Deductive Verification of Parallel Programs Using Why3

César Santos Francisco Martins Vasco Thudichum Vasconcelos
LaSIGE, Faculty of Sciences, University of Lisbon, Portugal

The Message Passing Interface specification (MPI) defines a portable message-passing API used to program parallel computers. MPI programs manifest a number of challenges on what concerns correctness: sent and expected values in communications may not match, resulting in incorrect computations possibly leading to crashes; and programs may deadlock resulting in wasted resources. Existing tools are not completely satisfactory: model-checking does not scale with the number of processes; testing techniques wastes resources and are highly dependent on the quality of the test set.

As an alternative, we present a prototype for a type-based approach to programming and verifying MPI-like programs against protocols. Protocols are written in a dependent type language designed so as to capture the most common primitives in MPI, incorporating, in addition, a form of primitive recursion and collective choice. Protocols are then translated into Why3, a deductive software verification tool. Source code, in turn, is written in WhyML, the language of the Why3 platform, and checked against the protocol. Programs that pass verification are guaranteed to be communication safe and free from deadlocks. We verified several parallel programs from textbooks using our approach, and report on the outcome.

1 Introduction

Background Message Passing Interface (MPI) [6], a standardized and portable message-passing API, is the *de facto* standard for High Performance Computing. Some of the challenges in developing correct MPI programs include: mismatches on exchanged values resulting in incorrect computations, and deadlocks resulting in wasted time and resources.

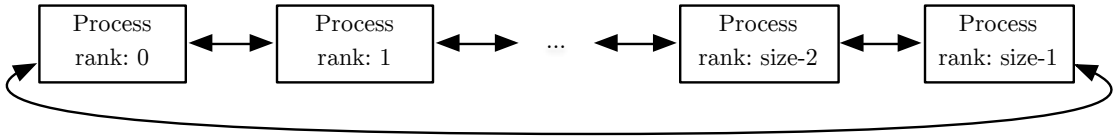
High performance computing bugs are quite costly. High-end HPC centers cost hundreds of millions to commission. On many of these centers, over 3 million dollars are spent in electricity costs alone each year and research teams apply for computer time through competitive proposals, spending years planning experiments [8]. A deadlocked program represents an exorbitant monetary cost, and such situations are hard to detect at runtime without resource wasting monitors.

The formal verification of MPI programs employs different methodologies such as runtime verification [10, 17, 18, 24] and model checking or symbolic execution [8, 10, 17, 20, 22]. Runtime verification, by its own nature, cannot guarantee the absence of faults. In addition, the process can become quite expensive due to the difficulty in producing meaningful tests. Model checking approaches typically face a scalability problem, since the verification state space grows exponentially with the number of processes. Verifying real-world applications may restrict the number of processes to only a few [21].

Motivation To illustrate the problem we present a classic MPI example that solves the finite differences problem.

S. Knight, I. Lanese, A. Lluch Lafuente and H. T. Vieira (Eds.):
8th Interaction and Concurrency Experience (ICE 2015)
EPTCS 189, 2015, pp. 128–142, doi:10.4204/EPTCS.189.11
© C. Santos, F. Martins & V.T. Vasconcelos

Finite differences is a numeric method for solving differential equations. The program starts with an initial solution X_0, and calculates $X_1, X_2, X_3, \ldots$ iteratively until a maximum number of iterations are executed. The problem vector is split amongst all processes, each calculating their part of the problem, and then joined at the end. The processes are setup in a ring topology as depicted below, in order to exchange the boundary values necessary for the calculation of the differences.



In MPI, every process (or participant) is assigned a rank; processes have access to their ranks. Ranks start at 0 and end at size −1, where size is a constant that indicates the total number of processes. The number of processes is chosen by the user at launch time.

MPI follows the Single Program Multiple Data (SMPD) model where all processes share the same code. Processes' behaviour diverge from one another based on their rank (typically via conditional statements). This model makes deployment very simple, since a single piece of code can be run on all machines.

Figure 1 shows an implementation of the finite differences problem in the C programming language (simplified for clarity). Line 2 initializes MPI, and is followed by calls to functions that return the current process rank (line 3) and the number of processes (line 4). The size of the input vector is broadcast (line 5), and the input vector at rank 0 is split and scattered (line 6) among all processes. Then, every processes iterates a certain number of times, exchanging messages with its left and right neighbors (lines 9–15) and calculating local values. Finally, process 0 calcutates the global error, and gathers all the parts of the resulting vector (lines 16–17).

This implementation deadlocks when using unbuffered communication. Process 0 attempts to send a message to its left neighbor, which in turn is attempting to send a message to its left neighbor, and so forth, with no process actually receiving the message. The correct implementation of this example requires three separate cases: one for the first process, one for the last process, and a third for all the others. These cases have specific send/ receive orders that are not at all obvious.

Solution Our approach is inspired by *multi-party session types* [12, 25], where types describe protocols. Types describe not only the data exchanged in messages, but also the state transitions of the protocol and hence the allowable patterns of message exchanges. Programs that conform to well-formed protocols are communication-safe and free from deadlocks. Our approach makes it possible to statically verify that participants communicate in accordance with the protocol, thus guaranteeing the properties above. A novel notion of type equivalence allows to type source code for individual processes against the same (global) type.

The general idea is as follows: first, a protocol is written in a type language designed for the purpose. A protocol compiler checks whether the protocol is well-formed and compiles it to a format that can be processed by a deductive program verifier. Parallel programs are then checked against the generated protocol.

```
1  int main(int argc, char** argv) {
2      MPI_Init(&argc, &argv);
3      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4      MPI_Comm_size(MPI_COMM_WORLD, &size);
5      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
6      MPI_Scatter(data, n/size, MPI_FLOAT, &local[1], n/size, MPI_FLOAT, 0, MPI_COMM_WORLD);
7      int left = rank == 0 ? size - 1 : rank - 1;
8      int right = rank == size - 1 ? 0 : rank + 1;
9      for (iter = 1; i <= ITERATIONS; iter++) {
10         MPI_Send(&local[1], 1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
11         MPI_Send(&local[n/size], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
12         MPI_Recv(&local[n/size+1], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD, &status);
13         MPI_Recv(&local[0], 1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
14         // Computation is performed here, removed for simplicity
15     }
16     MPI_Reduce(&localErr, &globalErr, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);
17     MPI_Gather(&local[1], n/size, MPI_FLOAT, data, n/size, MPI_FLOAT, 0, MPI_COMM_WORLD);
18     MPI_Finalize();
19     return 0;
20 }
```

Figure 1: An excerpt of an incorrect implementation of the finite differences problem

In line with all type-based approaches, our method requires writing a ParType (a protocol) for the program. Such a type serves as further documentation for the program. In addition, we require a few program annotations to guide the verification tool.

Method We developed a protocol language in the form of a dependent type language [23]. The language includes the most common MPI-like communication primitives, in addition to sequential composition, primitive recursion, and collective choice. Protocols are then translated into Why3 [4] a deductive software verification platform that features a rich well-defined specification language called Why. On the other hand, source code is written in a high level language with first class support for parallel MPI-like primitives, namely WhyML, a language that is part of Why3.

Why3 allows the programmer to split verification conditions in parts and prove each part using a different Satisfiability Modulo Theories (SMT) solver. In cases where the solvers cannot handle part of the proof, Why3 can generate code for use with proof assistants like Coq [2]. We chose the Why3 platform in order to avoid the annotation overhead required for static verification of C or Fortran programs, the languages typically used to program in MPI [13]. This should be considered an experiment in a new programming methodology for developing reliable parallel applications, and not a tool for verifying existing MPI applications.

Unlike other session type based approaches, our approach does not require explicit global-to-local protocol projection. This allows us to support not only MPMD programs, where the code for different ranks may be distinct, but also SPMD programs such as MPI-based ones.

Figure 2 shows the parametrized multi-party session types approach [25]. Global protocols are first projected into local protocols for each role, a communication pattern shared by one or more participants. In this case there are three roles, one for participant `0`, one for participants `1` to `size-2`, and finally one for participant `size-1`.

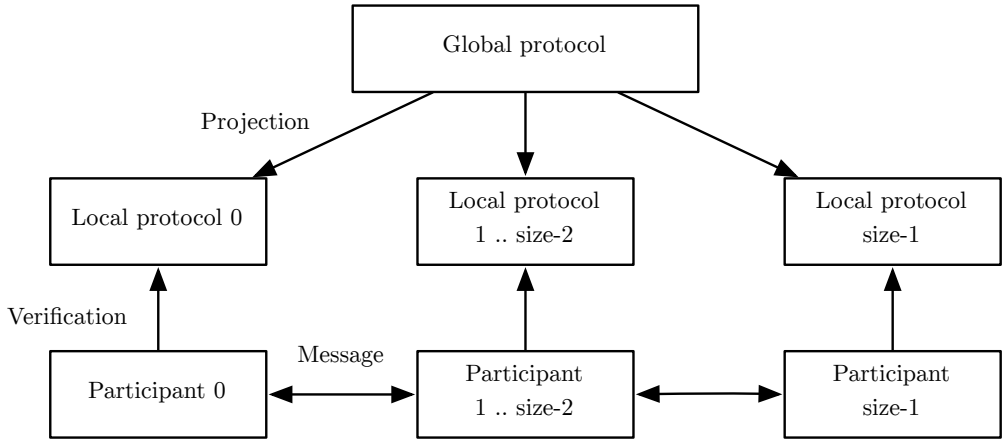
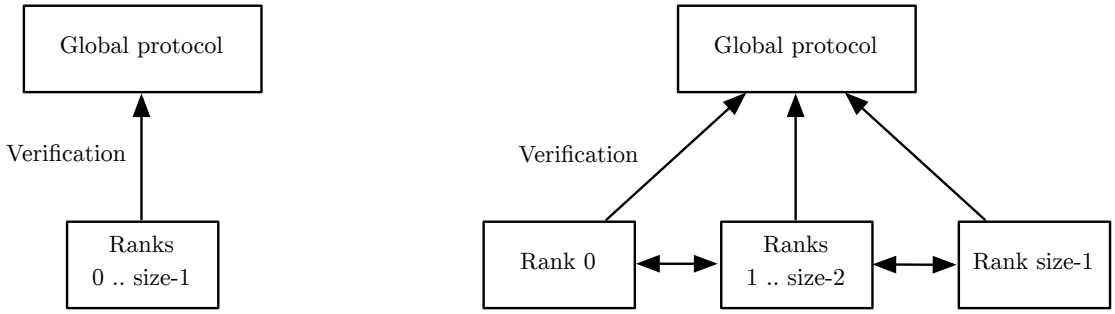


Figure 2: The multi-party session types approach



(a) SPMD (b) MPMD

Figure 3: The ParTypes approach

Figure 3 shows our approach for SPMD and MPMD. Unlike multi-party session types, the ParTypes approach does not require a separate projection step: participants are verified directly against the global protocol. Furthermore, participants can be separate programs (MPMD), or single program (SPMD).

Contributions The contributions of this work are:

- A protocol compiler (in the form of an eclipse plugin), which verifies protocol formation and translates it into Why3;
- A theory for protocols in Why3;
- An MPI-like library for parallel programming in WhyML;
- The verification of sample WhyML programs against protocols.

Outline The following two sections present the protocol language and our Why3 library for parallel programming, detailing the verification workflow. After that we present the results we

P	::= protocol $x\ p\ T$	protocol definition
T	::= message $i\ i\ D$	point-to-point comm.
	| broadcast $i\ x\colon D$	broadcast operation
	| scatter $i\ D$	scatter operation
	| gather $i\ D$	gather operation
	| reduce $op\ D$	reduce operation
	| allgather $x\colon D$	allgather operation
	| allreduce $op\ x\colon D$	allreduce operation
	| $\{T\ ...\ T\}$	sequence
	| foreach $x\colon i..i\ T$	repetition
	| if $p\ T$ else T	collective choice
	| val $x\colon D$	value
D	::= integer | float | $D[]$ | $\{x\colon D \mid p\}$ | ...	datatypes
i	::= $x \mid n \mid i+i \mid \mathsf{max}(i,i) \mid \mathsf{length}(i) \mid i[i] \mid \ldots$	index terms
p	::= true | $i \leq i \mid p$ and $p \mid a(i,\ldots,i) \mid \ldots$	index propositions
op	::= max | min | sum | ...	functions for reduce

Figure 4: Protocol language grammar

obtained when comparing our approach against a similar tool for the C programming language. After the related work section, we present our conclusions and pointers to future work.

2 Protocol language

In order to verify the finite differences example using our approach, we must first create a protocol the program must follow. The grammar for the protocol language is described in Figure 4. Not all protocols freely generated by the grammar are well formed. For instance, the from and to ranks of the **message** primitive must be distinct and lie between 0 and **size**-1. Refer to [23] for details.

The protocol for our running example is in Figure 5. Every protocol specification starts with the keyword **protocol** (line 1), followed by a protocol name, and a proposition (**size** $>= 2$) that describes the number of processes required. The variable representing the number of processes is called **size**, named after the MPI primitive (MPI_Size). The protocol requires two or more processes, in order to avoid having a single process sending messages to itself, which leads to a deadlock. This restriction can be omitted and will be inferred by the validator for simple cases.

The protocol starts by specifying a global value, the maximum number of iterations. Global values are known by every process but are not exchanged by communication. They tipically represent some relevant constants hardwired in the code or present in the command line. Such values are introduced with the keyword **val** (line 2). The value is given a name, iterations , so that it may be further used. Types in the protocol language can be refined. The language

```
1  protocol FiniteDifferences (size >= 2) {
2      val iterations: natural
3      broadcast 0 n: {x: natural | x % size = 0}
4      scatter 0 float[n]
5      foreach iter: 0 .. iterations {
6          foreach i: 0 .. size-1 {
7              message i (size+i-1)%size float
8              message i (i+1)%size float
9          }
10     }
11     reduce 0 max float
12     gather 0 float[n]
13 }
```

Figure 5: Finite differences protocol

supports some abbreviations for refined types, for example **natural**, which is any number greater than or equal to 0, is an abbreviation of {x: **integer** | x >= 0}. The type in line 3 could instead be written as: {x: {y: **integer** | y >= 0} | x % **size** = 0} or {x: **integer** | x >= 0 **and** x % **size** = 0}.

All processes perform a broadcast operation, with process `0` (the root) sending the size of the work vector to every process (line 3), followed by a **scatter** operation splitting an array of size `n` (line 4) among all ranks. These are examples of collective operations, where there is a root process sending data, and every process (including itself) receiving. Type **float**[n] is an abbreviation of {x: **float** [] | length(x) = n}. Our language allows for the specification of restrictions on the size of the array and every integer value it contains.

The reader may have noticed that, unlike broadcast, scatter does not introduce a variable. The reason for this is that the result of a scatter operation is not identical in all processes, hence could not be referred to in the rest of the protocol.

The protocol then enters a **foreach** loop (lines 5–10). The **foreach** operation is not a communication primitive, but a variant of primitive recursion: a type constructor that expands its body for a given number of iterations. The inner foreach loop (lines 6–9) is used to specify the point to point communication of every process (to exchange boundary values). This constructor allows, for example, protocols to be parametric on the number of processes. Intuitively, if we were to expand the loop, it would result in the following message exchanges.

```
message 0, size-1 float        message 1, 2 float          ...
message 0, 1 float             message 2, 1 float          message size-1, size-2 float
message 1, 0 float             message 2, 3 float          message size-1, 0 float
```

Process `0` first sends a **message** to the process on its left, then to the process on its right (lines 7–8). Process `1` does the same and so on until process **size**−1. Note that the protocol language does not require messages exchanges in the program to be globally sequential. Sequentiality restrictions happen on a per-process level.

The rest of the protocol is simple, process `0` performs a **reduce** operation (line 11), where every process sends process `0` its local error, and then it calculates the global error, which is the maximum of all local errors. Finally, process `0` collects the results of every process with a **gather** operation (line 12), thus building the final solution.

Table 1: Foreach expanded for each rank

i	Loop body	Rank 0	Rank 1	...	Rank size-2	Rank size-1
0	message 0 size-1 message 0 1	send size-1 send 1	recv 0			recv 0
1	message 1 0 message 1 2	recv 1	send 0 send 2			
2	message 2 1 message 2 3		recv 2			
...						
size-2	message size-2 size-3 message size-2 size-1				send size-3 send size-1	recv size-1
size-1	message size-1 size-2 message size-1 0	recv size-1			recv size-1	send size-2 send 0

The operations in the protocol are very close to those in the language, except that the language uses MPI_Send and MPI_Recv operations while the protocol uses **message**. Protocols provide a *global* point of view of the communication, while programs bear a *local* point of view of the communication.

If we expand the foreach loop as a table of send and receive operations, we can easily see why the example in the introduction does not conform to the protocol: Table 1 shows the message passing pattern for every process, where we have omitted the type of the message (**float**) for conciseness. There are three different orderings of **send** and **recv** operations: one for process 0, one for process **size**-1, and one for every other process. This sequencing is guaranteed not to deadlock. The example in the introduction does not match these projections.

3 Why3 theory for protocols

With the protocol out of the way, we concentrate on programming the algorithm. This ordering is not a requirement, the protocol could have been written by a different programmer, or both the protocol and the program could be developed simultaneously.

To enable the verification of MPI programs with Why3, we developed two libraries: the Why3 theory for protocols that provides a representation for protocols as a Why3 datatype and the WhyML MPI library that replicates part of the MPI API with pre and post-conditions for the various communication primitives.

Why3 theory for protocols The Why3 theory for protocols features a representation of protocols as a Why3 datatype (Figure 6). Every datatype constructor either has a continuation or takes the rest of the protocol as a parameter (except for Skip, the empty protocol), unlike the protocol language where primitives are sequenced with the sequencing operator ($\{T \mathinner{..} T\}$ in Figure 4). Continuations are implemented using the HighOrd theory of Why3, to allow values from the program to be introduced into the protocol during verification. Protocols in Why3 format are generated from the protocol language by a translator that first checks the good formation of types generated by the grammar in Figure 4. Each datatype constructor corresponds to a type constructor. Primitives that introduce values use the continuation format,

```
1  type protocol =
2        Val datatype continuation
3      | Broadcast int datatype continuation
4      | AllGather datatype continuation
5      | AllReduce op datatype continuation
6      | Scatter int datatype protocol
7      | Gather int datatype protocol
8      | Message int int datatype protocol
9      | Reduce int op datatype protocol
10     | If datatype protocol protocol protocol
11     | Foreach int int (cont int) protocol
12     | Skip
13 with
14     op = Max | Min | Sum | Prod | Land | Band | Lor | Bor | Lxor | Bxor
15 with
16     datatype =
17       IntPred (pred int)
18     | FloatPred (pred float)
19     | ArrayIntPred (pred (array int))
20     | ArrayFloatPred (pred (array float))
21 with
22     continuation =
23       IntAbs (cont int)
24     | FloatAbs (cont float)
25     | ArrayIntAbs (cont (array int))
26     | ArrayFloatAbs (cont (array float))
27 with
28     cont = func a protocol
```

Figure 6: The Why3 datatype for protocols

while those that do not feature the rest of the protocol as a parameter. Both kinds feature a datatype (D in Figure 4) representing the data exchanged.

The type pred a used in the datatype constructors (lines 17–20) is an abbreviation of func a bool, a function of any type to a boolean. Such a predicate is used to restrict values, encoding refinement datatypes ($\{x\colon D \mid p\}$ in Figure 4) This type abbreviation is part of the Why3 standard library. Similarly, the type cont a used in the continuation constructors (lines 23–26) is an abbreviation of func a protocol, a function of any type to a protocol (which is the continuation, used to introduce values into the protocol).

WhyML MPI library The WhyML MPI library includes MPI-like primitives such as init, broadcast, scatter, gather, send, recv, as well as *annotations* foreach, expand, and isSkip required to guide the verification process. In order to check that the program follows the protocol, each Why3/MPI primitive is annotated with pre and post-conditions.

The init primitive initializes the verification state, a structure used during the verification process. The verification state has a single mutable field containing the protocol datatype. Every ParTypes primitive takes the verification state as a parameter, verifies that the protocol is correct for the primitive and, if there are no errors, updates the protocol field of the verification state with the protocol continuation. Some of the ParTypes primitives and their annotations can be seen in Figure 7.

```
1  val send (dest:int) (v:'a) (s:state) : ()
2    writes { s.protocol }
3    requires { 0 <= dest /\ dest < size }
4      requires { match project s.protocol with
5             | Message src dst d _ -> src = rank /\ dest = dst /\ matches v d
6             | _ -> false
7                end }
8    ensures { s.protocol = next (old s).protocol }
9
10 val apply (v:'a) (s:state) : 'a
11     writes { s.protocol }
12     requires { match project s.protocol with
13                | Val d _ -> matches v d
14                | _ -> false
15                end }
16     ensures { s.protocol = continuation (old s).protocol v }
17     ensures { result = v }
18
19 val foreach (s:state) : foreach_data
20     writes { s.protocol }
21     requires { match project s.protocol with
22                | Foreach _ _ _ _ -> true
23                | _ -> false
24                end }
25     ensures { s.protocol = next (old s).protocol }
26     ensures { result = (
27         foreach_body (old s).protocol,
28         foreach_from (old s).protocol,
29         foreach_to (old s).protocol )}
30
31 val expand (fd:foreach_data) (i:int) : state
32     requires { let _,low,high = fd in low <= i <= high }
33     ensures { let body,_,_ = fd in result = {protocol = body i} }
34
35 function project (p:protocol) : protocol =
36     match p with
37         | Message source dest _ remainingProtocol ->
38             if rank <> source /\ rank <> dest
39                 then project remainingProtocol
40                 else p
41         | _ -> p
42     end
```

Figure 7: WhyML MPI-like library (excerpt)

Every ParTypes primitive calls the utility function `project` when verifying the protocol (lines 35–42). Obtaining the projection of a protocol yields the protocol itself in most cases. The exception is when the protocol is a message that is neither originating in nor addressed to the rank in question. In this case the function recurs, skipping messages unrelated to the current rank. Note that rank is not passed as a parameter, instead the `rank` constant is handled automatically by the Why3 verification process.

The send primitive (lines 1–8), receives the target rank, a value, and the current state. It

```
1  let main () =
2      let s = init fdiff_protocol in
3      let iterations = apply 100000 s in
4      let n = broadcast 0 input s in
5      let local = scatter 0 work s in
6      let left = if rank > 0 then rank-1 else size-1 in
7      let right = if rank < size-1 then rank+1 else 0 in
8      for iter = 1 to iterations do
9          let inbody = expand (foreach s) iter in (* Annotation *)
10         let body = foreach inbody in (* Annotation *)
11         if (rank = 0) then (
12             let f1 = expand body 0 in (* Annotation *)
13             send left local[1] f1;
14             send right local[n/size] f1; isSkip f1; (* Annotation *)
15             let f2 = expand body 1 in (* Annotation *)
16             local[n/size+1] <- recv right f2; isSkip f2; (* Annotation *)
17             let f3 = expand body (np-1) in (* Annotation *)
18             local[0] <- recv left f3; isSkip f3); (* Annotation *)
19         else if (rank = size-1) then (
20             let f1 = expand body 0 in (* Annotation *)
21             local[lsize+1] <- recv right f1; isSkip f1; (* Annotation *)
22             let f2 = expand body (np-2) in (* Annotation *)
23             local[0] <- recv left f2; isSkip f2; (* Annotation *)
24             let f3 = expand body (np-1) in (* Annotation *)
25             send left local[1] f3;
26             send right local[lsize] f3; isSkip f3); (* Annotation *)
27         else (
28             let f1 = expand body (rank-1) in (* Annotation *)
29             local[0] <- recv left f1; isSkip f1; (* Annotation *)
30             let f2 = expand body rank in (* Annotation *)
31             send left local[1] f2;
32             send right local[lsize] f2; isSkip f2; (* Annotation *)
33             let f3 = expand body (rank+1) in (* Annotation *)
34             local[lsize+1] <- recv right f3; isSkip f3); (* Annotation *)
35         isSkip inbody; (* Annotation *)
36         (* Computation is performed here, removed for simplicity *)
37     done;
38     globalerror := reduce 0 Max !localerror s;
39     gather 0 local s;
40     isSkip s; (* Annotation *)
```

Figure 8: The WhyML program for the corrected finite differences example

checks that the destination is a valid process (line 3), that the current verification state starts with a Message after calling the project function (with the current rank as the source and the same destination as in the program), and that the value being sent matches the refinement (line 5). The send primitive returns nothing and ensures that the next state is the protocol after the message (line 8).

The apply primitive is used to introduce program values into the protocol. It checks that the head of the protocol is a Val, and that the value introduced matches the restriction (line 13). The contract ensures that the protocol becomes the continuation of the Val constructor, after applying the value (line 16). Since a value is introduced, this primitive uses `continuation` instead

of next, which does not introduce values.

Finally, the foreach primitive, requires that the protocol must have a Foreach constructor at the head (line 22). It ensures that the protocol continues with whatever follows Foreach (line 25), updates the verification state, and returns a triple containing the body of the foreach and its range (lines 26–29). The expand function takes the output of the foreach primitive and an integer i, checks that the integer is in range (line 33), and returns the projection of the Foreach for that integer (line 34).

The ParTypes primitives apply and foreach are not related to any MPI primitive, they are simply annotations required to match the program against a protocol.

Checking WhyML code Figure 8 shows the finite differences example written in WhyML, following the three separate send/receive orderings. The necessary annotations are all marked with (∗ Annotation ∗). On line 2, the verification state is initialized with the finite differences protocol, the result of translating the protocol in Figure 5 into a Why3 protocol type (an instance of the type in Figure 6, not shown). On line 3, the apply primitive is used to introduce the number of iterations into the protocol, consuming the Val constructor at the head. The subsequent lines perform a broadcast and a scatter following the protocol, while consuming the corresponding constructors.

On line 10 the foreach and expand primitives are called to obtain the body of the Foreach constructor, with the current iteration count applied. The Foreach body contains another Foreach loop (see Figure 5), but that loop does not correspond to a loop in the program, instead, it is used to define the behavior of every process. There are three different send/recv orderings, following the message projections in Table 1. Each branch (lines 11–34) corresponds to one of the projections, with the Foreach constructor expanded for all intervening processes: the process on the left, the process itself, and the process on the right (lines 12, 15 and 17 for example). The isSkip function verifies that each of these projections is equivalent to Skip at that point in the code (lines 14, 16 and 18 for example), to guarantee the protocol is followed correctly (see [23] for the theory). The rest of the program is simple, with a final isSkip at the end (line 40) to guarantee the protocol was completely consumed.

4 Evaluation

We adapted a few classic parallel programming examples to WhyML, wrote their protocols, and checked them with Why3. To evaluate the results, we compared verification times and the ratio of annotations or of code. The closest work to ours checks programs written in C+MPI with VCC [13]. The experimental setup was a 2,4 GHz Intel Core 2 Duo machine running Windows 7 with 4 GB of RAM.

Sample programs We verified the following programs:

- **Pi**: a simple program that calculates an approximation of pi through numerical integration, taken from [9].
- **Finite differences**: used as our running example. The code is adapted from [7].
- **Parallel dot**: calculates the dot product of two vectors, taken from [16].

Program	Why3 Sub-Proofs	Why3 Time (s)	VCC Time (s)	Why3/VCC
Pi	27	1,6	2,4	66,7%
Finite differences	374	14,9	16,1	92,5%
Parallel dot	298	7.9	7,4	106,7%

Table 2: Results for Why3 and VCC verification times

Program	Why3 LOC	Why3 Anot	Ratio	VCC LOC	VCC Anot	Why3/VCC
Pi	33	6	18%	42	10	23%
Finite differences	86	29	33%	128	49	38%
Parallel dot	61	11	18%	99	30	30%

Table 3: Results for Why3 and VCC annotation requirements

Verification time The verification times obtained are in Table 2 (average of 10 runs). Though Why3 can be used interactively, all the runs were automated, and no manual proofs were necessary. As can be seen from the results, Why3 and VCC have similar performance. This is surprising as Why3 spawns a different Z3 process for each sub-proof. A possible explanation for the similarity is that each individual sub-proof is substantially easier on the solver, and VCC has to perform more verifications than Why3 due to concurrency and pointer related proofs.

The results are promising, more so since proofs can be done in parts if necessary.

Annotation effort The ratio for annotations can be seen in Table 3. The lines of code (LOC) count ignores library imports, comments, and empty lines. VCC requires more annotations than Why3 due to concurrency and pointer related annotations, but a lot of these can be automated with an annotator or by employing C macros. That said, something similar could be done for Why3. The only annotations the programmer would have to write would be foreach and collective choice marks, greatly reducing the effort required to use our verification methodology.

5 Related Work

Scribble [11] is a language to describe protocols for message-based programs based on the theory of multiparty session types [12]. 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 [15].

Pabble [14] is a parametric extension of Scribble, which adds indices to participants and represents Scribble protocols in a compact and concise notation for parallel programming. Pabble protocols can represent the interaction patterns of scalable MPI programs, where the number of participants in a protocol is decided at runtime through parameters.

In this work we depart from multiparty session types along two distinct dimensions: a) our protocol language is specifically built for MPI primitives, and b) we do not explicitly project a protocol but else check the conformance of code directly against a global protocol.

Tools for the verification of MPI programs The objectives of MPI verification tools are diverse and include the validation of arguments to MPI primitives as well as resource usage [24], ensuring interaction properties such as the absence of deadlocks [17, 20, 24], or asserting functional equivalence to sequential programs [20]. The methodologies employed are also diverse, ranging from traditional dynamic analysis up to model checking and symbolic execution. In comparison, our methodology is based on type checking and deductive program verification, thus avoiding testing and the state-explosion problem inherent to the model checking approaches described below.

Model checking TASS [22] employs model checking and symbolic execution, but is also able to verify user-specified assertions for the interaction behavior of the program, so-called collective assertions, and to verify functional equivalence between MPI programs and their sequential counterparts. The approach performs a number of checks besides deadlock detection (such as buffer overflows and memory leaks), but, as expected, does not scale with the number of processes.

MOPPER [5] is a verifier that detects deadlocks by checking formulae satisfiability, obtained by analyzing execution traces of MPI programs. It uses a propositional encoding of constraints and partial order reduction techniques, obtaining significant speedups when compared with 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.

CIVL [19] is a model checker that uses a C-like unified intermediate verification language for specifying concurrency in message-passing, multi-threaded, or GPU languages. The tool uses model checking techniques and symbolic execution to detect deadlocks, assertions and bounds violations, as well as illegal memory usages. The tool's underlying CVC3 theorem prover may fail due to state space explosion, and the user needs to specify input bounds on the command line to specify a finite subset of state space.

Runtime verification and testing ISP [17] is a deadlock detection tool that explores all possible process interleaving using a fixed test harness. Dynamic execution analyzers, such as DAMPI [24] and MUST [10], strive for the runtime detection of deadlocks and resource leaks.

6 Conclusion

We developed an eclipse plugin for the development and validation of protocols, and a programming language for the development of parallel programs by adding MPI-like primitives to WhyML. We also developed a Why3 theory of protocols for the verification of MPI-like WhyML programs. With this approach we can ensure that programs that pass the Why3 verification are free from deadlocks, all message exchanges are type safe, and the program adheres to the protocol.

Unlike model checkers (such as TASS [22]), our approach scales to any number of processes, running in constant time. No runtime verification of the software is necessary as in ISP [17], DAMPI [24] or MUST [10]. These tools do not require protocols and typically require less program annotations, but the runtime verifiers require a good test suite which is much harder to write than a protocol. Unlike Scribble [11], our approach can model MPI-like programs, including collective choices without communication.

Previous work used VCC [3] to verify C+MPI programs. The approach is very similar to ours, but requires extra annotations regarding concurrency and pointers since VCC is a tool for verifying concurrent C programs. The annotations are also more complex, while in our approach are more natural and fit with the code.

The foreach primitive annotation requires familiarity with how the foreach primitive is expanded, but writing correct programs already implies that sort of mental reasoning. Naïve approaches most likely result in deadlocks, as we illustrated in the finite differences example (Figure 1). The VCC based approach shares these problems.

We successfully verified a number of textbook examples of parallel programs, with verifications taking only a few seconds in the worst case, and none of the examples required manual proofs.

Our prototype is based on a verification language, WhyML, that is not an appropriate language for industry use. While OCaml programs can be extracted from WhyML, OCaml is not a language typically used in high performance computing. Performance is the major consideration in high performance computing, and Fortran and C are the fastest high-level languages available. To tackle these issues (including the annotation problem), an appropriate language should be developed. This language, like our WhyML based language, would have first class parallel programming primitives, but it would be essentially a C or Fortran superset with restrictions. This language would compile to either C or Fortran, and by having essentially the same semantics, performance should be the same.

Finally, other MPI primitives need to be supported, such as asynchronous communication primitives, topologies, communicators and wild card receive.

Acknowledgments. This work is supported by FCT through project Advanced Type Systems for Multicore Programming and project Liveness, Statically (PTDC/EIA-CCO/122547 and 117513/2010) and the LaSIGE lab (PEst-OE/EEI/UI0408/2011). We would like to thank Dimitris Mostrous for his insightful comments.

References

[1] S. Aananthakrishnan, G. Bronevetsky & G. Gopalakrishnan (2013): *Hybrid approach for data-flow analysis of MPI programs.* In: *ICS*, ACM, pp. 455–456, doi:10.1145/2464996.2467286.

[2] Y. Bertot, P. Castéran, G. Huet & C. Paulin-Mohring (2004): *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions.* Texts in theoretical computer science, Springer, doi:10.1007/978-3-662-07964-5.

[3] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte & S. Tobies (2009): *VCC: A practical system for verifying concurrent C.* In: *TPHOLs*, *LNCS* 5674, Springer, pp. 23–42, doi:10.1007/978-3-642-03359-9_2.

[4] J. Filliâtre & A. Paskevich (2013): *Why3 - Where programs meet provers.* In: *ESOP*, *LNCS* 7792, Springer, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[5] V. Forejt, D. Kroening, G. Narayanswamy & S. Sharma (2014): *Precise predictive analysis for discovering communication deadlocks in MPI programs.* In: *FM*, *LNCS* 8442, Springer, pp. 263–278, doi:10.1007/978-3-319-06410-9_19.

[6] MPI Forum (2012): *MPI: A message-passing interface standard—Version 3.0.* High-Performance Computing Center Stuttgart.

[7] I. Foster (1995): *Designing and building parallel programs.* Addison-Wesley.

[8] G. Gopalakrishnan, R.M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B.R. De Supinski, M. Schulz. & G. Bronevetsky (2011): *Formal analysis of MPI-based parallel programs.* *CACM* 54(12), pp. 82–91, doi:10.1145/2043174.2043194.

[9] W. Gropp, E. Lusk & A. Skjellum (1999): *Using MPI: portable parallel programming with the message passing interface.* MIT press.

[10] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski & M. S. Müller (2012): *MPI runtime error detection with MUST: advances in deadlock detection.* In: *SC*, IEEE/ACM, p. 30, doi:10.3233/SPR-130368.

[11] K. Honda, A. Mukhamedov, G. Brown, T. Chen & N. Yoshida (2011): *Scribbling interactions with a formal foundation.* In: *ICDCIT*, *LNCS* 6536, Springer, pp. 55–75, doi:10.1007/978-3-642-19056-8_4.

[12] K. Honda, N. Yoshida & M. Carbone (2008): *Multiparty asynchronous session types.* In: *POPL*, ACM, pp. 273–284, doi:10.1145/1328438.1328472.

[13] E. R. B. Marques, F. Martins, V. T. Vasconcelos, C. Santos, N. Ng & N. Yoshida (2015): *Protocol-based verification of MPI programs.* DI-FCUL 2, University of Lisbon, doi:10455/6901.

[14] N. Ng & N. Yoshida (2014): *Pabble: Parameterised Scribble for parallel programming.* In: *PDP*, IEEE Computer Society, pp. 707–714, doi:10.1109/PDP.2014.20.

[15] N. Ng, N. Yoshida & K. Honda (2012): *Multiparty Session C: Safe parallel programming with message optimisation.* In: *TOOLS Europe*, *LNCS* 7304, Springer, pp. 202–218, doi:10.1007/978-3-642-30561-0_15.

[16] P. S. Pacheco (1997): *Parallel programming with MPI.* Morgan Kaufmann.

[17] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Palmer, R. Thakur & W. Gropp (2007): *Practical model-checking method for verifying vorrectness of MPI programs.* In: *PVM/MPI*, *LNCS* 4757, Springer, pp. 344–353, doi:10.1007/978-3-540-75416-9_46.

[18] M. Schulz & B. R. de Supinski (2006): *A flexible and dynamic infrastructure for MPI tool interoperability.* In: *ICPP'06*, IEEE, pp. 193–202, doi:10.1109/ICPP.2006.6.

[19] S. F. Siegel, M. B. Dwyer, G. Gopalakrishnan, Z. Luo, Z. Rakamaric, R. Thakur, M. Zheng & T. K. Zirkel (2014): *CIVL: The concurrency intermediate verification language.* Technical Report UD-CIS-2014/001, Department of Computer and Information Sciences,University of Delaware.

[20] S. F. Siegel & G. Gopalakrishnan (2011): *Formal analysis of message passing.* In: *VMCAI*, *LNCS* 6538, Springer, pp. 2–18, doi:10.1007/978-3-642-18275-4_2.

[21] S. F. Siegel & L. F. Rossi (2008): *Analyzing BlobFlow: A case atudy using model checking to verify parallel scientific software.* In: *PVM/MPI*, *LNCS* 5205, Springer, pp. 274–282, doi:10.1007/978-3-540-87475-1_37.

[22] S. F. Siegel & T. K. Zirkel (2012): *Loop invariant symbolic execution for parallel programs.* In: *VMCAI*, *LNCS* 7148, Springer, pp. 412–427, doi:10.1007/978-3-642-27940-9_27.

[23] V. T. Vasconcelos, F. Martins, E. R. B. Marques, H. A. López, C. Santos & N. Yoshida (2015): *Type-based verification of message-passing parallel programs.* DI-FCUL 1, University of Lisbon, doi:10455/6902.

[24] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz & G. Bronevetsky (2010): *A scalable and distributed dynamic formal verifier for MPI programs.* In: *SC*, IEEE, pp. 1–10, doi:10.1109/SC.2010.7.

[25] N. Yoshida, P. Deniélou, A. Bejleri & R. Hu (2010): *Parameterised multiparty session types.* In: *FoSSaCs*, *LNCS* 6014, Springer, pp. 128–145, doi:10.1007/978-3-642-12032-9_10.