Specification and Verification of Protocols for MPI Programs

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

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

Abstract. We present an approach for the validation of Message Passing Interface (MPI) programs. The aim is to type-check programs against session-type based protocol specifications, enforcing properties such as protocol fidelity, absence of race conditions or deadlocks. We design and implement a protocol specification language based on dependent multiparty session types. The language is equipped with a type formation system and an operational semantics, and its correctness is ensured by a progress property for well formed types. For verification, the protocol is translated into the language of VCC, a software verifier for the C programming language. C+MPI programs are annotated with assertions that help the verifier either prove or disprove the conformance of programs against protocols. A large part of the necessary annotations are automatically introduced by an annotator we developed. We successfully validated several publicly available MPI programs with this approach.

1 Introduction

Message Passing Interface (MPI) [5] is the *de facto* standard for programming high performance parallel applications, with implementations that support hundreds of thousands of processing cores. Based on the message passing paradigm, presently there exists official MPI bindings for C and Fortran. MPI programs adhere to the Single Program Multiple Data paradigm, in which a single program specifies the behaviour of the various processes, each working on different data, and uses calls to MPI primitives whenever processes need to exchange data. MPI offers different forms of communication, notably point-to-point, collective, and one-sided communication.

MPI primitives may raise several problems: one can easily write programs that cause processes to block indefinitely waiting for a message, that exhibit race conditions, or that exchange data of unexpected sorts or lengths. Statically verifying that programs are exempt from communication errors is far from trivial. The verification of MPI programs regularly uses advanced techniques such as model checking or symbolic execution [7,19]. These approaches frequently stumble upon the problem of scalability, given that 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 [20]. The verification is further complicated by the different communication semantics for the various MPI primitives [19], or by the difficulty in disentangling processes' collective and individual control flow written on a single source file [2].

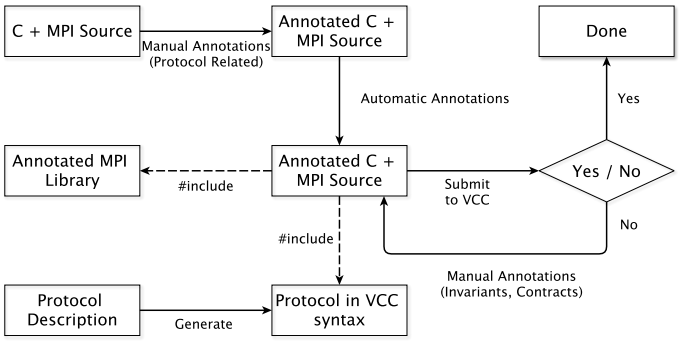

Fig. 1. Our approach to the verification of C+MPI programs

The approach we take for the verification of MPI programs is based on *multiparty session types* [11]. Given an MPI program, we start by specifying the communication protocol that participants must respect, expressed in a language defined for the purpose. Then, the program is checked against the protocol. If it conforms to the protocol, properties such as absence of deadlocks, race conditions, or mismatches in the message exchanges are automatically guaranteed.

Figure 1 illustrates the global view of the protocol specification and verification process. We defined a formal language to describe protocols based on dependent multiparty session types, and appropriate to express of the most common patterns of MPI programs. From a protocol (*Protocol description*) we generate a C header file that represents the protocol in a format compatible with the deductive verification tool VCC [4] (*Protocol in VCC syntax*). Apart from the protocol, verification is also guided by a set of predefined contracts for MPI primitives (*Annotated MPI library*) and by annotations in the body of the C program (*Annotated C+MPI source*). Annotations are either automatically generated or, in a typically smaller number, introduced by the programmer. These aspects form a significant evolution of our previous work [12], where we did not make use of a formal protocol definition language, and where the process depended exclusively on manually produced annotations.

The verification process is based on two tools: the first, in the form of an Eclipse plug-in, checks the good formation of the protocol and generates the C header file with the *Protocol in VCC syntax*; the second, in the form of a Clang plug-in, automatically inserts annotations on C source code, based on the protocol description and on the manually annotated C+MPI source.

In summary, our contributions are:

- the design and implementation of a language for describing MPI-like protocols based on dependent multiparty session types;
- the formal static and dynamic semantics for the language;
- a progress result that asserts that well-formed protocols do not deadlock;
- a tool (Eclipse plug-in) for checking the good formation of protocols;
- a tool (Clang plug-in) for automatically inserting annotations in MPI programs, guided by the protocol description.

The rest of the paper is structured as follows. In the next section we briefly review related work. We then present a program example that illustrates the kind of MPI primitives we address, and that will serve as a running example (Section 3). We then present the protocol specification language and its main result (Section 4). The process of checking C+MPI programs' conformance against a given protocol is then described (Section 5). We show benchmarks on a few real-world MPI programs (Section 6). We conclude the paper with a discussion on future work (Section 7).

2 Related work

Scribble [10,17] is a project closest to ours. Based on the theory of multiparty session types [11], Scribble describes, from an high level perspective, patterns of message-passing interactions. Protocols in Scribble describe the interactions from a global viewpoint, with explicit senders and receivers, thus ensuring that all senders have a matching receiver and vice versa. Global protocols are projected into each of their participants' counterparts, yielding one local protocol for each participant present in the global protocol. Developers can then implement programs for the various individual participants, based on the local protocols and using standard message-passing libraries. One such approach, Multiparty Session C, builds a library of session primitives to be used within the C language [14]. In this work we slightly depart from multiparty session types and Scribble by introducing process ranks as participants' identifiers and collective decision primitives, allowing for behaviours where all participants decide to enter or to leave a loop, or choose one of the two branches of a choice point, two patterns impossible to describe with Scribble. We found these primitives to be in line with the common practice of MPI programming.

The state-of-the-art in MPI program verification has been recently surveyed [7]. Overall, the aim of verification is diverse and includes the validation of arguments to MPI primitives and resource usage [22], ensuring interaction properties such as absence of deadlocks [19,22,23], or asserting functional equivalence to sequential programs [19,21]. The methodologies employed are also diverse, ranging from traditional static and dynamic analysis up to model checking and symbolic execution.

The concept of parallel control-flow graphs is proposed in [2] for static analysis of MPI programs, e.g., as a means to verify sender-receiver matching in MPI source code. An extension to dynamic analysis is presented in [1]. Dynamic execution analysers such as DAMPI [22], amongst others that build on top of the popular PNMPI interposition layer [16], strive for the runtime detection of deadlocks and resource leaks. ISP [23] is a deadlock detection tool that explores all possible process interleavings, using a fixed test harness. TASS [18,19] employs model checking and symbolic execution, but is also able to verify user-specified assertions for the interaction behaviour of the program, so-called collective assertions, and to verify functional equivalence between MPI programs and sequential ones [21].

```
1  int main(int argc, char** argv) {
2    int procs, rank;        // Number of processes, process rank
3    MPI_Init(&argc, &argv);
4    MPI_Comm_size(MPI_COMM_WORLD, &procs);
5    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6    ...
7    if (rank == 0) psize = atoi(argv[1]);
8    MPI_Broadcast(0, &psize, 1, MPI_INT, MPI_COMM_WORLD);
9    lsize = psize / procs;
10   if (rank == 0) read_vector(work, psize);
11   MPI_Scatter(work, lsize, MPI_FLOAT,&local[1],lsize,MPI_FLOAT,0,MPI_COMM_WORLD);
12   int left  = (procs + rank - 1) % procs;  // Left neighbour
13   int right = (rank + 1) % procs;          // Right neighbour
14   int iter = 0;
15   while (!converged(globalerr) && iter < MAX_ITER)) {
16     ...
17     if (rank == 0) {
18       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
19       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
20       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
21       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
22     } else if (rank == procs - 1) {
23       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
24       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
25       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
26       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
27     } else {
28       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
29       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
30       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
31       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status)
32     }
33     MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
34     ...
35   }
36   ...
37   if (converged(globalerr)) // Gather solution at rank 0
38     MPI_Gather(&local[1], lsize, MPI_FLOAT,work,lsize,MPI_FLOAT,0,MPI_COMM_WORLD);
39   ...
40   MPI_Finalize();
41   return 0;
42 }
```

Fig. 2. Excerpt of an MPI program for the finite differences problem (adapted from [6])

3 Running example

Our running example calculates finite differences in one dimension, through the iterative algorithm described in [6]. Given an initial vector X^0, it calculates successive approximations to the solution X^1, X^2, ..., until a certain convergence condition it attained or a pre-defined maximum number of iterations has been reached.

Figure 2 shows the code of a C program, also adapted from [6]. Function main stipulates the behaviour of all processes together; the behaviour of each process may diverge based on its process number, designated by *rank*. The number of processes (procs in the figure) and the rank of each process are obtained respectively through primitives MPI_Comm_size and MPI_Comm_rank (lines 4–5). Rank 0 starts by reading the input vector X^0 (line 7) and then distributes the vector by all participants (line 11, call to MPI_Scatter). Each participant is responsible for the calculation of a local part of the vector.

The program then enters a loop (lines 15–35), specifying point-to-point message exchanges (MPI_Send, MPI_Recv) between each process and its left and right neighbours,

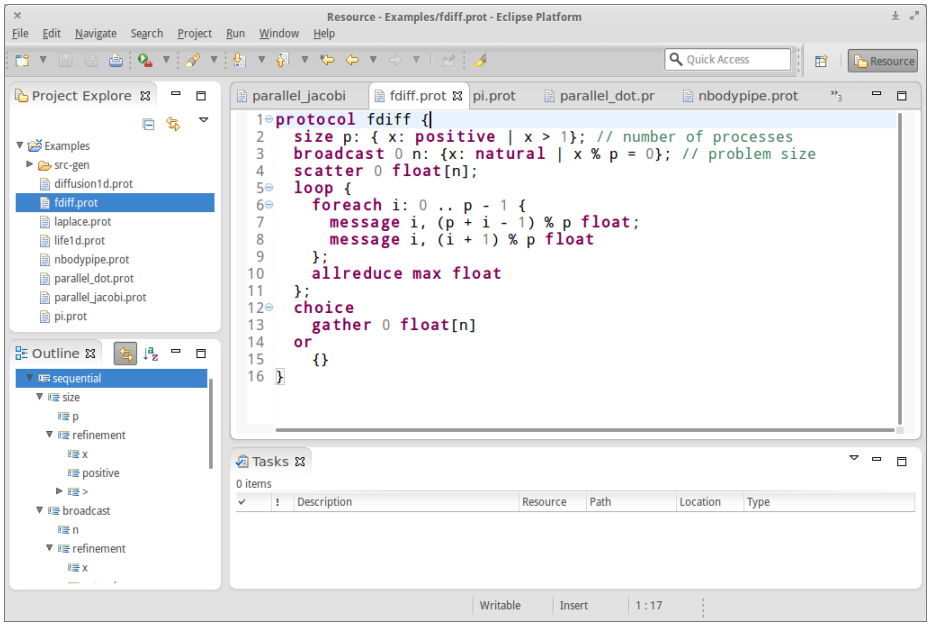

Fig. 3. Protocol for the finite differences problem

on a ring topology. The various message exchanges distribute boundary values necessary to local calculations. Different send/receive orders for different ranks (lines 18–21, lines 23–25, and lines 28–31) aim at avoiding deadlock situations, given that we use blocking (synchronous and unbuffered) primitives, both for the sender and the receiver. After value exchange, and still inside the loop, the global error is calculated via a reduction operation and communicated to all participants (MPI_Allreduce, line 33). The loop ends when the convergence condition is attained, or after a pre-defined number of iterations. Finally, after the loop, if the algorithm converged, rank 0 gathers the solution, receiving from each participant (including itself) a part of the vector (using MPI_Gather, lines 37–38).

The code for our running example is extremely sensitive to variations in the structure of MPI operations. For example, the omission of any message send/receive operation in lines 18–31 leads to a deadlock where at least one process will be eternally waiting for a send or receive operation to terminate. Another example: exchanging lines 19 and 20 leads to deadlock where ranks 0 and 1 will be forever waiting for one another.

4 The protocol specification language

The protocols governing communications in an MPI program are described by a dependent-type language. We informally introduce the language via the protocol type for the running example. The type is in Figure 3; the syntax of the language is in Figure 4. The appendix contains the complete definition. Line 2 in Figure 3 introduces the number

of participants, a value larger than 1 since the protocol uses point-to-point communications. This value is bound to variable p in all participants. In line 3, participant rank 0 broadcasts the dimension of the problem which must be a multiple of p; this value is bound to variable n in all participants. The values transmitted in messages (characterised by *datatypes* or *index types*, D, in Figure 4) are integer or floating points values, arrays (e.g., **float**[n]), and refined datatypes. For example, {x:**natural**|x%p==0} denotes a non-negative integer that is a multiple of the number of participants p. Datatype **natural** is itself an abbreviation for {x:**integer**|x>=0}.

The type in Figure 3 contains also communication primitives. Line 4 describes an array distribution operation, initiated by the participant with rank 0, of a floating point array of size n where each participant k ($0 \leq k < p$) receives the k-th part (of length n/p) of the original array. In the concrete syntax we allow omitting the identifier that describes the part of the array each participant receives (cf. the syntax for **scatter** in Figure 4). Lines 7–8 describe point-to-point message exchanges. In the first case there is a message exchange between participant rank i and participant rank (i+1)%p containing a floating point number.

Line 10 describes an operation where every participant communicates a floating point number, the maximum of which is calculated and distributed amongst all participants. **allreduce** is a derived operator, based on **reduce** and **broadcast**. Type **allreduce** op x:D is short for **reduce** 0 op _:D'; **broadcast** 0 x:D where **reduce** 0 op x:D describes a collective operation whereby rank 0 receives the result of applying operator op to a value transmitted by each participant; variable x denotes the kind of values transmitted. Line 13 describes the operation inverse of **scatter**, gathering on participant 0 an array of length n comprising sections originating from the various participants. Once again, the concrete syntax allows omitting the identifier for the sections (cf. Figure 4).

The loop **foreach** in lines 6–9 is intuitively equivalent to the sequential composition of p copies of lines 7–8, with i replaced by 0, 1, ..., p−1. Lines 5 and 12 are examples of what we call *collective control structures*. The first case describes a loop where all participants decide together, but without exchanging any message, whether to continue or to abandon the loop. The second case describes a choice where every participant decides jointly (again, without communicating) to continue to line 13 or to line 15. Line 15 denotes an empty block of operations, skip in Figure 4. Except for **val**, all constructors in our language where introduced through the example. **val** is the dependent type constructor, abstracting over concrete values on all ranks. It is used mainly to model constants in the program, including values read from the command line.

Not all types conforming to the grammar in Figure 4 are of interest. The most common case is related to ranks appearing in types. Examples include: a message to self: **message** 2 2 **integer**; a message to a non-existent participant: **message** 0 −5 **float** or **size** p :{y:int|y<10}; **message** 0 10 **float**. Errors need not be apparent, as in **size** p: {y:**integer**|y>1}; **broadcast** 0 n:**natural**; **message** n 0 **float** where n is not guaranteed to be in the range [0..p[unless we replace **natural** by {x:**natural**|x<p}.

An excerpt of the rules defining *type formation* are in Figure 5. All rules are algorithmic, so that they can be straightforwardly implemented. Types are checked against index contexts Γ, an ordered map from index variables x to datatypes D. A sequent of the form $\Gamma_1 \vdash T \dashv \Gamma_2$ says that type T is well formed on context Γ_1 and produces a

$$
\begin{array}{rl}
T & ::= \mathsf{val}\ x\colon D \mid \mathsf{message}\ i\ i\ D \mid \mathsf{size}\ x\colon D \mid \\
& \quad \mathsf{broadcast}\ i\ x\colon D \mid \mathsf{scatter}\ i\ x\colon D \mid \mathsf{gather}\ i\ x\colon D \mid \mathsf{reduce}\ i\ \mathrm{op}\ x\colon D \mid \\
& \quad \mathsf{loop}\ T \mid \mathsf{choice}\ T\ \mathsf{or}\ T \mid \mathsf{foreach}\ x\colon i..i\ T \mid T;T \mid \mathsf{skip} \\
D & ::= \mathsf{integer} \mid \mathsf{float} \mid D[i] \mid \{x\colon D \mid p\} \\
i & ::= x \mid n \mid i+i \mid \mathsf{max}(i,i) \mid \mathsf{length}(i) \mid i[i] \mid \ldots \\
p & ::= \mathsf{true} \mid i \leq i \mid p\ \mathsf{and}\ p \mid \ldots \\
\mathrm{op} & ::= \mathsf{max} \mid \mathsf{min} \mid \mathsf{sum} \mid \ldots \qquad\qquad \Gamma ::= \cdot \mid \Gamma, x\colon D
\end{array}
$$

Fig. 4. Syntax

$$
\frac{\Gamma_1 \vdash T_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash T_2 \dashv \Gamma_3}{\Gamma_1 \vdash T_1;T_2 \dashv \Gamma_3}
\qquad
\frac{\Gamma \vdash \mathsf{p} :: \_ \qquad \Gamma \vdash i :<: [0..\mathsf{p}[\qquad \Gamma \vdash D}{\Gamma \vdash \mathsf{broadcast}\ i\ x\colon D \dashv \Gamma, x\colon D}
$$

$$
\frac{\Gamma \vdash \mathsf{p} :: \_ \qquad \Gamma \vdash i_1 :<: [0..\mathsf{p}[\qquad \Gamma \vdash i_2 :<: [0..\mathsf{p}[\qquad \Gamma \vDash i_1 \neq i_2 \qquad \Gamma \vdash D}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D \dashv \Gamma}
$$

Fig. 5. Type formation (selected rules)

context Γ_2. The rules for *datatype formation*, $\Gamma \vdash D$, for *proposition formation*, $\Gamma \vdash p$, for *index kinding*, $\Gamma \vdash i :: D$, for *datatype subtying* $\Gamma \vdash D <: D$, and for *formulae entailment*, $\Gamma \vDash p$, are either straightforward or standard (cf. [8]) and are described in the appendix. Notation $\Gamma \vdash i :<: D$ combines index kinding with datatyping subtyping, and abbreviates $\Gamma \vdash i :: D' \wedge \Gamma, x\colon \{y\colon D \mid y = i\} \vdash D' <: D$ with x, y fresh.

The rule for the sequential composition in Figure 5 makes the nature of second context apparent: Γ_2, the output of checking T_1, is passed as input for checking T_2. In turn, Γ_3, the output of checking T_2 is the output of the sequential composition. The rule for **broadcast** is illustrative of how new index variables are introduced in the outgoing context. The rule reads the number of participants p (a distinguished datatype variable, not available to programmers, and introduced by the type formation rule for size) from the context, checks that i is a valid rank, and that D is well formed. The rule for messages checks that the "from" and the "to" ranks are valid (are in the range $[0..\mathsf{p}[$) and different from one another. Datatype D must be well formed. The resulting context is the incoming context, since no bindings are introduced by the message construct.

Describing the behaviour of a collection of participants, types are naturally equipped with an operational semantics. Types are endowed with a *structural congruence* relation, equating $\mathsf{skip};T \equiv T$, $\mathsf{loop}\ \mathsf{skip} \equiv \mathsf{skip}$, and $\mathsf{choice}\ \mathsf{skip}\ \mathsf{or}\ \mathsf{skip} \equiv \mathsf{skip}$. A vector of types, denoted by $\vec{T}$ or S, describes a *state*. States evolve via a reduction relation $S \to S$. Given a type T and a number of participantors n, the initial state of our machine is given by $\mathrm{load}(T, n)$, a vector of $n > 1$ identical copies of T. A few selected rules extracted from the reduction relation are in Figure 6. The rule for broadcast is an operation that synchronises on all types: they must all be of the form $\mathsf{broadcast}\ i_k\ x\colon D; T'_k$. By $i \downarrow n$ we mean that index term i evaluates to an integer value n. Such value, the rank of the root type, must be valid, i.e., $0 \leq n < |\vec{T}|$ where $|\vec{T}|$ denotes the length of the vector. Given any value v of datatype D, denoted by $\vdash v\colon D$, the resulting types are all of the form $T'_k[v/x]$, that is, all types evolve to T'_k where v replaces the free occurrences of x. There are two rules for messages depending on whether the types participate or not in the reduction. The first rule identifies two types ready to synchronise, of ranks j

$$\frac{T_k = \mathsf{broadcast}\ i_k\ x\colon D; T'_k \quad i_k \downarrow n \quad 0 \leq n < |\vec{T}| \quad \vdash v\colon D \quad T''_k = T'_k[v/x]}{\vec{T} \to \vec{T}''}$$

$$\frac{|\vec{T}_1| = j \qquad |\vec{T}_2| = k - j - 1 \qquad i_1 \downarrow j \qquad i_2 \downarrow k \qquad i_3 \downarrow j \qquad i_4 \downarrow k}{\vec{T}_1, (\mathsf{message}\ i_1\ i_2\ D; T_j), \vec{T}_2, (\mathsf{message}\ i_3\ i_4\ D; T_k), \vec{T}_3 \to \vec{T}_1, T_j, \vec{T}_2, T_k, \vec{T}_3}$$

$$\frac{|\vec{T}_1| = l \qquad i_1 \downarrow j \qquad i_2 \downarrow k \qquad j, k \neq l}{\vec{T}_1, (\mathsf{message}\ i_1\ i_{2\_}; T), \vec{T}_2 \to \vec{T}_1, T, \vec{T}_2}$$

Fig. 6. Operational semantics for machine states (selected rules)

and k. In this case both types advance to their continuations, T_j and T_k, whereas the remaining types remain unchanged. The second rule for messages advances the types not involved in communication. These two rules, perform a sort of on-the-fly projection, in the parlance of multiparty session types [11].

Equipped with the notions of type formation and reduction, we are in a position to state our main result, namely that well formed types do not get stuck. We say that a state is *halted* when all its types are equivalent to skip.

Theorem 1. *If* $\vdash T$ *and* $\mathrm{load}(T, n) \to^* S_1$ *then either* $\mathrm{halted}(S_1)$ *or* $S_1 \to S_2$.

Proof (outline). The proof is standard, albeit long. We rely on a soundness and on a type safety result. Soundness ($\Gamma \vdash S_1$ and $S_1 \to S_2$ implies $\Gamma \vdash S_2$) follows by induction on the structure of the derivation for $\Gamma \vdash S_1$. It relies, in turn, on results such as a substitution lemma and weakening. Type safety ($\Gamma \vdash S_1$ implies either $\mathrm{halted}(S_1)$ or $S_1 \to S_2$) follows by *reductio ad absurdum*, by analysing each non-halted stated that is impeded from progressing.

5 Protocol verification

The conformance of C+MPI programs against protocol specifications is checked by the VCC tool [4]. The verification process requires (*vide* Figure 1): (1) a description of the protocol as defined in Section 4; (2) an annotated MPI library that contains the implementation of the operational semantics, as (partially) described in Figure 6, and contract specifications for the MPI primitives that enforce the correct use of the prescribed protocol; and (3) the C+MPI source code.

The first thing we address is the annotation of the source code in order to persuade VCC to accept the program. This step includes the replacement of source code that VCC currently does not support, such as dynamic memory allocation (which we replace by fixed-size arrays), functions with a variable number of arguments as for example `printf` and `scanf` (which we either remove altogether or provide semantically equivalent replacements), and floating point operations (which we abstract under function calls). Moreover, VCC is a tool for verifying concurrent programs, and thus requires fine-grained annotations regarding memory usage.

Upon completion of this first step, we can focus entirely on protocol verification. The Eclipse plug-in tool checks that the protocol description is well formed. The tool

```
1  _(ghost SessionType sessionType =
2    seq(
3     action(size(), intRefin(\lambda \integer p; p > 2), 1),
4     abs(body(\lambda \integer p;
5      seq(
6       action(bcast(0), intRefin(\lambda \integer n; n > 1 && n % p == 0)), 1)),
7       abs(body(\lambda \integer n;
8       seq(
9        action(scatter(0), floatRefin(\lambda float _n2; \true, n)),
10       seq(
11        loop(
12         seq(
13          foreach(0, p - 1, body(\lambda \integer i;
14           seq(
15            message(i, (p + i - 1) % p, floatRefin(\lambda float x; \true, 1)),
16            message(i, (i + 1) % p, floatRefin(\lambda float x; \true, 1))))),
17          action(allreduce(MPI_MAX), floatRefin(\lambda float x; \true, 1)))),
18        choice(
19         action(gather(0), floatRefin(\lambda float x; \true, n)),
20         skip()))))))));)
```

Fig. 7. The protocol for the finite differences problem in VCC syntax

implements the rules described in Figure 5, while using the Z3 SMT solver [13] to handle the entailment relation ($\models$). If the protocol is well formed, the tool generates a C header file describing the protocol in VCC format. The next step is to include high-level annotation marks in the source code identifying collective loops/choices and, in some cases, also loops for **foreach** protocols, as well as providing values for the **val** operation. At this point, an automatic annotation tool, implemented using the Clang/LLVM framework [3], expands the high-level annotation marks into a more complex verification logic, while generating some complementary annotations. From this point on, VCC can check the conformance of the C+MPI program against the protocol. We describe the annotation tool in detail further down in this section.

Figure 7 contains the VCC datatype produced by the Eclipse plugin when run on the protocol for finite differences in Figure 3. In VCC, annotation blocks are expressed by `_(annotation block)`; the **ghost** keyword introduces declarations that assist the verification process but that have otherwise no impact on the C code. The `SessionType` datatype directly implements types T, as in Figure 9 (details in Appendix C). Primitive operations (e.g., **size**, **val**, **scatter**) are wrapped by a VCC `Action` datatype constructor that pairs the operation with the dependent type that characterises the value it uses. This extra wrapping level eases the writing of the VCC theory. For instance, **size** `p : {x: integer | x > 1}` is represented as `action(size(), intRefin(\lambda \integer p; p > 2), 1)`, where `intRefin` introduces an integer refinement type with two arguments: the first `\lambda \integer p; p > 2` encodes an anonymous function of `\integer` domain that defines the predicate; the second, indicates the length of the array (`1` in this case), given that MPI primitives exchange data as arrays only. Collective operations (and the **foreach** loop) bind a variable in the continuation type (or in the loop body). We make use of VCC anonymous functions to bind the value in the continuation and to handle substitutions. For example, **size** `p : {x: integer | x > 1}; T`, where `T` stands for the continuation type of Figure 3, is encoded as follows, while binding `p` in `T` (lines 3–15).

```
action(size(), intRefin(\lambda \integer p; p > 2), 1),
  abs(body(\lambda \integer p; T))
```

The VCC theory is divided in two parts: the first is a contract-annotated version of the MPI function signatures that ensures the conformance of the program operations against a protocol; the second encodes the type reduction relation. We illustrate contract annotation using the `MPI_Send` function.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, ...
             _(ghost GParam gParam)
             _(ghost SessionType type_in)
             _(out SessionType type_out))
 _(requires actionType(head(type_in, gParam->rank))   == send(dest))
 _(requires actionLength(head(type_in, gParam->rank)) == count)
 _(requires refTypeCheck(refType(head(type_in, gParam->rank)), buf, count))
 _(requires datatype == MPI_INT ==> \thread_local_array ((int *)buf, count))
 _(requires datatype == MPI_FLOAT ==> \thread_local_array ((float *)but, count))
 _(ensures type_out == reduce(type_in, gParam->rank))
 ...
```

Contracts are specified in VCC via pre- (**requires**) and post-conditions (**ensures**) clauses. In addition to the parameters defined by the MPI specification, we include three additional ghost parameters that keep track of the protocol reduction state: `gParam`, `type_in`, and `type_out`. Parameter `type_in` represents the type before executing the `send` operation, whereas `type_out` reflects the effect of consuming the `send` operation (we use two parameters because VCC does not allow read/write parameters). The `gParam` carries the global parameterisation for the verification logic, comprising bounds for the participant rank and a bound on the total number of participants. The bound on the number of participants may be redefined (see Section 6). Its definition is as follows.

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

When checking an `MPI_Send` operation, the type for the participant under verification must be of the form **message** i1 i2 D; T. This naturally induces a pre-condition of the form ∃ `SessionType T; seq(message i1 i2 D; T)== type_in`. But SMT solvers, in general, do not perform well with existential quantification, and VCC (which uses the Z3 SMT solver [13] internally) is not able to assert pre- or post-conditions of this form. We follow a different approach and write a ghost function `head` that computes the action at the head of the type, thus avoiding existential quantification. The first three preconditions for the `MPI_Send` contract check that the primitive operation at the head of the type is a `send` operation targeting the destination `dest` passed as an argument to `MPI_Send`, that the array dimension matches the `count` parameter, and that the value to be sent (`buf`) complies with the dependent type specification. Ghost functions `actionType`, `actionLength`, and `refType` are helper functions that extract information from the head action. Ghost function `refTypeCheck` simply applies the lambda function encoding the dependent type to `buf` and asserts the function body (with the correct substitution) for all (`count`) elements of the buffer. We follow the same route for writing the post-conditions, and make use of a ghost function `reduce`, which implements the reduction relation on protocol types, to advance the type. Notice that both ghost functions `head` and `reduce` receive an extra parameter for the rank of the participant under verification. We need this information to handle **message** operations (cf. Figure 6), in particular to check whether i_1

or i_2 evaluate to `gParam->rank`. The last two pre-conditions check that the `buf` memory region if valid and has the expected type. The other MPI communication primitives have similar contracts.

The VCC theory we developed follows the rules presented in Section 4. We illustrate some of its fragments. The verification process visits the program from MPI initialisation (call to `MPI_Init`, Figure 2, line 3) to shutdown (`MPI_Finalize`, same figure, line 40). A ghost `sessionType` variable, representing the protocol is initialised by the contract for primitive `MPI_Init`. Afterwards, the protocol is progressively reduced (i.e., the ghost variable is changed) through the contracts of MPI primitives or annotations that handle control flow. The goal is that the ghost variable reaches a state congruent to `skip()` at the shutdown point (the call to `MPI_Finalize`). This flow is illustrated by the contracts for `MPI_Init` and `MPI_Finalize`.

```
int MPI_Init(... _(ghost GParam param) _(out SessionType type_out))
  _(ensures type_out == sessionType)
  ...
int MPI_Finalize(... _(ghost SessionType type_in))
  _(ensures congruent(type_in, skip())
  ...
```

The `congruent` predicate is a straightforward implementation of the congruence relation in Section 4. We now focus on the `reduce` function, particularly the cases concerning message reduction. The signature of the function and its axioms are as follows.

```
_(pure SessionType reduce(SessionType st, \integer rank);)
_(axiom \forall \integer rank, dest; SessionData sd; SessionType t;
  reduce(seq(message(rank, dest, sd), t), rank) == t)
_(axiom \forall \integer rank, from; SessionData sd; SessionType t;
  reduce(seq(message(from, rank, sd), t), rank) == t)
_(axiom \forall \integer rank, from, to; SessionData sd; SessionType t;
  from != rank && to != rank ==>
    reduce(seq(message(from, to, sd), t), rank) == next(t, rank))
```

Following the rules in Figure 6, the first two axioms handle messages from or to the participant under verification, while the last axiom ignores messages not addressed to the participant. As for control flow, collective choices and loops in particular, direct annotations are necessary in the program body. These are automatically generated in a process detailed below. Here, we focus now on its meaning, using the collective loop of the finite differences as an example.

```
  _(ghost SessionType body = loopBody(type);)
  _(ghost SessionType continuation = next(type,gParam->rank);)
  while (!converged(globalerr) && iter < MAX_ITER) {
    _(ghost type = body;)
    ...
    _(assert congruent(type, skip())
  }
  _(ghost type = continuation);
```

The fragment illustrates the extraction of the protocols corresponding to the loop `body` and its `continuation`. The protocol for the body must be a `loop` type (as in, e.g., Figure 7, line 11). The verification procedure asserts that the loop protocol body is reduced to a term congruent to `skip()`. After the loop, verification proceeds by using the loop `continuation` as the type.

A significant part of the required program annotations are introduced automatically. The annotator uses the Clang/LLVM [3] framework to traverse the syntactic tree of a C

Original code	Annotated code

/* collective choices */
if (_collective_(e)) {
  ...
} else {
  ...
}
	_(ghost SessionType cIfT = choiceTrue(type);)
	_(ghost SessionType cIfF = choiceFalse(type);)
	_(ghost SessionType cCont = next(type, gParam->rank);)
	if (e) {
	  _(ghost type = cIfT;)
	  ...
	  _(assert congruent(type, skip()))
	} else {
	  _(ghost type = cIfF;)
	  ...
	  _(assert congruent(type, skip()))
	}
	_(ghost type = cCont;)

/* collective loops */
while (_collective_(e)) {
  ...
}
/* similarly for do-while
  and for loops */
	_(ghost SessionType lBody = loopBody(type);)
	_(ghost SessionType lCont = next(type, gParam->rank);)
	while (e) ... {
	  _(ghost type = lBody;)
	  ...
	  _(assert congruent(type, skip()))
	}
	_(ghost type = lCont;)

/* foreach */
int v = ...;
...
while (_foreach_(v,e)) {
  ...
}
/* similarly for do-while
  and for loops */
	int v = ...;
	...
	_(ghost STFunc fEval = foreachBody(type);)
	_(ghost SessionType fCont = foreachCont(type);)
	while (e) ... {
	  _(ghost type = fEval[v];)
	  ...
	  _(assert congruent(type, skip())
	}
	_(ghost type = fCont;)

Table 1. Automated generation of annotations

program and generate a new, annotated, version. The annotator is guided by calls to MPI primitives (e.g., for the purpose of declaring ghost parameters in function signatures), and, more importantly, by high-level programmer "hints" that should be understood as protocol-related annotations. The latter refers to control flow aspects, particularly collective loops/choices, that are hard to infer automatically using standard static analysis techniques (for this purpose, symbolic execution or model checking may be used [19]).

In any case, the protocol-related annotations are simple marks on the C program, whereas corresponding annotations produced by the tool are quite more complex, as illustrated in Table 1. The `_collective_` and `_foreach_` marks, shown on the left column of Table 1, are used to match collective loops/choices and (in some cases) **foreach** protocols, respectively. From these marks, the annotator produces annotations shown in the table's right column, that are in line with the above discussion on protocol reductions. A `_foreach_` mark is necessary when a **foreach** type is matched against an actual C loop. This is not the case in our running example, where the **foreach** protocol (lines 6–9, Figure 3) requires no C-loop, given that each participant executes a different step of the protocol **foreach** loop. Contrast this situation with the protocol below where participant with rank `0` sends a message to every other rank.

```
foreach i = 0 .. procs-1 message 0 i float
```

The protocol could be matched, for example, by a loop annotated as follows.

```
if (rank == 0)
  for (i = 0; _foreach_(i, i < procs); i++)
    MPI_Send (... i ...); // 0 sends to i
else
  MPI_Recv (... 0 ...); // receives from 0
```

Then, the `_foreach_` annotation will trigger the generation of annotations as described in the third row of Table 1. In this case, the loop essentially needs to be explicitly unfolded, and this is done by the verification logic. The overall job of the annotator, when run on the running example, is summarised in the appendix, Figure 8.

In spite of our endeavours towards a fully automated verification procedure, programs typically require complementary annotations that must be introduced manually by a programmer. A major source for manual annotations concerns the concurrent use of shared memory (remember that VCC is a tool for the verification of concurrent C code). For instance, in our running example we need to introduce the following annotations:

```
  while (!converged(globalerr) && iter < MAX_ITER)
    _(writes &globalerr)
    _(writes \array_range(local, (unsigned) lpsize + 2))  { ... }
```

The two **writes** clauses above identify memory that becomes updated within the loop.

Another major source for manual annotations concerns C function contracts, which VCC requires for modular (compositional) verification. Manual annotations may also reflect assumptions left implicit by the programmer, but that must be made explicit for verification. For instance in the C code for the running example (Figure 2, line 7) we must state that the problem size is a natural number multiple of the number of participants (cf. Figure 3, line 3).

```
psize = atoi(argv[1]);
_(assume psize > 1 && psize % procs == 0)
```

6 Evaluation

We evaluated the verification procedure by using textbook examples [6,9,15] and the FEVS suite [21]. The protocols for the tests are given in Appendix B. The examples had to be adjusted for different reasons. In some cases non-blocking communication primitives (`MPI_Isend`, `MPI_Irecv`, `MPI_Wait` and `MPI_Waitall`) were replaced by blocking primitives (`MPI_Send` and `MPI_Recv`). Other aspects had to be adjusted due to limitations of VCC as described in Section 5. We also stripped some sections of code that were irrelevant for the purpose of protocol verification. The evaluation considers two measures: the verification time and the comparison between the number of automatically generated and that of manually introduced annotations.

Table 2 shows the results for the verification time (in seconds) of VCC for each example, considering different upper bounds on the number of participants. The results shown correspond to the average time of 5 executions, measured on an Intel 2.4 Ghz machine with 4 GB of RAM running Windows 7. Empty entries indicate that the verification did not end complete due to memory exhaustion.

The first observation is the stable performance in the case of Jacobi iteration: all participants have the same communication behaviour, the code uses only collective communication operations, and does not include **foreach** protocols. In all other examples we

Program	2	4	8	16	32	64	128
Diffusion 1-D [21]	6.0	6.6	10.6	60.7	787.9	—	—
Finite differences [6]	3.6	4.5	13.5	157.5	—	—	—
Jacobi iteration [15]	3.9	4.2	4.2	4.3	4.0	4.2	4.2
Laplace solver [21]	10.2	64.0	1205.1	—	—	—	—
N-body simulation [9]	5.0	5.5	6.9	20.1	139.9	1289.9	—
Pi calculation [9]	1.4	2.2	1.7	4.0	20.5	1009.6	—
Vector dot product [15]	4.3	5.3	7.5	60.4	717.1	—	—

Table 2. Verification time (seconds)

witness a "participant-explosion" problem. In all of these, a combination of factors is at stake: point-to-point communications are employed, hence participants exhibit distinct and in some cases intricate behaviour, and the unfolding of **foreach** protocols depends directly on the number of participants.

Program	LOC	A	M	A+M	M/F
Diffusion 1-D [21]	206	49	10	59	2.0
Finite differences [6]	256	41	14	55	2.3
Jacobi iteration [15]	389	82	57	139	7.0
Laplace solver [21]	120	73	7	80	7.0
N-body simulation [9]	300	88	24	112	3.7
Pi calculation [9]	77	26	1	27	0.5
Vector dot product [15]	146	53	10	63	3.3

A: automated annotations; M: manual annotations; F: number of functions

Table 3. Annotation effort (LOC)

The results for the annotation effort are shown in Table 3. For each example the table shows the number of lines of code in each program, the number of automatically and manually generated annotations, the number of annotated functions, and the ratio of manual annotations per annotated function. The automatic annotations comprise the definition of the protocol in the VCC protocol header (generated by the Eclipse tool) and the other automatic annotations generated by the annotator tool. The overall observation is that the number of manual annotations is significantly lower than the number of automatic annotations, and that the number of lines of manual annotations per C function ranges between 2 and 7, a perfectly acceptable number in our view.

7 Conclusion

The results described in this article are part of a larger programme on verification of message passing, real-world programs. With the infrastructure we have built, we can, with limited human intervention, verify programs with 400 lines of C code.

We worked with programs taken from text books [6,9,15] and test suits [21], well known algorithms for which it was relatively simple to generate the protocol, programs with carefully crafted code that required few manual annotations, and programs that only work with a somewhat restricted subset of MPI primitives. But even in these cases we sometimes experienced problems with a large number of participants.

What happens with real-world programs, programs that don not verify some of these assumptions? We will have, among other things, to a) support more MPI primitives (we

plan to address immediate operations, MPI_Isend, MPI_Irecv and MPI_Wait); b) work to minimise the number of necessary manual annotations (by using inference techniques), and c) try to curb the foreach explosion (by using induction, rather than expansion).

References

1. Aananthakrishnan, S., Bronevetsky, G., Gopalakrishnan, G.: Hybrid approach for data-flow analysis of MPI programs. In: ICS. pp. 455–456. ACM (2013)
2. Bronevetsky, G.: Communication-sensitive static dataflow for parallel message passing applications. In: CGO. pp. 1–12. IEEE Computer Society (2009)
3. Clang: a C language family frontend for LLVM. http://clang.llvm.org/ (2012)
4. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs, LNCS, vol. 5674, pp. 23–42. Springer (2009)
5. Forum, M.: MPI: A Message-Passing Interface Standard—Version 3.0. High-Performance Computing Center Stuttgart (2012)
6. Foster, I.: Designing and building parallel programs. Addison-Wesley (1995)
7. Gopalakrishnan, G., Kirby, R.M., Siegel, S., Thakur, R., Gropp, W., Lusk, E., De Supinski, B.R., Schulz, M., Bronevetsky, G.: Formal analysis of MPI-based parallel programs. CACM 54(12), 82–91 (2011)
8. Gordon, A.D., Fournet, C.: Principles and applications of refinement types. In: International Summer School Logics and Languages for Reliability and Security, Marktoberdorf, August 2009. IOS Press (2010), also Technical Report MSR-TR-2009-147
9. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message passing interface, vol. 1. MIT press (1999)
10. Honda, K., Mukhamedov, A., Brown, G., Chen, T., Yoshida, N.: Scribbling interactions with a formal foundation. Distributed Computing and Internet Technology pp. 55–75 (2011)
11. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)
12. Marques, E., Martins, F., Vasconcelos, V., Ng, N., Martins, N.: Towards deductive verification of MPI programs against session types. In: PLACES. EPTCS (2013), to appear
13. Moura, L.D., N.Bjørner: Z3: an efficient SMT solver. In: TACAS. pp. 337–340 (2008)
14. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe parallel programming with message optimisation. In: TOOLS Europe. LNCS, vol. 7304, pp. 202–218. Springer (2012)
15. Pacheco, P.: Parallel programming with MPI. Morgan Kaufmann (1997)
16. Schulz, M., de Supinski, B.R.: PNMPI tools: a whole lot greater than the sum of their parts. In: Supercomputing. ACM (2007)
17. Scribble homepage. http://www.scribble.org/
18. Siegel, S.F., Zirkel, T.K.: Automatic formal verification of MPI-based parallel programs. In: PPoPP'11. pp. 309–310. ACM (2011)
19. Siegel, S., Gopalakrishnan, G.: Formal analysis of message passing. In: VMCAI. Lecture Notes in Computer Science, vol. 6538, pp. 2–18 (2011)
20. Siegel, S., Rossi, L.: Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In: EuroPVM/MPI. LNCS, vol. 5205, pp. 274–282 (2008)
21. Siegel, S., Zirkel, T.: FEVS: A Functional Equivalence Verification Suite for high performance scientific computing. Mathematics in Computer Science 5(4), 427–435 (2011)
22. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B.R., Schulz, M., Bronevetsky, G.: A scalable and distributed dynamic formal verifier for MPI programs. In: Supercomputing. pp. 1–10. IEEE (2010)
23. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: PPoPP. pp. 261–270. ACM (2009)

A Running example with annotations

```
1  int main(int argc, char** argv _ampi_arg_decl) {
2    int procs, rank;        // Number of processes, process rank
3    MPI_Init(&argc, &argv);
4    MPI_Comm_size(MPI_COMM_WORLD, &procs);
5    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6    ...
7    if (rank == 0) {
8      psize = atoi(argv[1]);
9      _(assume psize > 0 && psize % procs == 0)
10   }
11   MPI_Broadcast(0, &psize, 1, MPI_INT, MPI_COMM_WORLD);
12   lsize = psize / procs;
13   if (rank == 0) read_vector(work, psize);
14   MPI_Scatter(work, lsize, MPI_FLOAT,&local[1],lsize,MPI_FLOAT,0,MPI_COMM_WORLD);
15   ...
16   _(ghost SessionType lBody = loopBody(_type);)
17   _(ghost SessionType lCont = next(_type);)
18   while (!converged(globalerr) && iter < MAX_ITER)
19     _(writes &globalerr)
20     _(writes \array_range(local, (unsigned) lpsize + 2))
21   {
22     _(ghost _type = lBody;)
23     if (rank == 0) {
24       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
25       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
26       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
27       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
28     }
29     else if (rank == procs - 1) {  ... }
30     else { ... }
31     MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
32     _(assert congruent(_type,skip())
33   }
34   _(ghost _type = lCont;)
35   ...
36   _(ghost SessionType cTrue = choiceTrue(_type);)
37   _(ghost SessionType cFalse = choiceFalse(_type);)
38   _(ghost SessionType cCont = next(_type);)
39   if (converged(globalerr)) { // Gather solution at rank 0
40     _(ghost _type = cTrue)
41     MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0,
           MPI_COMM_WORLD)
42     ...
43     _(assert congruent(_type, skip()))
44   } else {
45     _(ghost _type = cFalse;)
46     ...
47     _(assert congruent(_type,skip()))
48   }
49   _(ghost _type = cCont)
50   MPI_Finalize();
51   return 0;
52 }
```

Fig. 8. The program for the finite differences with VCC annotations

B Protocols for the various tests in Section 6

Diffusion

```
protocol diffusion1d {
  size p: {x: integer | x > 1};
  val nx: {x: natural | x % p = 0};
  broadcast 0 integer;
  broadcast 0 float;
  broadcast 0 integer;
  broadcast 0 integer;
  foreach i: 1 .. p-1
    message  0, i float[nx/p];
  foreach i: 1 .. p-1
    message i, 0 float[nx/p];
  loop {
    foreach i: 1 .. p-1
      message i, i-1 float;
    foreach i: 0 .. p-2
      message i, i+1 float;
    choice
      foreach i: 1 .. p-1
        message i, 0 float[nx/p]
    or
      {}
  }
}
```

Finite differences

```
protocol fdiff {
  size p: {x: integer | x > 1};
  broadcast 0 n: {x: positive | x % p = 0};
  scatter 0 float[n];
  loop {
    foreach i: 0 .. p - 1 {
      message i, (p + i - 1) % p float;
      message i, (i + 1) % p float
    };
    allreduce max float
  };
  choice
    gather 0 float[n]
  or
    {}
}
```

Jacobi iteration

```
protocol parallel_jacobi  {
  val maxDim: natural;
  size p: {x: integer | x > 1 and maxDim % x = 0};
  val n :{y: natural | y <= maxDim and y % p = 0};
  scatter 0 float[maxDim * n];
  scatter 0 float[n];
  allgather float[n];
  loop
    allgather float[n];
  choice
    gather 0 float[n]
  or
    {}
}
```

Laplace solver

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

N-body simulation

```
protocol nbodypipe {
  size p: {x: integer | x > 1};
  val n: {x: natural | x % p = 0};
  allgather integer[p];
  loop {
    loop
      choice
        foreach i: 0 .. p-1
          message i, (i + 1) % p float[n * 4]
      or
        {};
    allreduce min float
  }
}
```

Pi calculation

```
protocol pi {
  size p: {x: integer | x > 1};
  foreach i: 1 .. p-1
    message 0, i positive;
  reduce 0 sum float
}
```

Vector dot product

```
protocol parallel_dot {
  size p: {x: integer | x = 4};
  val n: {x: natural | x % p = 0};
  broadcast 0 integer;
  foreach i: 1 .. p-1
    message 0, i float[n/p];
  foreach i: 1 .. p-1
    message 0,i float[n/p];
  allreduce sum float;
  foreach i: 1 .. p-1
    message i, 0 float
}
```

C Main VCC definitions for Section 5

Session type representation

```
// Dependent type support
_(ghost typedef \bool \IRefinement[\integer]);
_(ghost typedef \bool \FRefinement[float]);
_(datatype SessionData {
    case intRefin    (\IRefinement, \integer);
    case floatRefin (\FRefinement, \integer);
  })
// MPI primitives and actions
_(datatype Action {
    case size();                  case val();
    case send(\integer);          case recv \integer);
    case gather(\integer);        case scatter(\integer);
    case reduce(\integer, MPI_Op); case allreduce(MPI_Op);
    case bcast(\integer);         case allgather   ();
  })
// Session type term
_(ghost typedef SessionType STFunc[\integer];)
_(datatype STBody { case body(STFunc);)
_(datatype SessionType {
    case skip    ();
    case action  (Action, SessionData);
    case message (\integer, \integer, SessionData);
    case seq     (\SessionType, SessionType);
    case choice  (SessionType, SessionType);
    case loop    (SessionType);
    case foreach (\integer, \integer, STBody);
    case abs     (STBody);
  })
```

Structural congruence

```
_(pure \bool congruent(SessionType, SessionType);)
_(axiom \forall SessionType t; congruent(t,t))
_(axiom \forall SessionType t1,t2,t3;
   congruent(t1,t2) && congruent(t2,t3) ==> congruent(t1,t3))
_(axiom \forall SessionType t1,t2,t3;
   congruent(seq(seq(t1,t2),t3), seq(t1,seq(t2,t3))))
_(axiom \forall SessionType t; congruent(seq(skip(), t),t))
_(axiom \forall SessionType t; congruent(seq(loop(skip()),t),t))
_(axiom \forall SessionType t; congruent(seq(choice(skip(),skip()),t),t))
```

Session type reductions

```
_(pure SessionType head  (SessionType st, \integer rank);)
_(pure SessionType reduce(SessionType st, \integer rank);)
// Reduction through congruence
_(axiom \forall SessionType t1,t2,t3; \forall \integer rank;
  congruent(seq(t1,t2),t3) ==> head(seq(t1,t2),rank) == head(t3,rank))
_(axiom \forall SessionType t1,t2,t3; \forall \integer rank;
  congruent(seq(t1,t2),t3) ==> reduce(seq(t1,t2),rank) == reduce(t3,rank))
// Point to point messages (on-the-fly projection)
_(axiom \forall \integer rank, dest; SessionData sd;  SessionType t;
  head(seq(message(rank, dest, sd), t), rank) == action(send(dest),sd))
_(axiom \forall \integer rank, dest; SessionData sd; SessionType t;
  reduce(seq(message(rank,dest, sd), t), rank) == t)
_(axiom \forall \integer rank, from; SessionData sd;  SessionType t;
  head(seq(message(from, rank, sd), t), rank) == action(recv(from),sd))
_(axiom \forall \integer rank, from; SessionData sd; SessionType t;
  reduce(seq(message(from, rank, sd), t), rank) == t)
_(axiom \forall \integer rank, from, to; SessionData sd; SessionType t;
  from != rank && to != rank ==>
  head(seq(message(from, to, sd), t), rank) == head(t, rank))
_(axiom \forall \integer rank, from, to; SessionData sd; SessionType t;
  from != rank && to != rank ==>
  reduce(seq(message(from, to, sd), t), rank) == reduce(t, rank))
// MPI primitives and actions
_(axiom \forall Action p; \forall SessionData sd; \forall SessionType t;
        \forall \integer rank;
  head(seq(action(p, sd), t), rank) == action(p,sd))
_(axiom \forall Action p; \forall SessionData sd; \forall SessionType t;
        \forall \integer rank;
  reduce(seq(action(p, sd), t), rank) == t)
// For-each protocols
_(axiom \forall SessionType t; \forall STFunc b;
        \forall \integer lo,hi,rank;
   lo > hi ==> head(seq(foreach(lo,hi,body(b)),t), rank) == head(t,rank))
_(axiom \forall SessionType t; \forall STFunc b;
        \forall \integer lo,hi,rank;
   lo > hi ==> reduce(seq(foreach(lo,hi,body(b)),t), rank) == reduce(t,rank))
_(axiom \forall SessionType t; \forall STFunc b;
        \forall \integer lo,hi,rank;
 lo <= hi ==>
    head(seq(foreach(lo,hi,body(b)),t),rank)
 == head(seq(b[lo], seq(foreach(lo+1,hi,body(b)),t)), rank))
_(axiom \forall SessionType t; \forall STFunc b;
         \forall \integer lo,hi,rank;
 lo <= hi ==>
    reduce(seq(foreach(lo,hi,body(b)),t),rank)
 == reduce(seq(b[lo], seq(foreach(lo+1,hi,body(b)),t)), rank))
// Choices and loops
_(axiom \forall SessionType t1,t2; \forall \integer rank;
   reduce(seq(loop(t1),t2), rank) == t2)
_(axiom \forall SessionType t1,t2,t3; \forall \integer rank;
   reduce(seq(choice(t1,t2),t3), rank) == t3)
```

D The type language

Syntax The type language is generated by the grammar in Figure 9. It relies on two base sets: that of variables (denoted x, y, z), and integer values (j, k, l, m, n). There is one distinguished variable, p, not available in the programmer's syntax. We use it to denote the number of processes. Variable binding is as follows. Types $\mathsf{foreach}\ x\colon i_1..i_2\ T$ and $\mathsf{val}\ x\colon D; T$ bind the free occurrences of x in T. Similarly for size, broadcast, scatter, gather, and reduce. Datatype $\{x\colon D \mid p\}$ binds the occurrences of x in p.

Type formation Type, datatype, and proposition formation are defined by the rules in Figure 10. These relations rely on index kinding, datatype subtype and formulae entailment, all defined in Figure 11. Notation $\Gamma + x\colon D$, replaces the entry for x in Γ, by $x\colon D$, if x is in Γ; otherwise just adds $x\colon D$ to Γ. We assume all operators are equipped with a type signature, denoted $\mathrm{op}\colon \vec{D}' \to D$. All rules are algorithmic. We annotate each system with its mode. For example, type formation expects a context and a type and delivers a type, denoted by mode: `II0`.

Operational semantics Figure 12 introduces values and machine states. We rely on a further base set: floating point values, denoted by f. Figure 13 introduces the typing rules for values and machine states. Figure 14 presents the rules for structural congruence. Notation $|\vec{T}|$ denotes the length of vector $\vec{T}$. The operational semantics rely on an evaluation function, $i \downarrow n$ where i is an index type and n is a integer value. If i contains variables, evaluation is undefined. The operational semantics for states is in Figure 15.

Derived constructs Type constructors allgather and allreduce are derived. The syntactic expansion, type formation rules and reduction rules are in Figure 16.

Main result The main result is Theorem 4. We build the result gradually.

Lemma 1 (Weakening). *If $\Gamma \vdash D_1$ then $\Gamma, x\colon D_2 \vdash D_1$*

Lemma 2 (Agreement).

1. *If $\Gamma \vdash i :: D$ then $\Gamma \vdash D$*
2. *If $\Gamma \vdash D_1 <: D_2$ and $\Gamma \vdash D_1$ then $\Gamma \vdash D_2$*

Lemma 3. *If $\Gamma_1 \vdash T_1 \dashv \Gamma_2$ and $T_1 \equiv T_2$ then $\Gamma_1 \vdash T_2 \dashv \Gamma_2$.*

Lemma 4 (Substitution lemma). *If $\Gamma_1, x\colon D \vdash T \dashv \Gamma_2$ and $\vdash v\colon D$ then $\Gamma_1 \vdash T[v/x] \dashv \Gamma_2$*

The load function builds the initial machine state.

$$\mathrm{load}(T, n) \triangleq T, \ldots, T \qquad (n > 1 \text{ copies})$$

Lemma 5 (Load). *If $\Gamma_1 \vdash T \dashv \Gamma_2$ then $\Gamma_1 \vdash \mathrm{load}(T, n) \dashv \Gamma_2$.*

We write $\Gamma \vdash S$ when $\Gamma \vdash S \dashv \Gamma'$ for some uninteresting Γ'.

Types

$$T \;::=\; \mathsf{val}\; x\colon D \mid \mathsf{size}\; x\colon D \mid \mathsf{message}\; i\; i\; D \mid \mathsf{broadcast}\; i\; x\colon D \mid$$
$$\mathsf{scatter}\; i\; x\colon D \mid \mathsf{gather}\; i\; x\colon D \mid \mathsf{reduce}\; i\; \mathrm{op}\; x\colon D \mid$$
$$\mathsf{loop}\; T \mid \mathsf{choice}\; T\; \mathsf{or}\; T \mid \mathsf{foreach}\; x\colon i..i\; TT; T \mid \mathsf{skip}$$

Datatypes (index types)

$$D \;::=\; \mathsf{integer} \mid \mathsf{float} \mid D[i] \mid \{x\colon D \mid p\}$$

Index terms

$$i \;::=\; x \mid n \mid i + i \mid \mathsf{max}(i, i) \mid \mathsf{length}(i) \mid i[i] \mid \ldots$$

Index propositions

$$p \;::=\; \mathsf{true} \mid i \leq i \mid p\; \mathsf{and}\; p \mid \ldots$$

Index contexts

$$\Gamma \;::=\; \cdot \mid \Gamma, x\colon D$$

Shorthand

$$\mathsf{nat} \triangleq \{x\colon \mathsf{integer} \mid x \geq 0\}$$
$$[i_1..i_2[\triangleq \{x\colon \mathsf{integer} \mid i_1 \leq x\; \mathsf{and}\; x < i_2\} \quad \text{with } x \text{ not in } i_1, i_2$$

Fig. 9. Syntax of types

Theorem 2 (Soundness). *If* $\Gamma \vdash S_1$ *and* $S_1 \to S_2$ *then* $\Gamma \vdash S_2$.

For type safety we rely on the halted predicate:

$$\mathrm{halted}(\vec{T}) \;\triangleq\; T_k \equiv \mathsf{skip} \qquad (\forall 0 \leq k < |\vec{T}|)$$

Theorem 3 (Type safety). *If* $\Gamma \vdash S_1$ *then either* $\mathrm{halted}(S_1)$ *or* $S_1 \to S_2$.

Theorem 4 (Main result). *If* $\vdash T$ *and* $\mathrm{load}(T, n) \to^* S_1$ *then either* $\mathrm{halted}(S_1)$ *or* $S_1 \to S_2$.

Type formation, $\Gamma \vdash T \dashv \Gamma$, mode: `II0`

$$\frac{\Gamma \vdash D}{\Gamma \vdash \mathsf{val}\ x\colon D \dashv \Gamma, x\colon D} \qquad \frac{\Gamma \vdash D \qquad \Gamma, x\colon D \vDash x > 1}{\Gamma \vdash \mathsf{size}\ x\colon D \dashv \Gamma, x\colon D + \mathsf{p}\colon D}$$

$$\frac{\Gamma \vdash \mathsf{p} :: \_ \quad \Gamma \vdash i :<: [0..\mathsf{p}[\quad \Gamma \vdash D}{\Gamma \vdash \mathsf{broadcast}\ i\ x\colon D \dashv \Gamma, x\colon D} \qquad \frac{\Gamma \vdash \mathsf{p} :: \_ \quad \Gamma \vdash i :<: [0..\mathsf{p}[\quad \Gamma \vdash D \quad \mathrm{op}\colon \vec{D}' \to D}{\Gamma \vdash \mathsf{reduce}\ i\ \mathrm{op}\ x\colon D \dashv \Gamma, x\colon D}$$

$$\frac{\Gamma \vdash \mathsf{p} :: \_ \qquad \Gamma \vdash i_1 :<: [0..\mathsf{p}[\qquad \Gamma \vdash i_2 :<: \{y\colon \mathsf{integer} \mid y = \mathsf{p}\} \qquad \Gamma \vdash D[i_2]}{\Gamma \vdash \mathsf{scatter}\ i_1\ x\colon D[i_2] \dashv \Gamma, x\colon D[i_2 \div \mathsf{p}]}$$

$$\frac{\text{(see } \mathsf{scatter})}{\Gamma \vdash \mathsf{gather}\ i_1\ x\colon D[i_2] \dashv \Gamma, x\colon D[i_2 \div \mathsf{p}]}$$

$$\frac{\Gamma \vdash \mathsf{p} :: \_ \qquad \Gamma \vdash i_1 :<: [0..\mathsf{p}[\qquad \Gamma \vdash i_2 :<: [0..\mathsf{p}[\qquad \Gamma \vDash i_1 \neq i_2 \qquad \Gamma \vdash D}{\Gamma \vdash \mathsf{message}\ i_1\ i_2\ D \dashv \Gamma}$$

$$\frac{\Gamma_1 \vdash i_1 :<: \mathsf{integer} \qquad \Gamma_1 \vdash i_2 :<: \mathsf{integer} \qquad \Gamma_1, x\colon [i_1..i_2] \vdash T \dashv \Gamma_2}{\Gamma_1 \vdash \mathsf{foreach}\ x\colon i_1..i_2\ T \dashv \Gamma_1}$$

$$\frac{}{\Gamma \vdash \mathsf{skip} \dashv \Gamma} \qquad \frac{\Gamma_1 \vdash T_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash T_2 \dashv \Gamma_3}{\Gamma_1 \vdash T_1; T_2 \dashv \Gamma_3}$$

$$\frac{\Gamma \vdash T \dashv \Gamma}{\Gamma \vdash \mathsf{loop}\ T \dashv \Gamma} \qquad \frac{\Gamma_1 \vdash T_1 \dashv \Gamma_2 \qquad \Gamma_1 \vdash T_2 \dashv \Gamma_2}{\Gamma_1 \vdash \mathsf{choice}\ T_1\ \mathsf{or}\ T_2 \dashv \Gamma_2}$$

Datatype formation, $\Gamma \vdash D$, mode: `II`

$$\frac{}{\Gamma \vdash \mathsf{integer}} \qquad \frac{}{\Gamma \vdash \mathsf{float}} \qquad \frac{\Gamma \vdash D \quad \Gamma \vdash i :<: \mathsf{nat}}{\Gamma \vdash D[i]} \qquad \frac{\Gamma \vdash D \quad \Gamma, x\colon D \vdash p}{\Gamma \vdash \{x\colon D \mid p\}}$$

Proposition formation, $\Gamma \vdash p$, mode: `II`

$$\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1\ \mathsf{and}\ p_2} \qquad \frac{\Gamma \vdash i_1 :<: \mathsf{integer} \qquad \Gamma \vdash i_2 :<: \mathsf{integer}}{\Gamma \vdash i_1 \leq i_2}$$

Fig. 10. Type, datatype, proposition formation

Index kinding, $\Gamma \vdash i :: D$, mode: `II0`

$$\frac{\Gamma_1 \vdash D}{\Gamma_1, x\colon D, \Gamma_2 \vdash x :: D} \qquad \frac{}{\Gamma \vdash n :: \mathsf{integer}} \qquad \frac{\Gamma \vdash i_1 :<: \mathsf{integer} \qquad \Gamma \vdash i_2 :<: \mathsf{integer}}{\Gamma \vdash i_1 + i_2 :: \mathsf{integer}}$$

$$\frac{\Gamma \vdash i_1 :: D[i_3] \qquad \Gamma \vdash i_2 :<: [0..i_3[}{\Gamma \vdash i_1[i_2] :: D} \qquad \frac{\Gamma \vdash i_1 :: D[i_2]}{\Gamma \vdash \mathsf{length}(i_1) :: \mathsf{integer}}$$

Datatype subtyping, $\Gamma \vdash D <: D$, mode: `III`, rules tried in order

$$\frac{}{\Gamma \vdash \mathsf{integer} <: \mathsf{integer}} \qquad \frac{}{\Gamma \vdash \mathsf{float} <: \mathsf{float}} \qquad \frac{\Gamma \vDash 0 \le i_1 \le i_2 \qquad \Gamma \vdash D_1 <: D_2}{\Gamma \vdash D_1[i_1] <: D_2[i_2]}$$

$$\frac{\Gamma \vdash D_1 <: D_2 \qquad \Gamma, x\colon D_1 \vDash p}{\Gamma \vdash D_1 <: \{x\colon D_2 \mid p\}} \qquad \frac{\Gamma \vdash D_1 <: D_2}{\Gamma \vdash \{x\colon D_1 \mid p\} <: D_2}$$

Formulae entailment, $\Gamma \vDash p$, mode: `II`

$$\frac{\Gamma \vdash p \qquad \mathrm{forms}(\Gamma) \vDash p}{\Gamma \vDash p}$$

Abbreviation, $\Gamma \vdash i :<: D$, mode: `III`

$$\frac{\Gamma \vdash i :: D' \qquad \Gamma, x\colon \{y\colon D \mid y = i\} \vdash D' <: D \qquad x, y \text{ fresh}}{\Gamma \vdash i :<: D}$$

Formulae in a context, $\mathrm{forms}(\Gamma) = \{p\}$, mode: `I0`

$$\begin{aligned}
\mathrm{forms}(\cdot) &\triangleq \mathrm{forms}(x\colon \mathsf{integer}) \triangleq \mathrm{forms}(x\colon \mathsf{float}) \triangleq \{\mathsf{true}\} \\
\mathrm{forms}(x\colon \{y\colon D \mid p\}) &\triangleq \mathrm{forms}(x\colon D) \cup \{[y/x]p\} \\
\mathrm{forms}(\Gamma_1, \Gamma_2) &\triangleq \mathrm{forms}(\Gamma_1) \cup \mathrm{forms}(\Gamma_2) \\
\mathrm{forms}(x\colon D[i]) &\triangleq x.\mathsf{length} = i, 0 \le y < i \to (\mathrm{forms}(z\colon D) \wedge x[y] = z) \quad \text{with } y, z \text{ fresh}
\end{aligned}$$

Fig. 11. Index kinding, datatype subtyping, formulae entailment

Values

$$v \ ::= \ n \mid f \mid [\vec{v}]$$

Machine states

$$S \ ::= \ T_1, \ldots, T_n \qquad (n > 0)$$

Fig. 12. Runtime syntax

Value typing, $\vdash v\colon D$

$$\frac{}{\vdash n\colon \mathsf{integer}} \qquad \frac{}{\vdash f\colon \mathsf{float}} \qquad \frac{\vdash v_1\colon D \qquad \ldots \qquad \vdash v_n\colon D}{\vdash [v_1, \ldots, v_n]\colon D[n]}$$

State formation, $\Gamma \vdash S$

$$\frac{\Gamma \vdash T_1 \qquad \ldots \qquad \Gamma \vdash T_n}{\Gamma \vdash T_1, \ldots, T_n}$$

Fig. 13. Value and state typing

$$\mathsf{skip}; T \equiv T \qquad \mathsf{loop\ skip} \equiv \mathsf{skip} \qquad \mathsf{choice\ skip\ or\ skip} \equiv \mathsf{skip}$$

Fig. 14. Structural congruence

$$\frac{T_k = \mathsf{val}\ x\colon D; T'_k \qquad \vdash v\colon D \qquad T''_k = \vec{T}'_k[v/x]}{\vec{T} \to \vec{T}''}$$

$$\frac{|\vec{T}_1\vec{T}_2| = p - 1}{\vec{T}_1, (\mathsf{size}\ x\colon D; T), \vec{T}_2 \to \vec{T}_1, T[p/x], \vec{T}_2}$$

$$\frac{|\vec{T}_1| = j \qquad |\vec{T}_2| = k - j - 1 \qquad i_1 \downarrow j \qquad i_2 \downarrow k \qquad i_3 \downarrow j \qquad i_4 \downarrow k}{\vec{T}_1, (\mathsf{message}\ i_1\ i_2\ D; T_j), \vec{T}_2, (\mathsf{message}\ i_3\ i_4\ D; T_k), \vec{T}_3 \to \vec{T}_1, T_j, \vec{T}_2, T_k, \vec{T}_3}$$

$$\frac{|\vec{T}_1| = l \qquad i_1 \downarrow j \qquad i_2 \downarrow k \qquad j, k \neq l}{\vec{T}_1, (\mathsf{message}\ i_1\ i_2\ D; T), \vec{T}_2 \to \vec{T}_1, T, \vec{T}_2}$$

$$\frac{T_k = \mathsf{broadcast}\ i_k\ x\colon D; T'_k \quad i_k \downarrow n \quad 0 \leq n < |\vec{T}| \quad \vdash v\colon D \quad T''_k = T'_k[v/x]}{\vec{T} \to \vec{T}''}$$

$$\frac{T_k = \mathsf{reduce}\ i_k\ \mathsf{op}\ x\colon D; T'_k \quad i_k \downarrow n \quad 0 \leq n < |\vec{T}| \quad \vdash v_k\colon D \quad T''_k = T'_k[v_k/x]}{\vec{T} \to \vec{T}''}$$

$$\frac{T_k = \mathsf{scatter}\ i_k\ x\colon D[i'_k]; T'_k \quad i_k \downarrow n \quad 0 \leq n < |\vec{T}| \quad i'_k \downarrow |\vec{T}| \quad \vdash v_k\colon D[n \div |\vec{T}|] \quad T''_k = T'_k[v_k/x]}{\vec{T} \to \vec{T}''}$$

$$\frac{T_k = \mathsf{gather}\ i_k\ x\colon D[i'_k]; T''_k \quad i_k \downarrow n \quad 0 \leq n < |\vec{T}| \quad i'_k \downarrow |\vec{T}| \quad \vdash v_k\colon D[n \div |\vec{T}|] \quad T''_k = T'_k[v_k/x]}{\vec{T} \to \vec{T}''}$$

$$\frac{T_k = \mathsf{choice}\ T'_k\ \mathsf{or}\ T''_k; T'''_k}{\vec{T} \to \vec{T}'; \vec{T}'''} \qquad \frac{T_k = \mathsf{choice}\ T'_k\ \mathsf{or}\ T''_k; T''_k}{\vec{T} \to \vec{T}''; \vec{T}'''}$$

$$\frac{T_k = \mathsf{loop}\ T'_k; T''_k \qquad T'''_k = T'_k; \mathsf{loop}\ T'_k; T''_k}{\vec{T} \to \vec{T}'''} \qquad \frac{T_k = \mathsf{loop}\ T'_k; T''_k}{\vec{T} \to \vec{T}''}$$

$$\frac{i_1 \downarrow j \qquad i_2 \downarrow k \qquad j \leq k}{\vec{T}_1, (\mathsf{foreach}\ x\colon i_1\ i_2\ T; T'), \vec{T}_2 \to \vec{T}_1, (T[j/x]; \mathsf{foreach}\ x\colon i_1{+}1\ i_2\ T; T'), \vec{T}_2}$$

$$\frac{i_1 \downarrow j \qquad i_2 \downarrow k \qquad j > k}{\vec{T}_1, (\mathsf{foreach}\ x\colon i_1\ i_2\ T; T'), \vec{T}_2 \to \vec{T}_1, T', \vec{T}_2} \qquad \frac{T \equiv T'}{\vec{T}_1, T, \vec{T}_2 \to \vec{T}_1, T', \vec{T}_2}$$

Fig. 15. Operational semantics for machine states, $S \to S$

Types

$$\mathsf{allgather}\ x\colon D \triangleq \mathsf{foreach}\ y : 0..\mathsf{p}-1\ (\mathsf{gather}\ y\ x\colon D)$$

$$\mathsf{allreduce}\ \mathrm{op}\ x\colon D \triangleq \mathsf{reduce}\ 0\ \mathrm{op}\ \_\colon D'; \mathsf{broadcast}\ 0\ x\colon D \quad \text{where op}\colon \vec{D}' \to D$$

Type formation, $\Gamma \vdash T$

$$\frac{\Gamma \vdash \mathsf{p} :: \_ \quad \Gamma \vdash i :<: \{z\colon \mathsf{integer} \mid z\%\mathsf{p} = 0\} \quad \Gamma \vdash D[i]}{\Gamma \vdash \mathsf{allgather}\ x\colon D[i] \dashv \Gamma, x\colon D[i]} \qquad \frac{\Gamma \vdash D \quad \mathrm{op}\colon \vec{D}' \to D}{\Gamma \vdash \mathsf{allreduce}\ \mathrm{op}\ x\colon D \dashv \Gamma, x\colon D}$$

Reduction, $S \to S$

$$\frac{T_k = \mathsf{allreduce}\ \mathrm{op}\ x\colon D; T'_k \qquad \vdash v\colon D \qquad T''_k = T'_k[v/x]}{\vec{T} \to \vec{T}''}$$

$$\frac{T_k = \mathsf{allgather}\ x\colon D[i]; T'_k \qquad i \downarrow |\vec{T}| \qquad \vdash v\colon D[|\vec{T}|] \quad T''_k = T'_k[v/x]}{\vec{T} \to \vec{T}''}$$

Fig. 16. Derived constructs