Protocol-based verification of MPI programs

Eduardo R. B. Marques, Francisco Martins,
Vasco T. Vasconcelos, César Santos,
Nicholas Ng, Nobuko Yoshida

DI–FCUL–TR–2014–5

DOI:10455/6901

(http://hdl.handle.net/10455/6901)

November 2014

Published at Docs.DI (http://docs.di.fc.ul.pt/), the repository of the Department of Informatics of the University of Lisbon, Faculty of Sciences.

Protocol-based verification of MPI programs

Eduardo R. B. Marques[1], Francisco Martins[1], Vasco T. Vasconcelos[1]
César Santos[1], Nicholas Ng[2], and Nobuko Yoshida[2]

[1] LaSIGE, Faculty of Sciences, University of Lisbon
[2] Imperial College London

Abstract. We present a methodology for the verification of Message Passing Interface (MPI) programs written in C. The aim is to statically verify programs against protocol specifications, enforcing properties such as fidelity and absence of deadlocks. We make use of a protocol language based on a dependent type system for message-passing parallel programs. For the verification of a program against a given protocol, the protocol is first translated into a representation read by VCC, a software verifier for the C programming language. The program is then annotated with specific assertions that, together with a pre-established set of contracts for MPI primitives, guide the verifier to either prove or disprove the program's conformance to the protocol. We successfully verified MPI programs in a running time that is independent of the number of processes or other input parameters. This contrasts with other techniques, notably model checking and symbolic execution, that suffer from the state-explosion problem. We experimentally evaluated our approach against TASS, a state-of-the-art tool for MPI program verification.

1 Introduction

Message Passing Interface (MPI) [4] is the *de facto* standard for programming high performance parallel applications targeting hundreds of thousands of processing cores. MPI programs adhere to the Single Program Multiple Data (SPMD) paradigm, in which a single program, written in C or Fortran, specifies the behaviour of the various processes, each working on different data. Programs make calls to MPI primitives whenever they need to exchange data. MPI offers different forms of communication, notably, point-to-point and collective communication.

Developing MPI programs raises several problems: one can easily write code that causes processes to block indefinitely waiting for messages, or that exchange data of unexpected sorts or lengths. Statically verifying that such programs are exempt from communication errors is far from trivial. The state of the art verification tools for MPI programs use advanced techniques such as model checking and symbolic execution [6,21]. These approaches frequently stumble upon the problem of scalability, since the search space grows exponentially with the number of processes. Real-world applications may limit the number of processes subject to verification to less than a dozen, e.g., see [22].

The verification is further complicated by the different communication semantics for the various MPI primitives [4,21], or by the difficulty in disentangling processes' collective and individual control flow written on a single source file [1]. These also

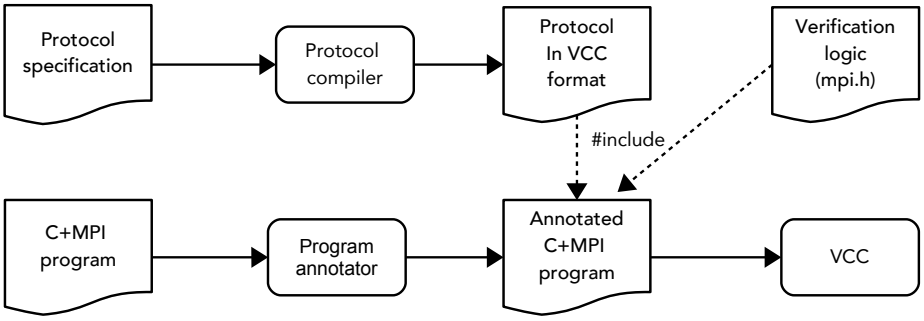

Fig. 1. Verification flow for C+MPI programs

naturally arise in other more recent standards for message-based parallel programs, such as MCAPI [12].

We attack the problem of verifying C+MPI code using a type theory for parallel programs [26]. Under such a framework a type describes a protocol, that is, the communication behaviour of a program. Programs that conform to one such type are guaranteed to follow the protocol, and, by implication, type-safe and free from deadlocks. The type system features a dependent type language including specific constructors for some of the most common communication primitives found in the field of parallel programming, in addition to sequential composition, primitive recursion, and a form of collective choice. Dependent types are restricted to index objects drawn from the restricted domain of integer, floating-point values and arrays, turning type checking into a decidable problem.

The verification workflow of C+MPI code against a given protocol is illustrated in Fig. 1. We employ a software toolchain composed of a protocol compiler, a program annotator for semi-automated insertion of verification logic in C+MPI code, and the VCC deductive software verifier for C [2]. A compiler validates protocols and generates VCC code. The VCC tool takes an annotated program, the protocol in VCC format, and a VCC-annotated MPI library containing contracts for various MPI primitives and complementary logic for protocol verification, and verifies that the program conforms to the protocol. In order to ease the burden of annotating program code, we employ a semi-automatic program annotator.

We evaluated our methodology on a benchmark comprising a series of MPI programs taken from the literature. Our approach, based on type checking, is immune to the state-explosion problem. The number of processes and other program inputs are merely parameterised by dependent type restrictions, and program verification takes constant time. To stress this point, we verified the same benchmark programs using TASS [20,23,24,25], a state-of-the-art tool for MPI program verification combining techniques from model checking and symbolic execution. Like our toolchain, TASS is able to check for deadlocks and type safety for MPI programs; in contrast, however, a precise value for the number of processes and other input parameters must be specified. As the number of process grows, the state-explosion problem inherent to model checking leads to unscalable verification times and/or memory exhaustion in the execution of TASS.

We originally proposed the idea of checking MPI programs using multiparty session types in [9]. Subsequent work concerned a preliminary evaluation of the approach and experiments [14]. This paper is a major evolution of [14], where: we did not make use of a protocol language, but merely employed unchecked specifications of the protocol directly in VCC; verification times did not scale and required an a priori defined number of processes, given that the verification logic required term unfolding for most examples and was not parametric; and program annotation was not semi-automated, but instead relied exclusively on manually introduced annotations.

The rest of the paper is structured as follows. The next section surveys related work. Sec. 3 introduces MPI programs and the protocol language; Sec. 4 describes the verification workflow; Sec. 5 the software toolchain; and Sec. 6 the benchmark results. Sec. 7 concludes the paper and discusses directions for future work. In appendix, we provide complementary material referenced throughout the text: protocols for the evaluation examples (A), code listings (B), core fragments of the VCC logic employed in program verification (C), and MPI function contracts also in VCC (D).

2 Related work

Scribble [10] is a language to describe protocols for message-based programs based on the theory of multiparty session types [11]. Protocols written in Scribble include explicit senders and receivers, thus ensuring that all senders have a matching receiver and vice versa. Global protocols are projected into each of their participants' counterparts, yielding one local protocol for each participant present in the global protocol. Developers can then implement programs based on the local protocols and using standard message-passing libraries, like Multiparty Session C [16]. In this work we depart from multiparty session types along two distinct dimensions: (1) our protocol language is specifically built for MPI primitives, and (2) we do not explicitly project a protocol but else check the conformance of SPMD code to a global protocol.

Gopalakrishnan, Kirby, et al. [6] authored a recent survey on the state-of-the-art in MPI program verification. The objectives of surveyed works are diverse and include the validation of arguments to MPI primitives as well as resource usage [27], ensuring interaction properties such as absence of deadlocks [21,27,28], or asserting functional equivalence to sequential programs [21,23]. The methodologies employed are also diverse, ranging from traditional static and dynamic analysis up to model checking and symbolic execution. In comparison, our novel methodology is based on type checking, thus avoiding the state-explosion problem inherent to other approaches.

TASS [20,23,24,25] employs model checking and symbolic execution, but is also able to verify user-specified assertions for the interaction behaviour of the program, so-called collective assertions, and to verify functional equivalence between MPI programs and their sequential counterparts [23]. ISP [28] is a deadlock detection tool that explores all possible process interleaving using a fixed test harness. MOPPER [3] is a verifier that detects deadlocks by checking formulae satisfiability, obtained by analysing execution traces of MPI programs. It uses a propositional encoding of constraints and partial order reduction techniques, obtaining significant speedups when compared to ISP. The concept of parallel control-flow graphs [1] allows for the static and dynamic

analysis of MPI programs, e.g., as a means to verify sender-receiver matching in MPI source code. Dynamic execution analysers, such as DAMPI [27] and MUST [8], strive for the runtime detection of deadlocks and resource leaks.

3 MPI programs and the protocol language

This section introduces MPI programs and the language for the specification of corresponding protocols. We present an MPI program for computing finite differences that is used as a running example, the corresponding protocol, and technical details regarding protocol formation rules.

```
1  int main(int argc, char** argv) {
2    // process rank; number of processes; problem size; max. iterations
3    int rank, procs, n, max_iter, iter;
4    ...
5    MPI_Init(&argc, &argv);
6    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7    MPI_Comm_size(MPI_COMM_WORLD, &procs);
8    n = atoi(argv[1]);
9    max_iter = atoi(argv[2]);
10   if (rank == 0) read_vector(data, n); // read initial data
11   ...
12   int local_n = n / procs;
13   MPI_Scatter(data,local_n,MPI_FLOAT,&local[1],local_n,MPI_FLOAT,0,MPI_COMM_WORLD);
14   int left  = rank == 0 ? procs - 1 : rank - 1; // left neighbour
15   int right = rank == procs - 1 ? 0 : rank + 1; // right neighbour
16   for (iter = 1; i <= max_iter; iter++) {
17     if (rank == 0) {
18       MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
19       MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
20       MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
21       MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
22     } else if (rank == procs - 1) {
23       MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
24       MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
25       MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
26       MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
27     } else {
28       MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
29       MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
30       MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
31       MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
32     }
33     ...
34   }
35   // Computes convergence error and final solution at rank 0
36   MPI_Reduce(&localErr, &globalErr, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);
37   MPI_Gather(&local[1],local_n,MPI_FLOAT,data,local_n,MPI_FLOAT,0,MPI_COMM_WORLD);
38   if (rank == 0) {
39     write_float("Convergence error: ", globalErr);
40     write_vector(data, n);
41   }
42   ...
43   MPI_Finalize();
44   return 0;
45 }
```

Fig. 2. Excerpt of an MPI program for the finite differences problem (adapted from [5])

3.1 The finite differences program

Fig. 2 presents a fragment of an MPI program that implements the method for computing finite differences, taken from [5]. The full code is provided in Appendix B.1. Given an initial vector X_0, the program calculates successive approximations X_1, X_2, ..., X_i of the solution, yielding, after i steps, the final approximation together with an upper bound on the error.

Function `main` defines the behaviour of all MPI processes together. The MPI runtime is initialised with a call to `MPI_Init`. Each process is assigned a unique process number, designated *rank*, via an `MPI_Comm_rank` call (line 6). The number of processes, `procs`, is obtained via `MPI_Comm_size` (line 7). After the initialisation sequence, all processes read the problem size and the maximum number of iterations, `n` and `max_iter` from the program arguments (lines 8– 9). The process with rank `0` reads the input vector X_0 (line 10), and distributes it in parts of size `n/procs` to all processes (including itself) using a call to `MPI_Scatter` (line 13).

Each process iterates `max_iter` times (line 16–34), communicating with its `left` and `right` neighbours within a ring topology, using point-to-point messages (`MPI_Send`, `MPI_Recv`). Different send/receive orders for different ranks (line 18–21, 23–26, and 28–31) aim at avoiding deadlocks. Safe programs must assume that `MPI_Send` and `MPI_Recv` are blocking, synchronous, and unbuffered operations (see [4], § 3.5, pp. 40–44). Once the loop is done, process `0` computes the global error from the maximum of the local errors (line 36, `MPI_Reduce`), and gathers the final solution, obtaining from each participant (including itself) a part of the vector (line 37, `MPI_Gather`). Finally, each process shuts down the MPI runtime (line 43, `MPI_Finalize`).

The code in Fig. 2 is extremely sensitive to variations in the use of MPI operations. For example, the omission of any send/receive operation (lines 20–35) leads to a deadlock where at least one process will be forever waiting for a complementary send or receive operation. Another example: exchanging lines 22 and 23 leads to a deadlock where ranks `0` and `1` will forever wait for one another. It is also easy to use mismatching types or payload lengths in MPI calls, compromising type and communication safety. For instance, replacing `MPI_FLOAT` for `MPI_INT` at line 21 will go unnoticed by the MPI runtime and the C compiler, since buffer arguments are **`void`** pointers. Similarly, the replacement `2` for `1` in the length argument will go unnoticed, given the unsafe memory model of C.

3.2 The finite differences protocol

To write protocols for MPI programs, we use a dependent type language for message-based parallel computations [26]. We introduce the language using the protocol for the finite differences program, shown in Fig. 3.

In line with the finite differences program, the protocol first introduces the number of processes, `procs`, a value greater than one given that the program exchanges point-to-point messages (line 1). The problem size and the maximum number of iterations, `n` and `max_iter`, are then introduced (lines 2–3), with the restriction that `n` must be positive and a multiple of `procs`, expressed by dependent type `{x:integer|x>0 && x%procs=0}`.

```
1  protocol fdiff procs: {x: integer | x > 1}  {
2    val n: {x: integer | x > 0 && x % procs = 0}
3    val max_iter: {x: integer | x > 0 }
4    scatter 0 float[n/procs]
5    foreach iter: 1 .. max_iter {
6      foreach i: 0 .. procs - 1 {
7        message i, (i = 0 ? procs-1 : i-1) float
8        message i, (i = procs-1 ? 0 : i+1) float
9      }
10   }
11   reduce 0 max float
12   gather 0 float[n/procs]
13 }
```

Fig. 3. Protocol for the finite differences program

Subsequently, the **scatter** operation (line 4) expresses the distribution of the initial solution among all participants, initiated by participant `0`; each participant receives its part of the array (of length `n/procs`), as in the C+MPI program.

The finite differences iteration is expressed by two nested **foreach** loops. The outer **foreach** (lines 5–10) expresses a loop involving all participants. It should be seen as a sequential composition of `max_iter` copies of the **foreach** body, with `iter` replaced by `1`, `2`, ..., `procs-1`. The inner **foreach** (line 6–9) encodes message exchanges of participants with their left and right neighbours. For instance, taking `i=0`, the inner **foreach** body expresses two messages involving at most three processes, participant 0 and its two neighbours, **message** `0,procs-1` **float** and **message** `0,1` **float** (a value of 2 for `procs` will mean that only participants `0` and `1` will be involved). The protocol introduces two final operations, **reduce** `0` **max float** and **gather** `0` **float**`[n/p]` (line 11–12 in Fig 3), in line with the similar steps in the MPI program for obtaining the convergence error and gathering the final solution at rank `0`.

3.3 Protocol formation

Well-formed protocols adhere to a set of formation rules, partially depicted in Fig. 4, adapted from [26]. Protocol formation judgements are of the form $\Gamma \vdash P : \mathbf{type}$, meaning that protocol P is well formed under the typing assumptions in Γ. We use dependent types to express the fact that protocols may depend on index terms, as in the rules for **val** and **foreach**, where protocol P may refer to variable x.

Examples of ill-formed protocols include messages to self: **message** `2 2` **integer**; a message to a non-existent participant: **message** `0 -5` **float**; or a message to a process that cannot be guaranteed to be valid (in the `0` to `procs-1` range), as in **protocol** `proto p:` **integer** {**message** `0 p` **float**}.

$$\frac{\Gamma \vdash 0 \leq i_1, i_2 < \textbf{procs} \wedge i_1 \neq i_2 \quad \Gamma \vdash D : \textbf{dtype}}{\Gamma \vdash \textbf{message}\ i_1\ i_2\ D : \textbf{type}} \qquad \frac{\Gamma \vdash 0 \leq i < \textbf{procs}}{\Gamma \vdash \textbf{reduce}\ i : \textbf{type}}$$

$$\frac{\Gamma \vdash 0 \leq i < \textbf{procs} \quad \Gamma \vdash D <: \{x\colon D'\ \textbf{array} \mid \textbf{length}(x) \bmod \textbf{procs} = 0\}}{\Gamma \vdash \textbf{scatter}\ i\ D : \textbf{type}}$$

$$\frac{\text{see } \textbf{scatter}}{\Gamma \vdash \textbf{gather}\ i\ D : \textbf{type}} \qquad \frac{\Gamma, x\colon D \vdash P : \textbf{type}}{\Gamma \vdash \textbf{val}\ x\colon D.P : \textbf{type}} \qquad \frac{\Gamma : \textbf{context}}{\Gamma \vdash \textbf{skip} : \textbf{type}}$$

$$\frac{\Gamma \vdash P_1 : \textbf{type} \quad \Gamma \vdash P_2 : \textbf{type}}{\Gamma \vdash P_1; P_2 : \textbf{type}} \qquad \frac{\Gamma, x\colon \{y\colon \textbf{integer} \mid i_1 \leq y \leq i_2\} \vdash P : \textbf{type}}{\Gamma \vdash \textbf{foreach}\ x\colon i_1..i_2\ \textbf{do}\ P : \textbf{type}}$$

Fig. 4. Excerpt of the protocol formation rules enforced by the protocol compiler

4 C+MPI program verification

This section describes the logical components defined by our methodology for program verification: a description of the protocol in VCC format; a contract-annotated library for the MPI primitives that enforce the correct use of the prescribed protocol; a VCC theory that describes how protocols advance (reduce); and complementary verification annotations on the program's source code. VCC is a deductive verifier for C, delegating proof obligations to Z3 [15]. The tool incorporates a theory for multithreaded programs allowing to check (local thread) safe memory and data structure invariants established by VCC contracts. We employ VCC logic for the definition of a `Protocol` datatype, defining MPI function contracts, and type-checking memory areas of MPI call arguments.

4.1 Protocol representation

A protocols is represented by a function, as illustrated for the finite differences example in Fig. 5: a ghost function `program_protocol` from the number of processes `procs` into a term of datatype `Protocol`. The definition of the `Protocol` VCC datatype is provided in Appendix C.2.

As illustrated in the figure, the `Protocol` datatype includes a constructor for each primitive of the protocol language, the `seq` constructor to represent the sequence operator (`;`), and the `skip()` constructor to denote the empty protocol. The correspondence between the protocol (in Fig. 3) and the VCC representation should be evident. We stress two technicalities: the representation of refinement types and the handling of variable bindings. We use `_float(n/procs)` (line 8) to represent a float array of size `n/procs`; see Appendix C.1 for the associated definitions. This information allows VCC to check that buffers used in MPI operations are both of the correct type and size. Variable bindings are handled using higher-order abstract syntax [18], by defining anonymous functions that bind variables to continuations. Continuations are represented by `abs` terms (lines 5 and 7), providing a convenient method to deal with term substitution.

```
1  _(ghost Protocol program_protocol (\integer procs) =
2     _(requires procs > 1)
3     _(ensures \result = seq(
4        val(\lambda \integer x; x > 0 && x % procs == 0),seq(
5        abs(\lambda \integer n; seq(
6          val(\lambda integer x; x >= 0),seq(
7          abs(\lambda \integer max_iter; seq(
8            scatter(0, _float(n/procs)), seq(
9            foreach(1, max_iter, \lambda \integer iter;
10             foreach(0, procs - 1, \lambda \integer i; seq(
11               message(i, i==0 ? procs-1 : i-1, _float(1)),
12               message(i, i==procs-1 ? 0 : i+1, _float(1))))),seq(
13         reduce(0, MPI_MAX, _float(1)),
14         gather(0, _float(n/procs))))))))))
15     );
```

Fig. 5. The protocol for the finite differences program in VCC syntax

4.2 VCC theory

In order to advance the protocol, while validating its conformance to the program, we define four functions: `head`, `continuation`, `isMessageFromTo`, and `verifyData`. The `head` function returns the next action in a protocol for a particular rank. The partial definition of the function is below (a full definition can be found in Appendix C.3).

```
Protocol head(Protocol p, \integer rank);
axiom \forall \integer rank, dest; Data data; Protocol p;
 head(seq(message(rank,dest,data),p),rank) == message(rank,dest,data);
axiom \forall \integer from, rank; Data data; Protocol p;
 head(seq(message(from,rank,data),p),rank) == message(from,rank,data);
axiom \forall \integer rank, from, to; Data data; Protocol p;
 from != rank && to != rank ==>
  head(seq(message(from,to,data),p),rank) == head(p,rank))
```

The first line describes the function signature: `head` receives a protocol and a rank, and returns a protocol. The remaining lines describe its behaviour for the case where a message is at the head of the protocol. The first rule is for the case when the message is sent from the current process (notice the `rank` in the source position). The second rule is for the case when the message is received by the current process (`rank` is now in the destination position). The third case is for when the message is neither from nor to the current process; in this case another action is recursively found in the protocol continuation.

The `continuation` function describes the effect of executing an MPI primitive and advancing the protocol as a result. Its implementation is similar to `head` (see Appendix C.3), except that in the first two cases the value of the function is the continuation `p`, and in the third case, the recursive call is `continuation(p,rank)`.

$$\frac{\Gamma \vdash e : P_1 \quad \Gamma \vdash P_1 \equiv P_2 : \mathbf{type}}{\Gamma \vdash e : P_2} \qquad \frac{\Gamma \vdash 0 \leq i < \mathbf{procs} \wedge i \neq \mathbf{rank} \quad \Gamma \vdash d : D}{\Gamma \vdash \mathbf{MPI\_Send}\ i\ d : \mathbf{message\ rank}\ i\ D}$$

$$\frac{\Gamma \vdash e_1 : P_1 \quad \Gamma \vdash e_2 : P_2}{\Gamma \vdash e_1; e_2 : P_1; P_2} \qquad \frac{\Gamma \vdash 0 \leq i < \mathbf{procs} \wedge i \neq \mathbf{rank} \quad \Gamma \vdash d : D}{\Gamma \vdash \mathbf{MPI\_Recv}\ i\ d : \mathbf{message}\ i\ \mathbf{rank}\ D}$$

$$\frac{\Gamma, x\colon \{y\colon \mathbf{integer} \mid i_1 \leq y \leq i_2\} \vdash e : P}{\Gamma \vdash \mathbf{for}(\mathbf{x} = \mathbf{i_1}; \mathbf{x} \leq \mathbf{i_2}; \mathbf{x{+}{+}})\ \mathbf{e} : \mathbf{foreach}\ x : i_1\ ..\ i_2\ P}$$

$$\frac{\Gamma \vdash 0 \leq i < \mathbf{procs} \quad \Gamma \vdash d : D}{\Gamma \vdash \mathbf{MPI\_Reduce}\ i\ d : \mathbf{reduce}\ i\ D} \qquad \frac{\Gamma \vdash 0 \leq i < \mathbf{procs} \quad \Gamma \vdash d : D}{\Gamma \vdash \mathbf{MPI\_Gather}\ i\ d : \mathbf{gather}\ i\ D}$$

Fig. 6. Verification of programs against protocols

Function `isMessageFromTo` checks that the protocol is composed of one message from a given source to a given destination. The function plays a role in the `MPI_Send` contract described below.

```
\bool isMessageFromTo(Protocol p, \integer from, \integer to);
axiom \forall \integer from, to; Data d;
  isMessageFromTo(message(from, to, d), from, to))
```

Finally, predicate `verifyData` asserts that the data communicated in a message conforms to the refinement type.

```
\bool verifyData(Protocol p, MPI_Datatype d, void* buf,\integer n)
axiom \forall \integer a,b,n; IPredicate ip; void* buf;
  verifyData(message(a,b,intRefinement(ip,n)), MPI_INT, buf, n)
    <==> (\forall \integer i; i >= 0 && i < n ==> ip[((int*)buf)[i]]))
```

4.3 Contracts for MPI primitives

Fig. 6 shows a fragment of the typing rules from [26], adapted to the C+MPI context. Each rule assigns a type (a part of a communication protocol) to an MPI operation (e.g., `MPI_Send`, `MPI_Recv`, and `MPI_Reduce`), and to C control flow structures (e.g., sequential composition and **`for`** loops).

We restrict the plethora of parameters to MPI primitives to those related to information exchange: the source or target ranks for an MPI primitive, and the type of data transmitted. For instance, sending a message d to a participant i is abstracted as $\mathbf{MPI\_Send}\ i\ d$, and the inference rule assigns it type **message rank** $i\ D$. The premises to the rule enforce the validity of the target process rank ($0 \leq i <$ **procs**), that processes do not send messages to themselves ($i \neq$ **rank**), and that d is of some valid datatype ($\Gamma \vdash d : D$). Variables in context, including the special variables **rank** and **procs**, are kept in typing environment Γ, essentially a variable-datatype map. The index i of an MPI operation can be an expression such as `i==0 ? procs-1 : i-1`, as found in the protocol (Fig. 5, line 11) and the program (Fig. 2, line 14). To ensure, for instance, that this expression is a valid rank or different from the current rank, we need to pass this assertion to an SMT solver, identified in the rules by formula $\Gamma \vdash \ldots$, as in

$\Gamma \vdash 0 \leq i < \textbf{procs} \wedge i \neq \textbf{rank}$. All rules but the first, that we detail below, should be easy to read.

The verification process follows the program control flow from MPI initialisation to shutdown. VCC verifies the program hand in hand with the protocol, deconstructing the protocol as the program advances. A contract for each MPI primitive enforces the premises of its inference rule (in Fig. 6), with the help of the supporting functions introduced above. Verification starts at `MPI_Init`, and makes sure that when the program calls `MPI_Finalize` the protocol reached termination, that is, it is equivalent to **skip**. Both of these primitives have associated function contracts in an annotated MPI library header that implements the verification logic. In addition, the library also provides contracts for point-to-point communications, `MPI_Send` and `MPI_Recv`, and some of the most commonly used collective communication primitives, including `MPI_Allgather`, `MPI_Allreduce`, `MPI_Bcast`, `MPI_Gather`, `MPI_Reduce`, and `MPI_Scatter`. Appendix D provides detailed contract definitions for the MPI functions.

A VCC execution reports verification errors if the input program does not conform to the desired protocol. For instance, in the finite differences example, if we change the `MPI_Scatter` root process from `0` to `1` (line 13, Fig. 2), VCC outputs the following errors:

```
fdiff.c(76,3) : error VC9502:
Call 'MPI_Scatter(data,local_n, MPI_FLOAT, &local[1], local_n,
MPI_FLOAT, 1, MPI_COMM_WORLD _(ghost _param) _(ghost _protocol)
_(out _protocol))' did not verify.
e:\aMPI\include\MPI_Scatter.h(45,14) : error VC9599: (related
information) Precondition: 'isScatter(head(in, param->rank),root)'.
...
Verification errors in 1 function(s)
```

To support the verification, we make use of two ghost variables, conveniently introduced at the entry to the program, that is, function `main`, with the following VCC signature.

```
int main(
  int argc, char** argv
  _(ghost Param* _param)
  _(ghost Protocol _protocol)
);
```

The ghost variables represent a `Protocol` instance and a `Param` instance, respectively. The latter captures the general invariant for the process ranks and for the number of processes. The definition of `Param` is as follows.

```
typedef struct {
  int rank;
  int procs;
  _(invariant rank >= 0 && rank < procs)
  _(invariant procs >= 1)
} Param;
```

Notice that no assumption is made for the process rank or, more importantly, for the number of processes[3]. This ghost parameterisation is propagated throughout the C program via the contracts of `MPI_Comm_size` and `MPI_Comm_rank`.

```
int MPI_Comm_size (... int* procs _(ghost Param* param))
  _(ensures *procs == param->procs; ...)
int MPI_Comm_rank (... int* rank _(ghost Param param))
  _(ensures *rank == param->rank; ...)
```

A call to `MPI_Init` initialises the MPI runtime. Its contract, below, sets the `protocol` ghost variable with the value of `program_protocol()` function on the number of processes.

```
int MPI_Init
( ... _(ghost Param* param) _(out Protocol protocol))
  _(ensures protocol == program_protocol(param->procs));
```

In turn, the MPI shutdown point is defined by a call to `MPI_Finalize` that has the following contract.

```
int MPI_Finalize(_(ghost Param param) _(ghost Protocol protocol))
  _(requires equivalent(protocol, param->rank, skip());
```

At the shutdown point we assert that the protocol is fully consumed by the program, i.e., that it is semantically equivalent to the empty protocol, a notion captured by the `equivalent` relation (*vide* the first rule in Fig. 6; see the corresponding VCC definition in Appendix C.4). This implies that every possible execution path of the program reaches `MPI_Finalize` by fully consuming the program protocol set at `MPI_Init`.

We illustrate the progressive matching of a program against a protocol based on the contract for the `MPI_Send` primitive, in line with the corresponding rule of Fig. 6.

```
int MPI_Send(void* data, int count, MPI_Datatype datatype, int dest, ...
             _(ghost Param* param)
             _(ghost Protocol pIn) _(out Protocol pOut)) _(
 requires isMessageFromTo(head(pIn, param->rank)), param->rank, dest);
 requires verifyData(head(pIn, param->rank), datatype, data, count);
 ensures  pOut == continuation(pIn, param->rank))
```

In the above contract, the `pIn` and `pOut` parameters respectively represent the protocol before and after an `MPI_Send` operation[4]. The two pre-conditions check that: (1) the primitive operation at the head of the protocol is a **`message`** operation from the process under verification `param->rank` to the destination `dest` process, and (2) that the data to be sent (jointly defined by the program parameters **`datatype`**, `buf`, `count`) complies with the dependent type specification for the **`message`** operation.

The send rule from Fig. 6 requires that (3) the first action to execute in the protocol is **message rank** `dest` D, (4) $0 \leq$ `dest` $<$ **procs**, (5) `dest` $\neq$ **rank**, and (6) `data` is an array with `count` elements of type `MPI_Datatype`. Our compiler for generating VCC protocols enforces that protocols comply with the rules in Fig. 4. Therefore, if the first

[3] In order to establish the invariants and use `_param` appropriately in MPI function contracts, we are forced to use a C **`struct`** datum for `Param` and a `Param*` pointer type for `_param`.

[4] VCC disallows input/output ghost parameters; we instead employ two `Protocol` arguments.

action of the protocol is **message** $i_1\ i_2\ D$, we know (7) $0 \leq i_1, i_2 <$ **procs** and that (8) $i_1 \neq i_2$. The first pre-condition clearly enforces that the type at head of the protocol is (3). Assertions (4) and (5) follow from the fact that predicate `isMessageFromTo` also enforces (9) $i_1 =$ **rank** and (10) $i_2 =$ `dest`. Therefore, from (7) and (10) we deduce (4), $0 \leq$ `dest` $<$ **procs**, and from (8) and (9) we conclude (5), `dest` $\neq$ **rank**. The second pre-condition guarantees (6), the correct type for parameter `data`.

The post-condition removes the head type from the protocol, keeping the protocol hand in hand with the program.

4.4 Annotation of program control flow

Complementing the base contracts for MPI primitives, program code requires control-flow related annotations for the `foreach` constructs. These can be semi-automatically generated by the program annotator. The **`for`** loop of the finite differences example is annotated as follows.

```
_(ghost ForeachBody _body      = foreachBody(state.protocol);
  ghost Protocol _continuation = foreachCont(state.protocol);)
for (iter  = 1; iter < max_iter; iter++) {
  _(ghost _protocol = _body[iter];)
  ...
  _(assert equivalent(_protocol, _param->rank, skip())
}
_(ghost _protocol = _continuation;)
```

The fragment illustrates the extraction of the protocols corresponding to the `foreach` body onto ghost variable `_body` (lines 11–12 in Fig. 5), and its continuation onto variable `_continuation` (lines 13–15). The verification asserts that `body` is reduced to the empty protocol by each iteration of the **`for`** loop, thus guaranteeing that the `i`'s term in the `foreach` expansion is fully consumed by the loop. After the loop, verification seeks to match the remainder of the program against protocol `continuation`.

In the finite differences protocol, the two message exchanges (line 6–9, Fig. 3) are expressed through an inner **`foreach`** that must be handled in a slightly different manner, since the sequence of actions in the protocol does not correspond to an actual loop in the program. For instance, the message exchange for rank `0` (lines 18–21, Fig. 2) must be annotated as follows:

```
_(ghost ForeachBody _ibody = foreachBody(_protocol);
_(ghost Protocol    _icont = foreachCont(_protocol);
if (rank == 0) {
  for (i = 0; i < procs; i++) {
     _(ghost _protocol = _ibody[i];)
     if (rank == i) {
       MPI_Send( ...  left ...  );
       MPI_Send( ...  right ... );
     }
     if (rank == (i == 0 ? procs-1 : i-1 ))
       MPI_Recv( ...  right ... );
     if (rank == (i == procs-1 ? 0 : i+1))
```

```
      MPI_Recv( .... left ...  );
    _(assert equivalent(_protocol, _param->rank, skip())
  }
  _(ghost _protocol = _icont;)
} else ... /* other two cases handled similarly */
```

Here we introduce a C **for** loop [5] to match the `foreach` in the protocol. The loop does not change the semantics of the program and, in any case, is used only for verification purposes, very much like all the remaining **ghost** code. The **for** loop body matches the two messages

```
message i, (i = 0 ? procs-1 : i-1) float
message i, (i = procs-1 ? 0 : i+1) float
```

in the **foreach** body. Such verification logic can also be handled in semi-automated manner by the annotator tool, as described in the next section.

To match `val` constructs, `apply(value,_(protocol) _(protocol))` annotations supply a program value, consumed by the subsequent `abs` construct. For instance, lines 4–5 in Fig. 5 must be matched by a `apply(n _(protocol) _(protocol))` annotation in the finite differences program (line 8, Fig. 2).

4.5 Memory-related annotations

As discussed earlier, we employ VCC features to type-check MPI calls' arguments and verify compliance with dependent-type restrictions in a protocol. To deal with these, however, VCC may require specific annotations in program flow or function contracts. These typically take form using VCC **reads** and **writes** clauses and through the use of built-in **\thread_local** and **\thread_local_array** predicates. The program listings in Appendix B and the MPI function contracts in Appendix D make use of these VCC constructs.

5 Software toolchain

We now describe the core traits of our software toolchain, called MPI Sessions, available from `http://gloss.di.fc.up.pt/MPISessions`.

5.1 Protocol compiler

The protocol compiler is implemented as an Eclipse plugin, a screenshot of which is depicted in Fig. 7. The tool implements an algorithmic version of the rules in Fig. 4 (cf. [26]), and uses the Z3 SMT solver for dependent-type constraint satisfaction [15]. From well-formed protocols, the tool generates corresponding VCC representations. The plugin also generates a protocol representation for WhyML parallel programs [19], and synthesises C+MPI program skeletons [13], two features not addressed in this paper. Protocols are edited in a user-friendly manner, with all the features of a modern IDE. The figure illustrates syntax highlighting and a particular error report.

[5] There can be no **ghost** loops mixed with program code.

5.2 Annotated MPI library

The annotated MPI library comprises contracts for a subset of MPI, in addition to VCC datatypes supporting protocol encoding and the program-protocol matching logic (cf. Sec. 4), totalling ∼1200 lines of VCC code. The MPI primitives include point-to-point communication, `MPI_Send` and `MPI_Recv`, and the most common collective communication operations: `MPI_Allreduce`, `MPI_Allgather`, `MPI_Bcast`, `MPI_Gather`, `MPI_Reduce`, and `MPI_Scatter`; these roughly correspond to the MPI subset supported by state-of-the-art tools [6,20,28].

5.3 Program annotator

The program annotator processes C+MPI code and adds VCC verification logic. The code listings given in appendix (B.2 and B.3) illustrate the task performed by the tool in detail for the finite differences program.

First, the annotator introduces several fragments in the code:

- an **`#include`** for the VCC protocol header, just after the inclusion of `mpi.h`;
- the ghost parameters in the `main` function of the program, `_protocol` and `_param`, in the previous section;
- ghost parameters for MPI calls, e.g., a `MPI_Send( ... )` call is transformed into `MPI_Send(... _(ghost _param) _(ghost _protocol) _(out _protocol)}`.

Moreover, the annotator processes special marks for handling control flow logic in the program. For instance, the annotator handles the outer **`foreach`** loop in the finite differences example by processing `_foreach(iter,1,max_iter) {... }` and generating in response the following code.

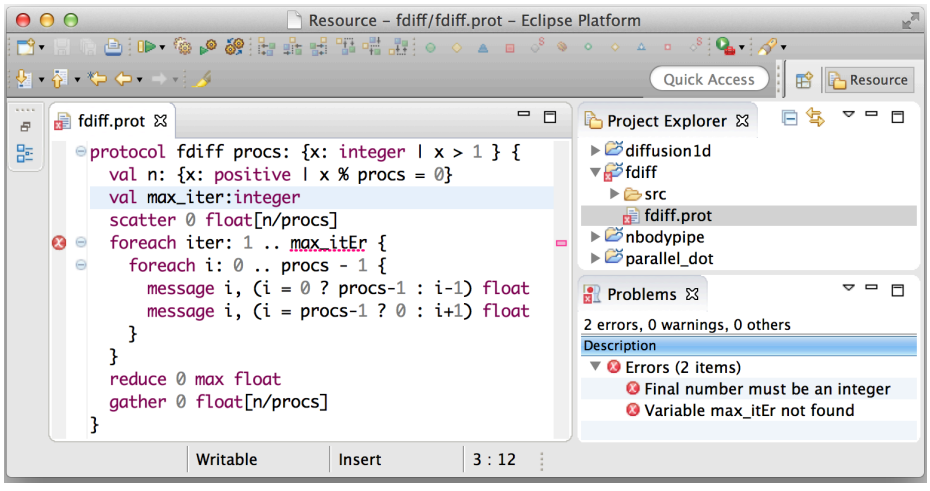

Fig. 7. Eclipse plugin for the protocol compiler

```
_(ghost ForeachBody _iter_body = foreachProtocol(_protocol);)
_(ghost Protocol    _iter_cont = foreachProtocol(_protocol);)
for (iter = 1; iter <= max_iter; iter ++) {
  _(ghost _protocol = _iter_body[iter];)
  ...
  _(assert equivalent(_protocol, _param->rank, skip()))
}
_(ghost _protocol = _iter_cont;)
```

The inner foreach loop in the same example requires `_case` marks to yield annotations similar to those shown in Sec. 4.

```
if (rank == 0) {
  _foreach(i,0,procs-1) {
    _case(rank == i) {
      MPI_Send( ...  left ...  );
      MPI_Send( ...  right ... );
    }
    _case(rank == (i == 0 ? procs-1 : i-1))
       MPI_Recv( ...  right ... );
    _case (rank == (i == procs-1 ? 0 : i+1))
       MPI_Recv( .... left ...  );
} ...
```

Finally, the annotator also expands `_apply(v)` marks, denoting the introduction of program values (cf. Sec. 4), onto `apply(v _(ghost protocol) _(out protocol))`.

6 Evaluation

6.1 Programs

We evaluated our approach using MPI programs from textbooks [5,7,17] and the FEVS suite [23], some of which are usually considered in MPI benchmark analysis (e.g., see [3,23,24]). The programs concern:

- a 1-D heat diffusion simulation [23];
- the finite differences program that we used as running example [5];
- a Jacobi iteration solver [17];
- a Laplace equation solver [23];
- a N-body simulation [7];
- and the calculation of a vector dot product [17].

Except for the vector dot product, all programs define number-crunching iterations for the problem at stake, as in the finite differences program.

6.2 Program annotation and parameterisation

We prepared each program for verification using our toolchain: (1) we wrote the corresponding protocol (given in Appendix A) and generated corresponding VCC format

using the protocol compiler; and (2) annotated their source code with the aid of the program annotator (see Appendix B for the finite differences program listings as an example). For a comparative analysis, we also annotated the programs in the TASS format, in regard to input parameters such as the number of iterations or the size of data buffers (cf. [20,24]); see the TASS listing in Appendix B.4 for the running example. There were also some adjustments due to technicalities, e.g., our framework supports **`float`** / `MPI_FLOAT` types, whereas TASS supports the **`double`** / `MPI_DOUBLE` types.

The number of processes and other input parameters impact heavily on the execution time of TASS, since the tool employs symbolic execution and model-checking techniques. For the best possible comparison (that is, the best possible case for TASS), we parameterised TASS input values as follows: just one iteration for all the iterative programs, and the minimum possible values for data buffer sizes in order for all programs to work correctly, e.g., a problem/buffer size equal to the number of processes in the finite differences program.

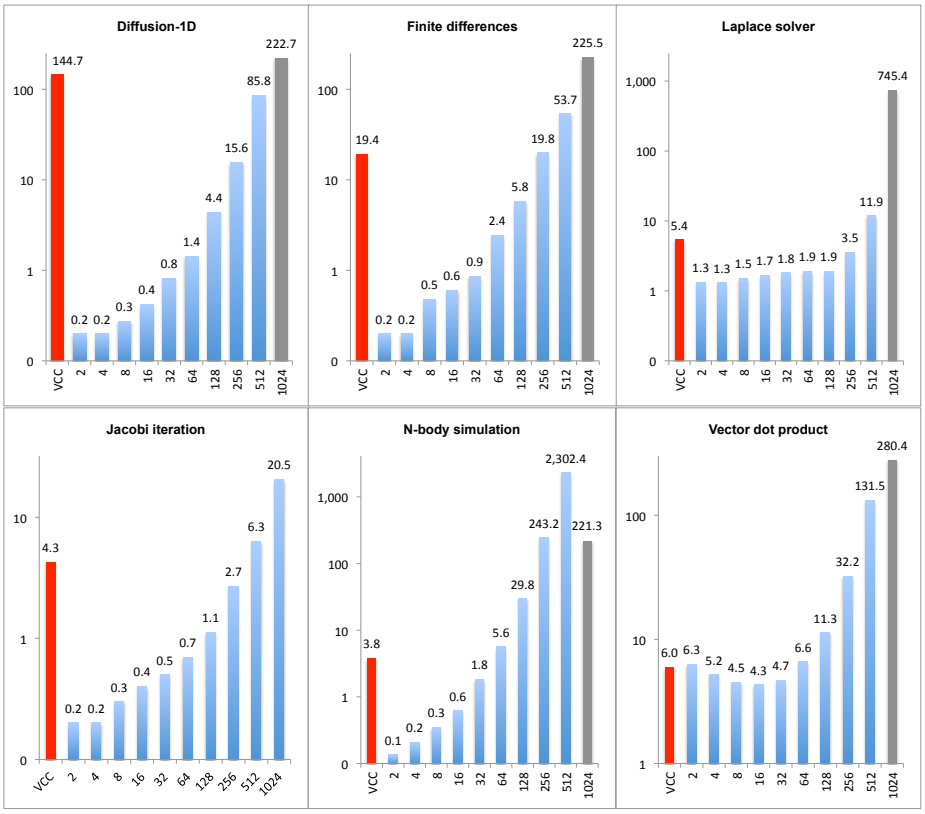

Fig. 8. Verification times for VCC and TASS (seconds)

6.3 Measurements

For each program, we measured the verification times for both VCC and TASS. Fig. 8 depicts the execution times in seconds using a logarithmic scale. The left (red) bar in each plot concerns VCC, while the right bars respect to TASS considering a parameterisation of the number of processes, ranging from 2 to 1024. The gray bars indicate that TASS terminated with an out-of-memory error, the case for all programs with a 1024-process parameterisation except the Jacobi iteration. The VCC times are the average of 10 runs measured on an Intel 2.4 GHz dual-core machine with 4 GB of RAM running Windows 7. We took similar measures for TASS version 1.1 on a MacBook 2.6 GHz quad-core Intel machine with 8 GB of RAM. Note that VCC only runs on Windows, while TASS runs on MacOS or Linux.

As the number of processes grows, the TASS execution times tend to grow exponentially and/or the memory is exhausted. This is specially observable when TASS runs with a parameterisation of more than 128 processes, where bootstrap overhead becomes negligible and actual verification effort becomes relevant. In spite of a careful design and implementation [24], TASS cannot escape the state-explosion problem, arising from the enumeration of program states inherent to model checking; moreover, the size of the representation of each program state is proportional to the number of processes subject to verification, and number of possible messages in transit for that state (cf. Sec. 5, [24]). Aside from the key observation regarding scalability, the verification times for VCC had roughly the same order of magnitude as TASS for 256/512 processes, except for the N-body simulation program where the execution times of TASS grow more rapidly.

7 Conclusion

We presented a novel methodology and toolchain for the formal verification of the communication structure of MPI programs, combining the foundation of type-based protocol verification of parallel programs [26] with the deductive software verification paradigm. A key result is that we avoid the pervasive state explosion problem found in other approaches, when it comes to verifying interaction properties such as absence of deadlocks or protocol fidelity.

Our approach captures and deals with many of the essential traits of MPI programs that challenge verification [6], in particular: the support of a core subset of some of the most traditional primitives for MPI blocking communication; the tight, intricate coupling between collective and rank-based control flow; and the possibility of verifying MPI programs that run on an arbitrary number of process.

We plan addressing several key challenges as future work: covering a larger MPI subset, e.g., support for non-blocking primitives such as `MPI_Isend` and `MPI_Irecv`; further automation in the verification process, e.g., inference of annotations related to control flow; and inclusion of more real-world programs in our test suite. Following an orthogonal approach, already ongoing work considers the synthesis of correct-by-construction MPI programs directly from protocol specifications [13], and the application of the methodology to WhyML programs [19].

References

1. Aananthakrishnan, S., Bronevetsky, G., Gopalakrishnan, G.: Hybrid approach for data-flow analysis of MPI programs. In: ICS. pp. 455–456. ACM (2013)
2. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs, LNCS, vol. 5674, pp. 23–42. Springer (2009)
3. Forejt, V., Kroening, D., Narayanswamy, G., Sharma, S.: Precise predictive analysis for discovering communication deadlocks in mpi programs. In: FM. LNCS, vol. 8442, pp. 263–278. Springer (2014)
4. Forum, M.: MPI: A Message-Passing Interface Standard—Version 3.0. High-Performance Computing Center Stuttgart (2012)
5. Foster, I.: Designing and building parallel programs. Addison-Wesley (1995)
6. Gopalakrishnan, G., Kirby, R.M., Siegel, S., Thakur, R., Gropp, W., Lusk, E., De Supinski, B.R., Schulz, M., Bronevetsky, G.: Formal analysis of MPI-based parallel programs. CACM 54(12), 82–91 (2011)
7. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message passing interface, vol. 1. MIT press (1999)
8. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: MPI runtime error detection with MUST: Advances in deadlock detection. In: SC '12. pp. 30:1–30:11. IEEE (2012)
9. Honda, K., Marques, E., Martins, F., Ng, N., Vasconcelos, V., Yoshida, N.: Verification of MPI programs using session types. In: EuroMPI. LNCS, vol. 7940, pp. 291–293. Springer (2012)
10. Honda, K., Mukhamedov, A., Brown, G., Chen, T., Yoshida, N.: Scribbling interactions with a formal foundation. Distributed Computing and Internet Technology pp. 55–75 (2011)
11. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)
12. Huang, Y., Mercer, E., McCarthy, J.: Proving MCAPI executions are correct using SMT. In: ASE. pp. 26–36. IEEE (2013)
13. Lemos, F.: Synthesis of correct-by-construction MPI programs. Master's thesis, Dep. Informática, FCUL (2014)
14. Marques, E., Martins, F., Vasconcelos, V., Ng, N., Martins, N.: Towards deductive verification of MPI programs against session types. In: PLACES. EPTCS, vol. 137, pp. 103–113 (2013)
15. Moura, L.D., N.Bjørner: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
16. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe parallel programming with message optimisation. In: TOOLS Europe. LNCS, vol. 7304, pp. 202–218. Springer (2012)
17. Pacheco, P.: Parallel programming with MPI. Morgan Kaufmann (1997)
18. Pfenning, F., Elliot, C.: Higher-order abstract syntax. SIGPLAN Not. 23(7), 199–208 (1988)
19. Santos, C.: Protocol-based programming of concurrent systems. Master's thesis, Dep. Informática, FCUL (2014)
20. Siegel, S.F., Zirkel, T.K.: Automatic formal verification of MPI-based parallel programs. In: PPoPP'11. pp. 309–310. ACM (2011)
21. Siegel, S., Gopalakrishnan, G.: Formal analysis of message passing. In: VMCAI. LNCS, vol. 6538, pp. 2–18. Springer (2011)
22. Siegel, S., Rossi, L.: Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In: EuroPVM/MPI. LNCS, vol. 5205, pp. 274–282. Springer (2008)

23. Siegel, S., Zirkel, T.: FEVS: A Functional Equivalence Verification Suite for high performance scientific computing. Mathematics in Computer Science 5(4), 427–435 (2011)
24. Siegel, S.F., Zirkel, T.K.: The Toolkit for Accurate Scientific Software. Tech. Rep. UDEL-CIS-2011/01, Department of Computer and Information Sciences, University of Delaware (2011)
25. Siegel, S.F., Zirkel, T.K.: Loop invariant symbolic execution for parallel programs. In: VMCAI. LNCS, vol. 7148, pp. 412–427. Springer (2012)
26. Vasconcelos, V., Martins, F., Marques, E., López, H., Santos, C., Yoshida, N.: Type-based verification of message-passing parallel programs. DI-FCUL 4, University of Lisbon (2014), also available at `http://www.di.fc.ul.pt/~vv/papers/tbvmppp.pdf`
27. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B.R., Schulz, M., Bronevetsky, G.: A scalable and distributed dynamic formal verifier for MPI programs. In: Supercomputing. pp. 1–10. IEEE (2010)
28. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: PPoPP. pp. 261–270. ACM (2009)

A Program protocols

A.1 Diffusion-1D

```
protocol diffusion1d p: {x: integer | x > 1} {
  val maxIter: {x: natural | x > 0 }
  val n: {x: natural | x > 0 and x % p = 0}
  broadcast 0 integer
  broadcast 0 float
  broadcast 0 integer
  broadcast 0 integer
  foreach i: 1 .. p-1
    message  0, i float[n/p]
  foreach iter:1..maxIter {
    foreach i: 1 .. p-1
      message i, i-1 float;
    foreach i: 0 .. p-2
      message i, i+1 float
  }
}
```

A.2 Laplace solver

```
protocol laplace p: {x: integer | x > 3} {
  val nx: positive
  val ny: positive
  foreach i: 1 .. p-1 {
    foreach j: 1 .. ny-2 {
      message i, 0 float[nx]
    }
  }
  message p-1, 0 float[nx]
  loop {
    foreach i: 1 .. p-1 {
      message i, i-1 float[10]
    }
    foreach i: 0 .. p-2 {
      message i, i+1 float[10]
    }
    allreduce sum float
  }
}
```

A.3 Jacobi iteration

```
protocol jacobi_iteration p: {x: integer | x > 1} {
  val n : {y: integer | y > 0 and y % p = 0};
  val maxIter : integer
  scatter 0 float[n * n]
  scatter 0 float[n]
  allgather float[n/p]
  foreach i:0 .. maxIter {
    allgather float[n/p]
  }
  gather 0 float[n/p]
}
```

A.4 N-body simulation

```
protocol nbody_simulation p: {x: integer | x > 1}{
  val n: {x: natural | x % p = 0}
  val maxIter : { x : natural | x > 0}
  foreach iter:1..maxIter {
    foreach pipe:1..p-1 {
      foreach i:1..p {
        message i, (i + 1 <= p-1 ? i+1 : 0) float[n * 4]
      }
    }
    allreduce min float
  }
}
```

A.5 Vector dot product

```
protocol vector_dot p: {x: integer | x > 1 }{
  val n: {x: integer | x > 0 and x % p = 0};
  broadcast 0 { x: integer | x = n}
  foreach i: 1 .. p-1{
    message 0, i float[n/p]
  }
  foreach i: 1 .. p-1{
    message 0,i float[n/p]
  }
  allreduce sum float
  foreach i: 1 .. p-1{
    message i, 0 float;
  }
}
```

B Finite difference program listings

B.1 MPI program

```
#include <mpi.h>
#include <stdlib.h>
#include "fdiff_aux.h"
#define N 2048
int main(int argc, char** argv) {
  // process rank; number of processes; problem size; max. iterations
  int rank, procs, n, max_iter, iter;
  float localErr, globalErr; // error variables
  float data[N], local[N+2]; // data buffers
  MPI_Status status;
  // Initialize
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  n = atoi(argv[1]);
  max_iter = atoi(argv[2]);
  if (rank == 0) read_vector(data, n); // read initial data
  // Scatter data
  int local_n = n / procs;
  MPI_Scatter(data,local_n,MPI_FLOAT,&local[1],local_n,MPI_FLOAT,0,MPI_COMM_WORLD);
  // Iterate
  int left  = rank == 0 ? procs - 1 : rank - 1; // left neighbour
  int right = rank == procs - 1 ? 0 : rank + 1; // right neighbour
  for (iter = 1; iter <= max_iter; iter++) {
    if (rank == 0) {
      MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
      MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
      MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
      MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
    } else if (rank == procs - 1) {
      MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
      MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
      MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
      MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
    } else {
      MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
      MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
      MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
      MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
    }
    localErr = fdiff_compute(local, local_n);
  }
  // Computes convergence error and final solution at rank 0
  MPI_Reduce(&localErr, &globalErr, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);
  MPI_Gather(&local[1], local_n,MPI_FLOAT,data,local_n,MPI_FLOAT,0,MPI_COMM_WORLD);
  if (rank == 0) {
    write_float("Convergence error: ", globalErr);
    write_vector(data, n);
  }
  MPI_Finalize();
  return 0;
}
```

B.2 C+MPI+VCC code – annotator input

```
#include <mpi.h>
#include "fdiff_aux.h"
#define N 2048
float fdiff_compute(float* data, int n) _(writes \array_range(data, (unsigned) n));
void read_vector(float* data, int n) _(writes \array_range (data, (unsigned) n));
void write_vector(float* data, int n) _(reads \array_range (data, (unsigned) n));
int main(int argc, char** argv) {
  int rank, n, procs, max_iter, iter,i;
  float localErr, globalErr, data[N], local[N+2];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  _(assume n % procs == 0 && n > 0 && n <= N)
  _apply(n);
  _(assume max_iter > 0 && max_iter < 100000) // upper bound avoids overflow errors
  _apply(max_iter);
  if (rank == 0) read_vector(data, n);
  int local_n = n / procs;
  MPI_Scatter(data,local_n,MPI_FLOAT,&local[1],local_n,MPI_FLOAT,0,MPI_COMM_WORLD);
  int left  = rank == 0 ? procs - 1 : rank - 1;
  int right = rank == procs - 1 ? 0 : rank + 1;
  _foreach(iter,1,max_iter)
    _(writes \array_range(local,(unsigned)local_n)) _(writes &localErr)  {
    if (rank == 0) {
      _foreach (i, 0, procs-1) {
        _case(rank == i)
        {  MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
          MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD); }
        _case(rank == (i == 0 ? procs - 1 : i - 1))
          MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
        _case(rank == (i == procs-1 ? 0 : i + 1))
          MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
      }
    } else if (rank == procs - 1) {
      _foreach (i, 0, procs-1) {
        _case(rank == (i == 0 ? procs - 1 : i - 1))
          MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
        _case(rank == (i == procs-1 ? 0 : i + 1))
          MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
        _case(rank == i)
        { MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
          MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD); }
      }
    } else {
      _foreach (i, 0, procs-1) {
        _case(rank == (i == procs-1 ? 0 : i + 1))
          MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
        _case(rank == i)
        { MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
          MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD); }
        _case(rank == (i == 0 ? procs - 1 : i - 1))
          MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
      }
    }
    localErr = fdiff_compute(local, local_n);
  }
  MPI_Reduce(&localErr, &globalErr, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);
  MPI_Gather(&local[1], local_n,MPI_FLOAT,data,local_n,MPI_FLOAT,0,MPI_COMM_WORLD);
  if (rank == 0) {
    write_float("Convergence error: ", globalErr); write_vector(data, n);
  }
  MPI_Finalize();
  return 0;
}
```

B.3 C+MPI+VCC code – annotator output

```
#include <mpi.h>
#include "fdiff_protocol.h"
#include "fdiff_aux.h"
#define N 2048
float fdiff_compute(float* data, int n) _(writes \array_range(data, (unsigned) n));
void read_vector(float* data, int n) _(writes \array_range (data, (unsigned) n));
void write_vector(float* data, int n) _(reads \array_range (data, (unsigned) n));
int main(int argc, char** argv _(ghost Param* _param) _(ghost Protocol _protocol) ) {
  int rank, n, procs, max_iter, iter,i;
  float localErr, globalErr, data[N], local[N+2];
  MPI_Status status;
  MPI_Init(&argc, &argv _(ghost param) _(out _protocol));
  MPI_Comm_rank(MPI_COMM_WORLD, &rank _(ghost _param));
  MPI_Comm_size(MPI_COMM_WORLD, &procs _(ghost _param));
  _(assume n % procs == 0 && n > 0 && n <= N)
  apply(n _(ghost _param) _(ghost _protocol) _(out _protocol));
  _(assume max_iter > 0 && max_iter < 100000) // upper bound avoids overflow errors
  apply(max_iter _(ghost _param) _(ghost _protocol) _(out _protocol));
  if (rank == 0) read_vector(data, n);
  int local_n = n / procs;
  MPI_Scatter(data,local_n,MPI_FLOAT,&local[1],local_n,MPI_FLOAT,0,MPI_COMM_WORLD
              _(ghost _param) _(ghost _protocol) _(out _protocol));
  int left  = rank == 0 ? procs - 1 : rank - 1;
  int right = rank == procs - 1 ? 0 : rank + 1;
  _(ghost ForeachBody _iter_body = foreachBody(_protocol);)
  _(ghost Protocol _iter_cont = foreachCont(_protocol);)
  for (iter = 1; iter <= max_iter; iter ++ )
    _(writes \array_range(local,(unsigned)local_n)) _(writes &localErr) {
    _(ghost _protocol = _iter_body[iter];)
    if (rank == 0) {
      _(ghost ForeachBody _i_body = foreachBody(_protocol);)
      _(ghost Protocol _i_cont = foreachCont(_protocol);)
      for (i = 0; i < procs; i++) {
        _(ghost _protocol = _i_body[i];)
        if (rank == i) {
          MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD
                   _(ghost _param) _(ghost _protocol) _(out _protocol));
          MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD
                   _(ghost _param) _(ghost _protocol) _(out _protocol));
        }
        if (rank == (i == 0 ? procs - 1 : i - 1))
          MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status
                   _(ghost _param) _(ghost _protocol) _(out _protocol));
        if (rank == (i == procs-1 ? 0 : i + 1))
          MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status
                   _(ghost _param) _(ghost _protocol) _(out _protocol));
        _(assert equivalent(_protocol, _param->rank, skip()))
      }
      _(ghost _protocol = _i_cont;)
    } else if (rank == procs - 1) {
      _(ghost ForeachBody _i_body = foreachBody(_protocol);)
      _(ghost Protocol _i_cont = foreachCont(_protocol);)
      for (i = 0; i < procs; i++) {
        _(ghost _protocol = _i_body[i];)
        if (rank == (i == 0 ? procs - 1 : i - 1))
          MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status
                   _(ghost _param) _(ghost _protocol) _(out _protocol));
        if (rank == (i == procs-1 ? 0 : i + 1))
          MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status
                   _(ghost _param) _(ghost _protocol) _(out _protocol));
        if (rank == i) {
          MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD
                   _(ghost _param) _(ghost _protocol) _(out _protocol));
          MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD
                   _(ghost _param) _(ghost _protocol) _(out _protocol));
        }
```

```
      _(assert equivalent(_protocol, _param->rank, skip()))
    }
    _(ghost _protocol = _i_cont;)
  } else {
    _(ghost ForeachBody _i_body = foreachBody(_protocol);)
    _(ghost Protocol _i_cont = foreachCont(_protocol);)
    for (i = 0; i < procs; i++) {
      _(ghost _protocol = _i_body[i];)
      if (rank == (i == procs-1 ? 0 : i + 1))
        MPI_Recv(&local[0],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status
                 _(ghost _param) _(ghost _protocol) _(out _protocol));
      if (rank == i) {
        MPI_Send(&local[1],         1, MPI_FLOAT, left, 0, MPI_COMM_WORLD
                 _(ghost _param) _(ghost _protocol) _(out _protocol));
        MPI_Send(&local[local_n],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD
                 _(ghost _param) _(ghost _protocol) _(out _protocol));
      }
      if (rank == (i == 0 ? procs - 1 : i - 1))
        MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status
                 _(ghost _param) _(ghost _protocol) _(out _protocol));
      _(assert equivalent(_protocol, _param->rank, skip()))
    }
    _(ghost _protocol = _i_cont;)
  }
  localErr = fdiff_compute(local, local_n);
  _(assert equivalent(_protocol, _param->rank, skip()))
}
_(ghost _protocol = _iter_cont;)
MPI_Reduce(&localErr, &globalErr, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD
           _(ghost _param) _(ghost _protocol) _(out _protocol));
MPI_Gather(&local[1], local_n,MPI_FLOAT,data,local_n,MPI_FLOAT,0,MPI_COMM_WORLD
           _(ghost _param) _(ghost _protocol) _(out _protocol));
if (rank == 0) {
  write_float("Convergence error: ", globalErr); write_vector(data, n);
}
MPI_Finalize(_(ghost _param) _(ghost _protocol));
_(assert \false)
return 0;
}
```

B.4 TASS version

```
#include <mpi.h>
#include <stdlib.h>
#include "fdiff_aux.h"
// Input parameters for TASS
#pragma TASS input { n > 0 } int
#define n 2048
#pragma TASS input { max_iter > 0 } int
#define max_iter 1
int main() {
  int argc; char** argv; // TASS requires these here
  int rank, procs, iter, left, right, local_n;
  double localErr, globalErr;
  double data[2048];
  double local[2048+2];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  if (rank == 0) read_vector(&data[0], n); // read initial data
  local_n = n / procs;
  MPI_Scatter(&data[0],local_n,MPI_DOUBLE,&local[1],local_n,MPI_DOUBLE,0,MPI_COMM_WORLD);
  if (rank == 0) left = procs - 1; else left = rank - 1;
  if (rank == procs-1) right = 0; else right = rank + 1;
  for (iter = 1; iter <= max_iter; iter++) {
    if (rank == 0) {
      MPI_Send(&local[1],         1, MPI_DOUBLE, left, 0, MPI_COMM_WORLD);
      MPI_Send(&local[local_n],   1, MPI_DOUBLE, right,0, MPI_COMM_WORLD);
      MPI_Recv(&local[local_n+1], 1, MPI_DOUBLE, right,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Recv(&local[0],         1, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else if (rank == procs - 1) {
      MPI_Recv(&local[local_n+1], 1, MPI_DOUBLE, right,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Recv(&local[0],         1, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Send(&local[1],         1, MPI_DOUBLE, left, 0, MPI_COMM_WORLD);
      MPI_Send(&local[local_n],   1, MPI_DOUBLE, right,0, MPI_COMM_WORLD);
    } else {
      MPI_Recv(&local[0],         1, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Send(&local[1],         1, MPI_DOUBLE, left, 0, MPI_COMM_WORLD);
      MPI_Send(&local[local_n],   1, MPI_DOUBLE, right,0, MPI_COMM_WORLD);
      MPI_Recv(&local[local_n+1], 1, MPI_DOUBLE, right,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    localErr = fdiff_compute(&local[0], local_n);
  }
  MPI_Reduce(&localErr, &globalErr, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
  MPI_Gather(&local[1], local_n, MPI_DOUBLE, &data[0],local_n,MPI_DOUBLE,0,MPI_COMM_WORLD);
  if (rank == 0) {
    write_float("Convergence error: %d", globalErr);
    write_vector(&data[0], n);
  }
  MPI_Finalize();
  return 0;
}
```

C Base VCC logic

C.1 Dependent types

```
_(ghost typedef \bool SimpleRestr[\integer];)
_(ghost typedef \bool IPredicate[int*][\integer];)
_(ghost typedef \bool FPredicate[float*][\integer];)

_(
datatype Data {
  case intRefinement  (IPredicate);
  case floatRefinement(FPredicate);
}
)
_(pure Data _float(\integer len)
  _(ensures \result == floatRefinement(\lambda float* v; \integer n; n == len))
;
)
_(pure Data _integer(\integer len)
  _(ensures \result == intRefinement(\lambda int* v; \integer n; n == len))
;
)
```

C.2 The `Protocol` datatype

```
_(ghost typedef struct _vcc_math_type_Protocol ForeachBody[\integer];)
_(ghost typedef struct _vcc_math_type_Protocol AbsBody[\integer];)
_(ghost typedef \bool IRestr[\integer];)

_(
datatype Protocol {
  case       skip ();
  case       size (IRestr);
  case        val (IRestr);
  case        abs (AbsBody);
  case        seq (Protocol, Protocol);
  case    foreach (\integer, \integer, ForeachBody);
  case    message (\integer, \integer, Data);
  case  allgather (Data);
  case  allreduce (MPI_Op, Data);
  case      bcast (\integer, Data);
  case     gather (\integer, Data);
  case     reduce (\integer, MPI_Op, Data);
  case    scatter (\integer,Data);
}
)
```

C.3 Head and continuation functions

```
_( pure Protocol head(Protocol p, \integer rank);)
_( pure Protocol continuation(Protocol p, \integer rank);)
_(
axiom \forall Protocol p1,p2,p3; \integer rank;
  equivalent(seq(p1,p2),rank,p3) ==>
    head(seq(p1,p2),rank) == head(p3,rank);
axiom \forall Protocol p1,p2,p3; \integer rank;
  equivalent(seq(p1,p2),rank,p3) ==>
    continuation(seq(p1,p2),rank) == continuation(p3,rank);
axiom \forall Data d; Protocol p; \integer rank;
  head(seq(allgather(d),p),rank) == allgather(d);
axiom \forall Data d; Protocol p; \integer rank;
  continuation(seq(allgather(d),p),rank) == p;
axiom \forall MPI_Op op; Data d; Protocol p; \integer rank;
  head(seq(allreduce(op,d),p),rank) == allreduce(op,d);
axiom \forall MPI_Op op; Data d; Protocol p; \integer rank;
  continuation(seq(allreduce(op,d),p),rank) == p;
axiom \forall Data d; Protocol p; \integer root,rank;
  head(seq(bcast(root,d),p),rank) == bcast(root,d);
axiom \forall Data d; Protocol p; \integer root,rank;
  continuation(seq(bcast(root,d),p),rank) == p;
axiom \forall IRestr d; Protocol p; \integer rank;
  head(seq(size(d),p),rank) == size(d);
axiom \forall IRestr d; Protocol p; \integer rank;
  continuation(seq(size(d),p),rank) == p;
axiom \forall Data d; Protocol p; \integer root,rank;
  head(seq(gather(root,d),p),rank) == gather(root,d);
axiom \forall Data d; Protocol p; \integer root,rank;
  continuation(seq(gather(root,d),p),rank) == p;
axiom \forall MPI_Op op; Data d; Protocol p; \integer root,rank;
  head(seq(reduce(root,op,d),p),rank) == reduce(root,op,d);
axiom \forall MPI_Op op; Data d; Protocol p; \integer root,rank;
  continuation(seq(reduce(root,op,d),p),rank) == p;
axiom \forall Data d; Protocol p; \integer root,rank;
  head(seq(scatter(root,d),p),rank) == scatter(root,d);
axiom \forall Data d; Protocol p; \integer root,rank;
  continuation(seq(scatter(root,d),p),rank) == p;
axiom \forall IRestr d; Protocol p; \integer rank;
  head(seq(val(d),p),rank) == val(d);
axiom \forall IRestr d; Protocol p; \integer rank;
  continuation(seq(val(d),p),rank) == p;
axiom \forall AbsBody b; Protocol p; \integer rank;
  head(seq(abs(b),p),rank) == abs(b);
axiom \forall AbsBody b; Protocol p; \integer rank;
  continuation(seq(abs(b),p),rank) == p;
)
```

C.4 Protocol equivalence relation

```
_(pure \bool equivalent(Protocol p1, \integer rank, Protocol p2);)
_(
axiom \forall Protocol p; \integer rank;
  equivalent(p,rank,p);
axiom \forall Protocol p; \integer rank;
  equivalent(seq(skip(),p), rank, p);
axiom \forall Protocol p1,p2,p3; \integer rank;
  equivalent(p1,rank,p2) && equivalent(p2,rank,p3) ==>
    equivalent(p1,rank,p3);
axiom \forall Protocol p1,p2,p3; \integer rank;
  equivalent(seq(seq(p1,p2),p3),rank,seq(p1,seq(p2,p3)));
axiom \forall \integer rank,from,to;Protocol p1, p2;Data d;
  (rank != from && rank != to) && equivalent(p1,rank,p2)
     ==> equivalent(seq(message(from,to,d),p1),rank,p2);
)
```

C.5 Definition of `apply`

```
_(pure \bool isVal(Protocol p);)
_(pure IRestr valRestr(Protocol p);)
_(pure \bool isAbs(Protocol p);)
_(pure Protocol evalAbs(Protocol p, \integer v);)
_(axiom \forall IRestr p; isVal(val(p));
  axiom \forall IRestr d; valRestr(val(d)) == d;
  axiom \forall AbsBody b;  isAbs(abs(b));
  axiom \forall AbsBody b; \integer v; Protocol p;
    evalAbs(seq(abs(b),skip()),v) == b[v];)
void apply
(
  int value,
  _(ghost Param* param)
  _(ghost Protocol pIn)
  _(out   Protocol pOut)
)
  _(requires isVal(head(pIn, param->rank)))
  _(requires restriction(head(pIn, param->rank))[value])
  _(requires isAbs(head(continuation(pIn, param->rank), param->rank)))
  _(ensures pOut == evalAbs(continuation(pIn, param->rank),value));
```

D MPI function contracts

In this appendix we provide the contracts for the following MPI functions: `MPI_Init` (D.1), `MPI_Finalize` (D.2), `MPI_Comm_rank` (D.3), `MPI_Comm_size` (D.4), `MPI_Send` (D.5), `MPI_Recv` (D.6), and `MPI_Scatter` (D.7). The contracts for `MPI_Allgather`, `MPI_Allreduce`, `MPI_Bcast`, `MPI_Gather`, and `MPI_Reduce` are defined in similar manner to the one provided here for `MPI_Scatter`.

D.1 `MPI_Init`

```
int MPI_Init
(
  int* argc,
  char*** argv
  _(ghost Param* param)
  _(out Protocol pOut)
)
  _(ensures pOut == program_protocol(param->rank))
  _(ensures \result == MPI_SUCCESS);
```

D.2 `MPI_Finalize`

```
int MPI_Finalize
(
  _(ghost Param* param)
  _(ghost Protocol pIn)
)
  _(requires equivalent(pIn,param->rank, skip()))
  _(ensures  \result == MPI_SUCCESS);
```

D.3 `MPI_Comm_rank`

```
int MPI_Comm_rank
(
  MPI_Comm comm,
  int* rank
  _(ghost Param* param)
)
 _(requires comm == MPI_COMM_WORLD)
 _(maintains \thread_local(rank))
 _(writes rank)
 _(ensures *rank == param->rank)
 _(ensures \result == MPI_SUCCESS);
```

D.4 `MPI_Comm_size`

Note: see the contract of `apply` in C.5 for the definition of `isAbs` and `evalAbs`.

```
_(pure \bool isSize(Protocol p);)
_(pure IRestr sizeRestr(Protocol p);)
_(axiom \forall IRestr d; isSize(size(d));
  axiom \forall IRestr d; sizeRestr(size(d)) == d;)
int MPI_Comm_size
(
  MPI_Comm comm,
  int* size
  _(ghost Param* param)
  _(ghost Protocol pIn)
  _(out Protocol pOut)
)
  _(requires comm == MPI_COMM_WORLD)
  _(requires isSize(head(pIn, param->rank)))
  _(requires isAbs(head(continuation(pIn, param->rank), param->rank)))
  _(maintains \thread_local(size))
  _(writes size)
  _(ensures *size == param->procs)
  _(ensures \result == MPI_SUCCESS)
  _(ensures sizeRestr(head(pIn,param->rank))[param->procs])
  _(ensures pOut == evalAbs(continuation(pIn,param->rank),param->procs));
```

D.5 MPI_Send

```
int MPI_Send
( void *buf, int count, MPI_Datatype type,
  int to, int tag, MPI_Comm comm
  _(ghost Param* param) _(ghost Protocol pIn) _(out Protocol pOut) )
  // Type-safe memory validation: this must be stated here
  _(requires type == MPI_INT && count == 1
    ==> \thread_local((int*) buf))
  _(requires type == MPI_INT && count > 1
    ==> \thread_local_array((int*) buf, (unsigned) count))
  _(requires type == MPI_FLOAT && count == 1
    ==> \thread_local((float*) buf))
  _(requires type == MPI_FLOAT && count > 1
    ==> \thread_local_array((float*) buf, (unsigned) count))
  // Protocol verification
  _(requires isMessageFromTo(head(pIn, param->rank),param->rank, to))
  _(requires verifyData(head(pIn, param->rank), type, data, count))
  _(ensures pOut == continuation(pIn, param->rank))
  _(ensures \result == MPI_SUCCESS);
```

D.6 MPI_Recv

```
int MPI_Recv
( void *buf, int count, MPI_Datatype type,
  int from, int tag, MPI_Comm comm, MPI_Status* status
  _(ghost Param* param) _(ghost Protocol pIn) _(out Protocol pOut) )
  // Type-safe memory validation: this must be stated here
  _(requires type == MPI_INT && count == 1
    ==> \thread_local((int*) buf))
  _(requires type == MPI_INT && count > 1
    ==> \thread_local_array((int*) buf, (unsigned) count))
  _(requires type == MPI_FLOAT && count == 1
    ==> \thread_local((float*) buf))
  _(requires type == MPI_FLOAT && count > 1
    ==> \thread_local_array((float*) buf, (unsigned) count))
  // Protocol verification
  _(requires isMessageFromTo(head(pIn,param->rank),from,param->rank))
  _(ensures verifyData(head(pIn, param->rank),buf,count,type))
  _(ensures pOut == continuation(pIn, param->rank))
  _(ensures \result == MPI_SUCCESS);
```

D.7 MPI_Scatter

```
_(pure \bool isScatter(Protocol p, \integer root);)
_(axiom \forall Data d; \integer root; isScatter(scatter(root,d),root);)
int MPI_Scatter
(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm
  _(ghost Param* param)  _(ghost Protocol pIn) _(out Protocol pOut))
_(requires sendtype == recvtype)
_(requires sendcount == recvcount)
// Type-safe memory validation: this must be stated here
_(requires recvtype == MPI_INT && recvcount > 1
  ==> \thread_local_array((int*) recvbuf, (unsigned) recvcount))
_(requires recvtype == MPI_INT && recvcount == 1
  ==> \thread_local((int*) recvbuf))
_(requires recvtype == MPI_FLOAT && recvcount > 1
  ==> \thread_local_array((float*) recvbuf, (unsigned) recvcount))
_(requires recvtype == MPI_FLOAT && recvcount == 1
  ==> \thread_local((float*) recvbuf))
_(requires sendtype == MPI_INT  && param->rank == root
  ==> \thread_local_array((int*)sendbuf,(unsigned)(sendcount*param->procs)))
_(requires sendtype == MPI_FLOAT  && param->rank == root
  ==> \thread_local_array((float*)sendbuf,(unsigned)(sendcount*param->procs)))
// Protocol verification
_(requires isScatter(head(pIn,param->rank),root))
_(requires root == param->rank
  ==> verifyData(head(pIn,param->rank),type,data,
                 (unsigned)(sendcount*param->procs)))
_(ensures pOut == continuation(pIn, param->rank))
_(ensures \result == MPI_SUCCESS);
```