WORKING TITLE: A Comparison of Co-Arrays and Fusion Objects as Constructs for Parallel Scientific Programs

Terry Clark
twclark@cs.uchicago.edu

Ernesto Gomez
ernesto@cs.uchicago.edu

L. Ridgway Scott
ridg@cs.uchicago.edu

Department of Computer Science
The University of Chicago
Chicago, IL 60637
USA

January 14, 2000

Abstract

Parallelization of scientific applications remains a nontrivial task typically requiring considerable attention by the programmer to low-level details concerned with data movement and synchronization. In this paper, we describe and compare *co-arrays* and *fusion objects*, data movement primitives integrated into the high-level programming languages, Co-Array Fortran and the Planguages, respectively. Co-Array Fortran has developed from shared memory programming, whereas the Planguages have been developed from distributed memory programming; the semantics for the languages reflect those historical roots. *Fusion objects* imply a synchronization which we call *give&go semantics*. We use the property of fusion to show a restricted type of Planguage program is deterministic. It turns out that this restriction is not severe in the context of practical application as the current Planguage compilers implement a model slightly more restrictive, yet have been applied successfully to a number of production computational chemistry codes. (A less restrictive model is possible to formulate, but is the topic of another paper.) We develop a restricted shared memory based on the Planguages notion of guards, for which the determinism results hold.

1 Introduction

Well-established libraries such as MPI [11], PVM [12] and SHMEM [20] provide fundamental primitives for parallel code development. The approaches are at a level of abstraction requiring little or no new compiler technology, but the low-level methods can lead the programmer into complex and fragile implementations. To fill the gap between high-level approaches such as HPF and low-level approaches such as MPI, intermediate constructs have been developed which address the important issues of efficiency, concise notation, ease of use, access to low-level details, and portability. In this paper we consider the mechanisms used in two widely distributed parallel programming languages: Co-Array Fortran [18] and Planguages [4]. Both of these languages provide a higher level of abstraction than many popular libraries. Both Co-Array Fortran and Planguages have operational compilers [5, 18] and have been used for developing scientific applications [8, 7, 9, 19]. This paper compares and contrasts the Planguage and Co-Array Fortran models while laying ground work for understanding the program models and possible extensions.

We begin our study in section 3 by introducing the basic communication primitives. Co-Array Fortran provides notation for designating and referencing a type of shared memory, with accesses one-sided and asynchronous by definition. The Planguage model uses the notion of *fusions*, which connect the producer and consumer of data. By definition, once the fusion between producer and consumer is made, synchronization is implicit to both sides of the access. The fusion model builds on the *guarded memory model* [3].

In section 4 we discuss the the Planguage reduction operator which operates on a set of data at multiple processes. For example, the sum of a scalar i at P processes can be written as $sum = +\{i\}$. The statement implicitly involves the set of processes, P. Reductions in Planguages provide a motivation for a simplified notation for reductions in Co-Array Fortran.

The implementation of a compiler and runtime system that supports reduction statements requires some cognizance of the *implicit process group* involved. In contrast, the MPI *communicator* supports groups, however, the MPI *communicator set* is defined explicitly by the programmer. The Planguage model does not provide for such programmer input to define groups, instead, the process group participating in an operation consists of those processes that access the statement, hence the notion of *implicit groups*. The Planguage semantics permit the program to express process groups implicitly using only the control flow of the language, without the addition of communicator constructs.

To provide the runtime support for determining process groups, we have developed the concept of *Pstreams* which we present in section 5. Pstreams represent a collection of processes and a subpath in the control flow graph that they have in common. Pstreams permit the identification of process groups in a joint effort by the compiler and runtime system.

Co-Array Fortran provides a natural type of shared memory programming system that provides significant benefits for developing certain scientific applications [6, 17]. Shared memory systems typically provide asynchronous access of the shared data, thereby permitting independent operations by processes on data and simplified implementations of complex data access patterns. Most shared memory systems provide *distributed shared objects*, *a single name space* for the shared objects, and *asynchronous access* of the shared objects [15, 17, 19]. Asynchronous access permits one-sided accesses of data by making the access independent of a synchronization operation with the supplier. As a consequence most shared memory systems provide a synchronization primitive, as does Co-Array Fortran. In section 6 we examine a shared memory model implemented with fusion objects that does not permit item three, the asynchronous access of data. In section 7.1, we outline a proof that the fusion models we discuss in this paper are deterministic without synchronization primitives. Finally, in section 8 we overview Co-Array Fortran and Planguages, and discuss future research.

2 Terminology

We provide this section as a reference to highlight specialized terminology used in the paper. We refer to a parallel execution as a collection of *processes*. Co-Array Fortran uses the term *images*. We use both terms interchangeably in the paper. We use the term *producer* for suppliers of data, which could correspond to the right hand side of an assignment statement. Accordingly, *consumer* uses the supplied data, possible in an assignment statement. *Program statements* are different from *fusion statements*: a *program statement* refers to a single executable statement of text in the source code; a *fusion statement* refers to a match between a producer and a consumer in a data exchange (section 3). A *fusion statement* could be an assignment in the source text, but in general, it need not be a single statement. A *fusion statement* is comprised of two or more *fusion objects*. *Process groups* are a subset of processes that execute a given data movement statement (section 5). We refer to *explicit process groups* and *implicit process groups* (section 3). The *explicit groups* are defined by the program as in MPI, whereas *implicit groups* are deduced from the program, perhaps at runtime (section 5).

3 Simple assignments

A key problem in parallel programs is to read and write off-process data. In a serial program, we can write

$$x = y$$

with clear intention to move the value at the memory location represented by y to the memory location represented by x. Co-arrays and fusion objects are two constructs designed to access off-process variables such as x and y in parallel applications.

Co-Array Fortran [18] and the Planguages [4] are based on co-arrays and fusion objects, respectively. Both use the single-program multiple-data (SPMD) program model. In the SPMD model, all processes execute the same text, with parallelism exploited by explicit partitions of data and control flow. Operations involving arrays can be performed in parallel by decomposing and distributing arrays across the processors. All *ordinary* variables are replicated.

Co-Array Fortran and Pfortran both provide built-in functions to access the local process identifier and the number of processes. Co-Array Fortran provides functions, which the programmer can then assign to variables; Pfortran provides the variables *myProc* and *nProc*. We also use the variable *me* in both the Co-Array Fortran and Pfortran code fragments in this paper for the local process identifier. Using Planguages, the programmer distributes data and computational workload with program logic involving the *myProc* and *nProc* identifiers. With Co-Array Fortran, shared variables are distributed with a shared storage notation involving specialized array syntax [18].

3.1 Co-Arrays

Co-Array Fortran extends Fortran 90 with co-arrays for a type of shared-memory model. Co-arrays are variables distributed across processes, specified syntactically with an added dimension designated within square brackets and following Fortran 90 array-subscript syntax. For example, to assign the value of y at image q to the local variable t

$$t = y[q]$$

with the semantics of a *get*. Similarly, one can assign t to the value of x at image p

$$x[p] = t$$

with the semantics of a *put*. All other than co-array variables are considered replicated by way of the Co-Array Fortran SPMD model.

Access to co-arrays is one-sided, that is, a read or a write to a co-array can be performed by a single process through some statement which is not necessarily executed by the process that owns the variable. Co-ordinated computation requires explicit synchronization to ensure determinism. For example, in the code

```
if (me==q) baton = y
call sync_all
if (me==p) x = baton[q]
```

image p reads the value of *baton* defined by image q.[1] These syntax and semantics provide an abstraction to shared memory *puts* and *gets*, such as in the SHMEM library on the Cray T3E [20].

3.2 Fusions

The Planguages (Pfortran and PC) extend FORTRAN77 and C with *fusion objects*. Fusion objects are variables distributed across processes, specified syntactically with one of two operators @ and {} used to

[1] The absence of a co-index (within square brackets) for a co-array indicates the local (owning) process (or image).

manipulate off-process data. The following fusion object statement

$$x@p = y@q$$

specifies the assignment of the value of y at process q to the variable x at process p. In message passing terms, the statement is a *fusion of send and receive* and corresponds to a *fusion of a put and get* in Co-Array Fortran. The semantics require that the value assigned to $x@p$ be the value of $y@q$ at the point in the program text where q executes $y@q$ and p executes $x@p$. The matching pair of fusion objects correspond to the producer and the consumer.

A key property of fusion objects is the implicit synchronization for any fusion object access. In the statement above, p will not execute the assignment and progress beyond it until q has reached the statement, thereby making $y@q$ available to p. This can be thought of as a *one-sided synchronization* where p must wait for q to reach the assignment, but q need not wait for p. We call this *give&go semantics.*

Generally, a *fusion statement* formulates a *complete data transfer* through coordinated *fusion object* executions in some set of communicating processes. In this paper we address one case of *fusion statements*, the *same-stream* case. The *same-stream* case corresponds to interaction of all processes involved in the communication within one fusion-object-containing program statement, such as p and q interacting through the source statement $x@p = y@q$. We call all other cases – those which we do not address in this paper – *cross-stream* fusion statements. It is relatively straight forward to show that two processes which execute the same fusion statement can synchronize on that statement and execute deterministically (see section 7.1).

To determine generally the processes that interact through a fusion statement requires analysis of the program logic with runtime support. Data movement with unspecified consumers such as $a = b@p$ and the predicate $(j == i@p)$ and collective operations such as $sum = +\{n\}$ will always involve some *implicit process group.* The *implicit process group* is defined by control flow, and consists of the subset of processes reaching the statement in question. In contrast, an MPI communicator uses an *explicit process group.*[2] Our analysis of process execution involves the concept of *Pstreams* which we discuss in section 5. Briefly, a *Pstream* consists of some path through a portion of the control flow graph and the processes executing together on the path.

In general, fusion objects can appear anywhere a variable can appear such as in predicates, as loop bounds, and so on. A fusion object *instance* refers to one appearance in the program text, for example, in a specific assignment statement. A fusion object instance will be accessed zero or more times. In the case of a fusion object instance within a loop, the fusion object can be accessed numerous times; it is necessary to associate fusion object values corresponding to proper access instances. The i-th read of a fusion object instance corresponds to the fusion object value at the producer process during its i-th access of the same fusion object instance. In other words, there is a one-to-one correspondence between the series of fusion object values (*versions*) at the *producer* and those obtained by the *consumer*. For a fusion object accessed once in a program there is only one version; for a loop, there can be multiple versions. In the following

```
do i = 1, 10
   if (me.EQ.q) then
      a = f(i)
   endif
   b(i)@p = a@q
enddo
```

the i-th version of fusion object $a@q$ is assigned to fusion object $b(i)@p$, corresponding to the i-th loop iteration.

It is up to the implementation to determine how versioning is enforced. Since there could be uneven progress by processes, the implementation might buffer the versioned data, or stall the producer. Buffered message-passing, for example, retains versions in messages indexed by tags. The implementation could also synchronize processes at each fusion object statement.

[2] *Implicit process groups* are formed in MPI through predicates. If the communicator defined in MPI does not match the implicit process group defined by the control flow logic, deadlock could result.

3.3 Summary

Simple assignments with fusion objects and co-arrays, while similar syntactically, vary considerably semantically. Fusion objects build in synchronization which we call *give&go semantics*, whereas co-arrays are synchronization free. The syntax and semantics of fusion objects requires that *implicit process groups* be detected in the source code, where in the case of co-arrays, this task is the responsibility of the programmer, along with placing synchronization statements. To address the need to identify process groups we have developed the concept of *Pstreams*, which are discussed in section 5.

4 Collective communication

In this section we discuss some collective communication operations in Co-Array Fortran and the Planguages. The Planguages provide a special operator for reduction operations which we discuss in the context of both languages.

4.1 Broadcast

Co-arrays and fusion objects provide concise notation for all-to-all broadcasting. To broadcast the variable *temperature* from process p using fusion objects one can write

$$\text{temperature} = \text{temperature@}p$$

where all processes involved in the broadcast must execute the fusion statement. The same exchange can be written with co-arrays with

```
call sync_all
temperature = temperature[p]
call sync_all
```

The synchronization before and after the assignment in this fusion objects is needed to guarantee determinism due to the asynchronous access inherent to co-arrays.

Message-passing systems such as MPI and TCGMSG [14] provide various semantics for synchronization and buffering. TCGMSG, for example, implements synchronous sends and receives, with the sender blocking until the receive completes. Those models can be overly restrictive for certain data-movement requirements. The synchronization and buffering in fusion objects is specified at a higher level than the traditional message-passing models thereby providing flexibility to optimize in the implementation. Moreover, it is likely that the *one-sided synchronization* model (section 3.2) in fusion object communication can be implemented more efficiently than two complete barriers.

4.2 Reductions in Planguages

The *reduction operation* in Planguages is formed with a pair of curly braces and a leading function (strictly speaking, a binary operator). These have the syntax of $f\{v\}$, where in general, the function f forms an Abelian group operation with respect to the variable v. This operator lets one represent generalized reduction operations where the function f is applied to the variable within the curly braces across some collection of processes. For example, summation of an array distributed across $nProc$ processes, with one element per process, can be written

$$sum = +\{localvar\}$$

where *localvar* is a scalar variable at each process, with $+\{localvar\}$ the fusion object. To find the maximum of the scalar variable *time*, one could write

$$\text{maxtime} = \text{MAX}\{\text{time}\}$$

whereas using co-arrays one might write

```
call sync_all
if (me==0) then
  times(:) = time[:]          ! gather times
  maxtime[:] = maxval(times)   ! find and broadcast maximum
endif
call sync_all
```

Better algorithms could be written (e.g., in a library) but it might also be useful to introduce special reduction notation in co-arrays.

4.3 Shifts

A shift is written using co-arrays and fusion objects by addressing with the local process identifier. With fusion objects one can express a shift along a one-dimensional array of processes as

$$\mathrm{newvalue} = \mathrm{value}@((\mathrm{me}+1)\%\mathrm{nProc})$$

This can be written equally concisely with co-arrays

$$\mathrm{newvalue} = \mathrm{value}[(\mathrm{me}+1)\%\mathrm{nProc}]$$

apart from the synchronization which may be necessary before reading co-array memory.

4.4 Summary

The similar syntax of co-arrays and fusion objects results in similar expressions for broadcasts and shifts, but the Planguages provide a reduction operation that does not have a co-array counterpart. Since fusion objects themselves are synchronous, the implementation of fusions must be cognizant of process groups to coordinate communication operations. Specifically, process groups are required for operations where the recipients are unspecified in the program text, such as the broadcast and reduction operation. To implement the reduction operator in co-arrays requires programmer-supplied synchronization and formulation of efficient algorithms for the underlying systems. In the Planguages, the algorithm generation (with synchronization) is provided by the compiler and runtime system.

5 Program streams

We have seen the need in our consideration of collective communications to formalize a definition of process streams. So we digress from the comparison of fusions and co-arrays temporarily.

In the SPMD model it is possible that different processes executing concurrently may, as a consequence of program logic, follow different paths through the code. For example, an *if statement* may evaluate to true at one group of processes and false at others. To make sense of data transfer between processes in these situations we need to be able to identify and track changes in these implicit process groups. This identification can be relevant to any SPMD programming model, but it is particularly needed in the context of fusion objects to implement the fundamental semantics in statements where the participating processes are unspecified, but rather are indicated *implicitly*.

Fusion object statements can implicitly refer to multiple processes, for example, in the statement $+\{x\}$, which is the sum of x across *some subset of processes*, and in the statement $x = y@p$, which sets x at *some subset of processes* to the value of y. We proceed to define what we mean by *some subset of processes* in the context of an SPMD execution. Consider the following code fragment:

```
if( me == odd ) then
```

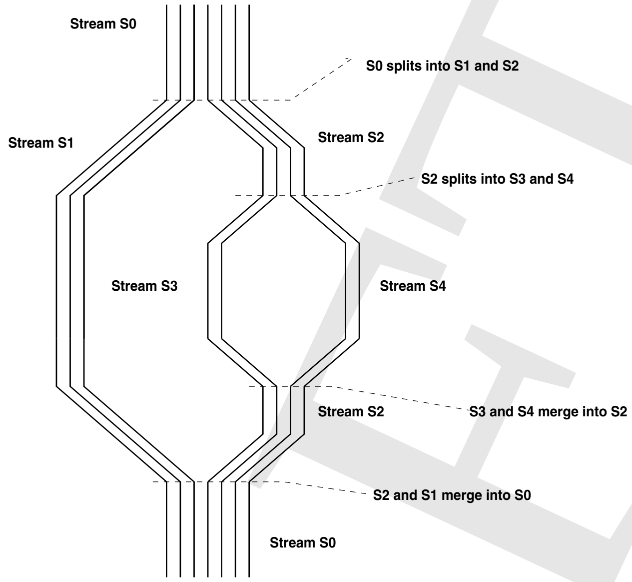

Figure 1: The formation of Pstreams with seven processes begins with the processes in pstream s0. At the first branch the processes split into stream s1 and stream s2; s2 further splits into s3 and s4. Subsequently, processes are aware of the process membership of merging streams since merging streams are from the same parent stream.

```
2       y=y@1
3     else
4       if ( me < nProc/2 ) then
5           y=0
6       else
7           y=1
8       endif
9     endif
```

Statement 1 is an *if statement* on a *non-uniform* predicate, that is, one which could evaluate differently at each process because it depends on the process number.[3] The result of this set of statements is that statement 2 will be executed by odd-numbered processes. Even-numbered processes split further using either statement 5 or statement 7 to define y. This code sets y=0 at even numbered processes. Odd numbered processes execute the fusion object y@1 and set their local y value to the value of y at process 1. The process stream representation in figure 1 illustrates the process of Pstream splitting and merging for the code fragment above.

We may refine the concept of *some subset of processes* above to be *all processes that evaluate the statement that contains the fusion object at the same stage of execution*. The restriction about stage of execution is needed because the fusion object may be inside a loop and be executed multiple times. Also, since the individual processes are asynchronous one can imagine situations in which the same statement in the text could be executed at roughly the same clock time by processes that are in different stages of execution, in which case the same textual fusion object would have to refer to two or more different data transfers corresponding to two or more execution instances of the same fusion object.

To support communication involving implicit process groups we have introduced an entity we call the *process stream* or *Pstream*. *Pstreams*, as mentioned in the introduction, link the implicit process group with the path in the *control flow graph* [1]. They are implemented by a combination of static analysis and a runtime system.

[3] A *uniform* variable is one with the same value with respect to some group of processes. Non-uniformity can only occur through the application of the process identifier either in the control flow or through an assignment.

A Pstream is an implicit process group together with supporting logic.[4]

5.1 Definition and identification of Pstreams

We take the initial set of processes at the start of a Planguage program to be the initial Pstream, since all processes begin by executing the same code. A Pstream may split into two or more Pstreams, forming implicit process groups, when its member processes encounter a conditional control statement on a predicate that is not uniform across all processes in the entering stream. The resulting implicit process groups are disjoint, and the union of their members gives the membership of the parent stream. Split points where new Pstreams may form correspond to program branches in the control flow graph of the program [1], which is the same for all processes in an SPMD program.

The key issue in the analysis of Pstreams is the correct identification of merge points and the membership of the implicit process group after a merge. Deadlock or incorrect execution can be the result of failure to do this. We can identify a merge in the control flow graph of a program (and in the corresponding code) by finding a point in the program which *must* be reached by all paths that exit a particular split point. At such a merge point the processes that split come together again and form the parent process group that existed before the split.

We state as a theorem that a unique closest merge point can be identified to match every branch point at which a split occurs, with the condition that the program must have a reducible flow graph [1]. Studies of random samples of Fortran programs show that most such programs do have reducible flow graphs [2, 16]. In a separate paper we prove this theorem and describe how to deal with programs that do not have reducible graphs.

Pstreams allow us to define "processes in the same stage of execution" to mean processes that are members of the same Pstream executing a particular fusion object at a specific access count inside the Pstream. We describe some of the run-time mechanisms required to support this in [13].

5.2 Runtime support for Pstreams

Identification of Pstream split and merge points in the program can be done statically, but actual identification of implicit process group membership requires information usually only available at runtime.

In an SPMD program we start with an initial Pstream in which the membership of the corresponding implicit process group is all processes. When execution reaches a branch point at which a split occurs, communication must take place between all members of the entering Pstream so that each process knows the other members of its new implicit process group. These points are identified at compile time so the appropriate code can be generated.

A merge point always matches a previous split point, and pairs of split and merge points are identified by static analysis. The runtime system is able to define the implicit process group after a merge by maintaining a data structure which stores the implicit process group at the matching split. Therefore under the assumption of progress at all processes, no communication is required at a merge.

5.3 Summary

Pstreams provide a way to implement fusion object *give&go* semantics by structuring the program so that the process group participating in a fusion statement can be identified. A Pstream itself consists of some subpath in the control flow graph and the processes executing that subpath together.

[4] We would call it simply a stream, but this term is also used for many other things and we wish to avoid confusion.

6 Shared Memory and Planguages

We now return to comparisons of co-arrays and fusion objects with respect to shared memory. Shared memory can simplify programming certain applications by obviating the need to distribute explicitly data and by permitting asynchronous access to data [17, 6]. A conventional shared memory implementation of an object typically provides: 1) distributed shared objects, 2) single name space for the shared objects, and 3) asynchronous access of the shared objects. Co-Array Fortran has these properties of conventional shared memory. Shared memory accesses are problematic to analyze due to the third property. That property allows processes that are not executing a section of code to influence its behavior or be influenced by it. It also permits the existence of multiple writers to the same memory.

In general, shared memory requires additional synchronization or ordering instructions to ensure any specific sequence of writes and reads. Without such synchronization results may be non-deterministic. We proceed in this section to consider a *shared memory fusion statement* that uses *give&go* semantics (see section 3.2). A guard is used to implement *give&go* semantics for the shared memory object resulting in a semantics where determinism can be guaranteed *without* explicit synchronization.

6.1 Guarded shared memory

We consider a restricted shared memory that uses a *guard* for all shared memory accesses, with the implicit synchronization described in section 3.2. To distinguish shared variables in the Planguages we propose a compiler directive indicating which variables are to be shared. To emphasize the fact that the variables are "guarded" and to distinguish from other possible mechanisms for shared variables, we introduce a `guard` statement of the form

```
guard X(1:N) [block, cyclic, etc.]
```

where the terms `block, cyclic, etc.` are optional directives to indicate the type of distribution to be used. Thus `block` would mean that process 0 guards `X(1:N/P)`, process 1 guards `X(1+(N/P):2*N/P)`, and so forth. Guardianship can be implemented as ownership, but this is an implementation issue. The main point is that the guard controls access to the variable.

6.2 Semantics of the `guard` directive

Consider the `guard` statement

```
guard X(1:N) block
```

which means that process `i` guards

```
 X( 1+i*(N/P) : (i+1)*(N/P) ).
```

This means that all accesses to `X(j)` are controlled by the guarding process, `p`. In this case, `p` equals the integer part of `j/N`. More precisely, references to the variable `X(j)` are logically equivalent to references to

```
X(j)@guardX(j)
```

where

```
guardX(j) := j/N
```

(integer part). Here we could imagine that the compiler will generate an integer-valued function `guardVar` for every variable `Var` appearing in a `guard` statement, based on the type of guard distribution specified in the `guard` directive.

For example, an assignment

```
X(j) = ...
```

means

```
X(j)@guardX(j) = ...
```

and will be executed by process `p=guardX(j)`. On the other hand, a memory reference of the form

```
...= ... X(j) ...
```

likewise means

```
...= ... X(j)@guardX(j) ...
```

and the access to `X(j)@guardX(j)` will again be controlled by process `p=guardX(j)`.

This gives us a semantics in which we do not have one sided access, since the process that "owns" the memory must execute the guard statement. Guarded shared memory references become fusion objects and have the same properties and limitations. This is a very significant difference from the semantics of co-arrays, in which there is no relation between the accessing process and the process to which the data is distributed.

It may be necessary, and it is syntactically possible, to make references to memory that has been distributed to processes that are not in the same Pstream (section 5). Such a circumstance is not always detectable by the compiler, since the membership of a Pstream may not be known until runtime.

A Pstream-aware runtime system can detect the case of a guard that must be run by a process not in the stream (since it keeps track of local stream membership). To perform the fusion memory access the system must be aware of which guard to access. In general, since the guard may be executing in any of multiple concurrent Pstreams, the selection of the correct point in the code at which data transfer occurs must be done at runtime. In a future paper we will consider the conditions necessary to placing guards in different sections of code to allow deterministic execution of a fusion object that spans more than one Pstream, and the required runtime support.

6.3 Summary

Co-array shared memory is easier to implement than guarded shared-memory using fusion objects stemming from the use of user-supplied synchronization. A restricted shared memory using fusions was described in this section using a *guard* for shared memory accesses. The guard implements *give&go* semantics described in section 3, and consequently, this model of shared memory does not require a synchronization primitive. The use of a guard in the restricted *same-stream fusion* (section 3) formulation of the shared-memory model excludes cross-stream applications such as described in [17]. However, the model provides the single name space and distribution of the shared object. Fusions offer interesting challenges for correct implementation of the shared-memory object.

7 Determinism

The semantics of the Planguage programming model we consider in this paper establishes deterministic execution resulting from structuring communication between process groups within a Pstream, specifically, communication is through the same program statement containing the producer and consumer fusion objects. The property is analogous to structuring control flow in a sequential program [10].

7.1 Determinism with Planguages

Determinism is the property whereby a program produces the same output for a given input independent of the underlying operational details. Determinism is a useful program property, with most applications requiring it. A Planguage program conforming to *same-stream* fusions is deterministic. In this section we outline a proof of this case.

Let a parallel execution M consist of P processes, $M = \{p \mid 0 \leq p < P\}$. Let each process p execute the same program. We define a process execution as a series of states, $\mathcal{S}_p = [s_0^p,\ s_1^p,\ \ldots,\ s_{n_p}^{p\,*}]$, with initial state s_0^p and final state $s_{n_p}^{p\,*}$. We say a parallel execution is *strongly deterministic* if $s_0^p \Rightarrow \mathcal{S}_p$ for all $p \in M$. That is, for any initial state s_0^p, the series of states $\mathcal{S}_p$ is uniquely determined for each process p. Furthermore, should the program enter an infinite loop state, it will always do so. We annotate the final state with an asterisk to indicate that the "final" state may be an infinite-loop state.

Planguage processes execute two types of statements: those that access only local data, and those that interact with other processes. Call them *local* and *non-local* statements, respectively. Let the state s_k^p correspond to process p's state after the k-th statement execution.

We assume that any local statement is deterministic. It follows that any sub-sequence of local statements is deterministic. Let statement 0 be a local statement corresponding to state s_0^p. For each p, let the first $k_p > 0$ states be locally derived states, with state $k_p + 1$ non-locally derived. Then the series of states w_p, where $|w_p| = k_p$, is a prefix of $\mathcal{S}_p$. This establishes that the values for the first fusion operation (process identifiers and data) will be exactly reproduced for all executions that begin in state s_0^p. For each p, the state derived from the first non-local statement, $k_p + 1$, results in deterministic state $s_{k_p+1}^p$, since w_p is deterministic and the fusion statement is deterministic by definition.

Thus the key point is that the program proceeds deterministically at each process from one fusion to the next. For each process, either it will get to the next fusion, or it will loop indefinitely. If the latter, since everything is deterministic to that point, it will always loop indefinitely. If none of the processes loop indefinitely, then all will arrive deterministically at the next fusion, and so on by induction. These results also hold for the guarded shared memory described in section 6.

7.2 Determinism with Co-Array Fortran

The flexibility provided by Co-Array Fortran permits programs that are not necessarily deterministic. To achieve determinism, the programmer must properly synchronize the program "by hand." Without synchronization, accesses to a co-array object may occur in different order in two executions of a program M, even given the same starting data. Therefore the final state may be different and M is not deterministic.

8 Discussion

Simple assignments with fusion objects and co-arrays, while similar syntactically, provide different semantics. Fusion objects have an implied synchronization with *give&go semantics*, whereas co-array access is without implicit synchronization. In the case of co-arrays, synchronization is the responsibility of the programmer. We have discussed the *same-stream* fusion model which restricts processes to interact through a single fusion-object-containing program statement. It could be useful to permit the fusion statement to consist of fusion objects that are not the same textual object (we are currently researching the implications of this). That co-arrays achieve synchronization separate and independent from data access results in some flexibility over fusions in specifying data exchange between processes.

To implement the general semantics of the fusion model requires that the runtime system have some notion of process groups. This requirement arises from statements where the consumer is unspecified, thereby implicitly specifing processes. Examples of this scenario include statements of the form $a = b@p$ where the consumer processes are unspecified. We contrast this to MPI where process groups are explicitly defined through communicators. We have developed a framework which we call Pstreams for identifying and managing process groups at runtime. Pstreams are formulated jointly by the compiler and runtime system. The similiarity in the guard of fusion accesses and synchronizations in Co-Array Fortran suggests that analysis for fusion object semantics might also be applicable to co-array synchronization.

We have outlined a proof showing that the *same-stream* fusion-object model discussed in this paper is deterministic. We introduced a restricted shared-memory model for fusions with a guard on the supply side of the data exchange which is also deterministic. We add, while the restricted fusion object implementation has been demonstrated effective in a number of applications [7, 6, 9], there are important applications where the restricted fusion is inadequate. Those applications require a one-sided, asynchronous method of communication (for example [17]).

Reductions are common operations performed in parallel scientific applications (for example [8]). The Planguages provide a concise notation for reduction operations, permitting the compiler to generate efficient code. We suggest that it may be useful to introduce special reduction notation for Co-Array Fortran.

Acknowledgements

The ultimate test of an application language rests with its users, and accordingly, the authors are grateful for the use that Piotr Bala gives the Planguages in his research. We also acknowledge Piotr's invigorating collaboration in pursuing the Co-Array Fortran and Planguage connection. We would like to thank Roman Grigoriev for extensive comments on the Planguages and their compilers. The work was supported in part by the National Partnership for Advanced Computational Infrastructure, a cooperative agreement between the National Science Foundation and UCSD (ACI-9619020), and by the ASCI Flash Center at the University of Chicago under DOE contract B341405.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] F. E. Allen. Control flow analysis. In *ACM SIGPLAN Notices*, pages 1–19, 1970.

[3] Babak Bagheri. Parallel programming with guarded objects. *PhD dissertation*, Pennsylvania State University, Department of Computer Science, 1994. Available at `http://planguages.cs.uchicago.edu/papers/babak.ps`.

[4] Babak Bagheri, Terry Clark, and L. Ridgway Scott. IPfortran: a parallel dialect of Fortran. *Fortran Forum*, 11:20–31, Sept. 1992.

[5] Babak Bagheri, Terry Clark, and L. Ridway Scott. Pfortran (a parallel extension of Fortran) reference manual. Technical report, 1991. Revised March, 1999. Available at `http://planguages.cs.uchicago.edu/`.

[6] Babak Bagheri, Andrew Ilin, and L. Ridgway Scott. A comparison of distributed and shared memory scalable architectures: 1. KSR shared memory. In *Scalable High Performance Computing Conference*, pages 9–16, Knoxville, TN, May 1994. IEEE Computer Society.

[7] P. Bala, P. Grochowski, B. Lesyng, and J.A. McCammon. Quantum-classical molecular dynamics and its computer implementation. *Comp. Chem.*, 19:155–160, 1995.

[8] Piotr Bala and Terry Clark. Experiences with Co-Array Fortran and Pfortran as tools for parallelization of a large-scale scientific application. In *Proceedings of the Third International Conference on Parallel Processing & Applied Mathematics*, pages 367–377. Technical Univeristy of Czestochowa, 1999.

[9] Terry Clark, Reinhard von Hanxleden, J. Andrew McCammon, and L. Ridgway Scott. Parallelizing molecular dynamics using spatial decomposition. In *Proceedings of the Scalable High Performance Computing Conference*, pages 95–102, Knoxville, TN, May 1994. IEEE Computer Society. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93356-S`.

[10] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[11] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[12] G. A. Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, 1994.

[13] Ernesto Gomez and L. Ridgway Scott. Overlapping and Shortcutting Techniques in Loosely Synchronous Irregular Problems. In *Solving Irregularly Structured Problems in Parallel*, volume LNCS 1457. Springer, August 1998.

[14] Robert J. Harrison. TCGMSG, 1992. Theoretical Chemistry Group, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL. 60439, `http://www.emsl.pnl.gov:2080/docs/nwchem/nwchem.html`.

[15] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Sared Memory on Standard Workstations and Operating Systems. In *USENIX Winter '94 Technical Conference*, pages 115–131, San Francisco, California, 1994.

[16] D. E. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 2, 1971.

[17] Jarek Nieplocha, Robert J. Harrison, and Rick J. Littlefield. Global arrays: A portable *shared memory* programming model for distributed memory computers. In *Proceedings of Supercomputing'94*, pages 340–349, San Francisco, California, 1994.

[18] R. W. Numrich. F^{--}: a parallel extension to Cray Fortran. *Scientific Programming*, 6(3):275–284, 1997. Co-Array Fortran home page at `http://www.co-array.org/`.

[19] R. W. Numrich, J. Reid, and K. Kim. Writing a multigrid solver using Co-array Fortran. In *Recent Advances in Applied Parallel Computing, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, Springer Lecture Notes in Computer Science, 1541*, pages 390–399. Springer–Verlag Berlin, 1998.

[20] SGI/Cray. CRAY T3E Fortran Optimization Guide, SG-2518 3.0. Technical Report `http://manul.nsc.liu.se:8080/library/`, 1996.