Semantics and Implementation of a Generalized `forall` Statement for Parallel Languages.

Paul Dechering Leo Breebaart Frits Kuijlman Kees van Reeuwijk
Henk Sips
Delft University of Technology, The Netherlands
BoosterTeam@cp.tn.tudelft.nl

Abstract

In this paper we present a generalized forall statement for parallel languages. The forall statement occurs in many (data) parallel languages and specifies which computations can be performed independently. Many different definitions of such a construct can be found in literature, with different conditions and execution models. We will show how forall constructs of a wide class of parallel languages can be mapped to this generalized forall statement. In addition, the forall statement we propose has the ability to spawn more complex independent activities than can be found in these languages.

Denotational semantics are used to define the meaning of the forall and define only one possible program state change. It is shown that it is easy to use and that it is feasible to implement this forall efficiently.

1. Introduction

The *forall* statement is an important language construct in many (data) parallel languages [3], [4], [5], [9], [13], [17]. It specifies which computations can be performed independently. Although its necessity is widely accepted, the *forall* definition differs per language. The *forall* statement in each of the languages was designed with specific implementation criteria in mind.

We think it is important to have a clear and generalized semantics for *forall* statements in all languages in which they occur. This paper defines a generalized *forall* statement and discusses its semantics and implementation. We will show how *forall* constructs as found in the languages *Booster* [3], Connection Machine Fortran (CM Fortran) [5], and High Performance Fortran (HPF) [9] are mapped to this generalized *forall* statement without forfeiting semantics and efficiency. Furthermore, the *forall* statement we propose has the ability to spawn more complex independent activities than can be found in these languages. Having a single language construct that spawns a parallel loop increases the orthogonality of a language. It is our opinion that this *forall* statement is not only suited to an intermediate representation, but can also be adopted at the syntactic level in high-level parallel languages.

The context of our *forall* statement is supplied by *V-nus*, a concise intermediate language we have defined for data parallel programs [6]. The purpose of *V-nus* is providing a language platform to which other data parallel languages can be translated, and subsequently optimized. We use denotational semantics to define the meaning of the *V-nus* language constructs, which will allow us to verify and optimize *forall* statements.

Our goal is to find a *forall* statement that complies with the following requirements: (1) The denotational semantics of a *forall* statement must represent a deterministic outcome. (2) It must be possible to implement the *forall* statement efficiently. This means that the administration that is needed to execute the *forall* should not use excessive amounts of computational resources. (3) The *forall* statement must be capable of representing a wide class of *forall* definitions as can be found in (data) parallel languages. (4) It must be possible to give a concise operational semantics of the *forall* statement that can easily be understood.

2. Different types of *iteration*

In the set of *iteration* statements, we can identify two extremes: the sequential loop and the completely parallel loop.

The **sequential *iteration*** is equivalent to the conventional FOR-loop. The body-instances are executed one after another, in a predefined order. Data dependencies are of no consequence. In the **chaotic *iteration***, the body-instances are executed completely concurrently. All body-instances work on the same memory locations, and no assumptions are made about the order in which writes to and reads from these variables take place. A non-deterministic behaviour can be

a result of this model of execution.

Besides these extremes we present a number of other *iteration* statements.

In the **merge *iteration***, the body-instances are executed completely concurrently as well. But now, all body-instances work on their own copy of the program state, so determinism is guaranteed. At the end of the *iteration* statement all the now-changed individual program states of the body-instances must be merged back into a single parent program state by a merge function.

In the **statement-atomic *iteration***, the body-instances are executed concurrently, but the statements within the body are considered to be atomic. This means that during the execution of a statement S it is guaranteed that no other body-instances will be updating the value of any of the variables used in S. In the **body-atomic *iteration***, the entire body is considered to be atomic; i.e. during the execution of a body-instance i it is guaranteed that no other body-instances will be updating the value of any of the variables used in body-instance i.

These intermediate forms of *iteration* statements are called *forall* statements. Both the statement-atomic and the body-atomic *forall* statement imply a certain amount of synchronization and variable-shielding. We have chosen the merge *forall* in *V-nus*, because it has the most potential parallelism, and is well-suited for use in programming.

3. Existing approaches

Both data parallel languages as well as control parallel languages use the concept of a *forall* statement to denote the spawning of concurrent actions. There is a common trade off in the definitions of *forall* statements in these languages: constraints on the body decrease the potential parallelism, but lack of these constraints may cause non-determinism. An assignment in a specific body-instance may affect the computation of another body-instance, when these body-instances share the same variable. The outcome of a *forall* statement is then dependent on the order of computation. In general, it is impossible to know at compile time which data elements are assigned to. The solution for this problem is putting restrictions to *forall* statements to reduce undesirable behaviour. Function and procedure calls complicate the task of finding well-defined restrictions even more, since it is hard to analyse their effect on the program context in general.

One of the first versions of the *forall* statement was introduced by Thinking Machines Corporation in CM Fortran [5]. It is used to distribute computations over the processing elements of the Connection Machine (CM). The keyword FORALL indicates that the body-instances can be executed independently. The body-instances consist of one assignment with a left-hand side that is not assigned to by another body-instance. The use of certain kinds of expressions, such as user defined functions and assignments to array sections that depend on the index variable, always causes the *forall* statement to be executed serially.

Vienna Fortran [17] defines a broader *forall* statement by permitting private variables. These variables are known only in the *forall* statement in which they are declared, and each body-instance has a separate copy. A body-instance can consist of any legal FORTRAN 77 executable statement. Tightly nested *forall* statements can be used to specify multiple levels of parallelism. Vienna Fortran also restricts the *forall* body by requiring that a value written in one body-instance is neither read (define-use dependence) nor written (define-define dependence) in any other body-instance (see [18] for a description of define-use and define-define dependencies). The result is always deterministic.

Experiences with the *forall* statement in the Fortran dialects CM Fortran, Vienna Fortran, and Fortran D [10] led to the construction of the HPF *forall*. CM Fortran uses the *forall* statement to create parallelism explicitly by distributing body-instances over the CM. Vienna Fortran uses the *forall* statement to indicate that the different body-instances are independent and can be logically executed in parallel. In HPF [9] it is the distribution of data that introduces parallelism. The HPF *forall* statement consists of a single assignment statement. The left-hand side of each body-instance of this assignment can only be assigned to once. This excludes define-define dependencies. Execution of the *forall* statement requires the right-hand sides of the body-instances to be evaluated before these are assigned to the left-hand sides. This implies that a synchronization is needed. Only function calls to pure functions (functions that have no side effect) may be used in the right-hand side. It is then assured that define-use dependencies leave the outcome of the *forall* statement deterministic.

It is allowed to have multiple statements in the HPF *forall* body[1], but this means that each assignment of the body is executed completely; i.e. as if the assignments were written as *forall* statements in the same order (see Section 7). In addition a directive INDEPENDENT has been introduced for both DO loops and FORALL statements. The directive assures the compiler that the body-instances can be executed in an arbitrary order, without any computational differences in the result. In case of the multiple statement *forall* this means no synchronization is needed between the statements. Both the single assignment and the multiple assignment *forall* statement of HPF are used in the same form with the same semantics in Fortran 95, according to the proposed revision [8].

The data parallel language *Booster* [3] has no FORALL keyword. It is possible to assign array sections in parallel

[1] HPF distinguishes between *forall* statements and *forall* constructs; the latter may have multiple statements in their bodies.

by using an aggregate assignment. Unambiguous semantics are enforced by the requirement that no element is used as a target before it is used as a source. Function calls do not complicate analysis, since *Booster* requires the functions to be referentially transparent; i.e. no side effects occur and no global variables are accessed.

In the control parallel language SuperPascal [13] the *forall* statement is used to denote an array of parallel processes. A severe restriction is imposed on the *forall* body to prevent ambiguous computations: the body may not assign to a variable. This implies that a body-instance must output its results through a communication channel or a file. Procedure calls can be used in the body, which causes no problems under the given circumstances.

The *forall* statement in Compositional C^{++} [4], denoted by the keyword PARFOR, also initiates the parallel execution of the body-instances. Multiple statements are allowed in the *forall* body, where the statements of a specific body-instance are executed sequentially. This is in contrast to the multiple statement *forall* of HPF. No copies are made of data that is used in the body-instances, so loop carried dependencies can lead to non-deterministic results.

The Myrias PARALLEL DO uses a copy-in/copy-out semantics [2]. When a program executes a PARALLEL DO construct, parallel tasks are created, one for each iteration of the PARALLEL DO. Each task gets a separate copy of the parent program state. At the end of the PARALLEL DO all child program states are merged to form the new program state. It is, however, not explained how this merging can be done efficient.

Li and Wolfe [14] mention the difficulties in defining well-behaved parallel constructs without making arbitrary decisions. They have developed a framework for analyzing the behaviour and relations of various sequential and parallel control constructs. Their DOPAR iteration has a similar meaning as the merge *forall*described in Section 2, and is based on the PARALLEL DO of the Myrias system. Here too, it is not mentioned how to implement this general iteration construct efficiently. Using their framework they present how and when different loop constructs can be substituted by another loop construct.

In the remainder of this paper we will use the *forall* statements of *Booster*, CM Fortran, and HPF as representatives of the many *forall* definitions that can be found in literature on data parallel languages.

4. The semantics of the *V-nus forall*

Similar to the other languages, the *V-nus forall* statement is represented by the syntax: `forall` *IndexSpace Body*. The term *IndexSpace* specifies the range of the index variable; the term *Body* represents the block of statements that will be executed for each value of the index variable (see Example 4.1).

Example 4.1 *The V-nus* forall *statement*
Consider: `forall [i:3] {a := i}`. The index variable is `i` and ranges over $0, 1$ and 2. The body is `a := i;` an example of a body-instance is `a := 1`. □

Body-instances of the *V-nus forall* statement are to be executed completely independently. By this we mean that data that can be changed by a body-instance i will not affect the computation of another body-instance j. However, a global interference is still possible when there is a define-define dependence between the possible body-instances; i.e. two body-instances that write to the same variable. We say that *a forall statement is deterministic if no define-define dependence is present between any two different body-instances of the forall statement.*

We want to record the concept of the *forall* statement in a semantic model, such that we can use this model to reason about a program. We use denotational semantics [1] [16], in which the meaning of a program can be expressed by the composition of the meanings of its parts. The denotational semantics are useful when we want to rewrite only parts of a program, and leave the meaning of the whole program as it is.

In denotational semantics a program state captures all necessary information about the context in which a program fragment is executed. A program state is valid only if each variable of the program state is given exactly one value (see Example 4.2).

Example 4.2 *Program states.*
Consider the *forall* statement of Example 4.1. A valid program state after execution of the body-instance `a := 1` is: $(\mathtt{a} = 1)$. The program state $(\mathtt{a} = 0, \mathtt{a} = 1)$ is invalid, because the variable `a` is given two values. □

The semantics of a program fragment are given by a program state change, represented by a pair (ps, ps') of program states. In case of the *forall* statement, program state changes are computed for all body-instances. Say, for body-instance i the state change (ps, ps_i) is computed. Then the different program states ps_i (for all i) are merged into the final program state ps', which will be the program state after the *forall* statement has been executed. This merge operation consists of two actions. First ps_i is compared with ps, providing only the difference $diff_i$ between these program states. Secondly, all elements of $diff_i$ will be put into ps. This is done for all ps_i in arbitrary order.

The mathematical framework for the denotational semantics of *V-nus* (including the *forall* statement) is described in [6].

5. Mathematical model

In this section we will show how the *forall* statement of *V-nus* can be expressed by a semantic function. It is therefore necessary to introduce some mathematical concepts.

In our model, functions are just special sets. A (partial) function f from a set X to a set Y is a set $f \subseteq X \times Y$ such that

$$\forall x \in X\ \forall y, y' \in Y.\ ((x,y) \in f \wedge (x,y') \in f) \Rightarrow y = y'$$

For a function $f \subseteq X \times Y$ we will also write $f : X \to Y$.

Functions are used to represent the state of the variables of a program. When another value is assigned to an existing variable x, the function representing the state of x needs to be updated. For this purpose we introduce a replacement function which will change a pair or add a pair to the set of pairs defining a function. Let $P = \mathbb{P}(X \times Y)$ be the powerset of the Cartesian product of X and Y. For two functions $f, g : X \to Y$ the replacement function $\triangleleft : P \times P \to P$ is defined as

$$\begin{array}{rcl} f \triangleleft \emptyset & = & f \\ \forall (x,y) \in g.\ f \triangleleft g & = & f' \triangleleft g\backslash_{(x,y)} \end{array}$$
$$\text{where} \quad f'(z) = \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}$$

We have constructed a set Σ of all possible program states σ for an arbitrary *V-nus* program. A program state σ is a function that assigns a value to a program variable. Such a program state is defined for all variables in a given *V-nus* program, except for the variables defined in an index-space specification. These index variables play an important role in specifying the meaning of an *iteration*. A separate set Φ of index states φ has been constructed that represent the current values of the index variables in a loop nest. The meaning of some *V-nus* program fragment is then given by the meaning function

$$\mathcal{M} : \mathbf{Vnus} \to \Phi \to \Sigma \to \Sigma$$

So given a *V-nus* fragment c, an index state φ, and a program state σ, the program state σ' after c has been executed is represented by $\sigma' = \mathcal{M}(c)(\varphi)(\sigma)$.

Now we can present the semantic function $\mathcal{PL}$ that defines the state change of a parallel loop.

Definition 5.1 Let $(j_0, \ldots, j_n)$ be a predefined permutation of $(0, \ldots, n)$. The function $\mathcal{PL} : \mathbf{Vnus} \to \mathbb{P}(\Phi) \to \Phi \to \Sigma \to \Sigma$ is defined as follows:

$$\mathcal{PL}(Body)(\{f_0, \ldots, f_n\})(\varphi)(\sigma) = \sigma'$$
$$\text{where} \begin{cases} \forall i \in \{0, \ldots, n\}.\ \varphi_i & = & \varphi \triangleleft f_i \\ \forall i \in \{0, \ldots, n\}.\ \sigma_i & = & \mathcal{M}(Body)(\varphi_i)(\sigma) \\ \forall i \in \{0, \ldots, n\}.\ \sigma'_i & = & \sigma_i \backslash \sigma \\ \sigma' & = & \sigma \triangleleft \sigma'_{j_0} \triangleleft \ldots \triangleleft \sigma'_{j_n} \end{cases}$$

□

The argument *Body* represents the loop body of the *forall* statement. The second argument is a set of functions, each defining an element of the index space. For instance, the index space specification `[i:2,j:3]` would be represented by the set of functions: $\{f_0 = \{(i,0),(j,0)\}, f_1 = \{(i,0),(j,1)\}, \ldots, f_5 = \{(i,1),(j,2)\}\}$. The third and fourth argument represent the current index state and program state respectively. The result is the program state after execution of the body instances.

The index states φ_i all have a different index value for the index variables of the loop. The program state σ_i represents the meaning of the loop body *Statements* executed in the index state φ_i and the original program state σ. Then the differences between these final program states and the original program state can be computed, which is represented by σ'_i. The last step in the computation merges all differences into the final program state σ'.

The meaning of the program construct `forall` *IndexSpace Body* is given by the following definition of the meaning function $\mathcal{M}$:

$$\begin{array}{l} \mathcal{M}(\mathsf{forall}\ IndexSpace\ Body)(\varphi)(\sigma) = \\ \qquad \mathcal{PL}(Body)(F)(\varphi)(\sigma) \\ \text{where} \quad F = \mathcal{DP}(IndexSpace)(\varphi)(\sigma) \end{array}$$

The domain propagation function $\mathcal{DP}$ computes an index state for each element of the index space. It is out of the scope of this paper to define this in more detail. For a complete definition of the semantic functions we refer to [6].

6. The implementation

Implementing the *forall* statement as presented in Section 4 and 5 may cause problems. Merging the different program states of the body-instances is inefficient, since computing the difference between program states is time consuming.

To arrive at an efficient implementation of the *forall* statement, we take the following approach. At the start of a *forall* statement the program state ps is preserved. For the execution of a body-instance a subset qs_i of ps is used for the context in which this body-instance will be executed. Only the data that is needed in the body-instance is extracted from ps and will be used for qs_i. Each time something must be read from memory, it is read from qs_i. When something must be written to memory, it is not only stored in qs_i, but also in ps. In this way, each change that is made by a single body-instance is also visible in the global program state, but will not affect the other body-instances. This is how the final program state ps' arises from the original program state ps, without the need for a merge or a difference operation (see Figure 1).

The construction of qs_i is dependent on the information the compiler has about the data that is used in the body-instance. This information can be generated automatically by standard dependence analysis techniques and manually by pragmas. A pragma is an annotation for the compiler that

gives additional information about a certain program construct. Pragmas that can be used for a *forall* statement specify which data should be copied in qs_i.

If a *forall* statement is not annotated by a pragma, then the local program states qs_i are created as explained above. If a pragma is present the compiler relies on this information and only copies the given data structures for the accompanying program states qs_i. In our opinion, it is more useful to specify for which data structures a dependency exists, than those for which no dependency exists. The syntax of a pragma for a *forall* statement is: `<<dependsOn Expressions>>` which expresses a dependency for the data structure(s) *Expressions*. An empty list of specifications (i.e. `<<dependsOn []>>`) means that no data needs to be copied. Of course, it is the responsibility of the programmer to avoid the introduction of non-determinism due to a pragma.

Especially when the compiler can not determine at compile-time what dependencies exist between the body-instances, it is useful to be able to give additional information to the compiler. In Example 6.1 is shown how the efficiency of a *forall* statement can be optimized by introducing a pragma.

Example 6.1 *Using pragmas for a forall statement.*
Consider the program fragment:

```
forall [i:n] {A[i]:=B[C[i]]; B[C[i]]:=A[i+1];}
```

At compile-time it is unknown what elements of `B` are referenced. The conservative approach is taken so that this *forall* is characterized as non-deterministic. Furthermore, for each body-instance qs_i a complete copy of `B` is created. If all elements of `C` are different then each body-instance will write to a different element of `B`. In that case, there is no need to create a copy of `B` in each qs_i. Note that for `A` it is necessary to create a local copy. So we can safely annotate the *forall* statement as follows:

```
<< dependsOn [A[i+1]] >> forall [i:n] {
   A[i] := B[C[i]]; B[C[i]] := A[i+1]; }
```

which means that each body-instance qs_i must have a copy of `A[i+1]`, and no other copies are needed. When a pragma is used, it is assumed that the *forall* is deterministic. □

When using pragmas the execution model is slightly changed. Each time something must be read from memory, it is read from qs_i if it exists in qs_i; otherwise it is read from ps. Proper use of pragmas still guarantees determinism provided the original program was deterministic.

In the implementation of a deterministic *forall* statement, all differences between the program states qs_i are collected in the global program state ps'. This is exactly as it is described by the denotational semantics.

The denotational semantics use the same computation for both deterministic and non-deterministic *forall* statements. That makes the result of a non-deterministic *forall* statement dependent on the computation order. In this case the efficient implementation of a *forall* statement may compute other results than the theory prescribes. In Example 6.2 a possible difference is presented between the computation used in the implementation, and the computation used in the semantics.

Example 6.2 *Difference between theory and implementation.*
Consider the program fragment: `forall [i:2] {a := i; b := i}`. The denotational semantics predict that the body-instance for $\mathsf{i} = 0$ will result in the program state $ps_0 = (\mathsf{a} = 0, \mathsf{b} = 0)$. The body-instance for $\mathsf{i} = 1$ will result in the program state $ps_1 = (\mathsf{a} = 1, \mathsf{b} = 1)$. ps' will then be either ps_0 or ps_1.
The implementation, on the other hand, may cause the following execution orders:

```
a := 0, b := 0, a := 1, b := 1   or
                 ⋮
a := 0, a := 1, b := 1, b := 0   or
a := 1, a := 0, b := 0, b := 1
```

which will lead to the same possible program states as predicted by the theory, *plus* the program states $(\mathsf{a} = 0, \mathsf{b} = 1)$ and $(\mathsf{a} = 1, \mathsf{b} = 0)$. □

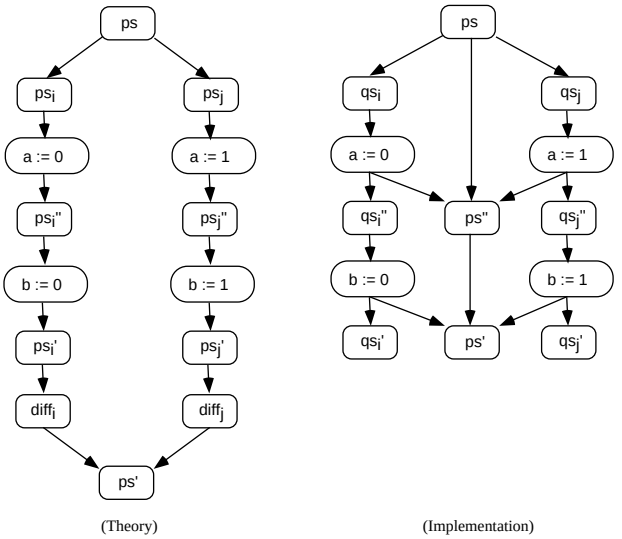

Figure 1. Program state changes caused by a *forall* statement.

In Example 6.2 both the body-instances write to the variables a and b, which makes the *forall* statement non-deterministic. Theory and implementation only differ for non-deterministic *forall* statements. We want to use a semantic model in which the outcome of a program (fragment) is unambiguous. When non-determinism is forced by a non-deterministic *forall* statement it is sufficient to mention that the outcome is unpredictable. For now, there is no need for a semantic function that defines the set of all possible outcomes.

7. The *forall* compared

As shown in Section 3, many languages have a notation that describes some independent iteration over an index space. However, the semantics of these constructs differ for

each language. In this section, we compare the *forall* statements of the data parallel languages *Booster*, CM Fortran, and HPF, and we show how these differently defined *forall* statements can be mapped to the *V-nus forall* statement.

CM Fortran as well as HPF use the same method for the evaluation of `forall` *IndexSpace Body*: first, evaluate the expressions in *IndexSpace*, then, evaluate all expressions present in *Body*, and finally, perform the assignments of *Body*. More detailed descriptions are given in the appropriate language specifications.

Consider the following examples in pseudo code:

(7.1) `forall i=0,n j=0,m a[i,j]=`$expr$ `end`

(7.2) `forall i=0,n j=0,m a[i,j]=F(X) end`

(7.3) `forall i=0,n j=0,m a[i,j]=`$expr$`, a[i+1,j]=F(X) end`

where the expressions `n` and `m` are not dependent on each other, $expr$ is some arbitrary expression that does not contain a function call, `F` represents a function, and `X` is an actual argument list that is not dependent on the array `a`. In each of the languages *Booster*, CM Fortran, and HPF the index space over which is iterated is the Cartesian product $[0 \ldots n] \times [0 \ldots m]$.

In CM Fortran, Example 7.1 will cause the assignments to be executed on the CM in parallel. The assignments of Example 7.2 will be executed sequentially because of the function call on the right hand side. Example 7.3 is not valid since CM Fortran allows only one statement in a *forall* body.

In *Booster*, both Example 7.1 and Example 7.2 will perform the assignments in arbitrary order. Because in *Booster* functions are referentially transparent, the function call causes no side effects, and therefore it is guaranteed that each element is used as a source before it is used as a target. In *Booster* too, only one assignment is allowed in the *forall* body, which makes Example 7.3 invalid.

In HPF, Examples 7.1 and 7.2 have the same meaning as in *Booster*. Although pure functions in HPF need not be referentially transparent, it is forbidden for those functions to have side effects. This allows the different body instances of a *forall* statement to be evaluated in arbitrary order. Example 7.3 is semantically equivalent to the following consecutive *forall* statements:

```
forall i=0,n j=0,m a[i,j] = expr,
forall i=0,n j=0,m a[i+1,j] = F(X)
```

Note that the second *forall* statement only starts when the first *forall* statement has finished. It can not be rewritten to one `INDEPENDENT DO` loop, because a define-define dependence exists for `a[i]`, $1 \leq \mathsf{i} \leq \mathsf{n} - 1$.

Example 7.1 interpreted in *Booster*, CM Fortran, or HPF can be represented in *V-nus* by:

```
forall [i:n+1, j:m+1] a[i,j] := expr
```

Example 7.2 interpreted in CM Fortran needs a sequential loop in *V-nus*, such as:

```
for [i:n+1, j:m+1] a[i,j] := F(X)
```

In *Booster* and HPF this example can be represented in the same way as Example 7.1 is represented. Example 7.3 interpreted in HPF can be rewritten to two single assignment *forall* statements as presented above. These can easily be translated to *V-nus*. Note that if Example 7.3 was interpreted in *V-nus* directly, it would denote a non-deterministic *forall* statement because of the define-define dependencies. Define-define dependencies are allowed if they occur in the same body-instance. For example, if the subscript `i+1` of Example 7.3 is replaced by `i` then the *forall* statement has become deterministic.

Every `INDEPENDENT DO` loop in HPF can be represented by the *V-nus forall* statement, since no loop carried dependencies occur at all. Due to *V-nus* pragmas the effectuality of the `INDEPENDENT` directive can also be utilized.

The `NEW` directive in HPF is used to create variables that are local to a single body-instance. In *V-nus* it is possible to use loop-bodies as scope-boundaries. So, the named variables in the `NEW` directive of HPF can be represented in *V-nus* by locally declared variables in a loop.

Since *V-nus* requires functions to be referential transparent, functions of other languages that are less restrictive need to be rewritten in *V-nus*. If a non-*V-nus* function uses (or writes to) a global variable, it can be represented by a corresponding *V-nus* function where this global variable is passed via another function parameter (and consequently becomes local to the function). As a result, an HPF *forall* statement with a call in its body to a pure function that uses a global variable can be represented in *V-nus* while fully preserving the semantics and effectiveness.

Now, we show an example of an optimization that can only be expressed by using the *V-nus forall*. Consider the following matrix operation:

```
for [j:m] forall [i:n]
   a[i,j]:=a[i,j-1]+a[i,j+1]+a[i-1,j]+a[i+1,j]
```

The optimization we have in mind is based on synchronization elimination [12]. By reversing the `i` and `j` loop the operation can be expressed as

```
forall [i:n] for [j:m]
   a[i,j]:=a[i,j-1]+a[i,j+1]+a[i-1,j]+a[i+1,j]
```

which has no computational differences in the result. Instead of executing *forall* statements in sequence, the *forall* body-instances can now be executed concurrently, yet obeying the `j` sequence. It is easy to see that no define-define dependence occurs, which makes it a deterministic *forall* statement. This *forall* statement is not 'valid' in the other parallel languages mentioned in this paper.

8. Conclusion

For non-deterministic *forall* statements an unambiguous program state change is forced by the specification of a computation order. The program state change of a deterministic *forall* statement is not dependent on the computation order.

The approach taken in the implementation requires some computation overhead compared to a sequential loop. This overhead is due to the following computations: (1) Before the body-instances can be executed, each body-instance must get its own (small subset of the) program state. (2) During execution of a body-instance, each write action is performed twice (to update the local *and* global program state). In many cases, one of these two write actions can be omitted. Computation and space overhead can be adjusted by pragmas. The computation time for the construction of the program state ps' is in the order of the number of variables that are used in the *forall* body. A direct implementation of the theoretical scheme would need linear time in the number of variables of the entire program and the number of body-instances of the *forall* statement.

V-nus can be used to capture the meaning of different definitions of *forall* statements. Therefore, we think that our *forall* definition is suitable for an intermediate representation. Furthermore, it allows the spawning of more complex concurrent computations than can be found in other data parallel languages. The semantics is easy to understand and is unambiguous.

However, the programmer must be able to verify whether the condition for determinism is met. Partially, this can be done at compile-time. A run-time solution for the other cases requires too much overhead in general. But while using execution trace techniques it is possible to recognize a define-define dependence, when *different* values are written to the same variable. When the *same* value is written twice to that variable a define-define dependence is not recognized, but nevertheless the result is deterministic.

More *forall* examples are available at: `ftp://ftp.cp.tn.tudelft.nl`.

References

[1] J.W. de Bakker. *Mathematical Theory of Program Correctness*. Series in Comp. Sc. Prentice Hall Intl, 1980.

[2] M. Beltrametti et al. The Control Mechanism for the Myrias Parallel Computer System. *Computer Architecture News*, 16(4):21–30, 1988.

[3] L.C. Breebaart et al. The Booster Language, Syntax and Static Semantics. Comp. Phys. report series CP–95–02, Delft Univ. of Technology, 1995.

[4] P. Carlin et al. The Compositional C++ Language Definition. Revision 0.9 `ftp://ftp.compbio.caltech.edu /pub/CC++/Docs/cc++-def`, March 1 1993.

[5] Thinking Machines Corporation. CM Fortran Programming Guide. Technical report, January 1991.

[6] P.F.G. Dechering. The Denotational Semantics of Booster, A Working Paper 2.0. Comp. Phys. report series CP–95–05, Delft Univ. of Technology, 1995.

[7] P.F.G. Dechering et al. V-cal: a Calculus for the Compilation of Data Parallel Languages. In C.-H. Huang et al, editors, *LCPC*, vol 1033 of *LNCS*, pp 111–125, USA, 1995. Springer Verlag.

[8] Fortran Forum. Special Issue, Fortran95, Committee Draft, May 95. *Fortran Forum*, 12(2), 1995.

[9] High Performance Fortran Forum. High Performance Fortran Language Specification. Techn. report, Nov. 1994.

[10] G. Fox et al. Fortran D Language Specification. COMP TR90079, Dep. of Comp. Sc., Rice University, March 1991.

[11] A. Geist et al. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Network Parallel Computing*. Scientific and Eng. Comp. series. MIT Press, 1994.

[12] A.J.C. van Gemund. *Performance Modelling of Parallel Systems*. PhD thesis, Delft Univ. of Technology, 1996.

[13] P.B. Hansen. Interference Control in SuperPascal – A Block-Structured Parallel Language. *The Computer Journal*, 37(5):399–406, 1994.

[14] J. Li and M. Wolfe. Defining, Analizing and Transforming Program Constructs. *IEEE Par. and Distr. Technology*, pp 32–39, 1994.

[15] J.A. Trescher et al. A Formal Approach to the Compilation of Data Parallel Languages. In K. Pingali et al, editors, *LCPC*, vol 892 of *LNCS*, pp 155–169, USA, 1994. Springer Verlag.

[16] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Comp. Series. MIT Press, 1993.

[17] H. Zima et al. Vienna Fortran – A Language Specification, version 1.1. Internal Report 21, ICASE, 1992.

[18] H. Zima and B. Chapman. *Supercomputers for Parallel and Vector Computers*. Frontier Series. Addison-Wesley, 1990.