DO and FORALL: Temporal and Spatial Control Structures

Min-You Wu
Wei Shu

CRPC-TR91189
November, 1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

SCCS-139
CRPC-TR91189

DO and FORALL: Temporal and Spatial Control Structures

Min-You Wu and Wei Shu
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260

Geoffrey C. Fox
Syracuse Center for Computational Science
Syracuse University
111 College Place
Syracuse, NY 13244-4100

Abstract — Control structures provided in programming languages relate program statements to one another and are influenced by the underlying hardware machine architecture. Due to the rapid progress of parallel computing, the development of new control structures for the latest generation of parallel machines is going to be an issue of great interest. This paper describes *do* and *forall*, two major control structures in temporal and spatial domains, respectively. We examine different control structures during the transitional period from a traditional temporal structure to a newly introduced spatial structure. The general format and semantics of the temporal and spatial control structures are described, as well as the nested hierarchical structures. A survey of currently existing implementations of *forall* statements is then followed.

1. Introduction

In programming languages, control abstraction mechanisms are heavily influenced by the underlying machine hardware architecture. Most programming languages were developed several decades ago when machine architectures were dominated by single processor schemes. These kinds of computers can execute only a single statement at a time. The execution of program statements is restricted to a temporal axis. Hence, people spend most of their efforts on design and implementation of control structures to specify the *order* in which statements are to be executed. Figure 1 shows an example. Typical repetition structures include *do* in Fortran, *for* and *while* in Pascal and C, etc. We call them a *do–family.*

```
do i = 1, 4
  a(i) = b(i-1)
  b(i) = a(i)
end do
```

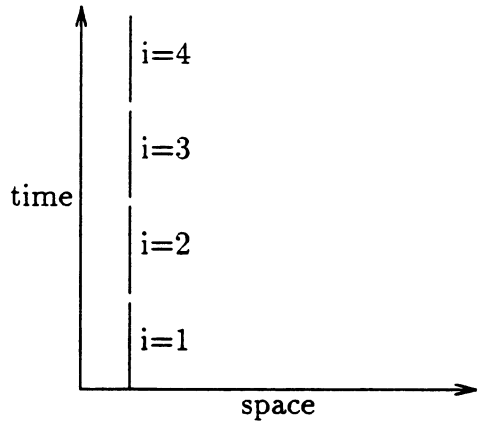

Figure 1: *do*

As the multiprocessor architectures were gradually developed, the execution of programs shows a more complicated picture. At this stage, more than one program statement can be processed at the same time. The program flow must then be defined at both temporal and spatial axes. It brought out a demand for new control structures in programming languages. Since the *do*–family control structures have been well defined in most programming

languages and their usage has been widespread, introducing a family of conceptually new control structures and rewriting numerous existing programs certainly was not a task that could be accomplished in a short time. Instead, people conducted a tremendous amount of work to utilize the *do*–family control structures, to make restructured sequential programs survive on new parallel architecture machines.

One major step of this task is to embed a *do* loop originally written for temporal repetition in the spatial axis. With dependency analysis, some sequential loops may be found to be parallelizable. The loop without dependency between iterations can be restructured into a *doall* loop [1]. An important feature of the *doall* loop is that no information is exchanged between the iterations of the loop. Every iteration can be executed simultaneously without synchronization. The *do* statement in Figure 2 is one that can be transferred into a *doall* statement.

```
do i = 1, 4
  a(i) = b(i)
  b(i) = c(i)
end do
```

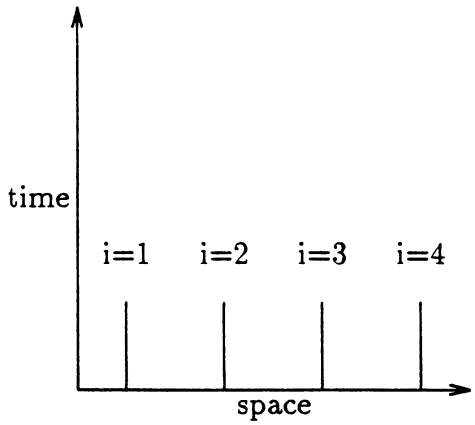

Figure 2: *doall*

Other loops may have dependencies but some iterations can be executed in parallel. They can be restructured into *doacross* loops [2]. In a *doacross* loop data dependencies allow for partial overlap of successive iterations during execution. An iteration may have to wait during its execution for a previous iteration in order to satisfy dependencies. The user or compiler must insert some synchronization primitives, such as *await*, to preserve

ordering of he loop for correct execution sequence. In Figure 3, the *do* statement cannot be simply converted into a *doall* statement. Instead, it can be embedded in the spatial axis with some temporal constrains, forming a *doacross* statement.

```
do i = 1, 4
  a(i) = a(i-1)
  b(i) = a(i)
end do
```

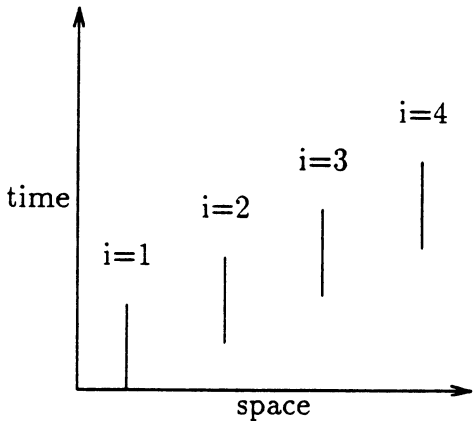

Figure 3: *doacross*

Although *doall* and *doacross* paved the way to restructuring the *do*–family structures in the existing sequential programs for parallel machine utilization, we can hardly expect them to become an ultimate methodology to solve control abstraction on parallel machines. *Doall* assumes every iteration is independent, however, many loops have dependencies in it. *Doacross* allows dependencies among iterations, but does not provide a clear picture of the temporal-spatial structure.

The development of new control structures for the latest generation of parallel machines becomes of great interest. That is, in addition to the well-established temporal control structures in *do*–family, another family of control structures needs to be introduced for representation in spatial domains, which can be defined as *forall–family.* The potential usefulness of spatial control structures relies in part on the successful experience with CM Fortran [3]. There are at least two reasons to consider parallelism in space as a language design issue. First and foremost, it provides a new method of expression. A program is written to simulate the real world, and the universe exists in space and time. Many problem

domains lend themselves naturally to parallelism. Therefore, languages should provide the framework for creating problem solutions that use the concept of parallelism, regardless of how the program will actually be executed. The second reason to discuss parallelism is that parallel machines are now becoming readily available, thus creating the need for languages that allow users access to that hardware capability. Such kind of spatial control structures are attractive to the user, as they allow a cleaner and shorter representation of application problems. Users can reliably understand what parallelism will be exploited. These control structures improve the readability and writability of a programming language. They are also helpful to the compiler, as the use of spatial control structures makes it easier and more efficient to exploit parallelism as well as to improve performance.

The IVTRAN language might be the early one to provide a spatial control structure, named as *do for all* statement [4, 5]. Here is an example:

```
      do label for all (i,j) / 1..N1 .across. 1..N2
        statements
label continue
```

All instances in the *do for all* can be executed in parallel. The body of the *do for all* constructs can be a block of statements, conditional statements, and *do* loops. Subroutine and function calls can appear within the body of the *do for all.* However, the *do for all* cannot be nested, and only limited conditional statements can be used. The array subscripts in a *do for all* are constrained to the simple linear expression of the indices.

The *forall* construct, proposed for Fortran 8X [6], is similar to the *do for all.* The *forall* construct was adopted in CM Fortran [3]. The body of a *forall* statement must be a single assignment statement. No nested *forall* statements, conditional statements, do loops, or subroutine calls are allowed in the *forall* body. Only some intrinsic functions can appear in the *forall* body. The *forall* defined in Fortran D [7] allows multiple statements, but not

dependencies between these statements.

We discuss the following problems in this paper:

- How can a temporal-spatial structure be constructed with *do* and *forall*?
- Which definition of the *forall* statement can maximize the expressive power?

In Section 2, we present the temporal-spatial structure and introduce the temporal synchronization separator. In Section 3, the control structure hierarchy is described. Comparison of different *forall* statements is presented in Section 4, followed by a discussion of open problems in Section 5.

2. The Temporal and Spatial Representation

In a traditional programming language, the control structures are emphasized in the order of execution, that is, in the temporal logic of programs. For a modern language design, in addition to the temporal axes, we need to expand control structures to the spatial axis as well, since the underlying machine architecture is no longer only a single processor.

For the underlying machine architecture, we use an idealized parallel computer known as the PRAM (parallel random-access machine) [8]. There are a number of processors working synchronously and communicating through the common random-access memory, forming a space region in the computer system. Now, lets consider the time-space aspects from a more fundamental, or perhaps philosophical, point of view. The space in a problem can be directly mapped into the space in a computer. Then as in nature, each processor evolves different elements in problem space. In this case, we have a rather clean association:

$$\text{Space in Problem} \rightarrow \text{Space in PRAM}$$
$$\text{Time in Problem} \rightarrow \text{Time in PRAM}$$

This association is particularly precise in the case when each processor holds a single element in problem space. In the normal case, where each processor holds and evolves a number of elements, then we have an intermediate situation. That is, we have partially mapped the spatial extent of the problem into a temporal extent in the computer implementation:

$$\text{Space in Problem} \rightarrow \text{Space and Time in PRAM}$$
$$\text{Time in Problem} \rightarrow \text{Time in PRAM}$$

Here we see the space in a problem can be mapped into the time in a PRAM. However, on the other hand, the time in a problem cannot be mapped into the space in a PRAM [9].

Next, we will briefly describe specifications of a language with both temporal and spatial control structures. We start with a set of elementary statements, denoted as $\mathcal{S}$, which includes most single statements, such as the assignment statement, the conditional statement, etc. Before discussing a temporal-spatial structure, we construct a temporal structure and a spatial structure. We need to define a temporal separator and a spatial separator to group statements together on the temporal axis and on the spatial axis, respectively. The temporal separator and spatial separator are denoted by ',' and '||', respectively.

Temporal Separator ','

Sequential composition of statements S_1 and S_2 is expressed by using the comma as a separator:

$$S_1 \ , \ S_2$$

where $S_1, S_2 \in \mathcal{S}$. As a convention, this separator clearly specifies the execution flow of two statements. That is, the execution of S_1 must be ahead of S_2 in a temporal axis. In some languages, such as Fortran, sequential composition is expressed by line separation, rather than by an explicit symbol, such as a comma.

We could group finite number of statements together by using the temporal separator ',' to specify a *temporal compound statement.* The syntax format of temporal compound statement $\mathcal{S}_{TC}$ is given as follows:

$$\begin{aligned} \mathcal{S}_{TC} \quad \rightarrow \quad & S \\ & \mid \; S \;\; ',' \;\; \mathcal{S}_{TC} \end{aligned}$$

where $S \in \mathcal{S}$.

Spatial Separator '||'

Compositing statements on the spatial axis can take the form:

$$S_1 \parallel S_2$$

where $S_1, S_2 \in \mathcal{S}$. The spatial separator allows the statements S_1 and S_2 to execute in parallel. Considering a spatial axis, the two statements are at different spatial points.

We could also group a finite number of statements together by using the spatial separator '||' to specify a *spatial compound statement.* The syntax format of a spatial compound statement $\mathcal{S}_{SC}$ is given as follows:

$$\begin{aligned} \mathcal{S}_{SC} \quad \rightarrow \quad & S \\ & \mid \; S \;\; '\parallel' \;\; \mathcal{S}_{SC} \end{aligned}$$

where $S \in \mathcal{S}$. Spatial compound statements are equivalent to the **cobegin ...coend** construct provided by other languages [10].

Notice that there is an essential difference between $\mathcal{S}_{TC}$ and $\mathcal{S}_{SC}$. Exchanging any two statements in $\mathcal{S}_{SC}$ does not affect its semantic meaning, whereas the exchange in $\mathcal{S}_{TC}$ can produce a totally different effect. That difference is due to the natural characteristics of the temporal logic and the spatial logic.

Now we can extend the elementary statement set to include both $\mathcal{S}_{TC}$ and $\mathcal{S}_{SC}$. In this way, we can integrate statements in any combination of temporal and spatial constructs.

The nested temporal-spatial statements shown below are simple combinations of temporal and spatial structures:

$$S_a = \{\{S_1 \| S_2\}, \{S_3 \| S_4\}\}$$

$$S_b = \{\{S_1, S_3\} \| \{S_2, S_4\}\}$$

Statement S_a specifies a temporal composition of two spatial structures in which neither S_3 nor S_4 can be executed until both S_1 and S_2 complete their execution. However, it does not specify the relation between different spatial points. When the user wants S_1 and S_3 to be executed in the same spatial point because of some storage preference, the statement alignment can be used. This will be discussed later.

Statement S_b specifies a spatial composition of two temporal structures in which S_2 and S_4 can be executed in parallel with S_1 and S_3, yet there is no interaction between S_1, S_3 and S_2, S_4. However, in many circumstances we need to express not only parallel actions at different spatial points, but also interactions between them. In this time-space system, a temporal separator ',' only defines the sequence of two events at the same spatial point. The relative time in the time-space system can only be defined by interactions between different spatial points. If two points do not interact, it is not necessary that they be synchronized because of the lack of synchronization would be not observable.

A new temporal separator is defined to specify a synchronization between spatial points at certain temporal points. We called the separator a *temporal synchronization separator.*

Temporal Synchronization separator ';'

Two statements can be composited in the form

$$S_1 \; ; \; S_2$$

where $S_1, S_2 \in \mathcal{S}$. The structure of temporal compound statements is expanded into

$$\begin{array}{rcl} \mathcal{S}_{TC} & \rightarrow & S \\ & | & S \ ',' \ S_{TC} \\ & | & S \ ';' \ S_{TC} \end{array}$$

where $S \in \mathcal{S}$. Here, the symbol ';' is a delimiter that acts logically like a time stamp. The nth ';' stands for the nth time step. All executions at different spatial points must synchronize at each time step. This is similar to a barrier [11, 12]. Semantically, every ',' only defines the execution order inside of S_{TC}, while every ';' defines the execution sequence between different spatial points. To clarify this, suppose we have two compound statements, S_{TC_a} and S_{TC_b}, which are composited by a spatial separator '||', both including one temporal synchronization separator ';'. For convenience, we call all statements before ';' in S_{TC_a} as $\mathcal{A}_1$, and the rest in S_{TC_a} as $\mathcal{A}_2$. Similarly, we call all statements before ';' in S_{TC_b} as $\mathcal{B}_1$, and the rest in S_{TC_b} as $\mathcal{B}_2$. Thus, execution of any statement in $\mathcal{A}_2$ has to wait for not only completion of all statements in $\mathcal{A}_1$ but also all statements in $\mathcal{B}_1$. Some rules of using the temporal synchronization separator will be discussed later in this section.

Temporal Repetition Statement: Do

Temporal repetition control structures provided in most languages allow us to specify looping over a finite set of statements. The *do*family constructs are typical temporal repetition statements.

$$\begin{array}{rcl} S_{TR} & \rightarrow & \textit{'do'} \ \textit{DoTemplet} \\ & & \quad S \\ & & \textit{'enddo'} \\ & | & \textit{'do'} \ \textit{DoTemplet} \\ & & \quad S \ \textit{';'} \\ & & \textit{'enddo'} \end{array}$$

where $S \in \mathcal{S}$. By default, S_{TR} is equivalent to a temporal compound statement separated by commas. If we need to use temporal synchronization separators, the statement S must be explicitly followed by a ';'.

Spatial Repetition Statement: Forall

The *forall* statement is a spatial repetition statement:

$$S_{SR} \rightarrow \begin{array}{l} 'forall' \ ForallTemplet \\ S \\ 'endforall' \end{array}$$

where $S \in \mathcal{S}$. It is equivalent to a spatial compound statement separated by '||'.

The elementary statement set is extended to include S_{TR} and S_{SR}. With the compound and repetition statements available for both temporal and spatial domains, we can easily express various computation patterns on the time-space. The following examples show some combinations that form simple temporal-spatial structures. The hierarchical temporal-spatial structure will be discussed in the next section.

(1) Spatial compound statement in the *do* loop:

```
do i = 1, N
  a(i) = ... || b(i) = ...
enddo
```

(2) Temporal compound statement in the *forall* statement:

```
     forall (i = 1 : N)
s1:    a(i) = ... ,
s2:    b(i) = ... ;
s3:    c(i) = b(i-1) + ...
     endforall
```

In this example, $s2$ can be executed even if every $s1$ at the other spatial points has not finished its execution. However, $s3$ must wait for completion of all $s2$ statements. Thus, the statements in a *forall* body can be partially synchronous, allowing multiple statements with dependencies.

(3) *Forall* statement in the *do* loop:

```
do i = 1 : N
  forall (j = 1 : N)
    a(i,j) = ...
  endforall
enddo
```

(4) *Do* loop in the *forall* statement:

```
forall (i = 1 : N)
  do j = 1, N
    a(i,j) = ... ;
  end do
endforall
```

In this example, each iteration of the *do* loop is synchronous. If the temporal synchronization separator ';' were not presented, execution at each spatial point would be independent.

Rules of Applying the Temporal Synchronization Separator ";"

When some dependencies between the statements at different spatial points exist in a *forall* body, a temporal synchronization separator ";" must be used to ensure the right sequence of execution. The dependencies can be classified into data dependencies, storage dependencies, and control dependencies.

a) Data dependency:

The following example shows a data dependency between two statements:

```
forall (i = 1:N)
  x(i) = ... ;
  ...  = x(i+1)
endforall
```

b) Storage dependencies:

There are two types of storage dependencies: anti-dependency and output dependency, shown by the following examples:

```
forall (i = 1:N)
  ...  = x(i+1);
  x(i) = ...
endforall

forall (i = 1:N)
  x(i+1) = ... ;
  x(i) = ...
endforall
```

c) Control dependency:

In the following example, execution of a statement depends on a condition:

```
forall (i = 1:N)
  tmp(i) = x(i+1);
  if (tmp(i) == 0) then
    x(i) = ...
  end if
endforall
```

When any one of the above dependencies exists, the user must insert a temporal synchronization separator. It also means that the user has control of eliminating unnecessary synchronizations. In other words, the user should use a temporal synchronization separator if, and only if, such a dependency exists, to avoid unnecessary synchronization. Furthermore, the number of synchronization separators can be reduced in some overlapping dependencies. For example, in the following *forall* statement, there is an anti-dependency between $s1$ and $s3$ and a data dependency between $s2$ and $s3$. Instead of using two syn-

chronization separators after $s1$ and $s2$, we can use only one synchronization separator after $s2$:

```
      forall (i = 1:N)
s1:     ...  = x(i+1)
s2:     y(i) = ... ;
s3:     x(i) = y(i-1)
      endforall
```

Many-to-One Operation: Reduction

There are three types of operations in a spatial domain: many-to-many, one-to-many, and many-to-one. (The one-to-one operation is a sequential operation.) The many-to-many operation is a parallel operation and can be constructed by a *forall* statement. The following *forall* statement is an example:

```
forall (i=1:N-1)
  a(i) = a(i+1)
endforall
```

The one-to-many operation — the *broadcasting* operation — can also be carried out with a *forall* statement without any special operator, such as:

```
forall (i=1:N-1)
  a(i) = x
endforall
```

and as a *multicasting* operation:

```
forall (i=1:N-1)
  a(i) = a(i/c)
endforall
```

On the other hand, special operators are required for many-to-one operations — the *reduction* operations. As an example, the following *forall* statement provides a *sum* reduction over $a(i)$ and assigns the result to a scalar variable, with '+=' as a *sum* reduction operator:

```
forall (i=1:N)
  x += a(i)
endforall
```

The following example is a *multiple reduction* operation:

```
forall (i=1:N)
  b(i/c) += a(i)
endforall
```

Table 1 lists some possible reduction operators.

Table 1: The Reduction Operators

+=	Sum of values
*=	Product of values
&=	Logical AND
|=	Logical OR
^=	Logical XOR
<? =	Minimum of values
>? =	Maximum of values

A reduction operator can also be used as a unary operator, as shown in the following example:

```
x = (+= a(1:N))
```

Using the reduction operators as unary operators, we can provide a special type of the reduction operation — the *scan* function. The following statement is an example for *scan* with *sum*:

```
forall (i=1:N)
  b(i) = (+= a(1:i))
endforall
```

Using reduction operators but reduction intrinsics allows reduction operations in the *forall* body.

3. Control Structure Hierarchy

The temporal and spatial constructs can be nested in any combination. That is, the temporal constructs can be nested to form a nested temporal structure. Similarly, the spatial constructs can be nested to form a nested spatial structure. As production rules are applied one by one, we rewrite syntax-correct nesting of temporal and spatial constructs to obtain a single compound statement. Every time we apply $S \rightarrow S_{TC}$ or $S \rightarrow S_{TR}$, we consider it as a temporal reduction level, $\mathcal{L}^T$; then when we apply $S \rightarrow S_{SC}$ or $S \rightarrow S_{SR}$, we consider it as a spatial reduction level, $\mathcal{L}^S$. Thus, for any reduced single statement S, we can find out a sequence of reduction levels, $\mathcal{L}_0\mathcal{L}_1...\mathcal{L}_k$, where $\mathcal{L}_0$ is the outmost level. This sequence describes the multiple level nests of temporal and spatial constructs. In this sequence, there could be some consecutive reduction levels that are the same type of $\mathcal{L}^T$ (or $\mathcal{L}^S$), which form a time complex, $\mathcal{C}^T$ (or a space complex $\mathcal{C}^S$):

$$\mathcal{C}^T = \mathcal{L}^{T+}/\{\mathcal{L}^S | \setminus n\}$$

$$\mathcal{C}^S = \mathcal{L}^{S+}/\{\mathcal{L}^T | \setminus n\}$$

As a result, we obtain a sequence of $\mathcal{C}^T$ and $\mathcal{C}^S$ ordered alternatively, called as temporal-spatial complex sequence $\mathcal{C}$, starting with either $\mathcal{C}^T$ or $\mathcal{C}^S$:

$$\mathcal{C} = \{\mathcal{C}^S|\epsilon\}\{\mathcal{C}^T\mathcal{C}^S\}^* \mid \{\mathcal{C}^T|\epsilon\}\{\mathcal{C}^S\mathcal{C}^T\}^*$$

A hierarchical structure can be formed with nested temporal and spatial structures, as shown below:

spatial
temporal
spatial
temporal
...

We call the number of complexes in a sequence the *rank*. If the rank is smaller than or equal to two, we then call it the *first-oder* time-space, otherwise it will be the *high-order* time-space. The traditional languages, such as Fortran77, C, and Pascal, have only the temporal constructs. In these kind of languages, the temporal separator ',' and the temporal synchronization separator ';' have the same meaning, since no any spatial construct presented.

The languages with the array features present one type of *first-order* temporal-spatial structure, $\mathcal{C}^T\mathcal{C}^S$, since the array operation is a flat spatial construct. The languages with the single-statement *forall*, such as CM Fortran, are the same since no temporal construct is allowed in the *forall* body. Fortran D does allow multiple statements but not dependency between these statements in the *forall* body. That is, there is no temporal synchronization separator ';' between the multiple statements and no temporal structure allowed in the *forall* body.

By introducing the temporal synchronization separator, we allow the other type of *first-order* temporal-spatial structure, $\mathcal{C}^S\mathcal{C}^T$, and the *high-order* temporal-spatial structure.

The following example shows a three-level temporal structure in a spatial structure:

```
forall (i = 1 : N)
  do j = 1, N
    a(i,j) = ... ;
    b(i,j) = ... ;
  enddo
  c(i) = ...
endforall
```

The following example shows a one-level temporal structure in a two-level spatial structure:

```
forall (i = 1 : N)
  forall (j = 1 : N)
    a(i,j) = ... ;
    b(i,j) = ...
  endforall
endforall
```

which is equivalent to:

```
forall (i = 1 : N, j = 1 : N)
  a(i,j) = ... ;
  b(i,j) = ...
endforall
```

As with the temporal synchronization separator in complex C_k^T, its scope is applied to its left spatial complex C_{k-1}^S, if exists. It will not synchronize any parallel action beyond the scope. Consider the following example:

```
      forall (i = 1 : N)
        do j = 1, N
          a(i,j) = ... ;
          forall (k = 1 : N)
s1:         b(i,j,k) = ... ;
            c(i,j,k) = ...
          endforall;
        enddo;
        d(i)= ...
      endforall
```

In this example, the temporal synchronization separator in *s1* only synchronizes the inner

forall statement but not the outer one.

4. Study of Different Forall Statements

There exist different syntax and semantic definitions of the *forall* statement, from the single-statement *forall* to the multiple-statement *forall*, and the most tightly synchronous *forall* to the most loosely synchronous *forall*. We may classify them into three types of *forall* statements: the *single-statement forall*, the *multiple-statement tightly-synchronous forall*, and the *multiple-statement loosely-synchronous forall*. The single-statement *forall* has only one statement in the *forall* body, so there is no any inter-statement dependency and no need of synchronization. However, the user may be forced to write many single-statement *forall*s with complex headers. A typical single-statement *forall* has been implemented in CM Fortran [3]. The following code written in CM Fortran for a search tree algorithm illustrates the use of the single-statement *forall*:

```
      pp = tree
      mask = .FALSE.
      mask = (pp(1:len) .NE. NIL) .AND. (key(1:len) .NE. k(pp(1:len)))
      do while (ANY(mask))
        forall (i=1:len, mask(i) .AND. (key(i) .LT. k(pp(i))))
     &     q1(i) = l(pp(i))
        forall (i=1:len, mask(i) .AND. (key(i) .GE. k(pp(i))))
     &     q1(i) = r(pp(i))
        forall (i=1:len, mask(i))
     &     pp(i) = q1(i)
        mask = (pp(1:len) .NE. NIL) .AND. (key(1:len) .NE. k(pp(1:len)))
      end do
```

Note that array $q1$ is necessary for buffering the intermediate values in the first *forall* to avoid incorrect results in the second *forall*. From this small example, the header is complicated. For a large application, the header could be extremely tedious to manage and

a multiple statement *forall* becomes necessary. The above example can be rewritten with the multiple statement *forall* as follows, eliminating repeated headers:

```
pp = tree
mask = .FALSE.
mask = (pp(1:len) .NE. NIL) .AND. (key(1:len) .NE. k(pp(1:len)))
do while (ANY(mask))
  forall (i=1:len)
    if (mask(i)) then
      if ((key(i) .LT. k(pp(i)))) then
        pp(i) = l(pp(i))
      else
        pp(i) = r(pp(i))
      end if
    end if
  endforall
  mask = (pp(1:len) .NE. NIL) .AND. (key(1:len) .NE. k(pp(1:len)))
end do
```

Next, we will discuss different semantics for the multiple-statement *forall*. The tightly synchronous *forall* synchronizes at each statement. For each statement, the entire *rhs* is completely evaluated before any stores take place. It is equivalent to many single-statement *foralls*. This type of *forall* often introduces extra synchronization points. For example, the following tightly synchronous *forall* statement must make many synchronizations between statements and between the *rhs* and the *lhs*:

```
forall (i = 1:N)
  x(i) = ...
  ...  = x(i)
  ...  = x(i+1)
endforall
```

With a temporal synchronization separator provided, only one synchronization is necessary:

```
forall (i = 1:N)
  x(i) = ... ;
  ...  = x(i),
  ...  = x(i+1)
endforall
```

The loosely synchronous *forall* synchronizes only at the end of the *forall* statement. It is equivalent to the *forall* with a temporal separator "," at the end of each statement. However, if there is a inter-statement *data* dependency in the *forall* body, we must split it into two *foralls*. For example, the above code must be written as two *forall* statements:

```
forall (i = 1:N)
  x(i) = ...
endforall

forall (i = 1:N)
  ...  = x(i)
  ...  = x(i+1)
endforall
```

Next, we consider the multiple statement *forall* with both the temporal separator "," and the temporal synchronization separator ";". There are several reasons for introducing this *forall*:

a) **Expressive power.** Compared to the single-statement *forall*, the multiple-statement *forall* is powerful in expressing real application problems. The temporal-spatial structures can be clearly described by the *forall* statement with the temporal synchronization separator, giving the user a flexible control of synchronization.

b) **Readability and Writability.** One of the most important criteria for judging a programming language is the ease with which programs can be read and understood. Most of the language characteristics that affect readability also affect writability. The degree

of abstraction allowed by a programming language and the naturalness of its expression are therefore very important to its writability. The multiple statement *forall* increases readability and writability drastically. If only single-statement *forall*s are permitted, the user may have to write many *forall* statements with repeated headers, which are also hard to understand. Similarly, for the loosely synchronous *forall*, the user has to break one *forall* into many smaller *forall*s to achieve synchronizations. However, with the temporal synchronization separator, the user can indicate where a synchronization should happen. Also, with the temporal synchronization separator, users can quickly identify existing dependencies. Furthermore, the number of temporal synchronization separators used may be an indication of how well a parallel program was written.

c) **Synchronizations.** One may suggest a smart compiler that can recognize the dependency between statements and add the synchronization automatically. It might be extremely difficult to identify all dependencies. In the case that the compiler cannot determine whether there is a dependency, it must assume so and add a synchronization for safety, which results in unnecessary synchronizations.

d) **Efficiency.** For the efficiency issue, we consider the cost of compiling programs as well as the cost of executing programs. Executing many *forall* statements with repeated headers is inefficient. Furthermore, when there are many unnecessary synchronizations, as mentioned above, communication costs may incur high overhead.

5. Discussion

There are some issues remaining to be studied. One of these is alignment. We have provided synchronization between two spatial constructs so that the two constructs can be aligned with the same temporal axis. At times, *code alignment* may need to be provided between temporal constructs so that the two constructs can be aligned with the same spatial

axis. This is different from the data alignment. The data alignment gives the relationship between different data arrays, but the code alignment establishes a relationship between different computations. The code alignment can take the form of a compiler directive.

Another problem remaining to be solved is the synchronization within a single statement. As an example, the following statement has dependencies between different spatial points:

```
forall (i = 1 : N)
  x(i) = x(i-1)
endforall
```

Without synchronization between the *rls* and the *lhs*, the execution result becomes undeterministic. This statement can be split in two with a temporary array as shown below:

```
forall (i = 1 : N)
  tmp(i) = x(i-1);
  x(i) = tmp(i)
endforall
```

The other solution is to define a *synchronous statement.* In a synchronous statement, all computations in the *rhs* will be completely evaluated before any store in the *lhs* takes place.

Next, we discuss implementation of control structures, especially for temporal synchronization separators. The temporal synchronization separator can be implemented with barriers on shared memory machines. The barrier is a strong implementation of the temporal synchronization separator, and it may be more severe than is actually necessary. It may be relaxed by using more focused schemes of synchronization, such as full/empty bit. For a distributed memory machine, the temporal synchronization separator can be implemented with message-passing barriers involving some kinds of global communication, such

as reduction or broadcasting. A simple reduction or broadcasting cannot guarantee that there is no overlap between the execution of the codes before and after the barrier. Instead, we should use a combination of a reduction and a broadcasting to meet the requirement of a message-passing barrier. Another method is to use the *combine* function [13]. Similar to shared memory systems, the implementation of temporal synchronization separators could also be relaxed. More specifically, when computation is static, that is, the dependencies are known at the compiler-time, temporal synchronization separators can be developed with focused message-passing instead of a barrier. However, if the computation is dynamic and a data-request scheme must be used, a strong barrier implementation is necessary. An example of this implementation is the inspector-executor pair [14, 15].

We have described two major control structures, *do*-family and the *forall*-family. The general format and semantics of the temporal and spatial control structures, and the hierarchical structures have been discussed. We will integrate these structures to our Fortran90D language and develop a compiler to compile them for distributed memory machines. With such a compiler available, we will be able to test various application problems written in Fortran90D and evaluate design issues of the language constructs.

Acknowledgments

The authors thank Diane Purser for her editorial efforts. The generous support of the Center for Research on Parallel Computation is gratefully acknowledged. This work was supported in part by the National Science Foundation under Grant No. CCR9109114 and Cooperative Agreement No. CCR-8809165 – the Government has certain rights in this material.

References

[1] M. D. Guzzi, D. A. Padua, J. P. Hoeflinger, and D. H. Lawrie. Cedar Fortran and other vector and parallel Fortran dialects. In *Supercomputing '88*, pages 114–121, November 1988.

[2] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua. Execution of parallel loops on parallel multiprocessor systems. In *Int'l Conf. on Parallel Processing*, August 1986.

[3] E. Albert, J.D. Lukas, and G.L. Steele. Data parallel computers and the FORALL statement. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Computation*, pages 390–396, College Park, Maryland, October 1990.

[4] R. Millstein. Control structure in Illiac IV Fortran. *Communications of ACM*, 16(10):621–627, October 1973.

[5] Massachusetts Computer Associates (COMPASS), Wakefield, Massachusetts. *IVTRAN Manual, TR CADD-7501-2811*, January 1975.

[6] Michael Metcalf and John Reid. *Fortran 8x Explained.* Clarendon Press, Oxford, 1986.

[7] G.C. Fox, S. Hiranadani, K. Kennedy, C. Koelbel, U. Kremer, C.W. Tseng, and M.Y. Wu. Fortran D language specifications. Technical Report COMP TR90-141, Rice University, December 1990.

[8] A. Gibbons and W. Rytter, editors. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[9] G.C. Fox. *Domain Decomposition in Distributed and Shared Memory Environments - I: A Uniform Decomposition and Performance Analysis for the NCUBE and JPL*

Mark IIIfp Hypercube. volume 297 of *Lecture Notes in Computer Science*, pages 1042–1073. Springer-Verlag, New York, 1987. Supercomputing, ed. E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos.

[10] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of ACM*, 8(5):569, September 1965.

[11] A. K. Jordan. A special purpose architecture for finite element analysis. In *Proc. of Int. Conf. on Parallel Processing*, pages 263–266, 1978.

[12] G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Comp., Inc., 1991.

[13] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice-Hall, 1988.

[14] J. Saltz, R. Mirchandaney, and K. Crowley. Runtime parallelization and scheduling of loops. In *Proceedings of the 1st Symposium on Parallel Algorithms and Architectures*, 1989.

[15] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, April 1990.