Inferring Types for Parallel Programs

Francisco Martins
LaSIGE, Faculty of Sciences, University of Lisbon
Vasco Thudichum Vasconcelos
LaSIGE, Faculty of Sciences, University of Lisbon
Hans Hüttel
Aalborg Universitet

The Message Passing Interface (MPI) framework is widely used in implementing imperative programs that exhibit a high degree of parallelism. The PARTYPES approach proposes a behavioural type discipline for MPI-like programs in which a type describes the communication protocol followed by the entire program. Well-typed programs are guaranteed to be exempt from deadlocks. In this paper we describe a type inference algorithm for a subset of the original system; the algorithm allows to statically extract a type for an MPI program from its source code.

1 Introduction

Message Passing Interface (MPI) has become generally accepted as the standard for implementing massively parallel programs. An MPI program is composed of a fixed number of processes running in parallel, each of which bears a distinct identifier—a rank—and an independent memory. Process behaviour may depend on the value of the rank. Processes call MPI primitives in order to communicate. Different forms of communication are available to processes, including point-to-point message exchanges and collective operators such as broadcast.

Parallel programs use the primitives provided by MPI by issuing calls to a dedicated application program interface. As such the level of verification that can be performed at compile time is limited to that supported by the host language. Programs that compile flawlessly can easily stumble into different sorts of errors, that may or not be caught at runtime. Errors include processes that exchange data of unexpected types or lengths, and processes that enter deadlocked situations. The state of the art on the verification of MPI programs can only address this challenge partially: techniques based on runtime verification are as good as the data the programs are run with; strategies based on model checking are effective only in verifying programs with a very limited number of processes. We refer the reader to Gopalakrishnan et al. [2] for a discussion on the existing approaches to the verification of MPI programs.

PARTYPES is a type-based methodology for the analysis of C programs that use MPI primitives [7, 9]. Under this approach, a type describes the protocol to be followed by some program. Types include constructors for point-to-point messages, e.g. message from to float[], and constructors for collective operations, e.g. allreduce min integer. Types can be further composed via sequential composition and primitive recursion, an example being foreach i: 1..9 message 0 i. Datatypes describe values exchanged in messages and in collective operations, and include integer and float, as well support for arrays float[] and for refinement types that equip types with refinement conditions, an example being \{v:integer|v>0\}. Index-dependent types allow for protocols to depend on values exchanged in messages; an example of this is allreduce min x:{v:integer|1<=v<=9}.message 0 x. Our notion of refinement types is inspired by Xi and Pfenning [11], where datatypes are restricted by indices drawn from a decidable domain.
The idea of describing a protocol by means of a type is inspired by multiparty session types (MPST), introduced by Honda et al. [5]. MPST feature a notion of global types describing, from a all-inclusive point of view, the interactions all processes engage upon. A projection operation extracts from a global type the local type of each individual participant. Partypes departs from MPST in that it does not distinguish between local and global types. Instead the notion of types is equipped with a flexible equivalence relation. Projection can be recovered by type equivalence in the presence of knowledge about process ranks, e.g., \( \text{rank} : \{ x : \text{integer} | x = 2 \} \vdash \text{message} \ 0 1 \ \text{integer} \equiv \text{skip} \), where skip describes the empty interaction.

The type equivalence relation is at the basis of our strategy for type reconstruction:

- We analyse the source code for each individual process, extracting (inferring) for each process a type that governs that individual process;
- We then gradually merge the thus obtained types, while maintaining type equivalence.

This approach is related to that of Carbone and Montesi [1], where several choreographies are merged into a single choreography, and to the work of Lange and Scalas [6] where a global type is constructed from a collection of contracts.

Typable programs are assured to behave as prescribed by the type, exchanging messages and engaging in collective operations as detailed in the type. Moreover, programs that can be typed are assured to be deadlock free [7]. As such, programs that would otherwise deadlock cannot be typed, implying that the inference procedure will fail in such cases, rendering the program untypable.

2 The \( n \)-body pipeline and its type

We base our presentation on a classical problem on parallel programming. The \( n \)-body pipeline computes the trajectories of \( n \) bodies that influence each other through gravitational forces. The algorithm computes the forces between all pairs of bodies, applying a pipeline technique to distribute and balance the work on a parallel architecture. It then determines the bodies’ positions [4].

The program in Figure 1 implements this algorithm. Each body (henceforth called particle) is represented by a quadruple of \( \text{floats} \) consisting of a 3D position and a mass. The program starts by connecting to the MPI middleware (line 15), and then obtains the number of available processes and its own process number, which it stores in variables \( \text{size} \) and \( \text{rank} \) (lines 16–17). The overall idea of the program is as follows: (a) each process starts by obtaining a portion of the total number of particles, \( \text{MAX\_PARTICLES} \), and computes the trajectories (line 19). Then, (b) each process enters a loop that computes \( \text{NUM\_ITER} \) discrete steps. In each iteration (c) the algorithm computes the forces between all pairs of particles. It accomplishes this in two phases: (c.1) compute the forces among its own particles (lines 22–23), and (c.2) compute the forces between its particles and those from the neighbour processes (lines 25–36). Towards this end, each process passes particles to the right process and receives new particles from the left (lines 26–32). Then it compute the forces against the particles received (line 33–34). After \( \text{size} - 1 \) steps all processes have visited all particles. Then, (d) each process computes the position of its particles (line 37), which results in the computation of a local time differential (\( \text{dt\_local} \)), and (e) updates the simulation time (\( \text{sim\_t} \)).

The simulation time is incremented by the minimum of the local time differentials of all processes. In order to obtain this value, each process calls an \( \text{MPI\_Allreduce} \) operation (line 38). This collective operation takes the contribution of each individual process (\( \text{dt\_local} \)), computes its minimum (\( \text{MPI\_MIN} \)), and distributes it to all processes (\( \text{dt} \)). The minimum is then added to the simulation time (line 39). The program terminates by disconnecting from the MPI middleware (line 41).
inferring types for parallel programs

#define MAX_PARTICLES 10000
#define NUM_ITER 5000000

void InitParticles(float* part, float* vel, int npart);
float ComputeForces(float* part, float* other_part, float* vel, int npart);
float ComputeNewPos(float* part, float* pv, int npart, float);

int main(int argc, char** argv) {
    int rank, size, iter, pipe, i;
    float sim_t, dt, dt_local, max_f, max_f_seg;
    float particles[MAX_PARTICLES * 4]; /* Particles on all nodes */
    float pv[MAX_PARTICLES * 6]; /* Particle velocity */
    float send_parts[MAX_PARTICLES * 4], recv_parts[MAX_PARTICLES * 4]; /* Particles from other processes */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    InitParticles(particles, pv, MAX_PARTICLES / size);
    sim_t = 0.0f;
    for (iter = 1; iter <= NUM_ITER; iter++) {
        max_f_seg = ComputeForces(particles, particles, pv, MAX_PARTICLES / size);
        if (max_f_seg > max_f) max_f = max_f_seg;
        for (pipe = 0; pipe < size - 1; pipe++) {
            if (rank == 0) {
                MPI_Send(send_parts, MAX_PARTICLES / size * 4, MPI_FLOAT, rank == size - 1 ? 0 : rank + 1, ...);
                MPI_Recv(recv_parts, MAX_PARTICLES / size * 4, MPI_FLOAT, rank == 0 ? size - 1 : rank - 1, ...);
            } else {
                MPI_Recv(recv_parts, MAX_PARTICLES / size * 4, MPI_FLOAT, rank == 0 ? size - 1 : rank - 1, ...);
                MPI_Send(send_parts, MAX_PARTICLES / size * 4, MPI_FLOAT, rank == size - 1 ? 0 : rank + 1, ...);
            }
            max_f_seg = ComputeForces(particles, recv_parts, pv, MAX_PARTICLES / size);
            if (max_f_seg > max_f) max_f = max_f_seg;
            memcpy(send_parts, recv_parts, MAX_PARTICLES / size * 4);
        }
        dt_local = ComputeNewPos(particles, pv, MAX_PARTICLES / size, max_f);
        MPI_Allreduce(&dt, &dt_local, 1, MPI_FLOAT, MPI_MIN, ...);
        sim_t += dt;
    }
    MPI_Finalize();
    return 0;
}

Communication is performed on a ring communication topology. The conditional statement within
the loop (lines 26–32) breaks the communication circularity. Because operations MPI_Send and MPI_Recv
implement synchronous message passing, a completely symmetrical solution would lead to a deadlock
with all processes trying to send messages and no process ready to receive.

From this discussion it should be easy to see that the communication behaviour of 3-body pipeline
can be described by the protocol (or type) in Figure 2. The rest of this abstract describes a method to
infer the type in Figure 2 from the source code in Figure 1.

3 The problem of type inference

Given a parallel program P composed of n processes (or expressions) e0, . . . , en−1, we would like to find
a common type that types each process ei, or else to decide there is no such type. We assume that size
is the only free variable in processes, so that the typing context only needs an entry for this variable.
We are then interested in a context where size is equal to n, which we write as size: {x: int | x = n} and
abbreviate to Γn. Our type inference problem is then to find a type T such that Γn ⊢ ei : T, or else decide
that there is no such type.
F. Martins, V.T. Vasconcelos & H. Hüttel

foreach iter: 1..5000000
foreach pipe: 1..2
message 0 1 float[1000000 / 3 * 4];
message 1 2 float[1000000 / 3 * 4];
message 2 0 float[1000000 / 3 * 4]
allreduce min float

Figure 2: Protocol for the parallel n-body algorithm with three processes

We propose approaching the problem in two steps:
1. From the source code $e_i$ of each individual process extract a type $T_i$ such that $\Gamma^n \vdash e_i : T_i$;
2. From types $T_0, \ldots, T_{n-1}$ look for a type $T$ that is equal to all such types, that is, $\Gamma^n \vdash T_i \equiv T$.

Then, from these two results, we conclude that $\Gamma^n \vdash e_i : T$, hence that $\Gamma^n \vdash P : T$, as required.

We approach the first step in a fairly standard way:
- Given an expression $e_i$, collect a system of equations $\mathcal{D}_i$ over datatypes and a type $U_i$;
- Solve $\mathcal{D}_i$ to obtain a substitution $\sigma_i$. We then have $\Gamma^n \vdash e_i : U_i \sigma_i$, as required for the first phase. If there is no such substitution, then $e_i$ is not typable.

For this step we introduce variables over datatypes. Then we visit the syntax tree of each process and, guided by the typing rules [7], collect restrictions (in the form of a set of equations over datatypes) and a type for the expression. We need rules for expressions, index terms (the arithmetic in types), and propositions. We omit the rules for extracting a system of equations and a type from a given expression. Based on the works by Vazou et al. [10] and Rondon et al. [8], we expect the problem of solving a system of datatype equations to be decidable.

We address the second step in more detail. The goal is to build a type $T$ from types $T_0, \ldots, T_{n-1}$. We start by selecting some type $T_i$ and merge it with some other type $T_j$ (for $i \neq j$) to obtain a new type. The thus obtained type is then merged with another type $T_k$ ($k \neq j, i$), and so forth. The result of merging all the types is the sought type $T$. The original inference problem has no solution if one of the merge operations fail.

4 Merging types

We give an intuitive overview of the merge operation, discuss its rules and apply them to our running example. The intuition behind the merge operator is the following:
- messages must be matched exactly once by the sender and the receiver processes (the two endpoints of the communication);
- collective operations (allreduce, for example) establish horizontal synchronisation lines among all processes, meaning that all processes must perform all communications (collective or not) before the synchronisation line, carry out the collective operation, and then proceed with the remainder of the protocol.

Having this in mind, the merge rules make sure that collective operations match each other and that messages are paired together before and after each collective operation.

The merge operation receives a typing context $\Gamma$, the type merged so far $T$, the type to be merged $U$ and its rank $k$, to yield a new type $V$. We write all this as follows $\Gamma \vdash T \parallel_k U \leadsto V$. The typing context
names try to capture these concepts. For instance, rule skip-msgS merges a merge at rank \(i\) between ranks 3 and 2 with the message \(i_3 i_4 D\). Notice that a message \(skip \parallel k\) contains entries for variables size and rank, the latter recording the ranks whose types have been merged. This context will then be updated with new entries arising from collective (dependently typed) operations, such as allreduce. An excerpt of rules defining the merge operation is in Figure 3.

We first discuss merging \(skip\) and \(message\) types. There are ten different cases that we group into the five categories detailed below. Notice that a message \(i_1 i_2 D\) appearing as the left operand of a merge is equivalent to \(skip\) when both \(i_1\) and \(i_2\) are different from all ranks merged so far, which we write as \(i_1, i_2 \neq \text{rank}\). Otherwise, when \(i_1 = \text{rank}\) or \(i_2 = \text{rank}\), the message is the endpoint of a communication between ranks \(i_1\) and \(i_2\) that are already merged. When message \(i_3 i_4 D\) appears as the right operand of a merge at rank \(k\) it is equivalent to \(skip\) when both \(i_3\) or \(i_4\) are not \(k\), which we abbreviate as \(i_3, i_4 \neq k\). Otherwise, when \(i_3 = k\) or \(i_4 = k\), the message is the endpoint of a communication with rank \(k\). Rule names try to capture these concepts. For instance, rule skip-msgS merges \(skip\) (left operand) with a

\[
\begin{align*}
\Gamma \vdash \text{skip} \parallel_k \text{skip} \rightsquigarrow \text{skip} & \quad \text{(skip-skip)} \\
\Gamma \vdash \text{skip} \parallel k \text{skip} \rightsquigarrow \text{skip} & \quad \text{(skip-msgS)} \\
\Gamma \vdash i_1, i_4 \neq k \text{true} & \quad \text{(msgS-msgS)} \\
\Gamma \vdash \text{skip} \parallel_k \text{message} i_3 i_4 D \rightsquigarrow \text{skip} & \quad \text{(msg-msgS)} \\
\Gamma \vdash \text{message} i_1 i_2 D_1 \parallel_k \text{message} i_3 i_4 D_2 \rightsquigarrow \text{message} i_3 i_4 D_2; \text{message} i_1 i_2 D_1 & \quad \text{(msg-msg-eq)} \\
\Gamma \vdash \text{message} i_1 i_2 D_1 \parallel_k \text{message} i_3 i_4 D_2 \rightsquigarrow \text{message} i_3 i_4 D_2; \text{message} i_1 i_2 D_1 & \quad \text{(msg-msg-right)} \\
\Gamma \vdash \text{allreduce} x: D_1. T_1 \parallel_k \text{allreduce} x: D_2. T_2 \rightsquigarrow \text{allreduce} x: D_1. T_3 & \quad \text{(allred-allred)} \\
\Gamma \vdash \text{foreach} x: i_1..i_1'. T_1 \parallel_k \text{foreach} x: i_2..i_2'. T_2 \rightsquigarrow \text{foreach} x: i_1..i_1'. T_3 & \quad \text{(seq-seq)} \\
\Gamma \vdash \text{message} i_1 i_2 D_1; T_1 \parallel_k \text{message} i_3 i_4 D_2; T_2 \rightsquigarrow T_3 & \quad \text{(msg-msg-eq)} \\
\Gamma \vdash \text{message} i_1 i_2 D_1; T_1 \parallel_k \text{message} i_3 i_4 D_2; T_2 \rightsquigarrow \text{message} i_3 i_4 D_2; \text{message} i_1 i_2 D_1 & \quad \text{(msg-msg-right)} \\
\Gamma \vdash \text{skip} \parallel_k \text{message} i_3 i_4 D \rightsquigarrow T_2 & \quad \text{(skip-msgT)} \\
\Gamma \vdash \text{skip} \parallel_k \text{message} i_3 i_4 D \rightsquigarrow T_2 & \quad \text{(skip-msgT)} \\
\end{align*}
\]

Figure 3: Rules defining the merge partial function (excerpt)
message (right operand) that is semantically equivalent to skip, whereas rule skip-msg designates the merging of skip with a message that is not equivalent to skip. We proceed by analysing each category.

**merge yields skip.** In this case both operands are semantically equivalent to skip. This category comprises rules skip-skip, skip-msgS, msgS-skip (not shown), and msgS-msgS. We include the appropriate premises for enforcing that one or both parameters are equivalent to skip, depending on the message being the left or the right operand. For instance, rule skip-skip has no premises, while rule msgS-msgS includes two premises to make sure that both messages are equivalent to skip.

**merge yields the left operand.** In this category the left operand is not equivalent to skip, whereas the right operand is. It encompasses rules msg-skip and msg-msgS (not shown). Apart from the condition enforcing that the left message is not equivalent to skip ($i_1 = \text{rank} \lor i_2 = \text{rank}$), rank $k$ being merged must not be the source or the target of the message. Would this be the case and the program has a deadlock, since the messages on the left talk about rank $k$ (either as a source or a target) and the type at rank $k$ is skip (or equivalent to it), meaning that the merged messages will never be matched.

**merge yields the right operand.** In this case the left operand is semantically equivalent to skip, and the right operand is not. The category includes rules skip-msg and msgS-msg (not shown). The message is from or targeted at rank $k$ ($i_3 = k \lor i_4 = k$). We also need to check that the other rank of the message (the source or target that is different from $k$) is still to be merged ($i_3, i_4 \neq \text{rank}$). Why? Because otherwise the type of the other endpoint is already merged and is skip (the left operand), therefore the message at rank $k$ (the right operand, which is not skip) is never going to be matched, indicating the program has a deadlock.

**messages are the endpoints of the same communication.** In this category (rule msg-msg-eq) the messages correspond to the two endpoints of a communication. The result of the merge is the left operand, which is semantically equivalent to the right one. No message is semantically equivalent to skip as witnessed by the premises. Additionally we need to check that the source and the target ranks, as well as the payload, of the two messages coincide.

**messages are the endpoints of different communications.** This last category includes messages that are the endpoints of two different communications. The result of the merge is an interleaving of the messages. The messages are semantically different from skip and are unrelated. The category includes rules msg-msg-left (not shown) and msg-msg-right. As in the previous category we check that no message is semantically equivalent to skip. Additionally, we check that the messages do not interfere, that is, that their ranks are not related. These two rules can be non-deterministically applied in an appropriate way to match the types.

There are no rules to merge messages against collective operations, since this is not admissible; the merging of messages against foreach loops is left for future work. Collective operations can only be merged against each other (cf. rule allred-allred). We omit the rules for other MPI collective operations for they follow a similar schema. In this paper we only merge foreach loops against foreach loops. Refer to the next section for a discussion about the challenges on this subject.

The last three rules apply to the sequential composition of types: rule seq-seq allows for types to be split at the sequential operator (;) and merged separately; rules msgT-msgT-left and msgT-msgT-right (not shown) allow for the non-deterministic ordering of unrelated messages, as described for rules msg-msg-left and msg-msg-right, but here at the level of the sequential composition of types. The last rule allows for messages after the last collective communication (if any) to be merged. For the sake of brevity, we also omit rules for the sequential composition of skip types.
We now outline how merging works on our running example. Fix size = 3. From the program in Figure 1 extract size programs, one per rank, in such a way that programs do not mention variable rank. We leave this to the reader.

Run the first step of our procedure on each program to obtain the three types below, where D is the datatype float[MAX_PARTICLES / size * 4].

For rank 0:

```plaintext
foreach iter: 1..5000000
  foreach pipe: 1..2
    message 0 1 D;
    message 2 0 D
  allreduce min float
```

For rank 1:

```plaintext
foreach iter: 1..5000000
  foreach pipe: 1..2
    message 0 1 D;
    message 1 2 D
  allreduce min float
```

For rank 2:

```plaintext
foreach iter: 1..5000000
  foreach pipe: 1..2
    message 1 2 D;
    message 2 0 D
  allreduce min float
```

Run the second step as follows. We only show the merging of the various messages; the cases of `foreach` and `allreduce` are of simple application.

We start by taking the type for the process at rank 0 and merge it with that of rank 1. The initial typing context $\Delta_1$ says that the type on the left corresponds to rank 0 in a total of 3, which we write as size: \{$x : \text{int} \mid x = 3\}$, rank: \{$x : \text{int} \mid x = 0\}$. Using rules seq-seq, msg-msg-eq, and msg-msg-right we have:

```plaintext
\[ \Delta_1 \vdash \text{message 0 1 D; } \| \text{message 0 1 D; } \]
\[ \text{message 2 0 D } \| \text{message 1 2 D } \Leftrightarrow \text{message 1 2 D; } \]
\[ \text{message 2 0 D} \]
```

Then we merge the resulting type with that of rank 2. This time we need a typing context $\Delta_2$ that records the fact that the type on the left corresponds to ranks 0 and 1. We write it as size: \{$x : \text{int} \mid x = 3\}$, rank: \{$x : \text{int} \mid x = 0 \lor x = 1\}$. Using rules msgT-msgT-left, seq-seq, msg-msg-eq (x2), we get:

```plaintext
\[ \Delta_2 \vdash \text{message 0 1 D; } \| \text{message 1 2 D; } \]
\[ \text{message 1 2 D; } \| \text{message 2 0 D } \Leftrightarrow \text{message 1 2 D; } \]
\[ \text{message 2 0 D } \]
```

The type obtained is that of Figure 2.

5 Discussion

The procedure outlined in this paper is not complete with respect to the PARTYPES type system [7]. We discuss some of its shortcomings.

Variables in MPI primitives In order to increase legibility, code that sends messages to the left or to the right process in a ring topology often declares variables for the effect. The original source code [3] declares a variable `right` with value `rank == size - 1 ? 0 : rank + 1`. The MPI_Send operation in line 27 is then written as follows:

```plaintext
MPI_Send(sendbuf, MAX_PARTICLES / size * 4, MPI_FLOAT, right, ...);
```

In this particular case the value of `right` is computed from the two distinguished PARTYPES variables—size and rank—and it may not be too difficult to replace `right` by `rank == size - 1 ? 0 : rank + 1` in the type. In general, however, the value of variables such as `right` may be the result of arbitrarily
complex computations, thus complicating type inference in step one of our approach. In addition, indices present in types can only rely on variables whose value is guaranteed to be uniform across all processes. It may not be simple to decide whether an index falls in this category or not.

**Parametric types** The type in Figure 2 fixes the number of bodies in the simulation (line 1). The original source code, however, reads this value from the command line using `atoi(argv[1])`. The PARTYPES language includes a dependent product constructor `val` that allows to describe exactly this sort of behaviour:

```plaintext
val n: natural.
foreach iter: 1..500000
  foreach pipe: 1..2
    message 0 1 float[n / 3 * 4]
...
```

The PARTYPES verification procedure seeks the help of the user in order to link the value of expression `atoi(argv[1])` in the source code to variable `n` in the type `[7, 9]`. When we think of type inference, it may not be obvious how to resolve this connection during the first step of our proposal.

**Type inference and type equivalence** PARTYPES comes equipped with a rich type theory, allowing in particular to write the three messages in the protocol (Figure 2, lines 3–5) in a more compact form:

```plaintext
foreach i: 0..2
  message i (i == 2 ? 0 : i + 1) float[n / size * 4]
```

It is not clear how to compute the more common `foreach` protocol from the three messages, but this intensional type is not only more compact but also conducive of further generalisations of the procedure, as outlined in the next example.

**The number of processes is in general not fixed** A distinctive feature of PARTYPES—one that takes it apart from all other approaches to verify MPI-like code—is that verification does not depend on the number of processes. The approach proposed in this paper, however, requires a fixed number of processes, each running a different source code (all of which can nevertheless be obtained from a common source code, such as that in Figure 1). Then, the first step computes one type per process, and the second step merges all these types into a single type. The PARTYPES verification procedure allows to check the program in Figure 1 against a protocol for an arbitrary number of processes (greater than 1), where the internal loop (lines 2–5) can be written as

```plaintext
foreach pipe: 1..size-1
  foreach i: 0..size-1
    message i (i + 1 < size ? i + 1 : 0) float[n / size * 4]
```

The merge algorithm outlined in this paper crucially relies on a fixed number of types, one per process, and is not clear to us how to relieve this constraint.

**One-to-all loops** The type presented in the paragraph above contains two `foreach` loops: the former corresponds to an actual loop in the source code (lines 23–33), the latter to a conditional (lines 26–32). By expanding the source code in Figure 1 for each different process rank, the first step of our proposal extracts types of the same “shape” for all processes, as we have seen in Section 4. Now consider the following code snippet, where process 0 sends a message to all other processes:
if (rank == 0)
    for (i = 1; i < size; i++)
        MPI_Send(sendbuf, n / size * 4, MPI_FLOAT, i, ...);
else
    MPI_Recv(recvbuf, n / size * 4, MPI_FLOAT, 0, ...);

Fixing size == 3 as before, the first phase yields the following types:

**foreach** i: 1..2 **message** 0 i **float**[n * 4] for rank 0,
**message** 0 1 **float**[n * 4] for rank 1, and
**message** 0 2 **float**[n * 4] for rank 2.

leaving for phase two the difficult problem of merging one **foreach** type against a series of **message** types. When the limits of the **foreach** loop are constant, we can unfold it and merge the thus obtained sequence of messages as in Section[3], but this is, in general, not the case.

References


