Protocol-Based Verification of Message-Passing Parallel Programs

Hugo A. López[1] * Eduardo R. B. Marques[2] Francisco Martins[2] Nicholas Ng[3]
César Santos[2] Vasco Thudichum Vasconcelos[2] Nobuko Yoshida[3]

[1] Technical University of Denmark, Denmark [2] LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
[3] Imperial College London, London, UK

Abstract

We present ParTypes, a type-based methodology for the verification of Message Passing Interface (MPI) programs written in the C programming language. The aim is to statically verify programs against protocol specifications, enforcing properties such as fidelity and absence of deadlocks. We develop a protocol language based on a dependent type system for message-passing parallel programs, which includes various communication operators, such as point-to-point messages, broadcast, reduce, array scatter and gather. For the verification of a program against a given protocol, the protocol is first translated into a representation read by VCC, a software verifier for C. We successfully verified several MPI programs in a running time that is independent of the number of processes or other input parameters. This contrasts with alternative techniques, notably model checking and runtime verification, that suffer from the state-explosion problem or that otherwise depend on parameters to the program itself. We experimentally evaluated our approach against state-of-the-art tools for MPI to conclude that our approach offers a scalable solution.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.2.4 [*Software/Program Verification*]; D.3.1 [*Formal Definitions and Theory*]; D.3.2 [*Language Classifications*]: Concurrent, distributed and parallel languages; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

General Terms Languages, Verification

* Work performed while the author was employed at LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal.

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2814270.2814302

Keywords Program verification, Parallel programming, MPI, Session types, Dependent types

1. Introduction

Background Message Passing Interface (MPI) [7] is the *de facto* standard for programming high performance parallel applications targeting hundreds of thousands of processing cores. MPI programs, written in C or Fortran, specify the behavior of the various processes, each working on different data. Programs make calls to MPI primitives whenever they need to exchange data. MPI offers different forms of communication, notably, point-to-point and collective operators.

Developing MPI applications raises several problems: one can easily write code that causes processes to block indefinitely waiting for messages, or that exchange data of unexpected sorts or lengths. Verifying that such programs are exempt from communication errors is far from trivial. The state-of-the-art verification tools for MPI programs use advanced techniques such as runtime verification [13, 28, 32, 40] or symbolic execution and model checking [10, 13, 28, 35, 38].

Runtime verification cannot guarantee the absence of faults. In addition, the task can become quite expensive due to the difficulty in producing meaningful tests, the time to run the whole test suite, and the need to run the test suite in hardware similar to that where the final application will eventually be deployed. On the other hand, model checking 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 [36].

Verification is further complicated by the different communication semantics for the various MPI primitives [35], or by the difficulty in disentangling processes' collective and individual control flow written on a single source file [1]. These also naturally arise in other more recent standards for message-based parallel programs, such as MCAPI [18].

We attack the problem of verifying C+MPI code using a type theory for parallel programs. In our framework a type describes a protocol, that is, the communication behavior of a program. Programs that conform to one such type are guaranteed to follow the protocol and not to run into deadlocks. The type system features a dependent type language including specific constructors for some of the most common communication primitives found in MPI, in addition to sequential composition, primitive recursion, and a form of collective choice. Our aim is to provide a typed verification basis for the safe development of parallel applications. The next paragraph explains our motivation via an example.

Motivation The finite differences algorithm illustrates the typical features present in a parallel application. 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 (usually process rank 0) disseminates the problem size via 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. Towards this end, in each iteration, each process exchanges boundary values with its left and right neighbours. When the pre-defined number of iterations is reached, process rank 0 obtains the global error via a reduce operation and collects the partial arrays in order to build a solution to the problem.

A stripped down version of the C+MPI code is depicted in Figure 1. Such code is extremely sensitive to variations in the use of MPI operations. For example, the omission of any send/receive operation (lines 11–24) leads to a deadlock where at least one process will be forever waiting for a complementary send or receive operation. Similarly, exchanging lines 21 and 22 leads to a deadlock where ranks `0` and `1` will forever wait for one another. Other sorts of deadlocks may occur when different ranks perform different collective operations at the same time (say, rank 0 broadcasts and all other ranks reduce), or when one of the ranks decides to abandon, at an earlier stage, a loop comprising MPI primitives. It is also easy to use mismatching types or array lengths in MPI calls, thus compromising type and communication safety. Finally, it may not be obvious at all why one needs a three-branched conditional (lines 10–25) in order to perform the "simple" operation of sending a message to the left and then to the right, in a ring topology.

Solution We attack the problem from a programming language angle. In particular, we:

- Propose a protocol (type) language suited for describing the most common scenarios in the practice of parallel programming; and
- Statically check that programs conform to a given protocol, effectively guaranteeing the absence of deadlocks for well-typed programs, regardless of the number of process involved.

```
int main(int argc,char** argv) {                                          1
 MPI_Init(&argc,&argv);                                                   2
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);                                     3
 MPI_Comm_size(MPI_COMM_WORLD,&size);                                     4
 MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);                                5
 MPI_Scatter(data,n/size,MPI_FLOAT,&local[1],n/size,                      6
      MPI_FLOAT,0,MPI_COMM_WORLD);
 int left = rank == 0 ? size - 1 : rank - 1;                              7
 int right = rank == size - 1 ? 0 : rank + 1;                             8
 for (iter = 1; i <= NUM_ITER; iter++) {                                  9
  if (rank == 0) {                                                        10
   MPI_Send(&local[1],1,MPI_FLOAT,left,0,MPI_COMM_WORLD                   11
        );
   MPI_Send(&local[n/size],1,MPI_FLOAT,right,0,                           12
        MPI_COMM_WORLD);
   MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,0,                         13
        MPI_COMM_WORLD,&status);
   MPI_Recv(&local[0],1,MPI_FLOAT,left,0,MPI_COMM_WORLD                   14
        ,&status);
  } else if (rank == size - 1) {                                          15
   MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,0,                         16
        MPI_COMM_WORLD,&status);
   MPI_Recv(&local[0],1,MPI_FLOAT,left,0,MPI_COMM_WORLD                   17
        ,&status);
   MPI_Send(&local[1],1,MPI_FLOAT,left,0,MPI_COMM_WORLD                   18
        );
   MPI_Send(&local[n/size],1,MPI_FLOAT,right,0,                           19
        MPI_COMM_WORLD);
  } else {                                                                20
   MPI_Recv(&local[0],1,MPI_FLOAT,left,0,MPI_COMM_WORLD                   21
        ,&status);
   MPI_Send(&local[1],1,MPI_FLOAT,left,0,MPI_COMM_WORLD                   22
        );
   MPI_Send(&local[n/size],1,MPI_FLOAT,right,0,                           23
        MPI_COMM_WORLD);
   MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,0,                         24
        MPI_COMM_WORLD,&status);
  }                                                                       25
 }                                                                        26
 MPI_Reduce(&localErr,&globalErr,1,MPI_FLOAT,MPI_MAX,0,                   27
      MPI_COMM_WORLD);
 MPI_Gather(&local[1],n/size,MPI_FLOAT,data,n/size,                       28
      MPI_FLOAT,0,MPI_COMM_WORLD);
 MPI_Finalize();                                                          29
 return 0;                                                                30
}                                                                         31
```

Figure 1. Excerpt of an MPI program for the finite differences problem (adapted from [8])

We develop our theory along the lines of intuitionistic type theory [22], demonstrating the soundness of our proposal via two main results: agreement of program reduction (cf. subject-reduction) and progress for programs.

Challenges Figure 2 presents a protocol for the finite differences algorithm. Special variable size represents the number of processes. The protocol captures the communication structure of the algorithm. A val constructor introduces the number of iterations; a broadcast operation initiated by rank 0 disseminates the problem dimension; scatter distributes the array. The external loop caters for the various iterations, whereas the inner one provides for the to-left and to-right message passing. The final reduce and gather collect the error and the solution.

There are two points worth noticing about this protocol. The first is that it talks about messages, whereas the source code mentions send and receive operations. The other is that its inner loop does not correspond to a loop in the program, instead it corresponds to a (three-branched) conditional. The

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

Figure 2. Protocol for the finite differences program

protocol talks about the *global* behavior of the program, in a form inspired by session types [14]. The sole presence of a global protocol ensures deadlock freedom of programs that conform to the protocol.

Each individual process contributes to the global protocol. A novel type equivalence relation equates individual protocols against the global description, in such a way that, if each individual process conforms to its protocol, and all protocols are equivalent, then the program is deadlock free. Type equivalence also naturally justifies the three branches in the source code, lines 10–25 in Figure 1.

Method In order to verify C+MPI source code against protocols we use the VCC deductive software verifier [2]. Our method can be summarised as follows.

- Write a protocol for the program (the protocol serves as further documentation for the program);
- Convert it to VCC;
- Introduce the required annotations in the C+MPI source code; and
- Use VCC to check code conformance 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 variables.

Contributions

1. A dependently typed protocol language featuring primitive recursion and a separate static language of indices, along the lines of DML [42] and Omega [34];
2. A type checking system for a core parallel programming language ensuring a progress property for well-typed programs; and
3. A methodology for checking C+MPI code against protocols, using a deductive program verifier.

Outline The next section presents a broad overview of the verification procedure. Sections 3 and 4 introduce the type theory, a core programming language, and the main results of the paper. Section 5 shows how we embodied the theory in VCC. Section 6 quantitatively compares our approach to state-of-the-art tools for the verification of C+MPI code. Section 7 discusses related work, and Section 8 concludes the paper.

2. Overview of the Verification Procedure

In order to check a program against a protocol, we need a program, a protocol, and a program verifier.

- Programs are written in the C programming language and make use of the MPI library interface;
- Protocols are written in a language described below;
- Programs are verified against a protocol using VCC.

In the sequel we detail how to construct protocols and how to prepare source code for VCC verification.

The Protocol Language by Example We follow a step-by-step construction of the protocol for finite differences algorithm discussed in the introduction. The end result, we have seen, is in Figure 2.

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

```
broadcast 0 natural
```

That process rank 0 divides X_0 among all processes is rendered in ParTypes as a **scatter** operation.

```
scatter 0 float[]
```

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

```
foreach i: 1..nIterations
```

`nIterations` is a variable that must be somehow introduced in the protocol. The variable 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 communications 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.

For the former case we could for instance add another **broadcast** operation in the first lines of the protocol. For the latter, the protocol language relies on the **val** constructor, allowing one to introduce a program value in the type:

```
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 inhabit `nIterations`.

We may now continue analyzing the loop body. In each iteration, each process sends a message to its left neighbor

and another message to its right neighbor. 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, an expression of the form `i=size-1 ? 0 : i+1` denotes its right neighbor. Similarly, the left neighbor is `i=0 ? size-1 : i-1`.

To send a message containing a value of a datatype `D`, from process rank `r1` to rank `r2` 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 i: 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. Scatter distributes X_0, dividing it in small pieces; gather collects the subarrays to build X_n. The arrays in scatter/gather protocols always refer to the whole array, not to the subarrays. So, we instead write:

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

Variable `n` must be introduced somehow (by means of a val, broadcast, or allreduce). In this case `n` is exactly 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 is such that **length**`(X0)%`**size** `= 0`. 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}`.

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

The topology underlying the protocol for the finite differences (Figure 2) is that of a ring: a linear array with a wraparound link. If a different mapping of ranks to processes is to be used, a new protocol must be derived. It turns out that the language of protocols is flexible enough to encode topologies in integer arrays. Such a topology may then be made known to all processes, in such a way that processes exchange messages as per the particular topology. This flexibility is particularly useful for applications that dynamically adequate the protocol to, say, the load of messages exchanged. A datatype of the form

```
{t: {x: integer | 0<=x and x<size}[size] |
  forall y: y in 0..length(t)-1 => t[y] != y}
```

encodes a one-dimensional network topology, where `t[x]=y` means `x` is a *direct neighbor* of `y`: each node has one direct neighbor (a number between `0` and **size**`-1`) that is different from itself. Such a type, call it `D`, can be distributed among all processes by, say, rank `0`.

```
broadcast 0 topology:D
```

Thereafter each process can exchange a message with its neighbor, as in:

```
foreach i: 0 .. length(topology)-1
  message i topology[i] float
```

A right-to-left ring topology of length five can be encoded as $[4, 0, 1, 2, 3]$.

How can one encode a topology where not all processes have direct neighbors, such as a star or a line? One possibility is to weaken the above condition on the elements of the array, while strengthening the subsequent message passing loop. We could for example drop the restriction that `t[y]!=y` and encode a right-to-left *line* of length five as $[0, 0, 1, 2, 3]$, a 0-centered *star* as $[0, 0, 0, 0, 0]$, and a full binary 0-rooted *tree* of depth 3 as $[0, 0, 0, 1, 1, 2, 2]$. In all cases, rank 0 has no direct neighbor. And this causes a problem if we try to send a message from `i` to `topology[i]`, as in the above example.

Given that the topology is a data structure known to all processes we can make use of a new primitive called *collective choice*. We start by broadcasting the topology and enter the loop as before. Then, within the loop, a message is exchanged only if the topology array contains a valid entry.

```
broadcast 0 topology:
  {x:integer | 0<=x and x<size}[size]
```

```
foreach i: 0 .. length(topology)-1
  if (i != topology[i])
    message i topology[i] float
  else
    {}
```

Arbitrary topologies can be encoded, e.g., using an adjacency matrix. More examples of protocols can be found in Section 6.

Verifying C+MPI Code against a Protocol In order to verify a program against a protocol, we need:

- The C+MPI source code;
- The protocol in VCC syntax (a C header file);
- The type theory in VCC format (another header file).

The protocol in VCC syntax can be generated by the ParTypes Eclipse plugin or by a web service [27]. Both check, in addition, the good formation of the protocol. The ParTypes VCC library can be obtained from the project's web site [27].

All that remains is preparing the source code, so that it may be successfully verified by VCC. First, VCC limitations force us to adapt the C source code. In particular, VCC does not support a theory for floating point numbers, and functions with a variable number of arguments. Thus, we must filter out lines that contain, e.g., `printf` or `scanf`, and also adapt floating point code that may have impact on the verification process, e.g., control flow predicates involving expressions of type **`float`** or **`double`**. In this process, we must preserve the control structure of the program, including calls to MPI primitives and variable declarations. The resulting program must still compile and exchange the messages the original program was intended to.

Next, our approach requires introducing some annotations, including:

- Those that distinguish C loops to be matched against **`foreach`** protocols. The core language described in Section 4 uses for expressions for code that is supposed to match a foreach protocol and while loops for all other purposes; C does not make this distinction;
- Those that distinguish C conditionals to be matched against collective choice protocols, **`if-else`**. Again, our core language distinguishes ifc expressions to be matched against collective choices from if expressions used for all other purposes, while C does not;
- The C expression that matches a **`val`** type; and
- Annotations that guide VCC in matching a **`foreach`**-type against a non-loop C constructor.

For C functions that make use of MPI operations we have two options:

- Inline the code (MPI programs are usually non recursive), or
- Write a contract to the function in the form of a pre- and a post-condition, stating the entry and exit values of p, the ghost variable that holds the protocol.

Further details on the annotations required by ParTypes are provided in Section 5.

We are finally in a position to run VCC on the source code. If VCC reports no errors, the program complies with the protocol, and we may conclude that it faithfully follows the protocol and is, in particular, free from deadlocks.

3. The Type Theory

This section introduces the notion of type and the novel notion of type equivalence. Type equivalence is shown to be decidable.

Index Terms Types rely on two base sets: that of variables (denoted x, y, z), and that of integer values (k, l, m, n). There are two distinguished variables: size and rank; we use them to denote the total number of processes and the unique number of a given process, respectively. It will always be the case that $1 \leq \mathsf{rank} \leq \mathsf{size}$. Unlike the case of MPI, the ranks in our type theory run from 1. We start by discussing a few notions types rely on.

Index terms, i, describe the values types may depend upon. Our language counts with variables, integer, and arithmetic operations, as well as the usual array operations: creation $[i_1, \ldots, i_n]$, access $i_1[i_2]$, and length $\mathsf{len}(i)$. Index term formation further includes the standard refinement introduction rule and datatype subsumption [11].

Datatypes *Datatypes*, D, describe integer values (int), arrays of an arbitrary datatype ($D\ \mathsf{array}$), and refinements of the form $\{x\colon D \mid p\}$. Refinement datatypes allow one to describe, e.g., integer values smaller than a given index term i, such as $\{y\colon \mathsf{int} \mid y \leq i\}$, or arrays of a given length n, as in $\{a\colon \mathsf{float\ array} \mid \mathsf{len}(a) = n\}$. Datatypes rely on *propositions* over index terms, including relational operations and conjunction.

All formation rules depend on *contexts*, intuitively ordered maps from variables into datatypes. Contexts are also subject to formation rules. Symbol ε denotes the empty context. A notion of *subtyping* is defined for datatypes, $\Gamma \vdash D_1 <: D_2$. The rules are standard and include those for refinement datatypes [11]. They allow us to conclude that $\varepsilon \vdash \{a\colon \mathsf{float\ array} \mid \mathsf{len}(a) = 512\} <: \mathsf{float\ array}$.

Type Formation We are now in a position to discuss types and type formation. The rules for type formation are in Figure 3. Given Γ and T, if one can deduce $\Gamma \vdash T : \mathbf{type}$, then T is a (well-formed) type under a (well-formed) context Γ. We do not provide a grammar for the constructs of our language; such a grammar, if desired, can be recovered from the blue text in the relevant figures.

A type of the form $\mathsf{message}\, i_1\, i_2\, D$ describes a point-to-point communication, from the i_1-ranked process to the i_2-ranked process, of a value of datatype D. Both index terms

$$\frac{\Gamma \vdash 1 \le i_1, i_2 \le \mathsf{size} \wedge i_1 \neq i_2 \ \mathbf{true} \quad \Gamma \vdash D : \mathbf{dtype}}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D : \mathbf{type}}$$

$$\frac{\Gamma \vdash 1 \le i \le \mathsf{size}\ \mathbf{true}}{\Gamma \vdash \mathsf{reduce}\ i : \mathbf{type}}$$

$$\frac{\Gamma \vdash 1 \le i \le \mathsf{size} \quad \Gamma \vdash D <: \{x\colon D'\mathsf{array} \mid \mathsf{len}(x)\%\mathsf{size} = 0\}}{\Gamma \vdash \mathsf{scatter}\ i\ D : \mathbf{type}}$$

$$\frac{\Gamma \vdash 1 \le i \le \mathsf{size} \quad \Gamma \vdash D <: \{x\colon D'\mathsf{array} \mid \mathsf{len}(x)\%\mathsf{size} = 0\}}{\Gamma \vdash \mathsf{gather}\ i\ D : \mathbf{type}}$$

$$\frac{\Gamma \vdash 1 \le i \le \mathsf{size}\ \mathbf{true} \quad \Gamma, x\colon D \vdash T : \mathbf{type}}{\Gamma \vdash \mathsf{broadcast}\ i\ x\colon D.T : \mathbf{type}}$$

$$\frac{\Gamma, x\colon D \vdash T : \mathbf{type}}{\Gamma \vdash \mathsf{val}\ x\colon D.T : \mathbf{type}} \qquad \frac{\Gamma, x\colon \{y\colon \mathsf{int} \mid y \le i\} \vdash T : \mathbf{type}}{\Gamma \vdash \forall x \le i.T : \mathbf{type}}$$

$$\frac{\Gamma \vdash p : \mathbf{prop} \quad \Gamma \vdash T_1 : \mathbf{type} \quad \Gamma \vdash T_2 : \mathbf{type}}{\Gamma \vdash p\,?\,T_1 : T_2 : \mathbf{type}}$$

$$\frac{\Gamma \vdash T_1 : \mathbf{type} \quad \Gamma \vdash T_2 : \mathbf{type}}{\Gamma \vdash T_1; T_2 : \mathbf{type}} \qquad \frac{\Gamma : \mathbf{context}}{\Gamma \vdash \mathsf{skip} : \mathbf{type}}$$

Figure 3. Type formation rules

must denote valid ranks, that is, they must lie between 1, the first rank, and size, the number of processes. Furthermore, the sending and the receiving processes must be different from each other, since under our semantics, a message from, say, rank 2 to rank 2 leads to a deadlock (see Section 4). A type of the form reduce i denotes a collective operation whereby all processes contribute with values that are used to produce a result (say, the maximum). This value is then transmitted to the i-ranked process, usually known as the *root* process.

A type of the form scatter $i\ D$ describes a collective operation by which the i-ranked process (the *root* process) distributes an array among all processes, including itself. The type formation rule requires i to be a valid rank, and D to be an array. Type gather $i\ D$ denotes the inverse operation, whereby each process proposes an array of identical length, the concatenation of which is delivered to the root process. In both operations, D describes the whole array, that is the array that is distributed in scatter or assembled in gather.

A type of the form broadcast $i\ x\colon D.T$ denotes a collective communication whereby the root process transmits a value of type D to all processes (including itself). The continuation type T may refer to the value transmitted via variable x. Type val $x\colon D.T$ is the dependent product type. In our case, it denotes a collective operation whereby all processes agree on a common value of datatype D, without resorting to communication. Typical applications include program constants and command-line values that programs may depend upon. A type $p\,?\,T_1 : T_2$ denotes a *collective conditional*, whereby all processes jointly decide on proceeding as T_1 or as T_2, again without resorting to communication. The type assignment system in Section 4 makes sure that the value of p is common to all processes.

$$\frac{(\Gamma \vdash T : \mathbf{type})}{\Gamma \vdash T; \mathsf{skip} \equiv T} \qquad \frac{(\Gamma \vdash T : \mathbf{type})}{\Gamma \vdash \mathsf{skip}; T \equiv T}$$

$$\frac{(\Gamma \vdash T_1, T_2, T_3 : \mathbf{type})}{\Gamma \vdash (T_1; T_2); T_3 \equiv T_1; (T_2; T_3)}$$

$$\frac{\Gamma \vdash i < 1\ \mathbf{true} \quad (\Gamma, x\colon \{y\colon \mathsf{int} \mid y \le i\} \vdash T : \mathbf{type})}{\Gamma \vdash \forall x \le i.T \equiv \mathsf{skip}}$$

$$\frac{\Gamma \vdash i \ge 1\ \mathbf{true} \quad (\Gamma, x\colon \{y\colon \mathsf{int} \mid y \le i\} \vdash T : \mathbf{type})}{\Gamma \vdash \forall x \le i.T \equiv (T\{i/x\}; \forall x \le i - 1.T)}$$

$$\frac{\Gamma \vdash i_1, i_2 \neq \mathsf{rank}\ \mathbf{true} \quad (\Gamma \vdash 1 \le i_1, i_2 \le \mathsf{size} \wedge i_1 \neq i_2\ \mathbf{true}) \quad (\Gamma \vdash D : \mathbf{dtype})}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D \equiv \mathsf{skip}}$$

Figure 4. Type equality (excerpt)

Type $T_1; T_2$ describes a computation that first performs the operations as described by T_1 and then those described by T_2. Type skip describes any computation that does not engage in communication. skip-typed processes are not necessarily halted; they may still perform local operations. Finally, type of the form $\forall x \le i.T$ is a concrete instance of primitive recursion. A type $\forall x \le i.T$ uniquely determines an indexed family of types $T\{i/x\}; T\{i-1/x\}; \ldots; T\{1/x\}; \mathsf{skip}$.[1]

In addition to the above type constructors, others could be easily added, either as primitives or as derived constructors, including: software barriers (adapted, e.g., from reduce), allreduce (defined as reduce followed by broadcast), and allgather (gather followed by broadcast).

The reader may have noticed that types such as message or reduce do not introduce value dependencies, whereas others such as broadcast and val do. The continuation of a message type, if exists, is captured by sequential composition, as in message 1 2 float; message 2 1 int. The continuation of a broadcast is built into the type constructor itself; as in broadcast 1 $x\colon$ int. broadcast 1 $y\colon \{z\colon \mathsf{int} \mid z \ge x\}$.skip. The fundamental reason for the difference lies in the target of the values exchanged. In the cases of message and reduce values are transmitted to a unique process, namely the root process. In the case of broadcast all processes receive the *same* value. This value may then be safely substituted in the continuation of the types for *all* processes, thus preserving the good properties of types.

Type formation is decidable; we have written an Eclipse plugin, using an SMT solver, that checks protocol formation [27].

Term Type Equality Type equality plays a central role in dependent type systems. In our case, type equality includes the monoidal rules for semicolon and skip, the (base and step) rules for primitive recursion, and a form of *projection* of message types. The rules in Figure 4 determine what it

[1] Contrary to the concrete syntax for protocols (cf. Figure 2), primitive

$$\frac{\Gamma \vdash i_1, i_2 \neq \mathsf{rank}\ \mathbf{true} \quad (\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D : \mathbf{type})}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D \Rightarrow \mathsf{skip}}$$

$$\frac{\Gamma \vdash i{<}1\ \mathbf{true} \quad (\Gamma \vdash \forall x \leq i.T : \mathbf{type})}{\Gamma \vdash \forall x \leq i.T \Rightarrow \mathsf{skip}}$$

$$\frac{\Gamma \vdash i \geq 1\ \mathbf{true} \quad \Gamma \vdash T\{i/x\} \Rightarrow T' \quad \Gamma \vdash \forall x \leq i{-}1.T \Rightarrow T'' \quad (\Gamma \vdash \forall x \leq i.T : \mathbf{type})}{\Gamma \vdash \forall x \leq i.T \Rightarrow T'; T''}$$

Figure 5. Type conversion (excerpt)

means for two types to be equal under a given context (rules pertaining to congruence and equivalence omitted).

Premises enclosed in parenthesis ensure type formation, playing no other role in the definition of type equality. If we abbreviate a context entry of the form $\mathsf{size}\colon \{x\colon \mathsf{int} \mid x = 3\}$ by $\mathsf{size} = 3$, we can easily show that:

$$\mathsf{size} = 3 \vdash \forall j \leq \mathsf{size.message}\ j\ (j\%\mathsf{size} + 1) \equiv \mathsf{message}\ 3\ 1; \mathsf{message}\ 2\ 3; \mathsf{message}\ 1\ 2$$

The last rule in Figure 4 says that a message type that plays no role for a given process rank is equal to skip. The rule effectively allows one to *project* a given type onto a given rank, a notion introduced in the context of multi-party session types [14], here cleanly captured as type equality. When projecting the above type onto rank 2 we obtain the following type equality,

$$\mathsf{size} = 3, \mathsf{rank} = 2 \vdash \forall j \leq \mathsf{size.message}\ j\ (j\%\mathsf{size} + 1) \equiv \mathsf{message}\ 2\ 3; \mathsf{message}\ 1\ 2$$

in such a way that the $\forall$-type is effectively equivalent to the sequential composition of two messages, when judged under rank 2.

Decidability of Type Equivalence The proof relies on a type conversion relation, and follows the strategy of Coquand [3], albeit in a simplified form. The type conversion relation expands $\forall$-types and projects message types. The result is a type where $\forall$ cannot be further expanded and each message is not equivalent to skip. The relevant rules are in Figure 5; the remaining are the ten congruence rules. The following lemmas establish validity and confluence for the type conversion relation.

Lemma 3.1 (agreement for type conversion)**.** *If* $\Gamma \vdash T \Rightarrow T'$ *then* $\Gamma \vdash T : \mathbf{type}$ *and* $\Gamma \vdash T' : \mathbf{type}$.

Lemma 3.2 (type conversion is deterministic)**.** *If* $\Gamma \vdash T \Rightarrow T'$ *and* $\Gamma \vdash T \Rightarrow T''$*, then* $T' = T''$.

The type equivalence algorithm relies on a further relation, structural congruence $\equiv_c$, defined as the smallest congruence relation that incorporates the commutative monoidal rules for semi-colon and skip.

It should be easy to see that conversion is (strongly) normalizing (primitive recursion is and projection reduces the size of terms). Structural congruence is also decidable (by converting types to skip-terminated lists and checking for equality). The algorithmic type equality, $\equiv_a$, is defined by the following rule,

$$\frac{\Gamma \vdash T_1 \Rightarrow T_1' \quad \Gamma \vdash T_2 \Rightarrow T_2' \quad \Gamma \vdash T_1' \equiv_c T_2'}{\Gamma \vdash T_1 \equiv_a T_2}$$

and works as follows: given a context Γ and two types T_1 and T_2, apply type conversion to both, obtaining T_1' and T_2'. Then check these types for congruence.

We can easily show that algorithmic type equality is sound and complete with respect to type equivalence.

Theorem 3.3 (correctness of algorithmic type equality)**.** $\Gamma \vdash T_1 \equiv_a T_2$ *if and only if* $\Gamma \vdash T_1 \equiv T_2$.

Program Types Program types are vectors of term types. Not all vectors are nevertheless of interest. Program types in particular must not deadlock. Below are a few candidates that, albeit composed of term types, cannot be judged as program types. For the sake of brevity we once more omit the datatype in types.

$$(\mathsf{message}\ 1\ 2), (\mathsf{message}\ 2\ 1)$$
$$(\mathsf{scatter}\ 1), (\mathsf{reduce}\ 1)$$
$$(\mathsf{msg}\ 1\ 3; \mathsf{scatter}\ 1), (\mathsf{msg}\ 1\ 3; \mathsf{reduce}\ 1), (\mathsf{msg}\ 1\ 3; \mathsf{scatter}\ 1)$$
$$(\mathsf{msg}\ 3\ 1; \mathsf{msg}\ 1\ 2), (\mathsf{msg}\ 1\ 2; \mathsf{msg}\ 2\ 3), (\mathsf{msg}\ 2\ 3; \mathsf{msg}\ 3\ 1)$$

The first vector of types is blocked since process rank 1 intends to send a message to rank 2, whereas rank 2 is ready to send a message to rank 1. In the second vector, rank 1 is trying to distribute an array, whereas rank 2 is not ready to receive its part. The third case involves a 1–3 message that leads to a deadlocked situation, namely $(\mathsf{scatter}\ 1), (\mathsf{message}\ 1\ 3; \mathsf{reduce}\ 1), (\mathsf{scatter}\ 1)$; notice that the second type is equivalent to $\mathsf{reduce}\ 1$. The fourth case involves a circular waiting situation: the message between 3 and 1 cannot happen before that of 2 and 3 (see type for rank 3); the 2–3 message cannot happen before the 1–2 (type for rank 2); and finally, the 1–2 message cannot happen before the 3–1 message (type for rank 1). We judge such vector of types as not constituting program types.

We abbreviate context $\mathsf{size} = n, \mathsf{rank} = k$ to $\Gamma^{n,k}$. The rule defining what constitutes a program type, that is determining the meaning of assertions $S : \mathbf{ptype}$, is defined as follows.

$$\frac{\Gamma^{n,k} \vdash T_k \equiv T : \mathbf{type} \quad (1 \leq k \leq n)}{T_1, \ldots, T_n : \mathbf{ptype}}$$

The central intuition of a program type is that it describes a non-deadlocked computation, that is, a computation that

is either halted or that may reduce. With this in mind it is easy to understand that all types must be aligned (if one is reduce i, then all are reduce j and $\Gamma \vdash i = j$ **true**). One exception is the non-collective message types. Yet, even in this case we require type equality by taking advantage of the "projection" rule in type equality, so that, for example, message 1 3, skip, message 1 3 is a program type.

4. A Core Message-Passing Programming Language

This section introduces a core Multiple-Program-Multiple-Data message-passing imperative programming language and its main results: agreement (cf. subject reduction) and progress for programs.

References To deal with imperative features, we introduce the notion of *references*. We rely on an extra base set, that of *reference identifiers*, ranged over by r. A new datatype, D ref, describes references to values of type D. Four new index terms are introduced: references r, and the conventional operations on references: creation mkref i, dereference $!i$, and assignment $i_1 := i_2$.

We designed our programming language in such a way that it directly handles the index terms present in types. The pure index terms introduced in Section 3 are however extended with side effects, such as reference creation and assignment. The meaning of expressions with effects when they occur as index objects to type families is undetermined. For this reason we are careful in requiring index objects appearing in types to remain pure.

Expressions The constructors of our language can intuitively be divided in two parts: conventional expressions usually found in a while-language and communication-specific expressions. A selection of the expression formation rules is in Figure 6; the remaining (skip, val, reduce, collective conditional ifc, conventional conditional if, and let) are standard.

In an expression of the form send $i_1\, i_2$, index term i_1 (of datatype int) denotes the target process and index term i_2 (of datatype D) describes the value to be sent. The type of the send expression is message rank $i_1\, D$, representing a message from process rank to process i_1 containing a value of datatype D. The premises come naturally if one considers the hypothesis necessary for message rank $i_1\, D$ to be considered a type under context Γ, namely, i_1 must denote a valid process number and must be different from the sender's rank. The value to be sent, i_2, must be of datatype D, so that it conforms to the value the message is supposed to exchange.

An expression of the form receive $i_1\, i_2$ denotes the reception of a value (of datatype D) from process i_1. The value is stored in the reference (of datatype D ref) denoted by index term i_2. The type of the expression is message i_1 rank D, expressing the fact that a message is transmitted from process i_1 to the target process rank.

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size} \wedge i_1 \neq \mathsf{rank}\ \mathbf{true} \quad \Gamma \vdash i_2 : D}{\Gamma \vdash \mathsf{send}\ i_1\ i_2 : \mathsf{message}\ \mathsf{rank}\ i_1\ D}$$

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size} \wedge i_1 \neq \mathsf{rank}\ \mathbf{true} \quad \Gamma \vdash i_2 : D\ \mathsf{ref}}{\Gamma \vdash \mathsf{receive}\ i_1\ i_2 : \mathsf{message}\ i_1\ \mathsf{rank}\ D}$$

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size}\ \mathbf{true} \quad \Gamma \vdash i_2 : D \quad \Gamma, x\colon D \vdash e : T \quad \mathsf{rank} \notin \mathrm{fv}(i_1)}{\Gamma \vdash \mathsf{let}\ x\colon D = \mathsf{broadcast}\ i_1\ i_2\ \mathsf{in}\ e : \mathsf{broadcast}\ i_1\ x\colon D.T}$$

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size}\ \mathbf{true} \quad \Gamma \vdash i_2 : \mathsf{float\ array\ ref} \quad \Gamma, \mathsf{rank} = i_1 \vdash i_3 : \mathsf{float}[\mathsf{size} * \mathsf{len}(i_2)]}{\Gamma \vdash \mathsf{scatter}\ i_1\ i_2\ i_3 : \mathsf{scatter}\ i_1\ \mathsf{float}[\mathsf{size} * \mathsf{len}(i_2)]}$$

$$\frac{\Gamma \vdash 1 \le i_1 \le \mathsf{size}\ \mathbf{true} \quad \Gamma \vdash i_2 : \mathsf{float\ array} \quad \Gamma, \mathsf{rank} = i_1 \vdash i_3 : \mathsf{float}\,[\mathsf{size} * \mathsf{len}(i_2)]\ \mathsf{ref}}{\Gamma \vdash \mathsf{gather}\ i_1\ i_2\ i_3 : \mathsf{gather}\ i_1\ \mathsf{float}[\mathsf{size} * \mathsf{len}(i_2)]}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1; e_2 : T_1; T_2} \qquad \frac{\Gamma, x\colon \{y\colon \mathsf{int} \mid y \le i\} \vdash e : T}{\Gamma \vdash \mathsf{for}\ x\colon i..1\ \mathsf{do}\ e : \forall x \le i.T}$$

$$\frac{\Gamma, \{p\} \vdash e : \mathsf{skip}}{\Gamma \vdash \mathsf{while}\ p\ \mathsf{do}\ e : \mathsf{skip}} \qquad \frac{\Gamma \vdash e : T_1 \quad \Gamma \vdash T_1 \equiv T_2}{\Gamma \vdash e : T_2}$$

In all rules, T and D contain no ref datatypes.

Figure 6. Expression formation (excerpt)

In a broadcast expression, index term i_1 denotes the root process and index term i_2 the value to be distributed. The root process cannot refer to the special variable rank, for this has different values at different processes, precluding all processes from agreeing on a common root process. Contrary to the expressions studied so far, where the object of communications is stored in a reference, the value distributed by the root process is collected in a variable x and made available to an explicit continuation expression e. This strategy provides for datatype dependency in broadcast operations, while keeping the expression and the type aligned, as made clear by the type formation rule: variable x (of datatype D) is moved into the context to type the continuation, while retaining its presence in the dependent type for broadcast.

The scatter expression requires three index term arguments: the first is the process that distributes the array (the *root*), the second is the reference that will hold the subarray, and the third is the array to be distributed. The premises reflect these conditions; notice how the types for the arrays embody the relation between their lengths. As discussed in Section 2, notation $D[p]$ abbreviates the refinement datatype of the form $\{a\colon D\ \mathsf{array} \mid \mathsf{len}(a) = p\}$. The rule for the gather expression is similar, except that the order of the last two parameters is reversed: i_2 denotes the subarrays proposed by each process and i_3 the array to be assembled at the root.

In both expressions, ref datatypes denote values written at each process (as in receive), and the last index denotes an expression that is evaluated only at the root process. We

recursion decreases the loop variable.

$$\frac{(\rho, i)\downarrow^{n,k}(\rho', v)\colon D \quad (\Gamma^{n,k}, \rho, x\colon D \vdash e : T) \quad (x \notin \mathrm{fv}(T))}{(\rho, \mathsf{let}\, x\colon D = i\, \mathsf{in}\, e) \to^{n,k} (\rho', e\{v/x\})}$$

$$\frac{\Gamma^{n,k}, \rho \vdash p\ \mathbf{true} \quad (\Gamma^{n,k}, p, \rho \vdash e_1 : T) \quad (\Gamma^{n,k}, \neg p, \rho \vdash e_2 : T)}{(\rho, \mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2) \to^{n,k} (\rho, e_1)}$$

Figure 7. Process reduction (a flavor)

omit similar rules for arrays on integer values, as well as multidimensional arrays.

The expression formation rule for sequential composition $e_1; e_2$ is particular: its type, $T_1; T_2$, is composed of the types T_1 and T_2 for expressions e_1 and e_2. The conventional rule is obtained when e_1 does not engage in communication operations, in which case its type is skip, and we know that $\Gamma \vdash \mathsf{skip}; T_2 \equiv T_2$.

In expression for $x\colon i..1$ do e, variable x takes values $i, i-1, \ldots, 1$ in each different iteration of the loop. The rule for while requires e to be of type skip, not allowing the loop to perform any communication action. An entry of the form $\{p\}$ in a context abbreviates $x\colon \{x\colon \mathsf{int} \mid p\}$, for x a fresh variable [11]. Non skip-types in the body of while loops may lead to deadlocks, since processes are not guaranteed to run the same number of iterations. If communications are required in a loop body, then a for loop must be used. Finally, the last rule in our selection introduces type equality in expression formation.

Stores For the operational semantics we make use of *stores*, maps from reference identifiers into values. Stores can be easily converted into contexts. A store entry of the form $r := v$ is transformed into a context entry $r\colon D\,\mathsf{ref}$, if the initial part of the store is transformed in context Γ and $\Gamma \vdash v : D$. In the sequel we abuse the notation and write ρ where a context is expected. For example $\rho \vdash i : D$ means $\Gamma \vdash i : D$ where ρ is interpreted as a context. *Store update*, notation $\rho[r := v]$, is the store $\rho', r := v, \rho''$ if ρ is of the form $\rho', r := v', \rho''$ and $\rho' \vdash r : D\,\mathsf{ref}$ and $\rho' \vdash v : D$.

Index terms are evaluated against a store; evaluation also resolves the distinguished variables size and rank. Assertion $(\rho_1, i)\downarrow^{n,k}(\rho_2, v)\colon D$ abbreviates "index term i of datatype D evaluates under store ρ_1, $\mathsf{size} = n$, and $\mathsf{rank} = k$, yielding a value v of datatype D and a new store ρ_2". The rules are straightforward and omitted.

Processes A *process* q is a pair (ρ, e) composed of a store ρ and an expression e. The rule below determines the meaning of assertions of the form $\Gamma \vdash q : T$.

$$\frac{\Gamma, \rho \vdash e : T}{\Gamma \vdash (\rho, e) : T}$$

A flavor of the process reduction rules is in Figure 7. The remaining rules (for if-false, while-true, while-false, skip, for-loop, for-end, and sequential composition) are standard. The rules should be self-explanatory. The let expression evaluates index i to value v and proceeds with expression e with v replacing variable x. Since let is a local (process) operation, x cannot be free in T, as discussed before. The premises in parenthesis guarantee the good formation of the stores and the expressions involved. Notice that process reduction does not change the type of the expressions involved.

Lemma 4.1 (agreement for process reduction)**.** *If* $q \to^{n,k} q'$ *then* $\Gamma^{n,k} \vdash q : T$ *and* $\Gamma^{n,k} \vdash q' : T$.

Lemma 4.2 (process reduction is deterministic)**.** *If* $q_1 \to^{n,k} q_2$ *and* $q_1 \to^{n,k} q_3$ *then* $q_2 = q_3$.

The following lemma ensures that processes do not get stuck and will play its part in the main result of the paper.

Lemma 4.3 (progress for processes)**.**

- *If* $\Gamma^{n,k}, \rho \vdash e : \mathsf{skip}$ *then* e *is* skip *or* $(\rho, e) \to^{n,k} q$.
- *If* $\Gamma^{n,k}, \rho \vdash i : D$ *and* $\Gamma^{n,k}, \rho, x\colon D \vdash e : T$ *and* $x \notin \mathrm{fv}(T)$ *then* $(\rho, \mathsf{let}\, x\colon D = i\, \mathsf{in}\, e) \to^{n,k} q$.
- *If* $\Gamma^{n,k}, \rho, p \vdash e_1 : T$ *and* $\Gamma^{n,k}, \rho, \neg p \vdash e_2 : T$ *then* $(\rho, \mathsf{if}\, p\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2) \to^{n,k} q$.
- *If* $\Gamma^{n,k}, \rho, p \vdash e : \mathsf{skip}$ *then* $(\rho, \mathsf{while}\, p\, \mathsf{do}\, e) \to^{n,k} q$.
- *If* $\Gamma^{n,k}, \rho, x\colon \{y\colon \mathsf{int} \mid y \le i\} \vdash e : T$ *then* $(\rho, \mathsf{for}\, x\colon i..1\, \mathsf{do}\, e) \to^{n,k} q$.

Programs A *program* is a vector of processes $q_1, \ldots, q_n$. Not all such vectors are of interest to us. The following rule is meaning determining for assertions of the form $P : S$.

$$\frac{\Gamma^{n,1} \vdash q_1 : T_1 \quad \ldots \quad \Gamma^{n,n} \vdash q_n : T_n \quad T_1, \ldots, T_n : \mathbf{ptype}}{q_1, \ldots, q_n : T_1, \ldots, T_n}$$

We can easily write the finite differences algorithm in our language. In fact, we can write it in an SPMD or in a MPMD style. In the former case, we follow the C+MPI program in Figure 1. In the latter case we prepare three different expressions: for rank 1, for rank **size**, and a third for all the intermediate ranks. The fundamental observation is that all four programs have equivalent types, under the appropriate value for the rank variable.

Program Reduction Figure 8 contains an excerpt of the reduction rules for programs. Program reduction is composed of message passing—send/receive—, five *collective* barrier-like rules—reduce, scatter, gather, broadcast, and val—, one rule for collective decisions, and one rule that provides for *local* process reduction. As in the previous cases, the premises to the rule may be divided in two parts: those governing the reduction process itself, and those guaranteeing the good formation of the programs involved. The latter are enclosed in parenthesis, as before.

Notation $i \downarrow^n v$ abbreviates the evaluation of an int index term under the empty store, $(\varepsilon, i)\downarrow^n(\varepsilon, v)\colon \mathsf{int}$. The proviso, in all rules, that types and datatypes do not contain ref datatypes impedes reference passing (and the associated problem of dangling references at the receiving process). A

$$\frac{\begin{array}{c} i_l \downarrow^n m \quad (\rho_l, i'_l)\downarrow^{n,l}(\rho'_l, v)\colon D \quad i_m \downarrow^n l \quad (\rho_m, i'_m)\downarrow^{n,m}(\rho'_m, r)\colon D\,\mathsf{ref} \quad (l \neq m) \\ (\Gamma^{n,l} \vdash e_l : T) \quad (\Gamma^{n,m} \vdash e_m : T) \quad (\Gamma^{n,k} \vdash q_k : T) \quad {\scriptstyle (k=1..n,\, k\neq l,m)} \end{array}}{\begin{array}{r} q_1, \ldots, q_{l-1}, (\rho_l, \mathsf{send}\ i_l\ i'_l; e_l), q_{l+1}, \ldots, q_{m-1}, (\rho_m, \mathsf{receive}\ i_m\ i'_m; e_m), q_{m+1} \ldots, q_n \rightarrow \\ q_1, \ldots, q_{l-1}, (\rho'_l, e_l), q_{l+1}, \ldots, q_{m-1}, (\rho'_m[r := v], e_m), q_{m+1} \ldots, q_n \end{array}}$$

$$\frac{i_k \downarrow^n l \quad (\rho_l, i'_l)\downarrow^{n,l}(\rho'_l, v)\colon D \quad (\Gamma^n \vdash 1 \leq i_k \leq n\ \mathbf{true}) \quad (\Gamma^{n,k}, \rho_k \vdash i'_k : D) \quad (\Gamma^{n,k}, x\colon D, \rho_k \vdash e_k : T) \quad {\scriptstyle (k=1..n)}}{(\rho_k, \mathsf{let}\, x\colon D = \mathsf{broadcast}\ i_k\ i'_k\ \mathsf{in}\ e_k)_{k=1}^n \rightarrow (\rho_k, e_k\{v/x\})_{k=1}^{l-1}, (\rho'_l, e_l\{v/x\}), (\rho_k, e_k\{v/x\})_{k=l+1}^n}$$

$$\frac{q_l \rightarrow^{n,l} q'_l \quad (\Gamma^{n,k} \vdash q_k : T_k) \quad (T_1, \ldots, T_n : \mathbf{ptype}) \quad {\scriptstyle (k=1..n)}}{q_1, \ldots, q_n \rightarrow q_1, \ldots, q_{l-1}, q'_l, q_{l+1}, \ldots, q_n}$$

In all rules, D and T contain no ref types and $\mathsf{rank} \notin \mathrm{fv}(D, T)$

Figure 8. Program reduction (excerpt)

similar reason forbids the rank variable in types, for this variable has a different value in each different process.

The rule for message-passing, evaluates both index terms in both the send and the receive process. There is a fundamental difference between the first and the second parameter in both cases. The first describes a process rank (target or source), the second the value to be passed, or the reference to hold the result. In general, index terms that denote process ranks cannot refer to the store, for these exact indices show up in the type of the processes (message $m\, i_l\, D$, in the send case). In such cases we use the abbreviated evaluation, as in $i_l \downarrow^n m$. In all other cases, we use evaluation under a generic store, as in $(\rho_l, i'_l)\downarrow^{n,l}(\rho'_l, v)\colon D$. The send/receive processes reduce to skip (the stores evolve accordingly); the others remain unchanged. In the rule for broadcast we follow a slightly different strategy. Since a value is transmitted to all processes, the broadcast expression features an explicit continuation, allowing one to substitute the value directly in the continuation process e_k (and in its type T), as opposed to using references.

Main Results We are finally in a position to state our main results.

Theorem 4.4 (agreement for program reduction)**.** *If $P_1 \rightarrow P_2$ then $P_1 : S_1$ and $P_2 : S_2$.*

Program reduction is Church-Rosser. As usual this does not mean that it is strongly normalising: taking advantage of while-loops, processes may engage in infinite computations.

Theorem 4.5 (program reduction is Church-Rosser)**.** *If $P_1 \rightarrow P_2$ and $P_1 \rightarrow P_3$ then $P_2 \rightarrow P_4$ and $P_3 \rightarrow P_4$.*

In preparation for the progress result, we determine the meaning of assertions of the form P **halted** using the following rule.

$$\frac{(\rho_1 : \mathbf{store}) \quad \ldots \quad (\rho_n : \mathbf{store})}{(\rho_1, \mathsf{skip}), \ldots, (\rho_n, \mathsf{skip})\ \mathbf{halted}}$$

We are finally in a position to establish our progress result.

Theorem 4.6 (progress for programs)**.** *If $P_1 : S$ then P_1* **halted** *or $P_1 \rightarrow P_2$.*

5. Verification of C+MPI Source Code

This section shows how the theory introduced in Sections 3 and 4 is rendered in VCC, so that C+MPI code may be checked with minimal effort.

The ParTypes VCC Library The ParTypes VCC library comprises roughly 800 lines of code and can be obtained from [27]. It comprises:

- The type theory of Section 3 rendered in VCC;
- Contracts for the MPI primitives supported by the theory in Section 4;
- Functions and predicates used in annotations for C control structures (loops and conditionals) that match protocol control structures (foreach and collective choice); and
- miscellaneous functions and predicates.

In what follows, we outline the contents of the library and complete the section by showing how to annotate our running example so that it can be verified by VCC.

Datatypes in VCC Format *Index terms*, i, are C integer expressions. *Propositions*, p, are C boolean expressions. *Datatypes*, D, are rendered as a VCC datatype named `Data`. We can easily show that, for each datatype D, there is an equivalent datatype of the form $\{x\colon B \mid p\}$ where B is a non-refined datatype. Given that VCC does not directly support multi-dimensional arrays, we consider only datatypes of the form $\{x\colon B \mid p\}$ where B is either int, int array or float array.Such refinements are rendered in VCC as predicates of one (int) or two arguments (arrays). The case for float array refinements corresponds to the following VCC function type definition.

```
_(ghost
  typedef \bool FloatArrayPred[float*][\integer])
```

VCC verification logic is introduced in C programs using annotations of the form `_(...)`, and in particular

`_(ghost ...)`. We omit `_(...)` whenever possible to facilitate readability. We also remove the backslash (\) at the end of lines in C macros.

Datatypes D are encoded as follows.

```
datatype Data {
  case intRefin        (IntPred);
  case intArrayRefin   (IntArrayPred);
  case floatArrayRefin (FloatArrayPred);
};
```

Types in VCC Format Types are rendered as a VCC datatype named `Protocol`. There is one VCC datatype constructor for each type constructor in Figure 3. In addition, dependent types need one different constructor for each `Data` constructor, for VCC does not support polymorphic type constructors. Following the type formation rules in Figure 3, the constructor for messages, e.g., is a triple of the form `(\integer,\integer,Data)`. The interesting cases are the dependent constructors: ∀, broadcast, and val, for which we use an higher-order abstract syntax (HOAS) [29]. We start by preparing the abstractions for the three basic datatypes that we support:

```
typedef Protocol IntAbs        [\integer];
typedef Protocol IntArrayAbs   [int*][\integer];
typedef Protocol FloatArrayAbs [float*][\integer];
```

The VCC `Protocol` datatype may then be defined as follows.

```
datatype Protocol {
  case skip ();
  case size (IntPred, IntAbs);
  case seq (Protocol, Protocol);
  case message (\integer, \integer, Data);
  case foreach (\integer, \integer, IntAbs);
  case intBcast (\integer, IntPred, IntAbs);
  case floatArrayBcast (\integer, FloatArrayPred,
    FloatArrayAbs);
  case intVal (IntPred, IntAbs);
  ...
}
```

As an example, type $\forall i \leq 10.\, \mathsf{message}\, i\, (i+1)\, \mathsf{int}$ is rendered in VCC as

```
foreach(1, 10, \lambda \integer i;
  message(i, i + 1,
    intRefin(\lambda \integer v; \true))
```

Similarly, type $\mathsf{broadcast}\, 1\, n : \mathsf{int}.\, \mathsf{skip}$ is rendered as

```
intBcast(1, \lambda \integer v; \true,
  \lambda \integer n; skip())
```

For the purpose of verifying C+MPI code we are interested in sequents of the form $\mathsf{size}\colon D \vdash T : \mathbf{type}$, for which we prepared a specific constructor, `size`, in the `Protocol` datatype.

Type formation (Figure 3) is checked by a dedicated tool [27]. The same tool translates a type T into a VCC `Protocol` datatype such as the one in Figure 9.

```
_(ghost Protocol program_protocol =
  size(\lambda \integer size; size >= 2,
         \lambda \integer size;
  intVal(\lambda \integer nIterations; nIterations > 0,
         \lambda \integer nIterations;
  intBcast(0, \lambda \integer n;
               n >= 0 && n % size == 0,
              \lambda \integer n; seq(
  scatter(0, floatArrayRefin(\lambda float* v;
              \integer len; len == n)), seq(
  foreach(1, nIterations, \lambda \integer iter;
    foreach(0, size-1, \lambda \integer i;seq(
      message(i, i == 0 ? size-1 : i-1,
              floatArrayRefin(\lambda float* v;
                \integer len; len == 1)),
      message(i, i == size-1 ? 0 : i+1,
              floatArrayRefin(\lambda float* v;
                \integer len; len == 1)))
    )
  ), seq(
  reduce(0, MPI_MAX, floatArrayRefin(\lambda float* v;
          \integer len; len == 1)),
  gather(0, floatArrayRefin(\lambda float* v;
            \integer len; len == n)))))))))
```

Figure 9. The protocol for finite differences in VCC syntax

Verification Flow The flow of program verification can be summarized as follows:

1. The contract for `MPI_Init` initializes a ghost variable `p` (of type `Protocol`) with the protocol the program must follow, such as the one in Figure 9;
2. Contracts for MPI communication primitives progressively match `p` against the expected communication primitive;
3. Each control structure in the C program that is related to the protocol is verified, relying on adequate annotations in the body of program;
4. The contract for `MPI_Finalize` asserts that `p` is be equivalent to `skip()`.

In addition to `p` above, we use two other ghost variables, `size` and `rank`, plus the following machinery:

- A total function, `cons`, that extracts, from a protocol `p`, a pair composed of a `head` and a `tail`, in such a way that
  - `p` is equivalent to `seq(head,tail)`,
  - protocol `head` is not `seq`, and
  - `head` is `skip` only when `p` is equivalent to `skip`.

  The function accepts `rank` in addition to a protocol, so that it may decide whether to convert `messages` to `skip`. The pair is rendered as a `Cons` datatype, which we equip with `head` and `tail` deconstructors;
- Partial functions that extract from a protocol (typically the `head`) the various parts of a non-`seq` `Protocol` constructor. As an example, for the `message(f,t,d)` constructor we

prepare three functions—`messageFrom`, `messageTo`, and `messageData`—returning `f`, `t`, and `d`, respectively; and

- An `isSkip(p,rank)` predicate defined as `head(cons(p,rank)) == skip()`.

Contracts for MPI Primitives We illustrate the cases of some representative MPI primitives.

`MPI_Init` declares the ghost variables `p`, `rank`, and `size`, and establishes the basic invariant between `rank` and `size`. It then calls function `cons` to extract from protocol `p` a pair `c` composed of the head and the tail. The `head` of `c` is supposed to be a `size` type, so we use the two deconstructors for `size`, namely `sizePred` and `sizeAbs`, to extract the predicate and the continuation of the type. The predicate is applied to `size` (VCC notation: `pred[size]`) as well as to the continuation of the protocol `abs`, in line with HOAS.

```
#define MPI_Init(argc, argv)
  _(ghost Protocol p = program_protocol)
  _(ghost int rank, size;)
  _(assume 0 <= rank && rank < size)
  _(ghost Cons c = cons(p,rank))
  _(ghost IntPred pred = sizePred(head(c)))
  _(ghost IntAbs abs = sizeAbs(head(c)))
  _(assume pred[size])
  _(ghost p = seq(abs[size],tail(c)))
```

`MPI_Comm_rank` propagates the constraints introduced for variable `rank` at `MPI_Init` to the actual variable in the source code.

```
#define MPI_Comm_rank(comm, my_rank)
  _(assume rank = *(my_rank))
```

`MPI_Comm_size` does the same to ghost variable `size`. Once we are done with the verification of MPI operations, we can check that the protocol is reduced to `skip`:

```
#define MPI_Finalize()
  _(assert isSkip(p, rank))
```

We now exemplify the contract for one of the MPI communication primitives. At `MPI_send` we check the conformance of the C code against the type, namely on what concerns the three components of the type: `from`, `to`, and the `data`. As in the case of `MPI_Init`, the macro uses function `cons` to extract from the protocol its head/tail pair. Since the head is supposed to be a `message` type, we use the three deconstructors for `message`, namely `messageFrom`, `messageTo`, and `messageDate`, to extract the components. These components are then asserted against the expect values in the protocol. In the end we "advance" the protocol to its `tail`. The simplified version when sending an integer array is as follows.

```
#define MPI_Send(buf,count,dtype,to,tag,comm)
  _(ghost Cons c = cons(p, rank))
  _(assert messageFrom(head(c)) == rank)
  _(assert messageTo(head(c)) == to)
  _(ghost Data data = messageData(head(c)))
  _(assert dtype == MPI_INT ==>
      conformsIntArray(data,(int*)buf,count))
  _(ghost p = tail(c))
```

In order to check that the source code conforms to the `data` part of the message, we use the following predicate:

```
_(pure \bool conformsIntArray (Data d, int* buf,
  \integer len))
_(axiom \forall IntArrayPred pred; int* buf;
  \integer len; pred[buf][len] <==>
  conformsIntArray(intArrayRefin(pred),buf,len))
```

Annotating Control Flow Annotations are required for C code that matches primitive recursion (**`foreach`**), collective choice (**`if-else`**), and the **`val`** protocol. Four cases arise:

- A **`val`** is matched against some C expression;
- A **`foreach`** is matched against a C **`for`** loop, e.g., the outer **`foreach`** in Figure 2 when matched against the loop starting at line 9, Figure 1;
- A **`foreach`** is matched against an n-branched C conditional, e.g., the inner **`foreach`** when matched against the C code in lines 10–25, Figure 1;
- A collective choice matched against a C conditional.

We analyze the first three cases; the fourth one is similar to the second. Line 2 in the finite differences protocol (Figure 2), namely

```
val nIterations: positive
```

requires an annotation of the form

```
applyInt(NUM_ITER)
```

to be placed in the source code somewhere after `MPI_Init` and before `MPI_Broadcast`. The `applyInt` macro "injects" the value into the protocol as follows. Once again, the macro uses function `cons` to extract from the protocol its head/tail pair. The head is now supposed to be a `val` type; we use the two deconstructors for `val`, namely `intValPred` and `intValAbs`, to extract the predicate and the continuation of the type. The predicate is asserted at value `NUM_ITER`; value `NUM_ITER` is further applied to the continuation of the protocol (cf. `MPI_Init` above). In this case, the macro expands to the following code.

```
ghost Cons c = cons(p, rank)
ghost IntPred pred = intValPred(head(c))
ghost IntAbs abs = intValAbs(head(c))
assert pred[NUM_ITER]
ghost p = seq(abs[NUM_ITER],tail(c))
```

As an example of a **`foreach`** protocol that must be matched against a **`for`** loop, consider the code in Figure 1, line 9,

```
for(iter=1; iter <= NUM_ITER; iter++)
```

matched against the **`foreach`** protocol in Figure 2, line 5:

```
foreach iter: 1 .. nIterations
```

We again start by using function `cons` to extract from protocol `p` a pair `c` composed of the head and the tail of the protocol. The `head` of `c` is now supposed to be a `foreach` type. We use the deconstructors—`foreachLower`, `foreachUpper`, and `foreachBody`— to extract the three components. We assert the loop boundaries `1` and `NUM_ITER` against the corresponding values of `foreachLower` and `foreachUpper`. For the body of the loop we set `p` to `foreachBody` and, at loop exit, assert that the protocol must have been reduced to `skip`. Finally, for the loop continuation we set `p` to the tail of the original protocol. All this is rendered as follows.

```
_(ghost Cons c = cons(p, rank))
_(assert 1 == foreachLower(head(c)))
_(assert NUM_ITER == foreachUpper(head(c)))
for(iter=1; iter <= NUM_ITER; iter++) {
  _(ghost p = foreachBody(head(c)[i]))
  ...
  _(assert isSkip(p, rank))
}
_(ghost p = tail(c))
```

Finally, as an example of a **`foreach`** protocol that is supposed to be matched against an **`if-else`** conditional, consider the fragment of our running example where each process sends a message to its right and left process, that is, lines 10–25 in Figure 1. The type is in Figure 2, line 6:

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

The general outline is as above: extract from `p` the head and the tail, and, at the end of the conditional, set `p` to the tail of the original protocol. The difference is that in the above case we have a loop body, whereas now, instead, there are several conditional branches. They are all treated alike. In each branch we, intuitively, unfold the **`foreach`** and take into account (in sequence) only the **`foreach`** body terms that are different from `skip`. At the end of each branch we assert that `p` has reduced to `skip`, and that all other **`foreach`** body terms are equivalent to `skip`. For example, in the first branch, the non-`skip` elements in the sequence are when `i` is `rank` (that of the two `MPI_Send`, lines 11 and 12), `right` (that of the first `MPI_Recv`, line 13), or `left` (that of the second `MPI_Recv`, line 14), in this order. We set `p` as the sequence of applying the `foreachBody` body to these three values and, at the end, check that `p` is equivalent to `skip`. Additionally, we assert that the only non-`skip` terms in the **`foreach`** expansion are *exactly* when `i` is `rank`, `right`, and `left`. The annotated code is as follows.

```
_(ghost Cons c = cons(p, rank))
_(ghost fb = foreachBody(head(c)))
 if (rank == 0) {
  _(ghost p = seq(fb[rank],
              seq(fb[right],
                  fb[left])))
  MPI_Send(&local[1],1,MPI_FLOAT, left,...);
  MPI_Send(&local[n/procs],1,MPI_FLOAT,...);
  MPI_Recv(&local[n/procs+1],1,MPI_FLOAT,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,0,...);
  _(assert isSkip(p, rank))
  _(assert \forall \integer i; i>=0 && i<size &&
    (i!=rank && i!=left && i!=right) ==>
      isSkip(fb[i], rank))
} else {...}
_(ghost p = tail(c))
```

6. Evaluation

This section provides an evaluation of the ParTypes approach. We performed a comparative analysis of ParTypes against state-of-the-art MPI verifiers with similar safety guarantees, by measuring the verification times of all tools with varying parameterizations, under a similar environments. We also provide complementary results regarding the ParTypes protocol compiler.

Tools under Test For a comparative analysis we considered the following tools:

TASS A model checker which uses symbolic execution [35];

ISP A dynamic verifier that employs dynamic partial order reduction to select the relevant process schedules [28];

MUST A dynamic verifier that employs a graph-based deadlock detection approach [13].

Even though all these tools are able to check deadlocks and type/communication problems, they address the problem of software verification in very distinct ways. Our tool and TASS statically verify source code, while ISP and MUST monitor program execution. TASS relies on model checking and symbolic execution to prove (or disprove), for instance, program deadlock situations, while our tool uses deductive program verification. ISP and MUST both use PnMPI [32] to intercept MPI calls and build a state that allow them to identify deadlocked situations (among others).

The Benchmark Suite We consider programs taken from textbooks [8, 12, 26] and the FEVS suite [37], usually used in MPI benchmark analysis: 1-D heat diffusion simulation [37], finite differences [8], N-body simulation [12], parallel Jacobi equation solver [26], parallel dot product [26], and pi calculation [12]. All programs are iterative except for the parallel dot example, that is, they have a core computation/communication loop that is repeated for a number of iterations. We changed of the parallel dot program to be iterative as well for benchmarking purposes. The protocols for five of the six programs are given below; the sixth is the finite differences in Figure 2.

The **Diffusion 1-D** program calculates the evolution of the diffusion (heat) equation in one dimension over time.

```
protocol Diffusion1D {
  broadcast 0 n: {x: positive | x % size = 0}
  broadcast 0 nIterations : positive
  broadcast 0 integer
  foreach i: 1 .. size-1
```

```
    message 0 i float[n/size]
  foreach iter: 1 .. nIterations {
    foreach i: 1 .. size-1
      message i i-1 float
    foreach i: 0 .. size-2
      message i i+1 float
  }
}
```

The **N-body simulation** program simulates a dynamic system of particles under the influence of physical forces.

```
protocol NbodySimulation {
  val n: {x: natural | x % size = 0}
  val nIterations: positive
  foreach iter: 1 .. nIterations {
    foreach pipe: 1 .. size-1
      foreach i: 0 .. size-1
        message i (i+1 <= size-1 ? i+1 : 0)
          float[n*4]
    allreduce min float
  }
}
```

The **Parallel Dot** program calculates the dot product of two vectors.

```
protocol ParallelDot {
  val nIterations: positive
  foreach iter: 1 .. nIterations {
    broadcast 0 n: {x: positive | x % size = 0}
    foreach i: 1 .. size-1
      message 0  i float[n/size]
    foreach i: 1 .. size-1
      message 0 i float[n/size]
    allreduce sum float
    foreach i: 1 .. size-1
      message i 0 float
  }
}
```

The **Parallel Jacobi** program solves linear systems of equations using Jacobi's method. [2]

```
protocol ParallelJacobi {
  val n: {y: positive |
    y % size = 0 and y*y % size = 0}
  val nIterations: positive
  scatter 0 float[n*n]
  scatter 0 float[n]
  allgather float[n]
  foreach i: 1 .. nIterations
    allgather float[n]
  gather 0 float[n]
}
```

Finally, the **Pi** program approximates π through numerical integration.

```
protocol Pi {
  val nIterations: positive
  foreach i: 1 .. nIterations {
    broadcast 0 integer
    reduce 0 sum float
  }
}
```

Experimental Setup For each benchmark program, we stripped all computation code that does not affect the program's behavior in terms of MPI calls made or their arguments. In this manner, we can measure the verification effort strictly in terms of MPI interactions. This approach rules out the overhead associated with computation code that comes from actual execution in the case of ISP and MUST, symbolic execution in the case of TASS, and verification of memory accesses by VCC for ParTypes. We fixed buffer sizes used by programs in communication to the minimum value necessary for the programs to work correctly. We then prepared two annotated versions of each program, one for ParTypes and the other for TASS. TASS requires annotations for the input parameters of a program, including the number of processes and the number of loop iterations. No annotations are required for ISP and MUST, since both tools execute the target program directly.

For benchmarking we use a Ubuntu 14.04 Linux machine with 64 GB of RAM and 4 AMD Opteron 6376 processors, each with 16 cores, totalling 64 cores. The MPI runtime (required for ISP and MUST) is MPICH 3.0 and the Java runtime (for TASS) is Oracle's JRE 1.8. Tools ISP 0.3, MUST 1.4, and TASS 1.2 all run natively in this machine. VCC 2.3, supported by Z3 3.2 [4], runs on a hosted Windows 7 virtual machine using a KVM/QEMU 2.0 hypervisor. Note that VCC runs only on Windows platforms; in spite of the virtualisation overhead, we tried to make the VCC setup as close as possible to that used for all other tools.

We run each tool varying the number of processes or the number of iterations for ISP, MUST, and TASS. These variations are not required for ParTypes. The first set of results resulted from varying the number of processes from 2 to 32. We do not consider 64 processes (the number of available cores in the host machine) since ISP and MUST use an extra process for runtime monitoring in addition to one MPI process per core. In conjunction, the number of iterations is fixed to 2^8 for ISP and TASS, and 4^8 for MUST. For the second set of results, we fix the number of processes to 32, and let the number of iterations range from 2^0 to 2^8 for ISP and TASS, and from 4^0 to 4^8 for MUST. The higher values for MUST are necessary in order to obtain a reasonable analysis of the scalability trend.

Under this setup, verification times are taken following the start-up performance methodology of [9]. For each benchmark and configuration of parameters, we took 31 samples of the verification time. We discard the first sample, and com-

[2] SMT solver Z3, used by the ParTypes protocol compiler and VCC, cannot infer `y*y%size=0` from `y%size=0`. Thus we need to state the second condition explicitly.

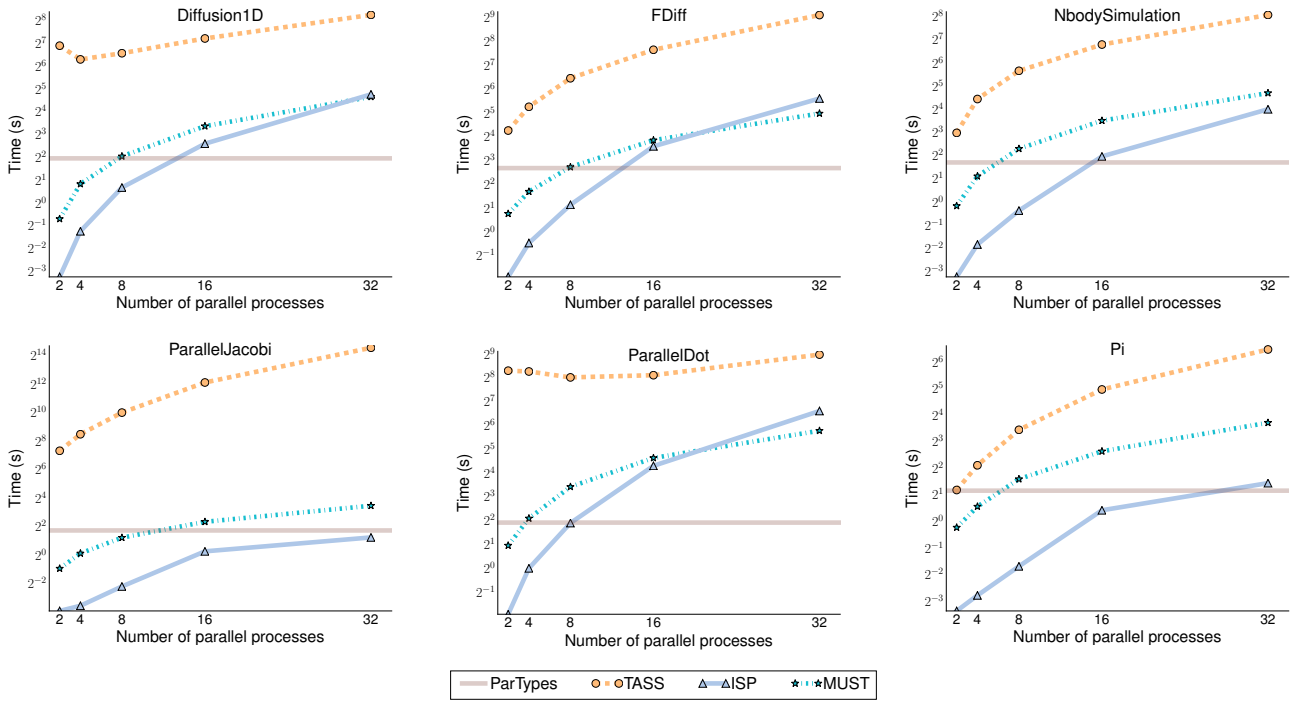

Figure 10. Results for the experiments varying the number of processes

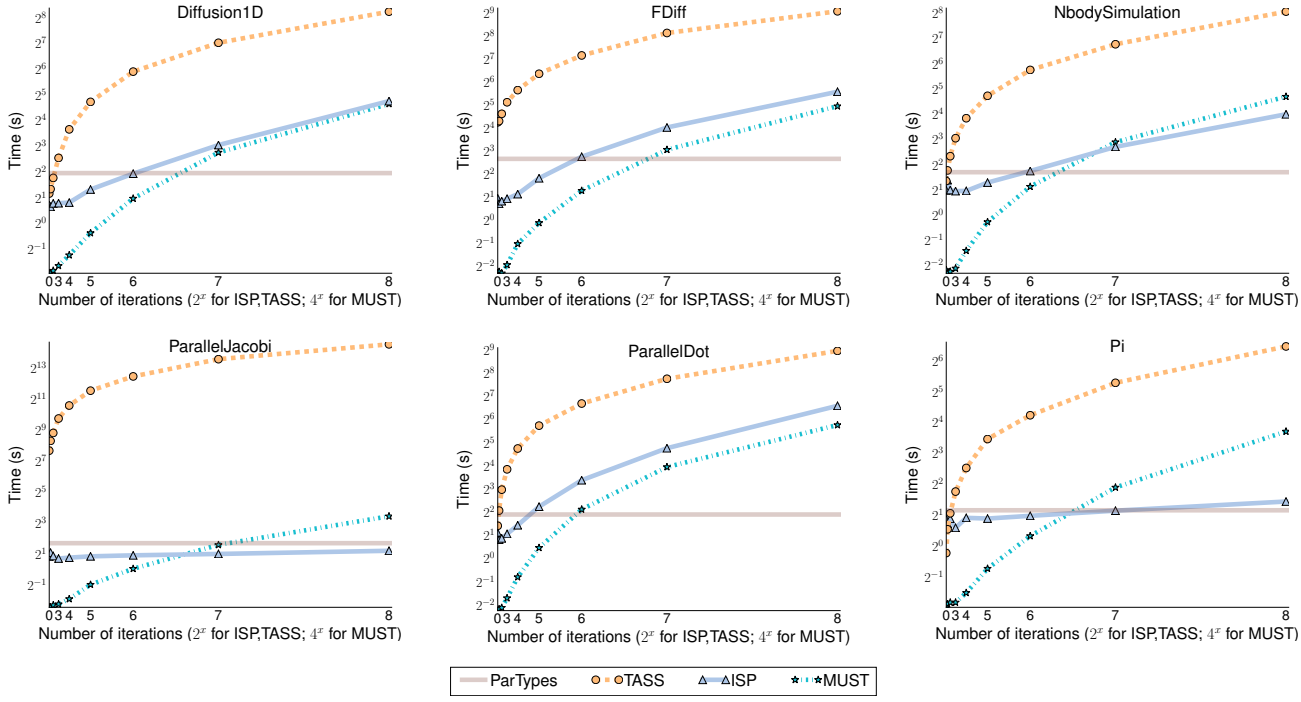

Figure 11. Results for the experiments varying the number of loop iterations

pute the average verification time using the remaining 30 samples.

Results The results are depicted in Figure 10 (for a variable the number of processes) and Figure 11 (for a variable number of iterations). All plots in each figure, one per benchmark program, have a linear scale for the *x* axis (process or iteration count) and a log-2 scale for the *y* axis (verification time in seconds). Each plot shows the verification times of ISP, MUST, and TASS versus ParTypes. The ParTypes verification time is shown constant, since it does not depend on the choice of values for the number of processes or the number of iterations.

As the number of processes and iterations grow larger, the ParTypes verification time can be (sometimes several) orders of magnitude lower than that of the other tools (particularly TASS). This trend is observable in all results, except for ISP in the case of ParallelJacobi and Pi, two programs that only use collective communication primitives. Our approach is immune to the growth of the number of processes or the number of iterations, in clear contrast with the remaining tools under test.

Evaluating the Protocol Compiler Table 1 presents results regarding the protocol compiler that is embedded in the ParTypes Eclipse plugin [27]. We evaluated the command-line version of the tool on the machine described above, by measuring the time it takes to validate a protocol and translate it to VCC form. For each protocol in our benchmark suite the table lists the total execution time, the protocol's validation time, the portion of time spent executing the Z3 SMT solver for proof discharges during validation, and the time elapsed in the final stage of VCC translation. All times are in milliseconds and represent the average of 30 measurements. The results show that the performance is essentially dominated by the time spent in Z3.

Table 1. Protocol compiler – execution times (ms)

	Total	Validation (Z3)	VCC translation
Diffusion1D	332	320 (309)	12
FDiff	314	304 (290)	10
NbodySimulation	185	178 (167)	7
ParallelJacobi	229	219 (206)	10
ParallelDot	313	304 (291)	9
Pi	136	131 (122)	5

7. Related Work

Tools for the Verification of MPI Programs A recent survey covers the state-of-the-art in MPI program verification [10], providing a comprehensive overview of the diverse dimensions of verification and of the methodologies employed. Verification may target the validation of arguments to MPI primitives as well as resource usage [40], ensuring interaction properties such as the absence of deadlocks [28, 35, 40], or asserting functional equivalence to sequential programs [35, 37]. Methodologies range from traditional static and dynamic analysis up to model checking and symbolic execution. In comparison, our novel methodology is based on type checking and deductive program verification.

TASS [37, 38] employs model checking and symbolic execution techniques in order to verify a number of safety properties such as deadlock detection, buffer overflows and memory leaks, plus user-specified assertions about the interactive behavior of processes in a MPI program. TASS also also checks functional equivalence between MPI programs and sequential counterparts [37]. CIVL [39], the recent successor to TASS, employs the same sort of techniques, but uses a unified intermediate verification language that handles not only MPI, but also code written using other popular standards for parallel programming like OpenMP or CUDA.

ISP [28] is a deadlock detection tool that explores all possible process interleavings using a fixed test harness. Other runtime verifiers such as DAMPI [40] or MUST [13], also allow for the detection of deadlocks. MOPPER [6] is a verifier that detects deadlocks by analyzing execution traces of MPI programs. The concept of parallel control-flow graphs [1] provides for the static and dynamic analysis of MPI programs, e.g., as a means to verify sender-receiver matching in MPI source code.

Session Type Theories Among all theoretical works on session types, the closest to ours is probably that of [5], introducing dependent types and a form of primitive recursion into session types. ParTypes provides for various communication primitives (in contrast to message passing only) and incorporates dependent collective choices. On the other hand, we do not allow session delegation. At the term level, we work with a while language, as opposed to a variant of the the π-calculus. Kouzapas et al. introduce a notion of broadcast in the setting of session types [19]. A new operational semantics system provides for the description of 1-to-n and n-to-1 message passing, where n is not fixed a priori, meaning that a non-deterministic number of processes may join the operation, the others being left waiting. Types, however, do not distinguish point-to-point from broadcast operations. We work on a deterministic setting and provide a much richer choice of type operators.

Scribble Based on the theory of multiparty session types [14], Scribble [15, 17, 33, 43] is a language to describe protocols for message-passing programs. 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, as in Multiparty Session C [24].

Pabble [23] 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 interaction patterns of MPI programs where the number of participants in a protocol is decided at runtime. Pabble was applied to generate communication safe-by-construction MPI programs [25], leveraging the close affinity between Pabble protocols and MPI programs. These works show how protocol languages can be used for verifying or constructing MPI programs.

In ParTypes we depart from multiparty session types along two distinct dimensions: 1) our protocol language is specifically built for MPI primitives, and 2) we do not explicitly project a protocol nor generate the MPI code but else check the conformance of code against a global protocol. In contrast to ParTypes, works on parameterised session types [23, 25] cannot deal with:

- Protocols where a given communication (say the source or the target) depends on the contents of previously exchanged data;
- Protocols whose behaviour does not depend directly on message passing, but else on a data-dependent common agreement among all processes (what we call collective operations); and
- Most of the collective operations (broadcast, gather, scatter, reduce) primitives, as well as general and array passing.
- We address the verification of real world code, while [5] works on the π calculus and is not implemented, and [23] does not check existing code.

Dependent Type Systems Following Martin-Löf's works on constructive type theory [22], a number of programming languages have made use of dependent type systems. Rather than taking advantage of the power of full dependent type systems (that brings undecidability to type checking), Xi and Pfenning [42] introduce a restricted form of dependent types, where types may refer to values of a restricted domain, as opposed to values of the term language. The type checking problem is then reduced to constraint satisfiability, for which different tools nowadays are available. Our language follows this approach. Xanadu [41] incorporates these ideas in a imperative C-like language. Omega [34] and Liquid Types [30] are two further examples of pure functional languages that either resort to theorem proving or type inference. All these languages are functional; their type systems cannot abstract program's communication patterns.

Previous Work on ParTypes We initially formulated the problem of verifying C+MPI programs using a type-based approach in [16]. Subsequent work proposes a preliminary evaluation of the approach and experiments [21], where we did not make use of a protocol language, verification did not scale and also required an a priori defined number of processes. We also considered the type-based verification of WhyML parallel programs [31] and the synthesis of correct-by-construction C+MPI programs from protocol specifications [20].

8. Conclusion and Future Work

We presented a type-based methodology to statically verify message-passing parallel programs. By checking that a program follows a given protocol, we guarantee a set of safety properties for the program, in particular that it 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:

- ParTypes addresses the core messaging primitives in MPI, namely: send/receive, broadcast, scatter/gather, reduce, and allreduce/allgather. Notable exceptions are non-blocking operations and wildcard receive (the ability to receive from any source). State-of-the art static verifiers for MPI (see Section 7) roughly deal with this core. On what concerns control primitives, ParTypes include primitive recursion and collective choice, a novel primitive.
- Our VCC methodology is sound but not complete with respect to the core programming language set forth in Section 3.

In view of these limitations, we plan to address further MPI communication primitives, including non-blocking message passing (the "immediate" operations of MPI) and non-determinism in the form of accepting messages from any source. Furthermore, we plan to take advantage of the rich notion of type equivalence to allow for programs with different control flows to be matched against the same protocol.

The idea of a global protocol that governs a parallel program offers further interesting applications, including support for correct-by-construction code generation, test suite generation, and runtime verification.

Acknowledgments This work is supported by FCT through project Advanced Type Systems for Multicore Programming, project Liveness, Statically and project Communication Contracts for Distributed Systems Development (PTDC/EIA-CCO/122547, 117513/2010 and PTDC/EEI-CTP/4503/2014); the LaSIGE Research Unit, (UID/CEC/00408/2013), EPSRC EP/K011715/1, EP/K034413/1, and EP/L00058X/1, the Danish Foundation for Basic Research, project IDEA4CPS (DNRF86-10), EU project FP7-612985 UpScale and COST Action IC1201 BETTY. We would like to thank Stephen Siegel and Zheng Manchun for help on using TASS, and Dimitris Mostrous for his insightful comments.

References

[1] S. Aananthakrishnan, G. Bronevetsky, and G. Gopalakrishnan. Hybrid approach for data-flow analysis of MPI programs. In *ICS*, pages 455–456. ACM, 2013.

[2] 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.

[3] T. Coquand. *Logical Frameworks*, chapter An algorithm for testing conversion in type theory. CUP, 1991.

[4] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[5] P. Deniélou, N. Yoshida, A. Bejleri, and R. Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012.

[6] 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.

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

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

[9] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76. ACM, 2007.

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

[11] A. D. Gordon and C. Fournet. Principles and applications of refinement types. In *International Summer School Logics and Languages for Reliability and Security*, pages 73–104. IOS Press, 2010.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message passing interface*. Scientific and Engineering Computation series. MIT press, 1999.

[13] 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.

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

[15] K. Honda, A. Mukhamedov, G. Brown, T. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.

[16] K. Honda, E. R. B. Marques, F. Martins, N. Ng, V. T. Vasconcelos, and N. Yoshida. Verification of MPI programs using session types. In *Recent Advances in the Message Passing Interface*, volume 7490 of *LNCS*, pages 291–293. Springer, 2012.

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

[18] Y. Huang, E. Mercer, and J. McCarthy. Proving MCAPI executions are correct using SMT. In *ASE*, pages 26–36. IEEE, 2013.

[19] D. Kouzapas, R. Gutkovas, and S. J. Gay. Session types for broadcasting. In *PLACES*, volume 155 of *EPTCS*, pages 25–31, 2014.

[20] F. Lemos. Synthesis of correct-by-construction MPI programs. Master's thesis, Department of Informatics, University of Lisbon, 2014.

[21] E. R. B. Marques, F. Martins, V. T. Vasconcelos, N. Ng, and N. Martins. Towards deductive verification of MPI programs against session types. In *PLACES*, volume 137 of *EPTCS*, pages 103–113, 2013.

[22] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis-Napoli, 1984.

[23] N. Ng and N. Yoshida. Pabble: parameterised Scribble. *Service Oriented Computing and Applications*, pages 1–16, 2014.

[24] 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.

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

[26] P. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[27] ParTypes. Partypes homepage. `http://gloss.di.fc.ul.pt/ParTypes`, July 2015.

[28] 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.

[29] F. Pfenning and C. Elliot. Higher-order abstract syntax. *SIGPLAN Notices*, 23(7):199–208, 1988.

[30] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169. ACM, 2008.

[31] C. Santos, F. Martins, and V. T. Vasconcelos. Deductive verification of parallel programs using Why3. In *ICE*. EPCTS, 2015.

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

[33] Scribble. Scribble homepage. `http://www.scribble.org/`, July 2015.

[34] T. Sheard and N. Linger. Programming in Omega. In *CEFP*, volume 5161 of *LNCS*, pages 158–227. Springer, 2007.

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

[36] S. F. Siegel and L. 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.

[37] S. F. Siegel and T. K. Zirkel. FEVS: A functional equivalence verification suite for high performance scientific computing. *Mathematics in Computer Science*, 5(4):427–435, 2011.

[38] 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.

[39] 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*. IEEE, 2015.

[40] 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.

[41] H. Xi. Imperative programming with dependent types. In *LICS*, pages 375–387. IEEE, 2000.

[42] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM, 1999.

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