16

Deductive Verification of MPI Protocols

Vasco T. Vasconcelos[1], Francisco Martins[1], Eduardo R. B. Marques[2], Nobuko Yoshida[3] and Nicholas Ng[3]

[1]LaSIGE, Faculty of Sciences, University of Lisbon, PT
[2]CRACS/INESC-TEC, Faculty of Sciences, University of Porto, PT
[3]Imperial College London, UK

Abstract

This chapter presents the ParTypes framework to statically verify C programs that use the Message Passing Interface, the widely used standard for message-based parallel applications. Programs are checked against a protocol specification that captures the interaction in an MPI program. The protocol language is based on a dependent type system that is able to express various MPI communication primitives, including point-to-point and collective operations. The verification uses VCC, a mechanical verifier for concurrent C programs. It takes the program protocol written in VCC format, an annotated version of the MPI library, and the program to verify, and checks whether the program complies with the protocol.

16.1 Introduction

Message Passing Interface (MPI) [3] is a portable message-passing API for programming parallel computers running on distributed memory systems. To these days, MPI remains the dominant framework for developing high performance parallel applications.

Usually written in C or Fortran, MPI programs call library functions to perform point-to-point send/receive operations, collective and synchronisation operations (such as broadcast and barrier), and combination of partial results of computations (gather and reduce operations). Developing MPI applications is an error-prone endeavour. For instance, it is quite easy to write programs that cause processes to wait indefinitely for a message, or that exchange data of unexpected sorts or lengths.

Verifying that MPI 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 runtime verification [7, 15, 16, 21] and model checking [5–7, 15, 17, 20]. These approaches frequently stumble upon the problem of scalability since the search space grows exponentially with the number of processes. It is often the case that the verification of real applications limits the number of processes to less than a dozen [18].

We approach the problem of verifying C+MPI code using a type theory for parallel programs. In our framework—PARTYPES—types describe the communication behaviour programs, that is, protocols. Programs that conform to one such type are guaranteed to follow the protocol and not to run into deadlocks. The verification is scalable, as it does not depend on the number of processes or other input parameters.

A previous work introduces the type theory underlying protocol specification, shows the soundness of the methodology by designing a core language for parallel programming and proving a progress result for well-typed programs, and provides a comparative evaluation of PARTYPES against other state-of-the-art tools [10].

This chapter takes a pragmatic approach to the verification of C+MPI code, by explaining the procedure from the point of view of someone interested in verifying actual code, omitting theoretic technical details altogether. Protocols are written in a dependent type language that includes specific constructors for some of the most common communication primitives found in MPI programs. The conformance of a program against a protocol is checked using VCC, a software verifier for the C programming language [1]. In a nutshell, one checks C+MPI source code against a protocol as follows:

1. Write a protocol for the program, that can be translated mechanically to VCC format;
2. Introduce special, concise marks in the C+MPI source code to guide the automatic generation of VCC annotations required for verification;
3. Use the VCC tool to check conformance of the source code against the protocol.

If VCC runs successfully, then the program is guaranteed to follow the protocol and to be exempt from deadlocks, regardless of the number of processes, problem dimension, number of iterations, or any other parameters. The verification process is guided by two tools—the Protocol Compiler and the Annotation Generator—and by the PARTYPES MPI library. All these can be found at the PARTYPES website [14]. The tools and the library almost completely insulate the user from working with the VCC language.

The rest of this chapter is organised as follows. The next section introduces a running example and discusses typical faults found in MPI programs. Then Section 16.3 describes the protocol language and Sections 16.4 and 16.5 provide an overview of the verification process. Section 16.6 discusses related work and Section 16.7 concludes the chapter.

16.2 The Finite Differences Algorithm and Common Coding Faults

This section introduces a running example and discusses common pitfalls encountered when developing MPI programs.

The *finite differences* algorithm computes an approximation of derivatives by the finite difference method. Given an initial vector X_0, the algorithm calculates successive approximations to the solution $X_1, X_2, \ldots,$ until a pre-defined maximum number of iterations has been reached. A distinguished process, say the one with rank 0, disseminates the problem size (that is, the length of array X) through a broadcast operation. The same process then divides the input array among all processes. Each participant is responsible for computing its local part of the solution. When the pre-defined number of iterations is reached, process rank `0` obtains the global error through a reduce operation and collects the partial arrays in order to build a solution to the problem (Figure 16.1, left). In order to compute its part of the solution, each process exchanges boundary values with its left and right neighbours on every iteration (Figure 16.1, right).

Figure 16.2 shows C+MPI source code that implements the finite differences algorithm, adapted from a textbook [4]. The `main` function describes the

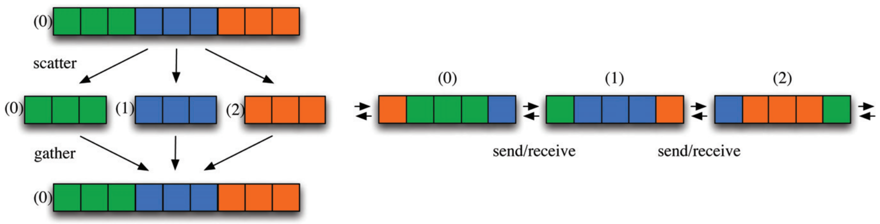


Figure 16.1 Communication pattern for the finite differences algorithm.

```
1  int main(int argc, char** argv) {
2   int rank, procs, n; // process rank; number of processes; problem size
3   ...
4   MPI_Init(&argc, &argv);
5   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6   MPI_Comm_size(MPI_COMM_WORLD, &procs);
7   if (rank == 0) {
8    n = read_problem_size(procs);
9    read_vector(work, n);
10  }
11  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
12  int local_n = n / procs;
13  MPI_Scatter(work, local_n, MPI_FLOAT, &local[1], local_n, MPI_FLOAT, ...);
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; iter <= NUM_ITER; iter++) {
17    ...
18    if (rank == 0) {
19      MPI_Send(&local[1],         1, MPI_FLOAT, left, ...);
20      MPI_Send(&local[local_n],   1, MPI_FLOAT, right, ...);
21      MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right, ...);
22      MPI_Recv(&local[0],         1, MPI_FLOAT, left, ...);
23    } else if (rank == procs - 1) {
24      MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right, ...);
25      MPI_Recv(&local[0],         1, MPI_FLOAT, left, ...);
26      MPI_Send(&local[1],         1, MPI_FLOAT, left, ...);
27      MPI_Send(&local[local_n],   1, MPI_FLOAT, right, ...);
28    } else {
29      MPI_Recv(&local[0],         1, MPI_FLOAT, left, ...);
30      MPI_Send(&local[1],         1, MPI_FLOAT, left, ...);
31      MPI_Send(&local[local_n],   1, MPI_FLOAT, right, ...);
32      MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right, ...);
33    }
34    ...
35  }
36  MPI_Reduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, 0, ...);
37  MPI_Gather(&local[1], local_n, MPI_FLOAT, work, local_n, MPI_FLOAT,0,...);
38  ...
39  MPI_Finalize();
40  return 0;
41 }
```

Figure 16.2 Excerpt of an MPI program for the finite differences problem.

behaviour of all processes together; the behaviour of each individual process may diverge based on its process number, designated by *rank*, and set on line 5 using the `MPI_Comm_rank` primitive. The number of processes (`procs` in the figure) is obtained through primitive `MPI_Comm_size` on line 6. Rank `0` starts by reading the problem size and the corresponding input vector X_0 (lines 8–9, variables `n` and `work`). The same participant then broadcasts the problem size (line 11, call to `MPI_Bcast`) and distributes the input vector to all other participants (line 13, call to `MPI_Scatter`).

Each participant is then responsible for computing its part of the solution. The program enters a loop (lines 16–35), specifying point-to-point message exchanges (`MPI_Send`, `MPI_Recv`) between each process and its `left` and `right` neighbours, based on a ring topology. The various message exchanges distribute boundary (`local[0]` and `local[local_n+1]`) values necessary to local calculations. Different send/receive orders for different ranks (lines 19–22, lines 24–27, and lines 29–32) aim at avoiding deadlock situations (`MPI_Send` and `MPI_Recv` are blocking, synchronous, unbuffered operations). The loop ends when a pre-defined number of iterations is attained. Once the loop is over, rank `0` computes the global error through a reduction operation (`MPI_reduce`, line 36) and gathers the solution obtaining from each process (including itself) a part of the vector (`MPI_Gather`, line 37).

For space reasons we have omitted a few actual parameters in some calls to MPI operations: the ellipsis in Figure 16.2 denote parameters `0` (the message tag number) and `MPI_COMM_WORLD` (the communicator) in all operations, except in `MPI_Recv` where they denote, in addition, parameter `&status`.

The code in Figure 16.2 is extremely sensitive to variations in the structure of MPI operations. We distinguish five kinds of situations that are further discussed below:

1. Type mismatches in messages,
2. Array length mismatches in messages,
3. Missing send or receive operations,
4. Wrong send-receive order in messages, and
5. Incompatible MPI operations for the different processes.

The first two situations are related to how MPI primitives describe data transmitted in messages: usually in the form of a pointer to a buffer, the length of the buffer, and the type of elements in the buffer. A *type mismatch* in a message exchange occurs when, for example, one replaces `MPI_FLOAT` by `MPI_DOUBLE` in line 19. Then process rank `0` sends a value of type `double`, while process rank `procs-1` expects a `float`. An *array length mismatch* happens, for example, if one replaces `1` with `2` as the second parameter on line 19. Then process rank `0` sends two floating point numbers, while process rank `procs-1` expects exactly one (line 24). It should be emphasised that these mismatches are caught at runtime, if caught at all.

The last three cases all lead to *deadlocks*. In general, MPI programs enter deadlocked situations when a communication operation is not matched by all the processes involved. For example, the *omission of the send operation* on line 19 will leave process rank `procs-1` eternally waiting for a message to come on line 24. For another example, *exchanging the two receive operations* in lines 21 and 22 leads to a deadlock where ranks `0` and `1` will be forever waiting for one another.

Incompatible MPI operations for the different processes come in different flavours. For example, replacing the receive operation by an `MPI_Bcast` on line 24 leads to a situation where process rank `0` tries to send a message, while rank `procs-1` tries to broadcast. For another example, replace the root process of the reduce operation at line 36 from `0` to `rank`. We are left with a situation where each process executes a different reduce operation, each trying to collect the maximum of the values provided by all processes. For a last example, enclose the gather operation on line 37 by a conditional of the form `if(rank == 0)`. In this case process rank `0` will be forever waiting for the remaining processes to provide their parts of the array.

16.3 The Protocol Language

This section introduces the protocol language, following a step-by-step construction of the protocol for our running example.

In the beginning, process rank 0 broadcasts the problem size, a natural number. We write this as

```
broadcast 0 natural
```

That process rank 0 divides X_0 (an array of floating pointing numbers) among all processes is described by a `scatter` operation.

```
scatter 0 float[]
```

Now, each process loops for a given number of iterations, `nIterations`. We write this as follows.

```
foreach iteration: 1..nIterations
```

Variable `nIterations` must be somehow introduced in the protocol. It denotes a value that must be known to all processes. Typically, there are two ways for processes to get to know this value:

- The value is exchanged, resorting to a collective communication operation, in such a way that *all* processes get to know it, or
- The value is known to all processes before computation starts, for example because it is hardwired in the source code or is read from the command line.

In the former case we could add another `broadcast` operation in the first lines of the protocol. In the latter case, the protocol language relies on the `val` constructor, allowing a value to be introduced in the program:

```
val nIterations: positive
```

Either solution would solve the problem. If a `broadcast` is used then processes must engage in a broadcast operation; if `val` is chosen then no value exchange is needed, but the programmer must identify the value in the source code that will replace variable `nIterations`.

We may now continue analysing the loop body (Figure 16.2, lines 17–34). In each iteration, each process sends a message to its left neighbour and another message to its right neighbour. Such an operation is again described as a `foreach` construct that iterates over all processes. The first process is `0`; the last is `size-1`, where `size` is a distinguished variable that represents the number of processes. The inner loop is then written as follows.

```
foreach i: 0..size-1
```

When `i` is the rank of a process, a conditional expression of the form `i=size-1 ? 0 : i+1` denotes the process' right neighbour. Similarly, the left neighbour is `i=0 ? size-1 : i-1`.

To send a message from process rank `r1` to process rank `r2` containing a value of a datatype `D`, we write `message r1 r2 D`. In this way, to send a message containing a floating point number to the left process, followed by a message to the right process, we write.

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

So, now we can assemble the loops.

```
foreach iteration: 1..nIterations
  foreach i: 0..size-1 {
    message i (i=0 ? size-1 : i-1) float
    message i (i=size-1 ? 0 : i+1) float
  }
```

Once the loop is completed, process rank 0 obtains the global error. Towards this end, each process proposes a floating point number representing the local error. Rank 0 then reads the maximum of all these values. We write all this as follows:

```
reduce 0 max float
```

Finally, process rank 0 collects the partial arrays and builds a solution X_n to the problem. This calls for a `gather` operation.

```
gather 0 float[]
```

Before we put all the operations together in a protocol, we need to discuss the nature of the arrays distributed and collected in the `scatter` and `gather` operations. In brief, the `scatter` operation distributes X_0, dividing it in small pieces, while `gather` collects the subarrays to build X_n. So, we instead write:

```
scatter 0 float[n]
...
gather 0 float[n]
```

Variable `n`, describing the length of the global array, must be introduced in the protocol. This is typically achieved by means of a `val` or a `broadcast` operation. In this case `n` stands for the problem size that was broadcast before. So we name the value that rank `0` provides as follows.

```
broadcast 0 n:natural
```

But `n` cannot be an arbitrary non-negative number. It must evenly divide X_0. In this way, each process gets a part of X_0 of equal length, namely `length(X0)/size`, and we do not risk accessing out-of-bound positions when manipulating the subarrays. So we would like to make sure that the length of X_0 equal divides the number of processes. For this we use a *refinement* datatype. Rather that saying that `n` is a natural number we say that it is of datatype `{x: natural | x % size = 0}`. The complete protocol is in Figure 16.3.

As an aside, `natural` can be expressed as `{x: integer | x >= 0}`. Similarly, `positive` abbreviates `{x: integer | x > 0}`, and `float[n]` abbreviates a refinement type of the form `{x: float[] | length(x) = n}`.

Further examples of protocols can be found in a previous work [10] and at the PARTYPES web site [14]. The current version protocol language supports:

```
protocol FiniteDifferences {
  val nIterations: positive
  broadcast 0 n: {x: natural | x % size = 0}
  scatter 0 float[n]
  foreach iteration: 1 .. nIterations
    foreach i: 0 .. size-1 {
      message i (i = 0 ? size-1 : i-1) float
      message i (i = size-1 ? 0 : i+1) float
    }
  reduce 0 max float
  gather 0 float[n]
}
```

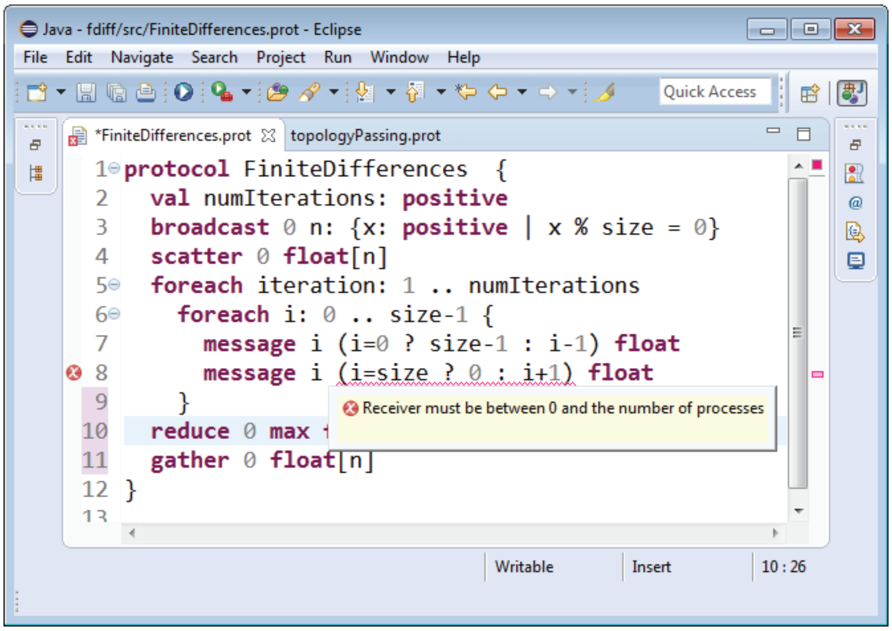
Figure 16.3 Protocol for the finite differences algorithm.

- Different MPI communication primitives such as `message`, `broadcast`, `reduce`, `allreduce`, `scatter`, `gather`, and `allgather`;
- Control flow primitives, including sequential composition (`;`), primitive recursion (`foreach`), conditional (`if-then-else`), and `skip` (that is, the empty block of operations).

Protocols are subject to certain formation rules [10], including:

- Variables must be properly introduced with `val`, `broadcast`, `allreduce`;
- Ranks must lie between `0` and `size-1`;
- The two ranks in a `message` must be different;
- The length of arrays in `scatter` and `gather` must equally divide `size`.

The PROTOCOLCOMPILER checks protocol formation and, in addition, generates a C header file containing the VCC code that describes the protocol. The tool comes as an Eclipse plugin; it may alternatively be used on a web browser from the PARTYPES web page [14]. Figure 16.4 shows a screenshot of Eclipse when the compiler did not manage to prove that the value of expression `i=size ? 0 : i+1` lies between `0` and `size-1`.


Java - fdiff/src/FiniteDifferences.prot - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
*FiniteDifferences.prot
topologyPassing.prot
Writable
Insert
10 : 26

```
protocol FiniteDifferences  {
  val numIterations: positive
  broadcast 0 n: {x: positive | x % size = 0}
  scatter 0 float[n]
  foreach iteration: 1 .. numIterations
    foreach i: 0 .. size-1 {
      message i (i=0 ? size-1 : i-1) float
      message i (i=size ? 0 : i+1) float
    }
  reduce 0 max f
  gather 0 float[n]
}
```

Receiver must be between 0 and the number of processes

Figure 16.4 Protocol compiler running under the Eclipse IDE.

16.4 Overview of the Verification Procedure

This section and the next present the PARTYPES methodology. Figure 16.5 illustrates the workflow of the verification procedure. Two inputs are required:

- The C+MPI source code (example in Figure 16.2);
- The protocol for the program (example in Figure 16.3).

First, the C+MPI source code must be adapted for verification, the reason being that VCC accepts only a subset of the C programming language. Then, special *marks* are inserted in the C source code. One of our tools, the ANNOTATIONGENERATOR (AG in the figure), expands the marks. The output is C source code with VCC annotations, which we denote by C+MPI+VCC. The VCC annotations allow the verification of the C code against the protocol.

A second tool, the PROTOCOLCOMPILER (PC in the figure), checks protocol formation and generates a C header file containing the protocol in VCC format. At this point two C header files need to be included in the C source code: the PARTYPES MPI library, and the protocol in VCC format. The PARTYPES MPI library, `mpi.h`, is a surrogate C header file containing the type theory (as described in a previous work [10]) in VCC format and available at PARTYPES web page [14].

The C code is now ready to be submitted to VCC. The outcome is one of three situations:

- VCC signals success. We know that the C+MPI code, as is, conforms to the protocol, hence is exempt from all the problems discussed in Section 16.2;

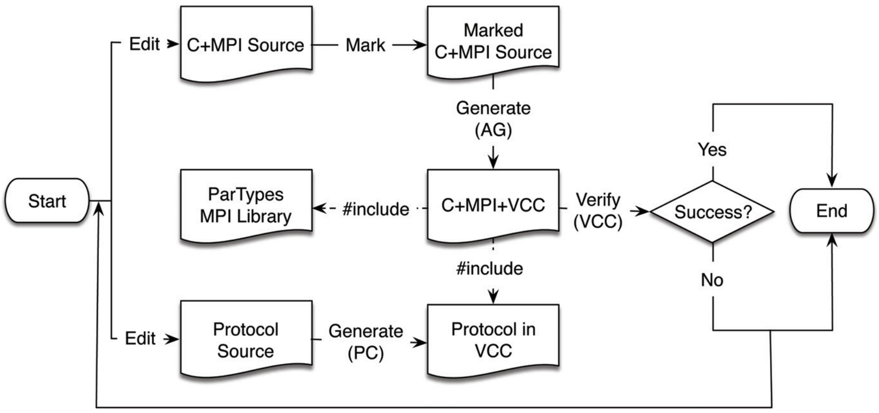


Figure 16.5 Workflow of the verification procedure for C+MPI programs.

- VCC complains presenting the list of failed assertions. In this case, the source of the problem may lie at three different places:
    - the protocol does not capture the communication pattern of the program and needs to be rectified;
    - the C+MPI program is not well annotated, either because it needs additional marks or because some existing marks are misplaced;
    - the C+MPI program itself has a fault that needs to be fixed. In our example, the problem size (stored in variable `n`) must be a multiple of the number of processes (stored in variable `procs`), so that the source code may conform to the protocol. Since the problem size is the value of function `read_problem_size` (line 8, Figure 16.2), we may add an explicit contract to the function:

      ```
      int read_problem_size(int procs)
        _(ensures \result>=0 && \result%procs==0);
      {
        ...
      }
      ```

      In such cases ParTypes users must make use of the VCC specification language.
- VCC times out. This situation typically happens when the underlying SMT solver fails to check some refinement condition. The ParTypes user should revise protocol refinements and possibly rewrite them. For instance, to describe that the process with rank `i` sends a floating point value to its right neighbour in a ring topology, we could have written

  ```
  message i (i+1)%size float
  ```

  It is well-known that non-linear integer arithmetics is undecidable in general and inefficiently supported by SMT solvers. Expressions such as `(i+1)%size` may complicate the verification procedure, possibly leading to timeouts. Instead, we include in our protocol (Figure 16.3) an equivalent proposition that is more amenable for the solver, namely, `i=size-1 ? 0 : i+1`.

The rest of this section describes the source code adaptation required to run VCC. In general, the original C+MPI source code requires routine adjustments in order to be accepted by VCC. Adjustments comprise the deletion or the replacement of code that is not supported by VCC. In particular we:

- delete functions with a variable number of arguments (such as `printf` and `scanf`);
- suppress all floating point arithmetic;
- replace multidimensional by single dimensional arrays and adjust the code accordingly.

VCC is a verifier for concurrent C. Even though C+MPI code is generally single-threaded, VCC tries to verify that the source code is thread-safe in any concurrent environment that respects the contracts on its functions and data structures. This complicates the verification process and demands additional VCC annotations that are not directly related to the verification of the adherence of code to protocols. In particular, the ParTypes user needs to guarantee that memory accesses do not introduce data races. He does so by proving that memory locations are not concurrently written (i.e., `\thread_local` in VCC terms) upon reading, and not concurrently written or read upon writing (`\mutable` or `\writable`).

In our running example, and in order to facilitate the explanation and to concentrate on the adherence to the protocol, we inlined all subsidiary functions in the `main` function, made all arrays local to `main`, and omitted the code concerned with the actual computation of the finite differences. This greatly simplifies the annotation process as we must only deal with local memory, and do not have to cope with other verification demands such as maintaining loop invariants or proving that integer arithmetics does not overflow. Such adjustments must be exerted with great care so as not to alter the interactive behaviour of the original program.

16.5 The Marking Process

This section completes the ParTypes methodology for checking C+MPI code by addressing the marking step.

In general, simple protocols require no marks. Imagine the protocol

```
reduce 0 sum integer
```

describing a simple algorithm where each process computes its part of the solution and process rank `0` collects the solution by adding the parts. Because the protocol uses a simple communication primitive no source code marking is required.

We require no marking for the MPI primitives supported by PARTYPES since their usage is taken care of by the contracts provided in the PARTYPES MPI library (`mpi.h`). The PARTYPES user must aid verification through appropriate marks when more advanced protocol features come into play, such as dependent functions (`val`), primitive recursion (`foreach`), and conditionals (`if-then-else`).

We start with `val`. We have seen in Section 16.3 that this primitive introduces a constant in the protocol:

```
val nIterations: positive
```

Users must provide the actual program value for `nIterations`. Analysing the code in Figure 16.2, one realises that the protocol variable `nIterations` corresponds to the program constant `NUM_ITER`. We then add the mark

```
@apply(MAX_ITER)
```

after the three MPI initialisation primitives (`MPI_Init`, `MPI_Comm_rank`, and `MPI_Comm_size`), that is, after line 6.

Next, we address `foreach`. Again, we seek the assistance of the user in pointing out the portion of the source code that matches each occurrence of this primitive. In the protocol of Figure 16.3, loop

```
foreach iteration: 1 .. nIterations
```

is meant to be matched against the `for` loop in Figure 16.2 starting at line 16. We then introduce the mark

```
@foreach(iter, 1, NUM_ITER)
```

just before the body of the `for` loop, thus associating the protocol loop variable and its bounds with those in the C code.

For the inner loop in the protocol (that is, lines 6–9 in Figure 16.3) we could proceed similarly would the source code be perfectly aligned with the protocol, as in the excerpt below meant to replace lines 18–33 in Figure 16.2:

```
for (i = 0; i < procs; i++) {
  if (rank == i)
    MPI_Send(&local[1],          1, MPI_FLOAT, left, ...);
  else if (rank == left)
    MPI_Recv(&local[0],          1, MPI_FLOAT, i, ...);
  if (rank == i)
    MPI_Send(&local[local_n],    1, MPI_FLOAT, right, ...);
  else if (rank == right)
    MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, i, ...);
}
```

However, efficient implementations do not exhibit loops to implement this kind of `foreach` protocols. The loop in the protocol states that each process (`0`, ..., `size-1`) must send a message to its left and to its right neighbour. This means that each process will be involved in *exactly four* message passing operations: send left, send right, receive from left, receive from right. Therefore the above `for` loop can be *completely unrolled* into a series of conditional instructions, each featuring two message send and two message receive operations, as witnessed by the code in Figure 16.2, lines 18–33.

How do we check `foreach` protocols against conditional instructions in source code? A possible approach would be to let the verifier unroll the protocol loop. This may work when `size` is known to be a small natural number. In general, however, protocols do not fix the number of processes. That is the case with our running example which must run on any number of processes (starting at 2, for processes cannot send messages to themselves). In such cases VCC takes `size` to be a 64 bits non-negative integer. This poses significant difficulties to the unrolling process both in terms of memory and verification time.

In the running example, the apparent mismatch between the protocol and the program is that there are three different behaviours in the program depending on the rank (Figure 16.2, lines 18–33), while the protocol specifies a single behaviour, namely:

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

At first sight, it may seem as if the protocol does not specify the required diversity of behaviours, but in fact it does. To see why, let us unroll the inner `foreach` loop. This is what we get when we omit the type of the message (`float`):

```
message 0 size-1; message 0 1;          // when i = 0
message 1 0; message 1 2;               // when i = 1
...
message size-2 1; message size-2 size-1;// i = size-2
message size-1 size-2; message size-1 0 // i = size-1
```

From the unrolled protocol we conclude that the behaviour of process rank `0` is the following:

1. send a message to its left neighbour (`size-1`);
2. send a message to its right neighbour (`1`);

3. receive a message from its right neighbour; and, finally,
4. receive a message from its left neighbour.

The behaviour is straightforward to obtain: just identify the messages that mention rank 0, and use a send when `0` is the source of the `message` or a receive otherwise. This exactly coincides with the four send/receive operations in the C code for rank `0`, lines 19–22.

For the last rank (that is, `size-1`) the relevant send/receive operations are the following:

1. receive a message from its right neighbour (`0`);
2. receive a message from its left neighbour (`size-2`),
3. send a message to its left neighbour; and, finally,
4. send a message to its right neighbour.

This pattern coincides with the source code, lines 24–27. All other behaviours (when rank is between `1` and `size-2`) are similarly obtained and are left as an exercise for the interested reader. The pattern thus obtained should match the code, lines 29–32. Notice that the order of the messages is important, and that we have identified as many behaviours as there are conditional branches in the source code (lines 18–33).

Based on this analysis, and in order to guide the verification process we seek the help of the user by selecting the relevant `foreach` steps (iterations) in each branch of the program. A *relevant step* for rank `k` corresponds to one `foreach` iteration where either the source or the target of a `message` appearing in the loop body is `k`. A step that does not mention `k` (as source or target) is equivalent to `skip`, the empty protocol, and hence irrelevant for verification purposes. In order to check that all non-relevant steps are `skip`, we must provide the loop bounds (`0` and `procs-1` in this case), in addition to the relevant steps.

For example, when rank is `0` the relevant steps are when `i` is equal to `rank`, `right`, and `left`, in this order. So we insert the mark

```
@foreach_steps(rank, right, left, 0, procs-1)
```

just before the code block in lines 19–22. For rank equal to `size-1` the relevant steps are the `right`, the `left`, and the `rank`, again in this order. The required mark at line 23 is

```
@foreach_steps(right, left, rank, 0, procs-1)
```

and the annotation to include in line 28 is

```
@foreach_steps(left, rank, right, 0, procs-1).
```

Figure 16.6 presents the marked version of the program in full.

```
int main(int argc, char** argv) {
 int rank, procs, n; // process rank; number of processes; problem size
 ...
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &procs);
 @apply(NUM_ITER)
 if (rank == 0) {
  n = read_problem_size(procs);
  read_vector(work, n);
 }
 MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
 int local_n = n / procs;
 MPI_Scatter(work, local_n, MPI_FLOAT, &local[1], local_n, MPI_FLOAT, ...);
 int left  = rank == 0 ? procs - 1 : rank - 1; // left neighbour
 int right = rank == procs - 1 ? 0 : rank + 1; // right neighbour
 for (iter=1; iter<=NUM_ITER; iter++) @foreach(iter, 1, NUM_ITER) {
   ...
   if (rank == 0) @foreach_steps(rank, right, left, 0, procs-1) {
     MPI_Send(&local[1],         1, MPI_FLOAT, left, ...);
     MPI_Send(&local[local_n],   1, MPI_FLOAT, right, ...);
     MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right, ...);
     MPI_Recv(&local[0],         1, MPI_FLOAT, left, ...);
   }else if (rank == procs-1) @foreach_steps(right, left, rank, 0, procs-1)
   {
     MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right, ...);
     MPI_Recv(&local[0],         1, MPI_FLOAT, left, ...);
     MPI_Send(&local[1],         1, MPI_FLOAT, left, ...);
     MPI_Send(&local[local_n],   1, MPI_FLOAT, right, ...);
   } else @foreach_steps(left, rank, right, 0, procs-1) {
     MPI_Recv(&local[0],         1, MPI_FLOAT, left, ...);
     MPI_Send(&local[1],         1, MPI_FLOAT, left, ...);
     MPI_Send(&local[local_n],   1, MPI_FLOAT, right, ...);
     MPI_Recv(&local[local_n+1], 1, MPI_FLOAT, right, ...);
   }
   ...
 }
 MPI_Reduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, 0, ...);
 MPI_Gather(&local[1], local_n, MPI_FLOAT, work, local_n, MPI_FLOAT,0,...);
 ...
 MPI_Finalize();
 return 0;
}
```

Figure 16.6 The code of Figure 16.2 with verification marks inserted.

16.6 Related Work

There are different aims and different methodologies for the verification of MPI programs [6]. The verification of interaction-based properties typically seeks to establish the absence of deadlocks and otherwise ill-formed communications among processes (e.g., compatible arguments at both ends in a point-to-point communication, in close relation to type checking safe communication). Several tools exist with this purpose, either for static or runtime verification, usually employing techniques from the realm of model checking and/or symbolic execution. All these tools are hindered by the inherent scalability and state-explosion problems. Notable examples include CIVL [19], DAMPI [21], ISP [15], MOPPER [2], MUST [7], and TASS [20].

In contrast to these tools, PARTYPES follows a deductive verification approach with the explicit aim of attaining scalable verification. A previous work [10] conducts a comparative evaluation by benchmarking PARTYPES against three state-of-the-art tools: ISP [15], a runtime verifier that employs dynamic partial order reduction to identify and exercise significant process interleavings in an MPI program; MUST [7], also a runtime verifier, but that employs a graph-based deadlock detection approach; and TASS [20], a static analysis tool based on symbolic execution. For the tools and the programs considered, PARTYPES runs in a constant time (the tool is insensitive to the number of processes, problem size, and other parameters), in clear contrast to the running time of all the other tools, which exhibited exponential growth in a significant number of cases.

In addition to PARTYPES, the theory of multi-party session types [9] inspired other works in the realm of message-passing programs and MPI in particular. Scribble [8, 22] is a language to describe global protocols for a finite set of participants in message-passing programs using point-to-point communication. Through a notion of projection, a local protocol can be derived for each participant from a global Scribble protocol. Programs based on the local protocols can be implemented using standard message-passing libraries, as in Multiparty Session C [13]. Pabble [12], an extension of Scribble, is able to express interaction patterns of MPI programs where the number of participants in a protocol is decided at runtime, rather than fixed a priori, and was used to generate safe-by-construction MPI programs [11].

In comparison to these works, PARTYPES is specifically aimed at protocols for MPI programs and the verification of the compliance of arbitrary programs against a given protocol. In conceptual terms, we address collective communication primitives in addition to plain point-to-point communication, and require no explicit notion of protocol projection.

16.7 Conclusion

This chapter presents PARTYPES, a type-based methodology to statically verify message-passing parallel programs. By checking that a program follows a given protocol, one guarantees a series of important safety properties, in particular that the program does not run into deadlocks. In contrast to other state-of-the-art approaches that suffer from scalability issues, our approach is insensitive to parameters such as the number of processes, problem size, or the number of iterations of a program.

The limitations of PARTYPES can be discussed along two dimensions:

- Even though PARTYPES addresses the core features of MPI, it leaves important primitives uncovered. These include non-blocking operations and wildcard receive (the ability to receive from any source), among many others.
- Our methodology is sound (in the sense that it does not yield false positives) but too intentional at times. For instance, it requires protocol loops and source code loops to be perfectly aligned, while the type theory [10] allows more flexibility, loop unrolling in particular.

Acknowledgements This work was supported by FCT (projects Advanced Type Systems for Multicore Programming PTDC/EIA–CCO/12254, HYRAX CMUP-ERI/FIA/0048/2013, and the LaSIGE Research Unit UID/CEC/00408/2013), the NORTE 2020 program (project SMILES, NORTE–01–0145–FEDER–000020), EPSRC (projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, and EP/N027833/1), and EU (UPSCALE FP7 612985, and Betty COST Action IC1201).

References

[1] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[2] V. Forejt, D. Kroening, G. Narayanswamy, and S. Sharma. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *FM*, volume 8442 of *LNCS*, pages 263–278. Springer, 2014.

[3] MPI Forum. *MPI: A Message-Passing Interface Standard—Version 3.0*. High-Performance Computing Center Stuttgart, 2012.

[4] I. Foster. *Designing and building parallel programs*. Addison-Wesley, 1995.

[5] X. Fu, Z. Chen, H. Yu, C. Huang, W. Dong, and J. Wang. Symbolic execution of mpi programs. In *ICSE*, pages 809–810. IEEE Press, 2015.

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

[7] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. MPI runtime error detection with MUST: advances in deadlock detection. In *SC*, pages 30:1–30:11. IEEE/ACM, 2012.

[8] K. Honda, R. Hu, R. Neykova, T. C. Chen, R. Demangeon, P. Denielou, and N. Yoshida. Structuring communication with session types. In *COB*, volume 8665 of *LNCS*, pages 105–127. Springer, 2014.

[9] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

[10] H. A. López, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida. Protocol-based verification of message-passing parallel programs. In *OOPSLA*, pages 280–298. ACM, 2015.

[11] N. Ng, J. G. F. Coutinho, and N. Yoshida. Protocols by default: Safe MPI code generation based on session types. In *CC*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015.

[12] N. Ng and N. Yoshida. Pabble: parameterised scribble. *Service Oriented Computing and Applications*, 9(3–4):269–284, 2015.

[13] N. Ng, N. Yoshida, and K. Honda. Multiparty Session C: Safe parallel programming with message optimisation. In *TOOLS Europe*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.

[14] Partypes homepage. `http://gloss.di.fc.ul.pt/ParTypes`.

[15] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Palmer, R. Thakur, and W. Gropp. Practical model-checking method for verifying correctness of MPI programs. In *PVM/MPI*, volume 4757 of *LNCS*, pages 344–353. Springer, 2007.

[16] M. Schulz and B. R. de Supinski. A flexible and dynamic infrastructure for MPI tool interoperability. In *ICPP*, pages 193–202. IEEE, 2006.

[17] S. F. Siegel and G. Gopalakrishnan. Formal analysis of message passing. In *VMCAI*, volume 6538 of *LNCS*, pages 2–18. Springer, 2011.

[18] S. F. Siegel and L.F. Rossi. Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In *EuroPVM/MPI*, volume 5205 of *LNCS*, pages 274–282. Springer, 2008.

[19] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers. CIVL: The concurrency intermediate verification language. In *SC*, pages 61:1–61:12. IEEE Press, 2015.

[20] S. F. Siegel and T. K. Zirkel. Loop invariant symbolic execution for parallel programs. In *VMCAI*, volume 7148 of *LNCS*, pages 412–427. Springer, 2012.

[21] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *SC*, pages 1–10. IEEE, 2010.

[22] N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In *TGC*, volume 8358 of *LNCS*, pages 22–41. Springer, 2013.