Towards deductive verification of MPI programs against session types

Eduardo R. B. Marques
LaSIGE, Faculty of Sciences
University of Lisbon, Portugal

Francisco Martins
LaSIGE, Faculty of Sciences
University of Lisbon, Portugal

Vasco T. Vasconcelos
LaSIGE, Faculty of Sciences
University of Lisbon, Portugal

Nicholas Ng
Imperial College London, UK

Nuno Martins
LaSIGE, Faculty of Sciences
University of Lisbon, Portugal

The Message Passing Interface (MPI) is the de facto standard message-passing infrastructure for developing parallel applications. Two decades after the first version of the library specification, MPI-based applications are nowadays routinely deployed on super and cluster computers. These applications, written in C or Fortran, exhibit intricate message passing behaviours, making it hard to statically verify important properties such as the absence of deadlocks. Our work builds on session types, a theory for describing protocols that provides for correct-by-construction guarantees in this regard. We annotate MPI primitives and C code with session type contracts, written in the language of a software verifier for C. Annotated code is then checked for correctness with the software verifier. We present preliminary results and discuss the challenges that lie ahead for verifying realistic MPI program compliance against session types.

1 Introduction

MPI is a library specification targeting the development of communication intensive parallel applications [11]. There are a number of libraries available that allow to use MPI primitives from within C or Fortran code. MPI supports a huge collection of communication primitives, including collective (barrier, broadcast, reduction, ...) and point-to-point communications (blocking and non blocking), supports persistence, datatypes (predefined and user defined), communication contexts, different process topologies, one-sided communications, file I/O, among others.

Communication mismatch is a major source of errors in MPI applications, often leading to deadlocks or to erroneous results. However, verification of MPI applications is non-trivial, accounting for an active research area. A recent survey [6] summarises the state of the art of verification methods for MPI. Most of these methods are based on model-checking. ISP [16] is a dynamic verifier which uses a scheduler to explore all possible thread interleavings of an execution. The tool exploits independence between thread actions as a heuristic to avoid state explosion. A fixed test harness is then used to detect common deadlock patterns. While this traditional model checking technique aims to capture most common deadlock patterns in MPI programs, their approach is limited to a finite number of tests, and remains to be an approximate solution for deadlock detection. TASS [14] is a tool that combines symbolic execution [15] and model checking techniques to verify safety properties of MPI programs. The tool takes a C+MPI application and an input $n \geq 1$ which restricts the input space, then constructs an abstract model with n processes and checks its functional equivalence with a sequential implementation by executing the model of the application. Parallel data-flow analysis is a static analysis technique applied in [2]. The work focuses on send-receive matching in MPI source code, which helps identify message leaks and communication mismatch, by constructing a parallel control-flow graph by simple symbolic analysis

N. Yoshida, W. Vanderbauwhede (Eds.): Programming Language Approaches to Concurrency- and Communication-cEntric Software 2013 (PLACES'13)
EPTCS 137, 2013, pp. 103–113, doi:10.4204/EPTCS.137.9

© Marques, Martins, Vasconcelos, Ng, and Martins
This work is licensed under the
Creative Commons Attribution License.

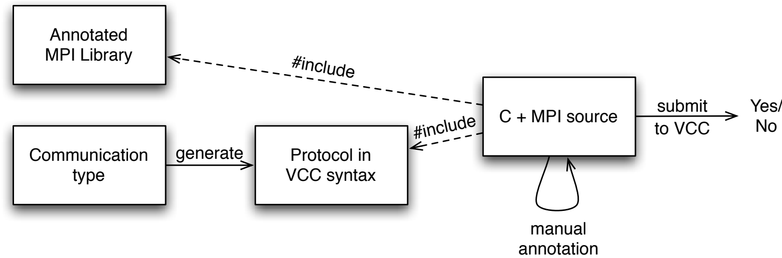

Figure 1: Outline of the approach

on the control-flow graph of the MPI program. In [1] the authors discuss extending the technique by combining static and dynamic analysis to improve precision of the data-flow analysis.

Model-checking based MPI verification methods rely on external testing, or equivalence checking of the model against a correct sample implementation. Parallel data-flow analysis is somewhat ad-hoc. In contrast we seek a simple constructive verification method for an overall understanding of the communication patterns of programs. Our method aims at ensuring that programs follow a predefined protocol; in the process, we ensure that programs are exempt from communication mismatches, deadlocked situations in particular.

Our approach is depicted in Figure 1. We start by writing the protocol in a language tailored for describing MPI communication patterns. Afterwards, we translate the protocol into a term written in the language of a software verifier tool for the C programming language, VCC [3]. The C+MPI code imports the protocol definition (in VCC form) and a VCC-annotated MPI header with session type contracts for the various MPI primitives. Depending on the specifics of the C code, further manual annotations may be required. In this setting, VCC is invoked to check whether the C code follows the communication type.

The project closest to ours is Scribble [9]. Based on the theory of Multiparty Session Types [10], Scribble describes, from an high level perspective, patterns of message-passing interactions. Protocol design with Scribble starts by identifying the communication participants. The body of the protocol describes the interactions from a global viewpoint, with explicit senders and receivers, thus ensuring that all senders have a matching receiver and vice versa. Global protocols are then projected into each of their participants' counterparts, yielding one local protocol for each participant present in the global protocol. The projection algorithm converts the user-defined global protocol into the local interaction behaviour of each participant automatically, while preserving the overall interaction patterns and the order of the interactions. Developers can then implement programs for the various individual participants, based on the local protocols and using standard message-passing libraries. One such approach, Multiparty Session-C, builds a library of session primitives to be used within the C language [12].

In this work we slightly depart from Multiparty Session Types and Scribble by introducing collective decision primitives, allowing for behaviours where all participants decide to enter or to leave a loop, or choose one of the two branches of a choice point, two patterns impossible to describe with Scribble. We found these primitives to be in line with the common practice of MPI programming. We have also included in the language of communication types MPI specific collective operations, as well as a dependent functional type constructor. Finally, and in contrast to Session-C where programmers use a particular library of communication operations, we directly check standard C+MPI code. Preliminary ideas were put forward in [8].

The outline of the this paper is as follows. The next section introduces our running example, written

```
1  int main(int argc, char** argv) {
2    int np;                          // Number of processes
3    int me;                          // My process rank
4    MPI_Init(&argc, &argv);
5    MPI_Comm_size(MPI_COMM_WORLD, &np);
6    MPI_Comm_rank(MPI_COMM_WORLD, &me);
7    ...
8    int psize = atoi(argv[1]);       // Global problem size
9    if (rank == 0)
10     read_vector(work, psize);
11   ...
12   // Scatter input data
13   MPI_Scatter(work, lsize, MPI_FLOAT, &local[1], lsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
14   ...
15   int left  = (np + me - 1) % np;  // Left neighbour
16   int right = (me + 1) % np;       // Right neighbour
17   // Loop until finite differences converge to a minimum error or max iterations attained
18   while (!converged(globalerr) && iter < MAX_ITER) {
19     ...
20     if (me % 2 == 0) {
21       MPI_Send(&local[1],       1, MPI_FLOAT, left,  0, MPI_COMM_WORLD);
22       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD, &status);
23       MPI_Recv(&local[0],       1, MPI_FLOAT, left,  0, MPI_COMM_WORLD, &status);
24       MPI_Send(&local[lsize],   1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
25     } else {
26       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD, &status);
27       MPI_Send(&local[1],       1, MPI_FLOAT, left,  0, MPI_COMM_WORLD);
28       MPI_Send(&local[lsize],   1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
29       MPI_Recv(&local[0],       1, MPI_FLOAT, left,  0, MPI_COMM_WORLD, &status);
30     }
31     ...
32     MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
33     ...
34   }
35   ...
36   if (converged(globalerr)) {
37     // Gather data at rank 0 for solution
38     MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
39     ...
40   } else
41     printf ("Failed to converge after %d iterations!", MAX_ITER);
42   MPI_Finalize();
43   return 0;
44 }
```

Figure 2: Excerpt of an MPI program for the finite differences algorithm (adapted from [5])

in C+MPI, as well as the language of communication types. Section 3 explains the process of verifying C+MPI code against communication types, and Section 4 presents results of running our system on four textbook examples. Section 5 concludes the paper and points a few directions for further work.

2 Communication types for MPI programs

Our running example is that of the one-dimensional finite differences problem, in which we start with a vector X^0 and must compute X^T iteratively, governed by a given recurrence formula [5]. The original C+MPI code is given in Figure 2. Even though the same program is executed by all processes (in line with the Single-Program-Multiple-Data paradigm), each process is endowed with a unique natural number, its *rank*, that can be used to partially specialise its behaviour.

MPI programs start with a call to `MPI_Init` and conclude with a call to `MPI_Finalize`, lines 4 and 42 in Figure 2. After initialising the framework, each process asks for the total number of processes

```
1  Π size: {n:nat|n%3==0}.
2  scatter(0,MPI_FLOAT,size/3).
3  loop(
4    message(2,1,MPI_FLOAT,1).
5    message(0,2,MPI_FLOAT,1).
6    message(1,0,MPI_FLOAT,1).
7    message(1,2,MPI_FLOAT,1).
8    message(2,0,MPI_FLOAT,1).
9    message(0,1,MPI_FLOAT,1).
10   allreduce(MPI_FLOAT,1,MPI_MAX).
11   end).
12 choice(
13   gather(0,MPI_FLOAT,size/3).end,
14   end).
15 end
```

```
Π size: {n:nat|n%3==0}.
scatter(0,MPI_FLOAT,size/3).
loop(
  send(2,MPI_FLOAT,1).
  receive(1,MPI_FLOAT,1).
  receive(2,MPI_FLOAT,1).
  send(1,MPI_FLOAT,1).
  allreduce(MPI_FLOAT,1,MPI_MAX).
  end).
choice(
  gather(0,MPI_FLOAT,size/3).end,
  end).
end
```

```
Π size: {n:nat|n%3==0}.
scatter(0,MPI_FLOAT,size/3).
loop(
  receive(2,MPI_FLOAT,1).
  send(0,MPI_FLOAT,1).
  receive(2,MPI_FLOAT,1).
  send(0,MPI_FLOAT,1).
  allreduce(MPI_FLOAT,1,MPI_MAX).
  end).
choice(
  gather(0,MPI_FLOAT,size/3).end,
  end).
end
```

Figure 3: Global communication type and two local types (ranks 0 and 1)

(`MPI_Comm_size`, line 5) and the process' own rank (`MPI_Comm_rank`, line 6), storing these values in variables `np` and `me`, respectively. The problem size is read, by all processes, from the arguments of the program and into variable `psize` (line 8). The process with rank `0` alone reads vector X^0 into memory (lines 9–10), and then distributes it among all participants (including itself), each participant receiving a slice of length `psize/np` elements (`MPI_Scatter`, line 13).

Each process then loops until the finite differences converges to a given threshold or a given number of iterations is attained, lines 18–34. The body of the loop specifies point-to-point message exchanges (`MPI_Send` and `MPI_Recv`) between each process and its left and right neighbours, following a ring topology. The purpose of these exchanges is to distribute the border values necessary for the calculations due to each participant. We assume the standard *non-buffered, synchronous semantics* of MPI operations, in that, e.g., an `MPI_Send` operation blocks until the target process issues the corresponding `MPI_Recv` operation. This justifies the different orderings of the message exchanges for the even and the odd ranked participants (lines 20–24 and 26–29): processes would deadlock otherwise. After the two (per participant) message exchanges, and before the end of the loop step, the global error is calculated with a reduction operation and propagated to all participants (`MPI_Allreduce`, line 32). If the procedure converges, the solution is gathered at rank `0` (`MPI_Gather`, line 38).

The overall protocol for the various processes is described as a term in the language of *communication types*. Such a term captures, not only the various message exchanges between processes (point-to-point, broadcast), but also the communication-related loops and choices programs make. We informally describe the language next. A possible communication type for our running example is presented in Figure 3, left column.

The atoms in our types describe the individual MPI communications and the special dependent function type constructor, `Π`. Line 2, **`scatter`**`(0,MPI_FLOAT,size/3)`, describes a data distribution operation, initiated at rank `0` and delivering a float array of length `size/3` to each process. The operation in line 13, **`gather`**`(0,MPI_FLOAT,size/3)`, behaves similarly except that it gathers at rank `0` the various slices of an array. Lines 4–9 introduce point-to-point communications. For example, **`message`**`(2,1,MPI_FLOAT,1)` describes a message exchange, from process rank `2` to process rank `1`, containing a float array of length 1.

Individual MPI communications are composed via *prefixing* and *collective decisions*. Prefixing is defined by the `.` (dot) operator, and the terminated protocol is denoted by **`end`**. A protocol that scatters and then terminates can be written as **`scatter`**`(0,MPI_FLOAT,size/3).`**`end`**. Collective decisions include *loops* and *choices*. A type **`loop`**`(`**`allreduce`**`(MPI_FLOAT,1,MPI_MAX).`**`end`**`).T` denotes a point in the protocol where *all* processes either decide to enter or to leave the loop. In case a process enters the loop, it performs an

`allreduce` operation; in case it decides not to enter the loop, it continues as `T`. The case of a choice is similar: a type `choice(gather(0,MPI_FLOAT,size/3).end,end).T` describes a point in the protocol where *all* processes either decide to gather a float array, or not to engage in any MPI operation. In either case, each process then continues as prescribed by `T`.

Ranks and array lengths are described by integer expressions. The communication type under discussion mentions variable `size`. Such a variable is introduced by a dependent function type constructor `Π`. A type `Πsize:{n:nat|n%3==0}.scatter(0,MPI_FLOAT,size/3).end` denotes a protocol parametric on the size of the problem that scatters a float array in chunks of length `size/3`. Expressions in communication types are formed from literals, variables and arithmetic expressions. Such expressions are of kind integer (`int`), floating point (`float`), or array (`float[n]`). Furthermore, any such kind can be *refined*. Kind `{n:nat|n%3==0}` in line 1 denotes a non-negative integer, multiple of 3. Kind `nat` is itself an abbreviation for `{n:int|n>=0}`. The program in Figure 2 works for any number of participants. In contrast the communication type in Figure 3 describes a protocol for exactly three participants, ranked `0` to `2`. The current version of our communication type language does not allow for iteration, hence we hard-coded the number of processes (`3` in the example) in the type.

We can easily see that there is roughly a one-to-one correspondence between the MPI primitives in our running program and those in the communication type, including loops and conditionals, except for the point-to-point communications, where we see `MPI_Send` and `MPI_Recv` in Figure 2 and `message` alone in Figure 3. There is a difference in perspective: the C code describes a per-process (or *local*) view, whereas the communication code presents a *global* view of the protocol. In order to check C code against communication types, we have to reduce the global view of communication types into a local view. Following [10], we *project* the communication type into each of the ranks present in the type, thus obtaining a series of *local* protocols. Local protocols are very much like global protocols. The only difference is that a `message(0,1,MPI_FLOAT,1)` point-to-point communication is replaced by `send(1, MPI_FLOAT,1)` when projecting on rank `0`, by `receive(0,MPI_FLOAT,1)` when projecting on rank `1`, and omitted altogether for all other ranks.

The projections of the global protocol on ranks `0` and `1` are shown in the central and right columns of Figure 3. We can easily see that the overall structure of the protocol is preserved, except for the point-to-point communications, where the six occurrences of `message` in the communication type are replaced by four occurrences in each local type (notice that each rank `0–2` occurs four times in lines 4–9 of the communication type). It worth noting that the projection operation alone yields different local types for the odd and for the even ranks, as witnessed by the two local types in Figure 3, and that these are aligned with the C code (Figure 2, lines 21–24 and 26–29).

3 Verifying C+MPI code against communication types

Communication types, as described in the previous section, are translated into the language of VCC so that they may be used in the verification process. The language of communication types is described in VCC by a datatype. The left column in Figure 4 describes the datatype `\Type` for communication types, that for convenience relies on datatype `\Comm` for MPI operations.

Rather than translating each local type individually as suggested by the central and right columns in Figure 3, the global type is translated as a VCC function that takes a process rank as parameter. It is this function, `type_func`, that internally projects *all* ranks (as described in the previous section), by making use of the conditional expression (`?:`) of the C programming language. Communication types can be parametric on program values; one such example is exhibited in Figure 3, line 1, where the problem size

```
typedef int MPI_Datatype;
typedef int Rank;
typedef int Length;
_(datatype \Comm {
  case send(Rank,MPI_Datatype,Length);
  case recv(Rank,MPI_Datatype,Length);
  case scatter(Rank,MPI_Datatype,Length);
  case gather(Rank,MPI_Datatype,Length);
  case bcast(Rank,MPI_Datatype,Length);
  ...
})
_(datatype \Type {
  case end();
  case comm(\Comm, \Type);
  case loop(\Type, \Type);
  case choice(\Type, \Type, \Type);
})
```

```
_(ghost _(pure) \Type type_func(int rank, int size)
  _(requires 0 <= rank && rank < 3)
  _(requires 0 <= size && size % 3 == 0)
  _(ensures \result ==
    comm(scatter(0,MPI_FLOAT,size/3),
    loop (
      rank == 0 ?
        comm(send(2,MPI_FLOAT,1), ...) :
      rank == 1 ?
        comm(recv(2,MPI_FLOAT,1), ...) :
      // rank == 2
        comm(send(1,MPI_FLOAT,1), ...),
    choice(
      comm(gather(0, MPI_FLOAT, size/3),end()),
      end(),
    end()))))))
```

Figure 4: The VCC datatype for communication types and the type function for the running example

is introduced in the type. Currently we allow dependent function types to occur only at the top level of types. The parameters for the various dependent functions are all gathered at `type_func`, in addition to the rank parameter.

The right column in Figure 4 contains the VCC function corresponding to the communication type in Figure 3, where we can find two parameters corresponding to the rank and to the problem size. Some explanation on the syntactic details of VCC are in order. VCC annotation blocks are introduced as `_(annotation block)`; keyword **`ghost`** introduces an annotation block necessary for the verification process, but inconsequential for the C program; keyword **`pure`** describes a function without side effects; keywords **`requires`** and **`ensures`** introduce the pre and post conditions of a function, respectively. Finally keyword **`\result`** denotes the value of the function.

As described in Figure 1, function `type_func` is placed in a C header file and included in our running example. Unfortunately, including header files alone is not enough to check C+MPI code against communication types. VCC annotations (some of which can be easily automated) must be added to the code. Figure 5 shows the code for our running example with the necessary annotations. The first annotation is in line 1. The `_ampi_arg_decl` is a convenience C preprocessor macro that introduces a number of ghost parameters used by the verification logic, employed by `main` or any other C function of interest. The declaration of `main` in line 1 expands to

```
int main(int argc, char** argv _(ghost _ampi_glue_t* _gd) _(ghost \Type _type) _(out \Type _type_out)
```

The ghost parameters specify the input and output session type for a function (`_type` and `_type_out`), discussed later in the text, and the `gd` argument characterises verification data for the overall restrictions on the number of processes and the process rank. The `_ampi_glue_t` declaration contains data fields and VCC data structure invariants for the restrictions, as follows:

```
typedef struct {
  int procs;  _(invariant 1 < procs && procs < 32768)
  int rank;   _(invariant 0 <= rank && rank < procs)
} ampi_glue_t;
```

The parameterisation of `gd->procs` and `gd->rank` are reflected in the contracts of primitives `MPI_Comm_size` and `MPI_Comm_rank`:

```
int MPI_Comm_size(MPI_Comm* comm, int* size _(ghost ampi_glue_t* gd))
  _(requires comm == MPI_COMM_WORLD)
  _(ensures *size == gd->procs)
  ...
```

```
1  int main(int argc, char** argv _ampi_arg_decl) {
2    ...
3    MPI_Init(&argc, &argv);
4    MPI_Comm_size(MPI_COMM_WORLD, &np);
5    MPI_Comm_rank(MPI_COMM_WORLD, &me);
6    _(assume np == 3)
7    ...
8    int psize = atoi(argv[1]);     // Global problem size
9    _(ghost type = type_func (me, psize))
10   ...
11   MPI_Scatter(work, lsize, MPI_FLOAT, &local[1], lsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
12   ...
13   int left = (np + me - 1) % np; // Left neighbour
14   int right = (me + 1) % np;     // Right neighbour
15   _(ghost \Type loop_body = loopBody(type);)
16   _(ghost \Type loop_continuation = next(type);)
17   while (!converged(globalerr) && iter < MAX_ITER)
18     _(writes &globalerr)
19     _(writes \array_range(local, (unsigned) lsize + 2))
20   {
21     _(ghost type = loop_body;)
22     ...
23     _(assert type == end())
24   }
25   _(ghost type = loop_continuation;)
26   ...
27   _(ghost \Type choice_true = choiceTrue(type);)
28   _(ghost \Type choice_false = choiceFalse(type);)
29   _(ghost \Type choice_continuation = next(type);)
30   if (converged(globalerr)) {
31     _(ghost type = choice_true;)
32     MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
33     ...
34     _(assert type == end())
35   } else {
36     _(ghost type = choice_false;)
37     printf ("failed to converge after %d iterations!", MAX_ITER);
38     _(assert type == end())
39   }
40   _(ghost type = choice_continuation;)
41   MPI_Finalize();
42   return 0;
43 }
```

Figure 5: Annotated version of Figure 2

```
int MPI_Comm_rank(MPI_Comm* comm, int* rank _(ghost ampi_glue_t* gd))
  _(requires comm == MPI_COMM_WORLD)
  _(ensures *rank == gd->rank)
  ...
```

Recall that these primitives are used to obtain the number of processes and the process rank (lines 4–5 in the example), and note that we currently restrict the MPI communicator to be only the global top-level communicator in MPI, that is, `MPI_COMM_WORLD`.

After line 1, the annotation of the program is resumed with a (refined) restriction for the number of processes (`_(assume np == 3)` at line 6) and a ghost call to the `type_func` function at line 9. If providing the number of processes to the function can be easily automated, only the programmer knows which expression in the program corresponds to the problem size. Furthermore, rather than annotating calls to MPI primitives at each call site, a contract is defined for each primitive. These contracts rely on the `first` and the `next` partial functions, both operating on `\Type`, and defined by the following axioms.

```
\forall \Type t; \forall \Comm c; first(comm(c, t)) == c
\forall \Type t; \forall \Comm c; next(comm(c,t)) == t
\forall \Type t1,t2; next(loop(t1,t2)) == t2
```

```
\forall \Type t1,t2,t3; next(choice(t1,t2,t3)) == t3
```

Using these partial functions, the contract for the `MPI_Send` primitive can be expressed as follows,

```
extern int _MPI_Send (void *buf, int count, MPI_Datatype dtype, int target, int tag,
    MPI_Comm c _(ghost ampi_glue_t* gd) _(ghost \Type _type) _(out \Type _type_out))
  _(requires first(_type) == send(target, dtype, count))
  _(requires dtype == MPI_INT ==> \thread_local_array((int*) buf, count))
  _(requires dtype == MPI_FLOAT ==> \thread_local_array((float*) buf, count))
  _(ensures _type_out == next(_type))
  ...
```

where we use two ghost variables, `_type` and `_type_out`, to represent the types before and after the call to `MPI_Send`, respectively (VCC does not allow in/out ghost parameters). The contract states that the first action of the incoming type must be of the form `send(target,dtype,count)`, where `target` is the rank of the destination process, and `dtype` and `count` define the data to be transmitted in the message. We must also check the data part of MPI primitives, and this is where VCC becomes handy. In this case we check that the type of the buffer array matches the declared MPI type, and that the buffer contains enough space. For example, `\thread_local_array((int*)buf,count)` means that the memory from `&buf[0]` to `&buf[count-1]` is valid and typed as `int`. The post condition expresses the effect of the `MPI_Send` operation on the type: after the send operation, the outgoing type is the incoming type from which the first communication has been removed. The remaining MPI primitives have similar contracts. At the end of the program, that is, at the calls to `MPI_Finalize`, we check that the type has reduced to `end`.

```
extern int _MPI_Finalize(_(ghost \Type _type) _(out \Type _type_out))
  _(requires _type == end())
  _(ensures _type_out == end())
```

For collective operations, loops in particular, we currently follow a very simple and intentional approach: a loop in the type (`loop`) must be matched by a loop in the code (`for` or `while`). We require a further (partial) function to extract the body of a `loop` type, governed by the following axiom.

```
\forall \Type t1,t2; loopBody(loop(t1,t2)) == t1
```

Equipped with such a function the main loop of our running example is annotated as in Figure 5, lines 15–25. The code is self-explanatory: we extract the loop body and the continuation types at loop entry (lines 15–16). Then, enter the loop with type `loop_body` and terminate the loop with type `end`. In order to analyse the rest of the program we use the `loop_continuation` type (line 25). The case of a `choice` is handled similarly in lines 27–40. In addition to annotations related to the session type, others are required by VCC in regard to the use of of memory, for proper inference of side-effects. For instance, in lines 18–19, the write clause annotations indicate that the variable `globalerr` and the array `local` (from position `0` to `lsize+2`) are changed in the loop body.

VCC analyses C code modularly, each function separately. This means that each such function needs a contract that, among other things, describes the communication type at entry and at exit points. Suppose for example that the reading and distribution of the data among all processes (lines 9–13 in Figure 2) is abstracted in a function `read_vector`. The (currently) manually annotated function signature would look as follows:

```
void read_vector(int psize, int me, int np, float local[]
    _(ghost ampi_glue_t* gd) _(ghost \Type _type) _(out \Type _type_out))
  ...
  _(requires psize >= 0)
  _(requires psize % np == 0)
  _(requires first(_type) == scatter(0, MPI_FLOAT, psize/np))
  _(ensures _type_out == next(_type))
  ...
;
```

Program	C code (loc)	Auto annot. (loc)	Manual annot. (loc)	Manual/loc (%)	VCC time (s)
Finite differences [5]	256	69	12	4.7	6.1
Parallel dot product [13]	357	81	14	3.9	3.7
Parallel Jacobi [13]	429	34	18	4.2	11.4
N-body simulation [7]	362	80	16	4.4	7.0

Figure 6: Lines of code, annotations and running times for four textbook programs

The logic is similar to that employed for MPI primitives. Ghost variables are used in the function declaration to represent the input and output session type, `_type` and `_type_out` respectively, and the contract states the actions performed within the function. In this case, we have that the first action of the input type should be `scatter(0, MPI_FLOAT, psize/np)` and that the output type is the continuation of the input type.

4 Results

We have manually annotated C+MPI code taken from standard textbooks. Even though all annotations were introduced by hand, we distinguish those that can be automatically generated in principle (e.g., the loop and choice annotations) from those that necessarily require the programmer's intervention (e.g, initialising the `type_func` function with the size of the problem, line 9 in Figure 5; loop invariants, lines 18–19; and non-MPI function annotations). In the table in Figure 6 we summarise the number of lines of code (loc) in the original program, the number of annotations that can be potentially automated, the number of programmer annotations, the number of manual annotations per 100 lines of original C code, and the average time VCC took to complete the verification on a Windows machine with two Intel 2.66 GHz cores and 2 GB of RAM [1].

Remark the small number of manual annotations required to successfully verify the code, in all cases below 5% when compared to the total number of lines of code. As described in the previous section, VCC analyses each C function separately. Hence, a major source of annotations comes from functions. Since currently we annotate functions by hand, we have analysed the function related annotations on the manual annotations. How many of such annotations are needed heavily depends on the nature of C code involved; however a fair amount of functions are to be expected on a well designed code. In the examples we tested, we added between 2.4 (finite differences) and 6.0 (parallel Jacobi) lines of annotations per function and in average.

In addition to the numbers above we must add approximately 800 lines of annotated C header files to describe the datatypes, axioms, and the contracts for the various MPI primitives. The annotated header files and examples are available from `http://www.di.fc.ul.pt/~edrdo/ampi-0.1.zip`.

5 Conclusion and future work

We have identified a framework for checking C+MPI code against a protocol description language. Terms of this language, called *communication types*, describe the overall communication structure of programs.

[1] VCC seems to make use of only one core in a multi-core platform. We have also performed our tests on a virtualised platform and observed no difference in performance between virtual machines configured with 1 and 2 cores.

Such a type is then translated into a datatype term of VCC, a verification tool for the C programming language, in the form of an `include` header file. This header, together with a VCC-annotated MPI-library header file, are fed into VCC that determines whether the code follows the protocol. We used our framework to check four non-trivial examples (260–430 lines of code), with promising results.

Much remains to be done. Currently, communication types are manually translated into VCC terms. We are working on an Eclipse plugin that checks the good formation of types (are variables declared? is the source and destination of messages valid ranks?) and generates the corresponding VCC term.

Currently, all annotations to C code are performed by hand. We are working on a tool that, given C source code, automatically inserts the required annotations on collective operations (loop and choice). Another major source of handcrafted annotations are C functions: for each function a pre- and a post-condition on the current communication type (in the form of **`requires`** and **`ensures`** predicates) needs to be manually introduced. We believe that, under some conditions (MPI programs are usually not recursive), these contracts can be automatically derived, given the communication type.

The communication language is currently quite limited. Even though it features a dependent function constructor, its usage in practice is quite restrictive (it can only occur at the top level). The next version of our language will allow using the construct anywhere in the type, will feature a for-each construct allowing to describe protocols with a variable number of processes (cf. [4]), as well dependencies between communication primitives so that types may refer to values exchanged in previous communications. Finally, we plan to address further MPI operations, including non-blocking primitives (`MPI_Isend`, `MPI_Irecv`, and `MPI_Wait`).

Currently, there is a too close connection between a **`loop`** type in a communication type and a **`for`** or a **`while`** loop in C code. We would like to relax the connection by using iso- or equi-recursive techniques. There is also the (deep) issue of checking whether all processes effectively follow the same branch in a collective operation. Finally, the theory of communication types need to be further developed.

Acknowledgements Work funded by EPRSC (EP/K011715/1, EP/K034413/1 and EP/G015635/1) and FCT (PTDC/ EIA-CCO/122547/2010) projects, and by LaSIGE (PEst-OE/EEI/ UI0408/2011). We thank Ernie Cohen for technical help on VCC and Nobuko Yoshida for comments and discussions.

References

[1] S. Aananthakrishnan, G. Bronevetsky & G. Gopalakrishnan (2013): *Hybrid approach for data-flow analysis of MPI programs*. In: *ICS'13*, ACM, pp. 455–456, doi:10.1145/2464996.2467286.

[2] G. Bronevetsky (2009): *Communication-Sensitive Static Dataflow for Parallel Message Passing Applications*. In: *CGO'09*, IEEE Computer Society, pp. 1–12, doi:10.1109/CGO.2009.32.

[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] P.-M. Deniélou, N. Yoshida, A. Bejleri & R. Hu (2012): *Parameterised Multiparty Session Types*. *Logical Methods in Computer Science* 8(4), doi:10.2168/LMCS-8(4:6)2012.

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

[6] 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*. *Communications ACM* 54(12), pp. 82–91, doi:10.1145/2043174.2043194.

[7] W. Gropp, E. Lusk & A. Skjellum (1999): *Using MPI: portable parallel programming with the message passing interface*. 1, MIT press.

[8] K. Honda, E.R.B. Marques, F. Martins, N. Ng, V.T. Vasconcelos & N. Yoshida (2012): *Verification of MPI programs using session types*. In: *EuroMPI'12*, *LNCS* 7940, Springer, pp. 291–293, doi:10.1007/978-3-642-33518-1_37.

[9] K. Honda, A. Mukhamedov, G. Brown, T.C. 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.

[10] K. Honda, N. Yoshida & M. Carbone (2008): *Multiparty asynchronous session types*. In: *POPL*, ACM, pp. 273–284, doi:10.1145/1328897.1328472.

[11] MPI Forum (2012): *MPI: A Message-Passing Interface Standard – Version 3.0*. High-Performance Computing Center Stuttgart.

[12] 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.

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

[14] S. F. Siegel & T. K. Zirkel (2011): *Automatic Formal Verification of MPI-Based Parallel Programs*. In: *PPoPP'11*, ACM, pp. 309–310, doi:10.1145/1941553.1941603.

[15] S.F. Siegel, A. Mironova, G.S. Avrunin & L.A. Clarke (2008): *Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs*. *ACM TOSEM* 17(2), pp. 1–34, doi:10.1145/1348250.1348256.

[16] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby & R. Thakur (2009): *Formal verification of practical MPI programs*. In: *PPoPP'09*, ACM, pp. 261–270, doi:10.1145/1504176.1504214.