, the message payload is 5, and thus $\mathcal{B}_m$ is updated to map t to 5. This behavior captures the case when the MPI implementation copies a message payload into a buffer, allowing the sender process to unblock before the matching receive is even initiated.

$$\frac{\langle e, \sigma\rangle \Downarrow v \quad v \neq 0}{\langle \textbf{if } e\ c_1\ c_2, \sigma, t, b\rangle \rightsquigarrow \langle c_1, \sigma, t, b\rangle}\ \textsc{IfTrue} \qquad \frac{\langle e, \sigma\rangle \Downarrow 0}{\langle \textbf{if } e\ c_1\ c_2, \sigma, t, b\rangle \rightsquigarrow \langle c_2, \sigma, t, b\rangle}\ \textsc{IfFalse}$$

$$\frac{}{\langle \textbf{while } e\ c, \sigma, t, b\rangle \rightsquigarrow \langle \textbf{if } e\ (c;\ \textbf{while } e\ c)\ \textbf{skip}, \sigma, t, b\rangle}\ \textsc{While}$$

$$\frac{\langle e, \sigma\rangle \Downarrow v}{\langle \textbf{set } x\ e, \sigma, t, b\rangle \rightsquigarrow \langle \textbf{skip}, \sigma[x \mapsto v], t, b\rangle}\ \textsc{Set} \qquad \frac{}{\langle \textbf{skip};\ c, \sigma, t, b\rangle \rightsquigarrow \langle c, \sigma, t, b\rangle}\ \textsc{SeqSkip}$$

(a) Single-Process Operational Semantics Rules.

$$\frac{\langle c, \sigma, t, b\rangle \rightsquigarrow \langle c', \sigma', t', b'\rangle}{(\langle c, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle c', \sigma', t', b'\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{Proc}$$

$$\frac{(\langle c_1, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle c', \sigma', t', b'\rangle \cup \mathcal{P}, \mathcal{B}'_r, \mathcal{B}'_s, \mathcal{B}'_m)}{(\langle c_1; c_2, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle c'; c_2, \sigma', t', b'\rangle \cup \mathcal{P}, \mathcal{B}'_r, \mathcal{B}'_s, \mathcal{B}'_m)}\ \textsc{SeqStep}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t'}{(\langle \textbf{isend } e\ x, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s[t' \mapsto x], \mathcal{B}_m)}\ \textsc{Send}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t'}{(\langle \textbf{irecv } e\ x, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r[t' \mapsto x], \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{Recv}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t' \quad t' \in \mathcal{B}_r \quad t' \in \mathcal{B}_m}{(\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma[\mathcal{B}_r(t') \mapsto \mathcal{B}_m(t')], t', b\rangle_i \cup \mathcal{P}, \mathcal{B}_r \setminus \{t'\}, \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{WaitRecv}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t' \quad t' \in \mathcal{B}_m}{(\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma, t', b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s \setminus \{t'\}, \mathcal{B}_m)}\ \textsc{WaitSend}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t' \quad \textsc{Receiver}(t', N) = i \quad t' \in \mathcal{B}_s \quad \mathcal{B}_s(t') = x \quad \langle x, \sigma_{\textsc{Sender}(t', N)}\rangle \Downarrow v}{(\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m[t' \mapsto v])}\ \textsc{TransferOnWait}$$

$$\frac{t' \in \mathcal{B}_s \quad i = \textsc{Sender}(t', N) \quad c_i \neq \textbf{wait } e; c' \quad c_i \neq \textbf{barrier}; c' \quad \mathcal{B}_s(t') = x \quad \langle x, \sigma_i\rangle \Downarrow v}{(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m[t' \mapsto v])}\ \textsc{TransferNoWait}$$

$$\frac{i = \textsc{Sender}(t', N) \quad j = \textsc{Receiver}(t', N) \quad t_i \geq t' \quad t_j \geq t' \quad t' \in \mathcal{B}_m}{(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m \setminus \{t'\})}\ \textsc{FreeBuffer}$$

$$\frac{}{(\bigcup_{0 \leq i < N} \langle \textbf{barrier}; c, \sigma, t, b\rangle_i, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\bigcup_{0 \leq i < N} \langle \textbf{skip}; c, \sigma, t, b+1\rangle_i, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{Barrier}$$

(b) Multi-Process Operational Semantics Rules.

Fig. 4. Core calculus reduction rules.

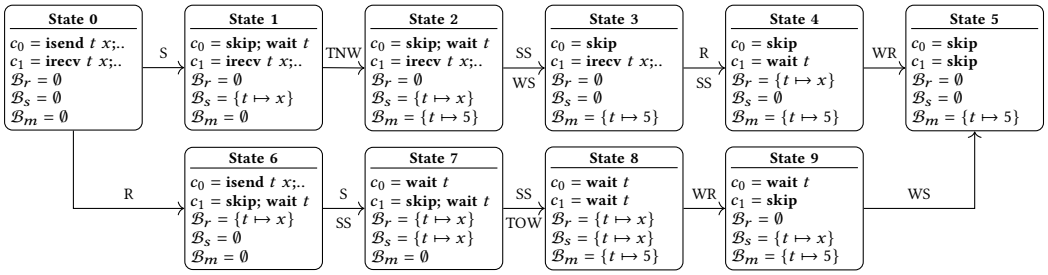

Fig. 5. Two partial reductions for the program **set** x 5; **if** rank = 0 { **isend** t x } { **irecv** t x }; **wait** t. The program is executed by two processes; edges are labeled with abbreviations of reduction rules (SS stands for SeqSkip, etc.). The reduction rule above the line is taken first.

Once the message is copied into the message payload buffer, the process with rank 0 is now ready to complete the send, as captured by the transition from State 2 to State 3 via WaitSend. The rule checks that the message with tag t' has been placed in $\mathcal{B}_m$ and then removes the tag from the set of pending sends. In our example, we see that this rule can fire because of the previous execution of TransferNoWait, and therefore tag t is removed from $\mathcal{B}_s$.

The transition from State 3 to State 4 uses Recv to add a request to receive the message with tag t, following which the WaitRecv can fire to move from State 4 to State 5. Analogously to WaitSend, WaitRecv can execute when a message with the given tag is in the message buffer, and it updates the variable to be written with the message payload, removing the tag from $\mathcal{B}_r$ in the process. Although it is not shown in the figure, x has been updated in c_1 to contain 5.

In contrast, the bottom row illustrates the case when an MPI send blocks until the matching receive is executed. Specifically, after the Send rule is used to transition from State 6 to State 7, the process with rank 0 cannot proceed any further. The WaitSend rule cannot fire because the message payload buffer $\mathcal{B}_m$ is empty and also the TransferNoWait rule cannot fire because the process is already blocked on a wait. The only rule that can unblock process with rank 0 is, therefore, TransferOnWait, which can fire whenever the receiving process with rank 1 reaches the corresponding wait call. This is illustrated with a transition from State 7 to State 8, where the process with rank 1 first gets to the **wait** call via the SeqStep rule and then initiates the transfer via the TransferOnWait rule. Subsequently, WaitRecv transitions the system from State 8 to State 9, and WaitSend completes the communication by advancing to State 5.

Note that in State 5, the final state in Figure 5, the message buffer $\mathcal{B}_m$ still contains the message tagged t. Once the communication associated with the tag has been completed, it may be removed from $\mathcal{B}_m$ at any point according to the FreeBuffer rule. The rule mimics the expected behavior of MPI runtime in that any memory used internally by the MPI should eventually be freed. The exact timing is implementation-dependent and irrelevant to our proof.

The last rule of our operational semantics, Barrier, specifies that all processes must execute a barrier simultaneously. Note that we adopt a slight abuse of notation by allowing the Barrier rule to apply to the final barrier, even though the rule formally presupposes the existence of a subsequent command.

3.3 Axioms

Our core calculus represents programs that a DafnyMPI user could write. For Dafny to verify these programs, the user must prove a number of properties. These include properties Dafny enforces

Table 1. Axioms of the core calculus (implicitly parameterized by program state, except for AtStart).

Axiom	Premises	Conclusions
AtStart		$S_0 = (\bigcup_{0 \le i < N} \langle c;\ \textbf{barrier}, \{\texttt{rank} \mapsto i, \texttt{size} \mapsto N\}, -1, 0\rangle, \emptyset, \emptyset, \emptyset)$
AtSend	$c_{i,k} = \textbf{isend}\ e\ x; c$ $\langle e, \sigma_{i,k}\rangle \Downarrow v$	$i = \textsc{Sender}(v, N) \quad \langle x, \sigma_{i,k}\rangle \Downarrow \textsc{Message}(v) \quad v \notin \mathcal{B}_{s,k}$
AtRecv	$c_{i,k} = \textbf{irecv}\ e\ x; c$ $\langle e, \sigma_{i,k}\rangle \Downarrow v$	$i = \textsc{Receiver}(v, N) \quad v \notin \mathcal{B}_{r,k}$
AtWait	$c_{i,k} = \textbf{wait}\ e; c$ $\langle e, \sigma_{i,k}\rangle \Downarrow v$	$v > t_{i,k}$ $\forall v'.\ t_{i,k} < v' < v \Rightarrow (\textsc{Sender}(v', N) \neq i \wedge \textsc{Receiver}(v', N) \neq i)$ $\textsc{Barrier}(b_{i,k}, N) \le v < \textsc{Barrier}(b_{i,k}+1, N)$ $(i = \textsc{Sender}(v, N) \wedge v \in \mathcal{B}_{s,k}) \vee (i = \textsc{Receiver}(v, N) \wedge v \in \mathcal{B}_{r,k})$
AtBarrier	$c_{i,k} = \textbf{barrier}; c$ $v = \textsc{Barrier}(b_{i,k}+1, N)$	$v > t_{i,k}$ $\forall v'.\ t_{i,k} < v' < v \Rightarrow (\textsc{Sender}(v', N) \neq i \wedge \textsc{Receiver}(v', N) \neq i)$ $\forall v.\ \textsc{Sender}(v, N) = i \Rightarrow v \notin \mathcal{B}_{s,k}$ $\forall v.\ \textsc{Receiver}(v, N) = i \Rightarrow v \notin \mathcal{B}_{r,k}$
AtEnd	$c_{i,k} = \textbf{skip}$	$b_{i,k} = \textsc{BarrierCount}(N)$
AtSet	$c_{i,k} = \textbf{set}\ x\ e; c$	$\forall v.\ \textsc{Sender}(v, N) = i \Rightarrow (v \notin \mathcal{B}_{s,k} \vee x \neq \mathcal{B}_{s,k}(v))$ $\forall v.\ \textsc{Receiver}(v, N) = i \Rightarrow (v \notin \mathcal{B}_{r,k} \vee x \neq \mathcal{B}_{r,k}(v))$ $x \neq \texttt{rank}$
AtRead	$\textsc{Receiver}(v, N) = i$ $\mathcal{B}_{r,k}(v) = x$	$x \notin$ direct subexpressions of $c_{i,k}$

by default, such as loop termination, absence of division by zero, and absence of references to undeclared variables. Several other properties are MPI-specific and are expressed as preconditions on the relevant DafnyMPI methods. We refer to these properties as *axioms*, since in developing our proof, we can assume they hold for any execution as they are discharged by the Dafny verifier. Each axiom other than AtStart is implicitly parameterized by a particular program state: for any state reachable from the initial state, if the axiom's premises hold, then so do its conclusions.

It is important to note that Dafny only models the behavior of a single (arbitrary) process and can only inspect the part of the program state that the process has access to. As a result, none of the axioms mentions the message buffer $\mathcal{B}_m$, whose contents is internal to the MPI runtime. Similarly, the axioms only check whether a given tag is present in the send or receive buffers when the process to which the axiom applies is the intended sender or receiver. Additionally, because the Dafny verifier assumes all method calls terminate, it cannot itself account for the possibility of deadlock. DafnyMPI builds on top of this foundation to establish deadlock freedom explicitly. At the same time, Dafny's non-blocking assumption guarantees that the axioms hold in any state reachable under that assumption. This allows us to reason about a program state by considering what must hold in future states that are guaranteed to be reached. In particular, because Dafny proves that the final barrier count for any process is equal to $\textsc{BarrierCount}(N)$, we can show that the barrier count never exceeds this value, which is one of the lemmas introduced in Section 3.5.

The axioms should also be understood in conjunction with the reduction rules presented above. For example, we say that the AtWait axiom in Table 1 guarantees that tags increase monotonically but this is only true when we consider the axiom in conjunction with the WaitRecv and WaitSend reduction rules in Figure 4. Specifically, the two reduction rules always set the new tag $t_{i,k+1}$ to match the argument of the **wait** call, which, in turn, is guaranteed to be greater than the previous tag $t_{i,k}$ by the AtWait axiom.

Table 1 presents the axioms of our language. The AtStart axiom describes the initial state of the program as outlined above in Section 3.1. AtSend and AtRead state that a message with a

Table 2. Definition of deadlock and related terms.

Term	Conditions
$\textsc{OnSend}((\langle \textbf{wait } e; c', \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m))$	$\langle e, \sigma\rangle \Downarrow t' \quad i = \textsc{Sender}(t', N) \quad t' \notin \mathcal{B}_m$
$\textsc{OnRecv}((\langle \textbf{wait } e; c', \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m))$	$\langle e, \sigma\rangle \Downarrow t' \quad i = \textsc{Receiver}(t', N) \quad t' \notin \mathcal{B}_m \quad t' \notin \mathcal{B}_s$
$\textsc{OnBarrier}(\langle \textbf{barrier; } c', \sigma, t, b\rangle)$	
$\textsc{Terminated}(\langle \textbf{skip}, \sigma, t, b\rangle)$	
$\textsc{Deadlock}((\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_4, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m))$	$\lvert\mathcal{P}_3\rvert \neq N \quad \lvert\mathcal{P}_4\rvert \neq N$ $\textsc{OnRecv}(p, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m),\ \forall p \in \mathcal{P}_1$ $\textsc{OnSend}(p, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m),\ \forall p \in \mathcal{P}_2$ $\textsc{OnBarrier}(p),\ \forall p \in \mathcal{P}_3$ $\textsc{Terminated}(p),\ \forall p \in \mathcal{P}_4$

given tag can only be sent or received by its designated Sender and Receiver. Both axioms also require that the corresponding buffers are empty to disallow repeatedly sending or receiving the same message before calling **wait**. Additionally, AtSend guarantees that the message payload is as specified by the Message function.

The AtWait axiom enforces the tag ordering property. First, it guarantees that message tags used with **wait** increase monotonically for each process. Second, it enforces that no **wait** call is skipped—if the process is the designated receiver or sender of a message, it must receive or send this message before moving on to messages with higher tags. Third, the axiom ensures that no barriers are skipped between successive **wait** calls. Finally, it ensures that **wait** is only called after the corresponding **isend** or **irecv**.

AtBarrier requires that processes call barriers in order and wait on all send and receive operations that should precede them. Similar to the Barrier rule above, the AtBarrier axiom should be understood to apply to the final barrier as well. The axiom also guarantees that the send and receive buffers are empty at the barrier, i.e., there may be no ongoing send or receive calls. Because the last command executed by every process is a barrier, this last requirement guarantees that every **irecv** and **isend** is eventually followed by a matching **wait**. At the end of each process's execution, the AtEnd axiom states that the process passed exactly $\textsc{BarrierCount}(N)$ barriers.

Finally, AtSet and AtRead prevent nondeterministic behavior by disallowing reads or writes to variables that are used in interprocess communication. We explain in Section 4 how these two axioms are enforced by Dafny in practice when discussing the safe use of read and write buffers. AtSet also prevents changes to variable `rank`, which means we can rely on it always being equal to the process's actual rank in the operational semantics.

3.4 Deadlock

Table 2 introduces the terminology used to formally define and reason about deadlock. We begin by listing the conditions under which we consider a process to be blocked:

- A process is blocked on a send if it is about to execute a **wait** for a message it sends, but the message has not yet been placed in the payload buffer. The sender remains blocked until the receiver executes the corresponding **wait**. (This condition is captured by the OnSend predicate in the table.)
- A process is blocked on a receive if it is about to execute a **wait** command for a message it receives, but neither the message payload buffer nor the send buffer contains the relevant tag. This means the sender has not yet initiated the communication. (This condition is defined by the OnRecv predicate in the table.)

- A process may be blocked on a barrier if it is about to execute the **barrier** command, as defined by the OnBarrier predicate in the table. As long as not all other processes have reached the barrier, the process will remain blocked.

With these definitions in place, we can now formally define deadlock. A deadlock is any state, other than the one in which all processes have terminated, where no reduction rule applies. Note that we exclude the FreeBuffer rule from this condition because any state where only FreeBuffer applies eventually leads to a deadlock. One can see that this definition of deadlock is equivalent to every process being blocked or terminated, except when all processes have terminated. This is captured by the definition of Deadlock in Table 2. All processes in a deadlocked state must belong to one of four groups: those blocked while waiting for a receive or send operation to complete ($\mathcal{P}_1$ and $\mathcal{P}_2$ in the table), those blocked on a **barrier** ($\mathcal{P}_3$), and those that have terminated ($\mathcal{P}_4$). Additionally, it is not possible for all processes to belong to $\mathcal{P}_4$ (because then the program would have terminated) or $\mathcal{P}_3$ (because then the processes would pass the barrier). Note that the OnRecv and OnSend predicates specifically rule out states where WaitSend, WaitRecv, TransferOnWait, or TransferNoWait could be applied. With deadlock formally defined, we now proceed to prove deadlock freedom.

3.5 The Deadlock Freedom Proof

In this section, we provide an outline of the proof that our core calculus does not admit deadlocks. The proof proceeds by selecting the process waiting on the message with the smallest tag and showing that its communication partner cannot be blocked, since the ordering on message tags would only permit blocking on a smaller tag. The main theorem relies on a number of helper lemmas, which we list without proof in the interest of conciseness. Both the lemmas and the theorem assume the axioms described in Section 3.3 hold, including the description of the initial program state. The complete proof is included in the supplementary material.

Lemma 1 (NoTwoTagsTheSame)**.** *If two processes are about to execute* **wait** *commands in a deadlocked state, the corresponding message tags must be different.*

Lemma 2 (NonEmptyBuffer)**.** *If a process is responsible for sending or receiving a message with tag $v \in \mathbb{N}$, and that process's max used tag value eventually reaches v, then at some point during execution, the message buffer must have contained the payload tagged v.*

Lemma 3 (BufferStaysNonEmpty)**.** *If the process is the intended receiver or sender of a message in the payload buffer but the message's tag is higher than that of any message the process has received or sent so far, then the message must remain in the buffer.*

Lemma 4 (NumberOfBarriers)**.** *If the process has a barrier at the end of its command sequence, its internal barrier counter is strictly less than BarrierCount(N).*

Theorem 1 (Deadlock Freedom)**.** There exists no reduction from the initial state S_0 to a deadlocked state.

Proof. Suppose, for the sake of contradiction, that there exists a reduction $S_0 \longrightarrow \cdots \longrightarrow S_k$ and Deadlock(S_k). By the definition of Deadlock, there must then exist some $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4, \mathcal{B}_s, \mathcal{B}_r$, and $\mathcal{B}_m$ such that $S_k = (\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_4, \mathcal{B}_s, \mathcal{B}_r, \mathcal{B}_m)$ and $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4$ have the properties described in Table 2. There are two cases, either $\mathcal{P}_1 \cup \mathcal{P}_2 = \emptyset$ or $\mathcal{P}_1 \cup \mathcal{P}_2 \neq \emptyset$.

- **Case 1**. Suppose $\mathcal{P}_1 \cup \mathcal{P}_2 = \emptyset$. This means that all processes that are still running are waiting on some barrier. By the definition of Deadlock, $|\mathcal{P}_3| \neq N$. In other words, there must be at least one process that has terminated. Let that process be p_m with rank m. Because this process has terminated, the AtEnd axiom applies, meaning that $b_{m,k} = \text{BarrierCount}(N)$.

Note that all processes must always share the same b, since all processes pass barriers simultaneously and all processes start with $b = 0$ according to the AtStart axiom. However, we know that at least one process, let us call it $p_i \in \mathcal{P}_3$, has not terminated yet. By the NumberOfBarriers Lemma, we know that $b_{i,k} < \text{BarrierCount}(N)$. However, we also know that $b_{i,k} = b_{m,k} = \text{BarrierCount}(N)$. Hence, we reached a contradiction. Therefore, $\mathcal{P}_1 \cup \mathcal{P}_2 \neq \emptyset$.

- **Case 2**. Suppose $\mathcal{P}_1 \cup \mathcal{P}_2 \neq \emptyset$. This means that at least one process must be blocked on a **wait** call for which it is the receiver or sender. Let m be the rank of the process that is waiting on a message with the smallest tag, namely that $c_{m,k} = \textbf{wait}\ e_m\textbf{;}\ c'$ and $\langle e_m, \sigma_{m,k}\rangle \Downarrow v_m$, where v_m is smaller than the respective value for any other process in $\mathcal{P}_1 \cup \mathcal{P}_2$. Such a minimum exists because, according to NoTwoTagsTheSame lemma, no two processes may wait on the same tag. We will now assume that $p_m \in \mathcal{P}_1$ (the reasoning is symmetric when $p_m \in \mathcal{P}_2$.) Because $p_m \in \mathcal{P}_1$, we know that $m = \text{Receiver}(v_m, N)$.
  Now consider the process with rank $i = \text{Sender}(v_m, N)$. We will now show that $t_{i,k} \geq v_m$. We know that $p_{i,k} \in \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_4$. So there are three subcases:
  - Subcase 1: $p_i \in \mathcal{P}_1 \cup \mathcal{P}_2$. Then $c_{i,k} = \textbf{wait}\ e_i\textbf{;}\ c'$ and $\langle e_i, \sigma_{i,k}\rangle \Downarrow v_i$. By construction, we know that $v_i > v_m$. By the AtWait axiom, we also know that $v_i > t_{i,k}$. By the same axiom, it must be the case that $t_{i,k} \geq v_m$ since there must be no tag between v_i and $t_{i,k}$ for which i is the sender.
  - Subcase 2: $p_i \in \mathcal{P}_3$. By the AtBarrier axiom applied to process i in state k, it follows that $\text{Barrier}(b_{i,k} + 1, N) > t_{i,k}$. By the AtWait axiom applied to process m in state k, $v_m < \text{Barrier}(b_{m,k}+1, N)$. As discussed above, all processes have the same value of b in any given state. Hence, we have $b_{i,k} = b_{m,k}$ and $v_m < \text{Barrier}(b_{i,k} + 1, N)$. By the AtBarrier axiom applied to process i in state k, there must be no tag between $t_{i,k}$ and $\text{Barrier}(b_{i,k} + 1, N)$ for which i is the sender, so, therefore, $t_{i,k} \geq v_m$.
  - Subcase 3: $p_i \in \mathcal{P}_4$ meaning that p_i has terminated. Note that by the AtStart axiom, p_m has a **barrier** command at the end of its command sequence. Since p_m has not passed this barrier yet, the NumberOfBarriers lemma applies, and $b_{m,k} < \text{BarrierCount}(N)$. At the same time, the AtEnd axiom applies to the process with rank i, so $b_{i,k} = \text{BarrierCount}(N)$. However, this leads to a contradiction because, as discussed above, all processes have the same value of b in any given state, so $b_{i,k} = b_{m,k}$. Therefore, Subcase 3 is not possible.

  We have now established that $t_{i,k} \geq v_m$. Consider also that by the AtWait axiom, $v_m > t_{m,k}$. Since $t_{i,k} \geq v_m > t_{m,k}$, we can invoke the NonEmptyBuffer lemma for process i and the BufferStaysNonEmpty lemma for process m. This gives us that $v_m \in B_{m,k}$. However, this is a contradiction because the Deadlock predicate requires that $v_m \notin B_{m,k}$. Therefore, $\mathcal{P}_1 \cup \mathcal{P}_2 = \emptyset$.

We have now shown that both cases lead to a contradiction. Therefore, our initial assumption must be wrong and Deadlock is not reachable. □

The proof above connects the behavior described by the MPI standard—formalized as the operational semantics of the core calculus—with the specifications we write in Dafny—formalized as the axioms of the core calculus—to establish that the specifications prevent deadlock in programs that use DafnyMPI. The only other instance of interprocess reasoning necessary to establish full program correctness concerns the contents of the messages, which we discuss next.

4 Proving Functional Equivalence

Our approach guarantees that a program written using DafnyMPI will never deadlock, but it does not by itself guard against domain-specific errors. At the same time, it is not unusual for software

engineers to make such mistakes when writing concurrent code. Therefore, to further ensure the correctness of DafnyMPI programs, we propose a method for establishing functional equivalence between a parallel program written using DafnyMPI and a sequential program written in plain Dafny. We follow this method to prove functional correctness of all three benchmarks described in Section 6. Specifically, we prove that the zeroth process—by convention the one that aggregates the results of all processes at the end—produces the same result as the sequential version. The method consists of the following key steps:

- First, the programmer writes in Dafny a sequential version of the computation they wish to parallelize. Sequential code is typically much easier to understand, reducing the likelihood of domain-specific bugs at this stage.
- Next, the programmer writes a ghost functional specification of the program and proves that the sequential version returns the same result as the functional specification for all inputs. The functional specification is typically more difficult to understand since all loops have to be replaced with recursion. This proof is reasonably straightforward, since this is what Dafny is commonly used for (e.g., see [5]). We discuss some proof engineering considerations, particularly how we deal with proof brittleness, in Section 5.
- Next, the programmer decides on the topology of the parallel version and writes the corresponding ghost specification of the communication as in the first snippet of Dafny code in Figure 2. This specification may call out to the previously developed functional specification to describe the payloads of all the messages.
- Finally, the programmer writes the parallel version of the program while ensuring that (i) the zeroth process returns the same result as the functional specification (therefore being equivalent to the sequential implementation by transitivity), and (ii) the program abides by topology specification (enforced by DafnyMPI). This is typically the most difficult part of the process. In our proofs, we follow the top-down approach by sketching the main program loop first and assuming lower-level methods behave as expected, then proving the correctness of these lower-level methods.

To reason about the return value of a process in an MPI program, and, consequently, to establish functional equivalence, we must reason about the values returned by the individual MPI operations. DafnyMPI allows such reasoning by establishing two properties. First, DafnyMPI uses rely-guarantee reasoning to prove that the messages received by any process are exactly as specified by the user. Second, DafnyMPI enforces safe use of buffers, to prevent situations when two receive operations write simultaneously to the same buffer. Together, these properties guarantee that the content of the write buffer is exactly as specified by the user at the time when the corresponding read operation terminates. We now discuss these two properties in more detail.

Rely-Guarantee Reasoning. For Dafny to verify properties about the values computed by each process, it must know the contents of the messages these processes receive. It is clear that the contents will always be exactly as specified by the Message function (introduced in Section 3.1). Specifically, the AtSend axiom guarantees that the variable from which the message originates abides by the Message function and the AtSet and AtRead axioms prevent the process or another MPI operation from writing to this variable until the communication completes. Because the Message function is a pure function of the message tag and the number of processes, any two processes will always agree on the contents of the message with the same tag. More formally, this is an example of rely-guarantee reasoning. Dafny can rely on each message received by a given process to abide by the user-provided specification because it guarantees that every message ever sent also abides by the specification. In the Dafny code, this is captured by a postcondition on each **wait** call, which describes the message being received.

Safe Use of Read and Write Buffers. Our core calculus prevents writes to variables used in MPI communication (via the AtSet axiom) and prevents any use of variables that an MPI operation may already be writing to (via the AtRead axiom). These restrictions are necessary because, in the presence of non-blocking operations, a process can execute concurrently with the MPI operations it initiated. To enforce these guarantees in Dafny, we must additionally account for aliasing and the fact that real MPI implementations perform communication through buffers, i.e., arrays of data shared by reference. In practice, MPI libraries may read from or write to the designated portion of a buffer at any time until the corresponding operation completes. Send commands can, and often do, use overlapping buffers, but each receive command must have exclusive access to the relevant portion of the buffer. Additionally, a process cannot write to a buffer currently in use by any MPI operation or read from any buffer that is being written to by an MPI call.

Enforcing the correct use of buffers is possible in Dafny because Dafny abstracts away direct memory access, meaning the only way to access a buffer is by calling an instance method on the relevant object. To support buffers, we introduce two custom classes for 1D and 2D arrays of floating-point numbers. Both classes implement a shared interface (a.k.a. *trait* in Dafny) for modeling memory contiguity and per-element read and write locks as part of the ghost state. For each array element, the object's ghost state tracks whether it is currently being written to by a receive operation or is being read by (possibly multiple) send operations. Write locks act as flags: when a portion of an array is locked for writing, that region cannot be read or written until the lock is released. Read locks, in contrast, act as counters: multiple MPI operations may read the same elements concurrently, but writes are disallowed as long as at least one operation is still reading them. Each send or receive command takes as input a reference to the array used as a buffer, together with the starting index and the size of the message. The preconditions of the corresponding MPI call ensure the specified array elements are available for reading or writing. If the MPI call is non-blocking, the postconditions then mark the relevant elements of the array as locked until the operation completes. For 2D arrays, we model C-style contiguity (which is guaranteed at runtime by Python's Numpy library as discussed in 5), meaning that a row or series of rows are contiguous in memory but columns are not, unless the array only has one column.

Together, the rely-guarantee reasoning about message payloads and the read/write-lock mechanism controlling buffer access guarantee that the content of the received messages is exactly as specified by the user, thereby enabling proofs of functional equivalence.

5 Implementation

DafnyMPI consists of two parts. The first part is the specifications of MPI primitives, which are instance methods of the `MPI.World` class. These specifications, along with a few supplementary lemmas and functions, comprise 379 lines of Dafny code and form the trust base of the library. This file does not get compiled to Python directly; instead, each specified MPI primitive is implemented as a Python wrapper function (about 60 lines of code total), which we provide alongside the library.

The second part of DafnyMPI defines two classes for manipulating 1D and 2D arrays of real numbers, with the added ability to track which elements are currently in use by MPI (see above). For maximum portability, all array operations such as rolling, slicing, scalar and per-element multiplication and addition, etc. are implemented in Dafny and rely only on calls to the constructor and three primitive `Get`, `Set`, and `Size` methods. This part of the library totals 1772 lines of Dafny code, which can be directly compiled to Python (or any other target language supported by Dafny) with the exception of the constructor and the three primitive methods mentioned above. The latter are all implemented in Python using the Numpy library, and the code is provided alongside the DafnyMPI library. Numpy's default array constructor also guarantees C-style contiguity of array elements in memory, which is a property our Dafny code assumes on constructing an array and

maintains throughout. In practice, all array operations could be delegated to Numpy at runtime to optimize for performance, but we opted to minimize the trust base instead.

Next, we discuss some of the relevant implementation challenges.

Dealing with Tag Overflow. The proof of deadlock freedom in Section 3.5 assumes that MPI tags are unbounded. In reality, different MPI implementations impose different limits on tag size, e.g., the Intel MPI library uses 20 bits to represent tags [35]. The only requirement imposed by the MPI Standard itself is that the upper bound on the tag must be no less than 32767 [25].

To ensure tags are always valid, DafnyMPI adds a proof obligation to require that the Barrier function we introduce in Section 3.1 never increases by more than 32767. This ensures that at most 32767 tags are used between any two barriers. Therefore, even if tag overflow occurs, tags remain unique between barriers, allowing processes to deterministically identify messages.

Supporting Blocking Send and Receive. Our core calculus models only non-blocking send and receive operations. As mentioned earlier, we can easily encode the blocking variants as an **isend** or **irecv** followed immediately with a **wait**. The DafnyMPI library supports blocking send and receive by expressing them in this way for verification purposes while using the corresponding `MPI_SEND` and `MPI_RECV` commands at runtime. Support for blocking sends and receives simplifies the code by removing the need to manage `MPI_REQUEST` objects when non-blocking communication provides no additional benefit.

Supporting Collective Operations. The MPI standard describes a number of one-to-many, many-to-one, and many-to-many collective operations such as `MPI_SCATTER`, `MPI_GATHER`, and `MPI_ALLREDUCE`. The core calculus does not include these features, but DafnyMPI does support the latter two operations by treating them as decorated barriers. Specifically, DafnyMPI imposes a restriction on collective operations requiring that they are used only when there are no pending send or receive operations running in the background. At runtime, DafnyMPI explicitly puts a barrier before any such collective operation. Thus, the addition of these new collective operations cannot introduce deadlocks. Any information necessary to reason about the output of a given operation is provided by the user when the MPI is initialized, similar to how the one-to-one messages are specified.

For example, to describe the `Gather` operation on line 35 of our running example, the user must specify that the final `nt`th barrier is a `Gather` operation and provide the exact functional specification of the information that will be collected (not shown in Figure 2 in the interest of conciseness). At the call to `Gather`, in addition to any conditions that are required for a regular barrier, the user must also prove that the data sent by each process matches the respective portion of the specified result and that the relevant buffers can be read from and written to. The specification of `Gather` then guarantees that the data received by the root process will be exactly as specified above.

Termination. Of all the correctness properties we list in Section 2.3, proving termination requires the least amount of work. Once deadlock freedom is established and the verifier successfully rules out runtime errors such as out-of-bounds array access, loops and recursions remain as the only obstacles to termination. We establish that loops and recursion terminate as is standard in Dafny, by specifying some positive quantity that decreases on each iteration and is bounded below by zero. For the outer for loop in our running example in Figure 2, this quantity is `nt-n`, the number of iterations that remain.

Proof engineering considerations. A common issue that arises in automated verification is *brittleness*: the more information one gives to the verifier, the more likely it is that the verification query times out. To address this problem, we follow the approach previously introduced by Dafny engineers [7, 24] in making most functions—particularly those involving quantifiers—*opaque*, hiding

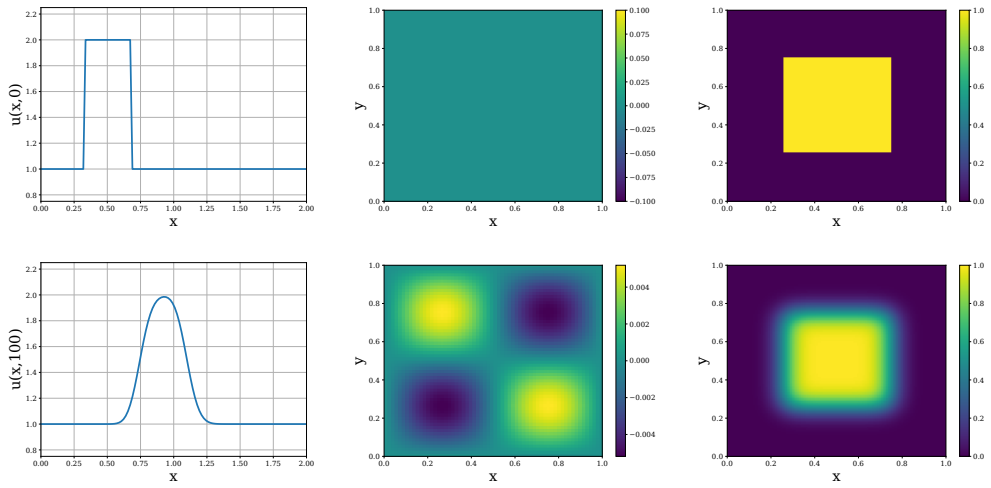

Fig. 6. Benchmark PDE simulations. Top row: initial conditions; bottom row: solutions after 100 time steps as computed by DafnyMPI. From left to right: Linear Convection, Poisson, Heat Diffusion

their definitions from the verifier unless explicitly revealed. We also specify the behavior of some such functions using specialized lemmas instead of using postconditions, which similarly gives the user more control over which facts the verifier has access to at any given point. We recommend that the clients of DafnyMPI follow the same approach to avoid brittleness in their proofs.

6 Evaluation

To evaluate whether DafnyMPI meets its design goals, we implemented and verified three common numerical PDE solvers. We evaluated the proof engineering effort, in terms of the amount of auxiliary proof-related code, and runtime performance, in terms of performance of the parallel implementation. Our results suggest that DafnyMPI can offer significant runtime improvements, with speedups of up to five times, while keeping the added proof burden manageable, even as the amount of proof-related code increases compared to sequential implementations. We conducted all experiments on a MacOS machine with an M3 processor and 48 GiB of memory, and the Dafny solver was configured to use an unlimited number of virtual cores for verification.

6.1 Benchmarks

We used DafnyMPI to implement numerical solvers for three different PDEs, each of which uses a different communication pattern. Figure 6 illustrates the solver results after simulating each PDE for 100 time steps using DafnyMPI. Figure 1 sketches the MPI communication pattern for each solver. The benchmarks are as follows:

- **Linear Convection.** As discussed in Section 2, we implemented the upwind scheme for solving the 1D linear convection PDE. The equation models a wave propagating at constant speed, as shown in the left column of Figure 6. Notice that while the analytical solution to linear convection preserves the wave shape exactly, the numerical solution introduces diffusion and cannot fully maintain sharp edges in the initial condition. The top left schematic in Figure 1 shows that, in the MPI version, each process must send the rightmost value in its domain to its neighbor at each iteration.

- **Poisson.** We implemented Jacobi iteration for solving the 2D Poisson equation with Dirichlet boundary conditions. The Jacobi method updates the value at a point based on all four of its neighbors on each iteration. For the MPI implementation, we partition the domain into equal horizontal stripes. Each process must send the top and bottom rows of its stripe to its neighbors and receive the two boundary rows in return. The top right schematic in Figure 1 illustrates this communication pattern (we use columns instead of rows in the figure due to space constraints.) The solution to the Poisson equation is a function whose second spatial derivative (Laplacian) is equal to another function, provided as input. Over the course of multiple iterations, the solution can be expected to converge to an equilibrium, so the processes must communicate on each iteration to check whether convergence has been reached. The latter is handled via a collective call to `MPI_ALLREDUCE`. The middle column of Figure 6 shows the solution to the Poisson equation after 100 iterations, where we set the initial conditions to a matrix of zeroes and the right-hand side of the equation to a function $f(x, y) = \sin(2\pi x)\sin(2\pi y)$, whose values on the discretized domain are precomputed using externally implemented trigonometric functions.
- **Heat Diffusion.** We implemented the fourth-order Runge-Kutta method (RK4) and applied it to the 2D heat diffusion equation. The right column of Figure 6 illustrates the behavior described by the equation using a heatmap. The heat that is originally at the center of the plot gradually dissipates as the simulation proceeds. The fourth-order Runge-Kutta method computes the value at a point based on a region around it that includes any points within the Manhattan distance of 4. For the MPI implementation, we again partition the domain into equal horizontal stripes. Unlike in Poisson, however, the processes must communicate 4 rows of data at a time, which introduces an additional layer of complexity to the problem. The bottom schematic in Figure 1 illustrates this process (as before, we use columns instead of rows.) Note that the data the process sends to its two neighbors can be overlapping, and the sends can happen concurrently. The receives, however, must use separate buffers.

6.2 Proof Engineering Effort

To provide insight into the engineering effort involved, Table 3 reports the number of lines of code (LOC) for each of the three benchmarks described above. While it is only an indirect measure, LOC roughly corresponds to the amount of effort that went into verifying each respective part of the code. We separately list the number of lines in the specification (Spec), sequential implementation (Seq), and parallel MPI implementation (Par) for each benchmark. For the Heat Diffusion and Poisson benchmarks, we also list the size of the codebase that is shared between the sequential and the parallel implementations (Shared), which corresponds to several methods for implementing the Jacobi and RK4 techniques, respectively. As can be expected, due to the increased complexity, the parallel version requires more lines of code to implement.

Ghost vs Executable Code. Table 3 also reports the percentage of lines in each group of files that are *ghost*. Ghost code is code that is not executable and whose only purpose is to support the proof. As can be seen from the figure, the percentage of ghost code increases significantly for the parallel version, which reflects the increased complexity of the proof. At the same time, the percentage of ghost code in the parallel implementations is comparable to that in some of the more complex subroutines of the sequential version. For example, the 4th-order Runge-Kutta method (the Shared subcategory of the Heat Diffusion benchmark in Table 3) requires a significant amount of auxiliary proof code to relate the functional specification to the rolling array operations used in the implementation. As a result, the ghost code makes up 81% of this shared component, which is close to the 89% observed in the parallel implementation of the same benchmark.

Table 3. Number of lines of code (LOC) and verification time as measures of proof complexity.

	Linear Convection			Poisson				Heat Diffusion				Total
	Spec	Seq	Par	Spec	Shared	Seq	Par	Spec	Shared	Seq	Par	
LOC	209	106	379	384	126	153	1626	328	406	137	1516	5370
% Ghost	100	45	65	100	72	55	91	100	81	57	89	87
Time to verify (s)	3	3	25	4	6	4	148	3	11	4	154	371

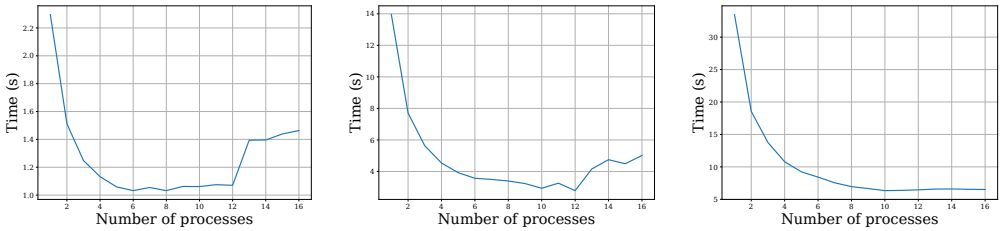

Fig. 7. Running time as function of the process count. From left to right: Linear Convection, Poisson, Heat Diffusion.

Verification and Compilation Speed. It takes about 6 minutes to verify all of the benchmarks and under 20 seconds to compile the code to Python using Dafny's built-in compiler. Heat Diffusion is the most expensive benchmark when it comes to verification time, as can be seen in Table 3. This is primarily due to the need to map the message payload between 2D and 1D representations multiple times during verification to ensure that the data being sent is contiguous in memory.

6.3 Runtime Performance

Given the considerable effort that goes into proving an MPI program correct, we want to be certain that the programs we write provide the desired performance boost when compared with corresponding sequential implementations. Thus, we transpiled all three benchmarks to Python using the Dafny compiler and ran them with different numbers of processes. [1] Figure 7 shows the results of this experiment. All time measurements are the median of three runs. We use the sequential implementation for the one-process run. As expected, increasing the number of processes reduces the running time, achieving up to a fivefold speedup over the sequential version in the Heat Diffusion benchmark. However, as is typical, increasing the number of processes past a certain threshold leads to diminishing returns, and, for Linear Convection and Poisson, the running time even goes up once we assign 12 or more processes to the MPI program. We hypothesize that the running time increases because we ran the experiments on a machine with 16 cores, so giving 12 cores to the program might have interfered with the other processes running in the background, thereby increasing the total running time. This effect is most noticeable in the Linear Convection benchmark because this is the most lightweight of the benchmarks in terms of running time.

[1]For simplicity of reasoning about the program's result, our current proofs assume that the domain size is divisible by the number of processes. This assumption does not affect the communication pattern or the proof of deadlock freedom. When the domain size is not divisible by the number of processes, we round it up to the nearest divisible value before measuring running time. In our benchmarks, all domain sizes are divisible by 16, so experiments with 1, 2, 4, 8, and 16 processes are unaffected by this adjustment.

7 Discussion and Future Work

Comparison with a Proof Assistants. An initial prototype of the DafnyMPI system was partially formalized in the Rocq proof assistant. To achieve this, we developed a domain-specific language similar to the core calculus in Section 3.1, alongside specifications for an interpreter. While we still believe that this approach is feasible, during the verification of the linear convection benchmark in Rocq we found that the proof engineering burden could be lessened considerably using the constructs built-in to Dafny. The benefit is most apparent when comparing the length of the partial proof of deadlock freedom for the Linear Convection example in Rocq (about 1000 lines, even with many admitted proof goals), which exceeded that of the entire program correctness proof in DafnyMPI (694 lines). While it could be both feasible and useful to implement reasoning similar to DafnyMPI with other verification frameworks, we leave this task to future work.

Floating-point Arithmetic. Throughout our codebase, we use Dafny's `real` type to model floating-point numbers. While common in verification, this approach does not provide a precise model of IEEE-754 or any other floating-point standard because it does not account for floating-point underflow, and does not prevent division-by-zero errors that may result from it. Several studies exist on automating the verification of programs that use floating-point arithmetic [1, 10], and it may be a possible direction for future work on DafnyMPI.

Running Time. Because we model most array operations in Dafny, the runtime performance suffers compared to what could be achieved with native libraries such as Python's Numpy. This is, in part, by design: we chose to minimize the trust base by only relying on Numpy for single-element set and get operations. In principle, it should be possible to replace our implementations of such operations as element-wise addition, slicing, and rolling, with corresponding Numpy operations at runtime, which should result in a significant performance boost.

8 Related work

There are several threads of related work.

Dynamic Analysis and Model Checking for MPI. Researchers have developed a number of verification systems for MPI. Practical tools such as ISP [32], its successor DAMPI [33], and MOPPER [9] execute MPI programs under controlled schedulers or otherwise analyze the results of a past execution. Others, including TASS [27], MPI-SV [6], and CIVL-C [23], combine symbolic execution with model checking to explore the space of possible message interleavings. These approaches do not require the same amount of proof engineering as DafnyMPI, but they are subject to potential state explosion, even though techniques exist that allow pruning the search space [31].

Lock Orders. Ordering on locks has been used previously to establish deadlock freedom in several verification contexts. A system that enforces a global order on lock acquisitions and receives has been implemented [21] for the Chalice verifier and an extension that supports condition variables [12] has been implemented for VeriFast. Recent work [14] uses more complex ordering topologies for managing higher-order locks. These studies focus on locks in the context of shared memory concurrency and dynamic thread creation, where communication obligations can be established or transferred when a new thread is forked. In contrast, we target the MPI setting, where the number of processes is arbitrary but fixed throughout execution, meaning the programmer can describe the entire communication topology as a function of process ID without the need to explicitly manage communication for each process. Moreover, data transfer in MPI can occur asynchronously relative to the initiating process, a behavior explicitly captured by our operational semantics via the message payload buffer and the separation between the initialization and completion stages

of each point-to-point operation. Finally, DafnyMPI allows simultaneous sends and receives and provides an easy way to integrate collective communication operations.

Concurrency with Dafny. Several projects have explored concurrency in the context of Dafny.

The DafnyInfoFlow project [28] has added support for concurrent reasoning, also using rely-guarantee, but its primary goal is to prevent leakage of information from private variables. In contrast, DafnyMPI focuses on deadlock freedom, termination, and functional equivalence.

Armada [22] is a tool and a programming language whose operational semantics is modeled in Dafny and which allows verification of C-like concurrent code. Because Armada is not a Dafny library but a distinct language, Armada programs cannot directly reuse Dafny constructs the way DafnyMPI can. Moreover, Armada explicitly targets low-level, shared-memory concurrency, which is a fundamentally different setting from MPI.

Other approaches. Concurrent separation logic (CSL), as exemplified by Iris [16] and Steelcore [30], extends Hoare logic with shared-memory semantics such as resources (heap), resource ownership, and the separating implication for heap predicates [3]. While CSL models the heap directly, DafnyMPI abstracts away these details, reducing the proof engineering burden but limiting applicability to MPI programs. CSL also lacks support for liveness or deadlock-freedom, typically requiring a supplemental logic, as in LinearActris [15], to establish these properties.

Several researchers have developed type systems that aim to prevent common concurrency bugs such as data races and deadlock. Examples include TyPiCal [18], based on session types; CLL [11], based on dependent types; and Asynchronous Liquid Separation Types [17], based on refinement types. In contrast to DafnyMPI, these approaches do not reason about program equivalence.

Finally, researchers have developed a variety of other formal verification frameworks for concurrent programs, such as VCC[8], VerCors[2]. A major difference between DafnyMPI and these frameworks lies in their core design. VCC requires annotating C code with invariants and pre- or postconditions; Vercors verifies Java code with separation logic annotations.

9 Conclusion

We have introduced DafnyMPI, a novel verification library that brings formal reasoning about message-passing interface (MPI) parallel programs into Dafny, a language designed for sequential code. Our approach enables verification of core MPI properties, including deadlock freedom and functional equivalence with sequential specifications, all without requiring custom concurrency logics. By leveraging Dafny's built-in verification capabilities and layering MPI-specific invariants on top, we believe our approach achieves a balance of rigor, accessibility, and scalability.

Our formal model demonstrates that a small set of preconditions on message tags, barriers, and buffer usage suffices to guarantee deadlock freedom. These preconditions are enforced through a core calculus and a set of Dafny assertions. Experimental results on PDE solvers demonstrate the viability of our method: although proof engineering remains nontrivial, the verified parallel implementations exhibit significant speedups over their sequential counterparts.

Looking forward, we envision extensions of DafnyMPI to support more expressive MPI features (e.g., communicators, wildcards), integrate more seamlessly with scientific programming ecosystems, and further reduce proof-engineering effort. We hope that DafnyMPI will serve as a foundation for future work on verified high-performance computing and encourage the adoption of formal methods in domains where correctness is critical but traditional techniques are impractical.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research was partially funded by the U.S. National Science Foundation under Award Nos. 2313998 and 2513872.

References

[1] Rosa Abbasi, Jonas Schiffl, Eva Darulova, Mattias Ulbrich, and Wolfgang Ahrendt. 2021. Deductive Verification of Floating-Point Java Programs in KeY. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 242–261. doi:10.1007/978-3-030-72013-1_13

[2] Stefan Blom and Marieke Huisman. 2014. The VerCors Tool for Verification of Concurrent Programs. In *International Symposium on Formal Methods*. 127–131. doi:10.1007/978-3-319-06410-9_9

[3] Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent Separation Logic. *ACM SIGLOG News* 3 (2016), 47–65. doi:10.1145/2984450.2984457

[4] Franck Cassez, Joanne Fuller, Milad K Ghale, David J Pearce, and Horacio MA Quiles. 2023. Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny. In *International Symposium on Formal Methods*. 571–583. doi:10.1007/978-3-031-27481-7_32

[5] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K. Rustan M. Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clement Pit-Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, Jean-Baptiste Tristan, Lucas Wagner, Michael W. Whalen, Remy Willems, Tongtong Xiang, Tae Joon Byun, Joshua Cohen, Ruijie Fang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. 2025. Formally Verified Cloud-Scale Authorization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2508–2521. doi:10.1109/ICSE55347.2025.00166

[6] Zhenbang Chen, Hengbiao Yu, Xianjin Fu, and Ji Wang. 2020. MPI-SV: A Symbolic Verifier for MPI Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea). 93–96. doi:10.1145/3377812.3382144

[7] Joseph W. Cutler, Michael Hicks, and Emina Torlak. 2024. Improving the Stability of Type Safety Proofs in Dafny. In *Dafny 2024 - POPL 2024*. https://popl24.sigplan.org/details/dafny-2024-papers/3/Improving-the-Stability-of-Type-Safety-Proofs-in-Dafny

[8] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. 2009. VCC: Contract-Based Modular Verification of Concurrent C. In *2009 31st International Conference on Software Engineering-Companion Volume*. 429–430. doi:10.1109/ICSE-COMPANION.2009.5071046

[9] Vojtundefinedech Forejt, Saurabh Joshi, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2017. Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. *ACM Trans. Program. Lang. Syst.* 39 (2017). doi:10.1145/3095075

[10] Clément Fumex, Claude Marché, and Yannick Moy. 2017. Automating the Verification of Floating-Point Programs. In *Verified Software. Theories, Tools, and Experiments: 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers 9*. 102–119. doi:10.1007/978-3-319-72308-2_7

[11] Deepak Garg and Frank Pfenning. 2005. Type-Directed Concurrency. In *International Conference on Concurrency Theory*. 6–20. doi:10.1007/11539452_5

[12] Jafar Hamin and Bart Jacobs. 2018. Deadlock-free monitors. In *European Symposium on Programming*. Springer, 415–441. doi:10.1007/978-3-319-89884-1_15

[13] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2019. Actris: Session-Type Based Reasoning in Separation Logic. *POPL* 4 (2019), 1–30. doi:10.1145/3373096

[14] Jules Jacobs and Stephanie Balzer. 2023. Higher-order leak and deadlock free locks. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1027–1057. doi:10.1145/3571229

[15] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *POPL* 8 (2024). doi:10.1145/3632889

[16] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris From the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28 (2018), 1–73. doi:10.1017/S0956796818000151

[17] Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. 2015. Asynchronous Liquid Separation Types. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 396–420. doi:10.4230/LIPIcs.ECOOP.2015.396

[18] Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR 2006–Concurrency Theory: 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006. Proceedings 17*. 233–247. doi:10.1007/11817949_16

[19] K Rustan M Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. 348–370. doi:10.1007/978-3-642-17511-4_20

[20] K Rustan M Leino. 2017. Accessible Software Verification With Dafny. *IEEE Software* 34 (2017), 94–97. doi:10.1109/MS.2017.4121212

[21] K Rustan M Leino, Peter Müller, and Jan Smans. 2010. Deadlock-free channels and locks. In *European Symposium on Programming*. Springer, 407–426. doi:10.1007/978-3-642-11957-6_22

[22] Jacob R Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R Wilcox, and Xueyuan Zhao. 2020. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 197–210. doi:10.1145/3385412.3385971

[23] Ziqing Luo, Manchun Zheng, and Stephen F Siegel. 2017. Verification of MPI programs using CIVL. In *Proceedings of the 24th European MPI Users' Group Meeting*. 1–11. doi:10.1145/3127024.3127032

[24] Sean McLaughlin, Georges-Axel Jaloyan, Tongtong Xiang, and Florian Rabe. 2024. Enhancing Proof Stability. In *Dafny 2024 - POPL 2024*. https://popl24.sigplan.org/details/dafny-2024-papers/14/Enhancing-Proof-Stability

[25] Message Passing Interface Forum. 1994. *MPI: A Message-Passing Interface Standard Version 1.0*. https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps Accessed: 2025-05-04.

[26] Salman Pervez, Ganesh Gopalakrishnan, Robert M Kirby, Robert Palmer, Rajeev Thakur, and William Gropp. 2007. Practical Model-Checking Method for Verifying Correctness of MPI Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI User's Group Meeting*. 344–353. doi:10.1007/978-3-540-75416-9_46

[27] Stephen F Siegel and Timothy K Zirkel. 2011. Automatic Formal Verification of MPI-based Parallel Programs. *ACM Sigplan Notices* 46 (2011), 309–310. doi:10.1145/1941553.1941603

[28] Graeme Smith. 2023. A Dafny-Based Approach to Thread-Local Information Flow Analysis. In *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 86–96. doi:10.1109/FormaliSE58978.2023.00017

[29] Gilbert Strang. 2014. *Differential Equations and Linear Algebra*. Wellesley-Cambridge Press. doi:10.1137/1.9780980232790

[30] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *POPL* 4 (2020). doi:10.1145/3409003

[31] Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M Kirby, Rajeev Thakur, and William Gropp. 2008. Implementing Efficient Dynamic Formal Verification Methods for MPI Programs. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. 248–256. doi:10.1007/978-3-540-87475-1_34

[32] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. ISP: A Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, UT, USA). 285–286. doi:10.1145/1345206.1345258

[33] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2010. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–10. doi:10.1109/SC.2010.7

[34] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M Kirby, and Rajeev Thakur. 2009. Formal Verification of Practical MPI Programs. *ACM Sigplan Notices* 44 (2009), 261–270. doi:10.1145/1594835.1504214

[35] Large MPI Tags with the Intel MPI Library. 2025. https://www.intel.com/content/www/us/en/developer/articles/technical/large-mpi-tags-with-the-intel-mpi.html Accessed: 2025-04-12.